Draft Diktat Struktur Data

Oleh :

Inggriani Liem

Program Studi Teknik Informatika

Institut Teknologi Bandung

Edisi 2008

PRAKATA

Diktat “Struktur Data” ini merupakan revisi dari Diktat “Algoritma dan Pemrograman Buku II” yang diterbitkan dan dipakai di Jurusan Teknik Informatika ITB sejak tahun 1990, dan belum pernah sempat diterbitkan sebagai buku karena perkembangannya yang terus-menerus.

Akibat dari perubahan kurikulum di Jurusan Teknik Informatika ITB, maka mulai tahun ajaran 1998/1999, kuliah Algoritma dan Pemrograman Prosedural pada kurikulum sebelumnya dipecah menjadi dua bagian, yaitu:

- IF225 Kuliah Algoritma dan Pemrograman Prosedural, yang difokuskan kepada konstruksi dasar pemrograman prosedural
- IF222 Kuliah Struktur Data, dengan mencakup struktur data yang semula tercakup di matakuliah Struktur Data dan Algoritma Lanjut.

Seperti halnya Buku II Algoritma dan Pemrograman, diktat ini terdiri dari dua bagian utama, yaitu:

- Bagian konsep dasar dan konstruktor struktur data
- Studi kasus implementasi struktur data dalam persoalan klasik/nyata

Selain dari matakuliah Algoritma dan pemrograman, pemahaman pada diktat ini memerlukan beberapa pemahaman pada kuliah IF221 Dasar Pemrograman.

Bahasa yang dipakai untuk menyampaikan modul struktur data dan kasus adalah:

- Bahasa Algoritmik,
- Bahasa Ada,
- Bahasa C.

Ketiga bahasa tersebut dipakai secara bergantian.

Selain fokus kepada Struktur Data, perbedaan utama diktat ini dengan diktat sebelumnya adalah dari struktur organisasi modul dengan memperhatikan bahasa pemrograman yang dipakai sebagai bahasa pengeksekusi program di Jurusan Teknik Informatika, yaitu bahasa Ada dan bahasa C.

Bahasa Algoritma masih dibutuhkan sebagai bahasa spesifikasi, karena memungkinkan perancang program untuk mengkonsentrasikan diri pada konsep tanpa dipusingkan oleh sintaks dan konsep-konsep yang merupakan “pernik-pernik” dari bahasa yang bersangkutan. Hal ini dapat dilihat pada contoh-contoh yang disajikan dalam bahasa algoritmik, dan komentar implementasinya dalam bahasa nyata.

Bahasa Ada dipilih karena merupakan bahasa yang sangat terstruktur dan ketat *type*, di samping konsep modularitasnya yang nantinya mendukung ke arah pemrograman Objek. Bahasa Ada juga mempunyai fasilitas *tasking*, yang akan mendukung pada perkuliahan Pemrograman Objek Konkuren pada kurikulum yang sama.

Bahasa C dipilih karena merupakan bahasa yang banyak dipakai dalam industri perrangkat lunak saat ini, dan nantinya akan dilanjutkan dengan C++ pada perkuliahan Pemrograman Objek Konkuren pada kurikulum yang sama.

Catatan ringkas mengenai Pemrograman dalam Bahasa Ada dan Bahasa C yang diacu pada diktat ini merupakan diktat kuliah yang dikembangkan penulis di Jurusan Teknik Informatika ITB.

DAFTAR ISI

DAFTAR PUSTAKA

1. Aho, Hopcroft, Ullman : "Data Structures and Algorithms", Prentice Hall, 1987.
2. Horowitz, E. & Sahni, S. : "Fundamentals of Data Structures in Pascal", Pitman Publishing Limited, 1984.
3. Knuth, D.E. : "The Art of Computer Programming", Vol. 1 : "Fundamentals Algoritms", Addison Wisley, 1968.
4. Sedgewick R. : "Algorithms", Addison Wisley, 1984.
5. [Wirth86] Wirth, N. : "Algorithms & Data Stuctures", Prentice Hall,1986.

BAGIAN I. STRUKTUR DATA

Abstract Data Type (ADT)

Abstract Data Type (ADT) adalah definisi ***TYPE*** dan sekumpulan **PRIMITIF** (operasi dasar) terhadap *TYPE* tersebut. Selain itu, dalam sebuah ADT yang lengkap disertakan pula definisi invarian dari *TYPE* dan aksioma yang berlaku. ADT merupakan definisi **statik**.
Definisi *type* dari sebuah ADT dapat mengandung sebuah definisi ADT lain. Misalnya:

- ADT Waktu yang terdiri dari ADT JAM dan ADT DATE.
- GARIS yang terdiri dari dua buah POINT.
- SEGI4 yang terdiri dari pasangan dua buah POINT (Top, Left) dan (Bottom,Right).

Type diterjemahkan menjadi *type* terdefinisi dalam bahasa yang bersangkutan, misalnya menjadi `record` dalam bahasa Ada/Pascal atau `struct` dalam bahasa C.

Primitif, dalam konteks prosedural, diterjemahkan menjadi fungsi atau prosedur.
Primitif dikelompokkan menjadi:

- Konstruktor/Kreator, pembentuk **nilai *type***. Semua objek (variabel) ber-*type* tersebut harus melalui konstruktor. Biasanya namanya diawali Make.
- Selektor, untuk mengakses komponen *type* (biasanya namanya diawali dengan Get).
- Prosedur pengubah nilai komponen (biasanya namanya diawali Get).
- Validator komponen *type*, yang dipakai untuk mentest apakah dapat membentuk *type* sesuai dengan batasan.
- Destruktor/Dealokator, yaitu untuk "menghancurkan" nilai objek (sekaligus memori penyimpannya).
- Baca/Tulis, untuk interface dengan input/output *device*.
- Operator relational, terhadap *type* tersebut untuk mendefinisikan lebih besar, lebih kecil, sama dengan, dan sebagainya.
- Aritmatika terhadap *type* tersebut karena biasanya aritmatika dalam bahasa pemrograman hanya terdefinisi untuk bilangan numerik.
- Konversi dari *type* tersebut ke *type* dasar dan sebaliknya.

ADT biasanya diimplementasi menjadi dua buah modul, yaitu:

- Definisi/spesifikasi *type* dan primitif.
  - Spesifikasi *type* sesuai dengan bahasa yang bersangkutan.
  - Spesifikasi dari primitif sesuai dengan kaidah dalam konteks prosedural, yaitu:
    - Fungsi : nama, domain, *range*, dan prekondisi jika ada
    - Prosedur : *Initial State*, *Final State* dan proses yang dilakukan
- *Body*/realisasi dari primitif, berupa kode program dalam bahasa yang bersangkutan. Realisasi fungsi dan prosedur harus sedapat mungkin memanfaatkan selektor dan konstruktor.

Supaya ADT dapat dites secara tuntas, dalam kuliah ini setiap kali membuat sebuah ADT, harus disertai dengan sebuah program utama yang dibuat khusus untuk mengetes ADT tersebut, yang minimal mengandung pemakaian (*call*) terhadap setiap fungsi dan prosedur dengan mencakup semua kasus parameter. Program utama yang khusus dibuat untuk tes ini disebut sebagai ***driver***. Urutan pemanggilan harus diatur sehingga fungsi/prosedur yang memakai fungsi/prosedur lain harus sudah dites dan direalisasi terlebih dulu.

Realisasi ADT dalam beberapa bahasa:

BAHASA	SPESIFIKASI	BODY
Pascal (hanya dalam Turbo Pascal)	Satu unit *interface*	*Implementation*
C	File *header* dengan ekstensi `.h`	File kode dengan ekstensi `.c`
Ada	Paket dengan ekstensi `.ads`	Paket body dengan ekstensi `.adb`

Dalam modul ADT tidak terkandung definisi variabel. Modul ADT biasanya dimanfaatkan oleh modul lain, yang akan mendeklarasikan variabel ber-*type* ADT tersebut dalam modulnya. Jadi, ADT bertindak sebagai ***Supplier*** (penyedia *type* dan primitif), sedangkan modul pengguna berperan sebagai ***Client*** (pengguna) dari ADT tersebut. Biasanya ada sebuah pengguna yang khusus yang disebut sebagai ***main program*** (program utama) yang memanfaatkan langsung *type* tersebut

Contoh dari ADT diberikan dalam bahasa Algoritmik di diktat ini, adalah:

- ADT JAM. Contoh tersebut sekaligus memberikan gambaran mengapa bahasa Algoritmik masih dibutuhkan karena bersifat "umum" dan tidak tergantung pada pernik-pernik dalam bahasa pemrograman, sehingga dapat dipakai sebagai bahasa dalam membuat spesifikasi umum. Perhatikanlah catatan implementasi dalam bahasa C dan Bahasa Ada yang disertakan pada akhir definisi ADT.
- ADT POINT. Contoh ini diberikan karena merupakan ADT yang sangat penting yang dipakai dalam bidang Informatika. Contoh ini bahkan dapat dikatakan "standar" dalam pemrograman, dan akan selalu dipakai sampai dengan pemrograman berorientasi objek.
- ADT GARIS. Contoh ini memberikan gambaran sebuah ADT yang memanfaatkan ADT yang pernah dibuat (yaitu ADT POINT), dan sebuah ADT yang mempunyai *constraint/invariant* (persyaratan tertentu).
- ADT SEGIEMPAT dengan posisi sejajar sumbu X dan Y, didefinisikan sebagai dua buah POINT (TopLeft dan BottomRight).

Dari contoh-contoh tersebut, diharapkan mahasiswa dapat membangun ADT lain, minimal ADT yang diberikan definisinya dalam latihan.
Pada kehidupan nyata, pembuatan ADT selalu dimulai dari definisi "objek" yang berkomponen tersebut. Maka sangat penting, pada fase pertama pendefinisian ADT, diberikan definisi yang jelas. Lihat contoh pada latihan.
Semua contoh ADT pada bagian ini diberikan dalam bahasa Algoritmik, untuk diimplementasikan dalam bahasa C dan Ada, terutama karena contoh implementasi (sebagian) sudah diberikan dalam diktat bahasa pemrograman yang bersangkutan.

Standar kuliah untuk implementasi ADT:

1. Setiap ADT harus dibuat menjadi spesifikasi, *body*, dan *driver*.
2. Prosedur untuk melakukan implementasi kode program ADT (cara ini juga berlaku untuk bahasa Ada) :
   - Ketik *header file* secara lengkap, dan kompilasilah supaya bebas kesalahan
   - Copy *header file* menjadi body file
   - Lakukan editing secara "cepat" sehingga spesifikasi menjadi *body*:
     - Buang tanda ";" di akhir spesifikasi.
     - Tambahkan *body* { } di setiap fungsi/prosedur.

- Khusus untuk fungsi, beri *return value* sesuai *type* fungsi.

- Kodelah instruksi yang menjadi ***body*** fungsi/prosedur per kelompok (disarankan untuk melakukan "*incremental implementation*") sesuai dengan urutan pemanggilan.
- Segera setelah sekelompok fungsi/prosedur selesai, tambahkan di *driver* dan lakukan tes ulang.

Setiap saat, banyaknya fungsi/prosedur yang sudah dikode *body*-nya harus seimbang.

Standar kuliah untuk implementasi ADT dalam Bahasa C:

- Dalam bahasa C, modul spesifikasi, realisasi, dan *driver* harus disimpan dalam sebuah direktori sesuai dengan nama ADT. Jadi, *sourcecode* lengkap sebuah ADT dalam bahasa C adalah sebuah direktori dan isinya adalah minimal *header file*, *body file*,dan *driver*. *Driver* harus disimpan dan dipakai untuk mengetes ulang jika ADT dibawa dari satu *platform* ke *platform* lain
- Perhatikanlah **syarat realisasi** *file header* dalam bahasa C dan paket dalam bahasa Ada sesuai dengan yang ada pada Diktat "Contoh Program Kecil dalam Bahasa C" dan "Contoh Program Kecil dalam Bahasa Ada".
- Bahasa C yang digunakan untuk mengkode harus bahasa C yang standar, misalnya ANSI-C.

ADT JAM dalam Bahasa Algoritmik

```
{ *** Definisi TYPE JAM <HH:MM:SS> *** }
type Hour   : integer [0..23]
type Minute : integer [0..59]
type Second : integer [0..59]

type JAM : < HH : Hour,    { Hour [0..23]   }
             MM : Minute,  { Minute [0..59] }
             SS : Second   { Second [0..59] } >

{ ************************************************************* }
{ DEFINISI PRIMITIF                                             }
{ ************************************************************* }
{ KELOMPOK VALIDASI TERHADAP TYPE                               }
{ ************************************************************* }
function IsJValid (H, M, S : integer) → boolean
{ Mengirim true  jika H,M,S dapat membentuk J yang valid }
{ dipakai untuk mentest SEBELUM membentuk sebuah Jam }

{ *** Konstruktor: Membentuk sebuah JAM dari komponen-komponennya *** }
function MakeJam (HH : integer, MM : integer, SS : integer) → JAM
{ Membentuk sebuah JAM dari komponen-komponennya yang valid }
{ Prekondisi : HH, MM, SS valid untuk membentuk JAM }

{ *** Operasi terhadap komponen : selektor Get dan Set *** }
{ *** Selektor *** }
function GetHour (J : JAM) → integer
{ Mengirimkan bagian HH (Hour) dari JAM }
function GetMinute (J : JAM) → integer
{ Mengirimkan bagian Menit (MM) dari JAM }
function GetSecond (J : JAM) → integer
{ Mengirimkan bagian Second (SS) dari JAM }
{ *** Pengubah nilai Komponen *** }
procedure  SetHH (input/output J : JAM, input newHH : integer)
{ Mengubah nilai komponen HH dari J }
procedure  SetMM (input/output J : JAM, input newMM : integer)
{ Mengubah nilai komponen MM dari J }
procedure  SetSS (input/output J : JAM, input newSS : integer)
{ Mengubah nilai komponen SS dari J }

{ *** Destruktor *** }
{ Tidak perlu. Akan dijelaskan kemudian. }

{ ************************************************************* }
{ KELOMPOK BACA/TULIS                                           }
{ ************************************************************* }
procedure BacaJam (output J : JAM)
{ I.S. : J tidak terdefinisi }
{ F.S. : J terdefinisi dan merupakan jam yang valid }
{ Proses : mengulangi membaca komponen H,M,S sehingga membentuk J }
{ yang valid. Tidak mungkin menghasilkan J yang tidak valid. }
procedure TulisJam (input J : JAM)
{ I.S. : J sembarang }
{ F.S. :   Nilai J ditulis dg format HH:MM:SS }
{ Proses : menulis nilai setiap komponen J ke layar }
```

```
{ ************************************************************** }
{ KELOMPOK KONVERSI TERHADAP TYPE                                }
{ ************************************************************** }
function JamToDetik (J : JAM) → integer
{ Diberikan sebuah JAM, mengkonversi menjadi Detik }
{ Rumus : detik = 3600*hour+menit*60 + detik  }
{ nilai maksimum = 3600*23+59*60+59*60 }
{ hati-hati dengan representasi integer pada bahasa implementasi }
{ kebanyakan sistem mengimplementasi integer, }
{ bernilai maksimum kurang dari nilai maksimum hasil konversi }
function DetikToJam (N : integer) → JAM
{ Mengirim  konversi detik ke JAM }
{ pada beberapa bahasa, representasi integer tidak cukup untuk }
{ menampung N }

{ ************************************************************** }
{ KELOMPOK OPERASI TERHADAP TYPE                                 }
{ ************************************************************** }
{ *** Kelompok Operator Relational *** }
function JEQ (J1 : JAM, J2 : JAM) → boolean
{ Mengirimkan true jika J1=J2, false jika tidak }
function JNEQ (J1 : JAM, J2 : JAM) → boolean
{ Mengirimkan true jika J1 tidak sama dengan J2 }
function JLT (J1 : JAM, J2 : JAM) → boolean
{ Mengirimkan true jika J1<J2 , false jika tidak }
function JGT (J1 : JAM, J2 : JAM) → boolean
{ Mengirimkan true jika J1>J2, false jika tidak }
{ *** Operator aritmatika JAM *** }
function JPlus (J1 : JAM, J2 : JAM) → JAM
{ Menghasilkan J1+J2, dalam bentuk JAM }
function JMinus (J1 : JAM, J2 : JAM) → JAM
{ Menghasilkan J1-J2, dalam bentuk JAM }
{ Prekondisi : J1<=J2 }
function NextDetik (J : JAM) → JAM
{ Mengirim 1 detik setelah J dalam bentuk JAM  }
function NextNDetik (J : JAM, N : integer) → JAM
{ Mengirim N detik setelah J dalam bentuk JAM }
function PrevDetik (J : JAM) → JAM
{ Mengirim 1 detik sebelum J dalam bentuk JAM }
function PrevNDetik (J : JAM, N : integer) → JAM
{ Mengirim N detik sebelum J dalam bentuk JAM }
{ *** Kelompok Operator Aritmetika *** }
function Durasi (Jaw : JAM , Jakh : JAM) → integer
{ Mengirim JAkh -JAw dlm Detik, dengan kalkulasi }
{ Hasilnya negatif jika Jaw > JAkhir }
{ ************************************************************** }
{ Dapat ditambahkan fungsi lain }
```

Catatan implementasi:

- Dalam implementasi dengan bahasa C, di mana representasi integer ada bermacam-macam, Anda harus berhati-hati misalnya:
  - Dalam menentukan *range* dari fungsi Durasi, karena nilai detik dalam dua puluh empat jam melebihi representasi `int`, fungsi Durasi harus mempunyai *range* dalam bentuk `long int` dan semua operasi dalam *body* fungsi harus di-*casting*.

- Representasi Hour, Minute, dan Second terpaksa harus dalam bentuk `int` (atau `byte`) dan tidak mungkin dibatasi dengan angka yang persis. Itulah sebabnya fungsi validitas terhadap *type* diperlukan.
- Fungsi selektor Get dan Set dapat digantikan dengan *macro* berparameter. Misalnya untuk Selektor terhadap Hour(P) , Minute(J), dan Second(J) dituliskan sebagai:

```
#define Hour(J) (J).HH
#define Minute(J) (J).MM
#define Second(J) (J).SS
```

- Dalam implementasi dengan bahasa Ada, fungsi IsJValid dapat dihilangkan, karena dalam pembentukan sebuah JAM dapat memanfaatkan *subtype* untuk HH, MM, dan SS serta memanfaatkan *exception* dalam menangani apakah 3 buah nilai H, M, dan S dapat membentuk sebuah JAM yang valid.

ADT POINT dalam Bahasa Algoritmik

```
{ *** Definisi ABSTRACT DATA TYPE POINT *** }
type POINT : < X : integer, { absis   }
               Y : integer  { ordinat } >

{ ************************************************************* }
{ DEFINISI PROTOTIPE PRIMITIF                                   }
{ ************************************************************* }
{ *** Konstruktor membentuk POINT *** }
function MakePOINT (X : integer, Y : integer ) → POINT
{ Membentuk sebuah POINT dari komponen-komponennya }

( *** Operasi terhadap komponen : selekstor Get dan Set *** }
{ *** Selektor POINT *** }
function  GetAbsis (P : POINT) → integer
{ Mengirimkan komponen Absis dari P }
function  GetOrdinat (P : POINT) → integer
{ Mengirimkan komponen Ordinat dari P POINT}
{ *** Set nilai komponen *** }
procedure  SetAbsis (input/output P : POINT, input newX : integer)
{ Mengubah nilai komponen Absis dari P }
procedure  SetOrdinat (input/output P:POINT, input newY : integer)
{ Mengubah nilai komponen Ordinat dari P }

{ *** Destruktor/Dealokator: tidak perlu *** }

{ ************************************************************* }
{ KELOMPOK INTERAKSI DENGAN I/O DEVICE, BACA/TULIS              }
{ ************************************************************* }
procedure BacaPOINT (output P : POINT)
{ Makepoint(x,y,P) membentuk P dari x dan y yang dibaca }
procedure TulisPOINT (input P : POINT)
{ Nilai P ditulis ke layar dg format "(X,Y}" }

{ Perhatikanlah nama fungsi untuk operator aritmatika dan relasional}
{ Dalam beberapa bahasa, dimungkinkan nama seperti }
{ operator aritmatika untuk numerik dan operator relasional }

{ ************************************************************* }
{ KELOMPOK OPERASI TERHADAP TYPE                                }
{ ************************************************************* }
{ *** Kelompok operasi aritmatika terhadap POINT *** }
function "+" (P1, P2: POINT) →  POINT
{ Menghasilkan POINT yang bernilai P1+P2 }
{ Melakukan operasi penjumlahan vektor }
function "-" (P1, P2: POINT) →  POINT
{ Menghasilkan POINT bernilai P1-P2 }
{ Buatlah spesifikasi pengurangan dua buah POINT }
function "." (P1, P2 : POINT) → POINT
{ Operasi perkalian P1.P2, melakukan operasi dot product }
function x (P1, P2 : POINT) → POINT
{ Operasi perkalian P1xP2, melakukan operasi cross product }

{ *** Kelompok operasi relasional terhadap POINT *** }
function EQ (P1, P2 : POINT) → boolean
{ Mengirimkan true jika P1 = P2: absis dan ordinatnya sama }
```

```
function NEQ (P1, P2 : POINT) → boolean
{ Mengirimkan true jika P1 tidak sama dengan P2 }
function "<" (P1, P2 : POINT) → boolean
{ Mengirimkan true jika P1 < P2. Definisi lebih kecil: lebih "kiri-
bawah" dalam bidang kartesian }
function ">" (P1, P2 : POINT) → boolean
{ Mengirimkan true jika P1 > P2. Definisi lebih besar: lebih "kanan-
atas" dalam bidang kartesian }

{ *** Kelompok menentukan di mana P berada *** }
function IsOrigin (P : POINT) → boolean
{ Menghasilkan true jika P adalah titik origin }
function IsOnSbX (P : POINT) → boolean
{ Menghasilkan true jika P terletak Pada sumbu X }
function IsOnSbY (P : POINT) → boolean
{ Menghasilkan true jika P terletak pada sumbu Y }
function Kuadran (P : POINT) → integer
{ Menghasilkan kuadran dari P: 1,2,3, atau 4 }
{ Prekondisi : P bukan Titik Origin, }
{              dan P tidak terletak di salah satu sumbu }

{ *** Kelompok operasi lain terhadap type *** }
function NextX (P : POINT) → POINT
{ Mengirim salinan P dengan absis ditambah satu }
function NextY (P : POINT) → POINT
{ Mengirim salinan P dengan ordinat ditambah satu }
function PlusDelta (deltaX, deltaY : integer) → POINT
{ Mengirim salinan P yang absisnya = Absis(P)+deltaX dan }
{ Ordinatnya adalah Ordinat(P)+ deltaY }
function MirrorOf (P : POINT, SbX, SbY : boolean) → POINT
{ Menghasilkan salinan P yang dicerminkan }
{ tergantung nilai SbX dan SBY }
{ Jika SbX bernilai true, maka dicerminkan thd Sumbu X }
{ Jika SbY bernilai true, maka dicerminkan thd Sumbu Y }
function Jarak0 (P : POINT) → real
{ Menghitung jarak P ke (0,0) }
function Panjang (P1, P2 :POINT) → real
{ Menghitung panjang garis yang dibentuk P1 dan P2 }
{ Perhatikanlah bahwa di sini spesifikasi fungsi "kurang" baik }
{ sebab menyangkut ADT Garis!!! }
procedure Geser (input/output P : POINT, input deltax, deltaY : integer)
{I.S. P terdefinisi }
{F.S. P digeser sebesar DeltaX dan ordinatnya sebesar Delta Y }
procedure GeserKeSbX (input/output P : POINT)
{ I.S. P terdefinisi }
{ F.S. P di Sumbu X dg absis sama dg absis semula }
{ Proses : P tergeser ke Sumbu X }
{ Contoh : Jika koordinat semula(9,9) menjadi (9,0) }
procedure GeserKeSbY (input/output P : POINT)
{ I.S. P terdefinisi }
{ F.S. P di Sumbu Y dengan absis yang sama dengan semula }
{ Proses : P digeser Ke Sumbu Y }
{ Contoh : Jika koordinat semula(9,9) menjadi (0,9) }
procedure Mirror (input/output P : POINT, input SbX, SbY : boolean)
{ I.S. P terdefinisi }
{ F.S.  P dicerminkan tergantung nilai SbX atau SBY }
{ Jika SbX true maka dicerminkan terhadap Sumbu X }
{ Jika SbY true maka dicerminkan terhadap Sumbu Y }
```

```
procedure Putar (input/output P : POINT, input Sudut : Real);
{ I.S. P terdefinisi }
{ F.S. P diputar sebesar Sudut derajat }
```

Catatan implementasi:

- Dalam implementasi dengan Bahasa C, nama fungsi untuk operator aritmatika dan operator relasional (seperti “<”) tidak dimungkinkan. Dalam hal ini, harus disesuaikan.
- Beberapa fungsi di atas belum mempunyai spesifikasi yang jelas. Oleh karena itu, sebelum implementasi harus dibuat spesifikasi yang lebih jelas.

ADT GARIS dalam Bahasa Algoritmik

```
{ Contoh ADT yang memanfaatkan ADT Lain }
{ Definisi : GARIS dibentuk oleh dua buah POINT }

{ *** ADT LAIN YANG DIPAKAI *** }
USE POINT
{ *** Definisi TYPE GARIS *** }
type GARIS : < PAw : POINT,  { Titik Awal  }
               PAkh : POINT  { Titik Akhir }  >

{ ************************************************************* }
{ DEFINISI PRIMITIF                                             }
{ ************************************************************* }

{ *** Konstruktor membentuk GARIS *** }
procedure MakeGARIS (input P1, P2 : POINT, output L : GARIS)
{ I.S. P1 dan P2 terdefinisi }
{ F.S. L terdefinisi dengan L.PAw= P1 dan L.Pakh=P2 }
{ Membentuk sebuah L dari komponen-komponennya }

{ *** Selektor GARIS *** }
function  GetPAw (G : GARIS) → POINT
{ Mengirimkan komponen Titik Awal dari L GARIS }
function  GetPAkh (G : GARIS) → POINT
{ Mengirimkan komponen Titik Akhir dari L GARIS }
{ *** Set nilai komponen *** }
procedure  SetPAw (input/output G : GARIS, input newPAw : POINT)
{ Mengubah nilai komponen PAw dari G }
procedure  SetPAkh (input/output G : GARIS, input newPAkh : POINT)
{ Mengubah nilai komponen PAkh dari G }

{ ************************************************************* }
{ KELOMPOK INTERAKSI DENGAN I/O DEVICE, BACA/TULIS              }
{ ************************************************************* }
procedure BacaGARIS (output L : GARIS)
{ I.S. sembarang }
{ F.S. mengulangi membaca dua buah nilai P1 dan P2 sehingga }
{ dapat membentuk GARIS yang valid }
{ MakeGARIS(P1,P2) dari P1 dan P2 yang dibaca }
procedure TulisGARIS (input L : GARIS)
{ Nilai L  ditulis ke layar dengan format ((x,y), (x,y)) }

{ ************************************************************* }
{ KELOMPOK OPERASI TERHADAP TYPE                                }
{ ************************************************************* }
{ *** Kelompok operasi relasional terhadap GARIS *** }
function EQ (L1, L2 : GARIS) → boolean
{ Mengirimkan true jika L1  = L2 }
{ L1  dikatakan sama dengan L2 }
{ jika Titik Awal dari L =Titik awal dari L dan }
{ Titik akhir L1  = Titik akhir dari L2 }
function NEQ (L1, L2 : GARIS) → boolean
{ Mengirimkan true jika L tidak sama dengan L }
{ Negasi dari fungsi EQ }

{ *** Kelompok menentukan di mana L  berada *** }
function IsOnSbX (L : GARIS) → boolean
{ Menghasilkan true jika L  terletak pada sumbu X }
```

```
function IsOnSbY (L : GARIS) → boolean
{ Menghasilkan true jika L terletak pada sumbu Y }
function Kuadran (L : GARIS)  → integer
{ Menghasilkan kuadran dari L  (dimana PAw dan PAkh berada) }
{ Prekondisi : L  tidak terletak di salah satu sumbu, dan }
{ PAw serta PAkh selalu terletak pada kuadran yang sama }

{ *** Kelompok predikat lain *** }
function IsTegakLurus (L, L1 : GARIS) → boolean
{ Menghasilkan true jika L tegak lurus terhadap L1 }
function IsSejajar (L, L1 : GARIS) → boolean
{ Menghasilkan true jika L "sejajar" terhadap L1 }
{ *** Kelompok operasi lain *** }
function HslGeser (L : GARIS, DX, DY : integer) → GARIS
{ Menghasilkan salinan L yang titik awal dan akhirnya }
{ digeser sejauh DX dan DY }
function MirrorOf (L : GARIS, SbX, SbY : boolean) → GARIS
{ Menghasilkan salinan L  yang dicerminkan }
{ tergantung nilai SbX dan SBY }
{ Jika SbX bernilai true, maka dicerminkan thd Sumbu X }
{ Jika SbY bernilai true, maka dicerminkan thd Sumbu Y }
function Panjang (L : GARIS) → real
{ Menghitung panjang garis L : berikan rumusnya }
function Arah (L:GARIS) → real
{ Menghitung arah dari garis L }
{ yaitu sudut yang dibentuk dengan sumbu X+ }
function SudutGaris (L, L1:GARIS) → real
{ Menghasilkan sudut perpotongan antara L  dengan L1 }
{ Prekondisi : L tidak sejajar dengan L1 dan }
{              L tidak berimpit dengan L1 }
procedure Geser (input/output L : GARIS, input deltax, deltaY : integer)
{ I.S. L terdefinisi }
{ F.S. L digeser sebesar deltaX dan ordinatnya sebesar deltaY }
{ PAw dan PAkh digeser! }
procedure Mirror (input/output L : GARIS, input SbX, SbY : boolean)
{ I.S. L terdefinisi }
{ F.S. L dicerminkan tergantung nilai SbX atau SBY }
{ Jika SbX true maka dicerminkan terhadap Sumbu X }
{ Jika SbY true maka dicerminkan terhadap Sumbu Y }
{ L dicerminkan tergantung nilai SbX atau SbY }
procedure Putar (input/output L : GARIS, input Sudut : real)
{ I.S. L terdefinisi }
{ F.S. L  diputar sebesar (Sudut) derajat : PAw dan PAkh diputar }

{ ************************************************************* }
{ CONSTRAINT/INVARIANT DARI ADT                                 }
{ ************************************************************* }
{ NEQ(Pakh,PAw) dengan NEQ adalah fungsi relational utk POINT   }
```

Catatan implementasi:

- Dalam implementasi dengan bahasa C, "use" ADT lain menjadi "include" file header dari ADT yang dipakai
- Dalam implementasi dengan bahasa Ada, maka "use" menjadi kata `"with"` dalam konteks (paket yang dipakai), yang dapat disertai dengan `use` untuk menghilangkan pemanggilan dengan prefiks

- *Constraint/invariant* tidak dapat dituliskan, tetapi menjadi sebuah fungsi yang akan memeriksa validitas PAw dan PAkh sehingga dapat membentuk sebuah GARIS (lihat contoh JAM).
- Nama fungsi yang sama untuk ADT POINT dan ADT GARIS (misalnya fungsi relasional EQ dan NEQ) akan menimbulkan masalah dalam bahasa C sehingga harus diganti. Nama yang sama ini tidak menimbulkan masalah dalam bahasa Ada karena pemanggilan disertai dengan prefix nama paket, atau tanpa kata “use” dalam konteks, dengan fasilitas parameter *overloading* konflik nama ini terselesaikan.

Latihan Soal

1. Buatlah implementasi semua ADT tersebut dalam bahasa C, Pascal, dan bahasa Ada (setelah Anda mengenal bahasa Ada), dengan memperhatikan catatan implementasi tersebut.
2. Buatlah ADT JAMT dengan representasi JAM sebagai berikut :

```
{ Definisi jam dengan am dan pm }
{ **  Jam : 0:00:00 pm s.d. 11:59:59 am **}
{ Definisi TYPE JAMT <HH:MM:SS AMPM>     }
  type ampm   : enumerasi [am, pm]
  type Hour   : integer [0..11]
  type Minute : integer [0..59]
  type Second : integer [0..59]
  type JAMT   : < HH : Hour,
                  MM : Minute,
                  SS : Second,
                  T  : ampm >
```

   a. Realisasikanlah, seolah-olah Anda memulai modul JAM dengan *type* ini.
   b. Realisasikanlah, dengan asumsi bahwa Anda sudah mempunyai ADT JAM dengan representasi sebelumnya. Dengan demikian, semua operasi aritmatika dan yang lain akan dilakukan dalam *type* JAM, sehingga Anda hanya perlu membuat sebuah "*type converter*" dari JAMX menjadi JAM. Apakah *converter* sebaiknya diletakkan di dalam modul JAM atau JAMX?
3. Buatlah ADT bernama DATE, yang terdiri dari Tanggal, Bulan, dan Tahun kalender. Terjemahkan spesifikasi *type* yang terdapat pada Diktat "Pemrograman Prosedural".
4. Buatlah ADT bernama WAKTU, yang terdiri dari JAM dan DATE yang pernah didefinisikan sebelumnya. Definisikan beberapa fungsi tambahan.
5. Buatlah ADT PECAHAN untuk merepresentasi bilangan pecahan yang terdiri dari numerator dan denumerator sebagai berikut:

```
{ Definisi TYPE Pecahan }
type Pecahan : < b : integer, { bagian bulat }
                 d : integer, { numerator, pembagi, n< d }
                 n : integer, { denumerator, penyebut }
                 s : integer  { [-1,1], tanda pecahan }
               >
```

   Jika pecahan bernilai nol, maka representasinya adalah: d = 0, n = 0, dan s = 1.
   Jika b>0 maka artinya ada bagian bulat..
6. Buatlah ADT Segi4 yang mewakili segi empat, dengan definisi segi empat adalah:
   a) Sebuah segi empat yang selalu "sejajar" terhadap sumbu X dan Y, dan terdefinisi oleh dua buah titik yaitu titik Atas-Terkiri dan titik Kanan-Terbawah (*Top-Left* dan *Bottom-Right*).
   b) Sebuah segi empat terdefinisi oleh sebuah Titik, arah terhadap sumbu X+ dan juga panjang sisi-sisinya.
7. Buatlah sebuah ADT bernama BUJURSANGKAR, yang sebenarnya adalah sebuah segi empat yang spesifik, yaitu yang sisi-sisinya sama panjang. Ada dua solusi: Anda membuat *type* baru, atau memanfaatkan *type* Segi4 dan menambahkan *constraint/invariant*. Inilah gunanya mendefinisikan invarian pada sebuah ADT.

Koleksi Objek

Seringkali kita harus mengelola sekumpulan (koleksi) objek, dan tidak hanya sebuah objek secara individual. Setiap elemen dalam suatu koleksi objek harus ber-*type* sama (mungkin *type* dasar, atau suatu ADT yang pernah didefinisikan, atau suatu objek dari kelas tertentu), dan biasanya mempunyai keterurutan tertentu.

Definisi koleksi dan elemen dapat dilakukan dengan dua cara:
- Definisi iteratif: dengan mendefinisikan setiap elemen
- Definisi rekursif: suatu koleksi objek bahkan dapat mengandung elemen berupa koleksi itu sendiri

Ada koleksi objek yang dapat didefinisikan dengan kedua cara tersebut, ada yang lebih mudah melalui salah satu cara saja. Pemrosesan yang dilakukan akan lebih transparan jika dituliskan sesuai dengan definisinya.

Pada Diktat "Struktur Data" ini, ada koleksi yang akan diajarkan secara iteratif karena secara rekursif sudah dicakup di Diktat "Pemrograman Fungsional", namun ada pula yang hanya dibahas secara rekursif karena memang pada dasarnya koleksi objek tersebut lebih natural (alamiah) jika didefinisikan secara rekursif, misalnya pohon.

Koleksi objek ini:
- Diberi nama kolektif, dan nama elemennya
- Diorganisasi elemennya baik secara linier, maupun tidak linier. Elemen juga dapat mempunyai keterurutan atau tidak ada keterurutan.
- Dapat diakses setiap elemen koleksinya secara individual, lewat nama atau address.
- Bagaimana mengakses elemen yang lain berdasarkan individu elemen terdefinisi yang sedang *"current"* diproses per elemen atau seluruh koleksi.
- Banyaknya elemen suatu koleksi objek dapat "tidak ada", berarti koleksi objeknya "kosong", atau bertambah/berkurang secara dinamik.

Alokasi memori untuk seluruh koleksi dapat dilakukan:
- Sekaligus (statik).
- Setiap kali sebuah elemen akan ditambahkan ke koleksi (dinamik).

Koleksi objek adalah sebuah ADT, dengan definisi:
- *Type* koleksi.
- *Type* elemen (dan *address*-nya jika perlu).
- Akses elemen lain.
- Primitif:
  - terhadap koleksi
    - penciptaan koleksi kosong (dan alokasi jika perlu).
    - penghancuran seluruh koleksi (dan dealokasi jika perlu).
    - penambahan sebuah elemen baru menjadi anggota koleksi.
    - penghapusan sebuah elemen dari koleksi.
  - terhadap sebuah elemen tertentu:
    - konsultasi kandungan informasi elemen.
    - perubahan informasi kandungan.

- iterasi untuk memproses semua elemen (traversal).
- pencarian sebuah atau sekumpulan elemen dengan sifat tertentu (*search*).
- pengurutan elemen-elemen berdasarkan kriteria tertentu.

Pada bidang komputer, beberapa koleksi objek didefinisikan secara persis aturan-aturan penyimpanan, penambahan, penghapusan elemennya. Ini melahirkan "struktur data" standar di bidang komputer seperti:
- *List* linier.
- Matriks.
- *Stack* (tumpukan).
- *Queue* (antrian) dengan variasinya.
- List tidak linier (pohon, *graph*, multi *list*).

Koleksi Objek Generik

Struktur data standar tersebut mempunyai aturan yang sama terhadap koleksi, namun dikehendaki agar elemennya dapat ber-*type* apa pun. Ini melahirkan kebutuhan akan pendefinisian koleksi objek generik, yaitu yang *type* elemennya baru ditentukan saat koleksi tersebut diciptakan. Beberapa bahasa pemrograman menyediakan fasilitas untuk itu (misalnya bahasa Ada, C++). Beberapa bahasa tidak menyediakan fasilitas tersebut sehingga duplikasi kode terpaksa dilakukan, namun akan diajarkan secara sistematis dalam kuliah ini.

Pada konteks prosedural, mendefinisikan sebuah koleksi objek dilakukan dalam tiga tingkatan:
a. Definisi fungsional.
b. Representasi lojik.
c. Representasi (implementasi) fisik.

Definisi koleksi objek dengan definisi fungsional dan representasi lojik tertentu tanpa definisi implementasi fisik disebut struktur data "abstrak". Struktur data abstrak ini diperlukan, untuk mempermudah merancang struktur data suatu persoalan dalam terminologi yang lebih mudah dimengerti manusia, namun sulit direpresentasi langsung oleh mesin riil (bandingkan dengan mesin abstrak yang pernah dibahas).

Definisi Fungsional

Definisi fungsional adalah pendefinisan nama struktur data dan operator-operator yang berlaku pada struktur tersebut. Pada tingkatan ini, kita belum membahas tentang representasi, melainkan hanya "nama" *type* dan semua operasi yang dilakukan terhadap *type*. Secara umum, **operasi fungsional** hanya terdiri dari :
- Pembuatan/penciptaan (konstruktor).
- Penambahan (*add*, *insert*).
- Penghapusan (*delete*).
- Perubahan (*update*).
- Penggabungan.
- Operasi-operasi lain.

Perhatikanlah bahwa operasi fungsional tersebut mengubah keadaan objek. Selain itu, ada operasi fungsional yang bersifat konsultasi informasi yang disimpan, misalnya:
- Selektor.

- Predikat-predikat khusus terhadap *type*.
- Membership (untuk koleksi objek).

Pada tingkatan ini, definisi *type* sama dengan definisi yang pernah dipelajari pada Diktat "Pemrograman Fungsional", dalam notasi fungsional.

Representasi Lojik

Representasi lojik adalah spesifikasi "*type*" dari struktur, yang menyangkut nama *type* dan spesifikasi semua operator, namun dalam definisi "*type*" ini, alamat masih belum ditentukan secara pasti. Representasi lojik tak tergantung pada memori komputer. Struktur ini memudahkan pemrogram untuk merancang data dan algoritma, serta tak bergantung pada mesin apapun.

Pada konteks prosedural, pada definisi ini, didefinisikan dengan lebih jelas, operator fungsional menjadi fungsi atau prosedur, dengan spesifikasi parameter yang lebih jelas. Selain itu, dapat pula didefinisikan primitif lain yang secara khusus erat hubungannya dengan prosedural yaitu traversal atau *search*.

Representasi (Implementasi) Fisik

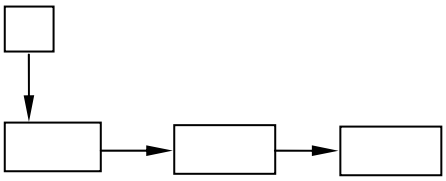
Representasi fisik adalah spesifikasi dari struktur data sesuai dengan implementasinya dalam memori komputer dan kesediaan dalam bahasa pemrograman. Pada dasarnya, hanya ada dua macam implementasi fisik: kontigu atau berkait.

Representasi fisik kontigu adalah sekumpulan data yang penempatannya dalam memori benar-benar secara fisik adalah kontigu, setiap elemen ditaruh secara berturutan posisi alamatnya dengan elemen lain. Karena itu untuk mencapai satuan elemen berikutnya, cukup melalui suksesor alamat yang sedang "*current*". Supaya suksesor tetap terdefinisi, tempat yang dialokasikan untuk struktur ini sudah ditetapkan sebelumnya, supaya data tidak "ke mana-mana". Dikatakan bahwa struktur ini adalah struktur yang statis karena alokasi memori dilakukan sekaligus untuk seluruh koleksi.

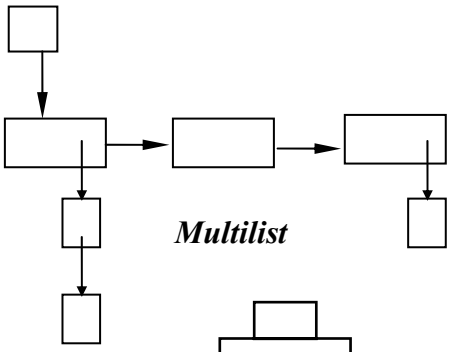
Representasi fisik berkait adalah sekumpulan data yang penempatannya pada memori komputer dapat terpencar-pencar, namun dapat ditelusuri berkat adanya informasi berupa alamat, yang menghubungkan elemen yang satu dengan yang lain. Karena alamat untuk mencapai elemen yang lain ada secara eksplisit, alamat yang bakal dipakai dapat saja dialokasikan pada waktunya atau sudah ditetapkan dari awal. Jika alokasi alamat baru diadakan pada waktu diperlukan dan juga dapat dikembalikan, maka memori yang dipakai bisa membesar dan mengecil sesuai dengan kebutuhan. Dikatakan bahwa struktur ini adalah struktur yang dinamis.

Sebuah representasi lojik yang sama dapat diimplementasikan dalam satu atau beberapa struktur fisik. Jadi, satu representasi lojik dapat mempunyai banyak kemungkinan implementasi fisik. Justru pemilihan struktur fisik yang tepat yang akan menentukan performansi (pemakaian memori dan kecepatan proses) dari program. Perancangan struktur data yang tepat merupakan bagian yang penting pada diktat ini, di samping pemilihan skema program yang sesuai terhadap struktur yang didefinisikan, sesuai dengan skema yang pernah dipelajari pada buku "Pemrograman Prosedural". Hal ini akan dipelajari melalui kasus-kasus yang ditulis pada bagian akhir buku ini.

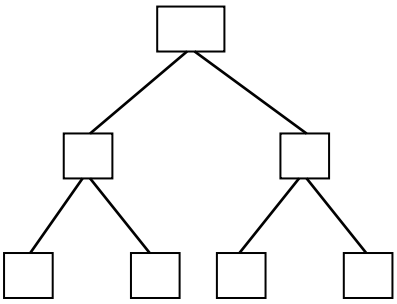
Ilustrasi struktur data yang umum dipakai untuk setiap struktur tersebut adalah :



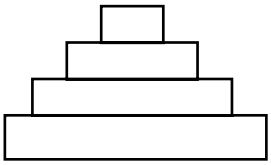
***List* Linier**



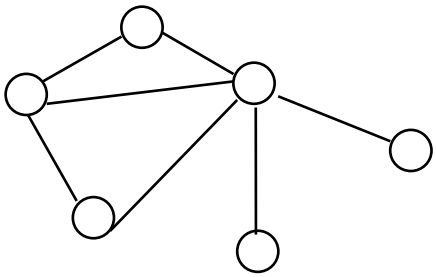
Multilist



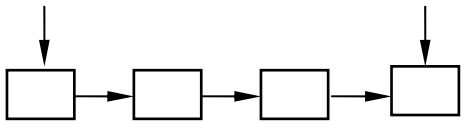
***Tree* (Pohon)**



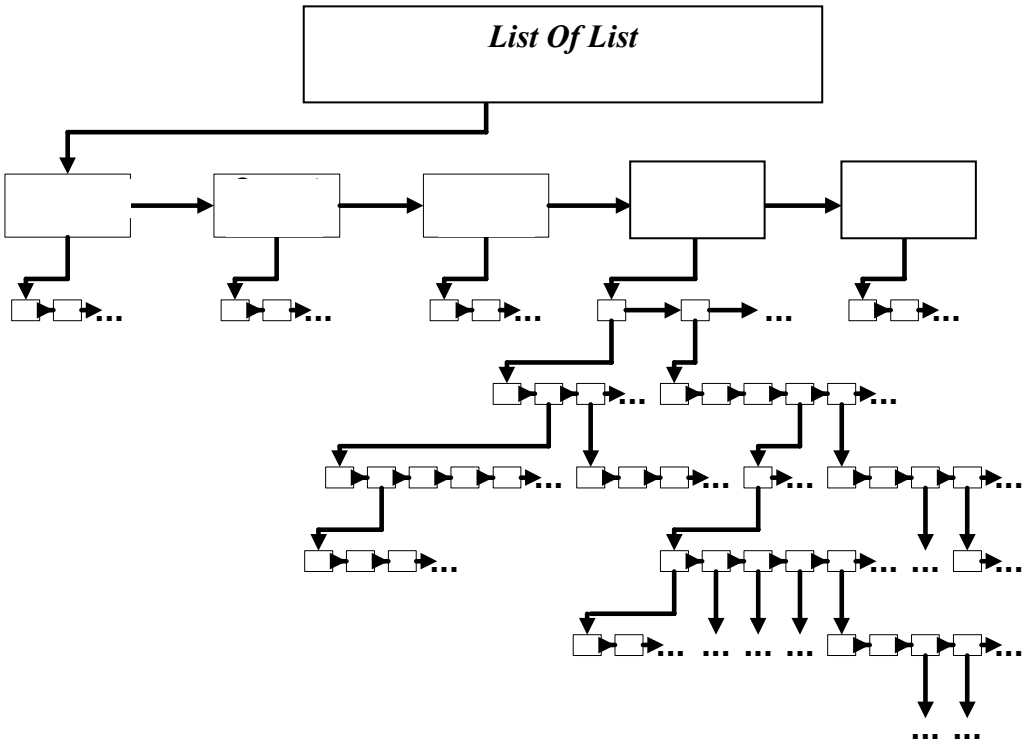
***Stack* (Tumpukan)**



***Graph* (Geraf)**



***Queue* (Antrian)**



Tabel

Tabel adalah koleksi objek yang terdiri dari sekumpulan elemen yang diorganisasi secara **kontigu**, artinya memori yang dialokasi antara satu elemen dengan elemen yang lainnya mempunyai address yang berurutan.

Pada tabel, pengertian yang perlu dipahami adalah :

- Keseluruhan **tabel** (sebagai koleksi) adalah ***container*** yang menampung seluruh elemen, yang secara konseptual dialokasi sekaligus.
- **Indeks** tabel, yang menunjukkan address dari sebuah elemen.
- **Elemen** tabel, yang apat diacu melalui indeksnya, ber*type* tertentu yang sudah terdefinisi.
- Seluruh **elemen** tabel **ber-*type*** "**sama"**. Dengan catatan: beberapa bahasa pemrograman memungkinkan pendefinisian tabel dengan elemen generik, tapi pada saat diinstansiasi, harus diinstansiasi dengan *type* yang "sama".

Memori Tabel

Alokasi memori tabel dilakukan sekaligus untuk seluruh elemen. Jika kapasitas tabel ditentukan dari awal ketika perancangan dan dilakukan pada saat deklarasi, maka. dikatakan bahwa alokasi tabel adalah **statik**. Namun, ada beberapa bahasa yang memungkinkan alokasi pada saat run time, misalnya bahasa C. Alokasi kapasitas tabel pada saat run time disebut sebagai alokasi **dinamik**.

Tabel dikatakan **"kosong"** jika memori yang dialokasi belum ada yang didefinisikan elemennya (walaupun dalam bahasa pemrograman tertentu sudah diisi dengan nilai *default*).

Tabel dikatakan **"penuh"** jika semua memori yang dialokasi sudah diisi dengan elemen yang terdefinisi.

Implementasi Fisik dari Indeks Tabel

Indeks tabel harus suatu *type* yang mempunyai keterurutan (ada suksesor dan predesesor), misalnya *type* integer, karakter, *type* terenumerasi.
Jika indeksnya adalah integer, maka keterurutan indeks sesuai dengan urutan integer (suksesor adalah plus satu, predesesor adalah minus satu)
Jika indeksnya ditentukan sesuai dengan enumerasi (misalnya ber*type* pada karakter atau *type* enumerasi lain yang didefinisikan), maka keterurutan indeks ditentukan sesuai dengan urutan enumerasi.

Pemroses bahasa akan melakukan kalkulasi *address* untuk mendapatkan elemen yang ke-i. Beberapa bahasa memberikan kebebasan kepada pemrogram untuk mendefinisikan *range* nilai dari indeks (misalnya Ada, Fortran77), beberapa bahasa lain secara default mendefinisikan indeks dari 0 atau 1 (Misalnya bahasa C: mulai dari 0)

Notasi Algoritmik:
Jika T adalah sebuah tabel dengan elemen ber*type* Eltype, dan indeks tabel terdefinisi untuk nilai IdxMin..IdxMax, maka deklarasinya secara algoritmik adalah :

```
T : array [idxmin..idxMax] of Eltype
```

Untuk mengacu sebuah elemen ke-i tuliskan `T(i)`.

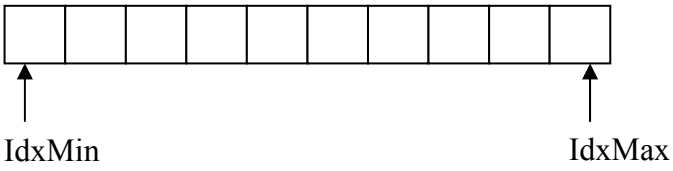
ADT Tabel dengan Elemen Kontigu

Tabel dapat dipakai sebagai salah satu cara representasi koleksi objek berupa list linier dengan representasi kontigu. Representasi kontigu artinya *subset* atau seluruh ruang memori tabel dipakai untuk merepresentasi sekumpulan elemen list

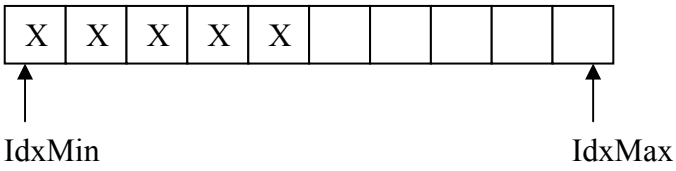
Ada dua model representasi tabel:

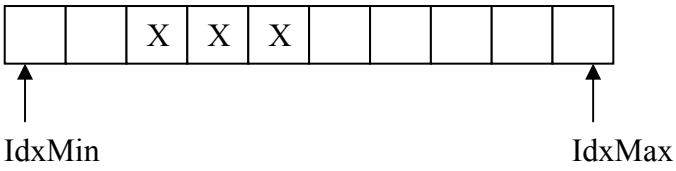
1. Model representasi dengan banyaknya elemen secara **implisit**

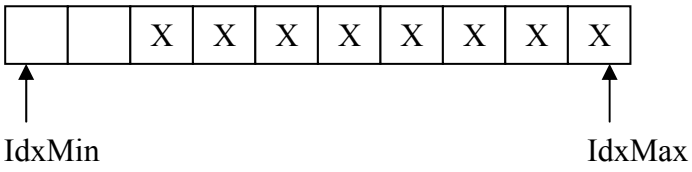
Contoh **tabel kosong**



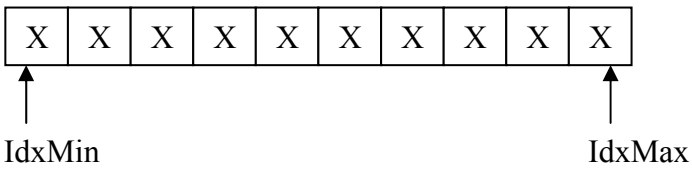
Contoh tabel terisi sebagian :







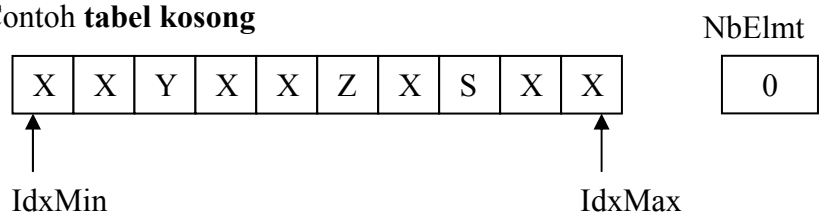
Contoh **tabel penuh**:



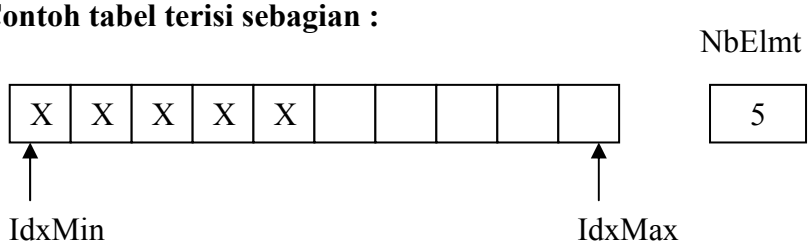
Dengan model representasi semacam ini, batas indeks elemen efektif tidak diketahui. Diperlukan suatu nilai yang khusus untuk mengetahui "batas" daerah memori yang diisi (efektif atau tidak). Nilai khusus ini kadang-kadang sulit untuk tabel dengan elemen yang domainnya tertentu, misalnya integer. Oleh karena itu, perlu dipikirkan cara representasi yang lain.

2. Model representasi dengan banyaknya elemen secara **eksplisit**

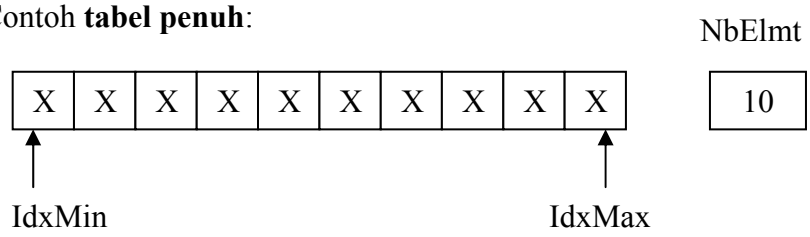
Contoh **tabel kosong**



Contoh tabel terisi sebagian :



Contoh **tabel penuh**:



Dengan representasi seperti ini, definisi tabel mencakup definisi ruang memori, dan juga definisi kandungan elemen yang "efektif" (artinya yang "terdefinisi").

Dengan cara ini, tidak dibutuhkan nilai khusus untuk menentukan ruang memori "kosong". Tabel kosong tidak perlu diinisialisasi nilai memorinya.

Hanya contoh ADT Tabel berelemen integer dengan penempatan elemen kontigu yang diberikan dalam diktat ini.

Perhatikanlah bahwa implementasi dalam beberapa bahasa, akan membutuhkan hal-hal yang spesifik pada bahasa tersebut. Dalam kuliah ini, akan distudi representasi dengan bahasa Ada dan bahasa C.

ADT Tabel dengan Elemen "Tersebar"

Tabel dapat dipakai sebagai salah satu cara representasi koleksi objek dengan elemen yang "tersebar". Artinya, pada suatu saat, tabel sebagai ruang memori dapat diisi atau tidak. Representasi "isi" atau "kosong" dapat bermacam-macam, antara lain:

- dengan nilai *boolean true* atau *false*. Dengan representasi ini, hanya diketahui informasi "isi" atau "kosong" tanpa informasi lain. Contoh: pemakaian pada tabel untuk merepresentasikan status tempat duduk atau memori.
- dengan nilai sesuai dengan domain nilai tabel. Misalnya: untuk tabel berelemen integer, maka "*blank*" artinya kosong, sedangkan sebuah huruf lain artinya berisi huruf yang bersangkutan.

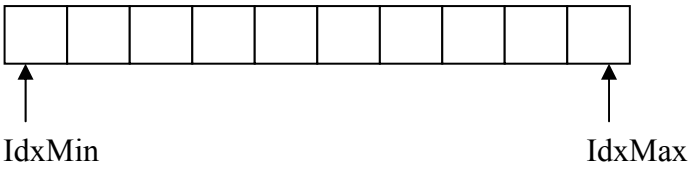
Penempatan sebuah elemen pada suatu posisi indeks dapat ditentukan oleh:

- sebuah algoritma pencarian tempat secara sekuensial.
- sebuah algoritma penempatan secara "random" (*hashing*), yang tidak diberikan dalam kuliah ini.
- input secara manual (misalnya dalam suatu sistem *ticketing* pesawat).

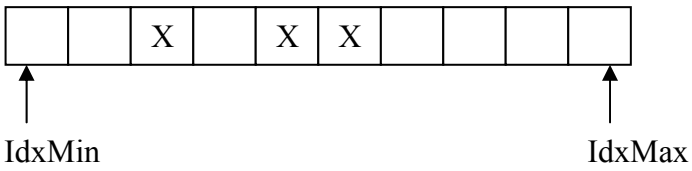
Apapun algoritma penempatan elemennya, maka pada penambahan sebuah elemen harus diperiksa apakah tempat untuk meletakkan memang berstatus "kosong", dan penghapusan sebuah elemen pada indeks tersebut memang merupakan posisi yang "isi"

Ilustrasi tabel dengan elemen dengan elemen tersebar dan banyaknya elemen tidak disimpan secara eksplisit adalah sebagai berikut :

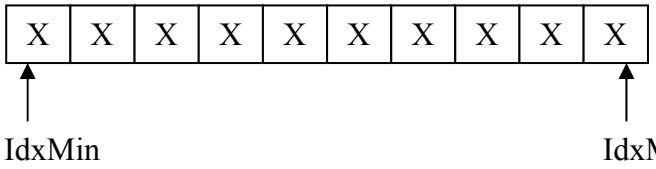
Contoh tabel kosong: jika "kosong"/"*blank*" adalah "kosong"



Contoh tabel dengan 3 elemen



Contoh tabel "penuh" dengan X adalah karakter sembarang



Pertanyaan:
Mungkinkah suatu tabel dengan elemen tersebar direpresentasi dengan banyaknya elemen secara eksplisit?

ADT Tabel dengan Elemen Kontigu dalam Bahasa Algoritmik

```
{  MODUL TABEL INTEGER DENGAN ALOKASI STATIK }
{  berisi definisi dan semua primitif pemrosesan tabel integer }
{  penempatan elemen selalu rapat kiri }
{  Kamus Umum }
{  Versi I : dengan Banyaknya elemen secara explisit, tabel statik }
constant IdxMax : integer = 100
constant IdxMin : integer = 1
constant IdxUndef : integer = -999 { indeks tak terdefinisi}
{ Definisi elemen dan koleksi objek }
type IdxType : integer  { type indeks }
type ElType  : integer  { type elemen tabel }
type TabInt  : < TI : array [IdxMin..IdxMax] of ElType,
                          { memori tempat penyimpan elemen (container) }
                 Neff : integer [IdxMin..IdxMax] { banyaknya elemen efektif }
               >
{ Jika T adalah Tabint, cara deklarasi dan akses: }
{ Deklarasi }
{ T : Tabint }
{ Maka cara akses: }
{ T.Neff untuk mengetahui banyaknya elemen }
{ T.TI  untuk mengakses seluruh nilai elemen tabel }
{ T.TI(i)  untuk mengakses elemen ke-i }
{ Definisi : }
{ Tabel kosong: T.Neff = 0}
{ Definisi elemen pertama : T.TI(i) dengan i=1 }
{ Definisi elemen terakhir yang terdefinisi: T.TI(i)dengan i=T.Neff }

{ ********** KONSTRUKTOR ********** }
{ Konstruktor: create tabel kosong  }
procedure MakeEmpty (output T : Tabint)
{ I.S. sembarang }
{ F.S. Terbentuk tabel T kosong dengan kapasitas Nmax-Nmin+1 }

{ ********** SELEKTOR ********** }
{ *** Banyaknya elemen *** }
function NbElmt (T : Tabint) → integer
{ Mengirimkan banyaknya elemen efektif tabel }
{ Mengirimkan nol jika tabel kosong }
{ *** Daya tampung container *** }
function MaxNbEl (T : Tabint) → integer
{ Mengirimkan maksimum elemen yang dapat ditampung oleh tabel }

{ ********** INDEKS ********** }
function GetFirstIdx (T : Tabint) → IdxType
{ Prekondisi : Tabel tidak kosong}
{ Mengirimkan indeks elemen pertama}
function GetLastIdx (T : Tabint) → IdxType
{ Prekondisi : Tabel tidak kosong}
{ Mengirimkan indeks elemen terakhir}

{ ********** Menghasilkan sebuah elemen ********** }
function GetElmt (T : Tabint, i : IdxType) → ElType
{ Prekondisi : Tabel tidak kosong, i antara FirstIdx(T)..LastIdx(T) }
{ Mengirimkan elemen tabel yang ke-i }

{ ********** Test Indeks yang valid ********** }
function IsIdxValid (T : Tabint, i : IdxType) → boolean
{ Prekondisi : i sembarang }
{ Mengirimkan true jika i adalah indeks yang valid utk ukuran tabel }
{ yaitu antara indeks yang terdefinisi utk container}
function IsIdxEff (T : Tabint, i : IdxType) → boolean
{ Prekondisi : i sembarang}
{ Mengirimkan true jika i adalah indeks yang terdefinisi utk tabel }
{ yaitu antara FirstIdx(T)..LastIdx(T) }
```

```
{ ********** Selektor SET : Mengubah nilai TABEL dan elemen tabel ********** }
{ Untuk type private/limitted private pada bahasa tertentu }
procedure SetTab (input Tin : Tabint, output Tout : Tabint)
{ I.S. Tin terdefinisi, sembarang }
{ F.S. Tout berisi salinan Tin }
{ Assignment THsl ← Tin }
procedure SetEl (input/output T : Tabint, input i : IdxType, input v : ElType)
{ I.S. T terdefinisi, sembarang }
{ F.S. Elemen T yang ke-i bernilai v }
{ Mengeset nilai elemen tabel yang ke-i sehingga bernilai v }
procedure SetNeff (input/output T : Tabint, input N : IdxType)
{ I.S. T terdefinisi, sembarang }
{ F.S. Nilai indeks efektif T bernilai N }
{ Mengeset nilai indeks elemen efektif sehingga bernilai N }

{ ********** TEST KOSONG/PENUH ********** }
{ *** Test tabel kosong *** }
function IsEmpty (T : Tabint) → boolean
{ Mengirimkan true jika tabel kosong, mengirimkan false jika tidak }
{ *** Test tabel penuh *** }
function IsFull(T : Tabint) → boolean
{ Mengirimkan true jika tabel penuh, mengirimkan false jika tidak }

{ ********** BACA dan TULIS dengan INPUT/OUTPUT device ********** }
{ *** Mendefinisikan isi tabel dari pembacaan *** }
procedure BacaIsi (output T : Tabint)
{ I.S. sembarang }
{ F.S. tabel T terdefinisi }
{ Proses : membaca banyaknya elemen dan mengisi nilainya}
procedure TulisIsi (input T : Tabint)
{ Proses : Menuliskan isi tabel dengan traversal}
{ I.S. T boleh kosong  }
{ F.S. Jika T tidak kosong: indeks dan elemen tabel ditulis berderet ke bawah,
jika tidak kosong }
{ Jika tabel kosong : Hanya menulis "Tabel kosong" }
procedure TulisIsiTab (input T : Tabint)
{ Proses : Menuliskan isi tabel dengan traversal, Tabel ditulis di antara kurung
siku; antara dua elemen dipisahkan dengan separator "koma"}
{ I.S. T boleh kosong }
{ F.S. Jika T tidak kosong: [ e1, e2, … ,en] }
{ Contoh : jika ada tiga elemen bernilai 1,20,30: [1,20,3] }
{ Jika tabel kosong : menulis [ ] }

{ ********** OPERATOR ARITMATIKA ********** }
{ *** Aritmatika tabel : Penjumlahan, pengurangan, perkalian, ... *** }
function PlusTab (T1, T2 : Tabint) → TabInt
{ Prekondisi : T1 dan T2 berukuran sama dan tidak kosong }
{ Mengirimkan  T1+T2 }
function MinusTab (T1, T2 : Tabint) → Tabint
{ Prekondisi : T1 dan T2 berukuran sama dan tidak kosong }
{ Mengirimkan T1-T2 }
function KaliTab (T1, T2 : Tabint) → TabInt
{ Prekondisi : T1 dan T2 berukuran sama dan tidak kosong }
{ Mengirimkan T1*T2 dengan definisi setiap elemen dengan indeks yang sama
dikalikan }
function KaliKons (Tin : TabInt, c : ElType) → Tabint
{ Prekondisi : Tin tidak kosong }
{ Mengirimkan tabel dengan setiap elemen Tin dikalikan c }

{ ********** OPERATOR RELASIONAL ********** }
{ *** Operasi pembandingan tabel : < =, > *** }
function IsEQ (T1 : Tabint, T2 : Tabint) → boolean
{ Mengirimkan true jika T1=T2 }
{ yaitu jika ukuran T1 = T2 dan semua elemennya sama }
```

```
function IsLess (T1 : Tabint, T2 : Tabint) → boolean
{ Mengirimkan true jika T1< T2, }
{ yaitu : sesuai dg analogi 'Ali' < Badu'; maka [0,1] < [2,3] }

{ ********** TRAVERSAL ********** }
procedure Trav1 (input T : Tabint)
{ I.S. Tabel mungkin kosong }
{ F.S. Tabel diproses dengan prosedur P (e: integer) }
{ Jika tabel tidak kosong: }
{ Proses sekuensial dengan penanganan kasus kosong, }
{ Model tanpa mark }
{ Memproses elemen tabel satu demi satu secara berturutan, }
{ Mulai dari elemen pertama s.d. elemen terakhir }
{ Jika tabel kosong : hanya menuliskan pesan tabel kosong }
{ ***  Search: tabel boleh kosong!! *** }
function Search1 (T : Tabint, X : ElType) → IdxType
{ Search apakah ada elemen tabel T yang bernilai X }
{ Jika ada, menghasilkan indeks i terkecil, dg elemen ke-i=X }
{ Jika tidak ada, mengirimkan IdxUndef }
{ Menghasilkan indeks tak terdefinisi(IdxUndef) jika tabel kosong }
{ *** Memakai skema search TANPA boolean Found ***}
function Search2 (T: Tabint, X : ElType) → IdxType
{ Search apakah ada elemen tabel T yang bernilai X }
{ Jika ada, menghasilkan indeks i terkecil, dg elemen ke-i=X }
{ Jika tidak ada, mengirimkan IdxUndef }
{ Menghasilkan indeks tak terdefinisi (IdxUndef) jika tabel kosong }
{ *** Memakai skema search DENGAN boolean Found *** }
function SearchB (T : Tabint, X : ElType) → boolean
{ Search apakah ada elemen tabel T yang bernilai X }
{ Jika ada, menghasilkan true, jika tidak ada menghasilkan false }
{ Menghasilkan indeks tak terdefinisi (IdxUndef) jika tabel kosong }
{ Memakai Skema search DENGAN boolean }
function SearchSentinel (T : Tabint, X : ElType) → integer
{ Prekondisi : Tabel belum penuh }
{ Search apakah ada elemen tabel T yang bernilai X }
{ Jika ada, menghasilkan true, jika tidak ada menghasilkan false }
{ dengan metoda sequential search dengan sentinel }

{ ********** NILAI EKSTREM ********** }
function ValMax (T : Tabint) → ElType
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan nilai maksimum tabel }
function ValMin (T : Tabint) → ElType
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan nilai minimum tabel }
{ *** Mengirimkan indeks elemen bernilai ekstrem *** }
function IdxMaxTab (T : Tabint) → IdxType
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan indeks i }
{ dengan elemen ke-i adalah nilai maksimum pada tabel }
function IdxMinTab (T : Tabint) → IdxType
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan indeks i }
{ dengan elemen ke-i nilai minimum pada tabel }

{ ********** OPERASI LAIN ********** }
procedure CopyTab (input Tin : Tabint, output Tout : Tabint)
{ I.S. sembarang }
{ F.S. Tout berisi salinan dari Tin (elemen dan ukuran identik) }
{ Proses : Menyalin isi Tin ke Tout }
function InverseTab (T : Tabint) → Tabint
{ Menghasilkan tabel dengan urutan tempat yang terbalik, yaitu: }
{ elemen pertama menjadi terakhir, }
{ elemen kedua menjadi elemen sebelum terakhir, dst.. }
{ Tabel kosong menghasilkan tabel kosong }
```

```
function IsSimetris (T : Tabint) → boolean
{ Menghasilkan true jika tabel simetrik }
{ Tabel disebut simetrik jika: }
{       elemen pertama = elemen terakhir, }
{       elemen kedua = elemen sebelum terakhir, dan seterusnya }
{ Tabel kosong adalah tabel simetris }

{ ********** SORTING ********** }
procedure MaxSortAsc (input/output T : Tabint)
{ I.S. T boleh kosong }
{ F.S. T elemennya terurut menaik dengan Maximum Sort }
{ Proses : mengurutkan T sehingga elemennya menaik/membesar }
{ tanpa menggunakan tabel kerja }
procedure InsSortDesc (input/output T : Tabint)
{ I.S. T boleh kosong }
{ F.S. T elemennya terurut menurun dengan Insertion Sort }
{ Proses : mengurutkan T sehingga elemennya menurun/mengecil }
{ tanpa menggunakan tabel kerja }

{ ********** MENAMBAH ELEMEN ********** }
{ *** Menambahkan elemen terakhir *** }
procedure AddAsLastEl (input/output T : Tabint, input X : ElType)
{ Menambahkan X sebagai elemen terakhir tabel }
{ I.S. Tabel boleh kosong, tetapi tidak penuh }
{ F.S. X adalah elemen terakhir T yang baru }
{ Menambahkan sebagai elemen ke-i yang baru }
procedure AddEli (input/output T : Tabint, input X : ElType, input i : IdxType)
{ Menambahkan X sbg elemen ke-i tabel tanpa mengganggu kontiguitas terhadap
elemen yang sudah ada }
{ I.S. Tabel kosong dan tidak penuh }
{       i adalah indeks yang valid. }
{ F.S. X adalah elemen ke-i T yang baru }
{ Proses : Geser elemen ke-i+1..terakhir }
{ Isi elemen ke-i dengan X }
{ Pertanyaan : Mengapa pada I.S. tabel tidak kosong dan juga tidak penuh ? }

{ ********** MENGHAPUS ELEMEN ********** }
procedure DelLastEl (input/output T : Tabint, output X : ElType)
{ Proses : Menghapus elemen terakhir tabel }
{ I.S. Tabel tidak kosong, }
{ F.S. X adalah nilai elemen terakhir T sebelum penghapusan, }
{ Banyaknya elemen tabel berkurang satu }
{ Tabel mungkin menjadi kosong }

procedure DelEli(input/output T : Tabint, input i : IdxType, output X : ElType)
{ Proses : Menghapus elemen ke-i tabel tanpa mengganggu kontiguitas }
{ I.S. Tabel tidak kosong, i adalah indeks efektif yang valid }
{ F.S. Elemen T berkurang satu }
{ Banyaknya elemen tabel berkurang satu }
{ Tabel mungkin menjadi kosong }
{ Proses : Geser elemen ke-i+1 s.d. elemen terakhir }
{ Kurangi elemen effektif }

{ ********** ADD/DELETE Tabel dengan elemen UNIK ********** }
{ Tabel elemennya UNIK (hanya muncul satu kali) }
procedure AddElUnik (input/output T : Tabint, input X : ElType)
{ Menambahkan X sebagai elemen terakhir tabel, pada tabel dengan elemen unik}
{ I.S. Tabel boleh kosong, tetapi tidak penuh }
{ dan semua elemennya bernilai unik, tidak terurut}
{ F.S. Jika tabel belum penuh,menambahkan X sbg elemen terakhir T,  jika belum
ada elemen yang bernilai X. Jika sudah ada elemen tabel yang bernilai X maka
I.S. = F.S. dan dituliskan pesan "nilai sudah ada" }
{ Proses : Cek keunikan dengan sequential search dengan sentinel}
{ Kemudian tambahkan jika belum ada }
```

```
{ *** Untuk tabel dengan elemen terurut membesar *** }
function SearchUrut (T : Tabint, X : ElType) → IdxType
{ Prekondisi: Tabel boleh kosong }
{ mengirimkan indeks di mana harga X dengan indeks terkecil diketemukan }
{ mengirimkan IdxUndef jika tidak ada elemen tabel bernilai X }
{ Menghasilkan indeks tak terdefinisi(IdxUndef) jika tabel kosong }

function Max (T : Tabint) → ElType
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan nilai maksimum pada tabel}
function Min (T : Tabint) → ElType
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan nilai minimum pada tabel}
function MaxMin (T : Tabint) → <ElType, ElType>
{ Prekondisi : Tabel tidak kosong }
{ Mengirimkan nilai maksimum dan minimum pada tabel }

procedure Add1Urut (input/output T : Tabint, input X : ElType)
{ Menambahkan X tanpa mengganggu keterurutan nilai dalam tabel }
{ Nilai dalam tabel tidak harus unik }
{ I.S. Tabel boleh kosong, boleh penuh }
{ F.S. Jika tabel belum penuh, menambahkan X }
{ Proses : Search tempat yang tepat sambil geser }
{ Insert X pada tempat yang tepat tersebut tanpa mengganggu keterurutan }

procedure Del1Urut (input/output T : Tabint, input X : ElType)
{ Menghapus X yang pertama kali (indeks terkecil) yang ditemukan }
{ I.S. Tabel tidak kosong, }
{ F.S. jika ada elemen tabel bernilai X , }
{ maka banyaknya elemen tabel berkurang satu. }
{ Jika tidak ada yang bernilai X, tabel tetap. }
{ Setelah penghapusan, elemen tabel tetap kontigu! }
{ Proses : Search indeks ke-i dg elemen ke-i = X }
{          Delete jika ada }

{ *** ADD/DELETE Tabel dengan elemen TERURUT dan UNIK *** }
{ Latihan membuat spesifikasi }
```

Catatan implementasi dalam bahasa Ada:

- Pada umumnya dalam bahasa Ada, indeks tabel baru ditentukan *range*-nya ketika variabel didefinisikan. Maka definisi indeks idealnya adalah (POSITIVE <>). Cobalah implementasi dan tuliskanlah akibatnya.
- Dalam bahasa Ada, jika *type* tabel diimplementasi menjadi *private* atau bahkan *limited private*, maka organisasi prosedur juga perlu disesuaikan (beberapa menjadi *private*).
- Deklarasi nama *type* "ElType", IdxType dapat menimbulkan persoalan serius mengenai kompatibilitas *type*.

Catatan implementasi dalam bahasa C:

- Indeks tabel dalam bahasa C dimulai dari 0.
- Representasi fisik tabel dalam bahasa C dapat secara dinamik dengan pointer. Namun secara konseptual, alokasi tetap dilakukan sekaligus (beberapa versi bahasa C memungkinkan tabel yang dapat berkembang/menyusut elemennya, tidak dibahas dalam kuliah ini). Jika tabel didefinisikan secara dinamik, maka harus ada alokasi untuk prosedur CreateEmpty. Jika alokasi gagal, apa akibatnya? Selain itu, untuk tabel dengan alokasi dinamik, harus dibuat destruktor untuk melakukan dealokasi memori

Catatan secara umum:

- Jika *type* dari elemen adalah sembarang *type* (generik), maka ada prosedur/fungsi yang berlaku dan ada yang tidak. Coba analisis semua prosedur dan sebutkanlah mana yang berlaku untuk tabel dengan elemen apapun, dan apa yang spesifik misalnya jika elemen tabel diinstansiasi menjadi *type* yang pernah dibahas sebelumnya:
  - character
  - JAM
  - POINT
  - GARIS
- Tidak semua prosedur/fungsi yang ditulis dapat dipakai untuk semua *type*. Misalnya ElType bukan integer, maka operasi penjumlahan, pengurangan, dan sebagainya tidak dapat dilakukan. Sebaiknya dilakukan pengelompokan terhadap prosedur/fungsi:
  - prosedur/fungsi mana yang tergantung *type* tertentu,
  - prosedur/fungsi yang dapat dipakai untuk *type* apapun (misalnya CopyTab).
- Implementasi modul tersebut secara lengkap mempunyai banyak persoalan dengan bahasa prosedural "biasa".
- Beberapa persoalan mengharuskan kita mengelola tabel sebagai variabel global, dan bukan sebagai parameter karena untuk seluruh sistem hanya ada satu tabel saja. Misalnya tabel *user* pada sistem jaringan; tabel simbol pada proses kompilasi. Apakah modul ADT di atas masih dapat dipakai? Jika tetap dibuat sama, apa akibatnya? Jika Anda sarankan untuk diubah, apa perubahannya?
- Pada pemakaiannya, jarang sekali kita memakai semua fungsi tabel di atas. Anda hanya akan memakai sebagian. Apa yang sebaiknya dilakukan jika Anda sudah mempunyai kode tersebut?

Matriks

Matriks adalah sekumpulan informasi yang setiap individu elemennya terdefinisi berdasarkan dua buah indeks (yang biasanya dikonotasikan dengan baris dan kolom). Setiap elemen matriks dapat diakses secara langsung jika kedua indeks diketahui, dan indeksnya harus ber*type* yang mempunyai keterurutan (suksesor), misalnya integer. Matriks adalah struktur data dengan memori internal. Struktur ini praktis untuk dipakai tetapi memakan memori! (Matriks integer 100 x 100 memakan 10000 x tempat penyimpanan integer.)

Sering dikatakan bahwa matriks adalah tabel atau array berdimensi 2. Tetapi patut diperhatikan, bahwa pengertian "dimensi 2", "baris dan kolom" adalah dalam pemikiran kita. Pengaturan letak elemen matriks dalam memori komputer selalu tetap sebagai deretan sel "linier". Pengertian 2 dimensi ini hanya untuk mempermudah pemrogram dalam mendesain programnya. Maka matriks adalah salah satu contoh struktur data "lojik".

Contoh : untuk matriks 3x4 sebagai berikut:

1	2	3	4
5	6	7	8
9	10	11	12

Dapat disimpan secara linier dan kontigu dengan dua alternatif sebagai berikut:

a. Per baris

1	2	3	4	5	6	7	8	9	10	11	12

b. Per kolom

1	5	9	2	6	10	3	7	11	4	8	12

Banyaknya baris dan banyaknya kolom biasanya disebut sebagai ukuran matriks. Contoh: matriks berukuran 4 x 5 artinya mempunyai baris sebanyak 4 dan kolom sebanyak 5, sehingga dapat menyimpan 20 elemen. Ada beberapa bahasa pemrograman yang meminta ukuran matriks pada pendefinisiannya, ada yang meminta penomoran minimum dan maksimum dari baris dan kolom. Pada notasi algoritmik yang kita pakai, cara kedua yang akan dipakai, sebab ukuran matriks dapat dideduksi dari penomorannya.

Matriks adalah struktur data yang “statik”, yaitu ukuran maksimum memorinya ditentukan dari awal. Batas indeks baris dan kolom harus terdefinisi dengan pasti saat dideklarasi dan tak dapat diubah-ubah. Seringkali dalam persoalan semacam ini, kita memesan memori secara “berlebihan” untuk alasan terjaminnya memori yang tersedia, dan hanya memakai sebagian saja. Biasanya memori yang dipakai (selanjutnya disebut efektif) adalah yang “kiri atas” seperti ilustrasi sebagai berikut, dimana pada saat deklarasi, memori maksimum yang disediakan adalah 10x10, dan hanya akan dipakai untuk 3X4

Jika bahasa yang menangani matriks tidak menentukan spesifikasi inisialisasi nilai pada saat memori dialokasi, maka :

	1	2	3	4	5	6	7	8	9	10
1	1	1	1							
2	2	2	2							
3	3	3	3							
4	4	4	4							
5										
6										
7										
8										
9										
10										

"Linierisasi" per baris akan menghasilkan nilai :
{1,1,1,?,?,?,?,?,?,?}, {2,2,2,?,?,?,?,?,?,?}, {3,3,3,?,?,?,?,?,?,?},{4,4,4,?,?,?,?,?,?,?}
{?,?,?,?,?,?,?,?,?,?}, {?,?,?,?,?,?,?,?,?,?}, {?,?,?,?,?,?,?,?,?,?},{?,?,?,?,?,?,?,?,?,?}
{?,?,?,?,?,?,?,?,?,?}, {?,?,?,?,?,?,?,?,?,?}

Sedangkan linierisasi per kolom akan menghasilkan nilai :
{1,2,3,4,?,?,?,?,?,?), {1,2,3,4,?,?,?,?,?,?), {1,2,3,4,?,?,?,?,?,?), {1,2,3,4,?,?,?,?,?,?),
{?,?,?,?,?,?,?,?,?,?}, {?,?,?,?,?,?,?,?,?,?}, {?,?,?,?,?,?,?,?,?,?},{?,?,?,?,?,?,?,?,?,?}
{?,?,?,?,?,?,?,?,?,?}, {?,?,?,?,?,?,?,?,?,?}

Akibat dari "linierisasi" yang tergantung kepada bahasa pemrograman tersebut, dan pemakaian memori yang "hanya sebagian" dari keseluruhan memori yang dipesan, maka passing parameter sebuah matriks dapat menimbulkan kesalahan.

Misalnya sebuah fungsi atau prosedur mempunyai parameter formal sebuah matriks dengan dimensi 6x6, dan bahasanya akan mengolah per kolom
Jika parameter aktual adalah sebuah matriks berukuran 3x4 dengan nilai

1	2	3	4
5	6	7	8
9	10	11	12

Maka ketika nilai ditampung dalam prosedur berparameter formal matriks 6x6, dengan traversal i dan j untuk i $\in$ [1..3], j $\in$ [1..4] akan di proses dengan beberapa nilai tak terdefinisi yaitu {1,5,9,10,3,7,8,12,?,?,?,?}, seperti digambarkan sebagai berikut :

1	10	8	?	?	?
5	3	12	?	?	?
9	7	?	?	?	?
2	11	?	?	?	?
6	4	?	?	?	?

Maka, sebaiknya jika merancang prosedur atau fungsi yang mempunyai parameter, ukuran parameter formal harus sama dengan parameter aktual.

Beberapa bahasa, misalnya bahasa Fortran, menyediakan fasilitas "*adjustable dimension*", yaitu ukuran parameter formal matriks belum ditentukan dimensinya, dan baru ditentukan saat parameter aktual diberikan. Fasilitas ini mengurangi kesalahan yang terjadi.

Beberapa contoh matriks dan isinya:

1. MatNamaHari [1..7,1..3]: Nama hari ke 1 s.d. 7 dalam 3 bahasa (Indonesia, Inggris, Prancis)

	1 = INDONESIA	2 = INGGRIS	3 = PRANCIS
1	Senin	Monday	Lundi
2	Selasa	Tuesday	Mardi
3	Rabu	Wednesday	Mercredi
4	Kamis	Thursday	Jeudi
5	Jumat	Friday	Vendredi
6	Sabtu	Saturday	Samedi
7	Minggu	Sunday	Dimanche

2. A [1..5,1..5]: Matriks bilangan real

	1	2	3	4	5
1	12.1	7.0	8.9	0.7	6.6
2	0.0	1.6	2.1	45.9	55.0
3	6.1	8.0	0.0	3.1	21.9
4	9.0	1.0	2.7	22.1	6.2
5	5.0	0.8	0.8	2.0	8.1

3. MatFrek [‘A’..’E’,1..7]: Matriks frekuensi kemunculan huruf ‘A’ s.d. ‘E’ pada hasil pemeriksaan 7 pita karakter

	1	2	3	4	5	6	7
‘A’	12	71	82	0	62	30	11
‘B’	0	1	2	45	5	3	10
‘C’	6	8	0	3	21	3	6
‘D’	9	1	2	22	6	9	7
‘E’	5	0	0	2	8	45	23

4. MatSurvey [1..4,1..7]: Matriks hasil survey pada titik kordinat. Mat(i,j) adalah hasil pengukuran <temperatur, kecepatan angin> pada titik koordinat i, j.

	1	2	3	4
1	<24,5>	<24,5>	<30,5>	<25,5>
2	<23,56>	<3,6>	<40,5>	<2,2>
3	<22,73>	<7,3>	<60,6>	<8,3>
4	<21,56>	<8,5>	<9,8>	<7,4>
5	<23,56>	<12,50>	<3,36>	<30,6>
6	<20,0>	<2,56>	<5,46>	<20,99>
7	<30,0>	<9,0>	<15,0>	<27,0>

5. MatSat [1..4,1..4]: Matriks satuan

	1	2	3	4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

6. MatSym [1..6,1..6]: Matriks simetris

	1	2	3	4	5	6
1	1	0	10	0	4	33
2	0	12	0	0	3	4
3	10	0	11	0	4	3
4	0	0	0	1	0	2
5	4	3	4	0	8	1
6	33	4	3	2	1	0

Contoh pemakaian matriks:

- Matriks banyak digunakan dalam komputasi numerik untuk representasi dalam *finite element*.
- Seperti penggunaan matriks dalam matematika. Perhitungan "biasa" terhadap matriks : penjumlahan, perkalian dua matriks, menentukan determinan, menginvers sebuah matriks, memeriksa apakah sebuah matriks: simetris, matriks satuan. Hanya saja dalam algoritma, semua "perhitungan" itu menjadi tidak primitif, harus diprogram.
- Dalam perhitungan ilmiah di mana suatu sistem diwakili oleh matriks (elemen hingga dalam teknik sipil dan mesin).
- Dalam persoalan pemrograman linier dan *operational research*.
- Dalam persoalan algoritmik: untuk menyimpan informasi yang cirinya ditentukan oleh 2 komponen (yang nantinya diterjemahkan dalam baris dan kolom) dan diakses langsung.
  Contoh: merepresentasi ”*cell*“ pada sebuah *spreadsheet*, merepresentasi ”ruangan“ pada sebuah gedung bertingkat dan sebagainya.

Pada kuliah ini, hanya akan diajarkan algoritma terhadap struktur matriks yang sangat sederhana karena algoritma lain akan dipelajari di mata kuliah lain.

Notasi algoritmik dari matriks:
NamaMatriks (indeks1, indeks2)
Domain :

- Domain matriks sesuai dengan pendefinisian indeks
- Domain isi matriks sesuai dengan jenis matriks

Konstanta :

- Konstanta untuk seluruh matriks tidak terdefinisi
- Konstanta hanya terdefinisi jika indeks dari matriks terdefinisi

Implementasi Fisik 1

Karena sering dipakai, *type* primitif yang namanya matrix sudah dianggap ada, seperti halnya *type* dasar array. Hanya saja kalau *type* array ditentukan oleh satu indeks, maka *type* matrix mampu menangani 2 indeks, yang diinterpretasikan oleh pemrogram seperti baris dan kolom.

Contoh (lihat gambar): Perhatikanlah "semantik" dari setiap pendefinisian berikut:

```
MatFrek :  matrix ['A'..'E', 1..7] of integer
```

Sebuah matriks yang merepresentasi frekuensi huruf 'A' s.d. 'E', untuk 7 buah teks.
Maka MatFrek(i,j) berarti frekuensi huruf ke-i untuk teks ke-j.

```
A : matrix [1..5, 1..5] of real
```
Sebuah matriks seperti dalam matematika biasa
```
NamaHari:  matrix [1..7, 1..3] of string
```
Untuk matriks nama hari pada contoh-1 yang merepresentasi nama-nama ke 7 (tujuh) buah hari ([1..7]) dalam 3 (tiga) bahasa ([1..3]). Maka, $NamaHari_{i,j}$ berarti Hari ke-i dalam bahasa ke-j.

Untuk matriks yang merepresentasi hasil survey pada setiap titik koordinat pengamatan. Koordinat yang diukur adalah (1,1) s.d. (4,7).dengan definisi:
```
type  Data : < Temp : integer, KecepAngin : integer>
MatSurvey  : matrix  [1..4,1..7] of Data
```
Oleh karena itu, MatSurvey(i,j) berarti hasil data pengukuran temperatur dan kecepatan angin pada grid kartesian (i,j).

Cara mengacu : melalui indeks

`MatHari(i,j)`	jika i dan j terdefinisi
`TabNamaHari(1,7)`	
`MatSurvey(3,5)`	untuk mengacu satu data survey
`MatSurvey(3,5).Temp`	untuk mengacu data temperatur

Implementasi Fisik 2: Struktur Fisik adalah Tabel dari Tabel (*Array of Array*)

Jika *type* dasar *matrix* tidak ada, maka matriks dibentuk dari *type* array. Maka matriks adalah array dari array.

```
MatFrek : array ['A'..'E'] of array [1..7] of integer
```
Sebuah matriks yang merepresentasi frekuensi huruf 'A' s.d. 'E', untuk 7 buah teks
Oleh karena itu, MatFrek(i,j) berarti Frekuensi huruf ke-i untuk teks ke-j ditulis sebagai:

MatFrek(i,j)

```
A : array  [1..5] of array [1..5] of integer
```
Sebuah matriks seperti dalam matematika ditulis sebagai :

A(i,j)

```
NamaHari : array [1..7] of array [1..3] of  string
```
Sebuah matriks yang merepresentasi nama-nama ke 7 hari ([1..7]) dalam tiga bahasa ([1..3]), maka NamaHari(i,j) berarti Hari ke-i dalam bahasa ke-j ditulis sebagai :

NamaHari (i,j)

```
type DataGeo : < Temp : integer, KecepAngin : integer >
MatSurvey : array [1..4] of array [1..7] of DataGeo
```
Sebuah matriks yang merepresentasi hasil survey pada setiap titik koordinat pengamatan. Koordinat yang diukur adalah (1, 1) s.d. (7, 4). Maka, MatSurvey(i,j) berarti hasil data pengukuran temperatur dan kecepatan angin pada grid kartesian (i,j) ditulis sebagai MatSurvey(i,j).
Sedangkan untuk mengacu kepada data kecepatan angin : MatSurvey(i,j).Temp.

Beberapa catatan mengenai matriks :

- Struktur matriks adalah struktur internal yang statis dan kontigu.
- Alokasi memori sebuah matriks berukuran N x M selalu dilakukan sekaligus. Dari ruang memori berukuran N x M tersebut, mungkin hanya "sebagian" yang dipakai. Karena itu ada pengertian :
  - Definisi ruang memori seluruh matriks.
  - Memori yang secara efektif dipakai oleh sebuah matriks tertentu.
- Nilai yang disimpan dalam sebuah matriks dapat disimpan di dalam ruang memori dipesan.
- Matriks dapat menimbulkan persoalan dalam passing parameter. Karena itu sebaiknya parameter aktual dan parameter formal sama ukuran memorinya.

Catatan implementasi dalam bahasa C :

- Bahasa C melakukan "linierisasi" elemen matriks per baris. Perhatikan cara penulisan "konstanta" matriks pada saat inisialisasi nilai matriks statis.
- Karena dalam bahasa C ada fasilitas untuk alokasi secara dinamis, maka ukuran matriks dapat ditentukan pada saat eksekusi. Jangan lupa melakukan alokasi.
- Walaupun dalam bahasa C ada fasilitas mengacu nilai elemen matriks lewat pointer, tidak disarankan menulis teks program dengan memakai pointer. Pakailah dua indeks i dan j yang menyatakan "nomor baris" dan "nomor kolom" sehingga arti program menjadi jelas.

Perluasan dari matriks dua "dimensi":

- Beberapa bahasa memungkinkan deklarasi variabel dengan lebih dari dua "dimensi" yaitu "indeks". Tidak disarankan untuk merancang struktur data internal dengan dimensi lebih dari 3!
- Pelajarilah fasilitas dari bahasa C untuk deklarasi, inisialisasi, dan memproses matriks berdimensi 3 atau lebih.

ADT MATRIKS dalam Bahasa Algoritmik

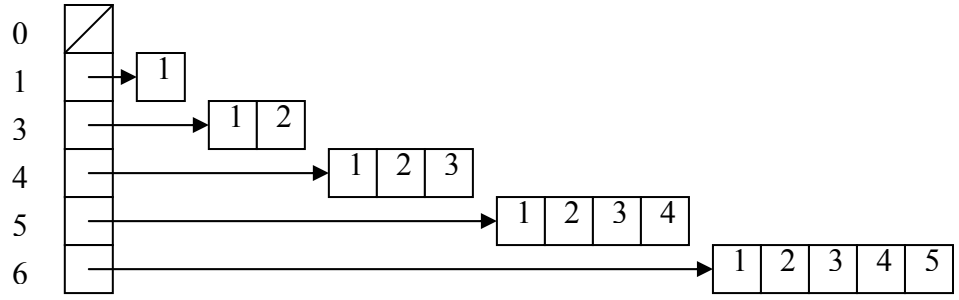
```
{ Definisi ABSTRACT DATA TYPE MATRIKS }
{ ********** HUBUNGAN DENGAN ADT LAIN ********** }
{ Tidak ada }
{ Alokasi elemen matriks selalu dilakukan sekaligus }

{ ********** Definisi TYPE MATRIKS dengan indeks integer ********** }
{ Ukuran minimum dan maksimum baris dan kolom }
type indeks : integer { indeks baris, kolom }
constant BrsMin : indeks = 1
constant BrsMax : indeks = 100
constant KolMin : indeks = 1
constant KolMax : indeks = 100
type el_type : integer
type MATRIKS :
      < Mem : matrix(BrsMin..BrsMax, KolMin..KolMax) of el_type,
        NbrsEff : integer,  { banyaknya/ukuran baris yg terdefinisi }
        NkolEff : integer   { banyaknya/ukuran kolom yg terdefinisi }
      >
{ Invarian ADT : Matriks "kosong" : NbrsEff=0 dan NkolEff=0}
{ NbrsEff ≥ 1 dan NkolEff ≥ 1 }
{ Memori matriks yang dipakai selalu di "ujung kiri atas" }

{ ********** DEFINISI PROTOTIPE PRIMITIF ********** }
{ *** Konstruktor membentuk MATRIKS *** }
procedure MakeMATRIKS[' (input NB, NK : integer)
{ Membentuk sebuah MATRIKS "kosong" berukuran NB x NK di "ujung kiri" memori }
{ I.S. NB dan NK adalah valid untuk memori matriks yang dibuat }
{ F.S. sebuah matriks sesuai dengan def di atas terbentuk }

{ *** Selektor "DUNIA MATRIKS" *** }
function GetIdxBrsMin → indeks
{ Mengirimkan indeks baris minimum matriks apapun }
function GetIdxKolMin → indeks
{ Mengirimkan indeks kolom minimum matriks apapun }
function GetIdxBrsMax → indeks
{ Mengirimkan indeks baris maksimum matriks apapun }
function GetIdxKolMax → indeks
{ Mengirimkan indeks kolom maksimum matriks apapun }
function IsIdxValid (i, j : indeks) → boolean
{ Mengirimkan true jika i, j adalah indeks yang valid }

{ *** Untuk sebuah matriks M yang terdefinisi: *** }
function FirstIdxBrs (M : MATRIKS) → indeks
{ Mengirimkan indeks baris terkecil M }
function FirstIdxKol (M : MATRIKS) → indeks
{ Mengirimkan indeks kolom terkecil M }
function LastIdxBrs (M : MATRIKS) → indeks
{ Mengirimkan indeks baris terbesar M }
function LastIdxKol (M : MATRIKS) → indeks
{ Mengirimkan indeks kolom terbesar M }
function GetNBrsEff (M : MATRIKS) → integer
{ Mengirimkan banyaknya baris efektif M }
function GetNKolEff →  integer
{ Mengirimkan banyaknya kolom efektif M }
function IsIdxEff (M : Matriks, i, j : indeks) → boolean
{ Mengirimkan true jika i, j adalah indeks efektif bagi M }
function GetElmt (M : MATRIKS, i, j : indeks) → el_type
{ Mengirimkan elemen M dg nomor baris i dan nomor kolom j }
function GetElmtDiagonal (M : MATRIKS, i : indeks) → el_type
{ Mengirimkan elemen M(i,i) }
```

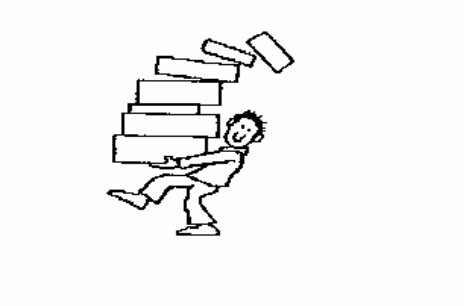
```
{ *** Operasi mengubah nilai elemen matriks: Set / Assign *** }
procedure SetBrsEff (input/output M : MATRIKS, input NB : integer)
{ I.S. M sudah terdefinisi }
{ F.S. Nilai M.BrsEff diisi dengan NB,  }
procedure SetKolEff (input/output M : MATRIKS, input NK : integer)
{ I.S. M sudah terdefinisi }
{ F.S. Nilai M.NKolEff diisi dengan NK }
procedure SetEl (input/output M : MATRIKS, input i, j : integer,
                 input X : el_type)
{ I.S. M sudah terdefinisi }
{ F.S. M(i,j) bernilai X }
{ Proses: Mengisi M(i,j) dengan X }

{ ********** Assignment  MATRIKS ********** }
procedure (input Min : MATRIKS, output MHsl : MATRIKS)
{ Melakukan assignment MHsl ← Min }

{ ********** KELOMPOK BACA/TULIS ********** }
procedure BacaMATRIKS (output M : MATRIKS, input NB, NK : integer)
{ I.S. IsIdxValid(NB,NK) }
{ F.S. M terdefinisi nilai elemen efektifnya, dan berukuran NB x NK }
{ Melakukan MakeMatriks(M,NB,NK) dan mengisi nilai efektifnya }
{ dari pembacaan dengan traversal per baris }
procedure TulisMATRIKS (input M : MATRIKS)
{ I.S. M terdefinisi }
{ F.S. Sama dengan I.S, dan nilai M(i,j) ditulis ke layar}
{ Menulis Nilai setiap indeks dan elemen M ke layar }
{ dengan traversal per baris }

{ ********** KELOMPOK OPERASI ARITMATIKA TERHADAP TYPE ********** }
function "+" (M1, M2 : MATRIKS) → MATRIKS
{ Prekondisi : M1  berukuran sama dengan M2 }
{ Mengirim hasil penjumlahan matriks: M1 + M2 }
function "-" (M1, M2 : MATRIKS) → MATRIKS
{ Prekondisi : M berukuran sama dengan M }
{ Mengirim hasil pengurangan matriks: salinan M1 - M2 }
function "*" (M1, M2 : MATRIKS) → MATRIKS
{ Prekondisi : Ukuran Baris efektif M = ukuran kolom efektif M }
{ Mengirim hasil perkalian matriks: salinan M1 * M2 }
function "*" (M : MATRIKS, X : integer) → MATRIKS
{ Mengirim hasil perkalian setiap elemen M dengan X }
procedure "*" (input/output M : MATRIKS, input K : integer)
{ Mengalikan setiap elemen M dengan K }

{ ********** KELOMPOK OPERASI RELASIONAL TERHADAP MATRIKS ********** }
function "="(M1, M2 : MATRIKS) → boolean
{ Mengirimkan true jika M1 = M2, }
{ yaitu NBElmt(M1) = NBElmt(M2) dan }
{ untuk setiap i,j yang merupakan indeks baris dan kolom }
{ M1(i,j) = M2(i,j) }
function StrongEQ (M1, M2 : MATRIKS) → boolean
{ Mengirimkan true jika M1 "strongly equal"  M2, }
{ yaitu FirstIdx(M1) = FirstIdx(M2) dan LastIdx(M1)=LastIdx(M2) dan }
{ untuk setiap i,j yang merupakan indeks baris dan kolom }
{ M1(i,j) = M2(i,j) }
function NEQ (M1, M2 : MATRIKS) → boolean
{ Mengirimkan true jika not strongEQ(M1,M2) }
function EQSize (M1, M2 : MATRIKS) →  boolean
{ Mengirimkan true jika ukuran efektif matriks M1 sama dengan }
{ ukuran efektif M2 }
{ yaitu GetBrsEff(M1) = GetNBrsEff (M2) }
{       dan GetNKolEff (M1) = GetNKolEff (M2) }
function "<"(M1,M2 : MATRIKS) → boolean
{ Mengirimkan true jika ukuran efektif M1 < Ukuran efektif M2 }
```

```
{ ********** Operasi lain ********** }
function NBElmt (M : Matriks) → integer
{ Mengirimkan banyaknya elemen M }

{ ********** KELOMPOK TEST TERHADAP MATRIKS ********** }
function IsBujurSangkar (M : Matriks) → boolean
{ Mengirimkan true jika M adalah matriks dg ukuran baris dan kolom sama }
function IsSymetri (M : Matriks)  → boolean
{ Mengirimkan true jika M adalah matriks simetri : IsBujurSangkar(M) dan untuk
setiap elemen M, M(i,j)=M(j,i) }
function IsSatuan (M : Matriks)   → boolean
{ Mengirimkan true jika M adalah matriks satuan: IsBujurSangkar(M) dan setiap
elemen diagonal M bernilai 1 dan elemen yang bukan diagonal bernilai 0 }
function IsSparse (M : Matriks)   → boolean
{ Mengirimkan true jika M adalah matriks sparse: mariks "jarang" dengan
definisi: hanya maksimal 5% dari memori matriks yang efektif bukan bernilai 0 }
function Invers1 (M : Matriks)  → MATRIKS
{ Menghasilkan salinan M dg setiap elemen "di-invers" }
{ yaitu dinegasikan }
function Invers (M : Matriks) → MATRIKS
{ Menghasilkan salinan M dg setiap elemen "di-invers" }
{ yaitu di-invers sesuai dengan aturan inversi matriks }
function Determinan (M : Matriks) → real
{ Menghitung nilai determinan M }
procedure traversalBrs (input M : MATRIKS)
{ Melakukan traversal terhadap M, per baris per kolom }
{ I.S. M terdefinisi }
{ F.S. setiap elemen M diproses dengan Proses P(el_type) yang terdefinisi }
procedure traversalKol (input M : MATRIKS)
{ Melakukan traversal terhadap M, per kolom per baris }
{ I.S. M terdefinisi }
{ F.S. setiap elemen M diproses dengan Proses P P(el_type) yang terdefinisi }
procedure Invers1(input/output M : MATRIKS)
{ I.S. M terdefinisi }
{ F.S. M di-invers, yaitu setiap elemennya dinegasikan }
procedure Invers(input/output M : MATRIKS)
{ I.S. M terdefinisi }
{ F.S. M "di-invers", yaitu diproses sesuai dengan aturan invers matriks }
procedure Transpose (input/output M : MATRIKS)
{ I.S. M terdefinisi dan IsBujursangkar(M) }
{ F.S. M "di-transpose", yaitu setiap elemen M(i,j) ditukar nilainya dengan
elemen M(j,i) }
```

Catatan implementasi dalam bahasa C:

1. Implementasi "matriks" dalam bahasa C harus memperhatikan alokasi elemen, dan dapat membentuk matriks yang "tidak persegi" (artinya jumlah elemen per baris tidak selalu sama).
2. Sebagai latihan, buatlah sebuah program kecil dalam bahasa C yang akan membereskan alokasi memori dan mengisi sebuah matriks segi tiga yang merupakan "array of array" sebagai berikut, dan kemudian mengisi setiap sel dengan angka sebagai berikut :





Stack (Tumpukan)

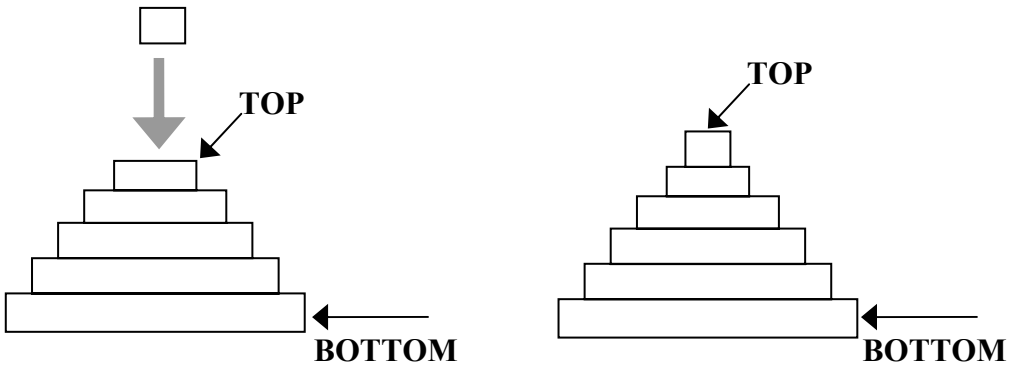
Definisi

STACK (tumpukan) adalah list linier yang:
1. dikenali elemen puncaknya (TOP)
2. aturan penyisipan dan penghapusan elemennya tertentu:
   Penyisipan selalu dilakukan "di atas" TOP
   Penghapusan selalu dilakukan pada TOP

Karena aturan penyisipan dan penghapusan semacam itu, TOP adalah satu-satunya alamat tempat terjadi operasi, elemen yang ditambahkan paling akhir akan menjadi elemen yang akan dihapus Dikatakan bahwa elemen Stack akan tersusun secara LIFO *(Last In First Out)*.

Struktur data ini banyak dipakai dalam informatika, misalnya untuk merepresentasi :
- pemanggilan prosedur,
- perhitungan ekspresi artimatika,
- rekursifitas,
- *backtracking*,

dan algoritma lanjut yang lain.



Perhatikan bahwa dengan definisi semacam ini, representasi tabel sangat tepat untuk mewakili Stack, karena operasi penambahan dan pengurangan hanya dilakukan di salah satu “ujung” tabel. “Perilaku” yang diubah adalah bahwa dalam Stack, operasi penambahan hanya dapat dilakukan di salah satu ujung. Sedangkan pada tabel, boleh di mana pun.

Definisi Fungsional

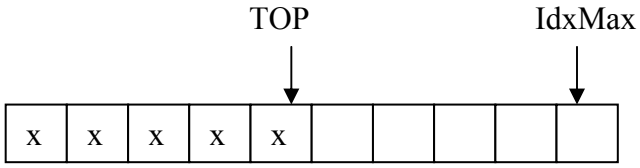
Jika diberikan S adalah Stack dengan elemen ElmtS, maka definisi fungsional stack adalah:

```
CreateEmpty : → S              { Membuat sebuah stack kosong}
IsEmpty     : S → boolean      { Test stack kosong, true jika stack kosong,
                                 false jika S tidak kosong }
IsFull      : S → boolean      { Test stack penuh, true jika stack penuh,
                                 false jika S tidak }
Push        : ElmtS x S → S    { Menambahkan sebuah elemen ElmtS sebagai
                                 TOP, TOP berubah nilainya }
Pop         : S → S x ElmtS    { Mengambil nilai elemen TOP,
                                 sehingga TOP yang baru adalah elemen yang
                                 datang sebelum elemen TOP,
                                 mungkin S menjadi kosong }
```

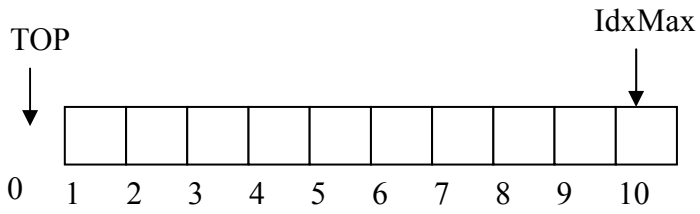
Definisi selektor adalah:
Jika S adalah sebuah Stack, maka
Top(S) adalah alamat elemen TOP,
di mana operasi penyisipan/penghapusan dilakukan,
InfoTop(S) adalah informasi yang disimpan pada Top(S).
Definisi Stack kosong adalah Stack dengan Top(S)=Nil (tidak terdefinisi).

Implementasi Stack dengan Tabel

Tabel dengan hanya representasi TOP adalah indeks elemen Top dari Stack. Jika Stack kosong, maka TOP=0.
Ilustrasi Stack tidak kosong, dengan 5 elemen :



Ilustrasi Stack kosong:



```
/* File : stackt.h */
/* Deklarasi stack yang diimplementasi dengan tabel kontigu */
/* dan ukuran sama */
/* TOP adalah alamat elemen puncak */
/* Implementasi Stack dalam bahasa C dengan alokasi statik */
#ifndef stackt_H
#define stackt_H
#include "boolean.h"
#define Nil 0
#define MaxEl 10
/* Nil adalah stack dengan elemen kosong */
/* Karena indeks dalam bhs C dimulai 0 maka tabel dg indeks 0 tidak */
/* dipakai */
typedef int infotype;
typedef int address;   /* indeks tabel */
/* Contoh deklarasi variabel bertype stack dengan ciri TOP : */
/* Versi I : dengan menyimpan tabel dan alamat top secara eksplisit*/
typedef struct { infotype T[MaxEl+1]; /* tabel penyimpan elemen */
                 address TOP;         /* alamat TOP: elemen puncak */
               } Stack;
   /* Definisi stack S kosong : S.TOP = Nil */
   /* Elemen yang dipakai menyimpan nilai Stack T[1]..T[MaxEl] */
/* Jika S adalah Stack maka akses elemen : */
   /* S.T[(S.TOP)] untuk mengakses elemen TOP */
   /* S.TOP adalah alamat elemen TOP */
/* Definisi akses dengan Selektor : Set dan Get */
#define Top(S) (S).TOP
#define InfoTop(S) (S).T[(S).TOP]
/*** Perubahan nilai komponen struktur ***/
/*** Untuk bahasa C tidak perlu direalisasi ***/
/********** Prototype **********/
/*** Konstruktor/Kreator ***/
void CreateEmpty(Stack *S);
/* I.S. Sembarang */
/* F.S. Membuat sebuah stack S yang kosong berkapasitas MaxEl */
/* jadi indeksnya antara 1.. MaxEl+1 karena 0 tidak dipakai */
/* Ciri stack kosong : TOP bernilai Nil */
/********** Predikat Untuk test keadaan KOLEKSI **********/
boolean IsEmpty (Stack S);
/* Mengirim true jika Stack kosong: lihat definisi di atas */
boolean IsFull(Stack S);
/* Mengirim true jika tabel penampung nilai elemen stack penuh */
/*********** Menambahkan sebuah elemen ke Stack **********/
void Push (Stack *S, infotype X);
/* Menambahkan X sebagai elemen Stack S. */
/* I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh */
/* F.S. X menjadi TOP yang baru,TOP bertambah 1 */
/*********** Menghapus sebuah elemen Stack **********/
void Pop (Stack *S, infotype *X);
/* Menghapus X dari Stack S. */
/* I.S. S  tidak mungkin kosong */
/* F.S. X adalah nilai elemen TOP yang lama, TOP berkurang 1 */
#endif
```

```
/* File : stack.h */
/* deklarasi stack yang diimplementasi dengan tabel kontigu */
/* TOP adalah alamat elemen puncak; */
/* Implementasi Stack dalam bahasa C dengan alokasi dinamik */
#ifndef stack_H
#define stack_H
#include "boolean.h"
#define Nil 0
/* Indeks dalam bhs C dimulai 0, tetapi indeks 0 tidak  dipakai */
typedef int infotype; /* type elemen stack */
typedef int address;   /* indeks tabel */
/* Contoh deklarasi variabel bertypetype stack dengan ciri TOP : */
/* Versi I : dengan menyimpan tabel dan alamat top secara eksplisit*/
/* Tabel dialokasi secara dinamik */
typedef struct { infotype *T;  /* tabel penyimpan elemen */
                 address TOP;  /* alamat TOP: elemen puncak */
                 int Size;     /* Ukuran stack */
               } Stack;
   /* Definisi stack S kosong : S.TOP = Nil */
   /* Elemen yang dipakai menyimpan nilai Stack T[1]..T[Size+1] */
   /* Perhatikan definisi ini, dan pakailah untuk mengalokasi T */
   /* dengan benar */
/* Elemen yang dipakai menyimpan nilai Stack T[1]..T[Size+1] */
/* Jika S adalah Stack maka akses elemen : */
   /* S.T[(S.TOP)] untuk mengakses elemen TOP */
   /* S.TOP adalah alamat elemen TOP */
/* Definisi akses : Get dan Set */
#define Top(S) (S).TOP
#define InfoTop(S) (S).T[(S).TOP]
#define Size(S) (S).Size
/* Perubahan nilai komponen struktur karena implementasi */
/* dengan macro spt di atas ; tidak perlu direalisasi */
/********** Prototype **********/
/*** Konstruktor/Kreator ***/
void CreateEmpty(Stack *S, int Size);
/* I.S. sembarang; */
/* F.S. Membuat sebuah stack S yang kosong berkapasitas Size */
/* jadi indeksnya antara 1.. Size+1 karena 0 tidak dipakai */
/* Ciri stack kosong : TOP bernilai Nil */
/* Destruktor */
void Destruct(Stack *S);
/* Destruktor: dealokasi seluruh tabel memori sekaligus */
/*********** Predikat Untuk test keadaan KOLEKSI **********/
boolean IsEmpty (Stack S);
/* Mengirim true jika Stack kosong: lihat definisi di atas */
boolean IsFull(Stack S);
/* Mengirim true jika tabel penampung nilai elemen stack penuh */
/*********** Menambahkan sebuah elemen ke Stack **********/
void Push (Stack *S, infotype X);
/* Menambahkan X sebagai elemen Stack S. */
/* I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh */
/* F.S. X menjadi TOP yang baru,TOP bertambah 1 */
/*********** Menghapus sebuah elemen Stack **********/
void Pop (Stack *S, infotype *X);
/* Menghapus X dari Stack S. */
/* I.S. S  tidak mungkin kosong */
/* F.S. X adalah nilai elemen TOP yang lama, TOP berkurang 1 */
#endif
```

```
-- File : stack.ads
-- Deklarasi stack yang diimplementasi dengan tabel kontigu
-- dan ukuran sama, TOP adalah alamat elemen puncak
-- Implementasi dalam bahasa Ada
package pSTACK is
   Nil : constant := 0;
   MaxEl : constant := 10;
-- Nil adalah stack dengan elemen kosong .
-- Contoh deklarasi variabel bertypetype stack dengan ciri TOP :
-- Versi I : dengan menyimpan tabel dan alamat top secara eksplisit
type TabMem is array (1..MaxEl) of integer;
type Stack is record
                T : TabMem;    -- tabel penyimpan elemen
                TOP: integer;  -- alamat TOP: elemen puncak
           End record;
   --  Definisi S: Stack kosong : S.TOP = Nil
   --  Elemen yang dipakai menyimpan nilai Stack T(1)..T(MaxEl)
--  Jika S adalah Stack maka akses elemen :
   --  S.T(S.TOP) untuk mengakses elemen TOP
   --  S.TOP adalah alamat elemen TOP
-- Selektor
function GetTop (S : Stack) return integer;
function GetInfoTop (S : STack) return integer;
---------- Prototype/Spesifikasi Primitif ----------
--- Predikat untuk test keadaan koleksi ---
function IsEmpty (S : Stack) return boolean;
-- Mengirim true jika Stack kosong: lihat definisi di atas
function IsFull (S : Stack) return boolean;
-- Mengirim true jika tabel penampung nilai elemen S: Stack sudah penuh
---------- Kreator ----------
procedure CreateEmpty (S : out Stack);
-- I.S. sembarang;
-- F.S. Membuat sebuah S: Stack yang kosong berkapasitas MaxEl
-- jadi indeksnya antara 1.. MaxEl+1 karena 0 tidak dipakai
-- Ciri stack kosong : TOP bernilai Nil
---------- Primitif Add/Delete ----------
procedure Push (S : in out Stack; X : in integer);
-- Menambahkan X sebagai elemen S
-- I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh
-- F.S. X menjadi TOP yang baru,TOP bertambah 1
procedure Pop (S : in out Stack; X : out integer);
-- Menghapus X dari S: Stack.
-- I.S. S  tidak mungkin kosong
-- F.S. X adalah nilai elemen TOP yang lama, TOP berkurang satu
PRIVATE
-- Definisi mengubah nilai Top dan Infotop
procedure SetTop (S : in out Stack; NewTop : in integer);
-- I.S. S terdefinisi
-- Nilai Top yang baru menjadi NewTop
procedure SetInfoTop (S : in out Stack; NewInfoTop : in integer);
-- I.S. S terdefinisi
-- Nilai InfoTop yang baru menjadi NewInfoTop
end pstack;
```

Latihan: Buatlah sebuah paket stack dengan elemen generik!

Contoh Aplikasi Stack

Evaluasi ekspresi matematika yang ditulis dengan notasi POLISH (Posfix)
Diberikan sebuah ekspresi aritmatika postfixdenganoperator ['.',':','+','-','^'], dan Operator mempunyai prioritas (prioritas makin besar, artinya makin tinggi) sebagai berikut :

OPERATOR	ARTI	PRIORITAS
^	pangkat	3
* /	kali, bagi	2
+ -	tambah, kurang	1

Contoh ekspresi: Arti
AB.C/ (A.B)/C
ABC^/DE*+AC*- (A/(B^C) + (D*E)) - (A*C)
Didefinisikan token adalah satuan "kata" yang mewakili sebuah operan (konstanta atau nama) atau sebuah operator. Berikut ini adalah algoritma untuk melakukan evaluasi ekspresi aritmatika yang pasti benar dalam notasi postfix.

```
Program EKSPRESI
{ Menghitung sebuah ekspresi aritmatika yang ditulis secara postfix }

USE STACK { memakai paket stack yang sudah terdefinisi, elemennya adalah TOKEN }
```

```
KAMUS
   type token : ... {terdefinisi }
   S : STACK  {stack of token }
   CT,Op1,Op2: token
   { Sebuah token yaitu kesatuan berupa operand atau operator }
   { Berikut ini adalah fungsi-fungsi yang didefinisikan di tempat lain }
   procedure First-Token { Mengirim token yang pertama }
   procedure Next-Token  { Mengirim token yang berikutnya }
   function EndToken → boolean
   { true jika proses akuisisi mendapat hasil sebuah token kosong
     Merupakan akhir ekspresi }
   function Operator (CT : token) → boolean
   { true jika CT adalah operator }
   function Hitung (OP1, OP2, Operator : token) → token
   { menghitung ekspresi, mengkonversi hasil menjadi token }
```

```
ALGORITMA
   First-Token
   if (EndToken) then
      output ("Ekspresi kosong")
   else
      repeat
         depend on (CT)      { CT adalah Current Token }
            not Operator (CT) : Push(S,CT)
            Operator(CT)      : { berarti operator }
                                Pop(S, Op2)
                                Pop(S, Op1)
                                Push(S,Hitung(OP1,OP2,CT))
         Next-Token (CT)
      until (EndToken)
      output (InfoTop(S)) { Tuliskan hasil }
```

Sebagai latihan:
Tuliskan algoritma untuk melakukan konversi sebuah ekspresi infix menjadi postfix.

Stack dengan Representasi Berkait dalam Bahasa C

Contoh berikut memberikan gambaran pemanfaatan *macro* dalam bahasa C untuk ”menggabung“ (minimalisasi) kode *source code* jika Stack direpresentasi secara ”lojik“ secara berkait, namun diimplementasi secara ”fisik“ dengan tabel berkait atau pointer. Akan dibahas setelah pembahasan list berkait.

```
/* File : linkstack.h */
/* Deklarasi : */
/* Representasi Lojik Stack : stack dikenali TOP-nya */
/* Type yang merepresentasi info: integer */
/*                           stack: pointer atau tabel berkait */
/* Prototype representasi type dan primitif menajemen memori */
/* Prototype primitif operasi stack */
#ifndef _LINKSTACK_H
#define _LINKSTACK_H
#include "boolean.h"
#include <stdlib.h>
/* Representasi lojik stack */
/* Nil adalah stack dengan elemen kosong  */
/* Definisi stack dengan representasi berkait */
/* Jika S adalah Stack maka akses elemen : */
   /* InfoTop(S) untuk mengakses elemen TOP */
   /* TOP(S) adalah alamat elemen TOP */
   /* Info (P) untuk mengakses elemen dengan alamat P */

#define Top(S) (S).TOP
/* Definisi stack S kosong : TOP(S) = Nil */

/* Deklarasi infotype */
typedef int infotype;

#define _POINTER_

#ifdef _POINTER_
/* Stack berkait dengan representasi pointer */
typedef struct tElmtStack * address;

/* Selektor, akses TOP dan info pada top */
#define Nil NULL
#define InfoTop(S) (S).TOP->Info
#define Next(P) (P)->Next
#define Info(P) (P)->Info

#else
/* else _POINTER_ */
#define Nil 0
#define MaxIdx 100
/* Definisi selektor */
#define Info(P) TabMem[(P)].Info
#define Next(P) TabMem[(P)].Next
#define InfoTop(S) TabMem[((S).TOP)].Info
typedef int address;

#endif
/* endif representasi _POINTER_ */

/* Deklarasi bersama */
/* Definisi type elemen stack */
```

```
typedef struct tElmtStack { infotype Info;
                            address Next;
                          } ElmtStack;

#ifdef _POINTER_
#else
extern ElmtStack TabMem[MaxIdx+1];
#endif

/* Contoh deklarasi variabel bertype stack dengan ciri TOP : */
typedef struct { address TOP;  /* alamat TOP: elemen puncak */
               } Stack;

/*****************************************************************/
/* Deklarasi manajemen memori stack */

/* Prototype manajemen memori */
void Inisialisasi();
/* I.S. sembarang */
/* F.S. memori untuk linked stack siap dipakai */
boolean IsFull();
/* Mengirim true jika tabel penampung nilai elemen stack sudah penuh */
void Alokasi (address *P, infotype X);
/* I.S. sembarang */
/* F.S. Alamat P dialokasi, jika berhasil maka Info(P)=X dan
        Next(P)=Nil */
/*      P=Nil jika alokasi gagal */
void Dealokasi (address  P);
/* I.S. P adalah hasil alokasi, P <> Nil */
/* F.S. Alamat P didealokasi, dikembalikan ke sistem */

/* ********* PROTOTYPE REPRESENTASI LOJIK STACK ***************/
boolean IsEmpty (Stack S);
/* Mengirim true jika Stack kosong: lihat definisi di atas */
void CreateEmpty(Stack *S);
/* I.S. sembarang; F.S. Membuat sebuah stack S yang kosong */
void Push (Stack * S, infotype X);
/* Menambahkan X sebagai elemen Stack S. */
/* I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh */
/*      X menjadi TOP yang baru*/
void Pop (Stack * S, infotype* X);
/* Menghapus X dari Stack S. */
/* S tidak mungkin kosong */
/* X adalah nilai elemen TOP yang lama */

#endif
```

```
/* File linkstack.c.c */

#include "linkstack.h"
#ifdef _POINTER_

/*****************************************************************/
/* Implementasi stack dengan representasi pointer */

void Inisialisasi()
/* I.S. sembarang */
/* F.S. memori untuk linked stack siap dipakai */
{ printf ("Representasi pointer \n");
  /* tidak ada sebab POINTER*/
}
boolean IsFull()
/* Mengirim true jika tabel penampung nilai elemen stack sudah penuh */
{  /* Kamus lokal */
   address P;
   /* Algoritma */
   P = (address) malloc (sizeof(ElmtStack));
   if (P == Nil) {
        return true;
   } else {
        free(P);
        return false;
   }
}
void Alokasi(address *P, infotype X)
/* I.S. sembarang */
/* F.S. Alamat P dialokasi, jika berhasil maka Info(P)=X dan
        Next(P)=Nil */
{  /* Algoritma */
   *P = (address) malloc (sizeof(ElmtStack));
   if ((*P) != Nil) {
        Info(*P) = X;
        Next(*P) = Nil;
   }
}

void Dealokasi(address P)
/* I.S. P adalah hasil alokasi, P <> Nil */
/* F.S. Alamat P didealokasi, dikembalikan ke sistem */
{
    free(P);
}

#else
/* else _POINTER_ */

/*****************************************************************/
/* Implementasi stack dengan representasi berkait */

/* deklarasi tabel memori */
ElmtStack TabMem[MaxIdx+1];
address FirstAvail;

/*** Realisasi manajemen memori tabel berkait ***/
void Inisialisasi ()
/* I.S. sembarang */
/* F.S. memori untuk linked stack siap dipakai */
```

```
{  /* Kamus lokal */
   address P;
   /* algoritma */
   printf ("representasi tabel berkait \n");
   for (P=1; P< MaxIdx; P++) { Next(P)=P+1; }
   Next(MaxIdx)=Nil;
   FirstAvail= 1;
}
boolean IsFull()
/* Mengirim true jika tabel penampung nilai elemen stack sudah penuh */
{
   return FirstAvail==Nil;
}
void Alokasi (address * P, infotype X)
/* I.S. sembarang */
/* F.S. Alamat P dialokasi, jika berhasil maka Info(P)=X dan
        Next(P)=Nil */
/*      P=Nil jika alokasi gagal */
{  /* Algoritma */
   *P = FirstAvail;
   if (*P !=Nil) {
      Info(*P) = X;
      Next(*P)=Nil;
      FirstAvail = Next(FirstAvail);
   }
}
void Dealokasi (address P)
/* I.S. P adalah hasil alokasi, P <> Nil */
/* F.S. Alamat P didealokasi, dikembalikan ke sistem */
{  /* Algoritma */
   Next(P) = FirstAvail;
   FirstAvail = P;
}

#endif
/* endif _POINTER_ */

/********************** BODY PRIMITIF STACK*******************/

boolean IsEmpty (Stack S)
/* Mengirim true jika Stack kosong: lihat definisi di atas */
{
   return (Top(S) == Nil);
}
void CreateEmpty(Stack * S )
{
   Top(*S) = Nil;
}
void Push(Stack * S, infotype X)
/* Menambahkan X sebagai elemen Stack S. */
/* I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh */
/* X menjadi TOP yang baru,TOP bertambah 1 */
{  /* Kamus lokal */
   address P;
   /* Algoritma */
   Alokasi(&P,X);
   if (P!=Nil) {
        Next(P) = Top(*S);
        Top(*S) = P; }
   else { printf ("alokasi gagal \n"); }
}
```

```
void Pop (Stack *S, infotype *X)
/* Menghapus X dari Stack S. */
/* S tidak mungkin kosong */
/* X adalah nilai elemen TOP yang lama, TOP berkurang 1 */
{  /* Kamus lokal */
   address PDel ;
   /* Algoritma */
   *X = InfoTop(*S);
   PDel = Top(*S);
   Top(*S) = Next(Top(*S));
   Dealokasi(PDel);
}
```

```
/* Nama File : mstack.c */
/* Driver untuk mengetes stack */
#include "linkstack.h"
int main()
{ /* KAMUS */
   Stack S;
   infotype X;
  /* Algoritmma */
   Inisialisasi();
   CreateEmpty(&S);
   Push(&S, 1);
   printf("%d   %d   \n", Top(S), InfoTop(S));
   Push(&S, 0);
   printf("%d   %d   \n", Top(S), InfoTop(S));
   Pop(&S, &X);
   printf("%d   %d   \n", Top(S), InfoTop(S));
   return 0;
}
```

Satu Tabel dengan Dua Buah Stack

```
/* File : stack2.h */
/* Persoalan : sebuah tabel dipakai bersama oleh dua buah stack */
/* "arah" perkembangan kedua stack "berlawanan" */
/* TOP adalah alamat elemen puncak */
/* T adalah memori penyimpan elemen stack S1 dan S2 */
/* T, S1 dan S2 adalah variabel global di stack2.c */
#ifndef stack2_H
#define stack2_H
#include "boolean.h"
#define Nil 0
/* Nil adalah stack dengan elemen kosong . */
/* Indeks dalam bhs C dimulai 0, tetapi indeks 0 tidak dipakai */
typedef int infotype;
typedef int address;   /* indeks tabel */
/* Contoh deklarasi variabel bertype stack dengan ciri TOP : */
/* Versi I : dengan menyimpan tabel dan alamat top secara eksplisit*/
/* Tabel dialokasi secara dinamik */
typedef struct { address TOP;  /* alamat TOP: elemen puncak */
               } Stack;
   /* Definisi stack S kosong : S.TOP = Nil */
/* Elemen yang dipakai menyimpan nilai Stack T[1]..T[memSize+1] */
/* Jika S adalah Stack maka akses elemen : */
   /* T[(S.TOP)] untuk mengakses elemen TOP */
   /* S.TOP adalah alamat elemen TOP */
/* Definisi akses */
#define TopS1 S1.TOP
#define InfoTopS1 T[S1.TOP]
#define TopS2 S2.TOP
#define InfoTopS2 T[S2.TOP]
/*** Perubahan nilai komponen struktur  ***/
/*** Untuk bahasa C tidak perlu direalisasi ***/

/********** Prototype **********/
/*** manajemen memori ***/
void InitMem(int Totalsize);
/* melakukan inisialisasi memori, satu kali sebelum dipakai */
void Destruct();
/* destruktor: dealokasi seluruh tabel memori sekaligus  */
boolean IsFull();
/* Mengirim true jika tabel penampung nilai elmt stack sudah penuh */

/*** Konstruktor/Kreator ***/
void CreateEmptyS1();
/* I.S. sembarang */
/* F.S. Membuat sebuah stack S1 yang kosong */
void CreateEmptyS2();
/* I.S. sembarang */
/* F.S. Membuat sebuah stack S2 yang kosong */

/* Ciri stack kosong : TOP bernilai Nil */
/********** Predikat untuk tes keadaan KOLEKSI **********/
boolean IsEmptyS1();
/* Mengirim true jika Stack kosong: lihat definisi di atas */
boolean IsEmptyS2();
/* Mengirim true jika Stack kosong: lihat definisi di atas */

/********** Menambahkan sebuah elemen ke Stack **********/
void PushS1 (infotype X);
/* Menambahkan X sebagai elemen Stack S1 */
/* I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh */
/* F.S. X menjadi TOP yang baru,TOP bertambah 1 */
void PushS2 (infotype X);
/* Menambahkan X sebagai elemen Stack S2. */
/* I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh */
/* F.S. X menjadi TOP yang baru,TOP bertambah 1 */
```

```
/********** Menghapus sebuah elemen Stack **********/
void PopS1(infotype *X);
/* Menghapus X dari Stack S1 */
/* I.S. S1  tidak mungkin kosong */
/* F.S. X adalah nilai elemen TOP yang lama, TOP berkurang 1 */
void PopS2(infotype *X);
/* Menghapus X dari Stack S2 */
/* I.S. S1  tidak mungkin kosong */
/* F.S. X adalah nilai elemen TOP yang lama, TOP bertambah 1 */

#endif
```

```
/* File : mstack2.c */
/* driver untuk pengelolaan dua buah stack */
#include "stack2.h"
/* variabel global di file lain */
extern Stack S1, S2;
extern int* T;
/******************** Program Utama ************/
int main ()
{
  /* Kamus */
  int X;

  /* Algoritma */
  /* Harus inisialisasi memori */
  InitMem (10);
  CreateEmptyS1 ();
  CreateEmptyS2 ();
  while (!IsFull ())
  {
     PushS1 (10);
  }
  printf ("Infotop = %d \n", InfoTopS1);

  printf ("stack penuh ..\n");
  while (!IsEmptyS1 ())
  {
     PopS1 (&X);
  }
}
```

Pembahasan:

1. Perhatikanlah kode di atas. Apakah stack dapat diparametrisasi, dan bukan ditulis menjadi dua prosedur?
2. Apakah tabel memori dapat dijadikan parameter (bukan variabel global)?

Stack Generik dalam Bahasa Ada

```
-- File : pstackgen.ads
-- Deklarasi stack  generik yang diimplementasi dengan tabel kontigu
-- dan ukuran sama
-- TOP adalah alamat elemen puncak
GENERIC
  type t_ELEMEN is private;

package pSTACKgen is
  Nil: constant:= 0;
  MaxEl : constant :=10;
-- Nil adalah stack dengan elemen kosong .
-- Contoh deklarasi variabel bertype stack dengan ciri TOP :
-- Versi I : dengan menyimpan tabel dan alamat top secara eksplisit
type TabMem is array (0..MaxEl) of t_elemen;
type Stack is record
              T : TabMem; --  tabel penyimpan elemen
              TOP: integer ;  --  alamat TOP: elemen puncak
            End record;
   -- Definisi S: Stack kosong : S.TOP = Nil
   -- Elemen yang dipakai menyimpan nilai Stack T(1)..T(MaxEl)
-- Jika S adalah Stack maka akses elemen :
   -- S.T(S.TOP) untuk mengakses elemen TOP
   -- S.TOP adalah alamat elemen TOP
-- Selektor
function Top(S : Stack) return integer;
function InfoTop(S : STack) return t_elemen;
--------------  Prototype/Spesifikasi Primitif -------------
----------- Predikat untuk tes keadaan koleksi -----------
function IsEmpty (S: Stack) return boolean;
--  Mengirim true jika Stack kosong: lihat definisi di atas
function  IsFull(S: Stack) return boolean;
--  Mengirim true jika tabel penampung nilai elemen S: sudah penuh
----------- Kreator -----------
procedure CreateEmpty(S : out Stack);
--  I.S. sembarang;
--  F.S. Membuat sebuah S Stack yang kosong berkapasitas MaxEl
--  jadi indeksnya antara 1..MaxEl+1 karena 0 tidak dipakai
--  Ciri stack kosong : TOP bernilai Nil
----------- Primitif Add/Delete -----------
procedure Push (S : in out Stack; X : in t_elemen);
--  Menambahkan X sebagai elemen S: Stack.
--  I.S. S mungkin kosong, tabel penampung elemen stack TIDAK penuh
--  F.S. X menjadi TOP yang baru, TOP bertambah 1
procedure Pop (S : in out Stack; X : out t_elemen);
--  Menghapus X dari S Stack.
--  I.S. S  tidak mungkin kosong
--  F.S. X adalah nilai elemen TOP yang lama, TOP berkurang 1
private
-- Definisi mengubah nilai Top dan Infotop
procedure AssignTop(S : in out Stack; NewTop : integer);
-- I.S. S terdefinisi
-- F.S. Nilai Top yang baru menjadi NewTop
procedure AssignInfoTop(S : in out Stack; NewInfoTop : in t_elemen);
-- I.S. S terdefinisi
-- F.S. Nilai InfoTop yang baru menjadi NewInfoTop
end pstackgen;
```

```
-- File mstackgen.adb
-- driver untuk test pstackgen.adb

-- Konteks
with pstackgen;
with text_io; use text_io;
procedure mstackgen is
-- instansiasi type address
-- type myaddress is new integer range 0..10;
-- paket yang diturunkan
package stackint is new pstackgen (integer); use stackint;
package stackchar is new pstackgen (character); use stackchar;
package stackfloat is new pstackgen (float); use stackfloat;

-- Kamus
   p: integer;
   f : float;
   c: character;
   S: stackint.Stack;
   Sf : stackfloat.stack;
   Sc : stackchar.Stack;

-- Algoritma
begin
   CreateEMpty(S); CreateEmpty (Sf); CreateEmpty (Sc);

   Push(S,1);
   put_line (integer'image(infotop(S)));
   Push(S,2);
   put_line (integer'image(infotop(S)));
   Pop(S,p);
   put_line (integer'image(infotop(S)));

   while (not IsFull(S)) loop
      Push(S,5);
   end loop;
   while (not IsEmpty(S)) loop
      Pop(S,P);
   end loop;

   while (not IsFull(Sf)) loop
      Push(Sf,5.0);
   end loop;
   while (not IsEmpty(Sf)) loop
      Pop(Sf,f);
   end loop;

   while (not IsFull(Sc)) loop
      Push(Sc,'A');
   end loop;
   while (not IsEmpty(Sc)) loop
      Pop(Sc,c);
   end loop;

end mstackgen;
```

Queue (Antrian)

Definisi

QUEUE (Antrian) adalah list linier yang :

1. dikenali elemen pertama (HEAD) dan elemen terakhirnya (TAIL),
2. aturan penyisipan dan penghapusan elemennya didefinisikan sebagai berikut:
   - Penyisipan selalu dilakukan setelah elemen terakhir
   - Penghapusan selalu dilakukan pada elemen pertama
3. satu elemen dengan yang lain dapat diakses melalui informasi NEXT

Struktur data ini banyak dipakai dalam informatika, misalnya untuk merepresentasi :

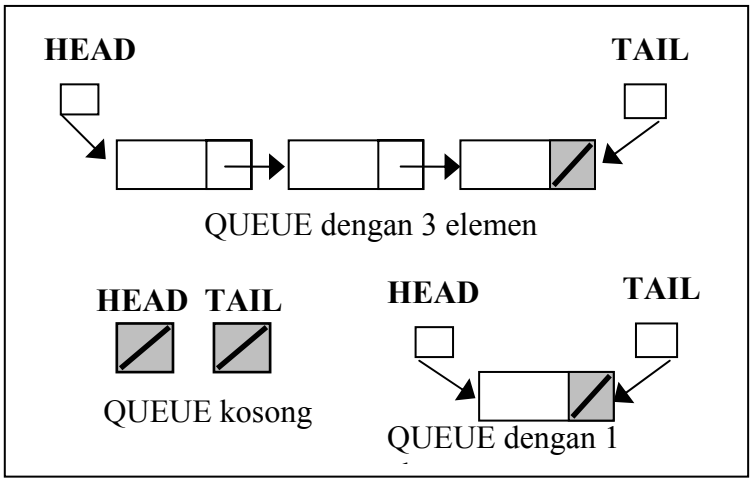
- antrian job yang harus ditangani oleh sistem operasi
- antrian dalam dunia nyata.

Maka secara lojik, sebuah QUEUE dapat digambarkan sebagai list linier yang setiap elemennya adalah:

type ElmtQ : < Info : InfoType, Next :address >

dengan InfoType adalah sebuah *type* terdefinisi yang menentukan informasi yang disimpan pada setiap elemen queue, dan address adalah "alamat" dari elemen
Selain itu alamat elemen pertama (**HEAD**) dan elemen terakhir(**TAIL**) dicatat :
Maka jika Q adalah Queue dan P adalah adaress, penulisan untuk Queue adalah :

Head(Q),Tail(Q), Info(Head(Q)), Info(Tail(Q))



Definisi Fungsional

Jika diberikan Q adalah QUEUE dengan elemen ElmtQ, maka definisi fungsional antrian adalah:

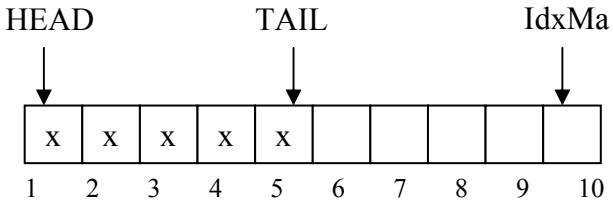
```
IsEmpty      : Q → boolean    { Tes terhadap Q: true jika Q kosong,
                                false jika Q tidak kosong }
IsFull       : Q → boolean    { Tes terhadap Q: true jika memori Q sudah
                                penuh, false jika memori Q tidak penuh }
NBElmt(Q)    : Q → integer    { Mengirimkan banyaknya elemen Q }
CreateEmpty  : → Q            { Membuat sebuah antrian kosong }
Add          : ElmtQ x Q → Q  { Menambahkankan sebuah elemen setelah
                                elemen  ekor QUEUE }
Del          : Q → Q x ElmtQ  { Menghapus kepala QUEUE,
                                mungkin Q menjadi kosong }
```

Implementasi QUEUE dengan Tabel

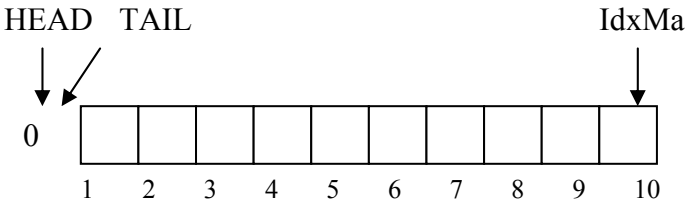
Memori tempat penyimpan elemen adalah sebuah tabel dengan indeks 1.. IdxMax. IdxMax dapat juga "dipetakan" ke kapasitas Queue. Representasi field Next: Jika i adalah "address" sebuah elemen, maka suksesor i adalah Next dari elemen Queue.

Alternatif I

Tabel dengan hanya representasi TAIL adalah indeks elemen terakhir, HEAD selalu diset sama dengan 1 jika Queue tidak kosong. Jika Queue kosong, maka HEAD=0.
Ilustrasi Queue tidak kosong, dengan 5 elemen:



Ilustrasi Queue kosong:

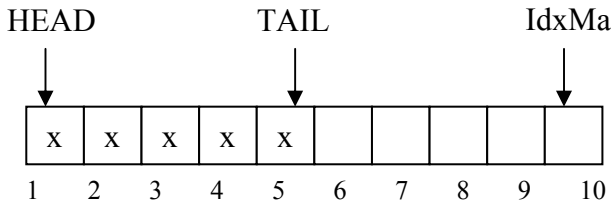


Algoritma paling sederhana untuk **penambahan elemen** jika masih ada tempat adalah dengan "memajukan" TAIL. Kasus khusus untuk Queue kosong karena HEAD harus diset nilainya menjadi 1.

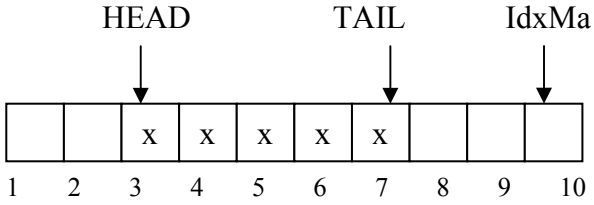
Algoritma paling sederhana dan "naif" untuk **penghapusan elemen** jika Queue tidak kosong: ambil nilai elemen HEAD, geser semua elemen mulai dari HEAD+1 s.d. TAIL (jika ada), kemudian TAIL "mundur". Kasus khusus untuk Queue dengan keadaan awal berelemen 1, yaitu menyesuaikan HEAD dan TAIL dengan DEFINISI. Algoritma ini mencerminkan pergeseran orang yang sedang mengantri di dunia nyata, tapi tidak efisien.

Alternatif II

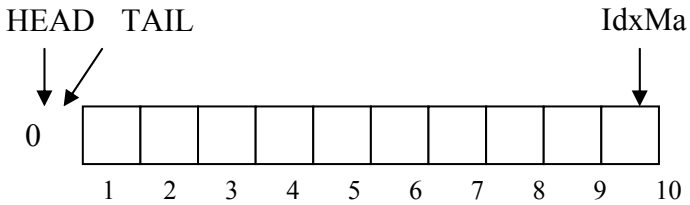
Tabel dengan representasi HEAD dan TAIL, HEAD "bergerak" ketika sebuah elemen dihapus jika Queue tidak kosong. Jika Queue kosong, maka HEAD=0.
Ilustrasi Queue tidak kosong, dengan 5 elemen, kemungkinan pertama HEAD "sedang berada di posisi awal:



Ilustrasi Queue tidak kosong, dengan 5 elemen, kemungkinan pertama HEAD tidak berada di posisi awal. Hal ini terjadi akibat algoritma penghapusan yang dilakukan (lihat keterangan)

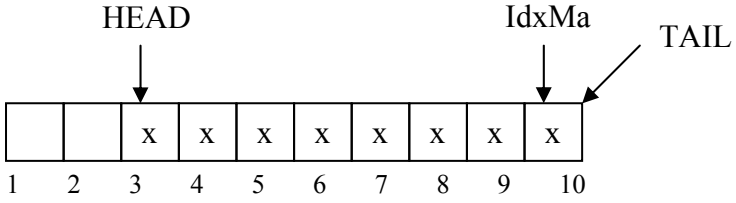


Ilustrasi Queue kosong:



Algoritma untuk **penambahan elemen** sama dengan alternatif I: jika masih ada tempat adalah dengan "memajukan" TAIL. Kasus khusus untuk Queue kosong karena HEAD harus diset nilainya menjadi 1. Algoritmanya sama dengan alternatif I

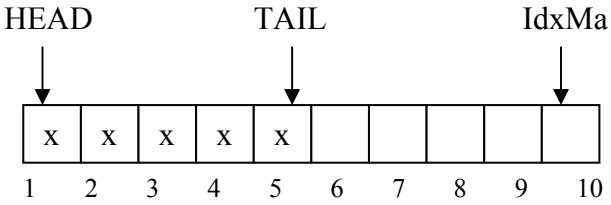
Algoritma untuk **penghapusan elemen** jika Queue tidak kosong: ambil nilai elemen HEAD, kemudian HEAD "maju". Kasus khusus untuk Queue dengan keadaan awal berelemen 1, yaitu menyesuaikan HEAD dan TAIL dengan DEFINISI. Algoritma ini **TIDAK** mencerminkan pergeseran orang yang sedang mengantri di dunia nyata, tapi **efisien**. Namun suatu saat terjadi keadaan Queue penuh tetapi "semu sebagai berikut :



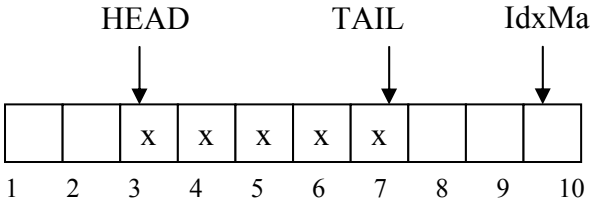
Jika keadaan ini terjadi, haruslah dilakukan aksi menggeser elemen untuk menciptakan ruangan kosong. Pergeseran hanya dilakukan jika dan hanya jika TAIL sudah mencapai IndexMax.

Alternatif III

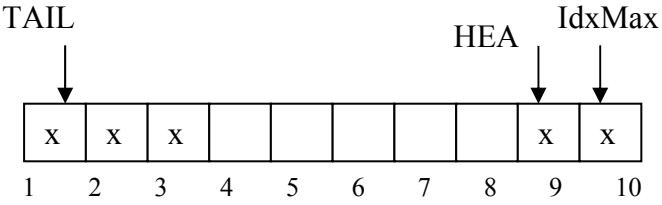
Tabel dengan representasi HEAD dan TAIL yang "berputar" mengelilingi indeks tabel dari awal sampai akhir, kemudian kembali ke awal. Jika Queue kosong, maka HEAD=0. Representasi ini memungkinkan tidak perlu lagi ada pergeseran yang harus dilakukan seperti pada alternatif II pada saat penambahan elemen.
Ilustrasi Queue tidak kosong, dengan 5 elemen, dengan HEAD "sedang" berada di posisi awal:



Ilustrasi Queue tidak kosong, dengan 5 elemen, dengan HEAD tidak berada di posisi awal, tetapi masih "lebih kecil" atau "sebelum" TAIL. Hal ini terjadi akibat algoritma penghapusan/Penambahan yang dilakukan (lihat keterangan).



Ilustrasi Queue tidak kosong, dengan 5 elemen, HEAD tidak berada di posisi awal, tetapi "lebih besar" atau "sesudah" TAIL. Hal ini terjadi akibat algoritma penghapusan/Penambahan yang dilakukan (lihat keterangan)



Algoritma untuk **penambahan elemen**: jika masih ada tempat adalah dengan "memajukan" TAIL. Tapi jika TAIL sudah mencapai IdxMax, maka suksesor dari IdxMax adalah 1 sehingga TAIL yang baru adalah 1. Jika TAIL belum mencapai IdxMax, maka algoritma penambahan elemen sama dengan alternatif II. Kasus khusus untuk Queue kosong karena HEAD harus diset nilainya menjadi 1.

Algoritma untuk **penghapusan elemen** jika Queue tidak kosong: ambil nilai elemen HEAD, kemudian HEAD "maju". Penentuan suatu suksesor dari indkes yang diubah/"maju" dibuat Seperti pada algoritma penambahan elemen: jika HEAD mencapai IdxMAx, maka suksesor dari HEAD adalah 1. Kasus khusus untuk Queue dengan keadaan awal berelemen 1, yaitu menyesuaikan HEAD dan TAIL dengan DEFINISI. Algoritma ini **efisien** karena tidak perlu pergeseran, dan seringkali strategi pemakaian tabel semacam ini disebut sebagai "circular buffer", dimana tabel penyimpan elemen dianggap sebagai "buffer".

Salah satu variasi dari representasi pada alternatif III adalah: menggantikan representasi TAIL dengan COUNT, yaitu banyaknya elemen Queue. Dengan representasi ini,

banyaknya elemen diketahui secara eksplisit, tetapi untuk melakukan penambahan elemen harus dilakukan kalkulasi TAIL. Buatlah sebagai latihan.

ADT Queue dalam Bahasa C, Diimplementasi dengan Tabel

```
/* File : queue.h */
/* Deklarasi Queue yang diimplementasi dengan tabel kontigu */
/* HEAD dan TAIL adalah alamat elemen pertama dan terakhir  */
/* Queue mampu menampung MaxEl buah elemen */
#ifndef queue_H
#define queue_H
#include "boolean.h"
#include <stdlib.h>
#define Nil 0
/* Definisi elemen dan address */
typedef int infotype;
typedef int address;   /* indeks tabel */
/* Contoh deklarasi variabel bertype Queue : */
/*Versi I: tabel dinamik,Head dan Tail eksplisit, ukuran disimpan */
typedef struct { infotype *T; /* tabel penyimpan elemen */
                 address HEAD;  /* alamat penghapusan */
                 address TAIL;  /* alamat penambahan */
                 int MaxEl;      /* MAx elemen queue */
               } Queue;
/* Definisi Queue kosong: Head=Nil; TAIL=Nil. */
/* Catatan implementasi: T[0] tidak pernah dipakai */
/********** AKSES (Selektor) **********/
/* Jika Q adalah Queue, maka akses elemen : */
#define Head(Q) (Q).HEAD
#define Tail(Q) (Q).TAIL
#define InfoHead(Q) (Q).T[(Q).HEAD]
#define InfoTail(Q) (Q).T[(Q).TAIL]
#define MaxEl(Q) (Q).MaxEl
/********** Prototype **********/
boolean IsEmpty (Queue Q);
/* Mengirim true jika Q kosong: lihat definisi di atas */
boolean IsFull(Queue Q);
/* Mengirim true jika tabel penampung elemen Q sudah penuh */
/* yaitu mengandung MaxEl elemen
int NBElmt(Queue Q);
/* Mengirimkan banyaknya elemen queue. Mengirimkan 0 jika Q kosong */
/*** Kreator ***/
void CreateEmpty(Queue *Q, int Max);
/* I.S. sembarang */
/* F.S. Sebuah Q kosong terbentuk dan salah satu kondisi sbb: */
/* Jika alokasi berhasil, Tabel memori dialokasi berukuran Max */
/* atau : jika alokasi gagal, Q kosong dg Maksimum elemen=0 */
/* Proses : Melakukan alokasi,Membuat sebuah Q kosong */
/*** Destruktor ***/
void DeAlokasi(Queue *Q);
/* Proses: Mengembalikan memori Q */
/* I.S. Q pernah dialokasi */
/* F.S. Q menjadi tidak terdefinisi lagi, MaxEl(Q) diset 0 */
/*** Primitif Add/Delete ***/
void Add (Queue * Q, infotype X);
/* Proses: Menambahkan  X pada Q dengan aturan FIFO */
/* I.S. Q mungkin kosong, tabel penampung elemen Q TIDAK penuh */
/* F.S. X menjadi TAIL yang baru, TAIL "maju" */
/* Jika Tail(Q)=MaxEl+1 maka geser isi tabel, shg Head(Q)=1 */
void Del(Queue * Q, infotype* X);
/* Proses: Menghapus  X pada Q dengan aturan FIFO */
/* I.S. Q tidak mungkin kosong */
/* F.S. X = nilai elemen HEAD pd I.S.,HEAD "maju";Q mungkin kosong */
#endif
```

```
/* File :queues.h */
/* Deklarasi Queue yang diimplementasi dengan tabel kontigu */
/* HEAD dan TAIL adalah alamat elemen pertama dan terakhir  */
/* Kapasitas Queue=MaxEl buah elemen, dan indeks dibuat "sirkuler" */
#ifndef queues_H
#define queues_H
#include "boolean.h"
#include <stdlib.h>
#define Nil 0
/* Definisi elemen dan address */
typedef int infotype;
typedef int address;   /* indeks tabel */
/* Contoh deklarasi variabel bertype Queue : */
/*Versi I: tabel dinamik,Head dan Tail eksplisit, ukuran disimpan */
typedef struct { infotype *T;  /* tabel penyimpan elemen */
                 address HEAD; /* alamat penghapusan */
                 address TAIL; /* alamat penambahan */
                 int MaxEl;    /* MAx elemen queue */
               } Queue;
/* Definisi Queue kosong: Head=Nil; TAIL=Nil. */
/* Catatan implementasi: T[0] tidak pernah dipakai */
/********** AKSES (Selektor) **********/
/* Jika Q adalah Queue, maka akses elemen : */
#define Head(Q) (Q).HEAD
#define Tail(Q) (Q).TAIL
#define InfoHead(Q) (Q).T[(Q).HEAD]
#define InfoTail(Q) (Q).T[(Q).TAIL]
#define MaxEl(Q) (Q).MaxEl
/********** Prototype **********/
boolean IsEmpty (Queue Q);
/* Mengirim true jika Q kosong: lihat definisi di atas */
boolean IsFull(Queue Q);
/* Mengirim true jika tabel penampung elemen Q sudah penuh */
/* yaitu mengandung MaxEl elemen */
int NBElmt(Queue Q);
/* Mengirimkan banyaknya elemen queue. Mengirimkan 0 jika Q kosong */
/*** Kreator ***/
void CreateEmpty(Queue *Q, int Max);
/* I.S. sembarang */
/* F.S. Sebuah Q kosong terbentuk dan salah satu kondisi sbb: */
/* Jika alokasi berhasil, Tabel memori dialokasi berukuran Max */
/* atau : jika alokasi gagal, Q kosong dg Maksimum elemen=0 */
/* Proses : Melakukan alokasi,Membuat sebuah Q kosong */
/*** Destruktor ***/
void DeAlokasi(Queue *Q);
/* Proses: Mengembalikan memori Q */
/* I.S. Q pernah dialokasi */
/* F.S. Q menjadi tidak terdefinisi lagi, MaxEl(Q) diset 0 */
/*** Primitif Add/Delete ***/
void Add (Queue * Q, infotype X);
/* Proses: Menambahkan  X pada Q dengan aturan FIFO */
/* I.S. Q mungkin kosong, tabel penampung elemen Q TIDAK penuh */
/* F.S. X menjadi TAIL yang baru, TAIL "maju" */
/* Jika Tail(Q)=MaxEl+1 maka Tail(Q) diset =1 */
void Del(Queue * Q, infotype* X);
/* Proses: Menghapus  X pada Q dengan aturan FIFO */
/* I.S. Q tidak mungkin kosong */
/* F.S. X = nilai elemen HEAD pd I.S.,Jika Head(Q)=MaxEl+1, */
/* Head(Q) diset=1; Q mungkin kosong */
#endif
```

QUEUE dengan Representasi Berkait

Buatlah sebagai latihan, setelah pembahasan list dengan repesentasi berkait.

List Linier

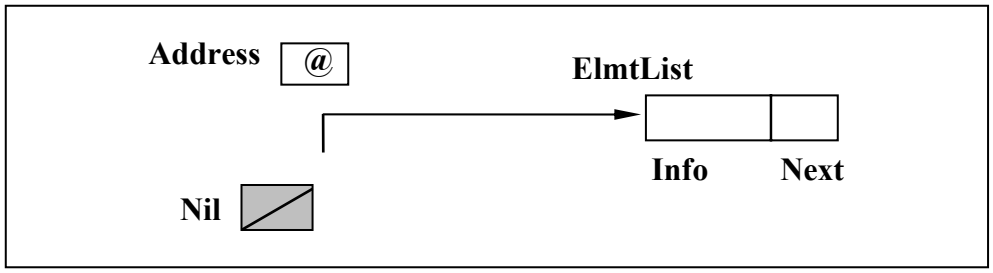
Definisi

List linier adalah s**ekumpulan elemen** ber-*type* sama, yang mempunyai "keterurutan" tertentu, dan setiap elemennya terdiri dari dua bagian, yaitu informasi mengenai elemennya, dan informasi mengenai alamat elemen suksesornya:

type ElmtList : < Info : InfoType, Next : address >

dengan InfoType adalah sebuah *type* terdefinisi yang menyimpan informasi sebuah elemen list; **Next** adalah address ("**alamat**") dari elemen berikutnya (suksesor). Dengan demikian, jika didefinisikan **First** adalah alamat elemen pertama list, maka elemen berikutnya dapat diakses secara suksesif dari **field** Mext elemen tersebut.

Alamat yang sudah didefinisikan disebut sudah "di-alokasi". Didefinisikan suatu konstanta **Nil**, yang artinya alamat yang tidak terdefinisi. Alamat ini nantinya akan didefinisikan secara lebih konkret ketika list linier diimplementasi pada struktur data fisik.



Jadi, sebuah list linier dikenali :

- **elemen pertamanya**, biasanya melalui alamat elemen pertama yang disebut : First
- alamat **elemen berikutnya** (suksesor), jika kita mengetahui alamat sebuah elemen, yang dapat diakses melalui informasi NEXT. NEXT mungkin ada secara **eksplisit** (seperti contoh di atas), atau secara **implisit** yaitu lewat kalkulasi atau fungsi suksesor.
- setiap elemen mempunyai **alamat**, yaitu tempat elemen disimpan dapat diacu. Untuk mengacu sebuah elemen, alamat harus terdefinisi. Dengan alamat tersebut Informasi yang tersimpan pada elemen list dapat diakses.
- **elemen terakhirnya**. Ada berbagai cara untuk mengenali elemen akhir

Jika L adalah list, dan P adalah address:
Alamat elemen pertama list L dapat diacu dengan notasi :

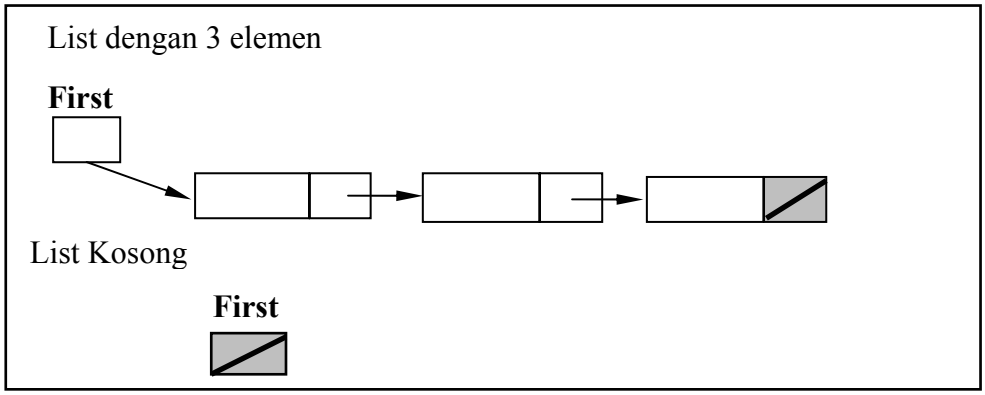
First(L)

Elemen yang diacu oleh P dapat dikonsultasi informasinya dengan notasi **Selektor** :

Info(P)
Next(P)

Beberapa definisi :

- List L adalah list kosong, jika First(L) = Nil.
- Elemen terakhir dikenali, misalnya jika Last adalah alamat element terakhir, maka Next(Last) = Nil.
- Biasanya list linier paling sederhana digambarkan dengan ilustrasi sebagai berikut:



Skema Traversal untuk List Linier

List adalah koleksi objek, yang terdiri dari sekumpulan elemen. Seringkali diperlukan proses terhadap setiap elemen list dengan cara yang sama. Karena itu salah satu primitif operasi konsultasi dasar pada struktur list adalah *traversal*, yaitu "mengunjungi" setiap elemen list untuk diproses. Karena definisi list linier, urutan akses adalah dari elemen pertama sampai dengan elemen terakhir, maka *traversal* list secara natural dilakukan dari elemen pertama, suksesornya, dan seterusnya sampai dengan elemen terakhir. Skema *traversal* yang dipakai adalah skema yang telah dipelajari pada Bagian I.

```
procedure SKEMAListTraversal1 (input L : List)
{ I.S. List L terdefinisi, mungkin kosong }
{ F.S. semua elemen list  L "dikunjungi" dan telah diproses }
{ Traversal sebuah list linier. Dengan MARK, tanpa pemrosesan khusus pada list
kosong }
```

```
KAMUS LOKAL
    P : address   { address untuk traversal, type terdefinisi }
    procedure Proses (input P : address )  { pemrosesan elemen
                                             ber-address P }
    procedure Inisialisasi  { aksi sebelum proses dilakukan }
    procedure Terminasi     { aksi sesudah semua pemrosesan elemen
                              selesai }
```

```
ALGORITMA
    Inisialisasi
    P ← First(L)
    while (P ≠  Nil) do
        Proses (P)
        P ← Next(P)
    Terminasi
```

```
procedure SKEMAListTraversal2 (input L : List)
{ I.S. List L terdefinisi, mungkin kosong }
{ F.S. Semua elemen list  L "dikunjungi" dan telah  diproses }
{ Traversal sebuah list linier yang diidentifikasi oleh elemen pertamanya L }
{ Skema sekuensial dengan MARK dan pemrosesan khusus pada list kosong }
```

```
KAMUS LOKAL
   P : address  {address untuk traversal }
   procedure Proses (input P : address )  { pemrosesan elemen ber-
                                            address P }
   procedure Inisialisasi   { aksi sebelum proses dilakukan }
   procedure Terminasi      { aksi sesudah semua pemrosesan elemen
                              selesai }
```

```
ALGORITMA
   if First(L) = Nil then
      output ("List kosong")
   else
      Inisialisasi
      P ← First(L)
      repeat
         Proses (P)
         P ←  Next(P)
      until (P = Nil)
      Terminasi
```

```
procedure SKEMAListTraversal3 (input L : List)
{ I.S. List L terdefinisi, tidak  kosong : minimal mengandung satu elemen }
{ F.S. Semua elemen list L "dikunjungi" dan telah diproses }
{ Skema sekuensial tanpa MARK, tidak ada list kosong karena tanpa mark }
```

```
KAMUS LOKAL
   P : address  { address untuk traversal, type terdefinisi}
   procedure Proses (input P : address )  { pemrosesas elemen ber-
                                            address P }
   procedure Inisialisasi { aksi sebelum proses dilakukan }
   procedure Terminasi { aksi sesudah semua pemrosesan elemen selesai }
```

```
ALGORITMA
   Inisialisasi
   P ← First(L)
   iterate
      Proses (P)
   stop (Next(P) = Nil)
      P ← Next(P)
   Terminasi
```

Skema *Sequential Search* untuk List Linier

Selain *traversal*, proses pencarian suatu elemen list adalah primitif yang seringkali didefinisikan pada struktur list. Pencarian dapat berdasarkan nilai atau berdasarkan alamat. Skema pencarian yang ditulis adalah skema pencarian berdasarkan pencarian pada tabel.

Search Suatu Nilai, Output adalah Address

Search ini sering dipakai untuk mengenali suatu elemen list berdasarkan nilai informasi yang disimpan pada elemen yang dicari. Biasanya dengan alamat yang ditemukan, akan dilakukan suatu proses terhadap elemen list tersebut.

```
procedure SKEMAListSearch1 (input L : List, input X : InfoType,
                            output P : address, input Found : boolean)
{ I.S. List linier L sudah terdefinisi dan siap dikonsultasi, X terdefinisi }
{ F.S. P : address pada pencarian beurutan, dimana X diketemukan, P = Nil jika
tidak ketemu }
{ Found berharga true jika harga X yang dicari ketemu, false jika tidak ketemu }
{ Sequential Search harga X pada sebuah list linier L  }
{ Elemen diperiksa dengan instruksi yang sama, versi dengan boolean }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   P  ←  First(L)
   Found ←  false
   while (P ≠ Nil) and (not Found) do
      if  (X = Info(P)) then
         Found ← true
      else
         P ← Next(P)
   { P = Nil or Found}
   { Jika Found maka P = address dari harga yg dicari diketemukan }
   { Jika not Found maka P = Nil }
```

```
procedure SKEMAListSearch2 (input L : List, input X : InfoType,
                            output P : address, input Found : boolean)
{ I.S. List linier L sudah terdefinisi dan siap dikonsultasi, X terdefinisi }
{ F.S. P : address pada pencarian beurutan, dimana X ditemukan, P = Nil jika
tidak ketemu }
{ Found berharga true jika harga X yang dicari ketemu, false jika tidak
ditemukan }
{ Sequential Search harga X pada  sebuah list linier L }
{ Elemen terakhir diperiksa secara khusus, versi tanpa boolean }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   { List linier L sudah terdefinisi dan siap dikonsultasi }
   if (First(L) = Nil) then
      output ("List kosong")
   else   { First(L) ≠ Nil, suksesor elemen pertama ada }
      P ← First(L)
      while ((Next(P) ≠ Nil) and (X ≠ Info(P)) do
         P ← Next(P)
      { Next(P) = Nil  or  X = Info(P) }
      depend on P, X
         X = Info(P) : Found ← true
         X ≠ Info(P) : Found ← false; P ← Nil
```

Search Suatu Elemen yang Beralamat Tertentu

Search ini sering dipakai untuk memposisikan *"current pointer"* pada suatu elemen list.

```
procedure SKEMAListSearch@ (input L : List, input P : address,
                            output Found : boolean)
{ I.S. List linier L sudah terdefinisi dan siap dikonsultasi, P terdefinisi }
{ F.S. Jika ada elemen list yang beralamat P, Found berharga true }
{      Jika tidak ada elemen list beralamat P, Found berharga false }
{ Sequential Search address P pada sebuah list linier L }
{ Semua elemen diperiksa dengan instruksi yang sama }
KAMUS LOKAL
    Pt : address

ALGORITMA
    Pt ← First(L)
    Found ← false
    while (Pt ≠ Nil) and (not Found) do
        if (Pt = P) then
            Found ← true
        else
            Pt ← Next(Pt)
    { Pt = Nil or Found }
    { Jika Found maka P adalah elemen list }
```

Search Suatu Elemen dengan KONDISI Tertentu

```
procedure SKEMAListSearchX (input L : List, input Kondisi(P) : boolean,
                            output P : address, input Found : boolean)
{ I.S. List linier L sudah terdefinisi dan siap dikonsultasi, Kondisi(P) adalah
suatu ekspresi boolean  yang merupakan fungsi dari elemen beralamat P }
{ F.S. Jika ada elemen list P yang memenuhi Kondisi (P), maka  P adalah alamat
dari elemen yang memenuhi kondisi tersebut, Found berharga true }
{ Jika tidak ada elemen list P yang memenuhi Kondisi(P), maka Found berharga
false dan P adalah Nil }
{ Semua elemen diperiksa dengan instruksi yang sama }
KAMUS LOKAL
    P : address

ALGORITMA
    { List linier L sudah terdefinisi dan siap dikonsultasi }
    P ← First(L)
    Found ← false
    while (P ≠ Nil) and (not Found) do
        if  Kondisi(P) then
            Found ← true
        else
            P ← Next(P)
    { P = Nil  or  Found}
    { Jika Found maka P adalah elemen list dengan Kondisi(P) true }
```

Contoh :

- Kondisi(P) adalah elemen list dengan Info(P) bernilai 5, maka ekspresi adalah Info(P)=5.
- Kondisi(P) adalah elemen terakhir, berarti ekspresi Kondisi(P) adalah Next(P)= Nil.
- Kondisi(P) adalah elemen sebelum terakhir, berarti ekspresi Kondisi(P) adalah: Next(Next(P)) = Nil.
- Kondisi(P) adalah elemen sebelum elemen beralamat PX, berarti ekspresi Kondisi(P) adalah Next(PX) = P.

Definisi Fungsional List Linier

Secara fungsional, pada sebuah list linier biasanya dilakukan **pembuatan**, **penambahan**, **perubahan**, atau **penghapusan** elemen atau proses terhadap keseluruhan list, yang dapat ditulis sebagai berikut:

```
Diberikan L, L1 dan L2 adalah list linier dengan elemen ElmtList
ListEmpty  : L → boolean  { Tes apakah list kosong }
CreateList : → L          { Membentuk sebuah list linier kosong }
Insert     : ElmtList x L → L  { Menyisipkan sebuah elemen ke dalam
                                 list }
Delete     : L → L x ElmtList  { Menghapus sebuah elemen list }
UpdateList : ElmtList x L → L  { Mengubah informasi sebuah elemen list
                                 linier }
Concat     : L1 x L2 → L       { Menyambung L1 dengan L2 }
```

Dari definisi fungsional tersebut, dapat diturunkan fungsi-fungsi lain. UpdateList tidak dibahas di sini, karena dapat dilakukan dengan skema *search*, kemudian mengubah info. Mengisi elemen sebuah list tidak dibahas di sini, karena dapat dijabarkan dari penyisipan satu per satu elemen list yang dibaca dari sebuah arsip atau dari alat masukan. Berikut ini hanya akan dibahas Penyisipan, Penghapusan dan Konkatenasi:

Operasi Primitif List Linier

Pemeriksaan Apakah List Kosong

```
IsEmptyList : L → boolean     { Tes apakah sebuah list kosong }
```

Pemeriksaan apakah sebuah list kosong sangat penting, karena I.S. dan F.S. beberapa prosedur harus didefinisikan berdasarkan keadaan list. Operasi pada list kosong seringkali membutuhkan penanganan khusus. Realisasi algoritmik dari definisi fungsional ini adalah sebuah fungsi sebagai berikut:

```
function  IsEmptyList (L : List) → boolean
{ Tes apakah sebuah list L kosong. Mengirimkan true jika list kosong, false jika
tidak kosong }

KAMUS LOKAL

ALGORITMA
   → (First(L) = Nil)
```

Pembuatan List Kosong

```
CreateList : → L   { Membentuk sebuah list linier kosong }
```

Pembuatan sebuah list berarti membuat sebuah list KOSONG, yang selanjutnya siap diproses (ditambah elemennya, dsb.). Realisasi algoritmik dari definisi fungsional ini adalah sebuah prosedur sebagai berikut:

```
procedure CreateList (output L : List)
{ I.S. Sembarang }
{ F.S. Terbentuk list L kosong: First(L) diinisialisasi dengan NIL }
KAMUS LOKAL

ALGORITMA
    First(L) ← Nil
```

Penyisipan Sebuah Elemen pada List Linier

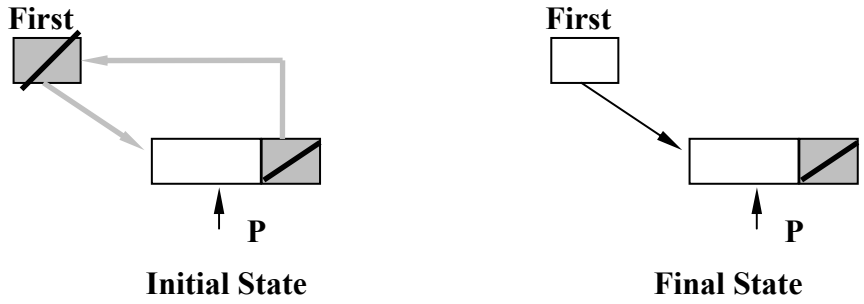
```
Insert : ElmtList x L → L { Menyisipkan sebuah elemen ke dalam list }
```

Fungsi *insert* (penyisipan) harus dijabarkan lebih rinci karena dapat menjadi penyisipan sebagai elemen pertama, setelah sebuah address P atau penyisipan menjadi elemen terakhir. Berikut ini akan diberikan skema prosedur penyisipan yang diturunkan dari definisi fungsional tersebut:

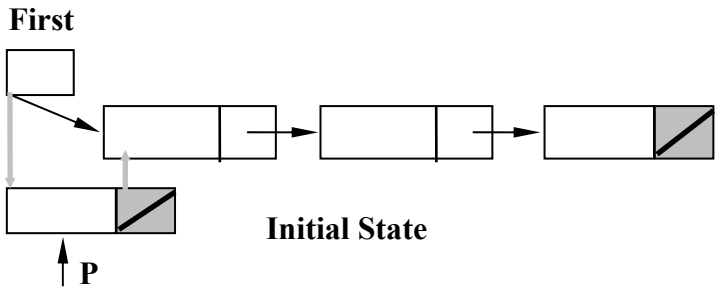
Penyisipan sebuah elemen dapat dilakukan terhadap sebuah elemen yang sudah dialokasi (diketahui address-nya), atau sebuah elemen yang hanya diketahui nilai Info-nya (berarti belum dialokasi). Perhatikanlah perbedaan ini pada realisasi primitif.

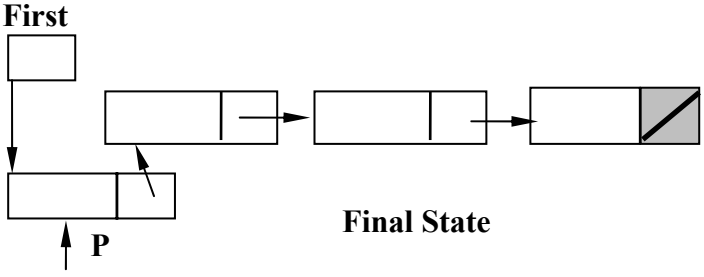
INSERT-First (diketahui alamat)
Menambahkan sebuah elemen yang diketahui alamatnya sebagai elemen pertama list.

Insert elemen pertama, List kosong :



Insert elemen pertama, List tidak kosong :





```
procedure InsertFirst (input/output L : List, input P : address)
{ I.S. List L mungkin kosong, P sudah dialokasi, P ≠ Nil, Next(P)=Nil }
{ F.S. P adalah elemen pertama list L }
{ Insert sebuah elemen beralamat P sebagai elemen pertama list linier L yang
mungkin kosong }

KAMUS LOKAL

ALGORITMA
    Next(P) ← First(L)
    First(L) ← P
```

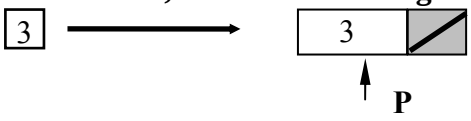
INSERT-First (diketahui nilai)

Menambahkan sebuah elemen yang diketahui nilainya sebagai elemen pertama list.

Tahap pertama :

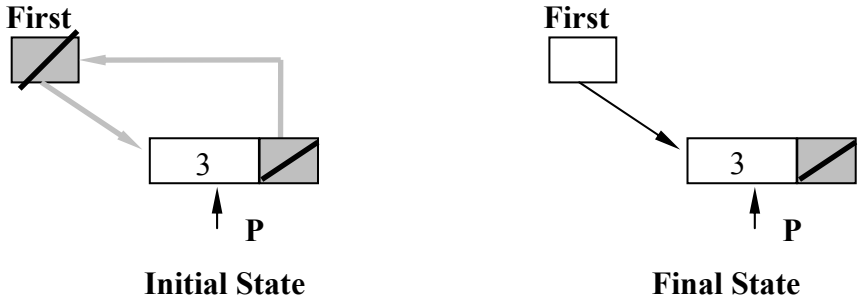
Insert Nilai 3 sebagai elemen pertama, List : karena yang diketahui adalah nilai, maka harus dialokasikan dahulu sebuah elemen supaya nilai 3 dapat di-insert

Jika alokasi berhasil, P tidak sama dengan Nil

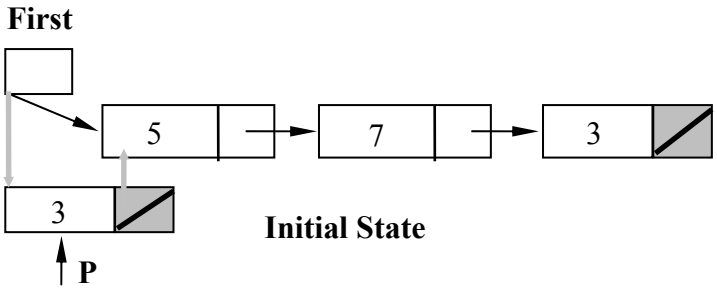


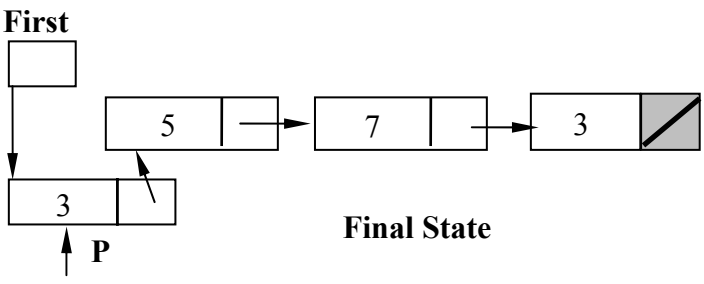
Tahap kedua : insert

Insert ke list kosong



Insert elemen pertama, List tidak kosong :





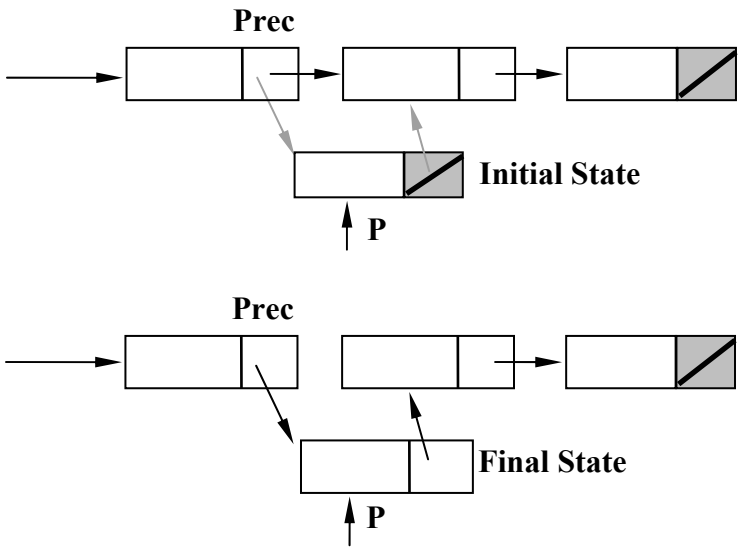
```
procedure InsFirst (input/output L : List, input InfoE : InfoType)
{ I.S. List L mungkin kosong }
{ F.S. Sebuah elemen dialokasi dan menjadi elemen pertama list L, jika alokasi
berhasil. Jika alokasi gagal list tetap seperti semula. }
{ Insert sebuah elemen sbg. elemen pertama list linier L yang mungkin kosong. }
```

```
KAMUS LOKAL
   function Alokasi (X : infotype) → address
   { Menghasilkan address yang dialokasi. Jika alokasi berhasil,
     Info(P)=InfoE, dan Next(P)=Nil. Jika alokasi gagal, P=Nil }
   P : address { Perhatikan: P adalah variabel lokal!
                 Akan dibahas di kelas }
```

```
ALGORITMA
   P ← Alokasi(InfoE)
   if P ≠  Nil then
       Next(P) ← First(L); First(L) ← P
```

INSERT-After

Menyisipkan sebuah elemen beralamat P setelah sebagai suksesor dari sebuah elemen list linier yang beralamat Prec.




```
procedure InsertAfter (input P, Prec : address)
{ I.S.  Prec adalah elemen list,  Prec ≠ Nil, P sudah dialokasi, P ≠ Nil, Next(P)
= Nil }
{ F.S. P menjadi suksesor Prec }
{ Insert sebuah elemen beralamat P pada List Linier L}
```

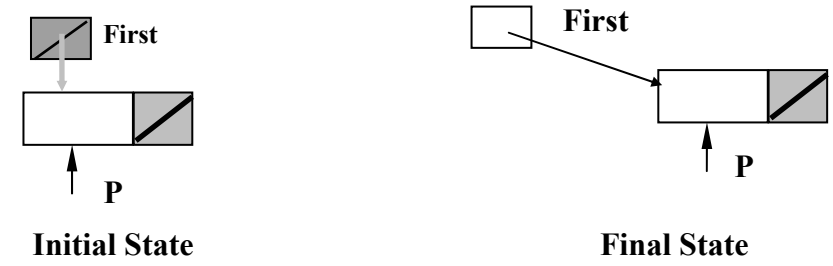
```
KAMUS LOKAL
```

```
ALGORITMA
   Next(P) ← Next(Prec)
   Next(Prec) ← P
```

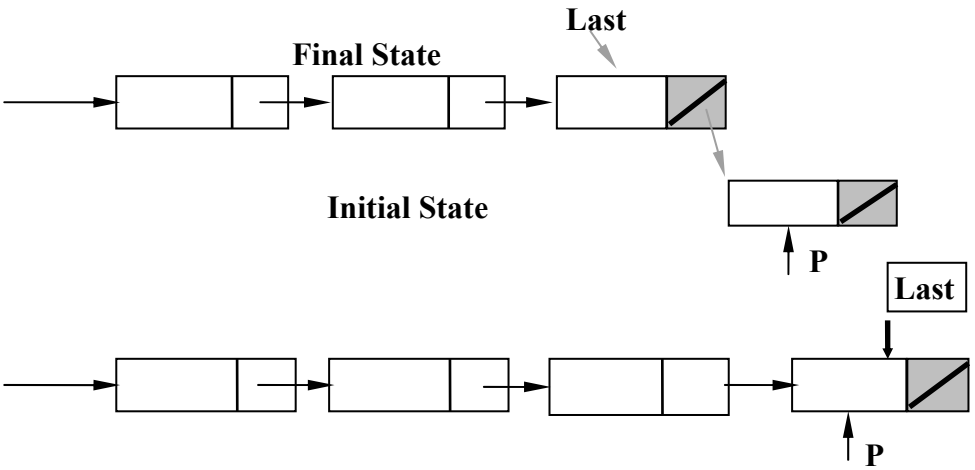
INSERT-Last

Menyisipkan sebuah elemen beralamat P setelah sebagai elemen terakhir sebuah list linier. Ada dua kemungkinan list kosong atau tidak kosong

Insert sebagai elemen terakhir list tidak kosong



Insert sebagai elemen terakhir list tidak kosong:




```
procedure InsertLast (input/output L : List, input P : address)
{ I.S. List L mungkin kosong, P sudah dialokasi, P ≠ Nil, Next(P)=Nil }
{ F.S. P adalah elemen trerakhir list L }
{ Insert sebuah elemen beralamat P sbg elemen terakhir dari list linier L yg
mungkin kosong }
```

```
KAMUS LOKAL
    Last : address { address untuk traversal,
                     pada akhirnya address elemen terakhir }
```

```
ALGORITMA
    if First(L) = Nil then { insert sebagai elemen pertama }
        InsertFirst (L,P)
    else
        { Traversal list sampai address terakhir }
        { Bagaimana menghindari traversal list untuk mencapai Last? }
        Last ← First(L)
        while (Next(Last) ≠ Nil) do
            Last ← Next(Last)
        { Next(Last) = Nil, Last adalah elemen terakhir }
        { Insert P after Last }
        InsertAfter(P,Last)
```

```
procedure InsLast (input/output L : List, input InfoE : InfoType)
{ I.S. List L mungkin kosong }
{ F.S. Jika alokasi berhasil, InfoE adalah nilai elemen terakhir L }
{ Jika alokasi gagal, maka F.S. = I.S. }
{ Insert sebuah elemen beralamat P (jika alokasi berhasil) sebagai elemen
terakhir  dari list linier L yg mungkin kosong }
```

```
KAMUS LOKAL
   function Alokasi(X : InfoType) → address
   { Menghasilkan address yang dialokasi. Jika alokasi berhasil,
     Info(P)=InfoE, dan Next(P)= Nil. Jika alokasi gagal, P=Nil }
   P : address
```

```
ALGORITMA
   P ← Alokasi(InfoE)
   if P ≠ Nil then { insert sebagai elemen pertama }
       InsertLast(L,P)
```

Penghapusan Sebuah Elemen pada List Linier

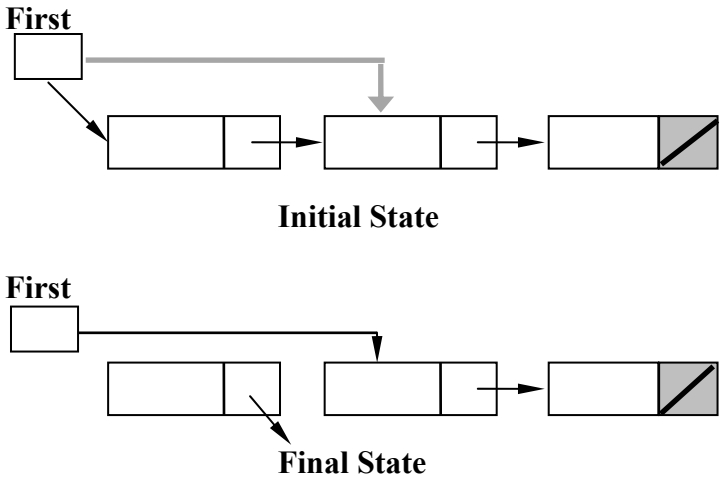
```
Delete : L → L x ElmtList { Menghapus sebuah elemen dalam list }
```

Penghapusan harus dijabarkan lebih rinci, karena penghapusan elemen dapat merupakan pertama, setelah sebuah address P atau penghapusan elemen terakhir. Perbedaan ini melahirkan tiga operasi dasar penghapusan elemen list yang diturunkan dari definisi fungsional ini menjadi realisasi algoritmik, yaitu penghapusan elemen pertama, di tengah dan terakhir.

Operasi penghapusan dapat mengakibatkan list menjadi kosong, jika list semula hanya terdiri dari satu elemen.

DELETE-First: menghapus elemen pertama list linier

a. Elemen yang dihapus dicatat alamatnya :



```
procedure DeleteFirst (input/output L : List, output P : address)
{ I.S. List L tidak kosong, minimal 1 elemen, elemen pertama pasti ada }
{ F.S. First "maju", mungkin bernilai Nil (list menjadi kosong) }
{ Menghapus elemen pertama L, P adalah @ elemen pertama L sebelum penghapusan, L
yang baru adalah Next(L) }
```

```
KAMUS LOKAL
```

```
ALGORITMA
    P ← First(L)
    First(L) ← Next(First(L))
    { Perhatikan bahwa tetap benar jika list menjadi kosong }
```

b. Elemen yang dihapus dicatat informasinya, dan alamat elemen yang dihapus didealokasi :

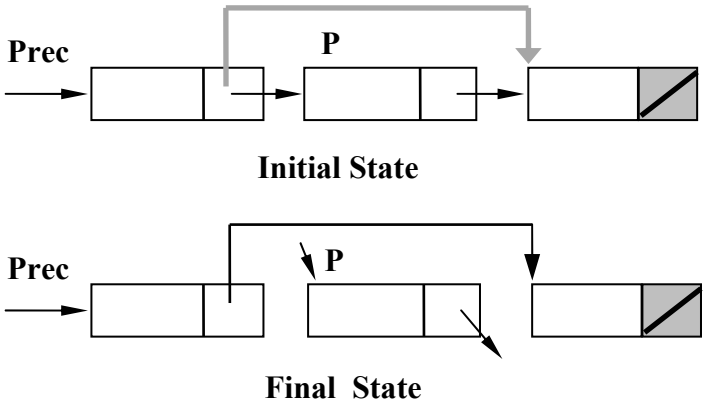
```
procedure DeleteFirst (input/output L : List, output E : InfoType)
{ I.S. List L tidak kosong, minimal 1 elemen, elemen pertama pasti ada }
{ F.S : Menghapus elemen pertama L
  E adalah nilai elemen pertama L sebelum penghapusan, L yang baru
  adalah Next(L) }
```

```
KAMUS LOKAL
   procedure DeAlokasi (input P : address)
   { I.S. P pernah dialokasi. F.S. P=Nil }
   { F.S. Mengembalikan address yang pernah dialokasi. P=Nil}
   P : address
```

```
ALGORITMA
    P ← First(L); E ← Info(P)
    First(L) ← Next(First(L)) { List kosong : First(L) menjadi Nil }
    Next(P) ← Nil; Dealokasi(P)
```

DELETE-After

Penghapusan suksesor sebuah elemen :



```
procedure DeleteAfter (input Prec : address, output P : address)
{ I.S. List tidak kosong, Prec adalah elemen list, Next(Prec) ≠ Nil }
{ F.S. Next(Prec), yaitu elemen beralamat P dihapus dari List.
       Next(P)=Nil }
{ Menghapus suksesor Prec, P adalah @ suksesor Prec sebelum penghapusan,
  Next(Prec) yang baru adalah suksesor dari suksesor Prec sebelum penghapusan }
```

```
KAMUS LOKAL
```

```
ALGORITMA
    P ← Next(Prec)
    Next(Prec) ← Next(Next(Prec))
    Next(P) ← Nil
```

Dengan primitif ini, maka penghapusan sebuah elemen beralamat P dapat dilakukan dengan :

- mencari predesesor dari P, yaitu alamat Prec.
- memakai DeleteAfter(Prec).

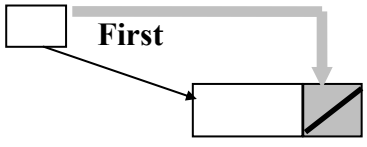
```
procedure DeleteP (input/output L : List, output P : address)
{ I.S. List L tidak kosong, P adalah elemen list L }
{ F.S. Menghapus P dari list L, P mungkin elemen pertama, "tengah", atau
terakhir }
```

```
KAMUS LOKAL
    Prec : address     { alamat  predesesor P }
```

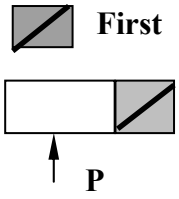
```
ALGORITMA
    { Cari predesesor P }
    if (P = First(L)) then   { Delete list dengan satu elemen }
        DeleteFirst(L,P)
    else
        Prec ← First(L)
        while (Next(Prec) ≠ P) do
            Prec ← Next(Prec)
        { Next(Prec) = P, hapus P }
        DeleteAfter(Prec,P)
```

DELETE-Last

Menghapus elemen terakhir list dapat dilakukan jika alamat dari eemen sebelum elemen terakhir diketahui. Persoalan selanjutnya menjadi persoalan DeleteAfter, kalau Last bukan satu-satunya elemen list linier. Ada dua kasus, yaitu list menjadi kosong atau tidak.
Kasus list menjadi kosong :

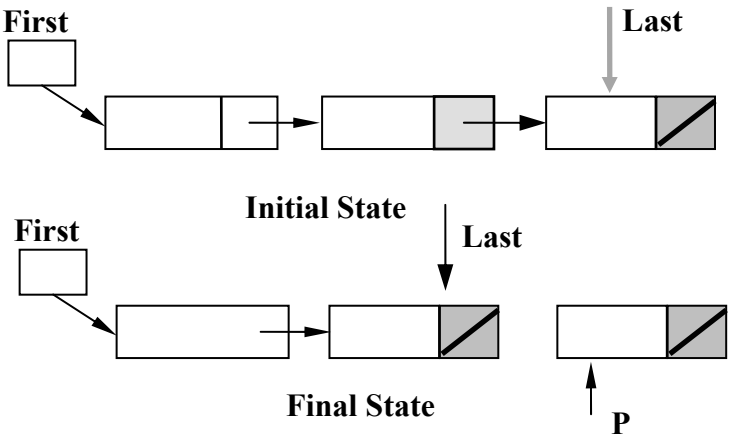


Initial State



Final State

List tidak menjadi kosong (masih mengandung elemen) :




```
procedure DeleteLast (input First : List, output P : address)
{ I.S. List L tidak kosong, minimal mengandung 1 elemen }
{ F.S. P berisi alamat elemen yang dihapus. Next(P)=Nil. List berkurang
elemennya }
{ Menghapus elemen terakhir dari list L, list mungkin menjadi kosong }
```

```
KAMUS LOKAL
    Last, PrecLast : address { address untuk traversal, type terdefinisi
                               pada akhirnya Last adalah alamat elemen
                               terakhir dan PrecLast adalah alamat
                               sebelum yg terakhir }
```

```
ALGORITMA
    { Find Last dan address sebelum Last }
    Last ← First(L)
    PrecLast ← Nil { predesesor dari L tak terdefinisi }
    while (Next(Last) ≠ Nil) do
        { Traversal list sampai @ terakhir }
        PrecLast ← Last
        Last ← Next(Last)
    { Next(Last) = Nil, Last adalah elemen terakhir }
    { PrecLast = sebelum Last }
    P ← Last
    if (PrecLast = Nil) then { list dengan 1 elemen, jadi kosong }
        First(L) ← Nil
    else
        Next(PrecLast) ← Nil
```

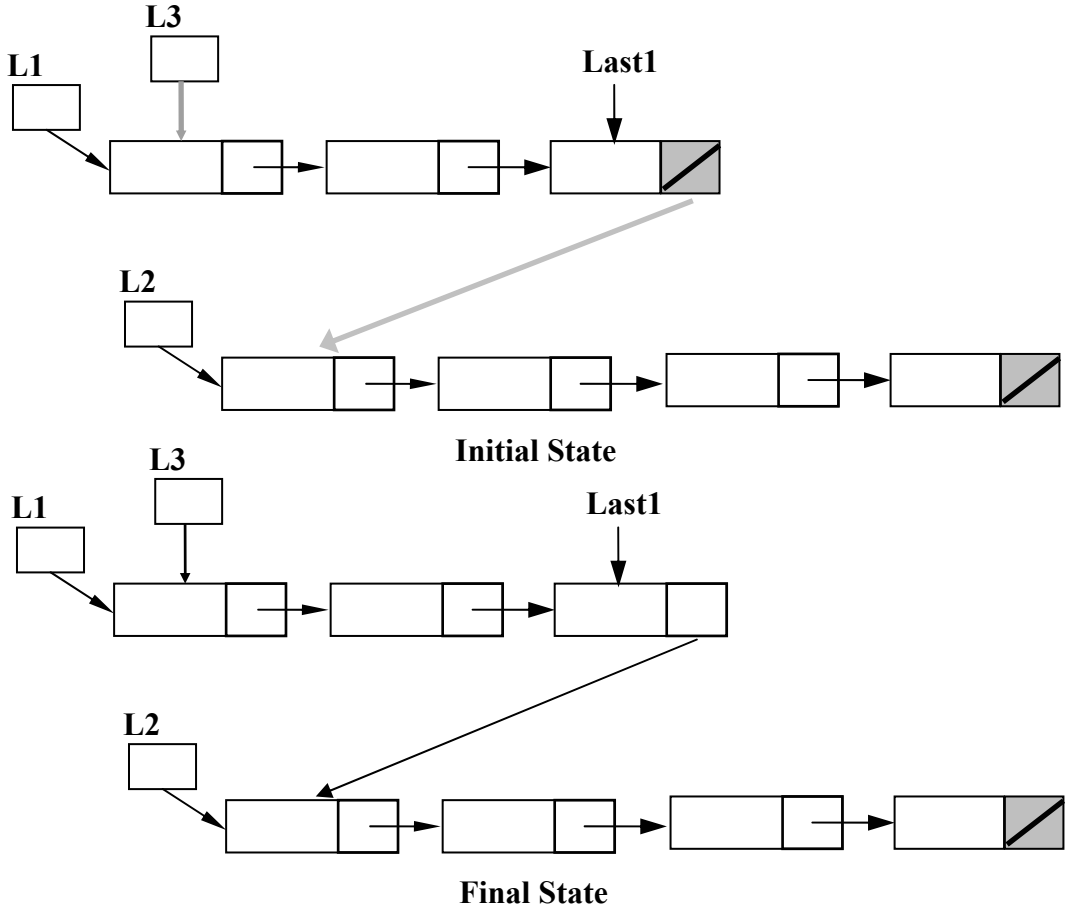
Catatan:

- Operasi *delete* yang tidak melakukan dealokasi addiress, ditawarkan jika *address* tersebut masih dibutuhkan. Misalnya elemen yang dihapus akan dijadikan anggota elemen list lain: dalam hal ini tidak perlu melakukan dealokasi, kemudian alokasi ulang ketika akan ditambahkan ke list yang baru. Beberapa list dengan elemen sejenis seringkali harus dikelola dalam program; misalnya list mahasiswa per matakuliah.
- Operasi *delete* yang sekaligus melakukan dealokasi addiress, biasanya ditawarkan ke pengguna yang tidak perlu mengetahui "keberadaan" *address*, dan yang hanya perlu memanipulasi informasi pada *field* Info.

Konkatenasi Dua Buah List Linier

```
Concat : L1 x L2  → L   { Menyambung L1 dengan L2 }
```

Konkatenasi adalah menggabungkan dua list. Dalam contoh berikut list kedua disambungkan ke list pertama. Jadi Last(L1) menjadi predesesor First(l2). Realisasi algoritmiknya adalah sebuah prosedur sebagai berikut:




```
procedure CONCAT (input L1, L2 : List, output L3 : List)
{ I.S. L1 ≠ L2, L1 ≠ L3, dan L3 ≠ L2; L1, L2 mungkin kosong }
{ F.S. L3 adalah hasil Konkatenasi ("Menyambung") dua buah list linier, L2
ditaruh di belakang L1 }
{ Catatan : Pelajari baik-baik algoritma berikut : apakah kedua list asal tetap
dapat dikenali? }
```

```
KAMUS LOKAL
    Last1 : address { alamat elemen terakhir list pertama }
```

```
ALGORITMA
    CreateList(L3)     { inisialisasi list hasil }
    if First(L1) = Nil then
        First(L3) ← First(L2)
    else { Traversal list 1 sampai address terakhir,
           hubungkan Last1 dengan First(L2) }
        First(L3) ← First(L1)
        Last1 ← First(L1)
        while (Next(Last1) ≠ Nil) do
            Last1 ← Next(Last1)
        { Next(Last1) = Nil, Last adalah elemen terakhir }
        Next(Last1) ← First(L2)
```

Latihan Soal

1. Coba tuliskan skema *sequential search* untuk sesudah list linier yang informasi elemennya terurut membesar.
2. Apakah teknik sentinel layak untuk diterapkan pada *sequential search* untuk list linier? Jelaskanlah alasannya.
3. Diskusikan kedua skema *sequential search* yang berbeda tersebut. Sebagai bahan pemikiran, tinjau pemakaian *mark* dan tidak.
4. Skema berikut adalah skema yang mengandung kesalahan FATAL. Di mana letak kesalahannya?

```
procedure SKEMAListSearch3 (input L : List, input X : InfoType)
{ Sequential Search harga X pada sebuah list linier sederhana }

KAMUS LOKAL
   P : address  { address pada pencarian beurutan, di mana X diketemukan
                  P = Nil jika tidak ketemu }
   Found : boolean { true jika tempat penyisipan yg benar ketemu }

ALGORITMA
   { List linier L sudah terdefinisi dan siap dikonsultasi }
   P ← First(L)(L)
   while (Next(P) ≠ Nil) and (X ≠ Info(P)) do
      P ← Next(P)
   { Next(P) = Nil  or  X = Info(P) }
   Found ← (X = Info(P))
```

5. Tuliskan skema *sequential search* untuk sebuah list linier jika yang dicari bukan suatu harga dari InfoType, tetapi sebuah alamat:

   procedure SearchAdr (input L : List, input P : address, output Found : boolean)

6. Tuliskan operasi konkat dengan "menyalin" elemen list. Dengan demikian, mungkin terjadi kegagalan alokasi.. Jika ada alokasi gagal, list yang telanjur dibentuk harus dihancurkan.
7. Tuliskan semua operasi list yang ada di Diktat ”Pemrograman Fungsional” menjadi versi iteratif.

Representasi Fisik List Linier

Representasi lojik List Linier L, dengan P adalah address telah didefinisikan sebelumnya dengan penulisan sebagai berikut :

First(L)
Next(P), Info(P)

Berikut ini akan diberikan bagaimana representasinya secara "fisik", yaitu implementasinya dalam struktur data yang nantinya dapat ditangani oleh pemroses bahasa-bahasa pemrograman. Tidak semua bahasa dapat mengimplementasi semua struktur fisik yang diuraikan di sini.

Untuk mendapatkan algoritma yang optimal, implementasi list linier dalam suatu struktur fisik harus disesuaikan dengan permasalahan dan juga ketersediaan bahasa. Pada umumnya aturan penulisan dan gambaran dari struktur fisik tidak mudah dipahami manusia. Maka pada tahapan analisis, yang harus dipakai sebagai pegangan adalah struktur lojik, karena mempermudah penganalisisan masalah dan pengembangan algoritma.

Setelah suatu struktur fisik dipilih, algoritma dalam struktur lojik yang telah dibahas tinggal diterjemahkan ke dalam struktur fisik yang dipilih.

Representasi fisik dari sebuah list linear dapat BERKAIT atau KONTIGU. Representasi BERKAIT dapat diimplementasi dalam dua macam struktur fisik yaitu pointer dan tabel, sedangkan representasi KONTIGU hanya dapat diimplementasi dalam struktur yang secara fisik kontigu yaitu tabel.

Representasi Berkait

Representasi Fisik List Linier BERKAIT dengan "Pointer"

```
KAMUS
{ List direpresentasi dg pointer }
   type InfoType : ... { terdefinisi }
   type ElmtList : < Info : InfoType, Next : address >
   Address : pointer to ElmtList
   type List : address
   First : List { Alamat elemen pertama list }

{ Deklarasi  nama untuk variabel  kerja }
   P : address  {address untuk traversal }
{ Maka penulisan First(L) menjadi First
                 Next(P) menjadi P↑.Next
                 Info(P) menjadi P↑.Info }
```

Contoh representasi fisik list linier sederhana yang "berkait" dengan pointer adalah:

```
KAMUS
{ List direpresentasi dengan pointer }
   type InfoType : ... { terdefinisi }
   type ElmtList : < Info : InfoType, Next : address >
   Address : pointer to ElmtList
   type List : < First : address >
   L : List  { Alamat elemen pertama list }

{ Deklarasi  nama untuk variabel kerja }
   P : address { address untuk traversal }
{ Maka penulisan First(L) menjadi L.First
                 Next(P) menjadi P↑.Next
                 Info(P) menjadi P↑.Info }
```

Representasi Fisik List Linier BERKAIT dengan Tabel

Jika bahasa permograman tidak menyediakan struktur pointer (dengan primitif Allocate, Free, Saturate), maka kita dapat melakukan implementasi fisik alamat dengan indeks tabel. Jadi, pengelolaan "memori" yang dilakukan oleh sistem dengan alamat memori yang sebenarnya sekarang diacu melalui nama tabel. Dengan demikian, kita harus mendefinisikan suatu tabel GLOBAL, yang setiap elemennya adalah elemen list yang diacu oleh alamat. Dalam hal ini, pemrogram melakukan simulasi pemakaian memori dan merealisasikan alokasi/dealokasi elemen sebagai bagian dari primitif struktur data list. Namun karena ukuran tabel harus didefiniskan pada kamus, pada dasarnya alokasi memori yang dilakukan adalah statis.

```
KAMUS
{ List direpresentasi secara berkait dengan tabel }
  type InfoType : ... { terdefinisi }
  type ElmtList : < Info : InfoType, Next : address >
  type Address  : integer [IndexMin..IndexMax, Nil]
{ TABEL MEMORI LIST, GLOBAL}
  constant IndexMin : integer = 1
  constant IndexMax : integer = 100
  constant Nil : integer = 0
  { Nil : address tak terdefinisi, di luar [IndexMin..IndekMax] }
  TabElmt : array [IndexMin..IndexMax] of ElmtList
  FirstAvail : Address { alamat pertama list siap pakai }
  { Maka penulisan First(L) menjadi L.First
                   Next(P) menjadi TabElmtList(P).Next
                   Info(P) menjadi TabElmtList(P).Info }
  function MemFull
  { Mengirim true jika memori list sudah "habis" : FirstAvail=Nil }
  procedure InitTab
  { Inisialisasi tabel yang akan dipakai sebagai memori list }
  { I.S. Sembarang. }
  { F.S. TabElmt[IndexMin..IndexMax] siap dipakai sebagai elemen list
    berkait, Elemen pertama yang available adalah FirsAvail=1.
    Next(i)=i+1 untuk i[IndexMin..IndexMax-1], Next(IndexMax)=Nil }
  procedure AllocTab (output P : address)
  { Mengambil sebuah elemen siap pakai P  pada awal list FirstAvail }
  { I.S. FirstAvail mungkin kosong. }
  { F.S. Jika  FirstAvail tidak Nil, P adalah  FirstAvail dan
         FirstAvail yang baru adalah  Next(FirstAvail) }
  {      Jika FirstAvail =Nil, P=Nil,
         tulis pesan 'Tidak tersedia lagi elemen siap pakai }
  procedure DeAllocTab (input P : address)
  { Mengembalikan sebuah elemen  P  pada awal list FirstAvail}
  { I.S. FirstAvail mungkin kosong. P tidak Nil }
  { F.S. FirstAvail = P }
```

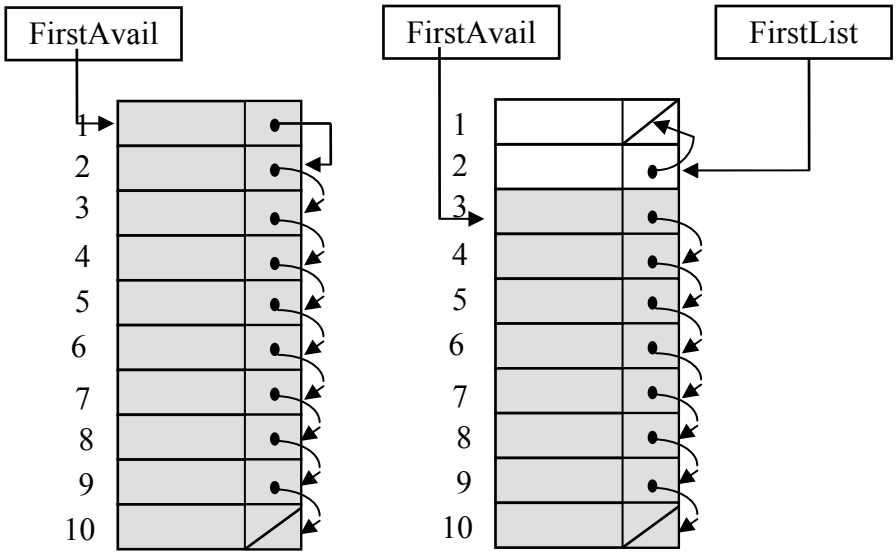
```
function MemFull
{ Mengirimkan true jika Memori sudah "habis": FirstAval=Nil }

KAMUS LOKAL

ALGORITMA
    → (FirstAvail = Nil)
```

```
procedure InitTab
{ Inisialisasi tabel yang akan dipakai sebagai memori list }
{ I.S. Sembarang }
{ F.S. TabElmt[IndexMin..IndekMax] terinisialiasi untuk dipakai sebagai
  elemen list berkait, elemen pertama yang available adalah
  FirstAvail=1.
  Next(i)=i+1 untuk i[IndexMin..IndexMax-1], Next(IndexMax)=Nil }

KAMUS LOKAL
    P : address

ALGORITMA
    P traversal [IndexMin..IndeksMax-1]
        TabElmt(P).Next ← P + 1
    TabElmt(IndexMax).Next ← Nil
    FirstAvail ← IndexMin
```

```
procedure AllocTab (output P : address)
{ Mengambil sebuah elemen siap pakai P  pada awal list FirstAvail }
{ I.S. FirstAvail mungkin kosong. }
{ F.S. Jika FirstAvail tidak Nil, P adalah FirstAvail
       dan FirstAvail yang baru adalah Next(FirstAvail) }
{       Jika FirstAvail=Nil, tuliskan pesan 'Tidak tersedia lagi elemen siap
       pakai', P=Nil }

KAMUS LOKAL

ALGORITMA
    if (not MemFull) then
        P ← TabElmt(FirstAvail)
        FirstAvail ← TabElmt(FirstAvail).Next
    else
        output ("Tidak tersedia lagi elemen siap pakai")
        P ← Nil
```

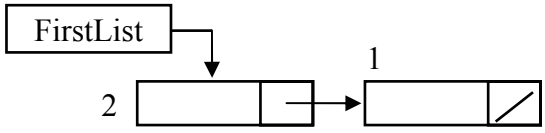
```
procedure DeallocTab (input P : address)
{ Mengembalikan sebuah elemen  P  pada awal list FirstAvail }
{ I.S. FirstAvail mungkin kosong. P ≠ Nil }
{ F.S. FirstAvail = P }

KAMUS LOKAL

ALGORITMA
    TabElmt(P).Next ← FirstAvail
    FirstAvail ← P
```

Ilustrasi dari urut-urutan pemanggilan dan status pemakaian memori list tersebut dapat dilihat pada gambar berikut. Perhatikan bahwa satu tabel memori yang sama dapat dipakai oleh satu atau beberapa list, jika elemen listnya sama. Satu elemen tabel pada suatu saat hanya dapat menjadi elemen sebuah list tertentu, namun pada saat lain dapat menjadi elemen list manapun. Satu elemen dapat dipakai secara "bergiliran" oleh beberapa list.

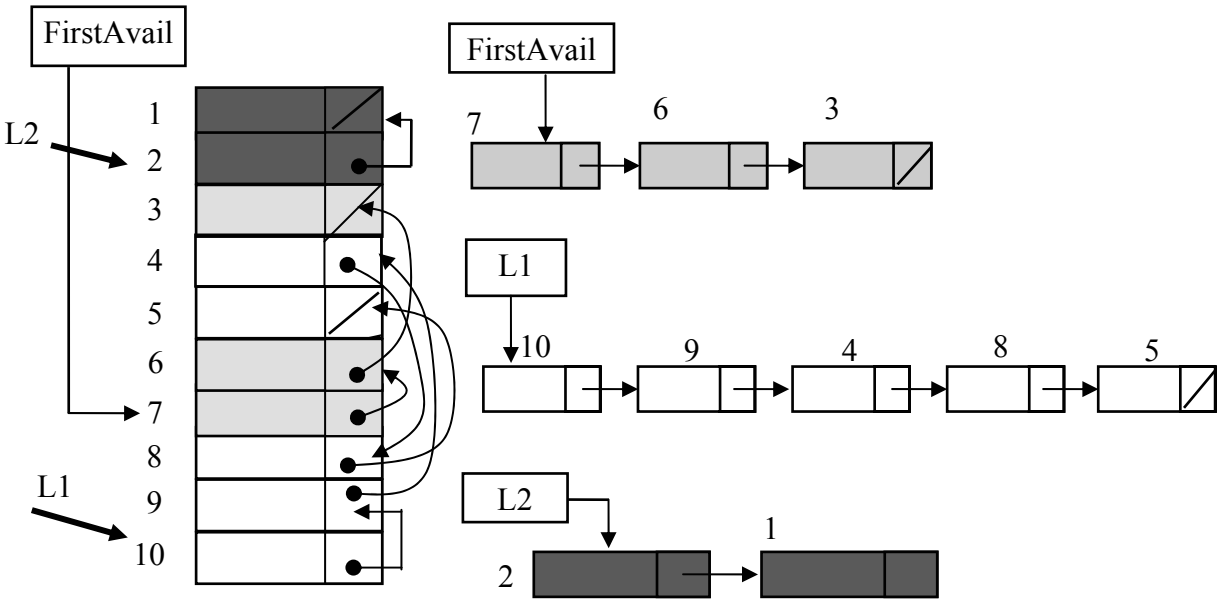


Keadaan Awal

List Terpakai 2 Elemen



Beberapa contoh keadaan list :



Perhatikan bahwa representasi seperti tabel di sebelah kiri sangat "kusut". Untuk selanjutnya, dipakai representasi lojik seperti di kanan karena lebih jelas.

Representasi Fisik List Linier secara KONTIGU

Dengan reprerentasi fisik kontigu, satu-satunya struktur data yang dapat menyimpannya adalah tabel, karena hanya tabel yang mempunyai struktur kontigu. Setiap elemen tabel mengandung informasi info, sedangkan informasi mengenai Next tidak perlu lagi disimpan secara eksplisit, karena secara implisit sudah tersirat dalam struktur data yang menjadi tempat penyimpanannya.

Elemen terakhir tidak mungkin dikenali dari NEXT, karena NEXT tidak disimpan secara eksplisit. Satu-satunya cara untuk mengetahui elemen terakhir adalah dari alamatnya : P=N, dengan N adalah lokasi pada tabel tempat menyimpan elemen terakhir. Karena alamat elemen terakhir harus diketahui secara eksplisit, maka representasi list bukan murni seperti di atas, tetapi harus mengandung First(L) dan Last(L), seperti yang pernah dibahas pada Queue. Lihat bab variasi representasi list.

Representasi fisik list linier secara kontigu dengan TABEL adalah:

```
{ List direpresentasi pada tabel secara kontigu}
KAMUS
    constant IndexMin : integer  = 1
    constant IndexMax : integer = 100
    constant Nil : integer = 0
    type InfoType : ... { Elemen Type : terdefinisi }
    Info : InfoType { tidak perlu mengandung Next, karena dapat
                      dikalkulasi }
    TabElmtList : array [IndexMin..IndexMax] of Info
    type Address : integer [IndexMin..IndexMax, Nil]

{ Deklarasi  nama untuk variabel  kerja }
    N : address  { alamat elemen terakhir. Karena field NEXT tidak ada secara
eksplisit, maka satu-satunya jalan untuk mengenali elemen terakhir adalah dengan
@-nya}
    type List : address
    First : List
{ Deklarasi alamat }
    P : address  {address untuk traversal }
{ Maka First(L)..Last(L) adalah indeks efektif elemen tabel anggota list
       Next(P) P ← P + 1   Next(P) tidak terdefinisi untuk P = N }
       Info(P) menjadi TabElmtList(P).Info }
```

Pemilihan representasi fisik dari list linier akan sangat mempengaruhi performansi dari algoritma. Pada bagian selanjutnya, akan dipelajari (melalui kasus-kasus), kapan masing-masing representasi cocok untuk dipakai.

Ringkasan Representasi List

Representasi Berkait :

Representasi lojik Berkait	Representasi fisik berkait dengan Pointer	Representasi fisik berkait dengan Tabel
KAMUS UMUM: L : List P : address {kamus belum terdef.}	KAMUS UMUM: type infotype:{terdef} type address: pointer to Elmt type ElmtList: < info:infotype, Next:address> type List:{rep.terdef} L : List P : address	KAMUS UMUM: type infotype:{terdef} type address:integer type ElmtList: < info:infotype, Next:address > {variabel GLOBAL} constant Nil=0 constant NMAx:address=... TabMem:array[Nil..NMax] of ElmtList FirstAvail:address type List:{rep.terdef} L : List P : address
AKSES: First(L) Next(P) Info(P)	AKSES: tergantung deklarasi P↑.Next P↑.Info	AKSES: tergantung deklarasi TabMem(P).Next TabMem(P).Info
PRIMITIF ALOKASI/DEALOKASI:	PRIMITIF ALOKASI/DEALOKASI: (tidak perlu realisasi, sistem)	PRIMITIF ALOKASI/DEALOKASI: (harus direalisasi) MemFull InitTab AllocTab(P) DeallocTab(P)
Pengenal elemen terakhir	Next(P) = Nil	

Representasi Kontigu :

Representasi lojik kontigu	**Representasi fisik kontigu dengan tabel**
KAMUS UMUM: L : List P : address	KAMUS UMUM: type address : integer type ElmtList : < info : infotype > constant First : address = ... constant Last : address = ... type List : < TabMem : array [First..Last] of ElmtList > L : List P : address
AKSES: First(L) Next(P) Info(P)	AKSES: tergantung deklarasi P ← P + 1 TabELmt(P).Info
PRIMITIF ALOKASI/DEALOKASI:	PRIMITIF ALOKASI/DEALOKASI: { tidak perlu dilakukan, pada saat pendefinisian tabel, secara statis sudah ditentukan. Kecuali jika deklarasi tabel secara dinamis seperti dalam bahasa C }
Pengenal elemen terakhir	Last

Latihan Soal

1. Terjemahkanlah semua skema pada list linier yang direpresentasi pada bab sebelumnya dengan:
   - pointer
   - tabel
   - kontigu

   Perhatikan ada 14 skema x 3 representasi fisik yang harus Anda buat sebagai latihan. Berarti ada 42 latihan soal.

2. Apa bedanya primitif Allocate(P) dan Free(P) pada representasi berkait dengan pointer dengan AllocateTab(P) dan DeallocTab(P) pada representasi berkait dengan tabel?

3. Jika definisi List linier dan elemennya secara logik adalah sebagai berikut:

```
{ InfoType dan address adalah suatu type yang telah terdefinisi }
   type ElmtList : < Info : InfoType, Next : address >
   type List : < First : address > { First adalah alamat elemen pertama list }
   L : List
{ Deklarasi  nama untuk variabel  kerja }
   P : address { address untuk traversal }
{ Maka penulisan First(L) menjadi ...
                 Next(P) menjadi ...
                 Info(P) menjadi ...  }
```

   Maka :
   - apa keuntungan yang diperoleh dengan definisi ini dibandingkan dengan yang pernah dibahas?
   - definisikan dan tuliskan kembali algoritma dasar untuk operasi list untuk setiap representasi fisiknya

4. Jika definisi List linier dan elemennya secara logik adalah sebagai berikut:

```
{ InfoType dan address adalah suatu type yang telah terdefinisi }
   type ElmtList : < Info : InfoType, Next : address >
   type List : < First:address,  P : address >
          { First adalah alamat elemen pertama list }
          { P adalah alamat elemen yang sedang 'current'}
   L : List

{ Maka penulisan First(L) menjadi ...
                Next(P) menjadi ...
                Info(P) menjadi ...    }
```

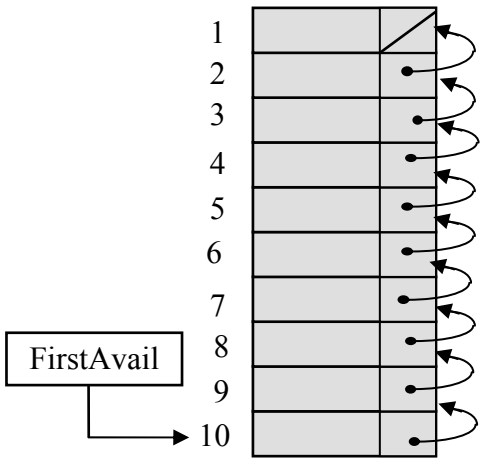
   Maka :
   - apa keuntungan yang diperoleh dengan definisi ini dibandingkan dengan pada soal 3)?
   - definisikan dan tuliskan kembali algoritma dasar untuk operasi list untuk setiap representasi fisiknya

5. Jika informasi elemen list disimpan pada suatu struktur lain, dan informasi yang tersimpan pada elemen list hanya berupa alamat dari struktur penyimpan informasi tersebut, maka definisi List linier dan elemennya secara logik adalah sebagai berikut:

```
{ InfoType  adalah suatu type yang telah terdefinisi , Address adalah alamat
mesin yang dapat mengacu ke semua type }
    type AdrElmtList : address
    type ElmtList :  < Info : AdrElmtList, Next : address >
    type List : < First : address, P : address>
                { First adalah alamat elemen pertama list }
                { P adalah alamat elemen yang sedang 'current' }
    L : List

{ Maka penulisan First(L) menjadi ...
                 Next(P) menjadi ...
                 Info(P) menjadi ...     }
```

Maka :
- berikan ilustrasi struktur ini
- apa keuntungan lebih yang diperoleh dengan definisi ini dibandingkan dengan pada soal 4)?
- definisikan dan tuliskan kembali algoritma dasar untuk operasi list untuk setiap representasi fisiknya

6. Buatlah algoritma InitTab untuk menginisialisasi tabel penyimpan elemen list dengan spesifikasi Final State seperti gambar berikut :



Keadaan Akhir

Adakah alternatif yang lain? Buatlah spesifikasinya dan tuliskanlah algoritmanya.

7. Untuk semua kamus kemungkinan definisi list di atas
   Tuliskanlah algoritma dengan implementasi fisik alamat adalah pointer, untuk:
   a. "menyalin" sebuah List L1 menjadi list L2.
   b. "membalik" sebuah list.
      Membalik sebuah list adalah membalik urutan elemen list.
      Contoh : jika list asal adalah 1,2,3 maka list hasil adalah 3,2,1.
      Jika list asal adalah 8,9,5,2 maka list hasil adalah 2,5,9,8
      List yang semula tidak perlu dipertahankan (tidak perlu alokasi elemen baru)
   c. Memusnahkan list, yaitu membuat sebuah list kosong, dan semua memori yang dipakai untuk elemen list dikembalikan ke sistem

8. Tuliskanlah realisasi dari prosedur penambahan berikut jika diimplementasi dengan pointer dan tabel berkait:

```
procedure  Insert-First (input/output L : List, input X : InfoType)
{ I.S. List L mungkin kosong }
{ F.S. P dialokasi, sebuah adalah elemen pertama list L, beralamat P dengan
Info(P)=X }
{ Insert sebuah elemen beralamat P dengan Info(P)=X sebagai elemen pertama dari
list linier L yg mungkin kosong }
```

```
procedure Insert-After (input/output L : List, input XPrec, XP : InfoType)
{ I.S. List L mungkin kosong }
{ F.S. P dialokasi, sebuah adalah elemen pertama list L, beralamat P dengan
Info(P)=XP }
{ Insert sebuah elemen beralamat P dengan Info(P)=XP sebagai elemen sesudah
elemen list beralamat Prec dengan info(Prec)=XPrec dari list linier L yg mungkin
kosong }
```

```
procedure  Insert-Last (input/output L : List, input X : InfoType)
{ I.S. List L mungkin kosong }
{ F.S. P dialokasi, sebuah adalah elemen terakhir list L, beralamat P dengan
Info(P)=X }
{ Insert sebuah elemen beralamat P dengan Info(P)=X sebagai elemen terakhir
dari list linier L yg mungkin kosong }
```

9. Tuliskanlah realisasi dari prosedur penghapusan elemen list sebagai berikut jika diimplementasi dengan pointer dan tabel berkait:

```
procedure  Delete-First (input/output L : List, output X : InfoType)
{ I.S. List L tidak  kosong }
{ F.S. Elemen pertama list L dihapus, dan didealokasi. Hasil penghapusan diimpan
nilainya dalam X.
List mungkin menjadi kosong. Jika tidak kosong, elemen pertama list yang baru
adalah elemen sesudah elemen pertama yang lama. }
```

```
procedure  Delete-Last (input/output L : List, output X : InfoType)
{ I.S. List L tidak  kosong }
{ F.S. Elemen terakhir list L dihapus, dan didealokasi. Hasil penghapusan
diimpan nilainya dalam X.
List mungkin menjadi kosong. Jika tidak kosong, elemen pertama list yang baru
adalah elemen sesudah elemen pertama yang lama. }
```

```
procedure  Delete-After (input/output L : List, output X : InfoType)
{ I.S. List L tidak  kosong }
{ F.S. Elemen list L yang bernilai X dihapus, dan didealokasi.
List mungkin menjadi kosong. }
```

ADT List Linier dengan Representasi Berkait (Pointer) dalam Bahasa C

```
/* File : list1.h */
/* contoh ADT list berkait dengan representasi fisik pointer  */
/* Representasi address dengan pointer */
/* infotype adalah integer */
#ifndef list1_H
#define list1_H

#include "boolean.h"

#define Nil NULL
#define info(P) (P)->info
#define next(P) (P)->next
#define First(L) ((L).First)
typedef int infotype;
typedef struct tElmtlist *address;
typedef struct tElmtlist { infotype info;
                           address next;
                         } ElmtList;
/* Definisi list : */
/* List kosong : First(L) = Nil */
/* Setiap elemen dengan address P dapat diacu info(P), Next(P) */
/* Elemen terakhir list : jika addressnya Last, maka Next(Last)=Nil */
typedef struct { address First;
               } List;

/* PROTOTYPE */
/****************** TEST LIST KOSONG ******************/
boolean ListEmpty (List L);
/* Mengirim true jika list kosong */

/****************** PEMBUATAN LIST KOSONG ******************/
void CreateList (List * L);
/* I.S. sembarang             */
/* F.S. Terbentuk list kosong */

/****************** Manajemen Memori ******************/
address Alokasi (infotype X);
/* Mengirimkan address hasil alokasi sebuah elemen */
/* Jika alokasi berhasil, maka address tidak nil, dan misalnya */
/* menghasilkan P, maka info(P)=X, Next(P)=Nil */
/* Jika alokasi gagal, mengirimkan Nil */
void Dealokasi (address P);
/* I.S. P terdefinisi */
/* F.S. P dikembalikan ke sistem */
/* Melakukan dealokasi/pengembalian address P */

/****************** PENCARIAN SEBUAH ELEMEN LIST ******************/
address Search (List L, infotype X);
/* Mencari apakah ada elemen list dengan info(P)= X */
/* Jika ada, mengirimkan address elemen tersebut. */
/* Jika tidak ada, mengirimkan Nil */
boolean FSearch (List L, address P);
/* Mencari apakah ada elemen list yang beralamat P */
/* Mengirimkan true jika ada, false jika tidak ada */
address SearchPrec (List L, infotype X, address *Prec);
/* Mengirimkan address elemen sebelum elemen yang nilainya=X */
/* Mencari apakah ada elemen list dengan info(P)= X */
/* Jika ada, mengirimkan address Prec, dengan Next(Prec)=P */
/* dan Info(P)=X. */
/* Jika tidak ada, mengirimkan Nil */
/* Jika P adalah elemen pertama, maka Prec=Nil */
/* Search dengan spesifikasi seperti ini menghindari */
/* traversal ulang jika setelah Search akan dilakukan operasi lain*/
```

```
/****************** PRIMITIF BERDASARKAN NILAI ******************/
/*** PENAMBAHAN ELEMEN ***/
void InsVFirst (List * L, infotype X);
/* I.S. L mungkin kosong */
/* F.S. Melakukan alokasi sebuah elemen dan */
/* menambahkan elemen pertama dengan nilai X jika alokasi berhasil */
void InsVLast (List * L, infotype X);
/* I.S. L mungkin kosong */
/* F.S. Melakukan alokasi sebuah elemen dan */
/* menambahkan elemen list di akhir: elemen terakhir yang baru */
/* bernilai X jika alokasi berhasil. Jika alokasi gagal: I.S.= F.S. */

/*** PENGHAPUSAN ELEMEN ***/
void DelVFirst (List * L, infotype * X);
/* I.S. List L tidak kosong  */
/* F.S. Elemen pertama list dihapus: nilai info disimpan pada X */
/*      dan alamat elemen pertama di-dealokasi */
void DelVLast (List * L, infotype * X);
/* I.S. list tidak kosong */
/* F.S. Elemen terakhir list dihapus: nilai info disimpan pada X */
/*      dan alamat elemen terakhir di-dealokasi */

/****************** PRIMITIF BERDASARKAN ALAMAT ******************/
/*** PENAMBAHAN ELEMEN BERDASARKAN ALAMAT ***/
void InsertFirst (List * L, address P);
/* I.S. Sembarang, P sudah dialokasi  */
/* F.S. Menambahkan elemen ber-address P sebagai elemen pertama */
void InsertAfter (List * L, address P, address Prec);
/* I.S. Prec pastilah elemen list dan bukan elemen terakhir, */
/*      P sudah dialokasi  */
/* F.S. Insert P sebagai elemen sesudah elemen beralamat Prec */
void InsertLast (List * L, address P);
/* I.S. Sembarang, P sudah dialokasi  */
/* F.S. P ditambahkan sebagai elemen terakhir yang baru */

/*** PENGHAPUSAN SEBUAH ELEMEN ***/
void DelFirst (List * L, address * P);
/* I.S. List tidak kosong */
/* F.S. P adalah alamat elemen pertama list sebelum penghapusan */
/*      Elemen list berkurang satu (mungkin menjadi kosong) */
/* First element yg baru adalah suksesor elemen pertama yang lama */
void DelP (List * L, infotype X);
/* I.S. Sembarang */
/* F.S. Jika ada elemen list beraddress P, dengan info(P)=X  */
/* Maka P dihapus dari list dan di-dealokasi */
/* Jika tidak ada elemen list dengan info(P)=X, maka list tetap */
/* List mungkin menjadi kosong karena penghapusan */
void DelLast (List * L, address * P);
/* I.S. List tidak kosong */
/* F.S. P adalah alamat elemen terakhir list sebelum penghapusan  */
/*      Elemen list berkurang satu (mungkin menjadi kosong) */
/* Last element baru adalah predesesor elemen pertama yg lama, */
/* jika ada */
void DelAfter (List * L, address * Pdel, address Prec);
/* I.S. List tidak kosong. Prec adalah anggota list  */
/* F.S. Menghapus Next(Prec): */
/*      Pdel adalah alamat elemen list yang dihapus  */

/****************** PROSES SEMUA ELEMEN LIST ******************/
void Printinfo (List L);
/* I.S. List mungkin kosong */
/* F.S. Jika list tidak kosong, */
/* Semua info yg disimpan pada elemen list diprint */
/* Jika list kosong, hanya menuliskan "list kosong" */
int NbElmt(List L);
/* Mengirimkan banyaknya elemen list; mengirimkan 0 jika list kosong */
```

```
/*** Prekondisi untuk Max/Min/rata-rata : List tidak kosong ***/
infotype Max (List L);
/* Mengirimkan nilai info(P) yang maksimum */
address AdrMax (List L);
/* Mengirimkan address P, dengan info(P) yang bernilai maksimum */
infotype Min (List L);
/* Mengirimkan nilai info(P) yang minimum */
address AdrMin (List L);
/* Mengirimkan address P, dengan info(P) yang bernilai minimum */
infotype Average (List L);
/* Mengirimkan nilai rata-rata info(P) */

/****************** PROSES TERHADAP LIST ******************/
void DelAll (List *L);
/* Delete semua elemen list dan alamat elemen di-dealokasi */

void InversList (List *L);
/* I.S. sembarang. */
/* F.S. elemen list dibalik : */
/* Elemen terakhir menjadi elemen pertama, dan seterusnya. */
/* Membalik elemen list, tanpa melakukan alokasi/dealokasi. */

List FInversList (List L);
/* Mengirimkan list baru, hasil invers dari L */
/* dengan menyalin semua elemn list. Alokasi mungkin gagal. */
/* Jika alokasi gagal, hasilnya list kosong */
/* dan semua elemen yang terlanjur di-alokasi, harus didealokasi */

void CopyList (List* L1, List * L2);
/* I.S. L1 sembarang. F.S. L2=L1 */
/* L1 dan L2 "menunjuk" kepada list yang sama.*/
/* Tidak ada alokasi/dealokasi elemen */

List FCopyList (List L );
/* Mengirimkan list yang merupakan salinan L */
/* dengan melakukan alokasi. */
/* Jika ada alokasi gagal, hasilnya list kosong dan */
/* semua elemen yang terlanjur di-alokasi, harus didealokasi */

void CpAlokList (List Lin, List * Lout);
/* I.S. Lin sembarang. */
/* F.S. Jika semua alokasi berhasil,maka Lout berisi hasil copy Lin */
/* Jika ada alokasi yang gagal, maka Lout=Nil dan semua elemen yang terlanjur
dialokasi, didealokasi */
/* dengan melakukan alokasi elemen. */
/* Lout adalah list kosong jika ada alokasi elemen yang gagal */

void Konkat (List L1, List L2, List * L3);
/* I.S. L1 dan L2 sembarang */
/* F.S. L1 dan L2 tetap, L3 adalah hasil konkatenasi L1 & L2 */
/* Jika semua alokasi berhasil, maka L3 adalah hasil konkatenasi*/
/* Jika ada alokasi yang gagal, semua elemen yang sudah dialokasi */
/* harus di-dealokasi dan L3=Nil*/
/* Konkatenasi dua buah list : L1 & L2 menghasilkan L3 yang "baru" */
/* Elemen L3 adalah hasil alokasi elemen yang "baru". */
/* Jika ada alokasi yang gagal, maka L3 harus bernilai Nil*/
/* dan semua elemen yang pernah dialokasi didealokasi */

void Konkat1 (List * L1, List * L2, List * L3);
/* I.S. L1 dan L2 sembarang */
/* F.S. L1 dan L2 kosong, L3 adalah hasil konkatenasi L1 & L2 */
/* Konkatenasi dua buah list : L1 dan L2    */
/* menghasilkan L3 yang baru (dengan elemen list L1 dan L2)*/
/* dan L1 serta L2 menjadi list kosong.*/
/* Tidak ada alokasi/dealokasi pada prosedur ini */


void PecahList (List *L1, List * L2, List L);
```

```
/* I.S. L mungkin kosong */
/* F.S. Berdasarkan L, dibentuk dua buah list L1 dan L2 */
/* L tidak berubah: untuk membentuk L1 dan L2 harus alokasi */
/* L1 berisi separuh elemen L dan L2 berisi sisa elemen L */
/* Jika elemen L ganjil, maka separuh adalah NbElmt(L) div 2 */
#endif
```

Latihan Soal

Latihan Bahasa C:

1. Buatlah sebuah modul list dengan representasi sama di atas, tetapi InfoType adalah float. Modifikasi apa yang harus dilakukan?
2. Bagaimana jika dibutuhkan misalnya modul list lain, yang intormasinya adalah string, karena list akan dipakai untuk menyimpan kata-kata yang akan dijadikan kata dalam sebuah kamus? Elemen list dalam hal ini harus terurut menurut abjad. Buatlah modul ini.

Latihan Bahasa Ada:

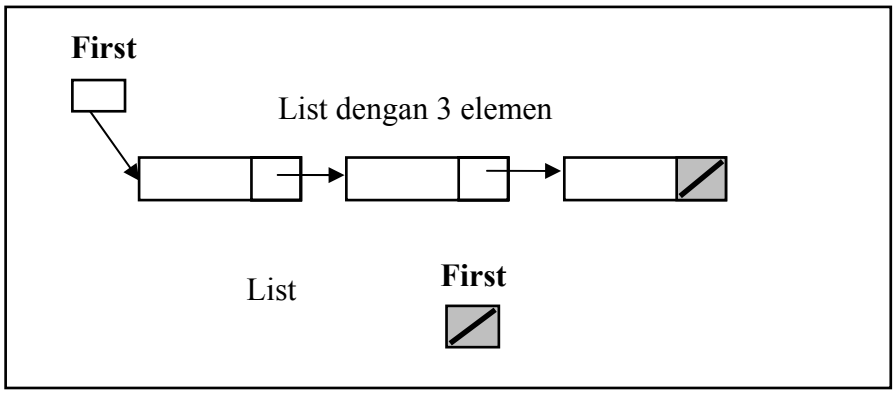
1. Terjemahkan modul ADT list1.h menjadi paket list. Pelajari baik-baik kesulitan yang timbul akibat batasan kompatibilitas *TYPE* yang sangat ketat dalam bahasa Ada.
2. Buatlah sebuah paket list dengan elemen generik, sehingga instansiasi infotype dapat dilakukan sesuai dengan kebutuhan.

Latihan Struktur Data:

Dalam Diktat “Pemrograman Fungsional”, didefinisikan struktur “Set” atau “Himpunan” berdasarkan list: Himpunan adalah sebuah list dengan elemen yang nilainya “unik”. Buatlah sebuah modul ADT Himpunan dalam bahasa C dan bahasa Ada, dengan list yang direpresentasi secara berkait dan kontigu. Definisikan semua operator himpunan yang mungkin dibutuhkan. Spesifikasi harus dibuat se-spesifik mungkin sesuai dengan implementasi bahasa.

Variasi Representasi List Linier

List linier biasanya direpresentasi sebagai berikut:



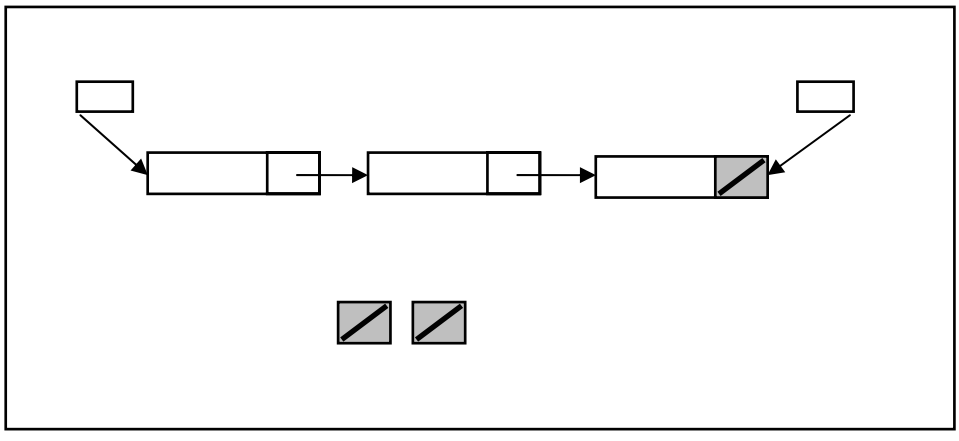
Ini adalah bentuk list linier yang paling dasar dan sederhana. List ini semacam ini mewakili banyak persoalan nyata, misalnya dapat mewakili *list of Windows, list of* item pada suatu menu, *list of object* pada suatu editor gambar, dan sebagainya. Jadi, elemen list adalah suatu objek yang sama.

List tersebut dapat direpresentasi dalam berbagi kamus seperti yang diuraikan pada latihan soal bab sebelumnya.

Jika L adalah sebuah list, dan P adalah alamat sebuah elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L) adalah sebuah elemen beralamat P,
dengan Next(P) = Nil
List kosong : First(L) = Nil

Untuk macam-macam keperluan, representasi lojik seperti yang digambarkan di atas dapat dibuat variasinya.

List Linier yang Dicatat Alamat Elemen Pertama dan Elemen Akhir

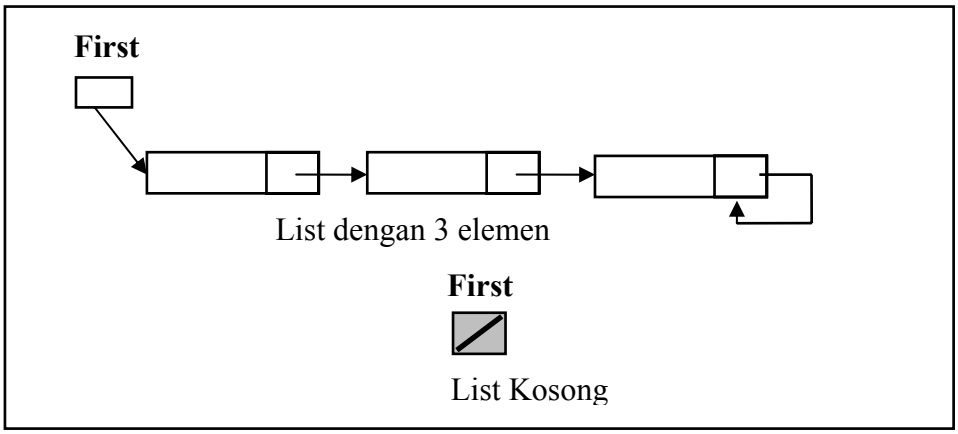


List ini dibutuhkan jika seringkali dilakukan operasi terhadap elemen terakhir, misalnya penambahan elemen yang selalu dilakukan di akhir. Fungsi dari LAST adalah untuk menghindari traversal untuk mencapai elemen terakhir. Jadi alamat elemen Last dimemorisasi. Konsekuensinya adalah setiap operasi yang menyangkut elemen terakhir harus memperhatikan apakah Last perlu diubah.

Jika L adalah sebuah list, dan P adalah alamat sebuah elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L)=P
List kosong : First(L) = Nil

List ini cocok untuk representasi Queue, dengan First adalah head, dan Last adalah Tail.

List yang Elemen Terakhir Menunjuk pada Diri Sendiri



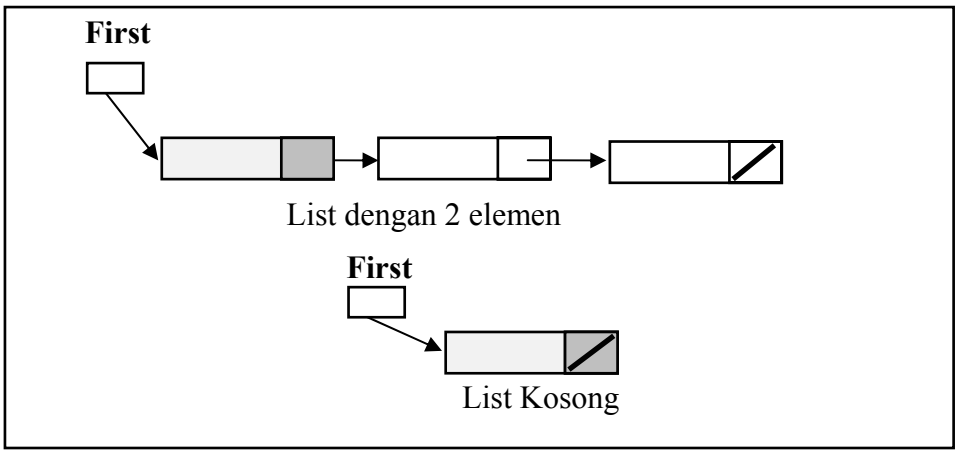
List dengan representasi ini dipilih jika tidak dikehendaki mendapatkan suatu alamat P yang bernilai Nil pada akhir traversal. Dengan representasi ini maka traversal selalu "terhenti" pada elemen terakhir. Operasi terhadap elemen terakhir (misalnya penambahan sebuah elemen menjadi element terakhir) tidak memerlukan memorisasi alamat elemen sebelumnya.

Jika L adalah sebuah list, dan P adalah alamat sebuah elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L) = P, degan Next(P)=P
List kosong : First(L) = Nil

List dengan Elemen Fiktif ("*Dummy Element*")

Elemen fiktif adalah elemen "*dummy*", yaitu elemen yang sengaja dialokasi untuk mempermudah oeprasi, namun sebenarnya bukan elemen list. List dengan elemen fiktif dibuat agar list kosong tidak berbeda dengan list biasa sehingga semua tes terhadap list kosong dapat dihapuskan.

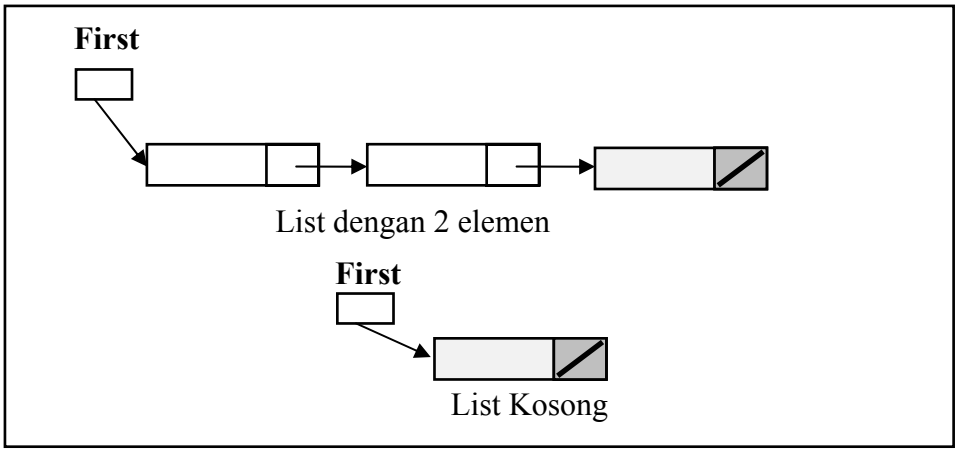
List dengan Elemen Fiktif adalah Elemen Pertama



Dengan elemen fiktif sebagai elemen pertama, maka insertLast pada list kosong menjadi sama dengan insertLast pada list biasa. First(L) tidak pernah Nil, melainkan selalu terdefinisi, pada saat CreateList.

Jika L adalah sebuah list, dan P adalah alamat sebuah elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : beraddress P, dengan Next(P)=dummy@
List kosong : First(L) = dummy@
dengan dummy@ yang terdefinisi pada saat list kosong dibuat

List dengan Elemen Fiktif sebagai Elemen Terakhir

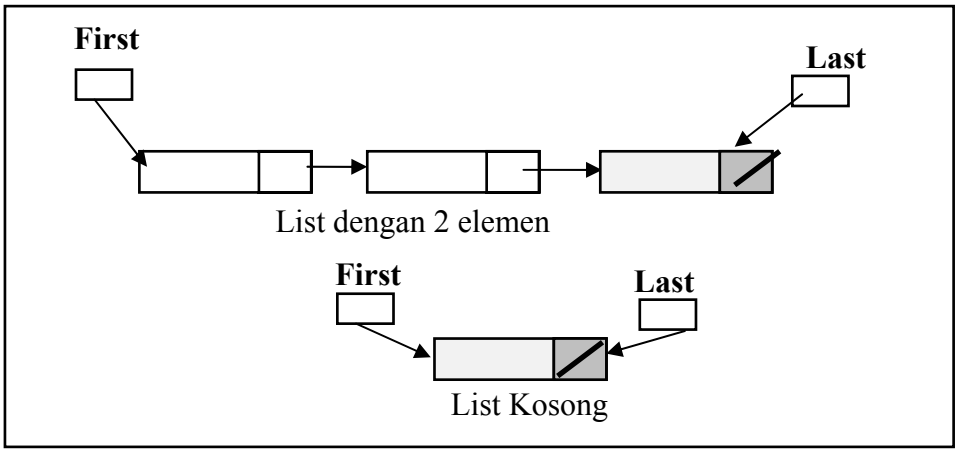


Dengan elemen fiktif sebagai elemen terakhir, maka First(L) tidak pernah Nil, melainkan selalu terdefinisi, pada saat CreateList. Elemen terakhir ini adalah "sentinel"

Jika L adalah sebuah list, dan P adalah alamat sebuah elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L) = dummy@
List kosong : First(L) = dummy@
dengan dummy@ yang terdefinisi pada saat list kosong dibuat

Dummy bisa berupa address yang tetap, bisa sebuah address yang berbeda (setiap kali *dummy* tersebut dipakai sebagai elemen list, dialokasi *dummy* yang baru). Representasi ini dipakai jika *dummy* dikehendaki sebagai "sentinel", apalagi jika dikombinasikan dengan pencatatan alamat *dummy* tersebut sebagai Last seperti pada 3.c.

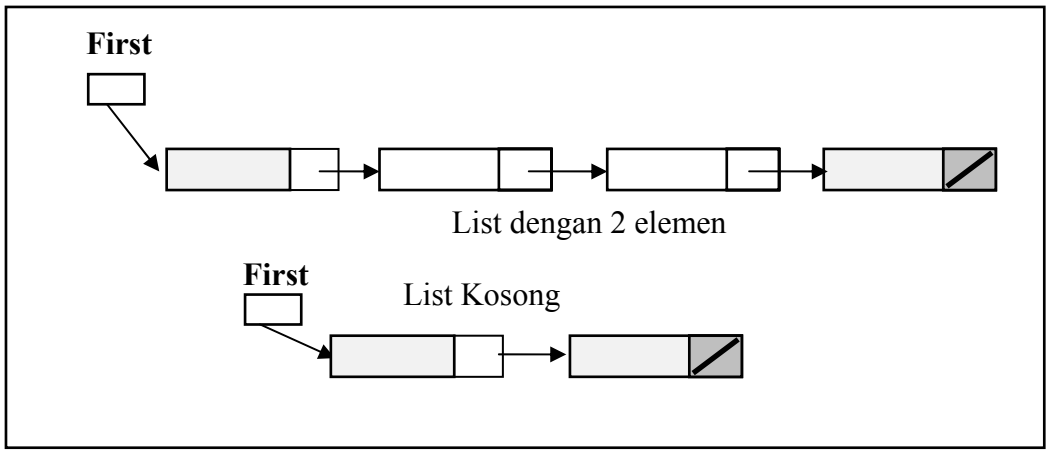
List dengan Elemen Fiktif di Akhir dan Pencatatan Alamat Elemen Akhir



Jika L adalah sebuah list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L), selalu berupa *dummy* elemen.
List kosong : First(L) = dummy@ =Last(L)
dengan dummy@ yang terdefinisi pada saat list kosong dibuat

Representasi ini seringkali dipakai jika *dummy* adalah sentinel, dan pencarian diperlukan sebelum penambahan elemen. Dengan representasi ini, nilai yang dicari dapat secara langsung disimpan untuk sementara pada *dummy*, kemudian dilakukan search. Jika search tidak berhasil, dan elemen akan ditambahkan, maka dialokasi sebuah *dummy* yang baru, nilai Last berubah. Contoh pemakaian sentinel untuk kombnasi search dan insert ini sangat efisien, dan dijelaskan pada topological sort.

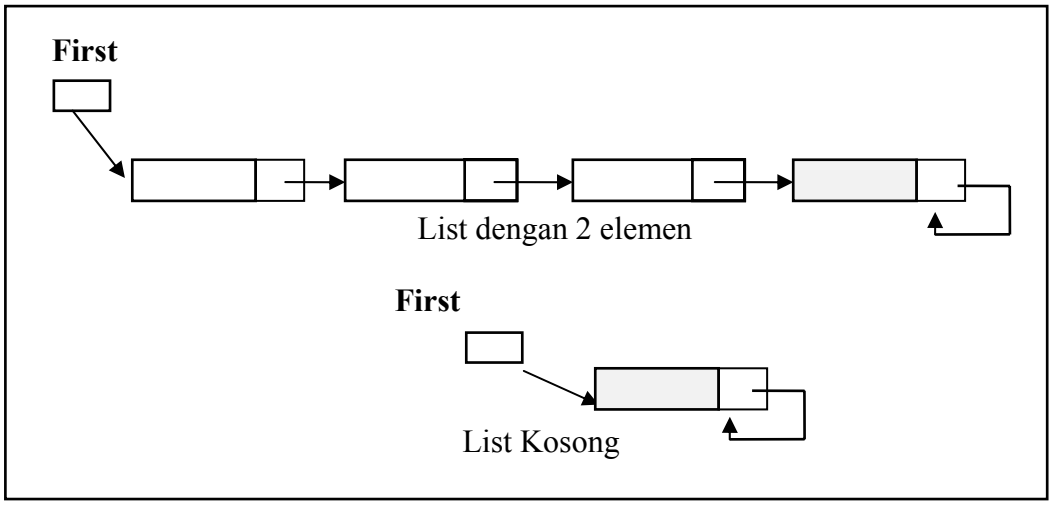
List dengan Elemen Fiktif sebagai Elemen Pertama dan Terakhir



List ini dipilih jika operasi penambahan dan penghapusan sebagai elemen pertama dan terakhir ingin dihindari. Dengan representasi semacam ini, semua operasi penambahan dan penghapusan menjadi operasi "di tengah" (After)

Jika L adalah sebuah list, dan P adalah alamat sebuah elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)= dummyFirst@
Elemen terakhir : Last(L) = dummyLast@
List kosong : First(L)=dummyFirst@
dengan dummyFirst@, dummyLast@ yang terdefinisi pada saat list kosong dibuat.

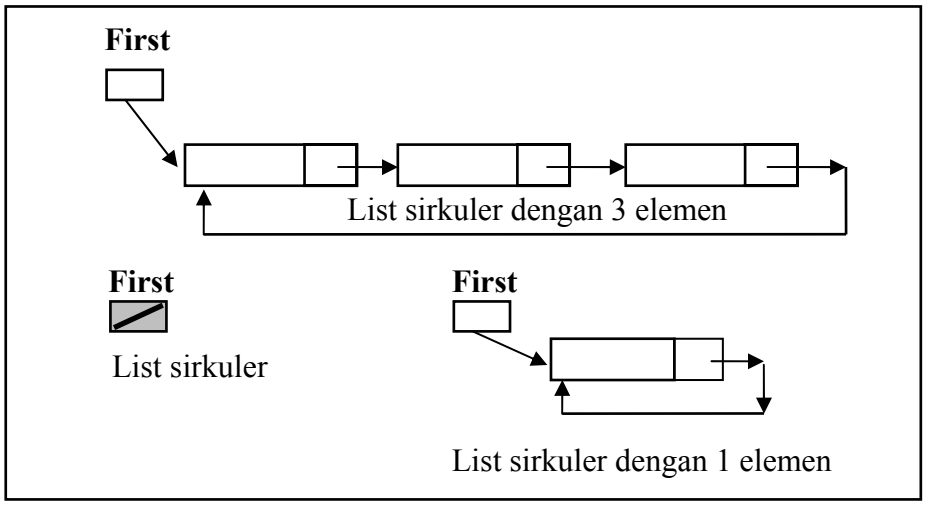
List dengan Elemen Fiktif dan Elemen Terakhir yang Menunjuk Diri Sendiri



List dengan representasi ini dipilih jika tidak dikehendaki mendapatkan suatu alamat P yang bernilai Nil pada akhir traversal. Dengan representasi ini, maka traversal "terhenti" pada elemen terakhir. Operasi terhadap elemen terakhir tidak memerlukan memorisasi elemen sebelumnya, misalnya jika ingindilakukan penambahan menjadi elemen terakhir.

Jika L adalah sebuah list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Jika P adalah alamat elemen terakhir, maka Next(P)=P.
Untuk elemen yang bukan terakhir : Next(Ptr) ≠ Ptr
List kosong : First(L) = Nil

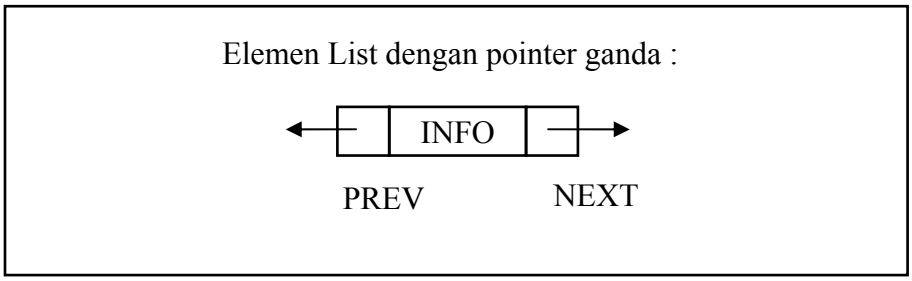
List Sirkuler

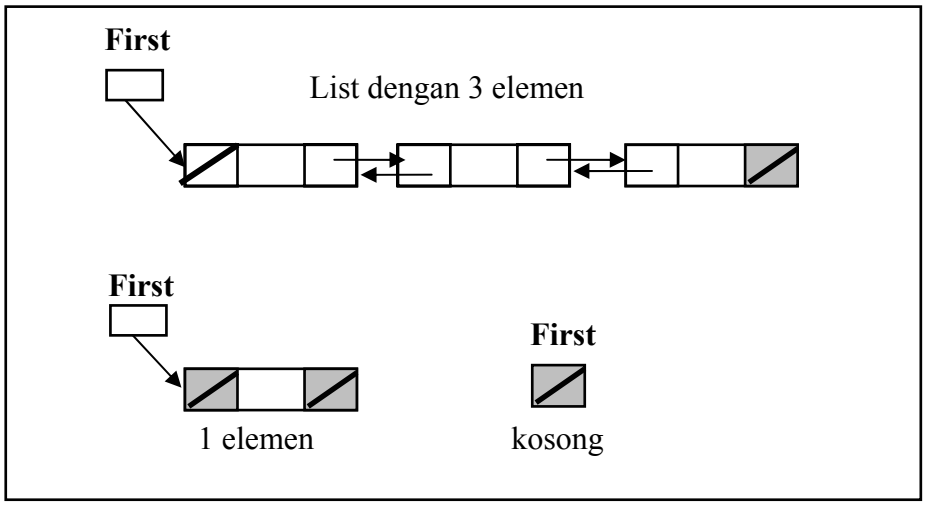


List dengan representasi ini sebenarnya tidak mempunyai "First". First adalah *"Current Pointer"*. Representasi ini dipakai jika dilakukan proses terus menerus terhadap anggota list (misalnya dalam *round robin services* pada sistem operasi). Seakan-akan terjadi suatu "loop". Perhatikan bahwa dengan representasi ini, penambahan dan penghapusan pada elemen pertama akan berakibat harus melakukan traversal untuk mengubah Next dari elemen Last.

Jika L adalah sebuah list, dan P adalah alamat elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)= P, dengan P adlah address salah satu elemen list
Elemen terakhir : Last(L)= P, Next(P)= First
List kosong : First(L) = Nil

List dengan Pointer Ganda





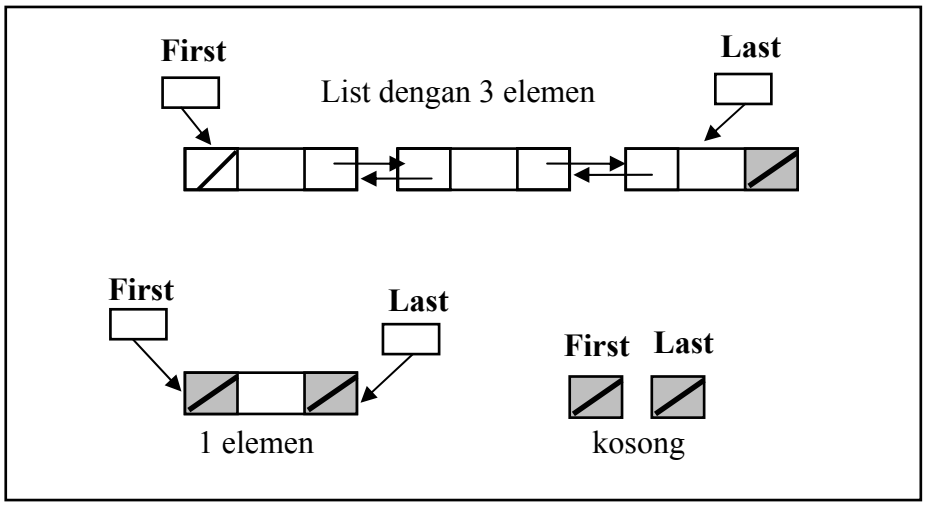
List dengan pointer ganda dibutuhkan jika harus dilakukan banyak operasi terhadap elemen suksesor dan juga predesesor. Dengan tersedianya alamat predesesor pada setiap elemen list, maka memorisasi Prec pada beberapa algoritma yang pernah ditulis dapat dihindari. Perhatikan bahwa dengan representasi ini:

- operasi dasar menjadi sangat "banyak" (buat sebagai latihan wajib)
- memori yang dibutuhkan membesar

Jika list lojik semacam ini direpresentasi secara kontigu dengan tabel, maka sangat menguntungkan karena memorisasi Prev dan Next dilakukan dengan kalkulasi.
List dengan pointer ganda pada contoh ini adalah list ber-pointer ganda yang paling sederhana. List bentuk lain dapat pula diimplementasi dengan pointer ganda seperti pada contoh-contoh yang berikutnya.

Jika L adalah sebuah list, dan P adalah alamat elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L)=P, dan Next(P)= Nil
List kosong : First(L) = Nil
Untuk sebuah address P yang merupakan address elemen list, maka didefinisikan
Next(P)
Prev(P)
Info(P)

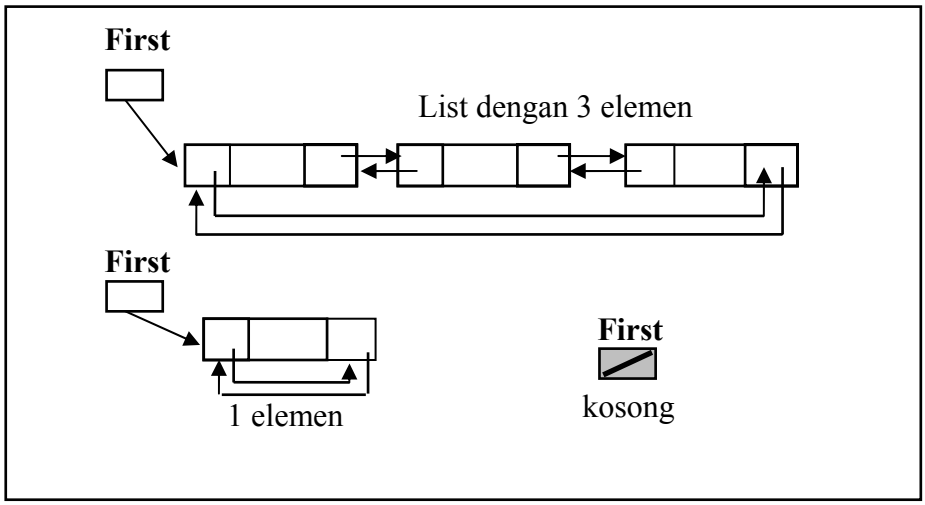
Representasi ini dapat diperluas dengan mencatat LAST:



Perhatikanlah bahwa jika representasi fisik dari list ini adalah kontigu, maka merupakan representasi dari ADT Queue yang pernah dipelajari!

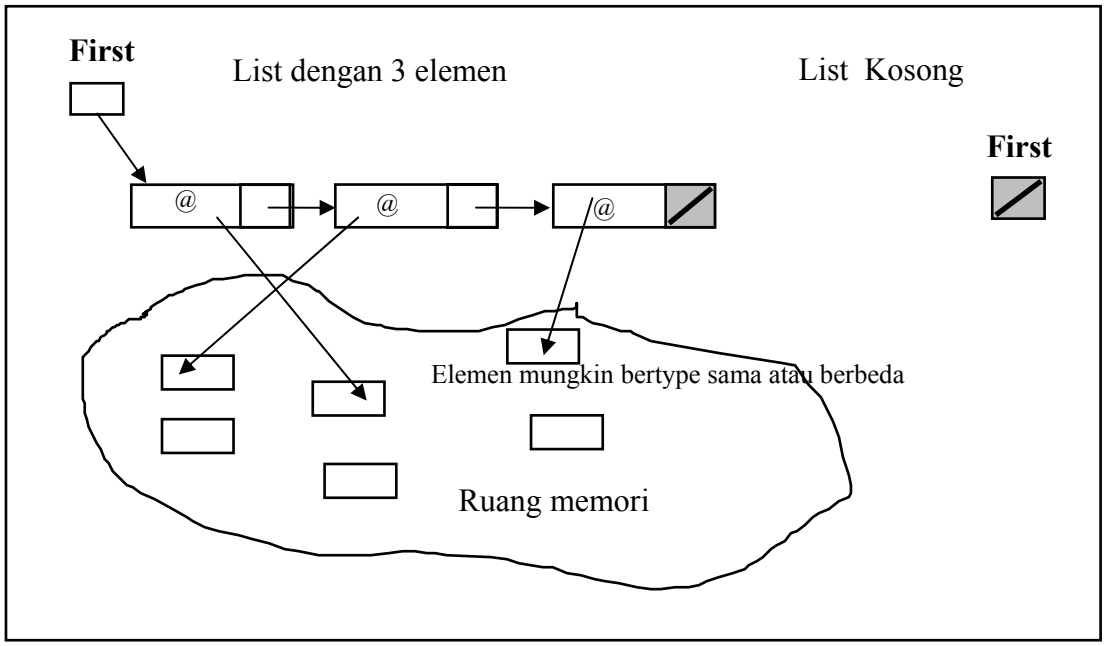
Jika L adalah sebuah list, dan P adalah alamat elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L)=P dan Next(P)= Nil
List kosong : First(L) = Last(P) = Nil

List dengan Pointer Ganda dan Sirkuler



Jika L adalah sebuah list, dan P adalah alamat elemen list, maka ciri dari list dengan representasi ini adalah :
Elemen pertama : First(L)
Elemen terakhir : Last(L)=P, Next(P) = First(L) dan Prev(First(L))= P
List kosong : First(L) = Nil

List yang Informasi Elemennya adalah Alamat dari Suatu Informasi



List dengan implementasi ini dapat menjadi dua kasus : @ menyimpan suatu pointer ke *type* yang sama, atau boleh ke *type* yang berbeda. List dengan representasi ini dibutuhkan jika pada saat didefinisikan, belum diketahui Info yang disimpan ber-*type* apa, dan baru dialokasi pada saat eksekusi.

Salah satu representasi list yang pernah digambarkan di atas dapat dipilih untuk list dengan definisi elemen list semacam ini.

Representasi ini mempunyai keuntungan dan kerugian. keuntungan yang diperoleh adalah info dapat menunjuk ke elemen ber-*type* "apapun", namun dengan demikian kita tidak memprogram "ketat" terhadap *type*.

Salah satu keuntungan dan pemakaian yang "bersih" dari list semacam ini adalah jika @ yang disimpan sebagai info dari elemen, sebenarnya adalah address dari elemen pertama list. Dalam hal ini, dipunyai list dengan elemen berupa list. Maka kita mempunyai struktur list yang tidak liner. Bahkan, jika definisi ini rekursif, maka dapat dipunyai sebuah struktur yang merepresentasi "S-expression" yang pernah dipelajari dalam Diktat "Pemrograman Fungsional"!

Jika bahasa pemrograman untuk melakukan implementasi list ini memungkinkan parametrisasi terhadap *type* (*type* generik), dan memungkinkan pula adanya bagian varian dari suatu *type* terstruktur, maka implementasi list ini menjadi jauh lebih mudah dan sederhana, karena kita tidak perlu menyimpan address, melainkan langsung memakai *type* yang ditentukan pada saat elemen list dibuat.
Jika elemen ber-*type* sama, maka ruang memori sendiri dapat dikelola sebagai list!

Latihan Soal

1. Carilah representasi lojik lain yang masih mungkin untuk list linier.
2. Saran untuk realisasi primitif pada variasi representasi list: ambillah algoritma yang pernah dibuat untuk list linier paling sederhana. Ubahlah dan sesuaikan untuk variasi representasi baru ini. Perubahan kode harus dilakukan secara sistematis dengan mengamati dan manganalisis perubahan representasi list.
3. Untuk masing-masing representasi lojik tersebut, tuliskan kamus untuk representasi fisik yang mungkin. kemudian tuliskanlah primitif sesuai dengan definisi operasi list (insert, delete dan konkatenasi) yang sesuai dengan representasi fisiknya. Berarti ada 42 algoritma x 9 variasi list! Jika setiap algoritma harus direalisasi dalam bahasa Pascal, Ada dan C, maka cukup banyak tugas mandiri yang dapat Anda lakukan sebagai latihan. Silakan bekerja.
4. Khusus untuk representasi terakhir, jika dipandang sebagai struktur yang rekursif, buatlah dengan algoritma rekursif jika sudah dibahas.

Studi Rekursif Dalam Konteks Prosedural

Pada bagian ini diberikan introduksi mengenai implementasi algoritma rekursif, dengan mengacu ke Diktat "Pemrograman Fungsional", sebagai introduksi pengolahan struktur data rekursif "sederhana", linier, sebelum mahasiswa mempelajari mengenai struktur data "Pohon".

Fungsi Faktorial

Berikut ini diberikan potongan program faktorial dalam berbagai sudut pandang implementasi, untuk memberikan pemahaman mengenai:

- implementasi fungsi yang merupakan terjemahan program fungsional rekursif dengan analisa rekurens
- implementasi fungsi yang merupakan terjemahan sebuah "loop" karena definisi rekursif di-*extend* menjadi sebuah deret, sehingga definisi faktorial ditransformasi menjadi perhitungan sebuah deret kali.

```
function factorial (N : integer) → integer
{ Mengirim N! sesuai dengan definisi faktorial definisi rekursif: }
{ 0!=1; 1!=1; N! = N* (N-1)! }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (N=0) then {Basis 0}
      → 0
   else {Rekurens}
      →  N *    factorial(N-1)
```

```
function factorial (N : integer) → integer
{ Mengirim N! sesuai dengan definisi faktorial: 1*1*2*3*4*...(N-1)*N }
{ Pada persoalan ini, definisi rekurens faktorial ditransformasi menjadi
perhitungan deret kali }
```

```
KAMUS LOKAL
   Count : integer
   F : integer
```

```
ALGORITMA
   F ← 1 {inisialisasi}
   Count ← 1    {first element}
   while (Count <=N) do
      F ← F * i          {Proses}
      Count ← Count + 1  {Next element}
   {Count > N, semua sudah dihitung}
   → F {Terminasi}
```

```
function factorial (N : integer) → integer
{ Mengirim N! sesuai dengan definisi faktorial: N*N-1*N-2*...*3*2*1 }
{ Mengubah parameter input: "kurang" baik" }
```

```
KAMUS LOKAL
   F : integer
```

```
ALGORITMA
   F ← 1 { inisialisasi }
   { first element }
   while (N > 0) do
      F ← F * N    { Proses }
      N ← N - 1    { Next element }
   { N=0 semua sudah dihitung }
   → F   { Terminasi }
```

```
procedure factorial (input N : integer, output F : integer)
{ I.S. N>0 }
{ F.S. F=N! }
{ Proses:menghasilkan N! dengan memanggil prosedur iterasi yang sesuai }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   FactIter1 (N,1,1,F)  { atau : factIter2(N,1,F ) }
```

```
procedure factIter1 (input N, Count : integer,
                     input/output Akumulator : integer, output F: integer)
{ I.S. N>0; Count:pencacah; Akumulator=Count! }
{ F.S. F=N! jika Count=N }
{ Proses : Mengirim N! sesuai dengan definisi faktorial: }
{ versi iteratif dengan mekanisme eksekusi rekursif }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (N=Count) then
      F ← Akumulator * N
   else
      factIter1(N, Count+1, Akumulator * Count, F)
```

```
procedure factIter2 (input N : integer, input/output Akumulator : integer,
                     output F : integer)
{ I.S. N>0, Akumulator=0 }
{ F.S. Menghasilkan F=N! }
{ Proses : Mengirim N! sesuai dengan definisi faktorial }
{ versi iteratif dengan mekanisme eksekusi rekursif }
```

```
KAMUS LOKAL
```

```
ALGORITMA
    if (N=0) then
        F ← Akumulator
    else
        factIter(N-1, Akumulator * N, F)
```

```
procedure factorialSalah1 (input N : integer, output F : integer)
{ I.S. N>=0 }
{ F.S. F=N! }
{ Proses : menghasilkan N! yang salah. Mengapa?? }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (N=0) then { Basis }
      F ← 1
   else { Rekurens }
      factorialSalah1(N-1, F)
```

```
procedure factorialSalah2 (input N : integer, output F : integer)
{ I.S. N>=0 }
{ F.S. F=N! }
{ Proses : menghasilkan N! yang salah. Mengapa?? }
```

```
KAMUS LOKAL
   Ftemp : integer
```

```
ALGORITMA
   if (N=0) then { Basis }
      F ← 1
   else {rekurens }
      factorialSalah2(N-1, Ftemp)
      F ← Ftemp
```

```
procedure factorialX (input N : integer, output F : integer)
{ I.S. N>=0 }
{ F.S. F=N! }
{ Proses : menghasilkan N! yang salah. Mengapa?? }
```

```
KAMUS LOKAL
   Ftemp : integer
```

```
ALGORITMA
   if (N=0) then { Basis }
      F ← 1
   else { Rekurens }
      factorialX(N-1, Ftemp)
   F ← Ftemp * N
```

Pemrosesan List Linier secara Prosedural dan Rekursif

Pada bagian ini diberikan contoh implementasi algoritma rekursif untuk memroses list linier, yang dipandang sebagai sebuah struktur data rekursif :

- list kosong adalah list
- list tidak kosong terdiri dari sebuah elemen, dan sisanya adalah list

Perhatikan bahwa implementasi dalam bahasa akan menimbulkan konflik *type* jika elemen pertama list dibedakan dari yang lain.

Implementasi dalam bahasa C untuk deklarasi list linier secara rekursif adalah sebagai berikut :

```
#ifndef list1_H
#define list1_H
#endif

#include "boolean.h"

#define Nil NULL
#define info(P) (P)->info
#define next(P) (P)->next

typedef int infotype;
typedef struct tElmtlist *address;
typedef struct tElmtlist
  { infotype info;
    address next;
  } ElmtList;
/* Definisi list : */
/* List kosong : First(L) = Nil */
/* Setiap elemen dengan address P dapat diacu info(P), Next(P) */
/* Elemen terakhir list : jika addressnya Last, maka Next(Last)=Nil*/
address List;

/* PROTOTYPE */
boolean IsEmpty (List L) ;
/* menghasilkan true jika list kosong */
```

Algoritma yang dituliskan berikut memakai deklarasi di atas:

```
procedure Printlist (input L : List)
{ I.S. L terdefinisi }
{ F.S. Setiap elemen list diprint }

KAMUS LOKAL

ALGORITMA
   if (IsEmpty(L)) then { Basis 0 }
       { tidak melakukan apa-apa }
   else { Rekurens }
       output (info(L))
        PrintList(Next(L))
```

```
function NBElmtlist (L : List) → integer
{ Mengirimkan banyaknya elemen list L, Nol jika L kosong }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (IsEmpty(L)) then { Basis 0 }
      → 0
   else { Rekurens }
       →  1 + NBElmtList(Next(L))
```

```
procedure  NBElmtlist1 (input L : List, output N : integer)
{ I.S. L terdefinisi }
{ F.S. N berisi banyaknya elemen list }
```

```
KAMUS LOKAL
   N1 : integer
```

```
ALGORITMA
   if (IsEmpty(L)) then { Basis 0 }
      N ← 0
   else { Rekurens }
       NBElmtList1(Next(L),N1)
       N ← 1 + N1
```

```
procedure  NBElmtlistAcc (input L : List, input/output Acc : integer,
                          output N : integer)
{ I.S. L terdefinisi }
{F.S. N berisi banyaknya elemen list }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (IsEmpty(L)) then { Basis: kondisi berhenti }
      N ← Acc
   else { Rekurens: Next element, Proses }
       Acc ← Acc + 1
       NBElmtListAcc(Next(L),Acc,N)
```

Cara Call : Acc ← 0; NBElmtlistAcc(L,Acc,0)

```
function Search (L : List, X : infotype) → boolean
{ Mengirim true jika X adalah anggota list, false jika tidak }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (IsEmpty(L)) then { Basis 0 }
      → false
   else { Rekurens }
      if (info(L) = X)  then
          true
       else
          Search(Next(L), X)
```

Fungsi Dasar untuk Pemrosesan List secara Rekursif

(Mengacu ke Diktat "Pemrograman Fungsional")

```
function FirstElmt (L : List) → InfoType
{ Mengirimkan elemen pertama sebuah list L yang tidak kosong }
KAMUS LOKAL
ALGORITMA
    → Info(L)
```

```
function Tail (L : List) → List
{ Mengirimkan list L tanpa elemen pertamanya, mungkin yang dikirimkan adalah
sebuah list kosong }
KAMUS LOKAL
ALGORITMA
    → Next(L)
```

```
function Konso (L : List, e : InfoType) → List
{ Mengirimkan list L dengan tambahan e sebagai elemen pertamanya }
{ Jika alokasi gagal, mengirimkan L }
KAMUS LOKAL
   P : address
ALGORITMA
    P ← Alokasi(e)
    if (P = Nil) then
      → L
    else
      {Insert First }
      Next(P)= L
      → P
```

```
function Kons• (L : List, e : infotype) → List
{ Mengirimkan list L dengan tambahan e sebagai elemen terakhir }
{ Jika alokasi gagal, mengirimkan L }

KAMUS LOKAL
    P : address
    Last : address

ALGORITMA
    P ← Alokasi(e)
    if (P = Nil) then
      → L
    else
      { Insert Last }
      if ISEmpty(L) then { insert ke list kosong }
         → L
      else
         Last ← L
         while (Next(Last) ≠ Nil) do
             Last ← Next(Last)
         { Next(Last=Nil; Last adalah alamt elemen terakhir }
         { Insert Last }
         Next(Last) ← P
         → L
```

```
function Copy (L : List) → List
{ Mengirimkan salinan list L }
{ Jika alokasi gagal, mengirimkan L }

KAMUS LOKAL

ALGORITMA
   if (IsEmpty(L)) then { Basis 0 }
     → L
   else { Rekurens }
     → Konso(FirstElmt(L),Copy(Tail(L))
```

```
procedure MengCopy (input Lin : List, output Lout : List)
{ I.S. Lin terdefinisi }
{ F.S. Lout berisi salinan dari Lin }
{ Proses : menyalin Lin ke Lout }
{ Jika alokasi gagal, Lout adalah ??? }

KAMUS LOKAL
    Ltemp : List

ALGORITMA
   if (IsEmpty(L)) then
      Lout ← Lin
   else
      Copy(Tail(L),Ltemp)
      Lout ← Konso(FirstElmt(Lin),Ltemp)
```

```
function Concat (L1, L2 : List) → List
{ Mengirimkan salinan hasil konkatenasi list L1 dan L2 }
KAMUS LOKAL
ALGORITMA
   if (IsEmpty(L1) then { Basis }
      → Copy(L2)
   else { Rekurens }
      → Konso(FirstElmt(L1),Concat(Tail(L1),L2))
```

```
function Concat (L1, L2 : List) → List
{ Mengirimkan salinan hasil konkatenasi list L1 dan L2 }
KAMUS LOKAL
ALGORITMA
   if (IsEmpty(L2) then  {Basis }
       → Copy(L1)
   else {rekurens}
       → Kons•(Concat(L1, Head(L2)),LastElmt(L2))
```

```
procedure Meng_Concat (input L1, L2 : List, output LHsl : List)
{ I.S. L1, L2 terdefinisi }
{ F.S. LHsl adalah hasil melakukan konkatenasi L1 dan L2 dengan cara "disalin" }
{ Proses : Menghasilkan salinan hasil konkatenasi list L1 dan L2 }
KAMUS LOKAL
ALGORITMA
   if (IsEmpty(L2) then  {Basis }
      LHsl ← Copy(L2)
   else {rekurens}
      LHsl ← Konso(FirstElmt(L1),Concat(L1,Tail(L2))
```

Pohon

Pendahuluan

Struktur pohon adalah struktur yang penting dalam bidang informatika, yang memungkinkan kita untuk :

- mengorganisasi informasi berdasarkan sutau struktur logik
- memungkinkan cara akses yang khusus terhadap suatu elemen

Contoh persoalan yang tepat untuk direpresentasi sebagai pohon:

- pohon keputusan,
- pohon keluarga dan klasifikasi dalam botani,
- pohon sintaks dan pohon ekspresi aritmatika

Definisi Rekurens dari Pohon

Sebuah POHON adalah himpunan terbatas tidak kosong, dengan elemen yang dibedakan sebagai berikut :

- sebuah elemen dibedakan dari yang lain, yang disebut sebagai AKAR dari pohon.
- elemen yang lain (jika masih ada) dibagi-bagi menjadi beberapa sub himpunan yang disjoint, dan masing-masing sub himpunan tersebut adalah POHON yang disebut sebagai SUB POHON dari POHON yang dimaksud.

Contoh :
Sebuah buku dipandang sebagai pohon. Judul buku adalah AKAR. Buku dibagi menjadi bab-bab. Masing-masing bab adalah sub pohon yang juda mengandung JUDUL sebagai AKAR dari bab tersebut. Bab dibagi menjadi subbab yang juga diberi judul. Subbab adalah pohon dengan judul subbab sebagai akar. Daftar Isi buku dengan penulisan yang diindentasi mencerminkan struktur pohon dari buku tersebut.

HUTAN
Definisi : hutan adalah sequence (list) dari pohon

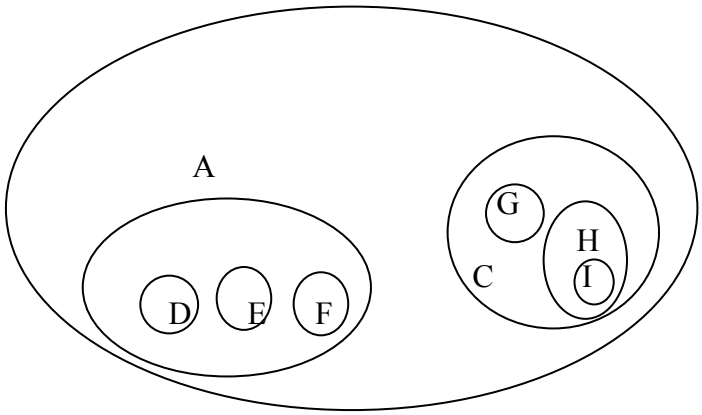
Jika kita mempunyai sebuah hutan, maka kita dapat menambahkan sebuah akar fiktif pada hutan tersebut dan hutan tersebut menjadi list dari sub pohon. Demikian pula sebaliknya: jika diberikan sebuah pohon dan kita membuang akarnya, maka akan didapatkan sebuah hutan.

Catatan:
Suffiks (akhiran) n-aire menunjukkan bahwa sub pohon bervariasi
semua elemen dari pohon adalah akar dari sub pohon, yang sekaligus menunjukkan pohon tersebut.
Pada definisi di atas, tidak ada urutan sub pohon, namun jika logika dari persoalan mengharuskan suatu strukturasi seperti halnya pada buku, maka dikatakan bahwa pohon berarah.

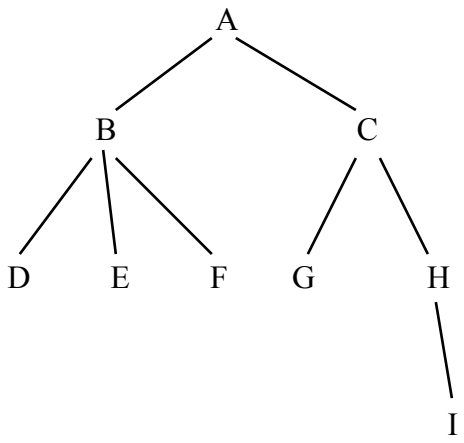
Cara Penulisan Pohon

Berikut ini merepresentasikan pohon yang sama:

a) Himpunan yang saling melingkupi



b) Graph



c) Indentasi

```
A
    B
        D
        E
        F
    C
    G
        H
            I
```

d) Bentuk linier :

Prefix : A (B(D,E,F), C(G,H(I)))

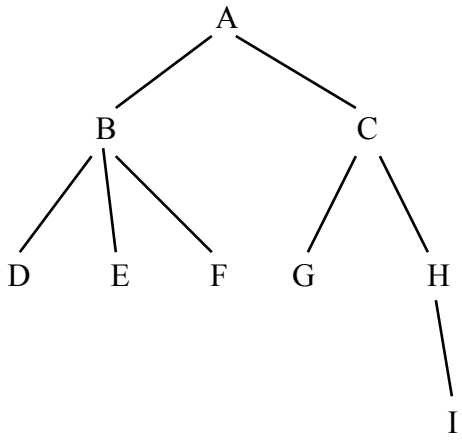
Posfix : ((D,E,F)B,(G,(I)H)C) A

Beberapa Istilah

SIMPUL (node): adalah elemen dari pohon yang memungkinkan akses pada sub pohon dimana simpul tersebut berfungsi sebagai AKAR.

CABANG (path): hubungan antara akar dengan sub pohon

Contoh: pada gambar pohon sebagai berikut:



Maka A dihubungkan dengan B dan C, untuk menunjukkan bahwa AKAR A dan kedua himpunan {B,D,E,F} dan {C,G,H,I} masing-masing adalah pohon dengan akar B dan C.

AYAH
Akar dari sebuah pohon adalah AYAH dari sub pohon.

ANAK
ANAK dari sebuah AKAR adalah sub pohon.

SAUDARA
SAUDARA adalah simpul-simpul yang mempunyai AYAH yang sama.

DAUN adalah simpul terminal dari pohon. Semua simpul selain daun adalah simpul BUKAN-TERMINAL.

JALAN (*path*) adalah suatu urutan tertentu dari CABANG.
DERAJAT sebuah pohon adalah banyaknya anak dari pohon tersebut. Sebuah simpul berderajat N disebut sebagai pohon N-aire. Pada pohon biner, derajat dari sebuah simpul mungkin 0-aire (daun), 1 -aire/uner atau 2-aire/biner.

TINGKAT (level) pohon adalah panjangnya jalan dari AKAR sampai dengan simpul yang bersangkutan. Sebagai perjanjian, panjang dari jalan adalah banyaknya simpul yang dikandung pada jalan tersebut. Akar mempunyai tingkat sama dengan 1. Dua buah simpul disebut sebagai SEPUPU jika mempunyai tingkat yang sama dalam suatu pohon.

KEDALAMAN atau **tinggi** sebuah pohon adalah nilai maksimum dari tingkat simpul yang ada pada pohon tersebut. Kedalaman adalah panjang maksimum jalan dari akar menuju ke sebuah daun.

LEBAR sebuah pohon adalah maksimum banyaknya simpul yang ada pada suatu tingkat.

Catatan:
Diberikan sebuah pohon biner dengan N elemen. Jika :

- b adalah banyaknya simpul biner
- u adalah banyaknya simpul uner
- d adalah banyaknya daun

Maka akan selalu berlaku:

$$N = b + u + d$$
$$n-1 = 2\,b + u$$

sehingga

$$b = d - 1$$

Representasi ponon n-aire : adalah dengan list of list

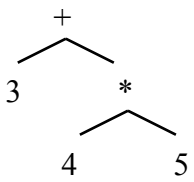
Pohon Biner

Definisi:
Sebuah pohon biner adalah himpunan terbatas yang:

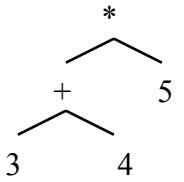
- mungkin kosong, atau
- terdiri dari sebuah simpul yang disebut akar
  dan dua buah himpunan lain yang *disjoint* yang merupakan pohon biner, yang disebut sebagai sub pohon kiri dan sub pohon kanan dari pohon biner tersebut

Perhatikanlah perbedaan pohon biner dengan pohon biasa : pohon biner mungkin kosong, sedangkan pohon n-aire tidak mungkin kosong.

Contoh pohon ekspresi aritmatika:

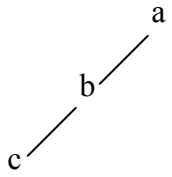


3 + (4 * 5)

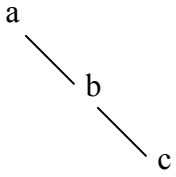


(3 +4) * 5

Karena adanya arti bagi sub pohon kiri dan sub pohon kanan, dua buah pohon biner sebagai berikut berbeda (pohon berikut disebut pohon condong/*skewed tree*).



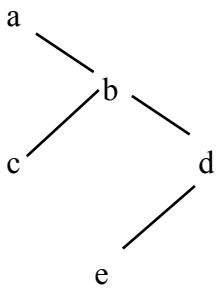
Pohon condong kiri



Pohon condong kanan

Sub pohon ditunjukkan dengan penulisan ( )

Notasi prefix :
a(( ),b(c(( ),( )),d(e(( ),( )),( ))))



ADT Pohon Biner dengan Representasi Berkait (Algoritmik)

```
{ Deklarasi TYPE }
   type node : < Info : infotype,
                 Left : address,
                 Right : address >
   type BinTree : address
   type ElmtNode : < Info : infotype,
                     Next : address >
   type ListOfNode : address { list linier yang elemennya adalah ElmtNode }

{ PRIMITIF }
{ Selektor }
function GetAkar (P : BinTree) → infotype
{ Mengirimkan nilai Akar pohon biner P }
function GetLeft (P : BinTree) → BinTree
{ Mengirimkan Anak Kiri pohon biner P }
function GetRight (P : BinTree) → BinTree
{ Mengirimkan Anak Kanan pohon biner P }

{ Konstruktor }
function Tree (Akar : infotype, L : BinTree, R : BinTree) → BinTree
{ Menghasilkan sebuah pohon biner dari A, L, dan R, jika alokasi berhasil }
{ Menghasilkan pohon kosong (Nil) jika alokasi gagal }

procedure MakeTree (input Akar : infotype, input L : BinTree,
                    input R : BinTree, output P : BinTree)
{ I.S. Sembarang }
{ F.S. Menghasilkan sebuah pohon P }
{ Menghasilkan sebuah pohon biner P dari A, L, dan R, jika alokasi berhasil }
{ Menghasilkan pohon P yang kosong (Nil) jika alokasi gagal }

procedure BuildTree (output P : BinTree)
{ Membentuk sebuah pohon biner P dari pita karakter yang dibaca }
{ I.S. Pita berisi "konstanta" pohon dalam bentuk prefix. Memori pasti cukup,
alokasi pasti berhasil }
{ F.S. P dibentuk dari ekspresi dalam pita }

{ Predikat Penting }
function IsUnerLeft (P : BinTree) → boolean
{ Mengirimkan true jika pohon biner tidak kosong P adalah pohon unerleft: hanya
mempunyai subpohon kiri }
function IsUnerRight (P : BinTree) → boolean
{ Mengirimkan true jika pohon biner tidak kosong P adalah pohon unerright: hanya
mempunyai subpohon kanan}
function IsBiner (P : BinTree) → boolean
{ Mengirimkan true jika pohon biner tidak kosong P adalah pohon biner: mempunyai
subpohon kiri dan subpohon kanan}
```

```
{ Traversal }
procedure Preorder (input P : BinTree)
{ I.S. P terdefinisi }
{ F.S. Semua simpul P sudah diproses secara Preorder: akar, kiri, kanan dengan
Proses (P) }
procedure Inorder (input P : BinTree)
{ I.S. P terdefinisi }
{ F.S. Semua simpul P sudah diproses secara Inorder: kiri, akar, kanan dengan
Proses (P) }
procedure Postorder (input P : BinTree)
{ I.S. P terdefinisi }
{ F.S. Semua simpul P sudah diproses secara Postorder: kiri, kanan, akar dengan
Proses (P) }
procedure PrintTree (input P : BinTree, input h : integer)
{ I.S. P terdefinisi, h adalah jarak indentasi }
{ F.S. Semua simpul P sudah ditulis dengan indentasi }

{ Search }
function Search (P : BinTree, X : infotype) → boolean
{ Mengirimkan true jika ada node dari P yang bernilai X }

{ Fungsi lain }
function nbELmt (P : BinTree) → integer
{ Mengirimkan banyaknya elemen (node) pohon biner P }
function nbDaun (P : BinTree) → integer
{ Mengirimkan banyaknya daun (node) pohon biner P }
function IsSkewLeft (P : BinTree) → boolean
{ Mengirimkan true jika P adalah pohon condong kiri }
function IsSkewRight (P : BinTree) → boolean
{ Mengirimkan true jika P adalah pohon condong kiri }

function Level (P : BinTree, X : infotype) → integer
{ Mengirimkan level dari node X yang merupakan salah satu simpul dari pohon
biner P. Akar(P) level-nya adalah 1. POhon P tidak kosong. }

{ Operasi lain }
procedure AddDaunTerkiri (input/output P : BinTree, input X : infotype)
{ I.S. P boleh kosong }
{ F.S. P bertambah simpulnya, dengan X sebagai simpul daun terkiri }

procedure AddDaun (input/Output P : BinTree, input X, Y : infotype,
                   input Kiri : boolean)
{ I.S. P tidak kosong, X adalah salah satu daun Pohon Biner P }
{ F.S. P bertambah simpulnya, dengan Y sebagai anak kiri X (jika Kiri), atau
sebagai anak Kanan X (jika not Kiri) }

procedure DelDaunTerkiri (input/output P : BinTree, output X : infotype)
{ I.S. P tidak kosong }
{ F.S. P dihapus daun terkirinya, dan didealokasi, dengan X adalah info yang
semula disimpan pada daun terkiri yang dihapus }

procedure DelDaun (input/output P : BinTree, input X : infotype)
{ I.S. P tidak kosong, X adalah salah satu daun }
{ F.S. X dihapus dari P }

function MakeListDaun (P : BinTree) → ListOfNode
{ Jika P adalah pohon kosong, maka menghasilkan list kosong. }
{ Jika P bukan pohon kosong: menghasilkan list yang elemennya adalah semua daun
pohon P, jika semua alokasi berhasil. Menghasilkan list kosong jika ada alokasi
yang gagal }

function MakeListPreorder (P : BinTree) → ListOfNode
{ Jika P adalah pohon kosong, maka menghasilkan list kosong. }
{ Jika P bukan pohon kosong: menghasilkan list yang elemennya adalah semua
elemen pohon P dengan urutan Preorder, jika semua alokasi berhasil. Menghasilkan
list kosong jika ada alokasi yang gagal }
```

```
function MakeListLevel (P : BinTree, N : integer) → ListOfNode
{ Jika P adalah pohon kosong, maka menghasilkan list kosong. }
{ Jika P bukan pohon kosong: menghasilkan list yang elemennya adalah semua
elemen pohon P yang levelnya=N, jika semua alokasi berhasil. Menghasilkan list
kosong jika ada alokasi yang gagal. }

{ Membentuk balanced tree }
function BuildBalanceTree (n : integer) → BinTree
{ Menghasilkan sebuah balanced tree dengan n node, nilai setiap node dibaca }

{ Terhadap Binary Search Tree }
function BSearch (P : BinTree, X : infotype) → boolean
{ Mengirimkan true jika ada node dari P yang bernilai X }

function InsSearch (P : BinTree, X : infotype) → BinTree
{ Menghasilkan sebuah pohon Binary Search Tree P dengan tambahan simpul X. Belum
ada simpul P yang bernilai X. }

procedure DelBtree (input/output P : BinTree, input X : infotype)
{ I.S. Pohon P tidak kosong }
{ F.S. Nilai X yang dihapus pasti ada }
{ Sebuah node dg nilai X dihapus }
```

```
KAMUS
   type node : < Info : infotype,
                 Left : address,
                 Right : address >
   type BinTree : address
```

```
function  Akar (P : BinTree) → infotype
{ Mengirimkan informasi yang tersimpan di Akar dari Pohon Biner yang tidak
kosong }
```

```
KAMUS LOKAL
```

```
ALGORITMA
    → Info(P)
```

```
function Left (P : BinTree) → BinTree
{ Mengirimkan anak kiri dari Pohon Biner yang tidak kosong }
```

```
KAMUS LOKAL
```

```
ALGORITMA
    → Left(P)
```

```
function Right (P : BinTree) → BinTree
{ Mengirimkan anak kanan dari Pohon Biner yang tidak kosong }
```

```
KAMUS LOKAL
```

```
ALGORITMA
    → Right(P)
```

```
function IsUnerLeft (P : BinTree) → boolean
{ Prekondisi: P tidak kosong. Mengirimkan true jika P adalah pohon unerleft:
hanya mempunyai subpohon kiri }

KAMUS LOKAL

ALGORITMA
    → (Right(P) = Nil) and (Left(P) ≠ Nil)
```

```
function IsUnerRight (P : BinTree) → boolean
{ Prekondisi: P tidak kosong. Mengirimkan true jika P adalah pohon unerright:
hanya mempunyai subpohon kanan }

KAMUS LOKAL

ALGORITMA
    → (Left(P) = Nil) and (Right(P) ≠ Nil)
```

```
function IsBiner (P : BinTree) → boolean
{ Prekondisi: P tidak kosong. Mengirimkan true jika P adalah pohon biner:
mempunyai subpohon kiri dan subpohon kanan}

KAMUS LOKAL

ALGORITMA
    → (Left(P) ≠ Nil) and (Right(P) ≠ Nil)
```

```
procedure PreOrder (input P : BinTree)
{ I.S. Pohon P terdefinisi }
{ F.S. Semua node pohon P sudah diproses secara PreOrder: akar, kiri, kanan }
{ Basis: Pohon kosong : tidak ada yang diproses }
{ Rekurens: Proses Akar(P); Proses secara Preorder (Left(P)); Proses secara
Preorder (Right(P)) }

KAMUS LOKAL

ALGORITMA
    if (P = Nil) then {Basis-0}

    else { Rekurens, tidak kosong }
       Proses(P)
       PreOrder(Left(P))
       PreOrder(Right(P) )
```

```
procedure InOrder (input P : BinTree)
{ I.S. Pohon P terdefinisi }
{ F.S. Semua node pohon P sudah diproses secara InOrder: kiri, akar, kanan }
{ Basis: Pohon kosong: tidak ada yang diproses }
{ Rekurens: Proses secara Inorder (Left(P)); Proses Akar(P); Proses secara
Inorder (Right(P)) }

KAMUS LOKAL

ALGORITMA
    if (P = Nil) then { Basis 0 }

    else { Rekurens }
       InOrder(Left(P))
       Proses(P)
       InOrder(Right(P))
```

```
procedure PostOrder (input P : BinTree)
{ I.S. Pohon P terdefinisi }
{ F.S. Semua node pohon P sudah diproses secara PostOrder: kiri, kanan, akar }
{ Basis: Pohon kosong: tidak ada yang diproses }
{ Rekurens: Proses secara Postorder(Left(P)); Proses secara Postorder
(Right(P)); Proses Akar(P) }
```

KAMUS LOKAL

ALGORITMA

```
    if (P ≠ Nil) then { Basis 0}

    else { Rekurens }
        PostOrder(Left(P))
        PostOrder(Right(P))
        Proses(P)
```

```
function Tinggi (R : BinTree) → integer
{ Pohon Biner mungkin kosong. Mengirim "depth" yaitu tinggi dari pohon }
{ Basis: Pohon kosong: tingginya nol }
{ Rekurens: 1 + maksimum (Tinggi(Anak kiri), Tinggi(AnakKanan)) }
```

KAMUS LOKAL

ALGORITMA

```
    if (R = Nil) then { Basis 0 }
       → 0
    else { Rekurens }
       → 1 + max (Tinggi(Left(R), Right(R))
```

```
function NBDaun (P : BinTree) → integer
{ Pohon Biner tidak mungkin kosong. Mengirimkan banyaknya daun pohon }
{ Basis: Pohon yang hanya mempunyai akar: 1 }
{ Rekurens: }
{ Punya anak kiri dan tidak punya anak kanan: NBDaun(Left(P)) }
{ Tidak Punya anak kiri dan punya anak kanan : NBDaun(Right(P)) }
{ Punya anak kiri dan punya anak kanan : NBDaun(Left(P)) + NBDaun(Right(P)) }
```

KAMUS LOKAL

ALGORITMA

```
    if (Left(P)=Nil) and (Right(P)=Nil) then { Basis 1 : akar }
       → 1
    else { Rekurens }
       depend on (P)
          Left(P)≠Nil and Right(P)=Nil  → NBDaun(Left(P))
          Left(P)=Nil and Right(P)≠Nil  → NBDaun(Right(P))
          Left(P)≠Nil and Right(P)≠Nil  → NBDaun(Left(P))+NBDaun(Right(P))
```

```
function Tree(X : infotype, L, R : BinTree) → BinTree
{ Menghasilkan sebuah tree, jika alokasi berhasil P }

KAMUS LOKAL
    P : address

ALGORITMA
   P ← Alokasi(X)
   if (P ≠ Nil) then
      Left(P) ← L
      Right(P) ← R
   { end if }
   → P
```

```
procedure MakeTree (input X : infotype, input L, R : BinTree,
                    output P : BinTree)
{ Menghasilkan sebuah tree, jika alokasi berhasil P }

KAMUS LOKAL

ALGORITMA
   P ← Alokasi (X)
   if (P ≠ Nil) then
      Left(P) ← L
      Right(P) ← R
   { end if }
```

Pohon Seimbang (Balanced Tree)

Pohon seimbang:

- Pohon seimbang tingginya: perbedaan tinggi sub pohon kiri dengan sub pohon kanan maksimum 1.
- Pohon seimbang banyaknya simpul: perbedaan banyaknya simpul sub pohon kiri dengan sub pohon kanan maksimum 1.

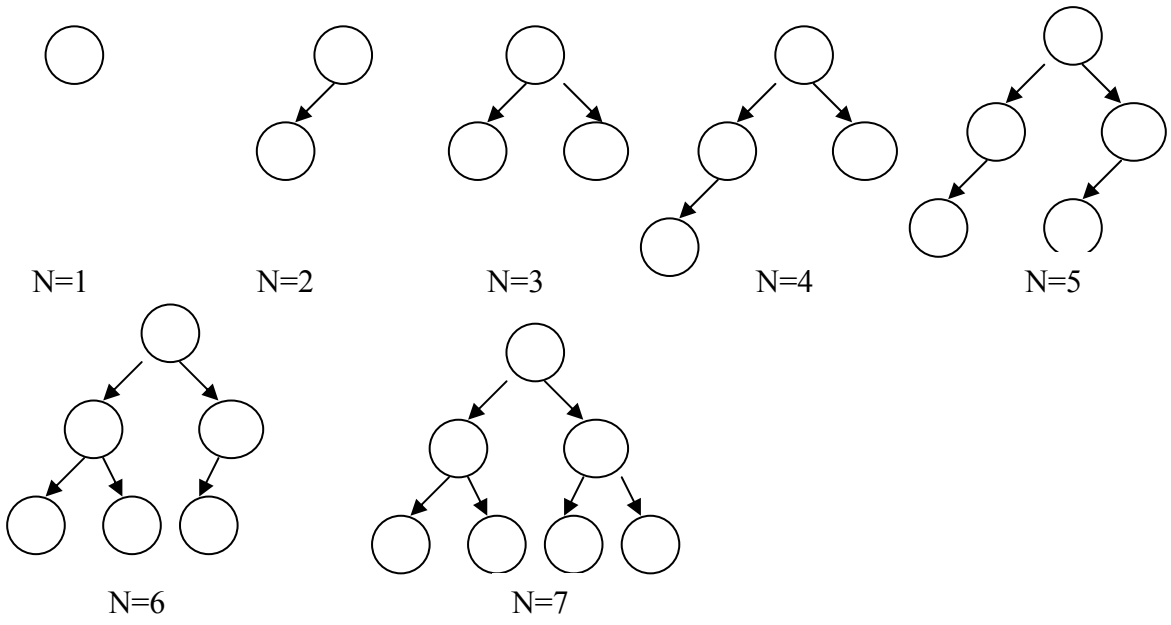
Berikut ini adalah algoritma untuk pohon yang seimbang banyaknya simpulnya.

```
function BuildBalancedTree (n : integer) → BinTree
{ Menghasilkan sebuah balanced tree }
{ Basis: n = 0: Pohon kosong  }
{ Rekurens: n>0: partisi banyaknya node anak kiri dan kanan, lakukan proses yang
sama  }
```

```
KAMUS LOKAL
     P : address
    nL, nR : integer
```

```
ALGORITMA
    if (n=0) then {Basis }
       → nil
    else {Rekurens }
       { bentuk akar }
       input(X)
       P ← Alokasi(x)
       if (P ≠ Nil) then
          { Partisi sisa node sebagai anak kiri dan anak kanan }
          nL ← n div 2; nR ← n – nL - 1
          L ← BuildBalancedTree(NL); R ← BuildBalancedTree(NR)
          Left(P) ← L; Right(P) ← R
       → P
```

Urutan pembentukan :



Contoh eksekusi jika data yang dibaca adalah 1,2,3,4,5,6,7,8,9,10, 11, 12
(buat sebagai latihan)

Pohon Biner Terurut (Binary Search Tree)

Pohon biner terurut P memenuhi sifat :

- Semua simpul subpohon kiri selalu < dari Info(P)
- Semua simpul subpohon kiri selalu > dari Info(P)

Untuk simpul yang sama nilainya : disimpan berapa kali muncul. Maka sebuah node P akan menyimpan informasi : Info(P), Left(P), Right(P) , Count(P) yaitu banyaknya kemunculan Info(P).

```
procedure InsSearchTree (input X : infotype, input/output P : BinTree)
{ Menambahkan sebuah node X ke pohon biner pencarian P }
{ infotype terdiri dari key dan count. Key menunjukkan nilai unik, dan Count
berapa kali muncul }
{ Basis : Pohon kosong  }
{ Rekurens : Jika pohon tidak kosong, insert ke anak kiri jika nilai <
info(Akar) }
{ atau insert ke anak kanan jika nilai > info(Akar) }
{ Perhatikan bahwa insert selalu menjadi daun terkiri/terkanan dari subpohon }
{ Alokasi selalu berhasil }
```

```
KAMUS LOKAL
```

```
ALGORITMA
   if (IsEmpty(P)) then {Basis: buat hanya akar }
      MakeTree(X,nil,nil,P)
   else {Rekurens }
      depend on X, Key(P)
         X = Akar(P) : Count(P) ← Count(P) + 1
         X < Akar(P) : InsSearchTree(X,Left(P))
         X > Akar(P) : InsSearchTree(X,Right(P))
```

```
procedure DelBinSearchTree (input/output P : BinTree, input X : infotype)
{ Menghapus simpul bernilai Key(P)=X }
{ infotype terdiri dari key dan count. Key menunjukkan nilai unik, dan Count
berapa kali muncul }
{ Basis : Pohon kosong  }
{ Rekurens :  }
```

```
KAMUS LOKAL
   q : address
   procedure DelNode (input/output P : BinTree)
```

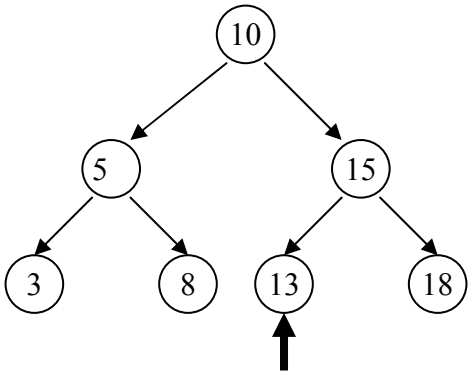
```
ALGORITMA
   depend on X, Key(P)
     X < Akar(P) : DelBTree(Left(P),X)
     X > Akar(P) : DelBTree(Right(P),X)
     X = Akar(P) : { Delete simpul ini }
                   q ← P
                   if Right(q) = Nil then p ← Left(q)
                   else if Left(q) = Nil then p ← Right(q)
                        else
                           DelNode(Left(q))
                           Dealokasi(q)
```

```
procedure  DelNode(inpu/output P: BinTree)
{ I.S. P adalah pohon biner tidak kosong }
{ F.S. q berisi salinan nilai Nilai daun terkanan disalin ke q }
{ Proses : }
{ Memakai nilai q yang global}
{ Traversal sampai daun terkanan, copy nilai daun terkanan P, salin nilai ke q
semula }
{ q adalah anak terkiri yang akan dihapus }
```

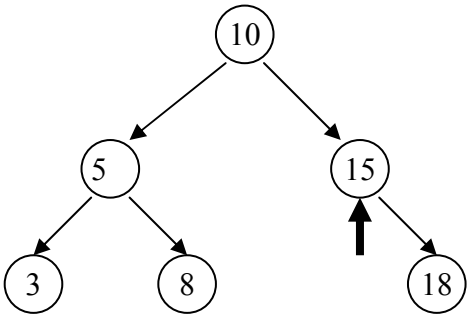
```
KAMUS LOKAL
```

```
ALGORITMA
   depend on P
      Right(P) ≠ Nil : DelNode(Right(P))
      Right(P) = Nil : Key(q) ← Key(P)
                       Count(q) ← Count(P)
                       q ← P
                       P ← Left(P)
```

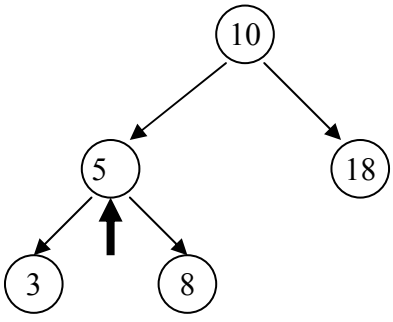
Contoh eksekusi penghapusan node pada pohon biner terurut:



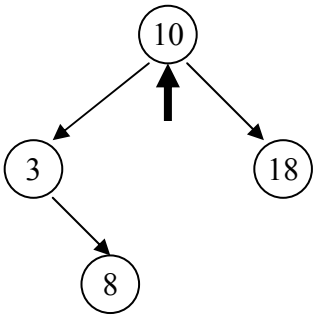
(a)



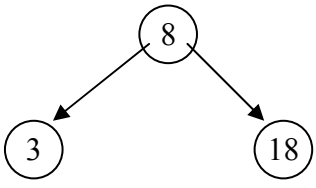
(b)



(c)



(d)



(e)

Algoritma untuk Membentuk Pohon Biner dari Pita Karakter

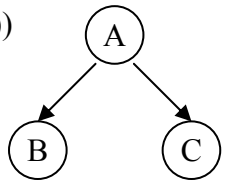
Representasi "pohon" dalam pita karakter adalah penting. Untuk persoalan ini, jika pita karakter yang isinya **ekspresi pohon dalam bentuk list**.. Untuk sementara, dibuat sebuah struktur dengan address Parent yang eksplisit, sehingga setiap Node adalah struktur sebagai berikut:

```
type Node: < Parent : address,
             Left   : address,
             Info   : character,
             Right  : address     >
```

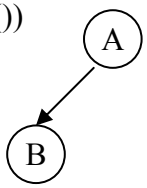
Contoh pita dan pohon yang direpresentasi
( ) adalah pohon kosong, yang akan diproses secara khusus

(A()())

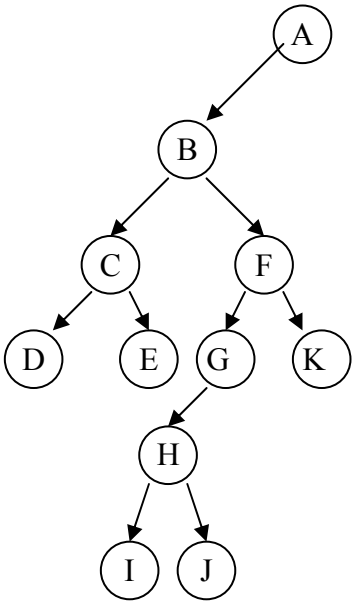
(A(B()())(C()()))



(A(B()())())



(A(B(C(D()())(E()()))(F(G(H(I()())(J()())) (K()()))



Ide untuk membangun algoritma secara umum :
Dengan menganalisis isi pita, maka ada tiga kelompok karakter:

- Karena pembacaan pita dilakukan secara sekuensial, maka pembentukan pohon selalu dimulai dari akar.
- Pembacaan karakter demi karakter dilakukan secara iteratif, untuk membentuk sebuah pohon, selalu dilakukan insert terhadap daun.
- Karakter berupa Abjad, menandakan bahwa sebuah node harus dibentuk, entah sebagai anak kiri atau anak kanan.
- Karakter berupa "(" menandakan suatu sub pohon baru.
- Jika karakter sebelumnya adalah ')' maka siap untuk melakukan insert sub pohon kanan.
- Jika karakter sebelumnya adalah abjad, maka siap untuk melakukan insert sub pohon kiri.
- Karakter berupa ")" adalah penutup sebuah pohon, untuk kembali ke "Bapaknya", berarti naik levelnya dan tidak melakukan apa-apa, tetapi menentukan proses karakter berikutnya.

Kesimpulan:
Untuk mesin akses, tidak cukup dengan mesin karakter (hanya CC), sebab untuk memproses sebuah karakter, dibutuhkan informasi karakter sebelumnya.

- Jika CC adalah abjad dan karakter sebelumnya adalah '(', maka node siap diinsert sebagai anak kiri).
- Jika CC adalah '(', maka harus siap melakukan insert sebuah node dan berikutnya adalah melakukan insert sebagai sub pohon kiri.
- Jika CC adalah ')' dan karakter sebelumnya adalah '(' maka tidak ada yang perlu diinsert pada level tersebut.

```
/* Nama file : mtree.h */
/* Dibuat oleh Riza Satria Perdana */
/* Deskripsi : Header primitif Tree */
/* Representasi pohon : pohon biner dengan tambahan informasi pointer ke Parent
*/

#ifndef _MTREE
#define _MTREE

typedef char Infotype;

#define Nil NULL;

/*** Selektor ****/
#define Info(P) (P)->Info;
#define Left(P) (P)->Left;
#define Right(P) (P)->Right;
#define Parent(P) (P)->Parent;

typedef struct tElmtTree *address;
typedef struct tElmtTree {
   InfoType info;
   address Left;
   address Right;
   address Parentt;
} ElmtTree;
typedef address Tree;
```

```
void Alokasi (address*P, InfoType X);
/* I.S. sembarang */
/* F.S. address P dialokasi, dan bernilai tidak Nil jika berhasil */
/* Alokasi sebuah address P */

void MakeTree(Tree *T);
void PrintTree (Tree T);

#endif
```

```
/* Nama file : mtree.c */
/* Dibuat oleh Riza Satria Perdana */
/* Deskripsi : membentuk sebuah pohon dari pita karakter */

#include <stdlib.h>
#include "boolean.h"
#include "tree.h"
#include "mesinkar.h"

void Alokasi (address *P,InfoType X)
{
  *P =(adress) malloc(sizeof(ElmtTree));
  if (*P !=Nil)
  { Info(*P)=X;
    Parent(*P)=Nil;
    Left(*P)=Nil;
    Right(*P)=Nil;
  }
}

void MakeTree (Tree *T)
{ /* Kamus */
  address CurrParent;
  address Ptr;
  int level;
  boolean Inski;

  /* Algoritma */
  START_COUPLE;
  Alokasi (T,CC);
  CurrParent = *T;
  ADV_COUPLE;
  while (!EOP)
  {  switch (CC)
     { case '(' : level++;
                  InsKi= C1 !='(';
                  break;
       case ')' : level--;
                  if( C1 !='(')
                  {
                    CurrParent=Parent (CurrParent);
                  }
                  break;
       default  : Alokasi (Ptr,CC);
                  if(InsKi)
                  { Left(CurrParent)= Ptr; }
                  else
                  { Right(CurrParent)= Ptr; }
                  Parent(Ptr)=CurrParent;
                  break;
     }
     ADV_COUPLE;
  }
}
```

```
int main ()
{
/* Kamus */
  Tree T;
/* Algoritma */
   MakeTree(&T);
   PrintTree (T);
return 0;
}
```

Pembangunan Pohon Biner secara Rekursif

```
/* File : tree.h */
/* dibuat oleh Riza Satria Perdana */

    #ifndef __TREE_H__
    #define __TREE_H__

    #include <stdlib.h>

    #define Nil NULL

    #define Info(T) (T)->info
    #define Left(T) (T)->left
    #define Right(T) (T)->right

    typedef char infotype;
    typedef struct TNode *address;
    typedef struct TNode
    { infotype info;
      address left;
      address right;
    } Node;

    typedef address Tree;

    void Allocate(address *P);
    void BuildTree(Tree *T);
    void BuildTreeFromString(Tree *T, char *st, int *idx);
    void PrintTree(Tree T);
    #endif
```

```
/* file : tree.c */
    #include "tree.h"
    #include "mesinkar.h"

    void Allocate(address *P)
    { *P=(address) malloc(sizeof(Node));
    }

    void BuildTree(Tree *T)
    /* dipakai jika input dari pita karakter */
    { ADV();
      if (CC==')')
        (*T)=Nil;
      else
      { Allocate(T);
        Info(*T)=CC;
        ADV();
        BuildTree(&Left(*T));
        BuildTree(&Right(*T));
      }
      ADV();
    }
```

```
void BuildTreeFromString(Tree *T, char *st, int *idx)
/* input dari string st */
{ (*idx)++;                              /* advance */
  if (st[*idx]==')')
    (*T)=Nil;
  else
  { Allocate(T);
    Info(*T)=st[*idx];
    (*idx)++;                            /* advance */
    BuildTreeFromString(&Left(*T),st,idx);
    BuildTreeFromString(&Right(*T),st,idx);
  }
  (*idx)++;                              /* advance */
}

void PrintTree(Tree T)
{ if (T==Nil)
    printf("()");
  else
  { printf("(%c",Info(T));
    PrintTree(Left(T));
    PrintTree(Right(T));
    printf(")");
  }
}
```

```
int main()
{ Tree T;
  int idx=0;

   /* test dari Pita */
   START();
   PrintTree(T);

   /* Test dari string yang diberikan sebagai ekspresi */
  BuildTreeFromString(&T,"(A(B()())(C()()))",&idx);
  PrintTree(T);
  return 0;
}
```

BAGIAN II. STUDI KASUS

Studi Kasus 1: Polinom

Deskripsi Persoalan

Sebuah polinom berderajad n didefinisikan sebagai fungsi P(x) sebagai berikut :

$P(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + ...+a_2 x^2 + a_1 x^1 + a_0$

Perhatikan bahwa untuk mempermudah pemrosesan, definisi P(x) tersebut berbeda dengan definisi polinom pada matematik, yang biasanya diberikan sebagai berikut

$P(x) = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + ...+a_{n-2} x^2 + a_{n-1} x^1 + a_n$

Contoh polinom :
$P1(x) = 4 x^5 + 2 x^4 + 7x^2 + 10$
$P2(x) = 23 x^{100} + 9 x^9 + 2 x^7 + 4 x^5 + 9 x^9 + 2 x^4 + 3x^2 + 1$
$P3(x) = 10$
$P4(x) = 3 x^2 + 2 x + 8$
$P5(x) = x^{1000}$

Dengan batasan bahwa $n \geq 0$ sedangkan $a_i x^i$ disebut sebagai suku ke-i dari polinom P, dan i disusun terurut mulai dari n s.d. 0 (jika muncul), harus direalisasikan sebuah paket program yang memanipulasi polinom dengan fungsi dan prosedur yang mampu menangani operasi polinom sebagai berikut:

1. Membentuk sebuah polinom P dari pasangan harga yang dibaca dari masukan, data yang dibaca adalah pasangan harga:
   (*) <Degree : integer, Coefficient : integer>
   (1) <-999, 0>
2. Menuliskan sebuah polinom P terurut mulai dari suku terbesar sampai terkecil
3. Menjumlahkan dua buah polinom P1 dan P2 dan menyimpan hasilnya pada P3, $P3 \neq P1$ dan $P3 \neq P2$; pada akhir porses P3 = P1 + P2.
4. Mengurangi dua buah polinom P1 dan P2 dan menyimpan hasilnya pada P3, $P3 \neq P1$ dan $P3 \neq P2$, pada akhir porses P3 = P1 - P2.
5. Membuat turunan dari sebuah polinom P dan menyimpan hasilnya pada P', $P' \neq P1$, pada akhir porses P' adalah turunan P1.

Program utama berupa menu yang menawarkan operasi dan memroses sesuai dengan kode operasi. Yang harus dituliskan adalah:

- kamus umum
- prosedur-prosedur yang merealisasikan kelima proses di atas
- kerangka program utama yang menawarkan operasi dasar

Pemilihan Struktur Data

Secara lojik sebuah suku polinom direpresentasi oleh sepasang harga integer:
<Degree:integer, Coefficient : integer >
Sedangkan P(x) adalah kumpulan pasangan harga tersebut yang diidentifikasi oleh namanya dan merupakan sebuah sebuah list linier dengan elemen Suku, yaitu pasangan harga <Degree:integer, Coefficient : integer>, dengan harga Degree yang maksimum sebagai derajat polinom. P(x) dapat direpresentasi secara KONTIGU atau BERKAIT.

Representasi KONTIGU

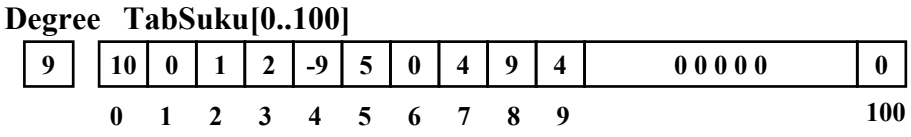
Dengan representasi kontigu, polinom dapat direpresentasi dalam sebuah tabel P, yang setiap elemennya berisi koefisien dari polinom. **Derajat polinom secara implisit adalah indeks dari tabel** sehingga tidak perlu direpresentasi secara eksplisit**.** Karena memakai tabel, indeks maksimum dari polinom harus diprediksi sebelumnya, sebutlah suatu konstanta NMax yang harganya terdefinisi.
Perhatikan bahwa dengan cara representasi seperti ini:

- semua suku yang tidak muncul harus direpresentasi dalam tabel dengan nol.
- jika banyak suku yang tidak muncul, maka banyak tempat pada tabel yang berisi nol, artinya "tidak dipakai", tetapi harus dipesan.

```
KAMUS
    { Definisi sebuah polinom P adalah }
    constant Nmax : integer = 100
    type polinom : < Degree : integer ≥ 0,
                     TabSuku : array [0..Nmax] of integer >
    P1, P2, P3 : polinom
    { P.TabSuku_i adalah koefisien dari suku ke-i dari sebuah polinom }
```

Dengan demikian, setiap operasi terhadap polinom dapat dijabarkan sebagai berikut :

1. Membentuk sebuah polinom dari pasangan harga yang dibaca dari alat masukan:
   Setiap kali memasukkan data, yang dimasukkan adalah pasangan
   (*) < Degree : integer, Coefficient : integer >
   Akhir pemasukan adalah pasangan harga yang diketikkan bernilai <-999, 0>
   Prosesnya adalah **proses sekuensial dengan mark** untuk membaca dari masukan dan menyimpannya dalam tabel polinom. Harus diperhatikan bahwa di sini elemen data tidak perlu diurut, tetapi sekaligus harus diperiksa derajat polinom, yaitu Degree yang berharga maksimum, untuk menentukan derajat. Jika pemakai memasukkan data lebih dari satu suku berderajat sama, maka selalu data yang terakhir yang direkam.
   Contoh : Jika dibaca P1:
   <0,10>, <4,-9>, <5,5>, <8,9>, <2,1>, <3,2>, <9,4>, <7,4>, <-999,0>
   maka tabel polinom yang dibentuk adalah polinom berderajat 8.



   Pada persoalan ini muncul masalah “polinom kosong” jika pengguna hanya memasukkan nilai <-999,0>. Untuk ini harus didefinisikan suatu polinom “kosong”, yaitu polinom dengan Degree = -999 dan bukan 0 karena Degree sama dengan nol

berarti polinom hanya terdiri dari konstanta. Definisi: Polinom kosong mempunyai nilai Degree = -999 dan semua sukunya pada tabel bernilai nol.

2. Menuliskan sebuah polinom:
   Prosesnya adalah **proses sekuensial tanpa mark**, traversal untuk i [Derajat..0]. Harga suku dituliskan hanya jika koefisiennya tidak nol. Untuk contoh di atas, keluaran secara sederhana adalah:

```
I   P(I)
9    4
8    9
7    4
5    5
4   -9
2    1
0   10
```

3. Menjumlahkan dua buah polinom P1 dan P2 dan menyimpan hasilnya pada P3, P3 ≠ P1 dan P3 ≠ P2:
   Yaitu menjumlahkan suku P1 dan P2 yang berderajat sama, menjadi suku berderajat tersebut pada P3. Prosesnya adalah mencari derajat tertinggi dari P1 dan P2, kemudian pemrosesan sekuensial untuk setiap pasangan suku P1 dan P2. Perhatikan bahwa mungkin hasilnya adalah polinom kosong atau derajatnya turun.
   Contoh: Jika diberikan :
   P1: <9,4>,<7,4>, <5,5>, <4,-9> <3,2>, <2,1>, <0,10>
   P2: <9,2>, <7,3>, <4,1>,<1,2>
   maka P3 akan mempunyai harga:
   P3: <9,6>, <7,7>, <5,5>, <4,-8>, <3,2>, <2,1>, <1,2>, <0,10>
   Jika P1 : <9,2> dan P2 : <9,-2> akan menghasilkan polinom kosong.
   Jika P1 : <9,2>, <8,1> dan P2 : <9,-2>
   akan menghasilkan polinom yang derajatnya turun.

P1

Degree	TabSuku[0..100]											
9	10	0	1	2	-9	5	0	4	0	4	0 0 0 0 0	0
	0	1	2	3	4	5	6	7	8	9		100

P2

Degree	TabSuku[0..100]											
9	0	2	0	0	1	0	0	3	0	2	0 0 0 0 0	0
	0	1	2	3	4	5	6	7	8	9		100

P3 = P1 + P2

Degree	TabSuku[0..100]											
9	10	2	1	2	-8	10	0	7	0	6	0 0 0 0 0	0
	0	1	2	3	4	5	6	7	8	9		100

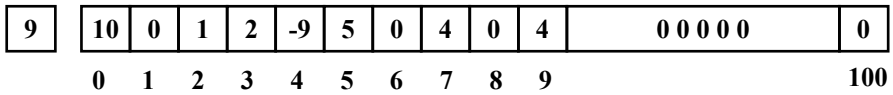
4. Mengurangi dua buah polinom P1 dan P2 dan menyimpan hasilnya pada P3, P3 ≠ P1 dan P3 ≠ P2:

Prosesnya sama dengan menjumlahkan, hanya operasi penjumlahan diganti operasi pengurangan. Pengurangan juga memungkinkan hasil berupa polinom kosong atau polinom yang derajatnya "turun".
Contoh: Jika diberikan:

P1: <9,4>,<7,4>, <5,5>, <4,-9> <3,2>, <2,1>, <0,10>
P2: <9,2>, <8,2>, <7,3>, <4,-1>,<1,2>
maka P3 akan mempunyai harga :
P3: <9,2>, <8,-2>, <7,1>, <5,5>, <4,-8>, <3,2>, <2,1>, <1,-2>, <0,10>

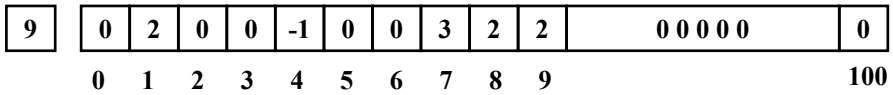
P1

Degree TabSuku[0..100]



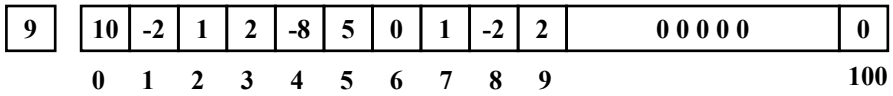
P2

Degree TabSuku[0..100]



P3 = P1 ≠ P2

Degree TabSuku[0..100]



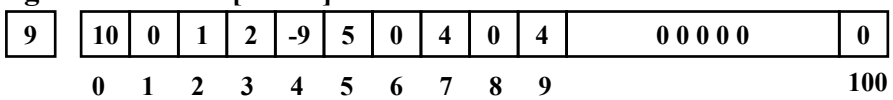
5. Membuat turunan P1 dari sebuah polinom P, P1 ≠ P:

Prosesnya adalah proses sekuensial, untuk setiap suku ke-i, i >0, yaitu $a_i\ x^i$ pada polinom P, dihitung i * a_i dan disimpan pada tabel ke i-1 pada polinom P1
Contoh jika diberikan :

P: <9,4>,<7,4>, <5,5>, <4,-9> <3,2>, <2,1>, <0,10>
maka P' akan mempunyai harga :
P1: <8,36>, <6,28>, <4,25>, <3,-36>, <2,6>, <1,2>

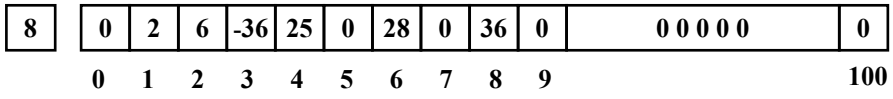
P

Degree TabSuku[0..100]



P1

Degree TabSuku[0..100]



Pada proses ini, polinom berderajat 0 akan menghasilkan polinom "kosong". Contoh: P = <0,5> akan menghasilkan P1 kosong.

Kamus umum dan program utama untuk representasi kontigu adalah sebagai berikut:

```
Program POLINOMIAL                                  { Representasi KONTIGU }

KAMUS
{ Struktur data untuk representasi polinom secara kontigu}
   constant Nmax : integer = 100 { Derajat tertinggi polinom yang
                                   diproses }
   type polinom : < Degree : integer,
                    TabSuku : array [0..Nmax] of integer >
   P1,P2 : polinom {Operan}
   P3 : polinom { Hasil }
{ Struktur data untuk interaksi }
   Finish : boolean { Mengakhiri proses }
   Pilihan : integer [0..5] { Nomor tawaran }
{ Primitif operasi terhadap polinom yang dibutuhkan untuk proses }
   procedure InitPol (output P : polinom)
   { Membuat polinom P yang kosong }
   procedure AdjustDegree (input/output P : polinom)
   { Melakukan adjustment terhadap Degree. Diaktifkan jika akibat suatu
     operasi, derajat polinom hasil berubah.
{ Primitif operasi terhadap polinom yang disediakan untuk pemakai }
   procedure CreatePol (output P : polinom)    { Mengisi polinom P }
   procedure TulisPol (input P : polinom)      { Menulis polinom P }
   procedure AddPol (input P1, P2 : polinom, output P3 : polinom)
   { Menjumlahkan P1 + P2 dan menyimpan hasilnya di P3,
     P3 ≠ P1 dan P3 ≠ P2 }
   procedure SubPol (input P1, P2 : polinom, output P3 : polinom)
   { Mengurangkan P1 - P2 dan menyimpan hasilnya di P3,
     P3 ≠ P1 dan P3≠ P2 }
   procedure DerivPol (input P : polinom, output P1 : polinom)
   { Membuat turunan P dan menyimpan hasilnya di P1, P1 ≠ P }

ALGORITMA
   Finish ← false
   repeat
      iterate
         output ("Ketik nomor di bawah [0..5] untuk memilih operasi")
         output ("1. Membentuk dua buah polinom P1 dan P2")
         output ("2. Menuliskan polinom P1,P2 dan P3")
         output ("3. Menjumlahkan polinom P1 dan P2 menjadi P3")
         output ("4. Mengurangkan P1 dan P2 menjadi P3")
         output ("5. Membentuk turunan polinom P1 yaitu P3")
         output ("0. Akhir proses")
         input (Pilihan)
      stop Pilihan ∈ [0..5]
         output ("Ulangi, pilihan di luar harga yang ditawarkan")
      { Pilihan sesuai, maka lakukan proses sesuai dengan Pilihan }
      depend on Pilihan
         Pilihan = 1 : CreatePol(P1); CreatePol(P2)
         Pilihan = 2 : TulisPol(P1); TulisPol(P2); TulisPol(P3)
         Pilihan = 3 : AddPol(P1,P2,P3)
         Pilihan = 4 : SubPol(P1,P2,P3)
         Pilihan = 5 : DerivPol(P1,P3)
         Pilihan = 0 : Finish ← true
   until Finish
```

```
procedure InitPol (output P : polinom)
{ I.S. : sembarang. F.S. Polinom P yang kosong dibentuk }
{ Membentuk sebuah polinom kosong }

KAMUS LOKAL
   i : integer   { indeks traversal }

ALGORITMA
   i traversal [0.. Nmax]
      P.TabSuku_i ← 0
   P.Degree ← -999
```

```
procedure CreatePol (output P : polinom)
{ I.S. : sembarang }
{ F.S. : Polinom P terdefinisi derajat dan koefisien-koefisiennya jika tidak
"kosong"}
{ Mengisi polinom P dengan membaca dari alat masukan, pemasukan harga diakhiri
dengan mark }

KAMUS LOKAL
   i : integer           { indeks traversal }
   Deg, Coef : integer   { input pasangan data derajat dan koefisien }
   { pemasukan data diakhiri dengan Deg=-999,
     mungkin terbentuk polinom kosong }
   MaxDegree : integer   { derajat masukan yang maksimum = derajat polinom }

ALGORITMA
   { Inisialisasi tabel penyimpan suku polinom }
   InitPol(P)

   { Input data : skema penanganan kasus kosong }
   input (Deg,Coef)
   if Deg = -999 then
      output ("Polinom kosong");
      P.Degree ← -999 { Definisi polinom kosong }
   else { Deg ≠ -999 }
      MaxDegree ← -999
      repeat
         P.TabSuku_Deg ← Coef
         if Deg > MaxDegree then
            MaxDegree ← Deg
         input (Deg,Coef)
      until (Deg = -999)
      P.Degree ← MaxDegree
```

```
procedure TulisPol (input P : polinom)
{ I.S. : P terdefinisi dan mungkin kosong }
{ F.S. : Menulis polinom P }

KAMUS LOKAL
   i : integer     { indeks traversal }

ALGORITMA
   if P.Degree ≠ -999 then
      output ('I','P(I)')
      i traversal [P.Degree..0]
         if (P.TabSuku_i ≠ 0 ) then
            output (i,P.TabSuku_i)
   else
      output ("Polinom kosong")
```

```
procedure AddPol (input P1, P2 : polinom, output P3 : Polinom)
{ I.S. : P1, P2 terdefinisi dan mungkin kosong }
{ F.S. : P3 = P1+P2, P3 adalah polinom baru }
{ Menjumlahkan P1 + P2 dan menyimpan hasilnya di P3, P3 ≠ P1 and P3 ≠P2 }
```

```
KAMUS LOKAL
   i : integer           { indeks traversal }
   MaxDegree : integer   { derajat masukan yang maksimum antara P1 dan P2 }
```

```
ALGORITMA
   InitPol (P3)
   if P1.Degree > P2.Degree then
      MaxDegree ← P1.Degree
   else
      MaxDegree ← P2.Degree
   if MaxDegree ≠ -999 then
      i traversal [MaxDegree..0]
         P3.TabSuku_i ← P1.TabSuku_i + P2.TabSuku_i
         P3.Degree ← MaxDegree
      { derajat P3 mungkin "turun" }
      AdjustDegree(P3)
```

```
procedure SubPol (input P1, P2 : polinom, output P3 : polinom)
{ I.S. : P1, P2 terdefinisi dan mungkin kosong }
{ F.S. : P3 = P1-P2, P3 adalah polinom baru }
{ Mengurangkan P1 - P2 dan menyimpan hasilnya di P3, P3 ≠ P1 and P3 ≠ P2 }
```

```
KAMUS LOKAL
   i : integer           { indeks traversal }
   MaxDegree : integer   { derajat masukan yang maksimum antara P1 dan P2 }
```

```
ALGORITMA
   InitPol (P3)
   if P1.Degree > P2.Degree then
      MaxDegree ← P1.Degree
   else
      MaxDegree ← P2.Degree
   if MaxDegree ≠ -999 then
      i traversal [MaxDegree..0]
         P3.TabSuku_i ← P1.TabSuku_i - P2.TabSuku_i
      { derajat polinom hasil mungkin "turun" !!!! }
      AdjustDegree(P3)
```

```
procedure DerivPol (input P : polinom, output P1 : polinom)
{ I.S. : P terdefinisi, mungkin kosong }
{ F.S. : P1 adalah turunan P }
{ Membuat turunan P dan menyimpan hasilnya di P1, P1 ≠ P }
```

```
KAMUS LOKAL
   i : integer       { indeks traversal }
```

```
ALGORITMA
   InitPol (P1)
   if P.Degree ≠ -999 then
      i traversal [P.Degree-1..1]
         P1.TabSuku_i ← (i+1) * P.TabSuku_i+1
      P.TabSuku_0 ← P.TabSuku_1
      AdjustDegree(P1)   { mungkin jadi kosong !!! }
```

```
procedure AdjustDegree (input/output P : polinom)
{ I.S. : P terdefinisi, mungkin kosong. P.Degree belum tentu merupakan derajat
polinom }
{ F.S. : P terdefinisi, P.Degree berisi derajat polinom berdasarkan keadaan
P.Tabsuku }
{ Derajat hanya mungkin "turun". Search i [P.Maxdegree} terbesar, dengan
TabSuku_i ≠ 0 }
```

```
KAMUS LOKAL
    i : integer   { indeks traversal }
```

```
ALGORITMA
    if P.Degree ≠ -9999 then
        i ← P.Degree
        while (i > 0) and (P.TabSuku_i = 0) do
            i ← i - 1
        { i = 0 or P.TabSuku_i ≠ 0 }
        if (P.TabSuku_i ≠ 0) then
            P.Degree ← i
        else
            P.Degree ← -9999      { polinom jadi kosong !!!! }
    { else: polinom kosong : tidak mungkin turun derajatnya }
```

Representasi BERKAIT

Dengan representasi berkait, hanya suku yang muncul saja yang disimpan datanya. Degree setiap suku harus disimpan secara eksplisit. Operasi akan lebih efisien jika suku yang muncul diurut mulai dari derajat tertinggi sampai derajat terendah, karena suku dengan derajat paling tinggi dapat diakses secara langsung dan menunjukkan derajat dari polinom. Perhatikan bahwa dengan representasi berkait dan informasi degree secara eksplisit, maka:

- hanya suku yang muncul yang disimpan.
- terjadi penghematan memori kalau suku-suku [0..N] banyak yang tidak muncul. Sebaliknya, jika semua suku muncul, maka representasi berkait akan lebih banyak memakan memori karena setiap derajat harus disimpan secara eksplisit.
- polinom kosong menjadi sangat “natural”.

```
KAMUS
    { Definisi sebuah polinom P adalah }
    type address : ... { terdefinisi alamat sebuah suku }
    type polinom : address { alamat elemen pertama list }
    type Suku : < Degree : integer, Coefficient : integer, Next : address >
    { Polinom adalah List Suku,
      dengan elemen yang selalu terurut menurun menurut Degree }
    P1, P2, P3 : polinom
```

Dengan catatan bahwa derajat polinom dapat diketahui dari elemen pertama list berkait, dan polinom kosong adalah sebuah list kosong.

Dengan demikian, setiap operasi terhadap polinom dapat dijabarkan sebagai berikut :

1. Membentuk sebuah polinom dari pasangan harga yang dibaca dari alat masukan:
   Data yang dimasukkan adalah pasangan
   (*) <Degree : integer, Coefficient : integer >
   (1) <-999, 0>
   Proses pembentukan polinom adalah proses sekuensial untuk membaca dari masukan dan melakukan penyisipan dalam list Suku polinom yang selalu terurut menurun

menurut Degree. Harus diperhatikan bahwa di sini elemen data yang dimasukkan tidak perlu diurut. Proses membuat sebuah polinom adalah search tempat yang sesuai, sisipkan pada tempat yang sesuai tersebut.

Contoh : jika dibaca P1:

<1,4>, <2,5>, <5,7>, <8,9>, <3,4>, <-999,0>

Polinom yang dibentuk adalah polinom berderajat 9, dengan urutan penyisipan elemen list berkait sehingga terbentuk list berkait sebagai berikut (elemen terakhir selalu menunjuk ke Nil.

<1,4>

<2,5>, <1,4>

<5,7>, <2,5>, <1,4>

<8,9>,<5<7>, <2,5>, <1,4>

<8,9>,<5,7>, <3,4>,<2,5>, <1,4>

Perhatikanlah bahwa proses penyisipan harus memperhitungkan pemasukan data dengan Degree yang sama supaya didapat polinom yang absah.

2. Menuliskan sebuah polinom:
   Prosesnya adalah proses sekuensial dengan mark, traversal sebuah list polinom. Otomatis harga suku yang dituliskan hanya jika koefisiennya tidak nol, karena hanaa jika koefisien tidak nol maka suku itu muncul dalam list. Maka untuk contoh di atas (polinom terakhir), keluaran secara sederhana adalah :

```
I   P(I)
8   9
5   7
3   4
2   5
1   4
```

3. Menjumlahkan dua buah polinom P1 dan P2 dan menyimpan hasilnya pada P3:
   Menjumlahkan dua buah polinom adalah menjumlahkan suku yang berderajat sama. Maka prosesnya adalah "merging" dua buah list linier yang terurut dan setiap suku P1 dan P2 dianalisis kasusnya:
   - derajat P1 sama dengan derajat P2, jumlahkan dan sisipkan pada P3, maju ke suku P1 dan P2 yang berikutnya
   - derajat P1 < derajat P2: sisipkan suku P1 pada P3, maju ke suku P1 yang berikutnya
   - derajat P1 > derajat P2: sisipkan suku P2 pada P3, maju ke suku P2 yang berikutnya

   Contoh: Jika diberikan :

   P1: <9,4>,<7,4>, <5,5>, <4,-9> <3,2>, <2,1>, <0,10>

   P2: <9,2>, <7,3>, <4,1>,<1,2>

   Maka urutan pembentukan P3 adalah :

<9,6>	InsertLast (P3,P1+P2), Next(P1), Next(P2)
<7,7>	InsertLast (P3,P1+P2), Next(P1), Next(P2)
<5,5>	InsertLast (P3,P1), Next(P1)
<4,-8>	InsertLast (P3,P1+P2), Next(P1), Next(P2)
<3,2>	InsertLast (P3,P1), Next(P1)
<2,1>	InsertLast (P3,P1), Next(P1)
<1,2>	InsertLast (P3,P2), Next(P2)
<0,10>	InsertLast (P3,P1), Next(P1)

   Keadaan akhir P3 : <9,6>,<7,7>,<5,5>,<4,-8>,<3,2>,<2,1>,<1,2>, <0,10>

Dengan catatan bahwa InsertLast (P3, P1+P2) adalah penyisipan harga koefisien suku P1+P2 sebagai elemen terakhir P3, dan P3 diinisialisasi sebagai list kosong. Karena penyisipan selalu pada akhir list, alamat elemen terakhir list P3 selama proses berlangsung layak untuk disimpan. Algoritma merging yang akan dipakai adalah versi "AND".

4. Mengurangi dua buah polinom P1 dan P3:
   Idem dengan menjumlahkan, hanya operasi penjumlahan diganti operasi pengurangan
   Contoh: Jika diberikan :
   P1: <9,4>,<7,4>, <5,5>, <4,-9>
   P2: <9,2>, <7,3>, <4,1>,<3,2>, <2,1>, <1,2>, <0,10>
   maka urutan pembentukan P3 adalah sebagai berikut:

<9,2>	InsertLast(P3,P1-P2), Next(P1), Next(P2)
<7,1>	InsertLast(P3,P1-P2), Next(P1), Next(P2)
<5,5>	InsertLast(P3,P1), Next(P1)
<4,-10>	InsertLast(P3,P1-P2), Next(P1), Next(P2)
<3,-2>	InsertLast(P3,P2), Next(P2)
<2,-1>	InsertLast(P3,P2), Next(P2)
<1,-2>	InsertLast(P3,P2), Next(P2)
<0,-10>	InsertLast(P3,P2), Next(P2)

   Dan keadaan akhir P3 adalah :
   <9,2>, <7,1>, <5,5>, <4,-10>, <3,-2>, <2,-1>, <1,-2>, <0,-10>

5. Membuat turunan dari sebuah polinom:
   Prosesnya adalah proses sekuensial, traversal list P, untuk setiap suku ke-i, i >0: yaitu $a_i x^i$ pada polinom P, dihitung i * a_i dan disisipkan P1 sebagai suku ke(i-1).
   Seperti pada proses penjumlahan, list P1 dibentuk dengan penyisipan elemen terakhir, dan P1 diinisialisasi sebagai list kosong. Karena penyisipan selalu pada akhir list, maka alamat elemen terakhir list P1 selama proses berlangsung layak untuk disimpan.
   Contoh: Jika diberikan:
   P: <9,4>, <7,4>, <5,5>, <4,-9>, <3,2>, <2,1>, <0,10>
   maka P1 akan mempunyai harga : <8,36>, <6,28>, <4,25>, <3,-36>, <2,6>, <1,2>

Perhatikan penulisan algoritma berikut. Teks algoritma dituliskan secara lojik, dan tergantung implementasi dilakukan aturan penulisan ulang seperti diberikan pada Tabel Polinom-I sebagai berikut

Tabel Polinom-I :

Kamus dan Aturan penulisan ulang representasi lojik dan berkait untuk polinom

Representasi lojik berkait	**Representasi fisik berkait dengan Pointer**	**Representasi fisik berkait dengan Tabel**
KAMUS UMUM : type address:... type Suku: <Degree:integer, Coefficient:integer, Next:address> type polinom:address P : polinom Pt : address	KAMUS UMUM: type address:pointer to Suku type Suku: <Degree:integer, Coefficient:integer Next:address> type polinom:address P : polinom Pt : address	KAMUS UMUM: constant NMax:integer=100 type address:integer[0..NMax] type Suku: <Degree:integer, Coefficient:integer, Next:address> type polinom:address TabSuku:array[0..NMax] of Suku FirstAvail:address P : polinom Pt : address
AKSES: First(P) Next(Pt) Degree(Pt) Coefficient(Pt)	AKSES: P Pt↑.Next Pt↑.Degree Pt↑.Coefficient	AKSES: P $TabSuku_{Pt}$.Next $TabSuku_{Pt}$.Degree $TabSuku_{Pt}$.Coefficient
PRIMITIF ALOKASI/DEALOKASI: { tidak perlu } AllocSuku(Pt) DeallocSuku(Pt)	PRIMITIF ALOKASI/DEALOKASI: { sistem } Aloccate(Pt) DeAllocate(Pt)	PRIMITIF ALOKASI/DEALOKASI: { Harus direalisasi } InitTabSuku AlocTabSuku(Pt) DealocTabSuku(Pt)

Kamus umum dan program utama untuk representasi berkait adalah:

```
Program POLINOMIAL1
{ Representasi BERKAIT, dengan notasi LOJIK }

KAMUS
{ Struktur data untuk representasi polinom }
   type Address : ... {type terdefinisi }
   type Suku : < Degree : integer,
                 Coefficient : integer,
                 Next : address >
   type polinom : address
   constant Nil : address { untuk address tidak terdefinisi }
   P1, P2: polinom { operan }
   P3 : polinom    { hasil }

{ Untuk interaksi: }
   Finish : boolean         { mengakhiri proses }
   Pilihan : integer [0..5] { nomor tawaran }

{ Primitif operasi polinom untuk operasi internal }
   procedure AllocSuku (output Pt : address)  { Alokasi sebuah suku }
   procedure DeAllocSuku (input Pt : address) { Dealokasi sebuah suku }
   procedure InitListPol (output P : polinom)
   { Membuat polinom kosong P }
   procedure InsertLast (input P : address, input/output P : polinom,
                         input/output Last : address)
   { Insert P sesudah elemen terakhir P dengan address elemen terakhir = Last }

{ Primitif operasi polinom yang ditawarkan ke pengguna }
   procedure CreateListPol (output P1 : polinom) { Mengisi polinom P1 }
   procedure TulisListPol (input P : polinom)   { Menulis polinom P }
   procedure AddListPol (input P1, P2 : polinom, output P3 : polinom)
   { Menjumlahkan P1 + P2 dan menyimpan hasilnya di P3,
     P3 ≠ P1 dan P3 ≠P2 }
   procedure SubListPol (input P1, P2 : polinom, output P3 : polinom)
   { Mengurangkan P1 - P2 dan menyimpan hasilnya di P3,
     P3 ≠ P1 dan P3 ≠P2 }
   procedure DerivListPol (input P : polinom, output P1 : polinom)
   { Membuat turunan P dan menyimpan hasilnya di P1, P1 ≠ P }

ALGORITMA
{ Sama dengan untuk representasi kontigu }
```

```
procedure InitListPol (output P : Polinom)
{ I.S. sembarang; F.S. P polinom kosong terbentuk}
{ Membuat polinom kosong }

KAMUS LOKAL

ALGORITMA
   P ← Nil      { Definisi Polinom kosong }
```

```
procedure CreateListPol (output P1 : Polinom)
{ I.S. : sembarang }
{ F.S. : Polinom P terdefinisi derajat dan koefisien-koefisiennya jika tidak
"kosong" }
{ Mengisi list polinom P1 }
```

```
KAMUS LOKAL
   Deg, Coef  : integer  { input pasangan data derajat dan koefisien }
   Pt, PrecPt : address  { address traversal, PrecPt adalah sebelum Pt }
   PNew       : address  { address alokasi, yang nantinya disisipkan }
   Found      : boolean
```

```
ALGORITMA
   { Skema proses sekuensial tanpa penanganan kasus kosong }
   InitListPol(P1)
   input (Deg, Coef)     { Input data }
   while (Deg ≠ -999) do
      AllocSuku(PNew)    { Alokasi sebuah suku polinom }
      Degree(PNew) ← Deg; Coefficient(PNew) ← Coef; Next(PNew)← Nil
      { Search tempat yang tepat, suku terurut mengecil menurut Degree }
      { Catat address sebelumnya untuk penyisipan }
      Pt ← P1; PrecPt ← Nil; Found ← false
      while (Pt ≠ Nil) and (not Found) do
         if Deg ≥ Degree(Pt) then
             Found ← true
         else PrecPt ← Pt; Pt ← Next(Pt)
      { Found : Deg ≥ Degree(Pt) or Pt = Nil, sisipkan P1 setelah PrecPt }
      if Deg > Degree(Pt) then
         depend on PrecPt
            PrecPt = Nil : { insert first }
                           Next(PNew) ← P1; P1 ← PNew
            PrecPt ≠ Nil : { insert after PrecPt }
                           Next(PNew) ← Next(PrecPt); Next(PrecPt) ← PNew
      { else error, duplikasi suku dengan derajat sama : tidak diproses }
      input (Deg, Coef)     { Next input data }
```

```
procedure TulisListPol (input P : polinom)
{ I.S. : P terdefinisi dan mungkin kosong }
{ F.S. : Menulis polinom P sesuai dengan spesifikasi }
```

```
KAMUS LOKAL
   Pt : address    { address traversal list }
```

```
ALGORITMA
   { Traversal list P dengan skema pemrosesan dengan penanganan kasus kosong,
     tulis setiap suku }
   Pt ← P
   if ( Pt = Nil) then
      output ("Polinom kosong")
   else
      output ("I","P(I)")        { Inisialisasi, tulis judul }
      repeat
         output (Degree(Pt), Coefficient(Pt))
         Pt ← Next(Pt)
      until (Pt = Nil)
```

```
procedure InsertLast (input Deg, Coef : integer, input/output P : polinom,
                      input/output Last : address)
{ I.S. : P mungkin kosong, <Deg, Coef> adalah derajat dan koefisien suku yang
         akan di-insert sebagai elemen terakhir }
{ F.S. : <Deg,Coeff> di-copy ke dalam elemen yang ditunjuk PNew, PNew di-insert
         sebagai elemen terakhir P, Last menunjuk elemen terakhir }
{ Alokasi elemen baru dianggap selalu berhasil }
```

```
KAMUS LOKAL
   PNew : address
```

```
ALGORITMA
   AllocSuku(PNew)
   Degree(PNew) ← Deg
   Coefficient(PNew) ← Coef
   Next(PNew) ← Nil
   if (P = Nil) then { insert first }
        P ← PNew
   else { insert after Last }
      Next(Last) ← PNew
   Last ← PNew
```

```
procedure AddListPol (input P1, P2 : polinom, output P3 : polinom)
{ I.S. : P1, P2 terdefinisi dan mungkin kosong }
{ F.S. : P3 = P1 + P2, P3 polinom baru }
{ Menjumlahkan P1 + P2 dan menyimpan hasilnya di P3, P3 ≠ P1 dan P3 ≠P2 }
```

```
KAMUS LOKAL
   Pt1, Pt2 : address
   Last : address        { alamat elemen terakhir P3 }
   Sum : integer         { penjumlahan coefficient P1 dan P2 }
```

```
ALGORITMA
   InitListPol(P3); Last ← Nil; Pt1 ← P1; Pt2 ← P2
   while (Pt1 ≠ Nil) and (Pt2 ≠ Nil) do
      depend on Pt1, Pt2
         Degree(Pt1) > Degree(Pt2) :
                   { Salin & insert Pt1 ke P3, Pt1 maju }
                   InsertLast(Degree(Pt1),Coefficient(Pt1),P3,Last)
                   Pt1 ← Next(Pt1)
         Degree(Pt1) = Degree(Pt2) :
                   { Jumlahkan Coefficient, salin & insert jika tidak nol,
                     Pt1 & Pt2 maju }
                   Sum ← Coefficient(Pt1) + Coefficient(Pt2)
                   if (Sum ≠ 0) then
                       InsertLast(Degree(Pt1),Sum,P3,Last)
                   { else tidak di-insert karena Coefficient bernilai nol }
                   Pt1 ← Next(Pt1); Pt2 ← Next(Pt2)
         Degree(Pt1) < Degree(Pt2) :
                   { Salin & insert Pt2, Pt2 maju }
                   InsertLast(Degree(Pt2),Coefficient(Pt2),P3,Last)
                   Pt2 ← Next(Pt2)
   { Pt1 = Nil or Pt2 = Nil }
   while (Pt1 ≠ Nil) do { proses sisa P1 }
      InsertLast(Degree(Pt1),Coefficient(Pt1),P3,Last)
      Pt1 ← Next(Pt1)
   while (Pt2 ≠ Nil) do { proses sisa P2 }
      InsertLast(Degree(Pt2),Coefficient(Pt2),P3,Last)
      Pt2 ← Next(Pt2)
```

```
procedure SubListPol (input P1, P2 : polinom, output P3 : polinom)
{ I.S. : P1, P2 terdefinisi dan mungkin kosong }
{ F.S. : P3 = P1-P2, P3 polinom baru }
{ Mengurangkan P1 - P2 dan menyimpan hasilnya di P3, P3 ≠ P1 dan P3 ≠ P2 }
```

```
KAMUS LOKAL
    Pt1, Pt2 : address    { address traversal }
    PNew : address        { alokasi sebuah suku baru }
    Last : address        { alamat elemen terakhir P3 }
    Diff : integer        { pengurangan coefficient P1 dan P2 }
```

```
ALGORITMA
    InitListPol(P3); Last ← Nil; Pt1 ← P1; Pt2 ← P2
    while (Pt1 ≠ Nil) and (Pt2 ≠ Nil) do
        depend on Pt1, Pt2
            Degree(Pt1) > Degree(Pt2) :
                    { Salin & insert Pt1 ke P3, Pt1 maju }
                    InsertLast(Degree(Pt1),Coefficient(Pt1),P3,Last)
                    Pt1 ← Next(Pt1)
            Degree(Pt1) = Degree(Pt2) :
                    { Kurangkan Coefficient, Salin dan insert jika tidak nol,
                      Pt1 & Pt2 maju }
                    Diff ← abs(Coefficient(Pt1) - Coefficient(Pt2))
                    if (Diff ≠ 0) then
                       InsertLast(Degree(Pt1),Diff,P3,Last)
                    { else tidak di-insert karena Coefficient bernilai nol }
                    Pt1 ← Next(Pt1); Pt2 ← Next(Pt2)
            Degree(Pt1) < Degree(Pt2) :
                    { Salin&insert Pt2 ke P3,Pt2 maju }
                    InsertLast(Degree(Pt2),Coefficient(Pt2),P3,Last)
                    Pt2 ← Next(Pt2)
    { Pt1 = Nil or Pt2 = Nil : tuliskan sisanya ke P3 }
    { Proses sisa P1, jika ada }
    while (Pt1 ≠ Nil) do { Proses sisa P1 }
        { Alokasi sebuah suku polinom }
        InsertLast(Degree(Pt1),Coefficient(Pt1),P3,Last)
        Pt1 ← Next(Pt1)
    { Proses sisa P2, jika ada }
    while (Pt2 ≠ Nil) do { Proses sisa P2 }
        { Alokasi sebuah suku polinom }
        InsertLast(Degree(Pt2),Coefficient(Pt2),P3,Last)
        Pt2 ← Next(Pt2)
```

```
procedure DerivListPol (input P : polinom, output P1 : polinom)
{ I.S. P terdefinisi, mungkin kosong }
{ F.S. P1 adalah turunan P, polinom baru }
{ Membuat turunan P dan menyimpan hasilnya di P1, P1 ≠ P }
```

```
KAMUS LOKAL
   Pt : address       { address untuk traversal }
   Last : address     { alamat elemen terakhir P1 }
```

```
ALGORITMA
   InitListPol(P1); Last ← Nil
   Pt ← P
   if (Pt ≠ Nil) then
      { Minimal satu elemen, proses elemen terakhir secara khusus }
      while (Next(Pt) ≠ Nil) do
         { Salin dan Insert Pt sebagai elemen terakhir P1 }
         InsertLast(Degree(Pt)-1,Degree(Pt)*Coefficient(Pt),P1,Last)
         { Next elemen list polinom }
         Pt ← Next(Pt)
      { Next(Pt) = Nil), elemen terakhir,
        cek apakah berderajat 0, jika tidak baru diproses }
      if (Degree(Pt) ≠ 0) then
         InsertLast(Degree(Pt)-1,Degree(Pt)*Coefficient(Pt),P1,Last)
```

Studi Kasus: Suku-Suku Polinom Hasil Berasal dari Operan

Persoalannya adalah jika pada operasi penjumlahan, pengurangan dan derivasi pada paket polinom tersebut suku dari polinom hasil berasal dari polinom operan, dan hasilnya disimpan pada salah satu operan. Contoh: Jika P1 dan P2 adalah polinom, maka hendak dilakukan operasi:

P1 = P1 + P2
P1 = P1 - P2
P1 = P1'

Pada persoalan ini, polinom P1 “tidak ada lagi”, karena “ditimpa” oleh hasil penjumlahan, Suku P2 “melebur menjadi suku polinom P1 yang baru.

Implikasi terhadap perubahan spesifikasi tidak sama untuk representasi kontigu dan representasi berkait. Berikut ini hanya dibuat studi perbandingan untuk algoritma penjumlahan P1 = P1 + P2.

Representasi KONTIGU

Untuk representasi kontigu, algoritma penjumlahan dua buah polinom tidak berubah, hanya dengan menggantikan penulisan P3 menjadi P1.

```
procedure AddPolBis (input/output P1, P2 : polinom)
{ I.S. P1 dan P2 terdefinisi, mungkin kosong }
{ F.S. P1 = P1 + P2. }
{ Representasi kontigu, menjumlahkan P1 + P2 dan menyimpan hasilnya di P1 }
```

```
KAMUS LOKAL
   i : integer           { indeks traversal }
   MaxDegree : integer   { derajat masukan maksimum antara P1 dan P2 }
```

```
ALGORITMA
   if P1.Degree > P2.Degree then
      MaxDegree ← P1.Degree
   else
      MaxDegree ← P2.Degree
   if MaxDegree ≠ -9999 then
      P1.Degree ← MaxDegree
      i traversal [MaxDegree..0]
         P1.TabSuku_i ← P1.TabSuku_i + P2.TabSuku_i
      AdjustDegree(P1)
   {else : kedua polinom kosong. Tidak ada operasi. }
```

Representasi BERKAIT

Perubahan spesifikasi hasil operasi tersebut membawa implikasi cukup banyak pada representasi berkait.

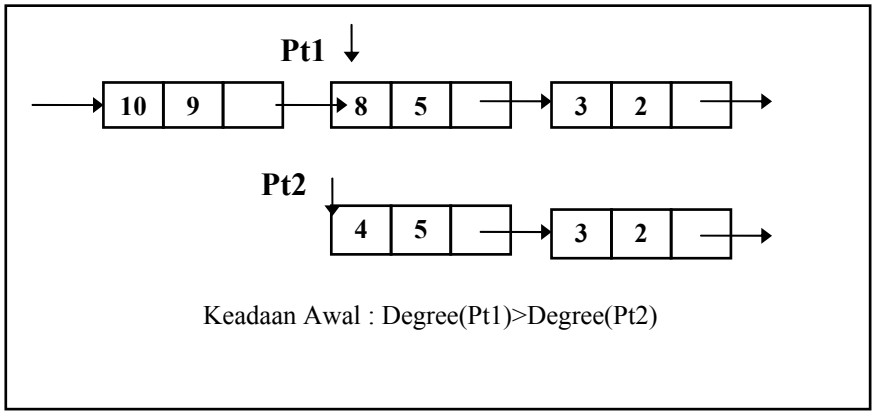
Prosesnya adalah traversal kedua list, setiap saat kita mengelola dua buah pointer Pt1 untuk traversal Polinom P1 dan Pt2 untuk traversal Polinom P2. Untuk setiap pasangan harga elemen yang ditunjuk oleh Pt1 dan Pt2, diadakan analisis kasus terhadap harga field Degree dan Coefficient. Operasi penjumlahan dapat mengakibatkan empat kemungkinan:.

1. Degree(Pt1) > Degree(Pt2) : Tidak ada perubahan terhadap elemen list, hanya pointer Pt1 yang maju, karena elemen Pt1 tetap menjadi elemen Polinom hasil.
2. Degree(Pt1)=Degree(Pt2)

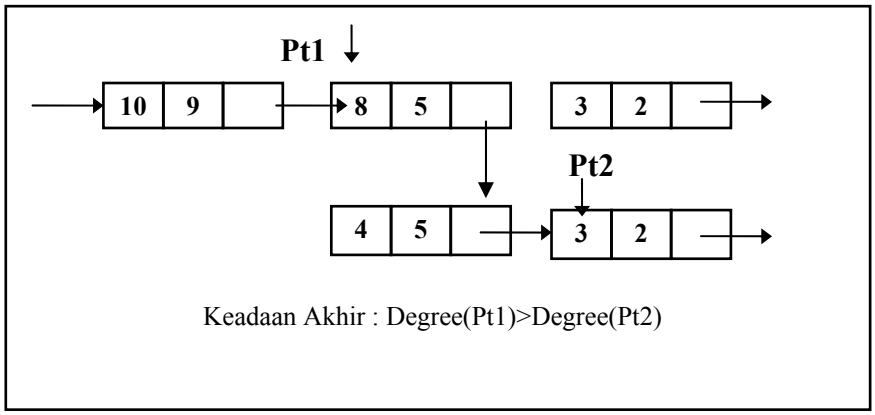
a. Hasil penjumlahan koefisien tidak sama dengan nol: Pengubahan nilai Coefficient pada Pt1 dan penghapusan pada Pt2 karena kedua Suku dijumlahkan membentuk sebuah suku baru, dalam hal ini yang dipertahankan adalah suku dari Pt1.
b. Hasil penjumlahan koefisien sama dengan nol: Penghapusan elemen Pt1 dan Pt2 karena hasil penjumlahan adalah nol, sehingga kedua Suku yang menghasilkan nol harus dihapus baik dari Pt1 maupun dari Pt2

3. Degree(Pt1) < Degree(Pt2) : Penambahan elemen ke polinom hasil yang asalnya dari Pt2.

Kasus 1: Degree(Pt1) > Degree(Pt2)

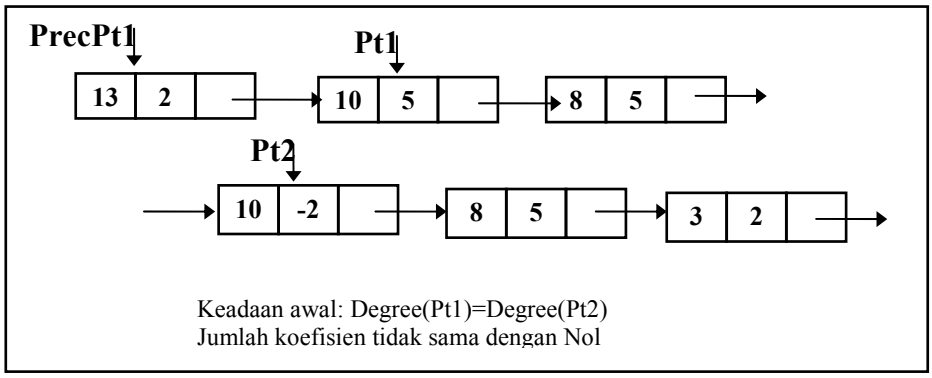


Keadaan Awal : Degree(Pt1)>Degree(Pt2)

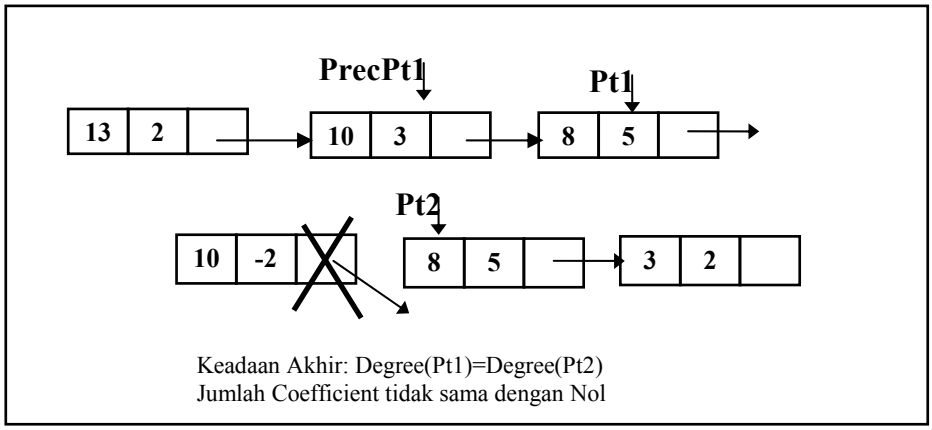


Keadaan Akhir : Degree(Pt1)>Degree(Pt2)

Kasus 2a: Pt1 = Pt2 dengan hasil penjumlahan koefisien tidak sama dengan nol: prosesnya adalah UPDATE(Pt1), hapus Pt2.

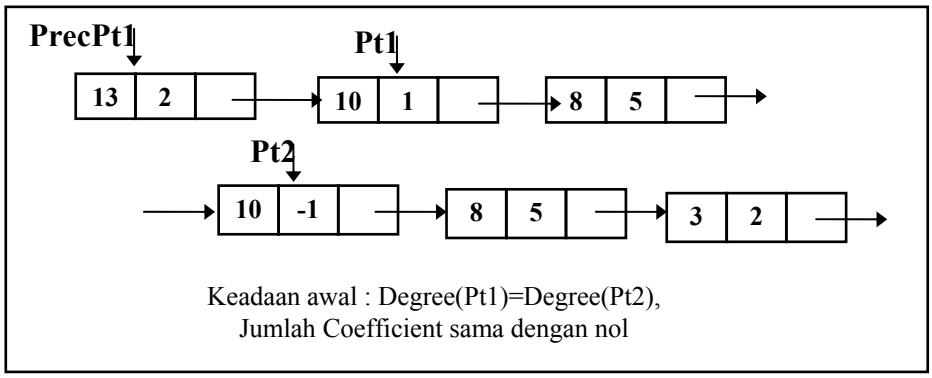


Keadaan awal: Degree(Pt1)=Degree(Pt2)
Jumlah koefisien tidak sama dengan Nol

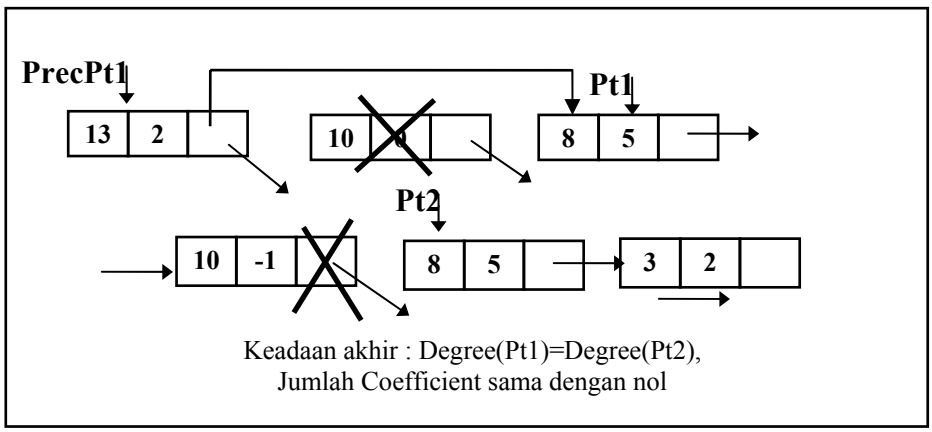


Keadaan Akhir: Degree(Pt1)=Degree(Pt2)
Jumlah Coefficient tidak sama dengan Nol

Kasus 2b. Degree(Pt1) = Degree(Pt2) dan hasil penjumlahan koefisien sama dengan Nol: Hapus (Pt1), Hapus(Pt2)

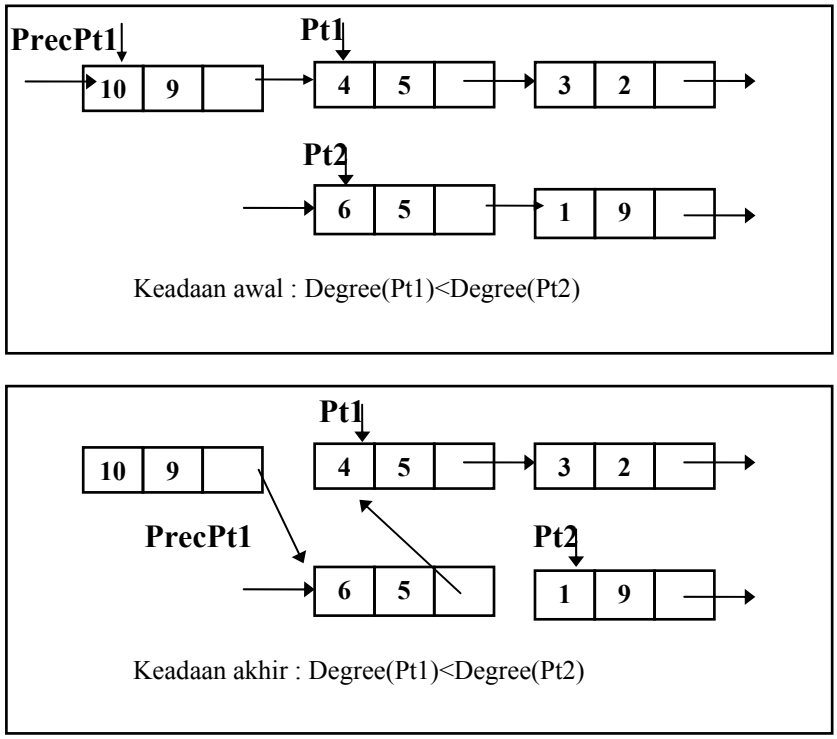


Keadaan awal : Degree(Pt1)=Degree(Pt2),
Jumlah Coefficient sama dengan nol



Keadaan akhir : Degree(Pt1)=Degree(Pt2),
Jumlah Coefficient sama dengan nol

Kasus 3 : Degree(Pt1) < Degree(Pt2) : Suku Pt2 dijadikan suku dari polinom hasil.



Perhatikan penulisan algoritma berikut. Teks algoritma dituliskan secara lojik, dan tergantung implementasi dilakukan aturan penulisan ulang sesuai dengan tabel Polinom I pada pembahasan representasi berkait

```
procedure AddListPolBis (input/output P1, P2 : polinom)
{ Representasi BERKAIT : }
{ I.S. : P1 dan P2 sembarang. }
{ F.S. : P1 = P1 + P2 }
{ Menjumlahkan P1 + P2 dan menyimpan hasilnya di P1 }
```

```
KAMUS LOKAL
   Pt1, Pt2 : address    { indeks traversal }
   PrecPt1  : address
   PDel     : address    { alokasi sebuah suku baru }
```

```
ALGORITMA
   if P1 = Nil then    { kaitkan P2 ke P1 }
   P1 ← P2
   else   { P1 minimal mempunyai 1 elemen }
      Pt1 ← P1; PrecPt1 ← Nil { predesesor elmt pertama tak terdefinisi }
      Pt2 ← P2
      while (Pt1 ≠ Nil) and (Pt2 ≠ Nil) do
         depend on Pt1, Pt2
            Degree(Pt1) > Degree(Pt2): { Maju ke elmt P1 yg berikut }
                  PrecPt1 ← Pt1; Pt1 ← Next(Pt1)
            Degree(Pt1) = Degree(Pt2): { Jumlahkan koefisien, cek hasil }
                  if Coefficient(Pt1) + Coefficient(Pt2) = 0 then
                     { Hapus Pt1, Pt1 ke elmt berikut, PrecPt1 tetap }
                     { Delete (PrecPt1,Pt1), mungkin delete First!!! }
                     PDel ← Pt1
                     if (Pt1 = P1) then
                        P1 ← Next(P1)  { Delete first }
                        Pt1 ← P1
                     else   { Delete after PrecPt1 }
                        Next(PrecPt1) ← Next(Pt1)
                        Pt1 ← Next(Pt1)
                     { Dealokasi PDel }
                     DeallocSuku(PDel)
                     { Hapus Pt2, maju }
                     PDel ← Pt2
                     Pt2 ← Next(Pt2)
                     DeallocSuku(PDel)
                  else  { Update koefisien Pt1 }
                     Coefficient(Pt1) ← Coefficient(Pt1) + Coefficient(Pt2)
                     PrecPt1 ← Pt1; Pt1 ← Next(Pt1)
                     Pt2 ← Next(Pt2)
            Degree(Pt1) < Degree(Pt2): { Insert Pt2 sebelum Pt1 }
                  { InsertAfter(Pt2,PrecPt1) }
                  Temp ← Next(PrecPt1); Next(PrecPt1) ← Pt2
                  Pt2 ← Next(Pt2); Next(Pt2) ← Temp
                  PrecPt1 ← Pt2
      { Pt1 = Nil or Pt2 = Nil }
      if (Pt1 = Nil) then     { gabungkan sisa Pt2 dg elemen terakhir Pt1 }
         if (PrecPt1 ≠ Nil) then
           Next(Pt2) ← Pt1  { insert first }
         else
           Next(PrecPt1) ← Pt2
```

Latihan Soal

1. Buatlah analisis perbandingan dari segi memori dan proses untuk semua representasi yang dibahas.
2. Polinom “kosong” merupakan fenomena yang menarik untuk didiskusikan. Coba berikan sedikit ulasan mengenai "polinom kosong".
3. Bagaimana jika urutan penulisan polinom dikehendaki mulai dari derajat terkecil sampai terbesar? Jika urutan Suku disusun menaik menurut Degree agar mempermudah penulisan, apa implikasinya terhadap representasi polinom? Tuliskanlah algoritmanya.
4. Bagaimana jika pada representasi berkait list suku tidak diurut?
5 Perhatikanlah bahwa menu pada program utama menimbulkan masalah.
   Jika pemakai memanggil pilihan 2 s.d. 5 tanpa pernah membentuk polinom P1 dan P2, maka semua operasi kecuali membentuk polinom tidak dapat dilakukan karena polinom belum terdefinisi. Usulkan beberapa solusi untuk persoalan ini.
   Create polinom boleh diaktifkan kapan saja, sedangkan operasi lain hanya boleh ditawarkan jika sudah ada polinom yang terdefinisi. Pikirkanlah hal ini, dan tuliskan modul interaksi yang memungkinkan sistem menawarkan operasi dan memanggil prosedur yang telah tersedia.

Studi Kasus 2: Kemunculan Huruf dan Posisi Pada Pita Karakter

Deskripsi Persoalan

Diberikan sebuah pita karakter yang hanya terdiri dari huruf kecil ['a'..'z'], karakter 'blank' dan diakhiri titik. Didefinisikan karakter ke-n sebagai nomor urutan karakter pada pita dengan karakter pertama (bukan titik) yang dibaca sebagai karakter yang pertama. Tuliskan algoritma untuk menuliskan : huruf yang muncul, muncul sebagai karakter keberapa saja, dan berapa kali muncul. Karakter "blank" diabaikan. Output harus ditulis sesuai urutan abjad, dan hanya ditulis jika ada kemunculan. Pita maksimum berisi 100 karakter.

Beberapa contoh isi pita dan output diberikan sebagai berikut:

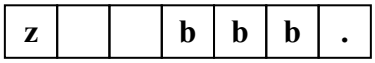
1. Isi pita :

a	k	u		i	n	i		m	i	s	k	i	n	.

Output :

```
a :
      1
Muncul 1 kali

i:
      5
      7
      10
      13
Muncul 4 kali

k:
      2
      12
Muncul 2 kali

m:
      9
Muncul 1 kali

n:
      6
      14
Muncul 2 kali

s:
      11
Muncul 1 kali

u:
      3
Muncul 1 kali
```

2. Isi pita :



Output :

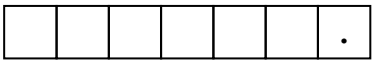
```
b:
      4
      5
      6
Muncul 3 kali

z:
      1
Muncul 1 kali
```

3. Isi pita :

Output :

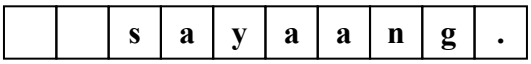
```
Pita kosong, tidak ada isinya
```

4. Isi pita :



Output :

5. Isi pita :



Output :

```
a:
      4
      6
      7
Muncul 3 kali

g:
      9
Muncul 1 kali

n:
      8
Muncul 1 kali


s:
      3
Muncul 1 kali

s:
      5
Muncul 1 kali
```

Solusi Pertama: Struktur Data Sederhana, Proses Tidak Efisien

Persoalan ini sepintas mirip dengan menghitung frekuensi setiap huruf yang muncul pada pita. Jika hanya frekuensi setiap huruf yang harus dihitung, maka kita dapat memakai algoritma menghitung 'A" dengan melakukan analisa kasus terhadap setiap huruf. Tetapi pada persoalan ini, selain kemunculan huruf, posisi huruf harus dicatat. Jika hanya kemunculan yang dihitung, maka hanya dibutuhkan satu counter untuk setiap huruf dan persoalan dapat diselesaikan dengan skema menghitung kemunculan setiap huruf. Tetapi persoalannya tidak sesederhana ini.

Persoalan pertama adalah bahwa posisi sebuah huruf mungkin tidak akan muncul, atau mungkin akan muncul lebih dari satu kali. Dalam hal ini, jelas bahwa posisi setiap huruf yang muncul pada pita yang harus dicatat adalah sebuah list yang elemennya adalah posisi karakter pada pita. Tanpa mencatat posisi sebagai elemen list, informasi posisi sudah hilang ketika karakter pada jendela maju ke karakter berikutnya. Persoalan kedua adalah urutan penulisan output. Urutan penulisan adalah urutan sesuai abjad, yang pada kasus umum berbeda dengan urutan kemunculan huruf pada pita.

Solusi pertama persoalan ini adalah dengan melakukan **proses berulang-ulang** terhadap pita, dan setiap kali kita akan menghitung kemunculan huruf yang sedang diproses (urut abjad), dan jika muncul akan dituliskan segera posisinya. Solusi ini tidak membutuhkan tempat penyimpanan untuk posisi (karena segera di-print), namun mengharuskan traversal isi pita sebanyak 26 kali (dari 'a'..'z'). Solusi ini sangat hemat memori (hanya perlu satu counter untuk kemunculan semua huruf dan hanya perlu satu integer untuk mencatat posisi), namun akan memakan waktu lama karena pembacaan pita yang berulang-ulang. Ini adalah contoh bahwa struktur data sangat sederhana dan memakai sedikit memori, namun algoritma "rumit" dan tidak efisien. Algoritma untuk solusi ini adalah :

```
Program MUNCULH1
{ Diberikan sebuah pita karakter hanya berisi blank dan 'a'..'z', menghitung
kemunculan setiap huruf dan posisi kemunculan, dengan output sesuai spesifikasi
pada halaman 1 }

KAMUS
   N : integer { banyaknya kemunculan huruf yang sedang diproses pada
                 pita }
   Posisi : integer  { posisi karakter yang sedang dibaca pada pita }

ALGORITMA
   START
   if (CC = '.') then
      output ("Pita kosong")
   else
    C traversal['a'..'z']
      N ← 0              { Banyaknya kemunculan huruf C adalah Nol }
      Posisi ← 1         { posisi pada karakter pertama }
      repeat
         if (CC = C) then
           N ← N + 1     { Huruf C muncul }
           if (N = 1) then
                output (C) { C hanya ditulis sekali pd awal kemunculan }
           output (posisi)
         ADV
         Posisi ← Posisi + 1   { Posisi bertambah 1 karena ADV }
      until (CC = '.')
      if (N > 0) then       { Hanya menulis jika C muncul minimal 1 kali}
         output ("Muncul ", N, " kali")
      START { untuk memproses huruf yang berikutnya }
   { CC = '.' semua huruf sudah diperiksa terhadap C dan sudah ditulis}
```

Solusi ini ingin dihindari, traversal terhadap pita ingin dilakukan satu kali saja. Karena traversal hanya ingin dilakukan satu kali, maka posisi merupakan list yang harus dikelola. Selain list huruf, kita harus mengelola list posisi dan kemunculan. Ada beberapa pilihan struktur data yang akan diberikan berikut ini.

Solusi Kedua: Struktur Data List

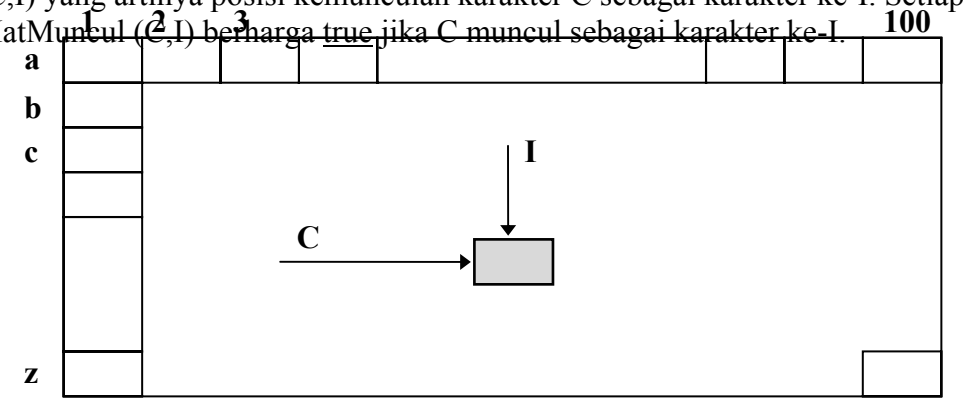
Posisi kemunculan setiap karakter harus dicatat, supaya dapat dituliskan kembali. Penulisan hanya dapat dilakukan setelah pita selesai diproses, dengan traversal isi pita yang hanya dilakukan satu kali.

Secara lojik, posisi kemunculan sebuah karakter direpresentasi oleh sebuah harga integer: <Posisi: integer> dan untuk setiap karakter yang mungkin muncul, harus dibentuk list posisi kemunculannya. Karena ada 26 karakter ['a'..'z'] maka kita harus mengelola 26 list linier, sebuah list linier untuk setiap karakter.

Representasi KONTIGU List Berkait dan List Posisi

Dengan representasi kontigu, list kemunculan dapat direpresentasi dalam sebuah tabel yang setiap elemennya berisi kemunculan keberapa (posisi). Karena pita maksimum berisi 100 karakter, maka setiap list direpresentasi oleh tabel berukuran 100, dengan indeks [1..100]. Indeks ke-i pada tabel menyatakan bahwa karakter muncul pada posisi ke-i. Karena ada 26 karakter, maka kita mengelola 26 tabel dengan elemen kemunculan ber-*type* boolean.

Salah satu pilihan untuk merepresentasikan list huruf dan list kemunculan menjadi satu struktur adalah dengan memilih struktur matriks. Setiap baris matriks dengan ciri baris C (karakter), akan mempunyai 100 kolom dengan setiap kolom yang menyatakan posisi kemunculan pada pita. Maka kemunculan adalah sebuah matriks boolean, MatMuncul (C,I) yang artinya posisi kemunculan karakter C sebagai karakter ke-I. Setiap elemen MatMuncul (C,I) berharga true jika C muncul sebagai karakter ke-I.



Algoritma menjadi sederhana dan terdiri dari 2 pass: traversal pita untuk membentuk matriks, tulis isi matriks. Secara garis besar sketsa algoritma adalah:

```
Inisialisasi matriks kemunculan
Traversal pita :
       untuk setiap huruf C (yang diketahui posisinya I),
       isi nilai MatMuncul(C,I) dengan true
Tulis output berdasarkan keadaan matriks
```

Algoritma secara detil diberikan berikut ini:

```
Program Kemunculan1
{ Representasi KONTIGU dengan matriks boolean }
{ Skema proses sekuensial dengan mark, dengan penanganan kasus kosong }

KAMUS
    constant Nmax : integer = 100 { maksimum isi pita yang diproses adalah
                                    100 }
    MatMuncul : matrix ['a'..'z', 1..Nmax] of boolean
    { MatMuncul(C,I) adalah kemunculan karakter C sebagai urutan ke-I
      dalam pita }
    Count : integer [1..Nmax] { jumlah kemunculan sebuah huruf }
    Posisi : integer          { posisi karakter pada pita }
    JumlahKar : integer       { banyaknya karakter pada pita }
    C : character ['a'..'z']  { indeks traversal matriks }
    i, j : integer [1..NMax]  { kemunculan karakter,
                                juga indeks traversal matriks }
```

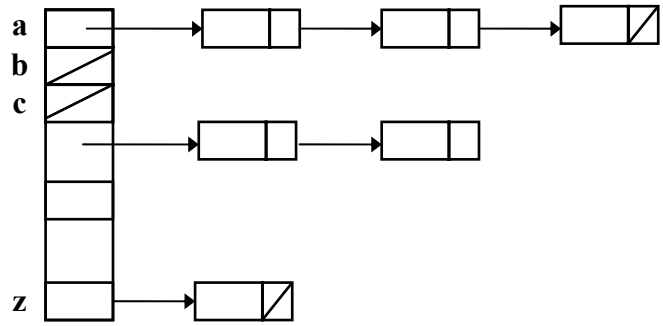
```
ALGORITMA
    START
    if CC = '.' then
       output ("Pita kosong, tidak ada isinya")
    else
       { Inisialisasi : traversal, mengosongkan matriks kemunculan }
       C traversal ['a'..'z']
          i traversal [1..100]
             MatMuncul C,i ← false
       Posisi ← 1
       { Proses sekuensial satu kali traversal isi pita }
       repeat
          depend on CC :
             CC = ' ' : -
             CC ≠ ' ' : MatMuncul CC,Posisi ← true
          Posisi ← Posisi + 1
          ADV
       until (CC = '.') or (Posisi > 100)
       { Penulisan hasil : traversal }
      JumlahKar ← Posisi - 1
       C traversal ['a'..'z']
          { Untuk setiap karakter: }
          { Search sampai ketemu true, berarti ada huruf ybs. muncul }
          i ← 1
          while (i < JumlahKar) and (not MatMuncul C,i) do
             i ← i + 1
          { i = Posisi or MatMuncul(C,i) }
          depend on MatMuncul(C,i)
             not MatMuncul C,i : - { berarti huruf C tidak muncul }
             MatMuncul C,i     : { berarti huruf C muncul }
                output (C)
                Count ← 1; output(i)
                if (i = JumlahKar) then
                   { outputkan yg muncul ,
                     traversal menghitung dimana saja muncul }
                   j traversal [i +1..100]
                     if MatMuncul C,j then
                        Count ← Count + 1; output(j)
                   output ("Muncul ", Count, " kali")
```

Skema solusi di atas untuk penulisan hasil adalah: **Search** kemunculan pertama pada suatu *row*, kemudian **traversal** sisanya. Variasi dari solusi di atas: Pengulangan "*search*" kemunculan sampai sebuah *row* selesai diproses.

Representasi BERKAIT Posisi Kemunculan, Representasi KONTIGU Huruf

Salah satu keberatan dari pemilihan struktur matriks di atas adalah harus mengetahui jumlah karakter pada pita yang maksimum untuk memesan matriks. Padahal pada umumnya jumlah karakter pada pita tidak diketahui. Keberatan kedua adalah bahwa kita harus memesan tabel posisi sejumlah karakter maksimum pada pita, untuk setiap karakter, dan bukannya jumlah maksimum karakter pada pita untuk seluruh karakter yang mungkin muncul.

Untuk menghindari pemborosan memori, list posisi akan disimpan secukupnya, dengan representasi lojik secara berkait yang dapat digambarkan sebagai berikut :



List huruf ‘a’..’z’ disimpan dalam tabel (sebagai list kontigu) karena jelas domain nilainya. Huruf menjadi indeks tabel, sedangkan elemen tabel berisi alamat elemen pertama list posisi huruf yang bersangkutan. Jadi, alamat elemen pertama dari setiap lit posisi kemunculan disimpan dalam sebuah tabel dengan huruf abjad (huruf kecil) sebagai indeks, karena ['a'..'z'] mempunyai keterurutan. Dengan demikian, jika C muncul, maka C dipakai sebagai indeks untuk mengakses address elemen pertama list, untuk selanjutnya dilakukan traversal sampai elemen terakhir untuk menyisipkan sebuah posisi kemunculan baru. List diisi dengan melakukan satu kali traversal terhadap pita.

Pencetakan dilakukan jika list tidak kosong, dengan melakukan traversal terhadap semua list dari 'a' s.d. 'z'. Sambil mencetak, jumlah kemunculan dihitung, dan pada akhir traversal dituliskan.

Walaupun penyisipan selalu terjadi pada akhir elemen list, alamat elemen terakhir setiap list tidak dicatat, karena dianggap bahwa memboroskan memori. dibandingkan dengan proses melakukan traversal menuju elemen terakhir list yang relatif elemennya sedikit Jadi untuk menyisipkan elemen di akhir list harus dilakukan traversal telebih dahulu. Jika alamat elemen terakhir dari setiap list dicatat, maka diperlukan lagi sebuah tabel untuk mencatatnya seperti halnya tabel untuk mencatat alamat elemen pertama.

Algoritma secara umum dari solusi ini adalah:

```
Buat list kemunculan kosong untuk seluruh huruf
Traversal pita :
       untuk setiap huruf C (yang diketahui posisinya I) ,
       insert posisi I sebagai elemen list kemunculan pada karakter C
Tulis output berdasarkan keadaan list :
       Traversal list huruf 'a'..'z'
              jika tidak kosong traversal untuk print posisi
```

Representasi berkait ini dapat diterapkan secara berkait dengan pointer ataupun dengan tabel. Contoh berikut diberikan untuk representasi dengan pointer dan representasi dengan tabel. Perbedaannya hanya pada cara penulisan elemen list dan primitif alokasi alamat.

```
Program Kemunculan2a
{ Representasi list huruf kontigu, list posisi BERKAIT dengan POINTER }
{ Skema proses sekuensial dengan mark, dengan penanganan kasus kosong }
```

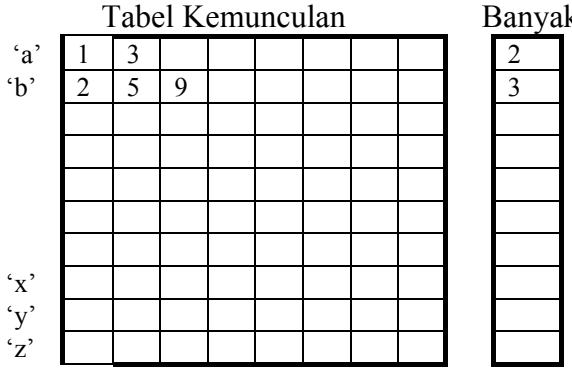
```
KAMUS
   type Address : pointer to ElmtList
   HeadList : array ['a'..'z'] of address { tabel alamat elemen pertama }
   type ElmtList : < Posisi : integer [1..100], Next : address >
   P : address
   Count : integer [1..100] { Kemunculan setiap huruf }
   JumlahKar : integer      { Jumlah karakter pada pita }
   C : character ['a'..'z'] { Untuk traversal karakter abjad }

   procedure InsertLast (input/output First, P : address)
   { Insert P sesudah elemen terakhir list First,
     P sudah dialokasi dan informasinya terdefinisi }
```

```
ALGORITMA
   START
   if CC = '.' then
      output ("Pita kosong, tidak ada isinya")
   else
      { Inisialisasi : traversal untuk mengosongkan list kemunculan}
      C traversal ['a'..'z']
         HeadList_C ← Nil
      JumlahKar ← 0
      { Proses sekuensial }
      repeat
         JumlahKar ← JumlahKar + 1
         depend on CC :
            CC = ' ' : -
            CC ≠ ' ' : Alokasi(P)
                       if (P ≠ Nil) then
                          Posisi(P) ← JumlahKar
                          Next(P) ← Nil
                          InsertLast(HeadList_CC,P)
         ADV
      until (CC = '.')
      { Penulisan hasil : traversal }
      C traversal ['a'..'z']
         P ← HeadList_C
         if (P ≠ Nil) then
            output (C, ":"); Count ← 0
            repeat
               Count ← Count + 1; output (Posisi(P))
               P ← Next(P)
            until (K = Nil)
         output ("Muncul ", Count, " kali")
```

```
procedure InsertLast (input/output First, P : address)
{ Insert P sesudah elemen terakhir list First, P sudah dialokasi dan
informasinya terdefinisi }

KAMUS LOKAL
   Last : address

ALGORITMA
   if (First = Nil) then    { Insert first }
      First ← P
   else   { traversal sampai Last dg mencatat address,
            kemudian insert sebagai suksesor Last }
      Last ← First
      while (Next(Last) ≠ Nil) do
         Last ← Next(Last)
      { Next(Last) = Nil }
      Next(Last) ← P
```

Latihan Soal

1. Untuk representasi KONTIGU, didefinisikan struktur sebagai berikut :
   List kemunculan dicatat dalam sebuah tabel, dengan model representasi tanpa mark (dicatat banyaknya kemunculannya. Lihat model representasi kata pada Bagian I.).

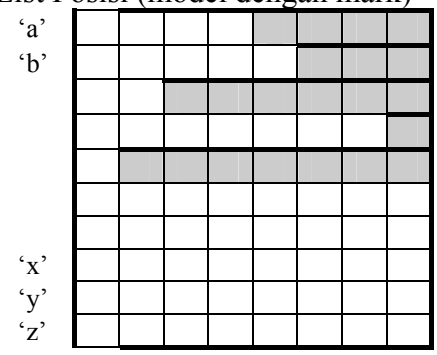


Kamus untuk representasi tersebut diberikan sebagai berikut, lengkapilah algoritmanya:

```
Program Kemunculan3
{ Representasi KONTIGU list huruf, representasi kontigu list posisi }

KAMUS
   constant Nmax : integer = 100 { maksimum isi pita yang diproses
                                   adalah 100 }
   TabMuncul : < array [1..NMax] of integer, MaxIsi : integer >
   { TabMuncul[1..MaxIsi] elemennya adalah posisi kemunculan }
   TabHuruf : array ['a'..'z'] of TabMuncul
   { TabHuruf_CC(i) = N, artinya Huruf CC muncul ke-i kalinya pada posisi N }

ALGORITMA
```

2. Untuk representasi KONTIGU, dapat juga didefinisikan struktur list posisi kemunculan yang direpresentasi dalam tabel, dengan model representasi memakai mark yang diberikan ilustrasinya sebagai berikut:

List Posisi (model dengan mark)

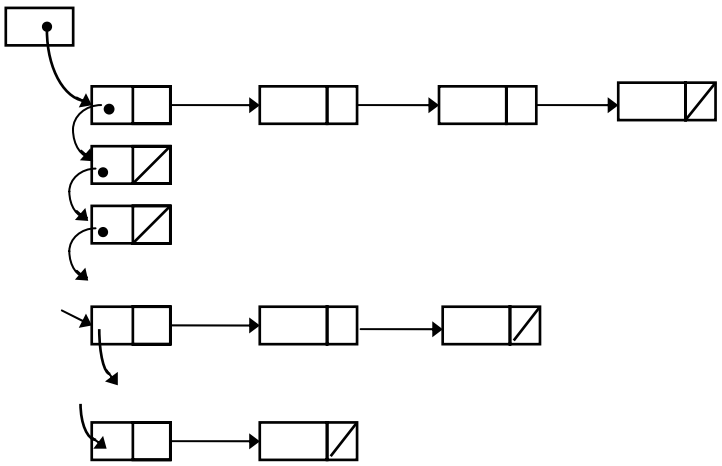


Kamus untuk representasi tersebut diberikan sebagai berikut, lengkapilah algoritmanya

```
Program Kemunculan4
{ Representasi KONTIGU list huruf, representasi kontigu list posisi}

KAMUS
    constant Nmax : integer = 100  { maksimum isi pita yang diproses
                                     adalah 100 }
    TabMuncul : array [1..NMax] of integer
    { TabMuncul[1..NMax] berisi nol jika tidak terpakai, jika tidak nol
      (misalnya sama dengan N), maka N adalah posisi kemunculan huruf }
    TabHuruf : array ['a'..'z'] of TabMuncul
    { TabHuruf_CC(i) = N, artinya Huruf CC muncul ke-i kalinya pada posisi N }

ALGORITMA
```

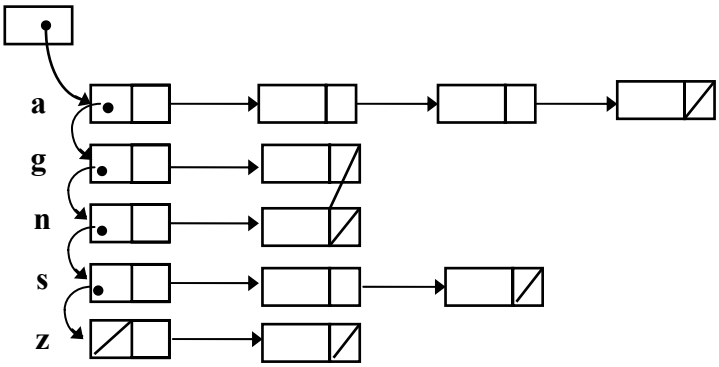
3. Untuk keempat struktur data (dua pada contoh solusi dan dua pada latihan soal), evaluasilah algoritma secara keseluruhan (dari segi pemakaian memori, kompleksitas algoritma). Struktur mana yang "terbaik"?

4. Kemungkinan lain representasi data adalah sebagai berikut : huruf dimunculkan sebagai list berkait dan disediakan kepala list untuk setiap huruf. List huruf tidak direpresentasi secara kontigu seperti pada contoh sebelumnya, melainkan secara berkait . List posisi juga direpresentasi berkait. Ilustrasi dari struktur ini adalah :



Tuliskanlah kamus dan algoritma untuk representasi ini.

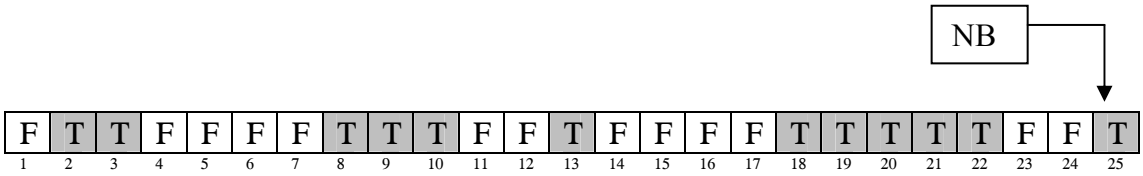
5. Dengan representasi list huruf maupun list posisi secara berkait, dapat pula dipilih representasi data dengan **hanya huruf yang muncul** akan menjadi angota list kuruh. Setiap huruf ini selanjutnya akan menjadi kepala list dari list posisi. Ilustrasi ini adalah sebagai berikut, andaikata huruf yang muncul hanyalah huruf ‘a’, ‘g’, ‘n’, ‘s’, ‘y’. List huruf yang muncul dapat disusun terurut atau tidak terurut. Tuliskanlah kamus dan algoritmanya.



Studi Kasus 3: Pengelolaan Memori

Deskripsi Persoalan

Suatu memori dipakai dalam satuan blok dan terdiri dari NB buah blok kontigu. Didefinisikan *zone* adalah sekumpulan blok kontigu. Ukuran dari sebuah *zone* bebas adalah banyaknya blok kontigu yang berstatus bebas. Sebuah *zone* bebas dicirikan oleh indeks blok bebas pertama dan ukurannya. Pada kasus ini, yang hendak dilacak hanyalah status setiap blok, kita tidak perlu mengelola ISI dari setiap blok. Contoh keadaan sistem, dengan jumlah maksimum blok NB = 24 digambarkan sebagai berikut:



Blok ditandai dengan F artinya KOSONG, dengan T artinya ISI. Beberapa contoh *zone* bebas:

***ZONE* BEBAS**	**AWAL**	**UKURAN**
Pertama	1	1
Kedua	4	4
Ketiga	11	2
Keempat	14	4
Kelima	23	2

Keadaan *zone* bebas akan berubah tergantung kepada peristiwa pemakaian memori (alokasi) atau pembebasannya (dealokasi). Alokasi dan dealokasi selalu dilakukan terhadap blok yang kontigu. Adanya banyak *zone* bebas yang kecil-kecil akan kurang menguntungkan dibandingkan dengan *zone* bebas yang berukuran besar karena alokasi memori menjadi sulit walaupun tempat yang tersedia masih ada.

Tujuan dari studi adalah merealisasi prosedur-prosedur berikut, yang diberikan spesifikasi umumnya. *Initial State*, *Final State*, dan gambaran umum proses dari setiap prosedur akan dibuat lebih rinci ketika struktur data ditentukan.

procedure InitMem
{ Mengeset semua blok menjadi blok KOSONG }

procedure AlokBlok (input X : integer, output IAw : integer)
{ Melakukan alokasi: membuat suatu *zone* kosong (jika ada) X blok di antaranya menjadi ISI }

procedure DeAlokBlok (input X, IAw : integer)
{ Membebaskan *zone* isi: membuat sebuah *zone* berukuran X yang berawal pada IAw yang tadinya ISI menjadi KOSONG }

procedure GarbageCollection
{ Memampatkan sehingga semua *zone* KOSONG ada di bagian kiri memori, dan *zone* ISI di bagian kanan memori }

Berikut ini akan diberikan rincian spesifikasi dan implementasi prosedur di atas dengan dua macam representasi, yaitu representasi kontigu dan representasi berkait. Untuk setiap representasi, akan diberikan KAMUS untuk mendeskripsikan status memori, dan uraian singkat prosedur serta deskripsi yang lebih persis dari porsedur yang direalisasi berdasarkan representasi tersebut. Akan terlihat, bahwa kompleksitas algoritma tergantung kepada pemilihan representasi. Silahkan mempelajari baik-baik, dan memilihnya untuk persoalan nyata yang tepat.

Representasi KONTIGU

Definisi Struktur Data

```
KAMUS
    constant NB : integer = 100
    STATMEM : array [1..NB] of boolean { tabel status memori:
                                         true jika ISI, false jika kosong }
```

Implikasi Representasi terhadap Realisasi Prosedur

procedure InitMemK
I.S. : Sembarang
F.S. : Semua blok memori dinyatakan KOSONG
Proses : Semua status blok dengan indeks [1..NB] dijadikan KOSONG dengan traversal blok [1..NB]. Proses sekuensial tanpa penanganan kasus kosong (NB ≥ 0).

Solusi umum algoritma adalah:

```
Set status blok [1..NB] menjadi false
```

procedure AlokBlokF (input X : integer, output IAw : integer)
I.S. : Sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu dijadikan isi. IAw adalah indeks pada tabel STATMEM, yang merupakan blok kosong pertama yang dialokasikan
F.S. :

- Jika ada *zone* yang memenuhi syarat (ada *zone* dengan jumlah blok kontigu sebanyak X atau lebih yang berstatus kosong), maka IAw akan berisi indeks blok bebas pertama pada *zone* yang dipilih untuk dialokasi; kemudian memutakhirkan status pemakaian memori *zone* tersebut. Dalam hal ada lebih dari satu *zone* yang memenuhi syarat, pemilihan dari *zone* yang memenuhi syarat ditentukan oleh strategi pengalokasian: **First Fit** atau **Best Fit**. Dengan strategi pengalokasian First Fit, alokasi dilakukan pada *zone* memenuhi syarat yang pertama kali diketemukan dalam proses pencarian. Dengan strategi Best Fit, alokasi dilakukan terhadap *zone* yang memenuhi syarat, namun berukuran sama dengan X, atau jika tidak ada *zone* berukuran X maka diambil yang ukurannya paling minimal. Stratagi Best Fit menguntungkan karena blok tidak

terpartisi menjadi blok-blok kecil. Untuk contoh keadaan blok seperti pada gambar di atas, jika X=2, dengan First Fit yang dialokasi adalah *zone* dengan indeks blok 4 dan 5 (akibatnya tersisa sebuah *zone* kosong berukuran 2), sedangkan dengan Best Fit, yang dialokasi adalah *zone* dengan blok 11 dan 12 (akibatnya, *zone* bebas dengan ukuran empat masih tetap berukuran 4)

- Jika tidak ada lagi *zone* yang memenuhi syarat (tidak ada *zone* dengan jumlah blok sebanyak X) maka IAw bernilai 0 dan status blok tetap seperti pada I.S. }

Proses : menentukan NKosong [0..NB]
Jika IAw[1..NB], status memori blok [IAw..IAw+X-1] menjadi ISI.
Jika IAw = 0, status memori tetap seperti I..S.

Proses alokasi dengan First-Fit:
Lakukan **proses pencarian *zone* pertama yang memenuhi syarat, yang merupakan proses pencarian** berikut sampai ada blok memenuhi syarat yang pertama:

- *Search* blok awal dari *zone* kosong terkiri: blok berstatus kosong paling kiri.
- Jika ada, lanjutkan *search* sembari mencacah banyaknya blok kosong kontigu pada *zone* yaitu NKosong ≥ X.

Karena pencarian dimulai dari indeks blok terkecil dan segera dihentikan ketika diketemukan sebuah *zone* kosong yang memenuhi syarat tanpa peduli berapapun ukurannya, maka *zone* memenuhi syarat yang diperoleh adalah *zone* kosong terkiri dengan NKosong ≥ X.

Solusi kasar untuk algoritma alokasi secara First-Fit adalah sebagai berikut:

```
procedure AlokBlokKF (input X : integer, output IAw : integer)
Inisialisasi
repeat
   Search blok kosong pertama :  skema search : while...
   if blok pertama zone kosong then
       Catat indeks sebagai NAwal, blok awal zone kosong }
       Hitung NKosong, banyaknya blok dlm zone kosong tersebut: while...
until semua blok diperiksa atau NKosong ≥ X
Terminasi
if ada zone memenuhi syarat then
   Ubah status blok pada zone tersebut [IAw..IAw+X-1] menjadi ISI
```

Proses alokasi dengan Best-Fit:
Pemeriksaan terhadap semua *zone* yang memenuhi syarat harus dilakukan untuk mencari yang "terbaik", yaitu yang ukurannya minimum (paling mendekati X).
Proses berikut dilakukan terhadap semua *zone* kosong untuk mencari yang ukurannya minimal dan memenuhi syarat:

- *Search* blok awal dari *zone* kosong terkiri: blok berstatus kosong paling kiri
- Jika ada, lanjutkan pemeriksaan terhadap *zone* sembari mencacah banyaknya blok kosong kontigu pada *zone* (NKosong ≥ X.). Jika *zone* yang diperiksa adalah *zone* pertama, maka NKosong yang memenuhi syarat dianggap sebagai NAwal. Jika merupakan *zone* kosong sesudah pertama, periksalah apakah NKosong pada *zone* yang juga memenuhi syarat ini mempunyai kriteria lebih baik (lebih mendekati X).

Proses penentuan *zone* kosong dengan jumlah blok kontigu minimum dapat dilakukan dengan menggunakan skema pencarian harga minimum. Ada dua versi penentuan harga minimum: perlakukan khusus terhadap elemen pertama untuk menentukan minimum, atau menginisialisasi harga minimum dengan suatu nilai khusus. Implementasi yang dipilih

pada algoritma yang akan ditulis merupakan kombinasi dari kedua skema tadi: elemen pertama tidak mendapat perlakukan khusus, melainkan diproses dalam badan pengulangan dengan melakukan analisa kasus, apakah menentukan minimum elemen pertama (inisialisasi), atau menggantikan elemen minimum. Cara ini dipilih untuk mempersingkat pengkodean. Skema solusi untuk algoritma alokasi secara Best-Fit adalah sebagai berikut:

```
procedure AlokBlokKB (input X : integer, output IAw : integer)
Inisialisasi
repeat
  Search zone kosong pertama : skema search : while...
  if blok pertama zone kosong then
      Traversal: Hitung banyaknya blok dlm zone kosong tsb.
  if ada zone kosong dan memenuhi syarat then
      if zone kosong pertama
       then Inisialisasi:
            Ukuran zone Minimum : NBMin dan Posisi Awal NAwal
       else Cek apakah lebih baik, jika ya, update NBMIn dan IAw}
until semua blok diperiksa atau blok berukuran=X
Terminasi
if ada zone memenuhi syarat then
  Ubah status blok pada zone tersebut [IAw..IAw+X-1] menjadi ISI
```

procedure DeAlokBlok (input X, IAw : integer)
I.S. : X adalah ukuran *zone*, bilangan positif dan IAw adalah alamat blok awal *zone* tersebut, dengan IAw ∈ [1..NB-X], Blok dengan indeks IAw s.d. IAw+X-1 pasti berstatus ISI.
F.S. : Tabel status memori dengan indeks blok IAw..IAw+X-1menjadi KOSONG }
Proses : Sebuah *zone* berukuran X dan bebawal pada blok IAw di-dealokasi (statusnya dijadikan kosong)
Proses dengan representasi ini sangat sederhana: adalah pemrosesan sekuensial (traversal) untuk membuat status blok [IAw..IAw+X-1] KOSONG

Skema solusi algoritma adalah sebagai berikut :

```
Set status blok [X..IAw+X-1] menjadi false
```

procedure GarbageCollectionK
I.S. : Sembarang
F.S. : Tabel status memori menjadi dua bagian: *zone* bebas di belahan "kiri" (indeks kecil), *zone* ISI di belahan "kanan". Jika K adalah integer [0..NB] yang merupakan indeks blok KOSONG terkahir pada *zone* bebas, maka ada 3 kemungkinan F.S.

- Jika K ≠ 0 dan K < NB, maka blok dengan indeks[1..K] adalah *zone* KOSONG, bagian dengan indeks [K+1 .. NB] adalah *zone* ISI.
- Jika K ≠ 0 dan K = NB: [1..K] maka semua blok adalah KOSONG, memori terdiri dari sebuah *zone* KOSONG.
- Jika K = 0 maka semua blok adalah ISI, tidak ada *zone* kosong.

Proses : Menggabungkan semua *zone* kosong, sehingga diperoleh satu *zone* kosong saja "di bagian kiri" tabel. Perhatikan bahwa proses penggeseran secara fisik tidak diperlukan jika kita tidak peduli dengan isi memori dan hanya mengelola status
Realisasi"menggeser" elemen tabel yang berstatus KOSONG ke kiri sangat sederhana dan dapat dilakukan dengan dua pass atau satu pass.
Dengan dua pass:

- Lakukan proses sekuensial (traversal) untuk mencacah banyaknya blok kosong, misalnya Nkosong
- Lakukan traversal sekali lagi, untuk memberi status [1..Nkosong] dengan KOSONG dan [Nkosong+1..NB] dengan ISI

Dengan satu pass: Harus dilakukan "penggeseran" secara fisik (ide : kasus bendera Indonesia):
- Lakukan proses sekuensial (traversal), kelompokkan KOSONG di kiri dan ISI di kanan dengan menukarkan dua elemen.

Skema solusi algoritma dengan dua pass adalah sebagai berikut :
NKosong = banyaknya blok kosong

```
Inisialisasi K dengan 0
Pass pertama : cacah banyaknya blok kosong
i traversal [1..NB]
   if statusnya KOSONG then K bertambah satu
if K=0 then
   { Tidak ada yang harus dikerjakan }
else   { Pass kedua : }
   Set status blok 1..K menjadi KOSONG
   Set status blok K+1..NB menjadi ISI
```

Realisasi Prosedur

```
procedure InitMemK
{ I.S. sembarang }
{ F.S. Semua blok memori dinyatakan KOSONG }
{ Proses : semua status blok dengan indeks [1..NB] dijadikan KOSONG  dengan
traversal blok [1..NB]. Proses sekuensial tanpa penanganan kasus kosong (NB ≥ 0)
}
```

```
KAMUS
   i : integer
```

```
ALGORITMA
   i traversal [1..NB]
      STATMEM_i ← false
```

```
procedure AlokBlokKF (input X : integer, output IAw : integer)
{ Strategi pengalokasian adalah First Fit }
{ I.S. sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu
dijadikan isi }
{ F.S. IAw akan berisi indeks blok kosong pertama pada tabel Status Memori jika
ada X blok kontigu berstatus KOSONG yang masih bisa dialokasi,  kemudian
memutakhirkan status pemakaian memori
IAw bernilai 0 jika tidak ada blok kontigu berukuran minimal X }
{ Proses : lihat penjelasan }
```

```
KAMUS LOKAL
    i : integer
    NKosong : integer { banyaknya blok kosong KOSONG }
```

```
ALGORITMA
    { Cari apakah ada X blok KOSONG }
    i ← 1
    repeat
        { Search blok KOSONG pertama, skema search tanpa boolean }
        { Selama isi, abaikan }
        while (i <  NB) and (STATMEM_i ) do
            i ← i + 1
        { i =  NB or not STATMEM_i }
        { Hitung jumlah blok kosong konsekutif }
        NKosong ← 0; IAw ← i
        while (i <  NB) and (not STATMEM_i) and (NKosong < X) do
            i ← i + 1; NKosong ← NKosong + 1
        { i =  NB or STATMEM_i or NKosong = X:
          jika masih kosong, Nkosong hrs ditambah }
        if not STATMEM_i then
            NKosong ← NKosong + 1
    until (i = NB) or (NKosong ≥ X)
    { Terminasi: Tentukan nilai IAw,
      Ubah status jika ada zone kosong memenuhi syarat }
    if (NKosong ≥ X) then
    { Update status memori }
        i traversal [IAw..IAw+X-1]
            STATMEM_i ← true
    else { Tidak perlu melakukan update, namun IAw harus diisi }
        IAw ← 0
```

```
procedure AlokBlokKB (input X : integer, output IAw : integer)
{ Strategi pengalokasian adalah Best Fit }
{ I.S. Sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu
dijadikan ISI }
{ F.S. IAw akan berisi indeks blok kosong pertama pada tabel Status Memori jika
ada X blok kontigu berstatus KOSONG yang masih bisa dialokasi,  kemudian
memutakhirkan status pemakaian memori
IAw bernilai 0 jika tidak ada blok kontigu berukuran minimal X }
{ Proses : lihat penjelasan }
```

```
KAMUS LOKAL
    i : integer
   NKosong : integer { # blok kosong KOSONG kontigu pada saat dihitung }
   IAwMin : integer  { Posisi awal blok kontigu terkecil yang
                       diketemukan }
    NbMin : integer   { Blok kontigu terkecil yang diketemukan,
                       dan masih memenuhi syarat ≥ X }
```

```
ALGORITMA
    { Cari apakah ada X blok KOSONG }
    i ← 1; IAwMin ← 0
    repeat
        { Search posisi KOSONG pertama, skema search tanpa boolean }
       { Selama isi, abaikan }
        while (i < NB) and (STATMEM_i) do
            i ← i + 1
        { i =  NB or not STATMEM_i }
        { Hitung jumlah blok konsekutif }
        NKosong ← 0;  IAwal ← i
        while (i <  NB) and (not STATMEM_i ) do
            i ← i + 1; NKosong ← NKosong + 1
        { i =  NB or STATMEM_i or NKosong = X :
          jika masih kosong, Nkosong hrs ditambah }
        if not STATMEM_i then
            NKosong ← NKosong + 1
        if (Nkosong  ≥ X) then { Blok memenuhi syarat }
            if ( IAwMin = 0) then { Inisialisasi zone memenuhi syarat }
                IAwMin ←  IAwal; NbMin ← NKosong
            else {mungkin zone kosong yg ditemukan < dr sebelumnya }
                if (Nkosong < NbMin) then
                    NbMin ← Nkosong; IAwMin ← IAwal
    until (i = NB) or (NKosong = X)
    { Jika NKosong = X, sudah minimum (the best)! }
    { Terminasi: Tentukan nilai IAw,
      ubah status jika ada zone kosong memenuhi syarat }
    if (IAwMin ≠ 0) then { update status memori }
        IAw ← IAwMin
        i traversal [ IAwMin..IAwMin+X-1]
            STATMEM_i ← true
    else { Tidak ada aksi, hanya beri nilai IAw }
        IAw ← 0
```

```
procedure DealokBlokK (input X, IAw : integer)
{ I.S. X adalah ukuran zone, bilangan positif dan IAw adalah alamat blok awal
zone tersebut, dengan IAw ∈ [1..NB-X], blok dengan indeks IAw s.d. IAw+X-1 pasti
berstatus ISI. }
{ F.S. Tabel status memori dengan indeks blok IAw..IAw+X-1 menjadi KOSONG }
{ Proses: Sebuah zone berukuran X dan bebawal pada blok IAw  di-dealokasi
(statusnya dijadikan KOSONG) }
```

```
KAMUS LOKAL
    i : integer
```

```
ALGORITMA
    i traversal [IAw..IAw+X-1]
        STATMEM_i ← false
```

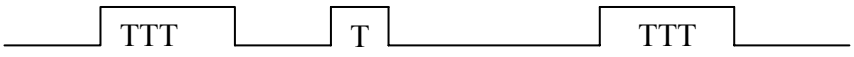
```
procedure GarbageCollectionK
{ I.S. Sembarang }
{ F.S. Terbentuk sebuah zone kosong di "kiri" }
```

```
KAMUS LOKAL
    i : integer
    NKosong : integer
```

```
ALGORITMA
    NKosong ← 0
    i traversal [1..NB]
        if (not STATMEM_i)then
            NKosong ← NKosong + 1
    { NKosong bernilai [0..NB] }
    { Update : mungkin tidak ada zone kosong, maka skema while... }
    i ← 1
    while (i ≤ NKosong) do
        STATMEM_i  ← false
        i ← i + 1
    { i = NKosong + 1 }
    while (i ≤ NB) do
        STATMEM_i  ← true
        i ← i + 1
    { i = NB + 1 : semua blok sudah diinisialisasi }
```

Latihan Soal

Zone kosong dan *zone* isi dapat digambarkan sebagai berikut:



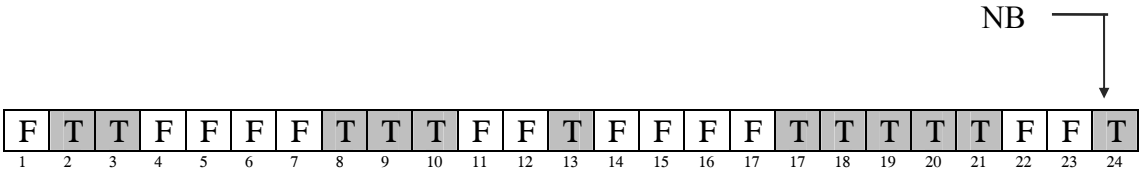
Dengan melakukan analogi terhadap mesin akusisi "kata" pada pita karakter, tuliskanlah sekali lagi semua algoritma yang diberikan. Sekarang Anda harus mendefinisikan START, ADV, CC dan EOP dalam terminologi indeks tabel. Analogi prosedur akuisisi adalah:

- indeks tabel adalah "kursor" yang memungkinkan sebuah blok yang sedang berada di "jendela" diketahui statusnya, analog dengan CC.
- "*zone* bebas" yang merupakan deretan kontigu dari blok kosong analog dengan kata.
- Ignore_BlokIsi adalah analogi dari Ignore_Blank.

Perhatikanlah bahwa dengan abstraksi ini, algoritma menjadi lebih jelas!

Representasi Berkait Blok Kosong

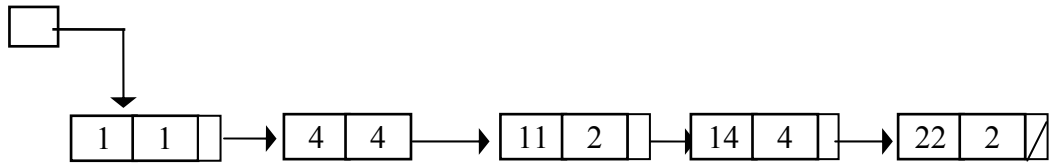
Untuk kasus berikut:



Maka secara berkait akan direpresentasi menjadi list linier *zone* kontigu secara berkait sebagai berikut :

<1,1> <4,4> <11,2> <14,4> <22,2>

Jadi, list akan direpresentasi dengan elemen yang terurut menurut posisi indeks awal blok dan diimplementasi dengan list linier biasa, yang diakhiri dengan suatu elemen ber-address Last, dengan properti Next (Last) = Nil:



Contoh keadaan list **mulai dari kosong** dan urutan proses jika alokasi first fit:

I.S.	F.S	Proses
<1,24>	<4,20>	AlokBlok (3,?) → AlokBlok(3,1)
<4,20>	<9,15>	AlokBlok (5,?) → AlokBlok(5,4)
<9,15>	<19,5>	AlokBlok (10,?) → AlokBlok(10,9)
<19,5>	<4,5>,<19,5>	DeAlokBlok (5,4)
<4,5> <19,5>	<1,8>,<19,5>	DeAlokBlok (1,3)
<1,8> <19,5>		dst

Maka dapat disimpulkan bahwa algoritma global prosedur yang harus direalisasikan adalah sebagai berikut:

Kamus untuk Representasi Berkait

```
KAMUS
   type  address { type terdefinisi }
   type ZB : < Aw : integer,     { indeks awal zone kontigu kosong }
               Nb : integer,     { banyaknya blok zone kontigu kosong }
               Next : address >
   FIRSTZB : address             { address zone kosong pertama }
   { Elemen list terurut menurut Aw }
   { Fungsi akses untuk penulisan algoritma secara lojik }
   { Jika P adalah sebuah address, maka dituliskan :
     - Next(P) adalah address elemen list sesudah elemen beralamat P
     - Aw(P) adalah nilai blok kosong pertama pada sebuah elemen list beralamat
       P yang mewakili sebuah zone kontigu
     - Nb(P) adalah ukuran blok sebuah zone kosong yang disimpan informasinya
       dalam elemen list beralamat P
     - Allocate(P) adalah prosedur utk melakukan alokasi sebuah ZB dengan alamat
       P, P tidak mungkinsama dengan Nil (alokasi selalu berhasil)
     - Dealollocate(P) adalah prosedur utk men-dealokasi sebuah alamat P }
```

Implikasi Struktur Data terhadap Prosedur

procedure InitMemB
I.S. : Sembarang.
F.S. : Semua blok memori dinyatakan KOSONG
Proses : Diinisialisasi satu *zone* kosong <1, NB> jika NB adalah jumlah maksimum blok.

Solusi umum algoritma adalah:

```
Create list kosong
Allocate sebuah elemen baru P dengan NAw=1 dan Ukuran NB
Insert elemen P dalam List
```

procedure AlokBlokBF (input X : integer, output IAw : integer)
Secara First Fit
I.S. : Sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu dijadikan ISI
F.S. : Tergantung kepada proses
Proses : *Sequential search* sebuah elemen list dengan properti jumlah blok kontigunya lebih besar atau sama dengan X. Pencarian segera dihentikan jika diketemukan elemen list yang memenuhi persyaratan tersebut.
Hasil pencarian menentukan F.S. :

- Jika ada, maka ada dua kemungkinan :
  - jika jumlah blok kontigu sama dengan X, hapus elemen list kosong tersebut,
  - jika jumlah blok kontigu lebih besar dari X, update elemen list kosong tersebut.
- Jika tidak ada elemen list yang memenuhi syarat : keadaan list tetap dan IAw diberi nilai 0.

Solusi umum algoritma adalah:

```
Sequential search List FirstZB, P sebuah address elemen
kondisi berhenti semua elemen list diperiksa atau Nb(P) ≥ X
if (Nb(P) ≥ X then { ada yg memenuhi syarat }
    if Nb(P) = X then { zone kosong menjadi isi }
        Delete elemen beralamat P; dealokasi P
    else { lebih besar : update zone kosong }
        Update Nb(P) dan Aw(P)
    Set nilai IAw dengan Aw(P)
else Set IAw = 0
```

Karena proses delete P membutuhkan alamat sebelum P, maka alamat sebelum P selalu dicatat.

procedure AlokBlokBB (input X : integer, output IAw : integer)
Secara Best Fit
I.S. : Sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu status memorinya dijadikan ISI.
F.S. : Tergantung kepada proses.
Proses : Periksa semua elemen list, elemen list dengan properti jumlah blok kontigunya lebih besar atau sama dengan X ditandai yang minimum.
Proses penentuan elemen dengan jumlah blok kontigu minimum dapat dilakukan dengan menggunakan skema pencarian harga minimum. Ada dua versi penentuan harga minimum: perlakukan khusus terhadap elemen pertama untuk menentukan minimum, atau

menginisialisasi harga minimum dengan suatu nilai khusus. Implementasi yang dipilih pada algoritma yang akan ditulis merupakan kombinasi dari kedua skema tadi: elemen pertama tidak mendapat perlakukan khusus, melainkan diproses dalam badan pengulangan dengan melakukan analisa kasus, apakah menentukan minimum elemen pertama (inisialisasi), atau menggantikan elemen minimum. Cara ini dipilih untuk mempersingkat pengkodean.

Setelah semua elemen diperiksa, ada dua kemungkinan :

- Jika alokasi dapat dilakukan, ada blok yang memenuhi syarat, masih ada dua kemungkinan:
  - jika jumlah blok kontigu sama dengan X, hapus elemen list kosong tersebut,
  - jika jumlah blok kontigu lebih besar dari X, update elemen list kosong tersebut.
- Jika alokasi tidak dapat dilakukan (tidak ada blok yang memenuhi syarat), maka list tetap keadaannya dan IAw diberi nilai 0

Solusi umum algoritma adalah:

```
Sequential search List FirstZB, P sebuah address elemen
kondisi berhenti Nb(P) =  X atau semua elemen list diperiksa
Skema search dengan boolean.
Untuk setiap elemen list beralamat P yang diperiksa :
    if elemen pertama then
        Inisialisasi NBMin dan Naw
    else { bukan elemen pertama }
        Cek apakah Nb(P) < NBMIN, jika ya update NBMin}
if (Nb(P) ≥ X) then { ada yg memenuhi syarat }
    if (Nb(P) = X) then { zone kosong menjadi isi }
        Delete elemen beralamat P; dealokasi P
    else { lebih besar: update zone kosong }
        Update Nb(P) dan Aw(P)
    Set nilai IAw dengan Aw(P)
else Set IAw = 0
```

Karena untuk operasi delete list dibutuhkan alamat elemen sebelumnya, alamat elemen sebelum P selalu dicatat.

procedure DeAlokBlokB (input IAw : integer, input X : integer)
I.S. : X adalah ukuran *zone*, bilangan positif dan IAw adalah alamat blok awal *zone* tersebut, dengan IAw ∈ [1..NB-X], Blok dengan indeks IAw s.d. IAw+X-1 pasti berstatus ISI.
F.S. : Tabel status memori dengan indeks blok IAw..IAw+X-1 menjadi KOSONG.
Proses : Sebuah *zone* berukuran X dan bebawal pada blbk IAw didealokasi (statusnya dijadikan KOSONG).
Tergantung kepada posisi *zone* yang hendak dialokasi terhadap elemen list *zone* kosong yang ada, proses dealokasi dapat menyangkut beberapa kasus, dan setiap kasus diberikan ilustrasinya sebagai berikut:

- Elemen pertama list, dalam hal ini masih dibedakan lagi menjadi dua macam kasus:
  - Hanya mengubah elemen pertama list (*update first*), karena ternyata blok yang didealokasi menjadi satu *zone* kontigu dengan *zone* kosong yang merupakan elemen petama list.

I.S.:

F.S.:

- Menyisipkan sebuah elemen list baru sebagai elemen pertama (*insert first*), karena blok yang didealokasi tidak kontigu dengan *zone* kosong pertama dan letaknya sebelum *zone* kosong pertama.

I.S.:

F.S.:

- Elemen terakhir list, dalam hal ini masih dibedakan lagi menjadi dua macam kasus:
  - Hanya mengubah elemen terakhir list (*update last*), karena ternyata blok yang di-dealokasi menjadi satu *zone* kontigu dengan *zone* kosong yang merupakan elemen terakhir list.

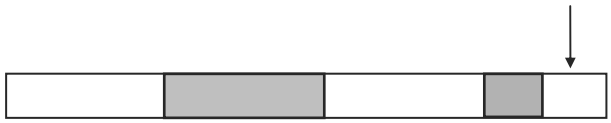
I.S.:

F.S.:

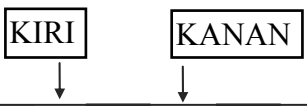
  - Menyisipkan sebuah elemen list baru sebagai elemen terakhir (*insert last*), karena blok yang didealokasi tidak kontigu dengan *zone* kosong pertama, dan letaknya sebelum *zone* kosong pertama

I.S.

F.S .

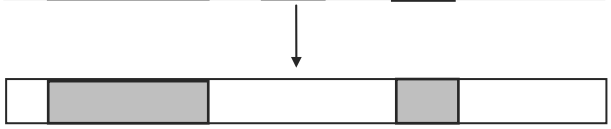


- Elemen list yang bukan merupakan elemen terakhir dan juga bukan elemen terakhir, dalam hal ini masih dibedakan lagi menjadi empat macam kasus:
  - Blok yang didealokasi dengan dua buah elemen list suksesif yang disebut KIRI dan KANAN menjadi sebuah *zone* kontigu karena blok yang didealokasi persis "di tengah" antara dua buah *zone* kontigu yang sudah ada. Peristiwa ini mengharuskan ***delete*** salah satu elemen yang sudah ada dan melakukan ***update*** terhadap elemen yang lain. Misalnya *delete* KANAN, *update* KIRI. Akan dipilih *delete* KANAN, *update* KIRI.

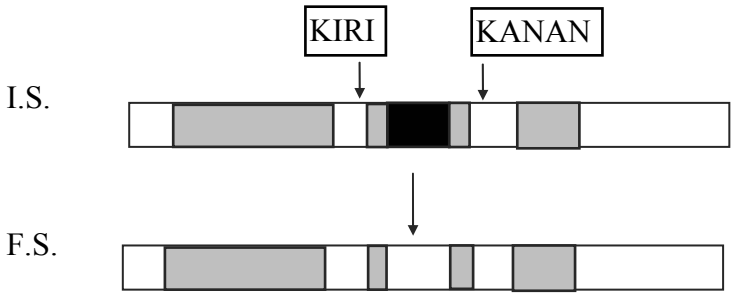


I.S.:

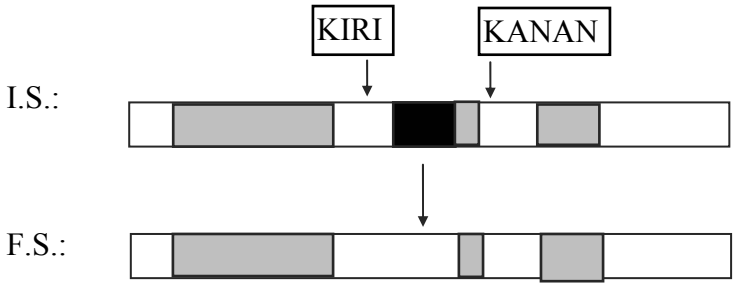
F.S.:



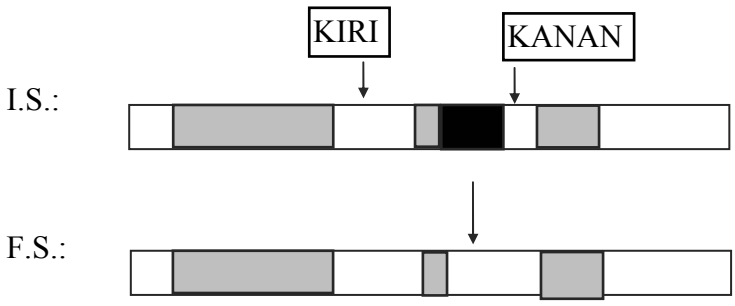
- Blok yang didealokasi terletak di antara dua buah *zone* kontigu yang sudah ada (misalnya KIRI dan KANAN), dan tidak kontigu dengan kedua *zone* tersebut. Peristiwa ini mengakibatkan *insert* sebuah elemen list baru yang mewakili *zone* didealokasi, di antara elemen list yang mewakili *zone* bebas KIRI dan KANAN (*insert after* KIRI, dan KIRI adalah predesesor kanan)



- Blok yang didealokasi ternyata membentuk sebuah *zone* kontigu dengan elemen list yang sudah ada, dan terletak sesudah sebuah elemen list KIRI. Peristiwa ini menyebabkan *update* KIRI.



- Blok yang didealokasi ternyata membentuk sebuah *zone* kontigu dengan elemen list yang sudah ada, dan terletak sebebelum sebuah elemen list KANAN. Peristiwa ini menyebabkan *update* KANAN.



Kesimpulannya adalah bahwa proses dealokasi dengan representasi ini dapat menjadi sebuah proses yang sangat rumit dan menimbulkan banyak kasus dibandingkan dengan proses dealokasi pada representasi kontigu!

procedure GarbageCollectionB
I.S. : Sembarang.
F.S. : Semua blok kosong ada di kiri dan blok isi di kanan:
- Jika ada *zone* kosong, hanya ada satu elemen lis, karena dijadikan satu *zone* kosong

- Jika tidak list *zone* kosong tidak ada elemennya (kosong), maka tidak dilakukan apa-apa

Proses : Jika list *zone* kosong tidak kosong, jadikan sebuah list dengan elemen tunggal beralamat P, Aw(P) bernilai 1 dan Nb(P) bernilai NB.

Skema solusi algoritma adalah :

```
Inisialisasi Nkosong dengan Nol
if List zone kosong tidak kosong then
   Akumulasi jumlah blok kosong,
   sambil delete list sehingga tersisa satu elemen
   Update elemen terakhir
else { tidak ada yang perlu dilakukan, karena semua zone ISI }
```

Realisasi Prosedur

```
procedure InitMemB                    { Representasi berkait zone kosong }
{ I.S. Sembarang }
{ F.S. Semua blok memori dinyatakan KOSONG }
{ Proses : create list kosong, buat elemen baru, insertFirst }
```

```
KAMUS LOKAL
   P : address
```

```
ALGORITMA
   FIRSTZB ← Nil  { Create list kosong }
   { Alokasi elemen baru: }
   Allocate(P); IAw(P) ← 1; Nb(P) ← NB; Next(P) ← Nil
   { Insert First: }
   FIRSTZB ← P
```

```
procedure AlokBlokBF (input X : integer, output IAw : integer)
{ I.S. Sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu
dijadikan ISI }
{ F.S. IAw adalah alamat awal sebuah zone bebas dengan X buah blok kontigu
kosong, zone bebas paling "kiri", jika ada. IAw=0, jika tidak ada zone kontigu
berukuran Iaw }
{ Strategi pengalokasian adalah First Fit }
```

```
KAMUS LOKAL
   P   : address
   Found : boolean
```

```
ALGORITMA
   { Search elemen list dengan properti Nb(P) ≥ X, }
   { Skema search dengan boolean }
   P ← FirstZB; PrecP ← Nil; Found ← false
   IAw ← 0
   while (P ≠ Nil) and not Found do
      if (Nb(P) ≥ X) then
         IAw ← Na(P); Found ← true
         if (Nb(P) = X) then
            { Delete }
            if (PrecP = Nil) then
               FirstZB ← Next(FirstZB) { Delete First }
            else
               Next(PrecP) ← Next (P)  { Delete after Prec P }
            Deallocate(P)
         else  { Nb(P) > X, update P, kurangi Nb(P) sebanyak X }
            Nb(P) ← Nb(P) - X ; Aw(P) ← Aw(P) + X  { update }
      else
         PrecP ← P); P ← Next (P)
```

```
procedure AlokBlokBB (input X : integer, output IAw : integer)
{ Representasi berkait zone KOSONG }
{ I.S. Sembarang. X adalah banyaknya blok yang diminta untuk dialokasi, yaitu
dijadikan ISI }
{ F.S. IAw adalah alamat awal sebuah zone bebas dengan X buah blok kontigu
kosong, zone bebas minimal ukurannya jika ada. IAw=0, jika tidak ada zone
kontigu berukuran IAw }
{ Strategi pengalokasian adalah Best Fit, skema search dengan boolean }
```

```
KAMUS LOKAL
    P, PrecP  : address { PrecP adalah address sebelum P }
    Found     : boolean
    PMin      : address { Posisi awal dari blok kontigu terkecil yang
                          diketemukan }
    PrecPMin  : address { Address sebelum PMin }
    FoundBest : boolean { Sudah ketemu yang terbaik : ukurannya = X }
```

```
ALGORITMA
    { Search elemen list dengan properti Nb(P) ≥ X,
      skema search dengan boolean }
    PrecPMin ← Nil; Pmin ← Nil
    P ← FirstZB; PrecP ← Nil; Found ← false; FoundBest ← false
    while (P ≠ Nil) and not FoundBest do
        if (Nb(P) ≥ X) then
            Found ← true
            if Nb(P)=X then { Sudah terbaik }
                  FoundBest ← true
            if Pmin = Nil  then   { Minimum diinisialisasi }
                  PMin ← P; PrecPMin ← Prec P
            else   if Nb(P) < Nb(Pmin) then {lebih baik }
                  PrecPMin ← PrecP;  Pmin ← P
        else
         PrecP ← P; P ← Next (P)
    { Semua elemen list selesai diperiksa }
    { Delete atau update elemen list jika ketemu }
    if Found then
        IAw  ←  Aw(Pmin)
        if (Nb(Pmin) = X ) then { delete }
            if PrecPMin = Nil   then { Delete First }
                FirstZB ← Next (FirstZB)
            else  { Delete After PrecPMin }
                Next(PrecPMin) ← Next (Pmin)
            Deallocate (Pmin)
        else {update }
            Nb(P) ← Nb(P) - X ; Aw(P) ← Aw(P) + X  { update }
    else
        IAw ← 0
```

```
procedure DeAlokBlokB (input X, IAw : integer )
{ Representasi berkait zone KOSONG }
{ I.S. X adalah ukuran zone, bilangan positif dan IAw adalah alamat blok awal
zone tersebut, dengan IAw ∈ [1..NB-X], Blok dengan indeks IAw s.d. IAw+X-1 pasti
berstatus ISI. }
{ F.S. Tabel status memori dengan indeks blok IAw..IAw+X-1menjadi KOSONG }
{ Proses : Sebuah zone berukuran X dan bebawal pada blok IAw didealokasi
(statusnya dijadikan kosong) }
```

```
KAMUS LOKAL
    P, PrecP : address
```

```
ALGORITMA
    if FirstZB = Nil then
        { insert 1 elemen ke list kosong }
        Alokasi (Pins) Nb(PIns) ← X; IAw(PIns) ← X; FirstZB ← PIns
    else
        P ← FirstZB; PrecP ← Nil; Found ← false
        while (P ≠ Nil) and not Found do
            if Aw(P) > IAw  then
                Found ← true
            else
                PrecP ← P; P ← Next(P)
        { P=Nil or Found }
        if not Found then
            { Pasti P= Nil, insertlast atau update PrecP }
            if Aw(PrecP)+ Nb(PrecP) = IAw then
                Nb(PrecP) ← Nb(PrecP) + X { update PrecP }
            else { bukan zone terkanan, isnert elmt}
                { insert last }
                Allocate(Pins); Aw(Pins) ← IAw; Nb(Pins) ← X; Next(PrecP) ← Pins
        else { Found, berarti Aw(P) > IAw }
            if PrecP = Nil then { elemen list = zone terkiri }
                { insert First, cek apakah perlu digabung dengan P }
                if IAw+X = Aw(P) then { kontigu dengan zone I, zone digabung,
                                        update P }
                    Aw(P) ← IAw; Nb(P) ←  Nb(P) + X     { update P saja }
                else{Insert First, bukan menjadi zone kontigu dg yg pertama }
                    Allocate (Pins) ); Aw(Pins) ← IAw; Nb(Pins) ← X
                    Next (Pins) ← FirstZB; FirstZB ← PIns
            else { PrecP ≠ Nil, ada 4 kemungkinan }
                {Jika PrecP disebut KIRI,Blok didealokasi TENGAH &P disebut KANAN }
                depend on PrecP, P, X, IAw
                    Aw(PrecP)+Nb(PrecP)=IAw and Aw(P) = IAw + X :
                        { a. 3 zone digabung: KIRI, TENGAH, KANAN
                             deleteP, update PrecP }
                        Next(PrecP) ← Next(P)
                        Nb(PrecP) ← Nb(PrecP)+Nb(P)+X
                        Deallocate (P)
                    Aw(PrecP)+ Nb(PrecP) ≠  IAw and Aw(P) ≠ IAw + X :
                        { b. Tidak ada penggabungan blok }
                        Allocate (Pins) ); Aw(Pins) ← IAw; Nb(Pins) ← X
                        Next(PrecP) ← Pins
                    Aw(PrecP)+Nb(PrecP)= IAw :
                        { c. 2 zone KIRI dan TENGAH digabung
                             gabung PrecP dg blok baru yg didealokasi,
                             update PrecP }
                        Nb(PrecP) ← Nb(PrecP) + X
                    Aw(P) = IAw+X :
                        { d. 2 zone TENGAH dan KANAN digabung
                             Gabung P dengan blok baru yg didealokasi }
                        Nb(P) ← Nb(P) + X; Aw(P) ← IAw
```

```
procedure GarbageCollectionB
{ I.S. Sembarang }
{ F.S. }
```

```
KAMUS LOKAL
    P : address
    NKosong : integer
```

```
ALGORITMA
    NKosong ← 0
    { Akumulasi jumlah blok kosong
      sambil delete list sehingga tersisa satu elemen }
    if FirstZB ≠ Nil then {minimal ada satu elemen }
        P ← FirstZB; PrecP  ←  Nil
        while (Next(P) ≠ Nil) do
            NKosong ← NKosong + Nb(P)
            PrecP ← P; P ← Next(P); Deallocate(PrecP)
        { Next(P) = Nil, elemen terakhir belum diproses
          NKosong bernilai [0..NB] }
        { Update elemen terakhir }
        Nb(P) ← Nkosong + Nb(P)
        Aw(P) ←  1
        FirstZB ← P
    else { tidak ada yang perlu dilakukan, karena semua zone ISI }
```

Studi Kasus 4: Multi-List

Deskripsi Persoalan

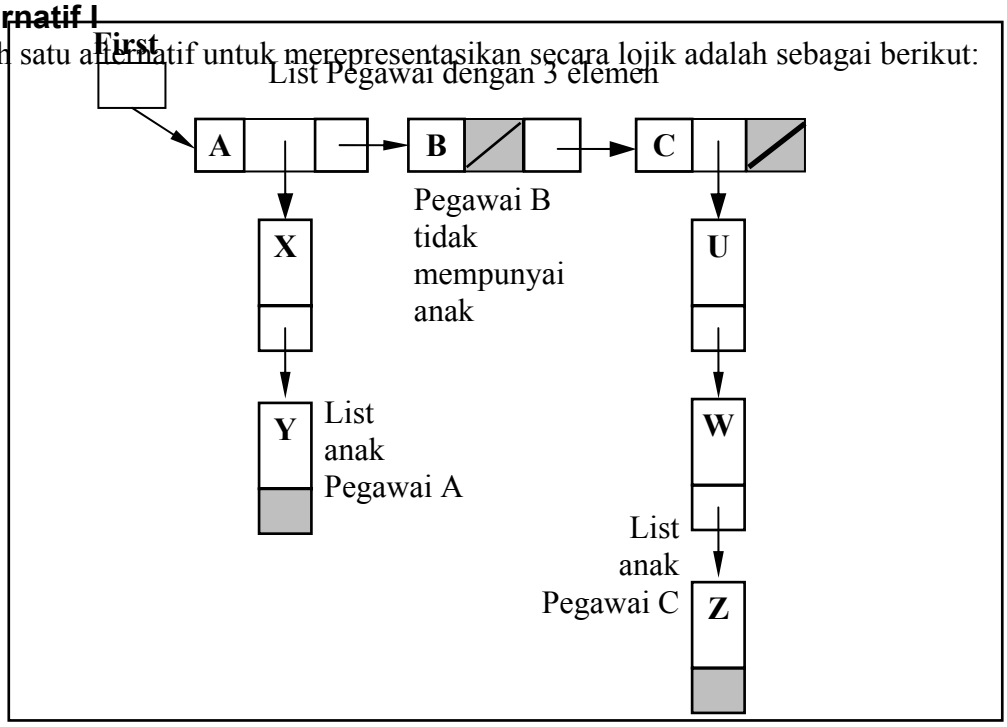
Kita harus mengelola sekumpulan pegawai, dan untuk setiap pegawai selain informasi mengenai dirinya kita juga harus menyimpan informasi tentang anak-anaknya (jika ada). Jika informasi tersebut harus direpresentasikan dalam struktur data internal, maka kita mempunyai list dari pegawai, dan juga list dari anak-anak pegawai.

Alternatif Struktur Data

Ada dua alternatif implementasi list Pegawai dan list anak pegawai. Masing-masing alternatif akan dibahas sebagai berikut

Alternatif I

Salah satu alternatif untuk merepresentasikan secara lojik adalah sebagai berikut:

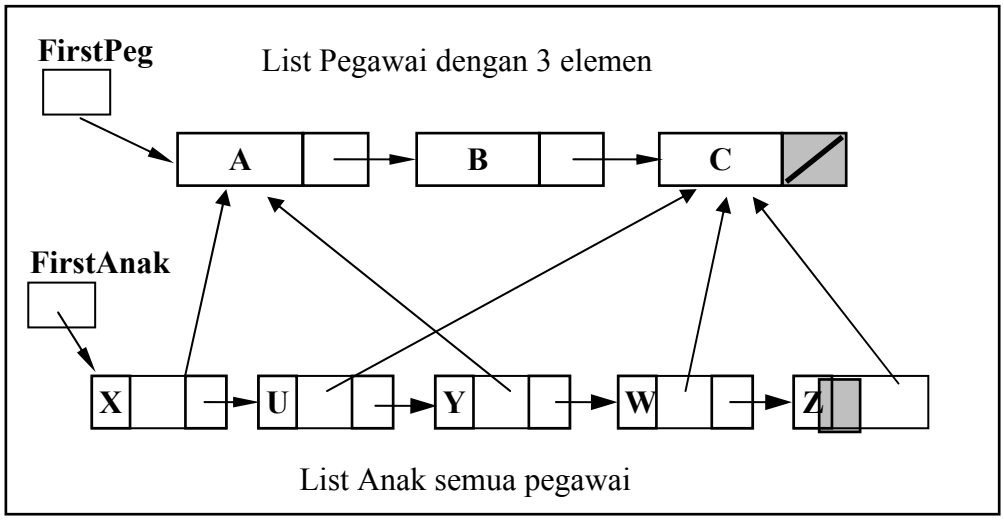


Pada gambar di atas, kita mempunyai dua list: elemen list Pegawai tidak sama informasinya dengan elemen list anak. Informasi pegawai yang disimpan misalnya : nomor pegawai, nama, jabatan, gaji pokok. Sedangkan informasi yang disimpan pada elemen list anak misalnya nama, tanggal lahir.

```
KAMUS
    { Definisi lojik List Pegawai untuk Alternatif pertama }
    type AdrPeg  : { type terdefinisi, alamat sebuah elemen list pegawai }
    type AdrAnak : { type terdefinisi, alamat sebuah elemen list anak }
    type Pegawai : < NIP : integer, Nama : string, Jabatan : string,
                     GajiPokok : real,
                     FirstAnak : adrAnak, NextPeg : AdrPeg >
    type Anak    : < Nama : string, TglLahir : integer,
                     NextAnak : AdrAnak >
    type ListPeg : AdrPeg
    FirstPeg     : ListPeg
```

Alternatif Kedua

Alternatif lain untuk merepresentasikan informasi di atas adalah sebagai berikut :



Pada alternatif kedua ini kita mempunyai dua buah list yang terpisah, dengan mengadakan "*indirect addressing*" untuk menemukan ayah dari seorang anak.

```
KAMUS
    { Definisi lojik List Pegawai untuk Alternatif kedua }
    type AdrPeg   : { type terdefinisi, alamat sebuah elemen list pegawai }
    type AdrAnak  : { type terdefinisi, alamat sebuah elemen list anak }

    type Pegawai  : < NIP : integer, Nama : string, Jabatan : string,
                      GajiPokok : real,
                      NextPeg : AdrPeg >
    type Anak     : < Nama : string, TglLahir : integer,
                      NextAnak : AdrAnak, Father : AdrPeg >
    type ListPeg  : AdrPeg
    type ListAnak : AdrAnak
    FirstPeg      : ListPeg
    FirstAnak     : ListAnak
```

Dapatkah dipilih, mana struktur data yang lebih baik untuk merepresentasikan informasi pegawai dan anak-anaknya? Jawabannya adalah TIDAK. Pertanyaan semacam ini (penentuan struktur data "terbaik") tidak ada artinya kalau fungsi-fungsi dan operasi yang ingin diimplementasi terhadap struktur data belum ditentukan.

Misalnya untuk representasi informasi di atas dikembangkan beberapa primitif (prosedur dasar yang sering dipakai pada saat program operasional) untuk membuat:

1. Daftar pegawai, dan untuk setiap pegawai harus dibuat juga nama anak-anaknya (jika ada).
2. Daftar anak-anak yang umurnya kurang dari 18 tahun (untuk keperluan tunjangan).
3. Daftar pegawai yang anaknya lebih dari 3 (keperluan KB).
4. Diketahui nama seorang anak, harus dituliskan nama bapaknya.
5. Mendaftarkan seorang anak yang baru lahir ke dalam list anak, jika diberikan tanggal lahir dan nama bapaknya.

Berikut ini akan diberikan algoritma untuk masing-masing struktur data, dengan representasi lojik list:

```
Program PEGAWAI1
{ Representasi BERKAIT,  Alternatif Pertama }

KAMUS
   type Pegawai : < NIP : integer, Nama : string, Jabatan : string,
                    GajiPokok : real,
                    FirstAnak : adrAnak, NextPeg : AdrPeg >
   type Anak : < Nama : string, TglLahir : integer,
                 NextAnak : AdrAnak >
   AdrPeg :  address { terdefinisi, alamat sebuah elemen list pegawai }
   AdrAnak : address { terdefinisi, alamat sebuah elemen list anak }
   FirstPeg : ListPeg

   procedure ListPegLengkap (input FirstPeg : ListPeg)
   { I.S.: FirstPeg terdefinisi, mungkin kosong }
   { F.S.: List FirstPeg ditulis informasinya
          beserta informasi semua anaknya jika tidak kosong }
   { Menuliskan daftar pegawai, untuk setiap pegawai dilist anaknya jika ada }
   { Menuliskan "List kosong, tidak ada pegawai" jika kosong }

   procedure ListTunjAnak (input FirstPeg : ListPeg)
   {I.S.: FirstPeg terdefinisi, mungkin kosong}
   {F.S.: Untuk setiap pegawai, anaknya yang berumur < 18 tahun ditulis
          Jika pegawai tidak mempunyai anak,
          menuliskan "Pegawai tidak mempunyai anak"}

   procedure ListPegNonKB (input FirstPeg : ListPeg)
   { Membuat daftar pegawai yang mempunyai lebih dari 3 anak }
   { I.S. List FirstPeg terdefinisi, mungkin kosong }
   { F.S. Semua pegawai yang anaknya lebih dari 3 orang ditulis informasinya }

   procedure OrTuAnak (input FirstPeg : ListPeg, input NamaAnak : string)
   { I.S. List Pegawai terdefinisi }
   { F.S. Jika ada anak yang bernama sesuai dengan NamaAnak, nama Pegawai
          ditulis. Jika tidak ada NamaAnak, tidak menuliskan apa-apa. }
   { Cari anak dg nama NamaAnak,
     tuliskan nama orangtua dari anak yg namanya = NamaAnak }

   procedure AddAnak (input/output FirstPeg : ListPeg, input NIPPeg : string,
                      input NamaAnak: string, input TglLahirAnak : integer)
   { Mendaftar seorang anak yang baru lahir, insert selalu pada awal list }
   { I.S. List Pegawai terdefinisi }
   { F.S. Jika pegawai dengan NIP=NIPeg ada, alokasi anak,
          insert seorang anak sebagai elemen pertama list anak }

ALGORITMA
{ Tidak dituliskan di sini, algoritma yang didefinisikan di atas hanya primitif-
primitif yg diperlukan dan dapat dipanggil dari modul lain }
```

```
procedure ListPegLengkap (input FirstPeg : ListPeg)
{ Alternatif Pertama }
{ I.S.: FirstPeg terdefinisi, mungkin kosong }
{ F.S.: List FirstPeg ditulis informasinya beserta informasi semua anaknya jika
tidak kosong.}
{ Menuliskan daftar pegawai, untuk setiap pegawai dilist anaknya jika ada }
{ Menuliskan "List kosong, tidak ada pegawai" jika kosong }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg   { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak { address untuk traversal, @ sebuah elemen list anak }
```

```
Algoritma :
   { Traversal pegawai : skema pemrosesan sekuensial dg penanganan kasus kosong
     Untuk setiap pegawai, traversal list anak untuk dituliskan namanya }
   PtrPeg ← FirstPeg  { First Pegawai }
   if (PtrPeg = Nil) then
      output ("List kosong, tidak ada pegawai")
   else { Minimal 1 Pegawai }
      repeat
         output (Nama(PtrPeg))
         { Traversal Anak }
         PtrAnak ← FirstAnak(PtrPeg)                { First Anak }
         if (PtrAnak = Nil) then
            output ("Pegawai ybs. tidak mempunyai anak")
         else
            repeat
               output (Nama(PtrAnak))               { Proses anak }
               PtrAnak ← NextAnak(PtrAnak)          { Next Anak }
            until (PtrAnak = Nil)
         PtrPeg ← NextPeg(PtrPeg)                   { Next Pegawai }
      until (PtrPeg = Nil)
```

```
procedure ListTunjAnak (input FirstPeg : ListPeg)
{ Alternatif Pertama }
{ I.S.: FirstPeg terdefinisi, mungkin kosong }
{ F.S.: Untuk setiap pegawai, anaknya yang berumur < 18 tahun ditulis
        Jika pegawai tidak mempunyai anak, menuliskan "Pegawai tidak mempunyai
        anak" }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg    { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak  { address untuk traversal, @ sebuah elemen list anak }
   UmurAnak : integer { umur anak pegawai }
   function Umur (TglLahir : integer) → integer
   { Fungsi yang mengirim umur jika yang dihitung dari tanggal hari ini dari
     sistem dibandingkan dengan TglLahir yang diberikan, tidak dituliskan }
```

```
ALGORITMA
   { Traversal list Pegawai, skema sekuensial dengan penanganan kasus kosong.
     Untuk setiap pegawai, traversal anaknya }
   PtrPeg ← FirstPeg            { First Pegawai }
   if (PtrPeg = Nil) then
      output ("List kosong, tidak ada pegawai")
   else { Minimal ada satu pegawai }
      repeat
         output (Nama(PtrPeg))   { Inisialisasi trav. anak }
         { Traversal Anak }
         PtrAnak ← FirstAnak(PtrPeg)   { First anak }
         if (PtrAnak = Nil) then
            output ("Pegawai ybs tidak mempunyai anak")
         else { Minimal ada 1 anak }
            repeat
               UmurAnak ← Umur(TglLahir(PtrAnak))  { Proses }
               depend on UmurAnak
                  UmurAnak < 18 : output (Nama(PtrAnak),UmurAnak)
                  UmurAnak ≥ 18 : -
               PtrAnak ← NextAnak(PtrAnak)   { Next Anak }
            until (PtrAnak=Nil)
         PtrPeg ← NextPeg (PtrPeg)           { Next Pegawai }
      until (PtrPeg=Nil)
```

```
procedure ListPegNonKB (input FirstPeg : ListPeg)
{ Alternatif Pertama }
{ Membuat daftar pegawai yang mempunyai lebih dari 3 anak }
{ I.S. List FirstPeg terdefinisi, mungkin kosong }
{ F.S. Semua pegawai yang anaknya lebih dari 3 orang ditulis informasinya }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg   { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak { address untuk traversal, @ sebuah elemen list anak }
   JumlahAnak : integer { banyaknya anak pegawai }
```

```
ALGORITMA
   { Traversal pegawai }
   PtrPeg ← FirstPeg                    { First-Pegawai }
   if (PtrPeg = Nil) then
      output ("List kosong, tidak ada pegawai")
   else  { minimal ada satu pegawai }
      repeat
         output (Nama(PtrPeg))
         { Traversal Anak }
         JumlahAnak ← 0                 { Inisialisasi }
         PtrAnak ← FirstAnak(PtrPeg)    { First Anak}
         while (PtrAnak ≠ Nil) do
            JumlahAnak ← JumlahAnak + 1      { Proses }
            PtrAnak ← NextAnak(PtrAnak)      { Next Anak }
         if (JumlahAnak > 3 ) then
            output ("Pegawai ybs mempunyai anak lebih dari 3")
         PtrPeg ← NextPeg(PtrPeg)            { Next Pegawai }
      until (PtrPeg=Nil)
```

```
procedure OrTuAnak (input FirstPeg : ListPeg, input NamaAnak : string)
{ Alternatif I}
{ I.S. List Pegawai terdefinisi }
{ F.S. Jika ada anak yang bernama sesuai dengan NamaAnak, nama Pegawai ditulis.
       Jika tidak ada NamaAnak, tidak menuliskan apa-apa }
{ Cari anak dengan nama NamaAnak,
  tuliskan nama orangtua dari anak yg namanya = NamaAnak }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg   { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak { address untuk traversal, @ sebuah elemen list anak }
   Found : boolean   { hasil pencarian orangtua anak }
```

```
ALGORITMA
   { Search }
   Found ← false
   PtrPeg ← FirstPeg
   while (PtrPeg ≠ Nil) and (not Found) do
      { Search anak dengan NamaAnak yang diberikan pada list anak }
      PtrAnak ← FirstAnak(PtrPeg)
      while (PtrAnak ≠ Nil) and (not Found) do
         if (Nama(PtrAnak) = NamaAnak) then
            Found ← true
         else
            PtrAnak ← NextAnak(PtrAnak)
      { PtrAnak = Nil or Found }
      if (not Found) then { eksplore pegawai yg berikutnya }
         PtrPeg ← NextPeg(PtrPeg)
   { PtrPeg = Nil or Found }
   if (Found) then
      output (Nama(PtrPeg))
```

```
procedure AddAnak (input/output FirstPeg : ListPeg, input NIPPeg : string,
                   input NamaAnak : string, input TglLahirAnak : integer)
{ Alternatif Pertama }
{ Mendaftar seorang anak yang baru lahir, insert selalu pada awal list }
{ I.S. List Pegawai terdefinisi }
{ F.S. Jika pegawai dengan NIP=NIPPeg ada, alokasi anak, jika berhasil insert
seorang anak sebagai elemen pertama list anak. Jika alokasi gagal atau NIPPeg
tidak ada, hanya menulis pesan }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg    { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak  { address untuk traversal, @ sebuah elemen list anak }
   FoundNIP : boolean { hasil pencarian NIP pegawai sebelum insert anak }
```

```
ALGORITMA
   { Search Pegawai }
   FoundNIP ← false
   PtrPeg ← FirstPeg
   while (PtrPeg ≠ Nil) and (not FoundNIP) do
      { search Pegawai dengan NIP yang diberikan }
      if (NIP(PtrPeg) = NIPPeg) then
         FoundNIP ← true
      else
         PtrPeg ←NextPeg(PtrPeg)
   { Akhir search pegawai : PtrPeg=Nil or FoundNIP }
   if (FoundNIP) then { add anak }
      Alokasi(PtrAnak)
      if (PtrAnak ≠ Nil) then
          TglLahir(PtrAnak) ← TglLahirAnak
          Nama(PtrAnak) ← NamaAnak
          NextAnak(PtrAnak) ← Nil
          if (FirstAnak(PtrPeg) = Nil) then   { Insert anak pertama }
             FirstAnak(PtrPeg) ← PtrAnak
          else  { Insert anak sebagai FirstAnak yang baru }
             NextAnak(PtrAnak) ← FirstAnak(PtrPeg)
             FirstAnak(PtrPeg) ← PtrAnak
      else { Alokasi gagal, tidak insert, hanya pesan }
          output ('Alokasi gagal')
   else  { NIPPeg tidak ada, error }
      output ("Pegawai tidak ada dalam list")
```

```
Program PEGAWAI2
{ Representasi BERKAIT, Alternatif Kedua }

KAMUS
    type Pegawai : < NIP : integer, Nama : string, Jabatan : string,
                     GajiPokok : real,
                     NextPeg : AdrPeg >
    type Anak : < Nama : string, TglLahir : integer,
                  NextAnak : AdrAnak, Father : AdrPeg >
    AdrPeg : address  { alamat sebuah elemen list pegawai }
    AdrAnak : address { alamat sebuah elemen list anak }

    FirstPeg : ListPeg
    FirstAnak : ListAnak

    procedure ListPegLengkap (input FirstPeg : ListPeg,
                              input FirstAnak : ListAnak)
    { Menuliskan daftar pegawai, untuk setiap pegawai dilist anaknya jika ada
      dan informasi semua anaknya jika tidak kosong.
      Menuliskan "List kosong, tidak ada pegawai" jika kosong }
    { I.S. FirstPeg terdefinisi, mungkin kosong }
    { F.S. List FirstPeg ditulis informasinya }

    procedure ListTunjAnak (input FirstAnak : ListAnak)
    { Traversal list anak,
      Menuliskan anak yg masih mendapat tunjangan : berumur < 18 tahun }
    { I.S. FirstPeg dan FirstAnak terdrfinisi, mungkin kosong }
    { F.S. Untuk setiap anak yang umurnya <18 tahun, tuliskan informasinya
           Jika list Anak kosong, tuliskan "List anak kosong" }

    procedure ListPegNonKB (input FirstPeg : ListPeg)
    { Membuat daftar pegawai yang mempunyai lebih dari 3 anak }
    { I.S. List First Peg terdefinisi, mungkin kosong }
    { F.S. Semua pegawai yang anaknya > 3 orang ditulis informasinya }

    procedure OrTuAnak (input FirstPeg : ListPeg,
                        input FirstAnak : ListAnak, input NamaAnak : string)
    { I.S. List Pegawai terdefinisi }
    { F.S. Jika ada anak yang bernama sesuai dengan NamaAnak,
           nama Pegawai ditulis.
           Jika tidak ada NamaAnak, tidak menuliskan apa-apa. }
    { Cari anak dg nama NamaAnak,
      tuliskan nama orangtua dari anak yg namanya = NamaAnak }

    procedure AddAnak (input/output FirstPeg : ListPeg, input NIPPeg : integer,
                       input/output FirstAnak : ListAnak,
                       input NamaAnak : string, input TglLahirAnak : integer)
    { Mendaftar seorang anak yang baru lahir,
      Cari Pegawai, insert anak pada list anak selalu pada awal list }
    { I.S. List Pegawai terdefinisi }
    { F.S. List FirstPeg terdefinisi.
           Jika pegawai dengan NIP=NIPeg ada, alokasi anak,
           insert seorang anak sebagai elemen pertama list anak, tentukan Bapak }

ALGORITMA
{ Tidak dituliskan di sini, algoritma yg didefinisikan di atas hanya primitif-
primitif yg diperlukan }
```

```
procedure ListPegLengkap (input FirstPeg : ListPeg, input FirstAnak : ListAnak )
{ Alternatif Kedua }
{ Menuliskan daftar pegawai, untuk setiap pegawai dilist anaknya jika ada dan
  informasi semua anaknya jika tidak kosong.
  Menuliskan "List kosong, tidak ada pegawai" jika kosong }
{ I.S. FirstPeg terdefinisi, mungkin kosong }
{ F.S. List FirstPeg ditulis informasinya
```

```
KAMUS LOKAL
    PtrPeg : AdrPeg   { address untuk traversal, @ sebuah elemen list pegawai }
    PtrAnak : AdrAnak { address untuk traversal, @ sebuah elemen list anak }
```

```
ALGORITMA
    { Traversal pegawai }
    PtrPeg ← FirstPeg    { First Pegawai }
    if (PtrPeg = Nil) then
        output ("List kosong, tidak ada pegawai")
    else
        repeat
            output (Nama(PtrPeg))
            { Traversal Anak }
            PtrAnak ← FirstAnak    { First Anak }
            while (PtrAnak ≠ Nil) do
                if (Father(PtrAnak) = PtrPeg) then            { Proses }
                    output (Nama(PtrAnak))
                PtrAnak ← NextAnak(PtrAnak)                   { Next Anak }
            { PtrAnak = Nil }
            PtrPeg ← NextPeg(PtrPeg)                          { Next Pegawai }
        until (PtrPeg = Nil)
```

```
procedure ListTunjAnak (input FirstAnak : ListAnak)
{ Alternatif Kedua }
{ Traversal list anak,
  Menuliskan anak yg masih mendapat tunjangan : berumur < 18 tahun }
{ I.S. FirstPeg dan FirstAnak terdefinisi, mungkin kosong }
{ F.S. Untuk setiap anak yang umurnya <18 tahun, tuliskan informasinya.
       Jika list Anak kosong, tuliskan "List anak kosong". }
```

```
KAMUS LOKAL
    PtrAnak : AdrAnak  { address untuk traversal, @ sebuah elemen list anak }
    UmurAnak : integer { umur anak pegawai }
    function Umur (TglLahir : integer) → integer
    { Fungsi yang mengirim umur dengan rumus: tanggal hari ini dari sistem
      dikurangi  TglLahir }
    { Tidak dituliskan di sini, buatlah sebagai latihan }
```

```
ALGORITMA
    { Traversal list Anak,
      skema proses sekuensial dg penanganan kasus kosong }
    { Untuk setiap anak periksa umurnya }
    PtrAnak ← FirstAnak         { First Anak }
    if (PtrAnak = Nil) then
        output ("List Anak kosong, tidak ada anak")
    else
        repeat
            UmurAnak ← Umur(TglLahir(PtrAnak))         { Proses }
            depend on UmurAnak
                UmurAnak < 18 : output (Nama(Father(PtrAnak)))
                                output (Nama(PtrAnak), UmurAnak)
                UmurAnak ≥ 18 : -
            PtrAnak ← NextAnak(PtrAnak)                { Next Anak }
        until (PtrAnak = Nil)
```

```
procedure ListPegNonKB (input FirstPeg : ListPeg, input FirstAnak : ListAnak)
{ Alternatif Kedua }
{ Membuat daftar pegawai yang mempunyai lebih dari 3 anak }
{ I.S. List First Peg terdefinisi, mungkin kosong }
{ F.S. Semua pegawai yang anaknya > 3 orang ditulis informasinya }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg   { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak { address untuk traversal, @ sebuah elemen list anak }
   JumlahAnak : integer { banyaknya anak pegawai }
```

```
ALGORITMA
   { Traversal list Pegawai,
     Skema proses sekuensial dg penanganan kasus kosong }
   { Untuk setiap pegawai, traversal list anaknya untuk mencacah banyaknya
     elemen list anak. Jika jumlah elemen list anak lebih dari tiga maka nama
     pegawai ditulis}
   { Traversal pegawai }
   PtrPeg ← FirstPeg                                  { First Pegawai }
   if (PtrPeg = Nil) then
      output ("List pegawai kosong")
   else
      repeat                                          { Proses }
         JumlahAnak ← 0
         { Traversal Anak }
         PtrAnak ← FirstAnak                          { First Anak }
         while (PtrAnak ≠ Nil) do
            if (Father(PtrAnak) = PtrPeg) then        { Proses Anak }
                  JumlahAnak ← JumlahAnak + 1
            PtrAnak  ← NextAnak(PtrAnak)              { Next Anak }
         { PtrAnak = Nil) }
         if (JumlahAnak > 3) then
            output (Nama(PtrPeg), " mempunyai anak lebih dari 3")
         PtrPeg ← NextPeg(PtrPeg)                     { Next Pegawai }
      until (PtrPeg = Nil)
      { semua elemen list pegawai selesai diproses }
```

```
procedure OrTuAnak (input FirstAnak : ListAnak, input NamaAnak : string)
{ Alternatif Kedua }
{ I.S. List Pegawai terdefinisi }
{ F.S. Jika ada anak yang bernama sesuai dengan NamaAnak, nama Pegawai ditulis.
       Jika tidak ada NamaAnak, tidak menuliskan apa-apa.}
{ Cari anak dg nama NamaAnak, tuliskan nama orangtua dari anak yg namanya =
  NamaAnak }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg   { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak { address untuk traversal, @ sebuah elemen list anak }
   Found : boolean   { hasil pencarian orangtua anak }
```

```
ALGORITMA
   { Search pada list Anak berdasarkan nama. Jika ketemu, akses Bapaknya }
   PtrAnak ← FirstAnak; Found ← false
   while (PtrAnak ≠ Nil) and (not Found) do
      if (Nama(PtrAnak) = NamaAnak) then
         Found ← true
      else
         PtrAnak ← NextAnak(PtrAnak)
   { PtrAnak = Nil or Found }
   if (Found) then
      output (Nama(Father(PtrAnak)))
```

```
procedure AddAnak (input/output FirstPeg : ListPeg, input NIPPeg : integer,
                   input/output FirstAnak : ListAnak,
                   input NamaAnak : string, input TglLahirAnak : integer)
{ Alternatif Kedua }
{ Mendaftar seorang anak yang baru lahir,
  Cari Pegawai, insert anak pada list anak selalu pada awal list }
{ I.S. List Pegawai terdefinisi }
{ F.S. List FirstPeg terdefinisi.
       Jika pegawai dengan NIP=NIPPeg ada, alokasi anak.
       Jika alokasi berhasil, insert seorang anak sebagai elemen pertama list
       anak, tentukan Bapak.
       Jika alokasi gagal atau NIPPeg tidak ada, hanya menulis pesan. }
```

```
KAMUS LOKAL
   PtrPeg : AdrPeg    { address untuk traversal, @ sebuah elemen list pegawai }
   PtrAnak : AdrAnak  { address untuk traversal, @ sebuah elemen list anak }
   FoundNIP : boolean { hasil pencarian NIP pegawai sebelum insert anak }
```

```
ALGORITMA
   { Search Pegawai dengan NIP yang diberikan: skema search dengan boolean}
   FoundNIP ← false
   PtrPeg ← FirstPeg
   while (PtrPeg ≠ Nil) and (not FoundNIP) do
      if (NIP(PtrPeg) = NIPPeg) then
         FoundNIP ← true
      else
         PtrPeg ← NextPeg(PtrPeg)
   { PtrPeg = Nil or FoundNIP }
   { Akhir search pegawai : PtrPeg=Nil or FoundNIP }
   if (FoundNIP) then { Insert anak }
      Alokasi(PtrAnak)
      if (PtrAnak ≠ Nil) then
         TglLahir(PtrAnak) ← TglLahir
         Nama(PtrAnak) ← NamaAnak
         Father(PtrAnak) ← Ptrpeg { Tentukan Bapaknya }
         NextAnak(PtrAnak) ← Nil
         { Insert Anak }
         if (FirstAnak ≠ Nil) then
            NextAnak(PtrAnak) ← FirstAnak
            FirstAnak ← PtrAnak
      else { Alokasi gagal : tidak melakukan apa-apa, hanya pesan }
         output ("Alokasi gagal")
   else  { NIPPeg tidak ada, error }
      output ("Pegawai tidak ada dalam list")
```

Latihan Soal

1. Buatlah spesifikasi yang lebih jelas dan persis tentang output (daftar) yang harus dikeluarkan oleh ke lima prosedur yang disebutkan di atas, misalnya tentang urutan daftar.

2. Pelajarilah pendefinisian primitif-primitif (prosedur) untuk kedua alternatif struktur data.
   Mengapa prosedur yang sama, untuk alternatif struktur data pertama dan kedua spesifikasi parameternya berbeda ?

3. Bandingkanlah algoritma dan kedua alternatif struktur data di atas, dari sudut :
   - kemudahan proses,
   - pemakaian memori.

4. List anak harus dikelola dengan cara lain :
   - Pada alternatif kedua, penyisipan anak harus dilakukan sehingga urutan dalam list adalah urutan kelahiran anak (anak termuda adalah elemen list yang terakhir). Adakah usulan modifikasi struktur data?
   - Pada alternatif kedua, list anak harus urut sesuai dengan urutan kemunculan pegawai dan juga umur yang mengecil (sesuai urutan kelahiran).

   Apa implikasinya terhadap proses-proses yang disebut di atas ?

5. Apa yang harus diubah, jika ada lebih dari satu pegawai atau anak dengan nama yang sama?
   Usulkan beberapa cara mengidentifikasi anak yang unik.

6. Pada alternatif kedua, algoritma untuk mendaftar anak yang mendapat tunjangan akan menghasilkan output dengan daftar anak tidak terkelompok menurut Bapaknya.
   Tuliskan algoritma yang akan mencetak dengan urutan sbb:

   Bapak: X Anak:
   - YYY
   - ZZZ

7. Satu element list menungkin menjadi anggota dari beberapa list. Misalnya pada alternatif kedua, selain list anak, setiap pegawai mempunyai juga list hobby, sehingga hobby yang sama dapat dipunyai oleh lebih dari seorang pegawai. Rancanglah struktur data yang mungkin dan operasi apa saja yang mungkin dilakukan terhadap pegawai dan hobby. Buat definisi kamus untuk persoalan ini.

Studi Kasus 5: Representasi Relasi N-M

Deskripsi Persoalan

Suatu sistem harus mengelola informasi Dosen, MataKuliah, dan Pengajaran (MataKuliah MK diajar oleh Dosen D). Aturan yang ada adalah: Seorang Dosen boleh mengajar lebih dari satu MataKuliah, dan satu MataKuliah dapat diajar oleh lebih dari satu Dosen. Setiap Dosen dan setiap MataKuliah mempunyai ciri unik. Dosen dikenal dari inisialnya dan MataKuliah dari kode kuliahnya. Dikatakan bahwa ada relasi antara Dosen dengan MataKuliah, dan relasinya adalah N-M (bandingkan dengan relasi pada ayah dan anak yang ada pada studi kasus sebelumnya, yang disebut sebagai relasi 1-N karena satu pegawai dapat mempunyai lebih dari satu anak, tetapi seorang anak hanya boleh mempunyai satu dan hanya satu ayah/pegawai).

Karena sistem mengelola lebih dari satu Dosen, sekumpulan Dosen dikelola sebagai list of Dosen, dan sekumpulan MataKuliah akan dikelola sebagai list of MataKuliah. Masalahnya, adalah bagaimana merepresentasi list of "relasi".

Ada **tiga alternatif** untuk merepresentasikan relasi tersebut:

- Relasi "Mengajar", yaitu untuk merepresentasi relasi dari sudut pandang setiap Dosen: setiap Dosen mempunyai list MataKuliah yang diajarnya. Hubungan ini 1-N dipandang dari list Dosen. Dengan representasi ini, setiap elemen list Dosen akan mempunyai list of MataKuliah yang diajarnya.
- Relasi "Diajar Oleh", yaitu untuk merepresentasi relasi dari sudut pandang setiap MataKuliah: sebuah MataKuliah diajarkan oleh siapa saja. Hubungan ini adalah hubungan 1-N dipandang dari list MataKuliah. Dengan representasi ini, setiap elemen list MataKuliah akan mempunyai list of Dosen pengajarnya
- Relasi hubungan "Dosen_MataKuliah", yaitu untuk merepresentasi setiap relasi MataKuliah dan Dosen: setiap elemen relasi <Dosen, MataKuliah> adalah unik, maka relasi hubungan ini dikelola sebagai list yang terpisah.

Jadi, sistem ini secara lojik mengelola tiga buah list: list Dosen, list MataKuliah dan list relasi.

Contoh situasi untuk Jurusan Teknik Informatika ITB, dengan hanya beberapa dosen dan matakuliah:
Dosen : IL, SP, SR, SA, ZZ
MataKuliah : IF221, IF621, IF624

Contoh:
Relasi pengajaran yang disebut **"Mengajar"** dapat digambarkan sebagai berikut:

Dosen	**Mengajar Matakuliah**
IL	IF221, IF621, IF624
SP	IF222, IF624
SR	IF222, IF358
SA	IF621, IF221
ZZ	-

Atau sebagai relasi **"Diajar Oleh"** sebagai berikut:

Matakuliah	Diajar Oleh
IF221	IL,SA
IF222	SP, SR
IF358	SR
IF621	IL,SA
IF624	IL,SP

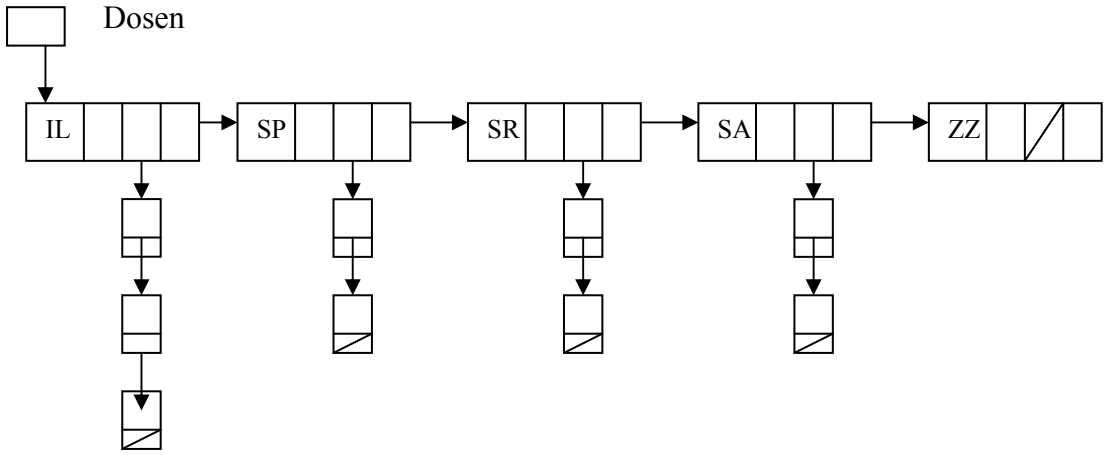
Atau sebagai relasi pasangan **"Dosen_MataKuliah"** sebagai berikut:

Dosen	Matakuliah
IL	IF221
SA	IF221
SP	IF222
SR	IF222
SR	IF358
IL	IF621
SA	IF621
IL	IF624
SP	IF624

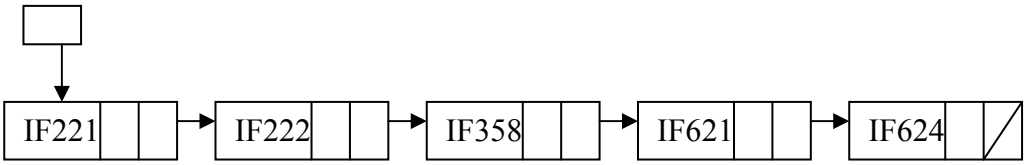
Jika informasi yang dikandung elemen list Dosen dan MataKuliah tidak hanya kode unik, melainkan juga informasi lain, maka relasi cukup digambarkan melalui *key*-nya seperti di atas. Misalnya Dosen selain disimpan inisialnya, juga disimpan nama, alamat, nomor telpon, dan sebagainya. MataKuliah selain kodenya, juga disimpan : judul, sks, prerequisit, dan sebagainya.

Alternatif Struktur Data

Alternatif 1: Relasi Direpresentasi sebagai "Mengajar"

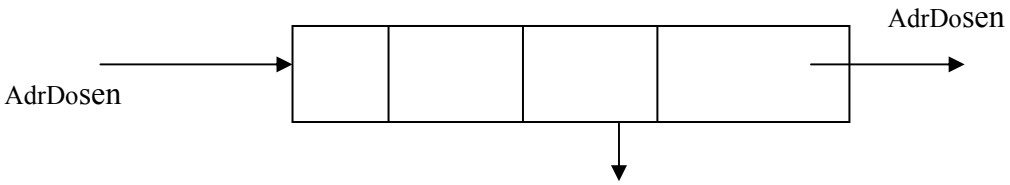


MataKuliah



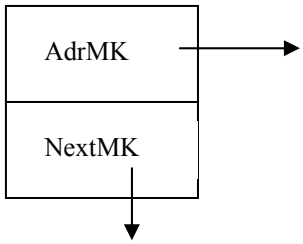
Dengan elemen list sebagai berikut:

Elemen list Dosen:



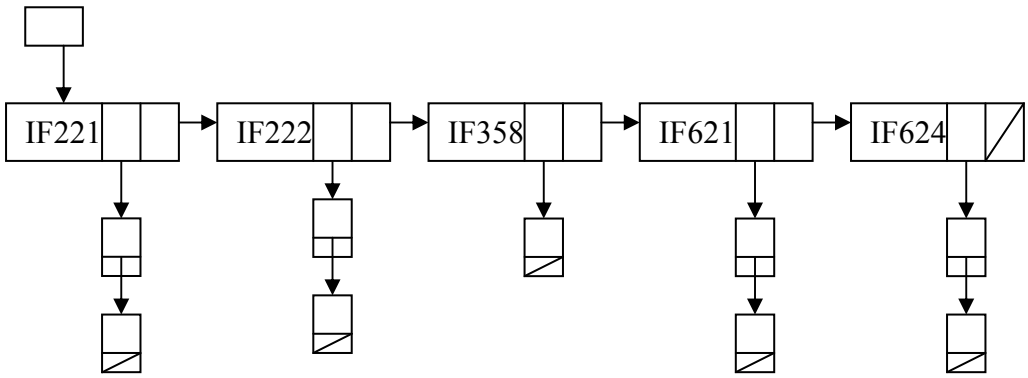
Elemen list MataKuliah:

Elemen list relasi:

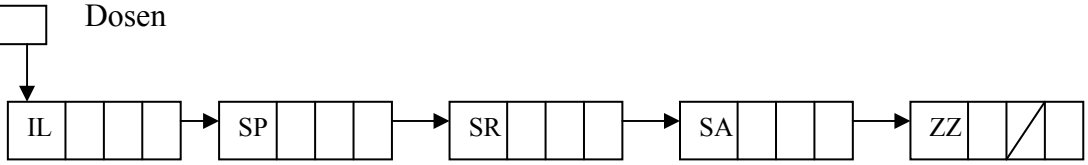


Alternatif 2: MataKuliah Menjadi Parent dan Relasi "Diajar Oleh"

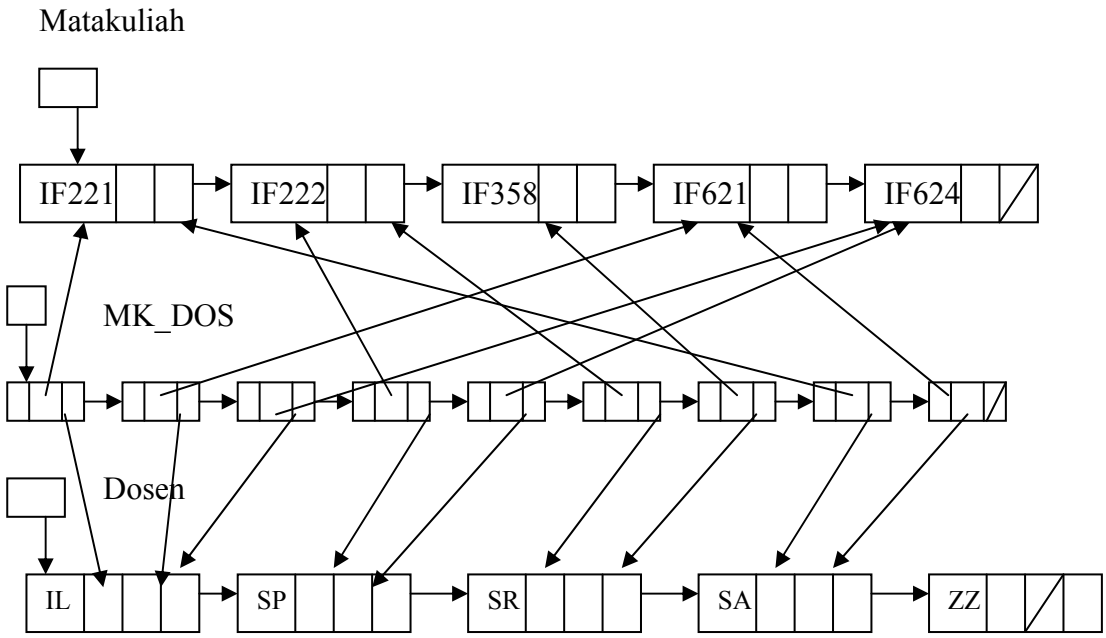
MataKuliah



Dosen



Alternatif 3: Setiap Relasi Dosen_MataKuliah Dibuat Menjadi Sebuah Elemen List Relasi



Buatlah studi perbandingan terhadap ketiga representasi tersebut dari segi memori dan efisiensi proses, jika misalnya sistem harus mampu minimal menjawab beberapa pertanyaan yang Anda definisikan.

Realisasikanlah prosedur dengan spesifikasi sebagai berikut :
Procedure **AddRel** yang menerima <D,MK> dan menambahkan sebuah relasi Dosen MataKuliah, dengan ketentuan:

- jika D belum ada di list Dosen maka ditambahkan lebih dulu sebagai elemen list. Demikian pula jika MK belum ada, maka sebelum menambahkan relasi, elemen list matakuliah ditambahkan lebih dulu.
- jika D dan MK sudah ada pada list Dosen dan Matakuliah, maka <D,M> harus belum muncul dalam list relasi (harus unik).

Sebuah relasi dapat terjadi antara elemen list yang sama. Buatlah sebuah studi, andaikata relasi ditambah dengan relasi sebagai berikut:

- relasi antar Dosen, misalnya “bersahabat”, yaitu dosen yang satu sama lain bersahabat.
- relasi antar MataKuliah yang disebut sebagai “prerequisit”.

Studi Kasus 6: Topological Sort

Deskripsi Persoalan

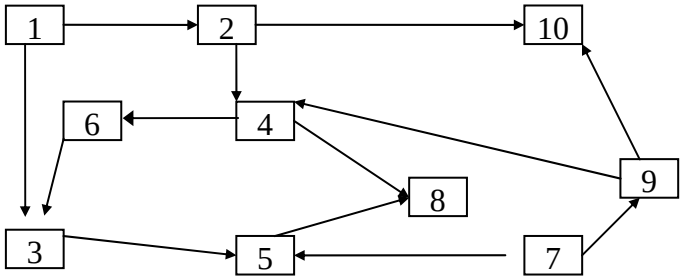
Diberikan urutan partial dari elemen suatu himpunan, dikehendaki agar elemen yang terurut partial tersebut mempunyai keterurutan linier.

Contoh dari keterurutan partial banyak dijumpai dalam kehidupan sehari-hari, misalnya:

1. Dalam suatu kurikulum, suatu mata pelajaran mempunyai prerequisit mata pelajaran lain. Urutan linier adalah urutan untuk seluruh mata pelajaran dalam kurikulum
2. Dalam suatu proyek, suatu pekerjaan harus dikerjakan lebih dulu dari pekerjaan lain (misalnya membuat pondasi harus sebelum dinding, membuat dinding harus sebelum pintu. Namun pintu dapat dikerjakan bersamaan dengan jendela. Dan sebagainya.
3. Dalam sebuah program Pascal, pemanggilan prosedur harus sedemikian rupa, sehingga peletakan prosedur pada teks program harus seuai dengan urutan (*partial*) pemanggilan.

Dalam pembuatan tabel pada basis data, tabel yang di-*refer* oleh tabel lain harus dideklarasikan terlebih dulu. Jika suatu aplikasi terdiri dari banyak tabel, maka urutan pembuatan tabel harus sesuai dengan definisinya.

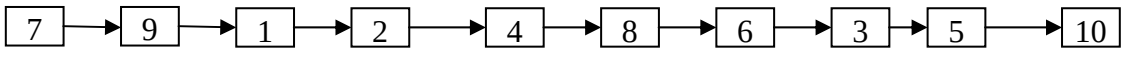
Jika X < Y adalah simbol untuk X "sebelum" Y, dan keterurutan *partial* digambarkan sebagai graf, maka graf sebagai berikut :



akan dikatakan mempunyai keterurutan *partial*:

1<2 2<4 4<6 2<10 4<8 6<3 1<3
3<5 5<8 7<5 7<9 9<4 9<10

Dan **SALAH SATU** urutan linier adalah graf sebagai berikut :



Kenapa disebut salah satu urutan linier? Karena suatu urutan *partial* akan mempunyai banyak urutan linier yang mungkin dibentuk dari urutan partial tersebut. Elemen yang membentuk urutan linier disebut sebagai "list".

Proses yang dilakukan untuk mendapatkan urutan linier :
1. Andaikata item yang mempunyai keterurutan partial adalah anggota himpunan S.
2. Pilih salah satu item yang tidak mempunyai predesesor, misalnya X. Minimal ada satu elemen semacam ini. Jika tidak, maka akan *looping*.
3. Hapus X dari himpunan S, dan insert ke dalam list.
4. Sisa himpunan S masih merupakan himpunan terurut partial, maka proses 2-3 dapat dilakukan lagi terhadap sisa dari S.
5. Lakukan sampai S menjadi kosong, dan list Hasil mempunyai elemen dengan keterurutan linier.

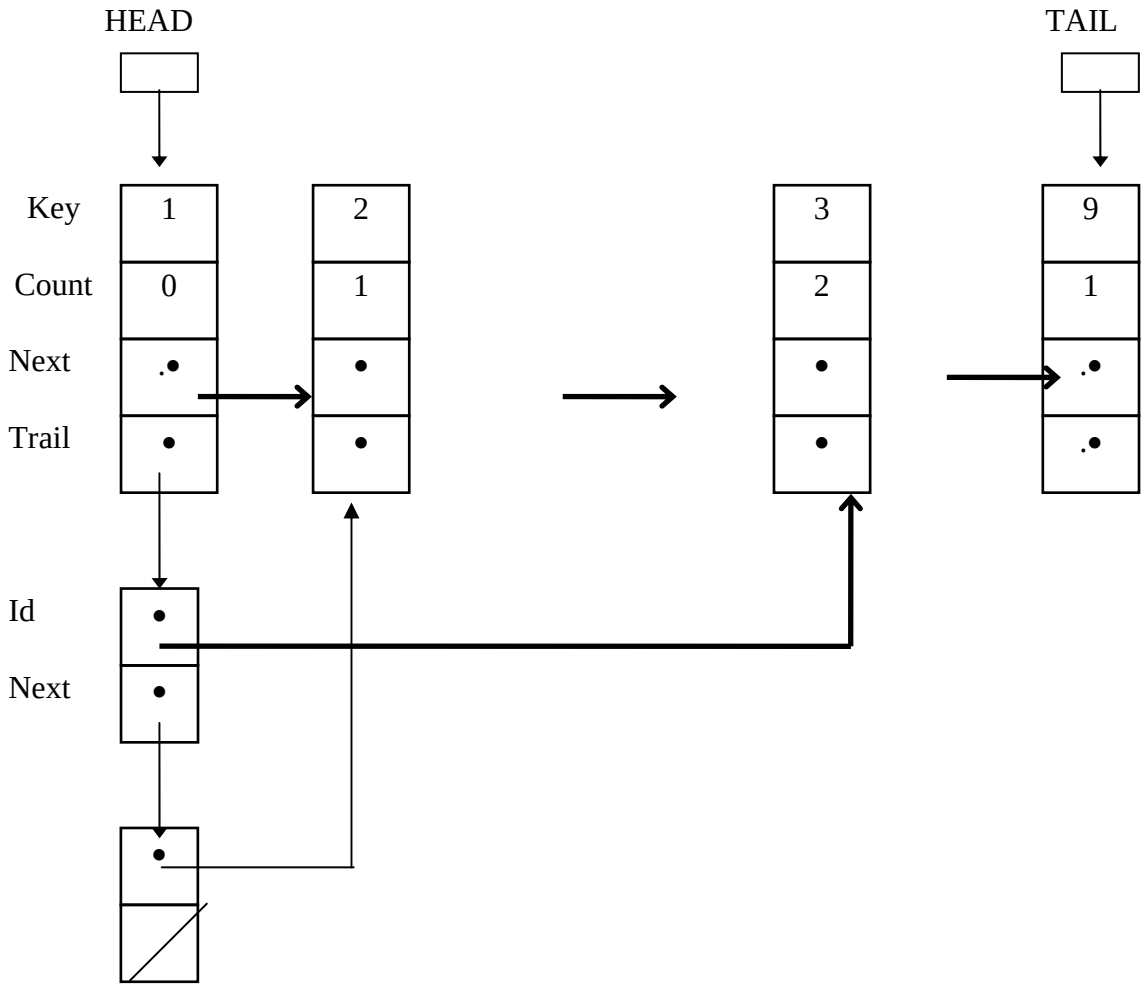
Solusi I

Untuk melakukan hal ini, perlu ditentukan suatu representasi internal. Operasi yang penting adalah memilih elemen tanpa predesesor (yaitu jumlah predesesor elemen sama dengan nol). Maka setiap elemen mempunyai 3 karakteristik: identifikasi, list suksesornya, dan banyaknya predesesor. Karena jumlah elemen bervariasi, representasi yang paling cocok adalah list berkait dengan representasi dinamis (pointer). List dari suksesor direpresentasi pula secara berkait. Ilustrasi struktur data dan program dalam bahasa Pascal yang diadaptasi dari buku [Wirth86] diberikan sebagai berikut.

Representasi yang dipilih untuk persoalan ini adalah multilist sebagai berikut :
1. List yang digambarkan horisontal adalah list dari banyaknya predesesor setiap item, disebut list "Leader", yang direpresentasi sebagai list yang dicatat alamat elemen pertama dan terakhir (Head-Tail) serta elemen terurut menurut *key*. List ini dibentuk dari pembacaan data. Untuk setiap data keterurutan partial X<Y: Jika X dan/atau Y belum ada pada list Leader, *insert* pada Tail dengan metoda *search* dengan sentinel.
2. List yang digambarkan vertikal (ke bawah) adalah list yang merupakan *indirect adressing* ke setiap predesesor, disebut sebagai "Trailer". Untuk setiap elemen list Leader X, list dari suksesornya disimpan sebagai elemen list Trailer yang setiap elemennya berisi alamat dari suksesor. Penyisipan data suatu suksesor (X<Y), dengan diketahui X, akan dilakukan dengan InsertFirst alamat Y sebagai elemen list Trailer dengan *key* X.

Algoritma secara keseluruhan terdiri dari dua pass
1. Bentuk list Leader dan Trailer dari data keterurutan partial: baca pasangan nilai (X<Y). Temukan alamat X dan Y (jika belum ada sisipkan), kemudian dengan mengetahui alamat X dan Y pada list Leader, InsertFirst alamat Y sebagai trailer X.
2. Lakukan *topological sort* dengan melakukan *search* list Leader dengan jumlah predesesor = 0, kemudian *insert* sebagai elemen list linier hasil pengurutan.

Ilustrasi umum dari list LEADER dan TRAILER untuk representasi internal persoalan topological sorting adalah sebagai berikut:



Solusi berikutnya diberikan pada [Wirth86].

Solusi II: Pendekatan "Fungsional" dengan List Linier Sederhana.

Pada solusi ini, proses untuk mendapatkan urutan linier diterjemahkan secara fungsional, dengan representasi sederhana. Graf partial dinyatakan sebagai list linier dengan representasi fisik First-Last dengan *dummy* seperti representasi pada Solusi I. Dengan elemen yang terdiri dari <Precc, Succ>. Contoh: sebuah elemen bernilai <1,2> artinya 1 adalah predesesor dari 2.

Langkah :

1. Fase input: Bentuk list linier yang merepresentasi graf seperti pada solusi I.
2. **Fase output: Ulangi langkah berikut sampai list "habis",** artinya semua elemen list selesai ditulis sesuai dengan urutan total.
   - P adalah elemen pertama (First(L)).
   - *Search* pada sisa list, apakah X=Precc(P) mempunyai predesesor.
     - Jika ya, maka elemen ini harus dipertahankan sampai saatnya dapat dihapus dari list untuk dioutputkan:
       - *Delete* P, tapi jangan didealokasi.
       - *Insert* P sebagai Last(L) yang baru.
     - Jika tidak mempunyai predesesor, maka X siap untuk dioutputkan, tetapi Y masih harus dipertanyakan. Maka langkah yang harus dilakukan :
       - Outputkan X.
       - *Search* apakah Y masih ada pada sisa list, baik sebagai Precc maupun sebagai Succ.
         - Jika ya, maka Y akan dioutputkan nanti. Hapus elemen pertama yang sedangkan diproses dari list.
         - Jika tidak muncul sama sekali, berarti Y tidak mempunyai predesesor, maka outputkan Y, baru hapus elemen pertama dari list.

Representasi node dengan address *type* terdefinisi:

```
type node : < Precc : integer,
              Succ  : integer,
              Next  : address  >
type ListTopo : < First : address,
                  Last  : address  >
```

Maka akses sebuah elemen list adalah : Precc(P), Succ(P), Next(P)

Untuk solusi II ini, diperlukan primitif *search* apakah sebuah nilai X muncul sebagai Precc(P) atau Succ(P) pada list "sisa" (list tanpa elemen pertama). Maka untuk efisiensi proses, kedua macam *search* bisa digabungkan dengan spesifikasi:

```
function SearchTopo (L : ListTopo, X : integer) → <boolean, boolean>
{ Mengirimkan <true,true> jika X muncul sebagai Precc(P) dan Succ(P).
  Mengirimkan <true,false> jika X muncul hanya sebagai Precc(P).
  Mengirimkan <false,true> jika X muncul hanya sebagai Succ(P).
  Mengirimkan <false,false> jika X tidak muncul sama sekali pada list L.
}
```