IF1210 Algoritma dan Pemrograman 1

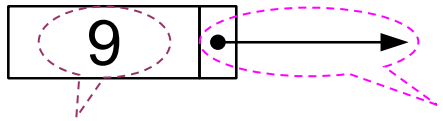
ADT List dengan Representasi Berkait

Tim Pengajar IF1210

Sekolah Teknik Elektro dan Informatika

Definisi

- **Struktur berkait** terdiri atas *node* yang terkait dengan *node* lain.
- **Node** merupakan sebuah *tuple* yang terdiri atas dua bagian:
  1) Sebuah nilai dengan tipe tertentu (info),
  2) Sebuah penunjuk ke *node* lain (next).
     Bisa jadi tidak menunjuk ke mana pun (NIL).
- Ilustrasi:



info bertipe integer **next** (Address ke *node* lain)

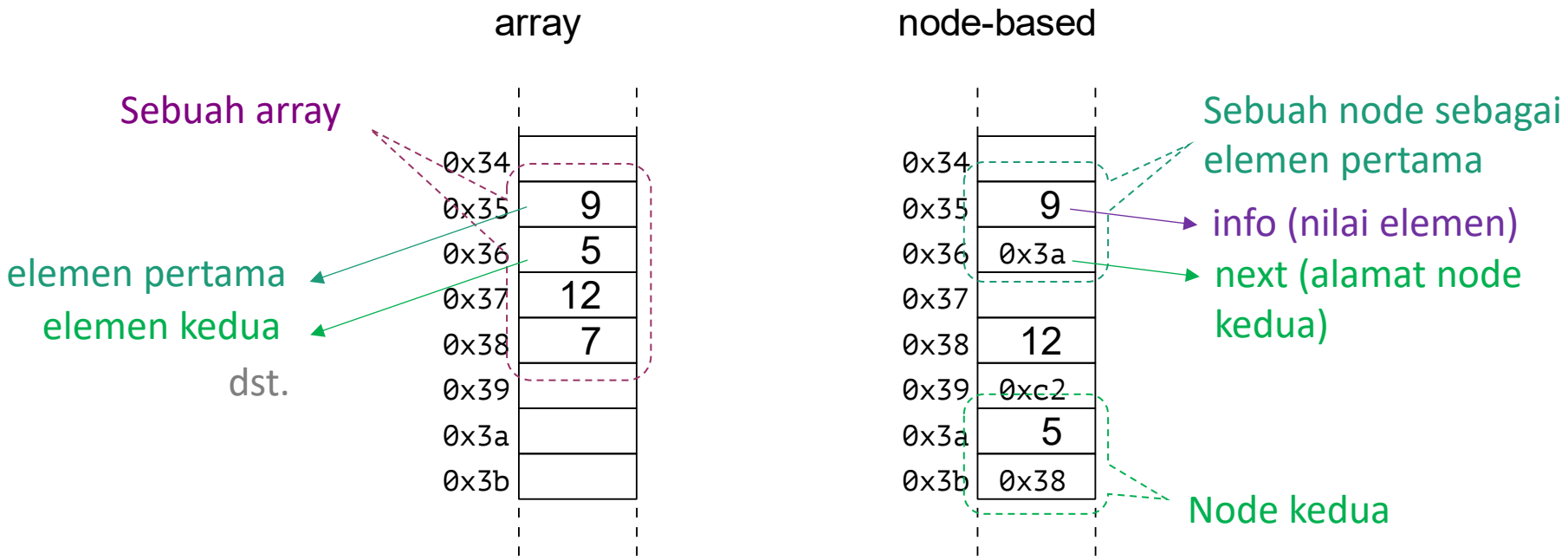
- Hal ini memungkinkan penyimpanan elemen-elemen tanpa harus kontigu.
- Disebut juga struktur *node-based*, *linked list*, atau *linear list*.

Memori fisik: array vs. node-based

Array*:* elemen-elemen berada pada lokasi bersebelahan.

Node*:* "next" mencatat alamat elemen berikutnya.

Contoh, dengan elemen bertipe *byte* (1 elemen mengisi 1 slot memori):



Karakteristik struktur data berbasis node

- Memori dialokasi sesuai kebutuhan.
  - Jika ada 3 elemen maka hanya perlu memori sebesar ukuran *node* ×3.
  - Berbeda dengan *array* yang, misalnya sudah dialokasikan 100 elemen maka menggunakan memori sebesar ukuran elemen ×100 meskipun pada suatu waktu hanya 3 elemen yang efektif.
- Ukuran memori per elemen menjadi lebih besar.
  - Ukuran elemen + ukuran *pointer*.
- Secara umum mengorbankan efisiensi ruang (memori) demi efisiensi waktu.
  - (Tidak untuk semua jenis operasi.)

Node dalam Notasi Algoritmik

```
{ Deklarasi tipe bentukan }
  type ElType: integer
  type Address: pointer to Node
  type Node: < info: ElType,
               next: Address >

{ Deklarasi variabel }
  p1: Address
  p2: Address

{ Inisialisasi dan penggunaan variabel }
  p1 ← alokasi(9) { p1 menunjuk ke Node dengan info=9 dan next=NIL }
  p2 ← alokasi(5) { p2 menunjuk ke Node dengan info=5 dan next=NIL }

  p1↑.next ← p2   { Address next pada p1 menunjuk ke node yang ditunjuk p2 }
```

Node dalam Bahasa C (deklarasi tipe bentukan)

```c
/* node.h */
#ifndef NODE_H
#define NODE_H

typedef int ElType;
typedef struct node* Address;
typedef struct node {
  ElType info;
  Address next;
} Node;

#define INFO(p) (p)->info
#define NEXT(p) (p)->next
Address newNode(ElType val);

#endif
```

```c
/* node.c */
#include "node.h"
#include <stdlib.h>

Address newNode(ElType val) {
  Address p = (Address)
              malloc(sizeof(Node));
  if (p!=NULL) {
    INFO(p) = val;
    NEXT(p) = NULL;
  }
  return p;
}
```

Node dalam Bahasa C (contoh penggunaan)

```
/* Deklarasi variabel */
Address p1, p2;

/* Inisialisasi dan penggunaan variabel */
p1 = newNode(9); /* p1 menunjuk ke Node dengan info=9 dan next=NIL */
p2 = newNode(5); /* p2 menunjuk ke Node dengan info=5 dan next=NIL */

NEXT(p1) = p2; /* Address next pada p1 menunjuk ke node yang ditunjuk p2 */
```

Implementasi ADT List dengan Struktur Berkait

Kembali ke definisi List

List, dikenal juga dengan *sequence*, merupakan sekumpulan elemen **bertipe sama** yang memiliki suatu **keterurutan tertentu** *(ordered*, tidak harus *sorted).*

Operasi-operasi:

- isEmpty
- indexOf
- length
- akses (getElmt, setElmt)
- concat
- insert-
- delete-
- pola traversal

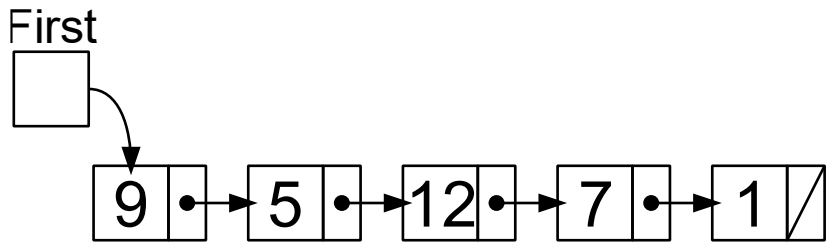
Implementasi List dengan struktur berkait

Elemen-elemen direpresentasikan dengan **Node** ⟨Info, Next⟩ yang saling berkait.

List diacu melalui **Address** elemen pertamanya (**first**).

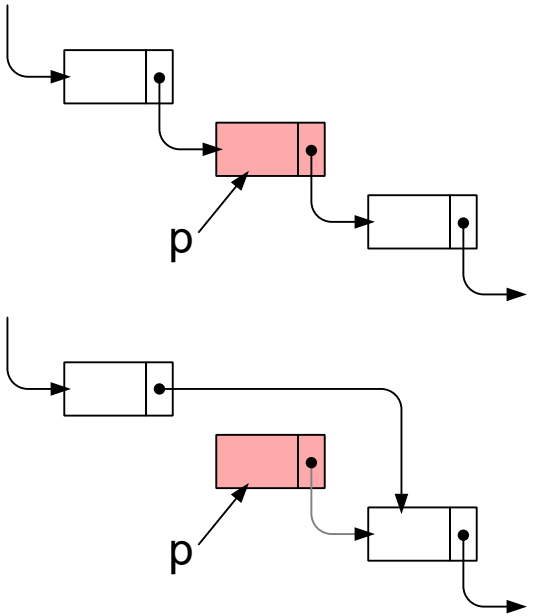
Alamat elemen berikutnya (suksesor) diakses dengan **next**.

Elemen terakhir ditandai dengan Next menunjuk ke **NIL**.

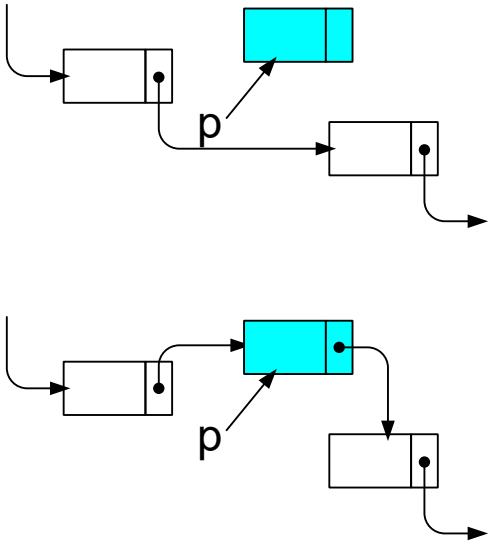


Karakteristik list linier

Penambahan & penghapusan elemen sangat sederhana.



penghapusan elemen



penambahan elemen

Tidak efisien untuk mengakses elemen melalui indeksnya.
(Harus menelusuri mulai dari *node* pertama sambil mencacah.)

Implementasi List dengan struktur berkait

- Jika l adalah List, dan p adalah Address:
  - Karena l diacu melalui alamat node pertamanya, maka first = l.
  - Elemen yang diacu oleh p dapat diakses informasinya dengan notasi:
    - **p↑.info**: nilai yang disimpan
    - **p↑.next**: alamat elemen berikutnya
- Definisi List Kosong
  - List l adalah list kosong: l = **NIL**
- Definisi Elemen terakhir
  - last↑.next = **NIL**, dengan last adalah alamat elemen terakhir

Dalam notasi algoritmik

```
{  Deklarasi tipe  }

type ElType: integer
type Address: pointer to Node
type Node: < info: ElType,
             next: Address >
type List: Address


{  Deklarasi variabel  }

l: List
p1: Address
p2: Address
```

```
{  Inisialisasi List  }

CreateList(l)

{  Akses node pertama:  }

p1 ← l

{  Cetak isi p1 & akses elemen
   setelah p1  }

output(p1↑.info)
p2 ← p1↑.next
```

Dalam bahasa C

```
/* Deklarasi tipe */

typedef int ElType;
typedef struct tNode* Address;
typedef struct tNode {
          ElType info;
          Address next; } Node;
typedef Address List;

/* Deklarasi variabel */
List l;
Address p1;
Address p2;
```

```
/* Inisialisasi List */

CreateList(*l)


/* Akses node pertama: */

p1 = l

/* Cetak isi p1 & akses elemen
   setelah p1 */

printf("%d\n", p1->info);
p2 = p1->next;
```

Skema Pemrosesan List Berkait

Skema dasar pemrosesan List berkait

- Traversal
- Pencarian sekuensial (*search*)

Skema traversal

- Digunakan untuk memroses setiap elemen List dengan cara yang sama.
- Mekanisme: mengunjungi setiap elemen List secara berurutan dimulai dari elemen pertama hingga elemen terakhir.
- Jenis traversal:
  - Dasar (tanpa perlakuan khusus untuk List kosong)
  - Dengan perlakuan khusus untuk List kosong
  - Untuk List yang tidak kosong

Skema traversal dasar

```
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 tanpa pemrosesan khusus untuk 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 ← l
  while (p ≠ NIL) do
    proses(p)
    p ← p↑.next
  terminasi
```

Skema traversal dengan penanganan list kosong

```
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 dengan pemrosesan khusus untuk List kosong. }
KAMUS LOKAL
  { SAMA SEPERTI SKEMA 1, tidak ditulis untuk menghemat tempat }
ALGORITMA
  if l = NIL then
    output("List kosong")
  else
    inisialisasi
    p ← l
    repeat
      proses(p)
      p ← p↑.next
    until (p = NIL)
    terminasi
```

Skema traversal untuk List tidak kosong

```
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. }
{ Traversal untuk List linier yang sudah dipastikan tidak kosong. }
KAMUS LOKAL
  { SAMA SEPERTI SKEMA 1 dan 2, tidak ditulis untuk menghemat tempat }
ALGORITMA
  inisialisasi
  p ← l
  iterate
    proses(p)
  stop (p↑.next = NIL)
    p ←  p↑.next
  terminasi
```

Skema pencarian sekuensial

- List linier tidak memungkinkan *binary search*.
- Mekanisme: mengunjungi elemen-elemen List secara berurutan dimulai dari elemen pertama hingga ditemukan elemen yang memenuhi syarat pencarian, atau semua elemen telah dikunjungi.
- Jenis skema pencarian:
  - Dengan boolean
  - Tanpa boolean

Pencarian berdasarkan nilai elemen (1)

```
procedure SKEMAlistSearch1(input l: List, input x: ElType,
                           output p: Address, output found: boolean)
{ I.S. List linier l sudah terdefinisi dan siap dikonsultasi, x terdefinisi }
{ F.S. p adalah address di mana x ditemukan, p = NIL jika tidak ketemu }
{      found menyatakan apakah nilai x yang dicari ditemukan }
{ Menggunakan skema search dengan boolean }
KAMUS LOKAL
  -
ALGORITMA
  p ← l
  found ← false
  while (p ≠ NIL) and (not found) do
    if (x = p↑.info) then
      found ← true
    else
      p ← p↑.next
  { p = NIL or found}
  { Jika found maka p = Address dari nilai yg dicari }
  { p = NIL jika nilai tidak ditemukan}
```

Pencarian berdasarkan nilai elemen (2)

```
procedure SKEMAlistSearch2(input l: List, input x: ElType,
                           output p: Address, output found: boolean)
{ I.S. List linier l sudah terdefinisi dan siap dikonsultasi, x terdefinisi }
{ F.S. p adalah address di mana x ditemukan, p = NIL jika tidak ketemu }
{      found menyatakan apakah nilai x yang dicari ditemukan }
{ Menggunakan skema search tanpa boolean }
KAMUS LOKAL
  -
ALGORITMA
  p ← l
  if p = NIL then  { List kosong }
    found ← false
  else { List tidak kosong }
    while (p↑.next ≠ NIL) and (p↑.info ≠ x) do
      p ← p↑.next
    { p↑.next = NIL or p↑.info = x }
    found ← (p↑.info = x)
    if (not found) then
      p ← NIL
```

Pencarian elemen yang memenuhi kondisi

```
procedure SKEMAlistSearchX(input l: List, input kondisi(p): boolean,
                           output p: Address, output 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. p adalah address di mana kondisi(p) terpenuhi, p = NIL jika tidak ketemu }
{      found menyatakan apakah ada p yang memenuhi kondisi(p) }
KAMUS LOKAL
  -
ALGORITMA
  p ← l
  found ← false
  while (p ≠ NIL) and (not found) do
    if kondisi(p) then
      found ← true
    else
      p ←  p↑.next
  { p = NIL or found}
  { Jika found maka p adalah elemen List dengan kondisi(p) true }
```

Latihan

1. Buatlah fungsi **countPos** yang menghitung banyaknya kemunculan bilangan positif (>0) dari sebuah list of integer l

```
function countPos(l: List) → integer
```

2. Buatlah fungsi **max** yang menghasilkan nilai maksimum dari suatu list of integer l yang tidak kosong

```
function max(l: List) → integer
```

3. Buatlah fungsi **searchPos** yang menghasilkan address di mana nilai positif pertama kali ditemukan di list of integer l

```
function searchPos(l: List) → address
```