TIB29 – Struktur Data dan Algoritma

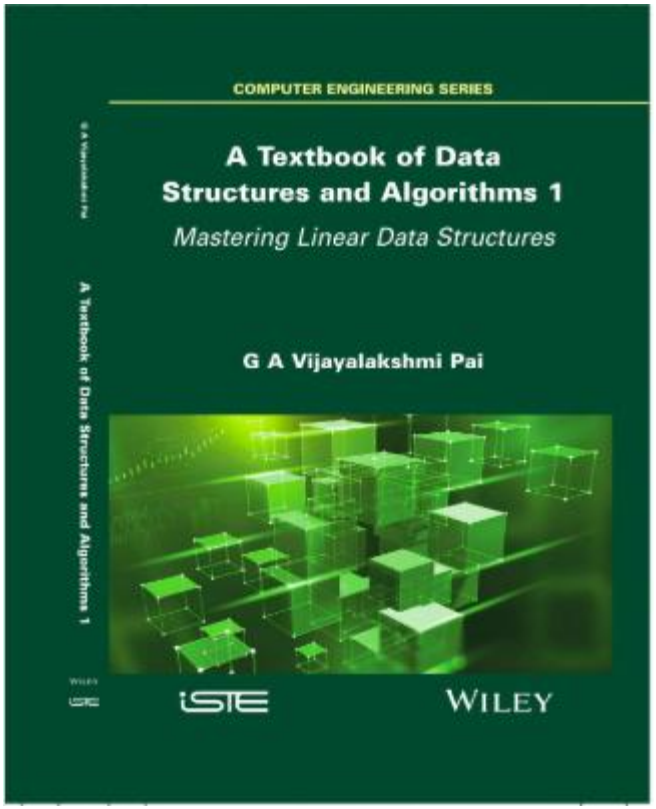
PERINGATAN HAK CIPTA

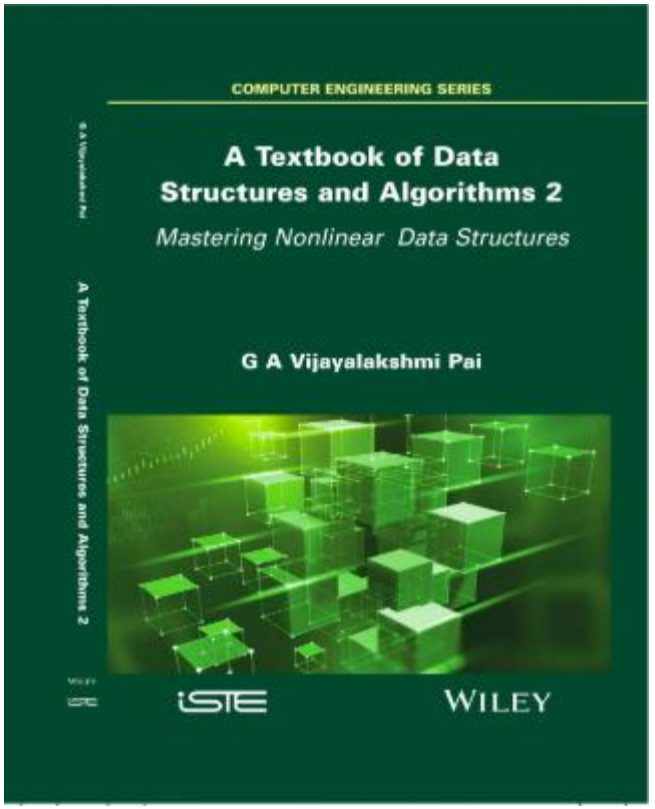
PERINGATAN HAK CIPTA

ubm
Universitas Bunda Mulia

Model-model *Linked-List*

Diadopsi Dari Sumber:





Sub-CPMK

- Mahasiswa mampu membuat berbagai model Linked-List beserta operasinya pada pemrograman. (C3, A3)

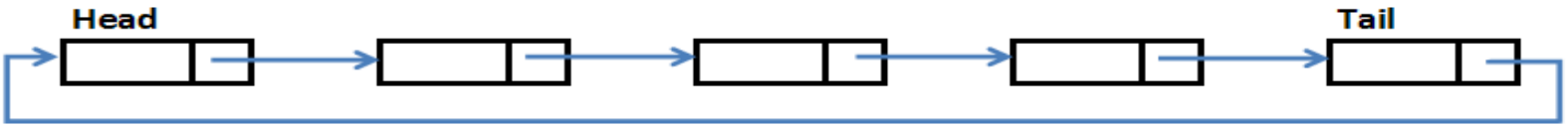
Materi

1. Circular Linked_List
2. Dobel Linked-List
3. Multiple Linked-List

1. Circular Linked-List

1.1. *Circular Linked List*

- *Next pointer* pada *tail*, menunjuk ke *Head*



1.2. Operasi pada *Circular Linked List*

- Sama seperti pada *Linked List* sebelumnya, *Circular linked list* juga memiliki operasi-operasi cari, sisip/tambah dan hapus dengan memperhatikan operasi jika dilakukan pada *Head* atau *Tail* agar *linked list* tetap circular.

1.3 *Insert Operation*

- Pada bagian depan *list*

  Sama seperti pada *single Linked-list*
- Pada bagian tengah *list*

  Sama seperti pada *single Linked-list*
- Pada bagian akhir *list*

  Perbedaan operasi sisip antara *single* LL dengan *Circular* LL adalah pada penentuan *next* dari node baru sebagai node terakhirnya. Jika pada *single* LL *next link* node baru di isi dengan NULL, maka pada *Circular* LL *link* node baru diisi dengan *Head*

1.3 *Insert Operation* (Lanj.)

Perbedaan operasi sisip antara *single* LL dengan *Circular* LL adalah pada penentuan *next* dari node baru sebagai node terakhirnya. Jika pada *single* LL *next link* node terakhir di isi dengan NULL, maka pada *Circular* LL *link* node terakhir diisi dengan *Head*

- Buat sebuah node baru

  Baru = malloc()
- Isi informasi node baru tersebut
- *Set next link* dari node Baru ke *Head*

  Baru->next = *Head*
- Arahkan *next link* pada *last* node atau *tail* ke node baru

  Tail->next = Baru

1.3 *Insert Operation* (Lanj.)

- Sebenarnya mengingat sifat *circular linked-list* yang *link* nya akhirnya kembali ke *link* awal, penambahan Node pada *circular linked list* dapat dilakukan dengan dengan cara yang sama seperti menambah node di tengah *list* pada *single Linked-List* tanpa membedakan penambahan pada *Head*, tengah atau *Tail*

1.4 Mencari Node Pada Circular LL

- Untuk mencari Node pada *Circular* LL memiliki keistimewaan, kita dapat menyimpan Alamat *current* node (node yang ditunjuk oleh *pointer* Ptr) ke suatu variabel *pointer*, kemudian lakukan pelacakan node satu persatu diawali dari *current* node sampai ditemui *next link* dari node yang diakses menunjuk ke current node atau sampai data yang dicari ditemukan.

- Contoh:

```
Head = Ptr;
//perhatikan!
While (Ptr->next != Head) OR (Ptr->dat != cari) do
{
  Ptr = Ptr->next
}
```

1.4 Mencari Node Pada Circular LL (Lanj.)

- Pencarian node Masih dapat dilakukan dengan cara yang sama dengan *single* LL juga dengan Pembatasan pencarian sampai ditemui Ptr->*next* = *Head*
- Contoh:

```
Ptr = Head;
While (Ptr->next != Head) OR (Ptr->dat != cari) do
{
   Ptr = Ptr->next
}
```

1.5 *Delete Operation*

- Pada bagian depan / *delete head*

  Proses sama seperti pada *single* LL
- Pada bagian tengah

  Proses sama seperti pada *single* LL
- Pada bagian akhir / *delete tail*

  Perbedaan operasi hapus antara *single* LL dengan *Circular* LL adalah pada penentuan *next* dari node sebelum node terakhir sebagai pengganti node terakhir yang akan dihapus.

  Jika pada *single* LL *next link* node sebelum node terakhir di isi dengan NULL, maka pada *Circular* LL link node sebelum node terakhir diisi dengan *Head*

1.5 *Delete Operation* (Lanj.)

- Lokasikan *pointer* ptrHapus ke node yang akan dihapus dan Ptr ke node sebelum node terakhir yang akan dihapus
- Isi *next link Pointer* Ptr dengan *Head* (dapat juga dilakukan dengan node berikut dari node terakhir yang tentu saja menunjuk ke *Head* juga)

  Ptr->*next* = *Head* atau Ptr->*next* = ptrHapus->*next*
- Hapus *current* node

  free(ptrHapus)

1.5 *Delete Operation* (Lanj.)

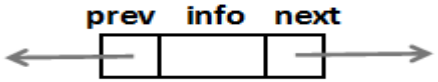
- Sebenarnya mengingat sifat *circular linked-list* yang *link* nya akhirnya kembali ke *link* awal, penghapusan Node pada *circular linked list* dapat dilakukan dengan dengan cara yang sama seperti menghapus node di tengah *list* pada single *Linked-List* tanpa membedakan penghapusan pada *Head*, tengah atau *Tail*

ubm
Universitas Bunda Mulia

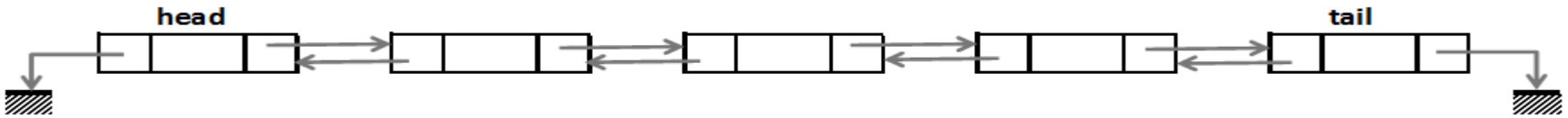
2. Double Linked-List

2.1. *Double Linked List*

- Setiap node memiliki dua *link*



- *Previous Link* menunjuk ke *previous* node
- *Next Link* menunjuk ke *next* node
- *Head* → Prev *Link Pointed as* NULL
- *Tail* → *Next Link Pointed as* NULL



2.2. Kelebihan *Double Linked List*

- Ketika kehilagan *Head*, asalkan *Pointer* masih menunjuk ke salah satu node/simpul, maka *Head* masih dapat dicari dengan melakukan *Previous* terus menerus sampai ditemukan simpul yang link *previous*nya menunjuk ke NULL

2.3 Double *Circular Linked List*

- *Double Linked List* Dapat menjadi *circular linked List* dengan cara mengarahkan *next link* pada *tail* ke *head* dan mengarahkan *previous link* pada *head* ke *Tail*

2.4 Operasi pada *Double Linked List*

- Sama seperti pada *Single Linked List*, *Double linked list* juga memiliki operasi-operasi cari, sisip/tambah dan hapus dengan mengarahkan *previous Link* ke *previous* node dari node yang akan dihapus
- Dengan adanya prev link pada tiap nodenya dapat mengurangi penambahan variabel *pointer* pada tiap operasinya

2.5 Penyisipan Node

- Pada Double Linked-List penyisipan dapat dilakukan pada awal, tengha maupun akhir linked-list
- Penyisipan harus memperhatikan link previous dan link next

2.5.1 *Insert* Pada Bagian Depan *List*

- Buat sebuah simpul/node baru

  ptrBaru = malloc(……..)
- Isi informasi simpul/node baru tersebut
- Arahkan *next link* ke simpul/node kepala

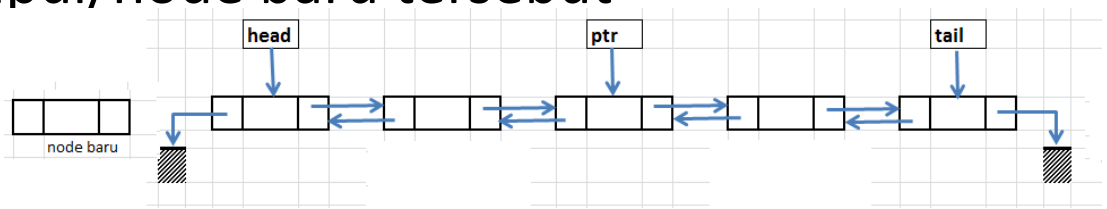
  ptrBaru->*next* = *Head*
- Arahkan prev *link* dari simpul kepala ke node baru

  Head->prev = ptrBaru
- Isi prev *link* node baru dengan NULL

  ptrBaru->prev = NULL
- *Set head pointer* ke simpul/node baru tersebut

  Head = ptrBaru

TMSM - Linked List



2.5.2 Insert di Tengah – Setelah *Current Cell*

- Arahkan Ptr ke node yang akan dibuat node baru pada *next link* nya
- Buat sebuah simpul/node baru

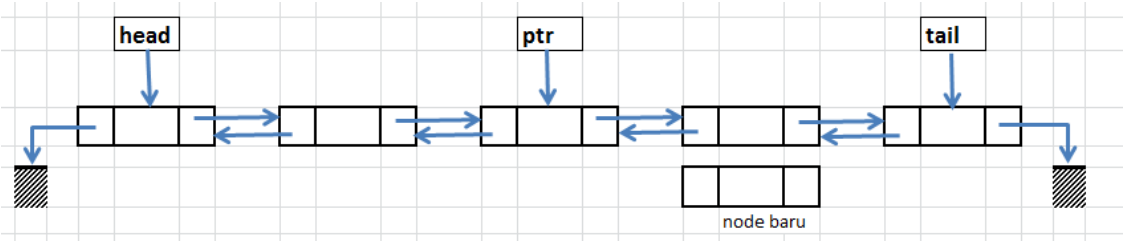
  Baru = malloc()
- Isi informasi simpul/node baru tersebut
- *Copy next link* dari *current* node/simpul *ke next link* simpul/node baru

  Baru->*next* = Ptr->*next*
- *Copy* prev *link* dari node setelah current node/simpul ke prev *link* simpul/node baru

  Baru->prev = Ptr->*next*->prev
- *Set* prev *link* pada simpul/node setelah current simpul/node kenode baru

  Ptr->*next*->prev = Baru
- *Set next link* pada current simpul/node ke simpul/node baru

  Ptr->*next* = Baru



2.5.3. Insert Di Tengah – Sebelum *Current Cell*

- Arahkan Ptr ke node yang akan dibuat node baru pada prev *link* nya
- Buat sebuah simpul/node baru

  Baru = malloc()
- Isi informasi simpul/node baru tersebut
- *Copy* prev *link* dari *current* node/simpul ke prev *link* simpul/*node* baru

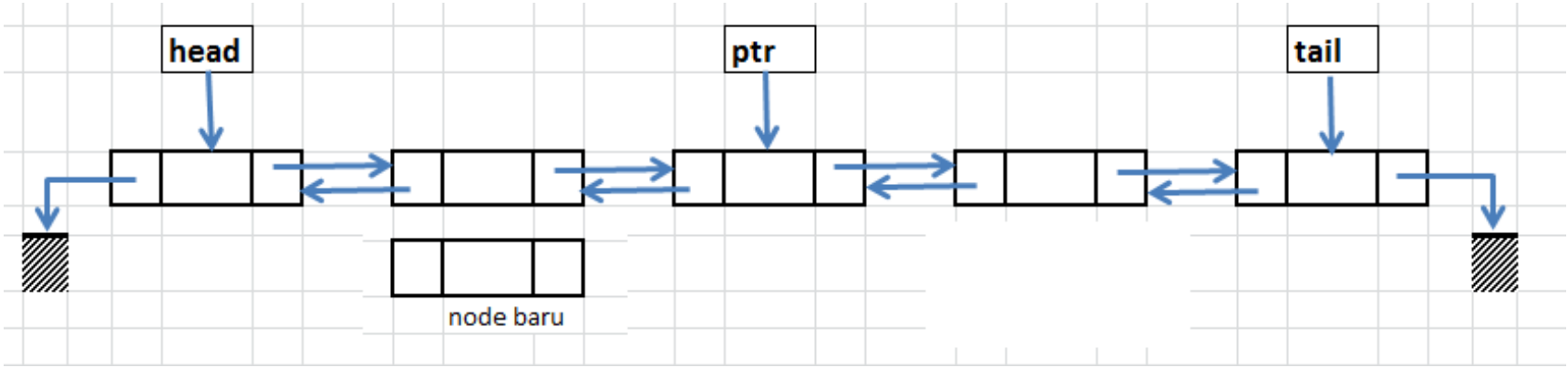
  Baru->prev = Ptr->prev
- *Copy next link* dari node sebelum *current* node/simpul ke *next link* simpul/node baru

  Baru->*next* = Ptr->prev->*next*
- *Set next link* pada simpul/node sebelum *current* simpul/node kenode baru

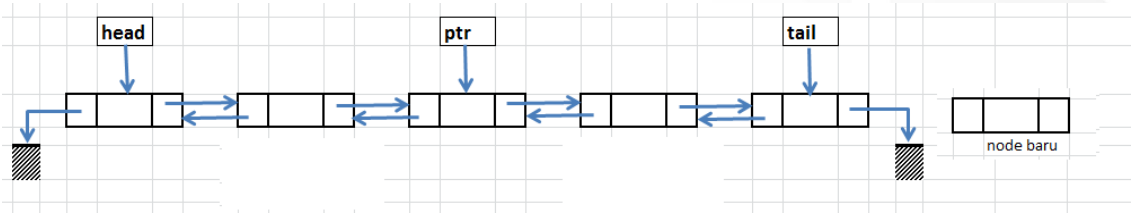
  Ptr->prev->*next* = Baru
- *Set* prev *link* pada *current* simpul/node ke simpul/node baru

  Ptr->prev = Baru

2.5.3. Insert Di Tengah – Sebelum *Current Cell* (Lanj.)



2.5.4. *Insert* Pada Bagian Akhir *List*



- Arahkan Ptr ke node terakhir
- Buat sebuah node baru

  Baru = malloc()
- Isi informasi node baru tersebut
- *Set next link* dari baru node sebagai NULL

  Baru->*next* = NULL
- Isi prev *link* dari node baru ke *Tail*

  Baru->prev = *TAIL*
- Arahkan *next link* pada node terakhir atau *tail* ke node baru

  Ptr->*next* = Baru
- Jika memiliki variabel *Tail*, Jangan lupa memindahkan *Tail* ke node baru

  Tail = Baru

2.6. Mencari Node Pada *Double* LL

- Mencari node pada *Double* LL dapat dilakukan dengan cara maju menggunakan *next link* atau mundur menggunakan prev *link*
- Jika maju maka Ptr diarahkan ke *Head*
- Jika mundur maka Ptr diarahkan ke *Tail*

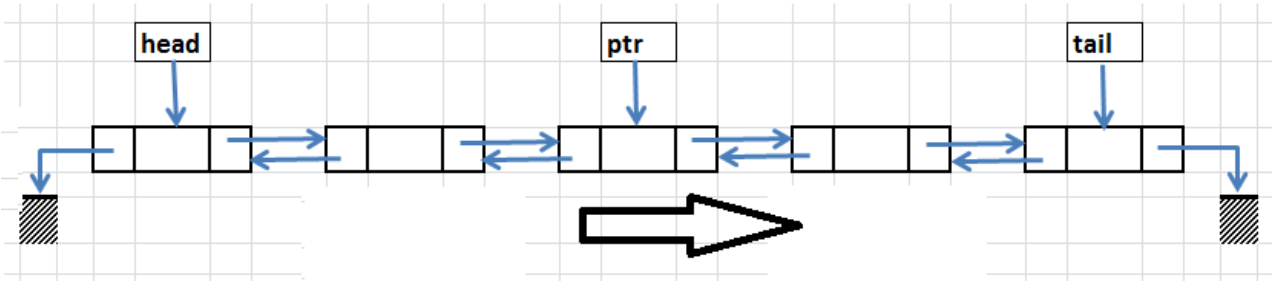
2.6.1 Pencarian Maju

- Proses sama seperti pada *single* LL
- Assign *PointerCell* sebagai *Head*

```
PointerCell = Head;
```

- Bergerak maju dengan mengarahkan *PointerCell* ke *next PointerCell* sampai ditemukan node/simpul yang sesuai.

```
PointerCell = PointerCell->Next;
```



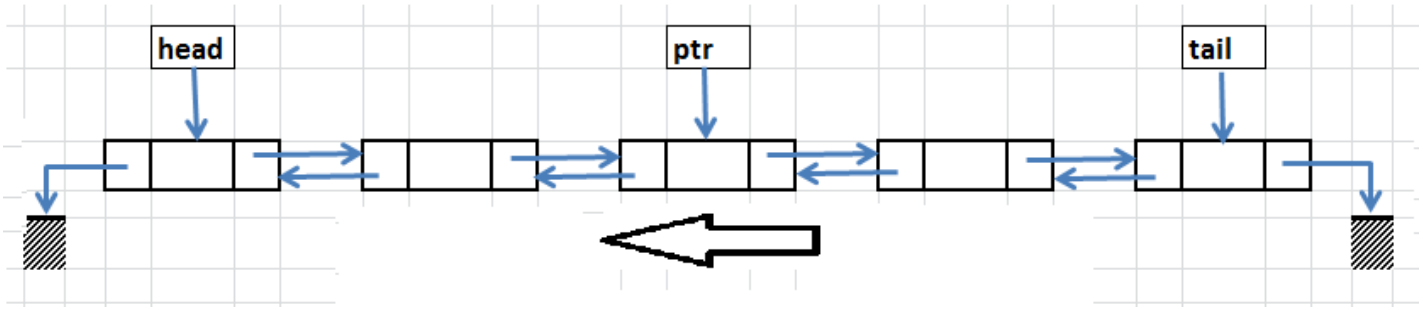
2.6.2 Pencarian Mundur

- *Assign PointerCell* sebagai *Head*

```
PointerCell = Tail;
```

- Bergerak mundur dengan mengarahkan *PointerCell* ke *prev PointerCell* sampai ditemukan node/simpul yang sesuai.

```
PointerCell = PointerCell->prev;
```



2.7 *Delete Operation*

Penghapusan dapat dilakukan:

- Pada bagian depan / *delete head (WARNING!!!: don't lose the head!)*
- Pada bagian tengah
- Pada bagian akhir */ delete tail*
- Karena ini adalah *double linked-list*, maka jangan lupa mengarahkan prev *link* dari node setelah node yang akan dihapus ke node pada prev *link* dari node yang akan dihapus

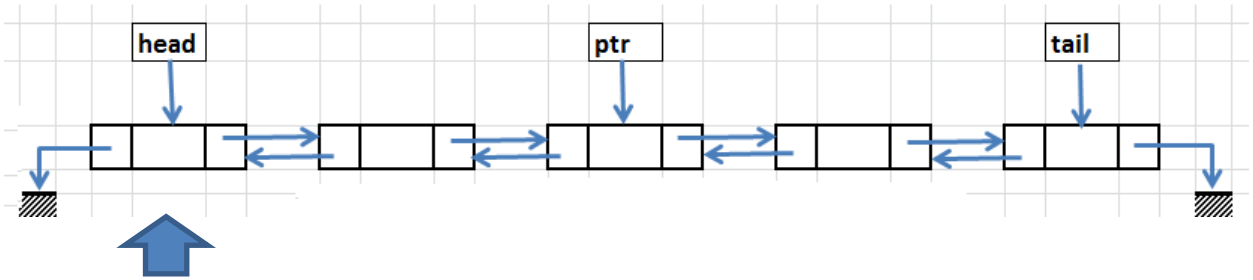
2.7.2. Hapus Pada Bagian Depan / *Delete Head*

- (*WARNING!!!: don't lose the head!*)
- Arahkan *pointer* Ptr ke *Head* node sebagai *current* node
- Set variabel *Head* ke *next* dari *current* node

  Head = Head->Next
- *Set* prev *link* dari *Head* dengan NULL

  Head->prev = NULL
- Hapus *current* Node

  Free(Ptr)



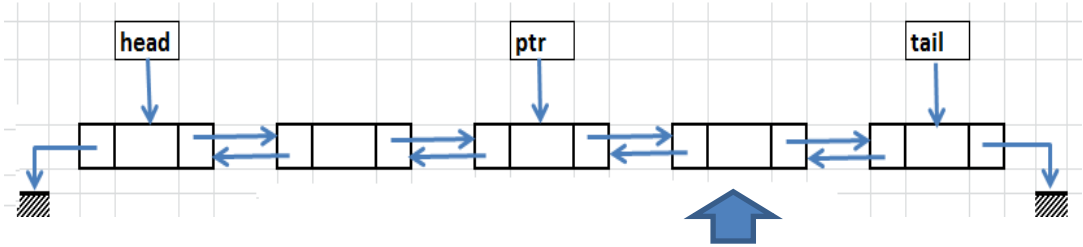
2.7.3. Hapus Pada Bagian Tengah

- Arahkan *Pointer* Ptr ke node yang akan dihapus
- *Copy link* prev node dari node yangakan dihapus ke prev link node setelah node yang akan dihapus

  Ptr->*next*->prev = Ptr->Prev
- *Copy link next* node dari node yang akan dihapus ke *next link* node sebelum node yang akan dihapus

  Ptr->prev->*next* = Ptr->prev
- Hapus *Current* Node

  Free(Ptr)

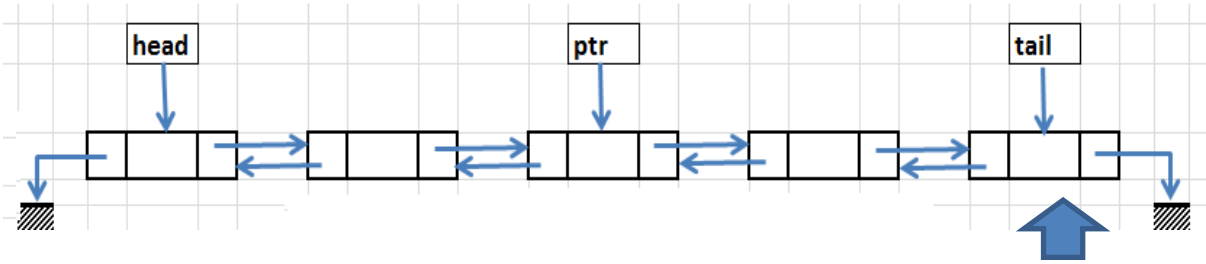


2.7.4 Hapus Pada Bagian Akhir / *Delete Tail*

- Arahkan *Pointer* Ptr ke node yang akan dihapus Atau dalam hal ini adalah node terakhir
- Ubahlah *next link* dari node sebelum node yang akan dihapus dengan NULL

  Ptr->prev->*next* = NULL
- Hapus *current* node

  Free(Ptr)



3. Multiple Linked-List

3.1. Varian Linked List Lainnya

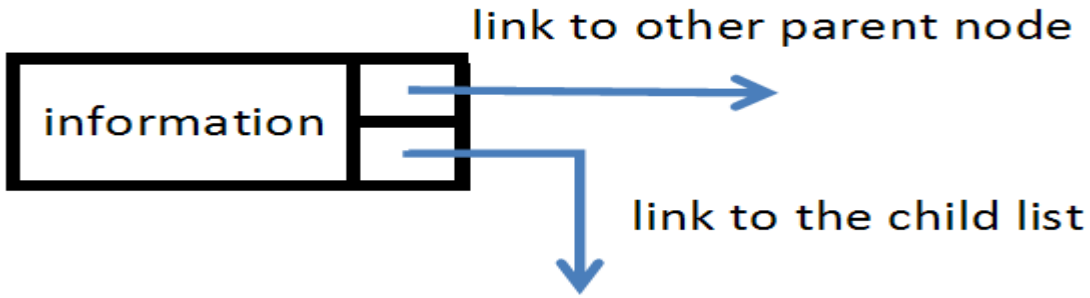
- *Linked List* selain *single Linked-List*, *double Linked List* maupun *circular linked-list* dapat dibentuk menjadi *linked list* lainnya seperti *Multiple Linked List*, *Multilevel Linked List*, maupun *Tree* Dengan operasi-operasi yang khusus diterapkan pada bentuk-bentuk *list* tersebut

3.2 *Multilevel List*

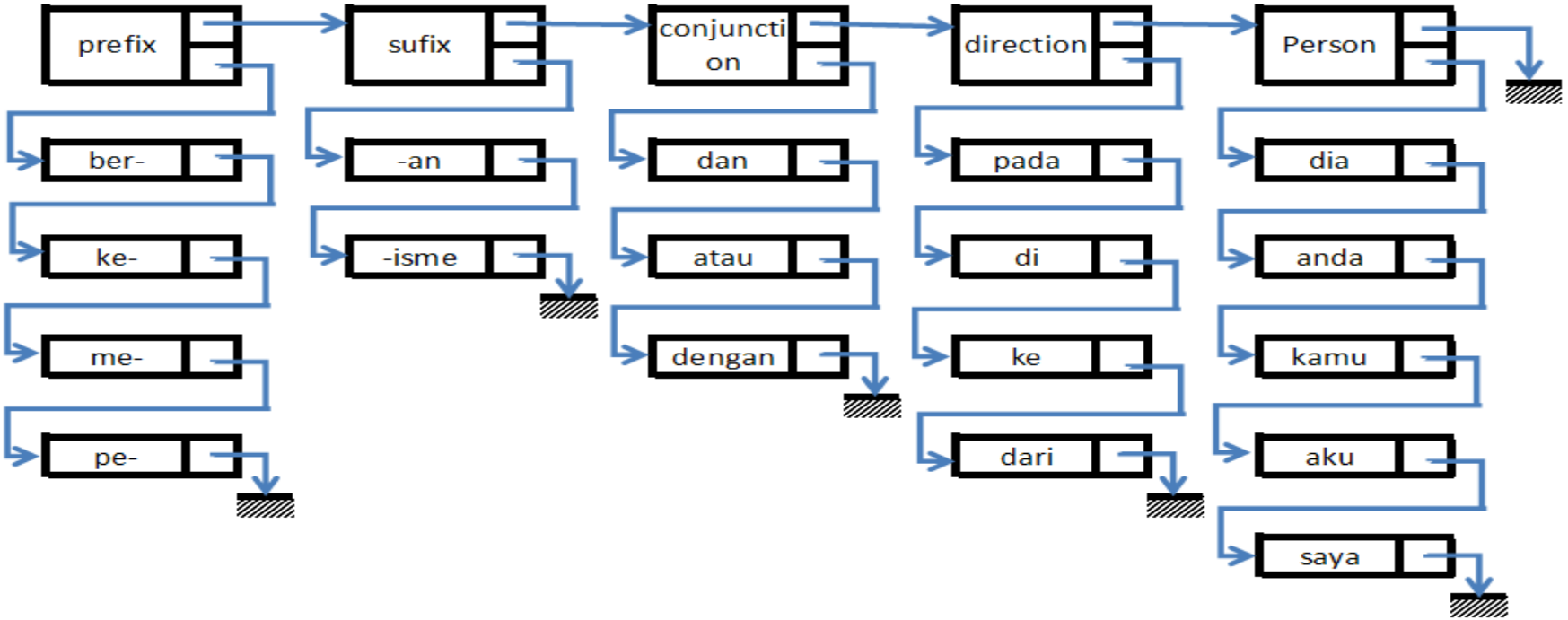
- *List* sebagai *group list* dimana ada node yang menjadi *parent* dari suatu *groups* memiliki *extra link* untuk menunjuk ke list lain sebagai *child list* di samping *link* ke *next group list* node.
- *Element*
  - *Information*
  - *Link* ke node *parent* lain
  - *Link* ke *child list*

3.3 Elemen *Multilevel List* (Lanj.)

Multilevel List Element



3.4 Contoh Multilevel list



3.4 Contoh Multilevel list (Lanj.)

- Contoh pada slide sebelumnya merupakan contoh dari penggunaan *multilevel list* pada aplikasi pemrosesan kata.
- *List* pertama merupakan *list* syntax, *link* kedua merupakan *list vocabulary* pada masing-masing syntax.

Contoh Program Double Linked-List

```
class Simpul:
    def __init__(self, dat):
        self.dat = dat
        self.next = None
        self.prev = None  # Menambah referensi ke simpul sebelumnya

class LinkedList:
    def __init__(self, head=None):
        self.head = head

    def append(self, simpulBaru):
        ptr = self.head
        if ptr:
            while ptr.next:
                ptr = ptr.next
            ptr.next = simpulBaru
            simpulBaru.prev = ptr  # Mengatur simpul sebelumnya
        else:
            self.head = simpulBaru
```

```
def sisip(self, setelah, simpulBaru):
    ptr = self.head
    while ptr and (ptr.dat != setelah):
        ptr = ptr.next
    if ptr.dat == setelah:
        simpulBaru.next = ptr.next
        simpulBaru.prev = ptr  # Mengatur simpul sebelumnya
        ptr.next = simpulBaru

def hapus(self, dihapus):
    ptr = self.head
    while ptr and (ptr.dat != dihapus):
        ptr = ptr.next
    if ptr.dat == dihapus:
        if ptr.prev:
            ptr.prev.next = ptr.next
        else:
            self.head = ptr.next
        if ptr.next:
            ptr.next.prev = ptr.prev
        del ptr
```

Contoh Program Double Linked-List (Lanj.)

```
    def cetak(self):
        print("HEAD: ", self.head)
        ptr = self.head
        while ptr:
            print("node: [", ptr.prev, "|", ptr.dat, "|", ptr.next, "]")
            ptr = ptr.next


a = [34, 77, 91, 23, 10, 32, 90, 60, 50, 11]
print("mulai")
l1 = LinkedList()  # membuat linked-list 1
l2 = LinkedList()  # membuat linked-list 2
l3 = LinkedList()  # membuat linked-list 3
for i in range(0, 10):
    temp = Simpul(a[i])
    l1.append(temp)
l1.sisip(10, Simpul(48))
l1.cetak()
print("ok")
print()
```

```
xptr = l1.head
print(xptr)
while xptr:
    temp = Simpul(xptr.dat)
    if (xptr.dat % 2 == 0):
        l2.append(temp)
    else:
        l3.append(temp)
    xptr = xptr.next
print("\ndata genap")
l2.cetak()
print("\ndata ganjil")
l3.cetak()
l1.hapus(10)
print("\nsetelah hapus 10")
l1.cetak()

l1.hapus(34)
print("\nsetelah hapus head")
print(l1.head)
l1.cetak()

l1.hapus(11)
print("\nsetelah hapus tail")
l1.cetak()
```

Contoh Program Circular Linked-List

```
class Simpul:
    def __init__(self, dat):
        self.dat = dat
        self.next = None
        self.prev = None  # Menambah referensi ke simpul sebelumnya

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def append(self, simpulBaru):
        if not self.head:
            self.head = simpulBaru
            simpulBaru.next = simpulBaru
            simpulBaru.prev = simpulBaru
        else:
            tail = self.head.prev
            tail.next = simpulBaru
            simpulBaru.prev = tail
            simpulBaru.next = self.head
            self.head.prev = simpulBaru
```

```
def sisip(self, setelah, simpulBaru):
    ptr = self.head
    while ptr and (ptr.dat != setelah):
        ptr = ptr.next
    if ptr.dat == setelah:
        simpulBaru.next = ptr.next
        simpulBaru.prev = ptr
        ptr.next.prev = simpulBaru
        ptr.next = simpulBaru

def hapus(self, dihapus):
    ptr = self.head
    while ptr and (ptr.dat != dihapus):
        ptr = ptr.next
    if ptr.dat == dihapus:
        ptr.prev.next = ptr.next
        ptr.next.prev = ptr.prev
        if ptr == self.head:
            self.head = ptr.next
        del ptr
```

Contoh Program Circular Linked-List (Lanj.)

```
def cetak(self):
    if not self.head:
        print("Circular Linked List Kosong")
    else:
        current = self.head
        while True:
            print("node: [", current.prev.dat, "|", current.dat, "|", current.next.dat, "]")
            current = current.next
            if current == self.head:
                break
```

```
a = [34, 77, 91, 23, 10, 32, 90, 60, 50, 11]
print("mulai")
cll = CircularLinkedList()

for i in range(0, 10):
    temp = Simpul(a[i])
    cll.append(temp)

cll.sisip(10, Simpul(48))
cll.cetak()
print("ok")
print()

# ... (bagian lain dari kode tetap sama)

# Mengganti variabel l1, l2, l3 dengan cll
xptr = cll.head
# ... (bagian lain dari kode tetap sama)
```

Ringkasan

- Penggunaan *Double Linked List* dapat mengurangi pemakaian variabel yang diperlukan pada proses tambah, hapus dan pindah node.
- Proses hapus, pindah dan tambah node pada *double linked-list* jangan sampai melupakan proses untuk memindahkan *previous link* dari node-node yang dipindahkan, dan node-node yang berhubungan pemindahan tersebut.
- Detail proses penghapusan, penambahan, pencarian dan pemindahan node dapat dilihat pada masing-masing *slide*, proses tersebut hanya salah satu contoh proses saja, banyak variasi proses yang lain.

PERINGATAN HAK CIPTA

PERINGATAN HAK CIPTA

Terimakasih

TUHAN Memberkati Anda

Teady Matius Surya Mulyana (tmulyana@bundamulia.ac.id)