TIB29 – Struktur Data dan Algoritma

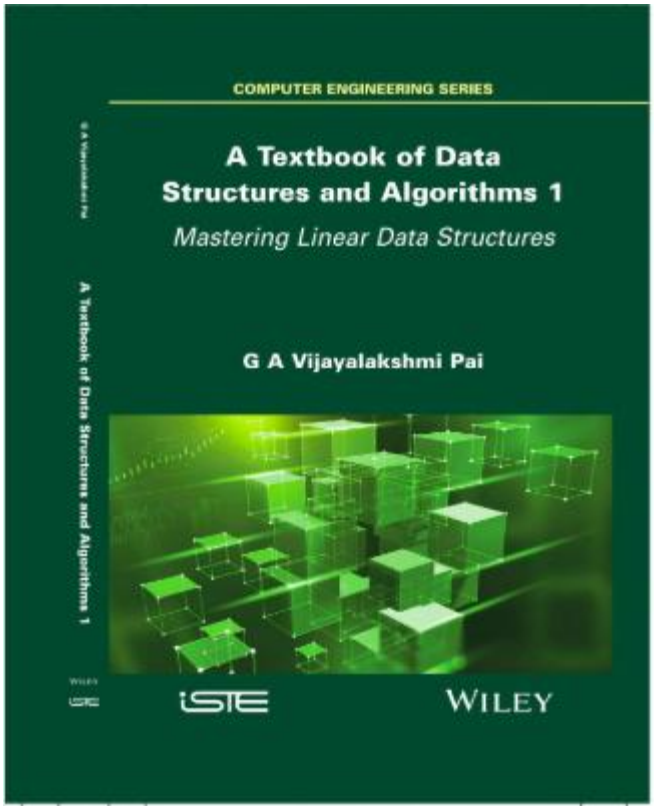
PERINGATAN HAK CIPTA

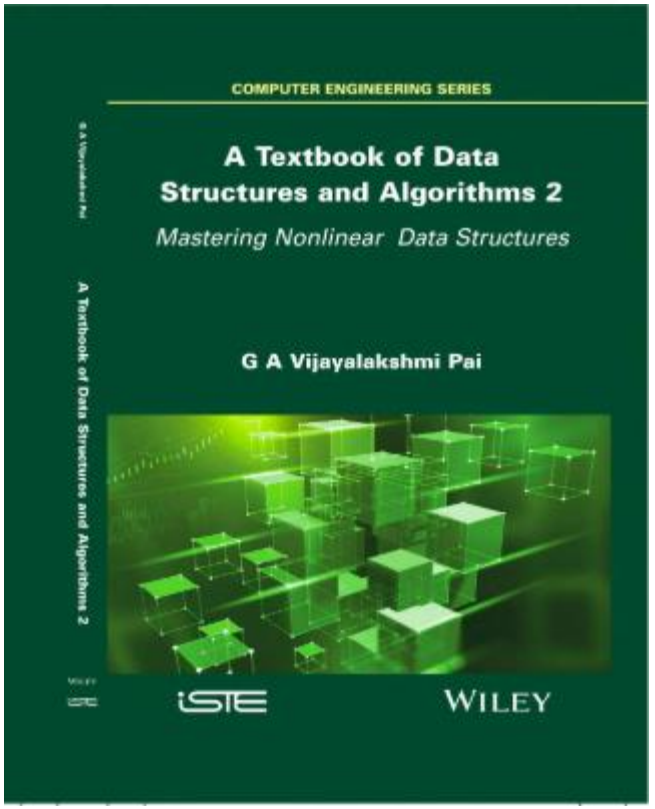
PERINGATAN HAK CIPTA

ubm
Universitas Bunda Mulia

Linked-List

Diadopsi Dari Sumber:





Sub-CPMK

- Mahasiswa mampu membuat single Linked-List berikut operasi-operasinya. (C3, A3)

Materi

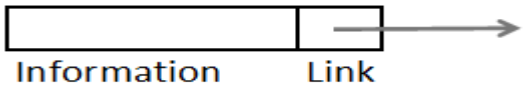
1. Konsep Linked-List
2. Pointer
3. Membuat Linked-List
4. Operasi-operasi Linked-List

1. Konsep *Linked-List*

1.1. *Linked List*

- *Linked-List* atau Senarai adalah sebuah urutan elemen yang terbatas yang diakses menggunakan *pointer*

  s1, s2, ..., sn
- Node/simpul : *Record* yang berisi informasi dan link ke node/simpul lainnya
- *Linked List elements*
  - *Information*
  - link: Penghubung ke node/simpul lain



1.2. Dua *Variable* Penting *Linked-list*

- *Head*/Kepala: variabel yang berisi informasi *address pointer* dari node/simpul pertama
- *CurrentCell* / *PointerCell*: berisi informasi dari *current* node/simpul yang sedang diakses

1.2. Dua *Variable* Penting *Linked-list* (Lanj.)

HARUS DIINGAT!!!

- *Head*/Kepala adalah variabel berisi informasi penting untuk mengarahkan *linked-list*
- Dengan '*Head*' atau 'Kepala' kita dapat menuju ke node/simpul pertama dan bergerak maju ke simpul/node tujuan
- Pada saat anda kehilangan '*Head*' berarti anda kehilangan *linked list* juga
- ***Never ever lose your 'HEAD'!!!***

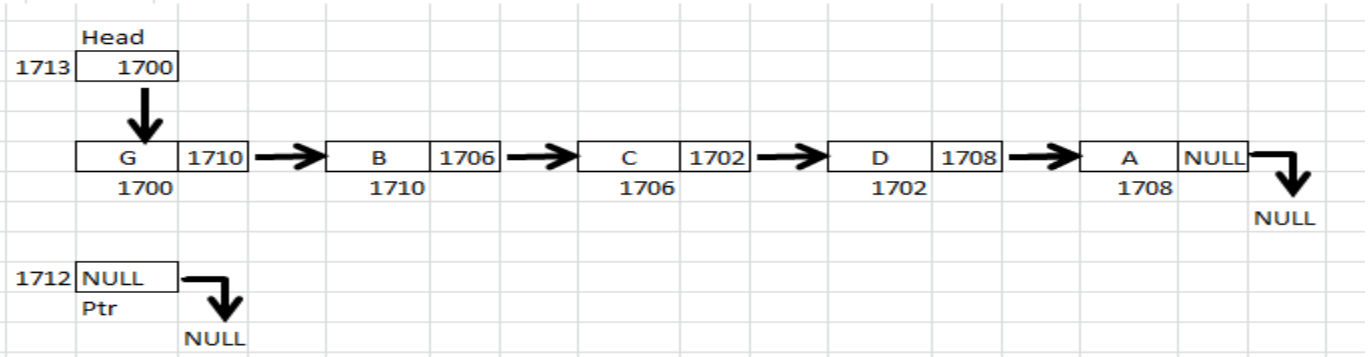
1.3. Beberapa *Variants Linked List* yang Umum

- *Single Linked List*
- *Double Linked List*
- *Circular Linked List*
- *Multilevel List*

1.3. Beberapa *Variants Linked List* yang Umum (Lanj.)

- Tiap simpul memilik alamat memory, link next yang tersimpan berisi alamat memory dari simpul berikutnya

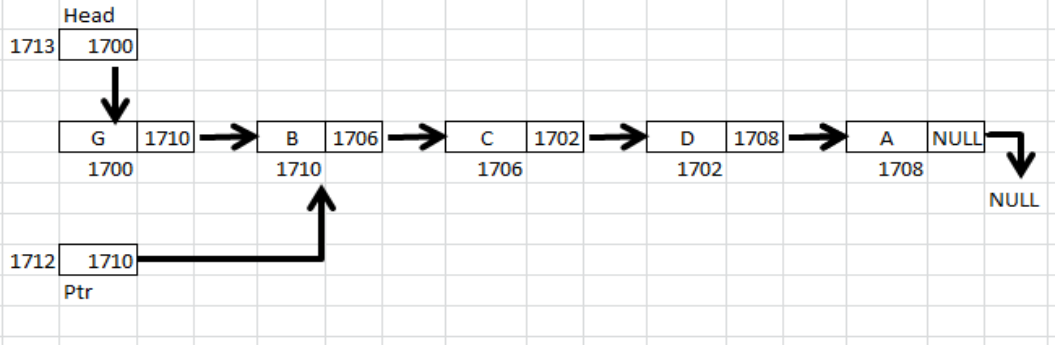
	alamat memory	Isi Memory	
Head	1713	1700	← Kepala Linked List
Ptr	1712	NULL	← Current Node
	1711	1706	
	1710	B	
	1709	NULL	
	1708	A	
	1707	1702	
	1706	C	
	1705		
	1704		
	1703	1708	
	1702	D	
	1701	1710	
	1700	G	
	1699		
	1698		
	1697		



1.3. Beberapa *Variants Linked List* yang Umum (Lanj.)

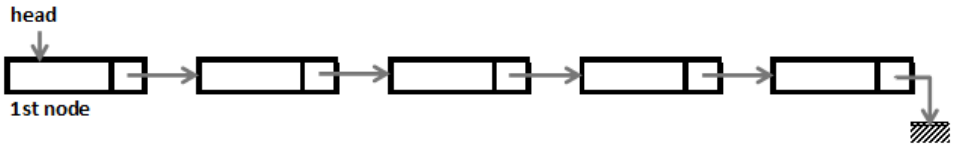
	alamat memory	Isi Memory	
Head	1713	1700	← Kepala Linked List
Ptr	1712	1700	← Current Node
	1711	1706	
	1710	B	
	1709	NULL	
	1708	A	
	1707	1702	
	1706	C	
	1705		
	1704		
	1703	1708	
	1702	D	
	1701	1710	
	1700	G	
	1699		
	1698		

```
Ptr = Head;
Ptr = Ptr.Next
```



- Untuk melakukan penulusuran linked-list, pointer ptr dapat diarahkan ke ke Head dahulu
- Kemudian untuk berpindah simpul, dapat dilakukan dengan mengisi ptr dengan isi dari next link simpul tsb, sehingga ptr dapat menunjuk ke simpul berikutnya (ptr=ptr->Next

1.4. *Single Linked List*



- Contoh Single Linked List,
- tiap simpul memilik alamat memory, link next yang tersimpan berisi alamat memory dari simpul berikutnya
- Head selalu menunjuk ke simpul pertama

2. *Pointer*

Catatan

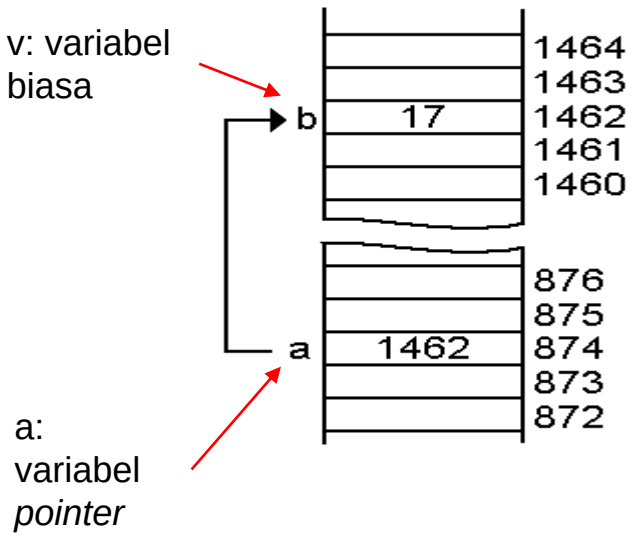
- Materi ini diperuntukkan bagi pengguna C atau pascal.
- Bagi pengguna Phyton hanya untuk informasi saja, berikut contoh membaca alamat memory dan menampilkan isi sel memory, karena nantinya untuk linked-list, phyton akan mempergunakan class dan objek.

2.1. *Pointer*

- *Pointer* adalah variabel yang berisi alamat dari suatu *memory* yang menyimpan data.

2.2. Contoh *Pointer*

- Variabel *Pointer a* menunjuk ke alamat variabel *b*.
- variabel *b* menyimpan sebuah bilangan (17)
- Variabel *a* dapat menyimpan alamat dari variabel *b* pada *memory* (1462)



2.3. Alamat *Memory* dan Variabel *Pointer*

- Untuk mengakses data pada alamat *memory* dapat dilakukan hanya dengan mengetahui alamatnya.
- Alamat data tersebut harus tersimpan pada suatu variabel agar dapat diakses melalui variabel yang menyimpan alamat sel *memory* tersebut.

2.4. Variabel Dinamis

- Ketika suatu variabel dideklarasikan, maka sistem operasi akan mencari suatu alamat di *memory* dan alamat serialnya yang mampu menampung panjang data yang dideklarasikan pada variabel tersebut.
- Demikian juga pada deklarasi suatu variabel *pointer*, maka sistem operasi akan mencari alamat *memory* yang mampu menampung panjang data dengan tipe data *pointer.*

2.5. Alokasi *Memory*

- Alokasi *memory* dimaksudkan untuk mencari dan memesan suatu alamat *memory* dengan sel serialnya untuk didaftarkan menyimpan data dengan tipe data yang dimaksudkan untuk ditunjuk oleh suatu variabel pointer.
- Penggunaan tipe data *pointer* dimaksudkan tidak untuk tujuan menghemat *memory*, karena selain mendeklarasikan variabel bertipe *pointer*, maka program juga perlu mengalokasikan alamat pada *memory* yang mampu menampung tipe data dari data yang sebenarnya hendak disimpan.

2.6 Variabel Pointer

- Variabel *pointer* dimaksudkan menyimpan alamat awal dari suatu *memory* yang berisi sel-sel *memory* yang dialokasikan untuk menyimpan data sesuai dengan tipe data yang dialokasikan.

2.7. Deklarasi Variabel bertipe *Pointer*

- Variabel *Pointer* dapat dideklarasikan dengan mencantumkan karakter * di depan variabel yang dideklarasikan
- Contoh:

  int *a;
- Deklarasi ini akan mendaftarkan variabel a untuk menyimpan data bertipe *pointer* yang ditujukan untuk menunjuk ke alamat yang berisi data int

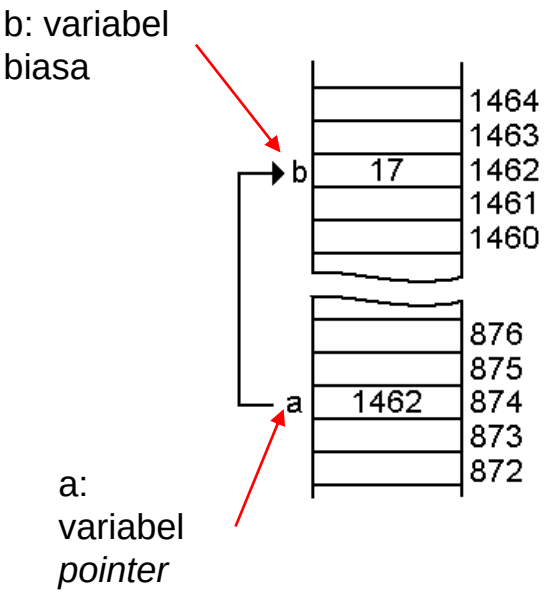
2.8 Menyimpan Alamat *Memory* (C dan Pascal)

- Kembali ke contoh di awal
- Variabel a merupakan variabel *pointer* yang akan menyimpan alamat *memory* dari variabel b
- Karena itu a di deklarasikan dengan perintah:

  int *a;
- Sedangan b dideklarasikan dengan perintah

  int b
- Sedangan b dideklarasikan dengan perintah

  int b



- Alamat *memory* dari variabel b yaitu 1462 dapat disimpan pada variabel a, dengan mencantumkan karakter &didepan variabel b.

  a = &b;

2.9 Alokasi *Memory*

- Pemilihan alamat *memory* yang menyimpan data yang sebenarnya dapat juga dilakukan melalui data yang bertipe *pointer* dengan perintah

```
a = (tipeData*)malloc(sizeof(tipeData));
```

- Contoh

```
int *a
a = (int *)malloc(sizeof(int));
```

2.10 Variabel Pointer Pada Phyton

- Materi ini hanya sebagai pengenalan alamat memory dan data nya dalam phyton, nantinya linked-list phyton akan mempergunakan class dan object.
- Pada phyton kita dapat mempergunakan modul ctypes untuk membaca isi sel memory pada alamat yang ditunjuk

Contoh Variabel Pointer pada Phyton

```
import ctypes

a = 10
print("alamat variabel a dalam desimal: ",id(a))
print("alamat variabel a dalam hexadesimal: ",hex(id(a)))
alamat = id(a)

dat = ctypes.cast(alamat, ctypes.py_object).value
print("isi alamat [{}] adalah {}".format(hex(alamat),dat))
```

```
alamat variabel a dalam desimal:  134861965558288
alamat variabel a dalam hexadesimal:  0x7aa7ff8d0210
isi alamat [0x7aa7ff8d0210] adalah 10
```

3. Membuat Linked-List

3.1 Membuat Linked-List

Untuk membuat Linked-List, diperlukan

- satu variable untuk menyimpan alamat Record yang ditunjuk sebagai KEPALA/HEAD
- Satu variable untuk menyimpan alamat Record yang sedang diakses
- Jika diperlukan kita dapat menambahkan variable lainnya yang diperlukan untuk menyimpan alamat Record baru, Record temporary, ataupun Record yang menunjuk ke TAIL/EKOR dari Linked-List

3.1 Membuat Linked-List (Lanj.)

- Pengunaan *Pointer* pada *Linked-list* merupakan suatu hal yang biasa dilakukan.
- Penggunaan ini memerlukan *define* suatu *struct/Record*, yang mana *struct/record* ini diperlakukan sebagai mana layaknya tipe data.
- Struct tersebut digunakan untuk Deklarasi variabel *pointer* yang menyimpan alamat awal data yang sesuai dengan struktur *record* tersebut pada suatu *memory.*

2.11.1 *Pointer - Pascal*

- *Record definition*

```
Type
   RecordName = Record
        VarName : vartype;
        NextPointer : ^RecordName;
   End;
   PointerName = ^RecordName;
```

- *Pointer declaration*

```
Var
   PointerHead : PointerName; PointerCell :
   PointerName;
```

- *Example*

```
Type
   OneCell = Record
        Data1 : char;
        Data2 : Integer;
        Data3 : string;
        Next : ^onesel;
   End;
   Ptr = ^ OneCell;
Var
   Head, P : Ptr;
```

2.11.1 *Pointer – Pascal* (Lanj.)

- *Assignment*

```
PointerCell^.VarName := value;
PointerCell^.NextPointer^.VarName := value;
```

- *Accessing*

```
PointerCell^.varname
PointerCell^.NextPointer^.VarName
PointerCell^.NextPointer^.NextPointer^.VarName
```

- Inisialisasi/Membentuk *Pointer* Baru

```
new(PointerCell);
```

2.11.2 *Pascal Record Definition With Two Links*

- *Record definition*

```
Type
  RecordName = Record
     VarName : vartype;
     PreviousPtr :
  ^RecordName;
     NextPtr :
  ^RecordName;
  End;
  PointerName =
  ^RecordName;
```

- *Pointer declaration*

```
Var
  PointerHead :
  PointerName; PointerCell
  : PointerName;
```

- *Example*

```
Type
  OneCell = Record
     Data1 : char;
     Data2 : Integer;
     Data3 : string;
     Prev : ^onesel;
     Next : ^onesel;
  End;
  Ptr = ^ OneCell;
Var
  Head, P : Ptr;
```

2.11.3 *Pointer* - C

- *Record definition*

```
struct StructName
{
     vartype VarName;
     struct StructName
  *NextPtr;
};
```

- *Pointer declaration*

```
struct StructName
  *PtrHead
struct StructName
  *PtrCell;
```

- *Example*

```
struct OneCell
{
     char Data1;
     int  Data2;
     char Data3[50];
     struct OneCell Next;
};


Int main(void)
{
  struct OneCell *Head,
  *Ptr;
  .
  .
}
```

2.11.3 *Pointer* – C (Lanj.)

- *Assignment*

```
PtrCell->VarName = value;
PtrCell->NextPtr->VarName = value;
PtrCell->NextPtr = NULL;
```

2.11.3 *Pointer* – C (Lanj.)

- *Accessing*

```
PtrCell->VarName
(*PtrCell).VarName
PtrCell->NextPtr->VarName
(*(*PtrCell).NextPtr).VarName
PtrCell->NextPtr->NextPtr->VarName
(*(*(*PtrCell).NextPtr).NextPtr).VarName
```

2.11.3 *Pointer* – C (Lanj.)

- Inisialisasi/membentuk *Pointer* Baru

  Menggunakan malloc

```
PtrCell=(struct StructName
*)malloc(sizeof(struct StructName));
```

  Menggunakan new

```
PtrCell=new StructName;
```

2.11.3. *Pointer* – C (Lanj.)

Note:

- *Assign pointer address to a* PtrCell

```
PtrCell=VarName; //located at stack or
  global variable
```

- *Pointer attribute*

```
Pointer->Variable //Field may writen as
  (*Pointer).VariableField
```

- *Statement* `new` *define at* new.h *header file*
- new.h *header file can be used only at* C++

2.11.4 *C Structure definition* Dengan Dua Links

- *Record definition*

```
struct StructName
{
  vartype VarName;
  struct StructName
  *PrevPtr;
  struct StructName
  *NextPtr;
};
```

- *Example*

```
struct OneCell
{
      char Data1;
      int  Data2;
      char Data3[50];
      struct OneCell Prev;
      struct OneCell Next;
};


Int main(void)
{
  struct OneCell *Head,
  *Ptr;
  .
  .
}
```

2.11.5 Pointer - Phyton

```
class Simpul:
  def __init__(self, nama, nim, nilai):
    self.nama = nama
    self.nama = nim
    self.nama = nilai
    self.next = None

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
    else:
      self.head = simpulBaru

l1 = LinkedList()
for i in range(0,10):
  temp = Simpul(a[i])
  l1.append(temp)
```

3.6. *Record* Dengan Satu Link

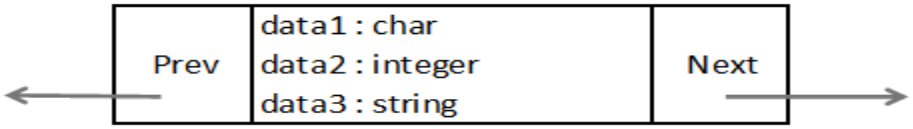
```
struct OneCell
{
  char Data1;
  int  Data2;
  char Data3[50];
  struct OneCell Next;
};
```

```
Type
  OneCell = Record
        Data1 : char;
        Data2 : Integer;
        Data3 : string;
        Next : ^onesel;
  End;
  Ptr = ^ OneCell;
```

3.7. *Record* Dengan Dua Link

```
struct OneCell
{
  char Data1;
  int  Data2;
  char Data3[50];
  struct OneCell Prev;
  struct OneCell Next;
};
```

```
Type
  OneCell = Record
        Data1 : char;
        Data2 : Integer;
        Data3 : string;
        Prev : ^onesel;
        Next : ^onesel;
  End;
  Ptr = ^ OneCell;
```



Single Linked-List pada Python

Pada Python, Single Linked-List dapat dibentuk dengan menggunakan kelas(*class*) dan objek

Kelas (*Class*)

•Kelas adalah suatu blueprint atau template untuk membuat objek. Ini menyediakan struktur atau cetakan dasar yang mendefinisikan properti (atribut) dan perilaku (metode) yang dimiliki oleh objek.
•Kelas digunakan untuk merepresentasikan entitas atau konsep tertentu dalam program

Objek

•Objek adalah instansi konkret dari suatu kelas. Dengan kata lain, objek adalah variabel yang dibuat dari kelas tertentu.
•Setiap objek memiliki atribut dan perilaku yang didefinisikan oleh kelasnya sendiri. Objek dapat dianggap sebagai "instance" atau "realisasi" dari suatu konsep atau entitas yang diwakili oleh kelas.

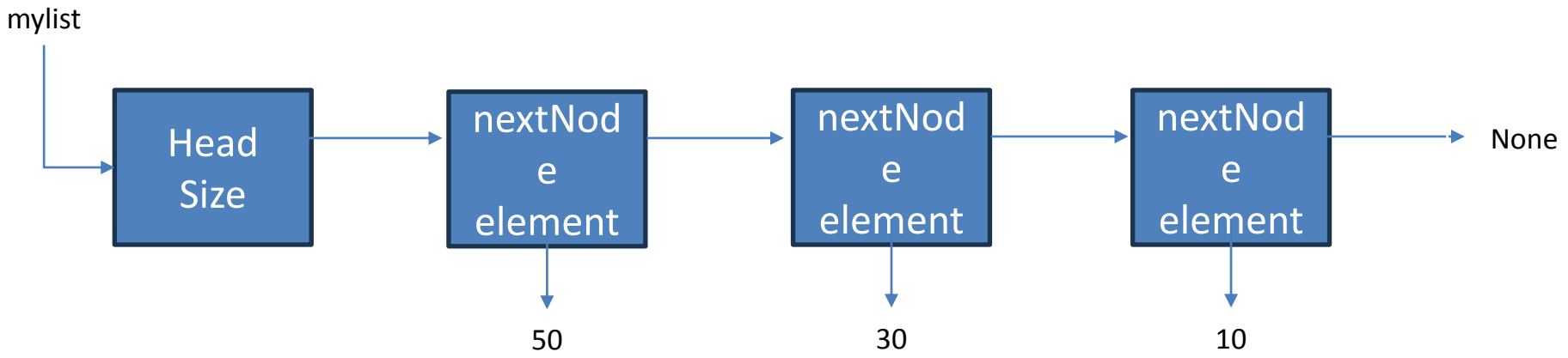
Contoh Program Pembuatan Kelas dan Objek

```python
# Mendefinisikan kelas
class Kendaraan:
    def __init__(self, jenis, roda):
        self.jenis = jenis
        self.roda = roda

    def info(self):
        print(f"Jenis: {self.jenis}, Roda:
{self.roda}")

# Membuat objek dari kelas Kendaraan
mobil = Kendaraan(jenis="Mobil", roda=4)
sepeda = Kendaraan(jenis="Sepeda", roda=2)

# Mengakses atribut dan metode objek
print(mobil.jenis)  # Output: Mobil
sepeda.info()       # Output: Jenis: Sepeda, Roda: 2
```

Penjelasan

- Kendaraan adalah kelas yang memiliki atribut ‘jenis’ dan ‘roda’, serta metode ‘info’
- Objek ‘mobil’ dan ‘sepeda’ adalah dua instansi dari kelas ‘Kendaraan’. Objek-objek ini memiliki karakteristik yang berbeda sesuai dengan nilai atribut yang diberikan pada saat pembuatan.

Contoh Pembuatan Single Linked List



Visualisasi dari Single Linked List yang akan dibuat

Pembuatan Single Linked List

```
#Pembuatan struktur Single Linked List
dan Node
class singleLL:
  #Pembuatan kelas Node
  class _Node:
    def __init__(self,element,nextNode =
None):
      self.elemet = elemet
      self.nextNode = nextNode

  #Pembuatan Single Linked List
  def __init__(self):
    self.head = None
    self._size=0
```

Pembuatan kelas Node ada di dalam kelas singleLL karna kelas Node hanya akan dibutuhkan di dalam kelas single LL

Pembuatan Single Linked List View All & Add First

```
#Pembuatan struktur Single Linked List dan Node
class singleLL:
  #Pembuatan kelas Node
  class _Node:
    def __init__(self,element,nextNode = None):
      self.element = element
      self.nextNode = nextNode
  #Pembuatan Single Linked List
  def __init__(self):
    self._head = None
    self._size=0
#Pembuatan Fungsi View All
  def __str__(self):
    result = ''
    pointer = self._head
    while pointer != None:
      result = result + str(pointer.element) + ' '
      pointer = pointer.nextNode
    return result
```

```
#Add First pada List yang masih kosong
  def add_first(self, element):
    newNode = self._Node(element)
    newNode.nextNode = self._head
    self._head = newNode
    self._size += 1
  def __len__(self):
    return self._size
def main():
  mylist = singleLL()
  mylist.add_first(50)
  mylist.add_first(30)
  mylist.add_first(10)
  print(str(mylist))
  print(len(mylist))
main()
```

Note:
*Add First digunakan untuk menambahkan elemen baru pada single linked list
Terdapat 2 jenis Add First:
1. Add First ketika list masih kosong
2. 2. Add First ketika list sudah ada
*Pada kasus ini kita membuat Add First untuk list yang masuk kosong

4. Operasi-operasi Linked-List

4.1 Operasi-operasi *Linked List*

Operasi-operasi linked List

- *Mencari Simpul / Search / Locate*
- *Menambah Simpul / Insert List*
- *Menghapus Simpul Delete*

Kemungkinan operasi dapat terjadi pada:

- bagian depan dari *list*
- tengah *list*
- bagian akhir dari *list*

4.2 *Locate Operation*

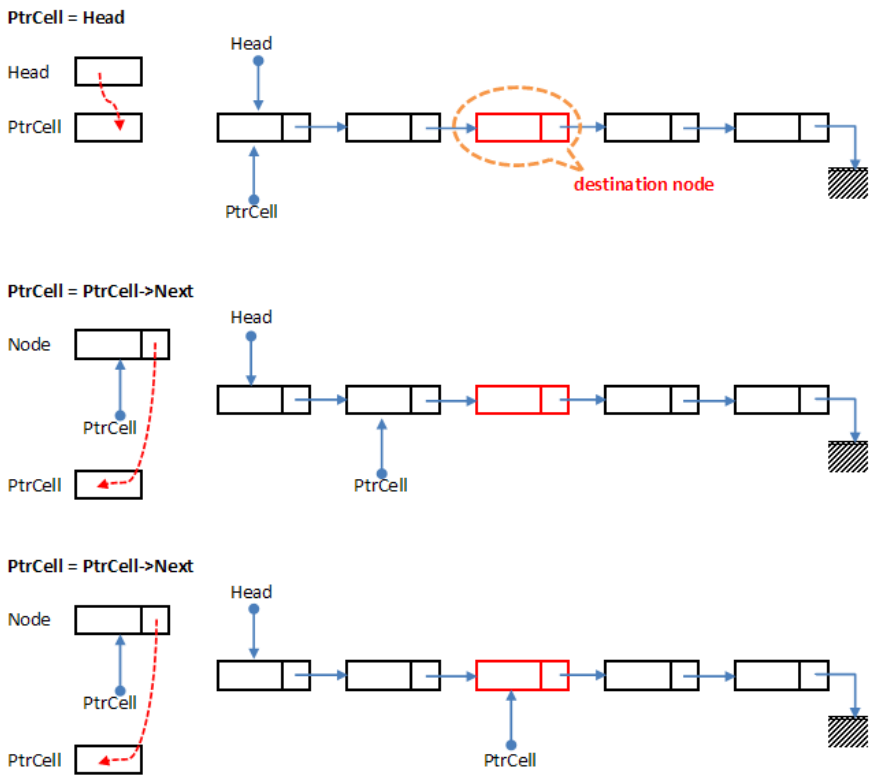
- *Assign PointerCell* sebagai *Head*

```
PointerCell = Head;
```

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

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

4.2 *Locate Operation (lanjut)*



- Arahkan Ptr ke Head (isi variabel Ptr dengan Head)

  PtrCell = Head
- Copykan isi next dari simpul yang di tunjuk oleh variabel Ptr ke variabel Ptr

  PtrCell = PtrCell->Next
- Ulangi sampai menemukan simpul yang dituju

4.2 *Locate Operation (lanjut)*

Contoh Program

```
PtrCell=Head;
While ((PtrCell->dat != datCari) &&
  (PtrCell->Next != NULL))
{
   PtrCell = PtrCell->Next;
}
```

4.2 *Insert Operation*

- Pada bagian depan *list*

  Hanya dapat terjadi pada operasi *insert* sebelum *current cell*
- Pada bagian akhir *list*

  Hanya dapat terjadi pada *insert* setelah *current cell*
- Pada bagian tengah *middle list*

Catatan: Gambar ilustrasi masing-masing operasi dapat dilihat pada penjelasan slide berikutnya

4.2.1 *Insert* Pada Bagian Depan *List*

1. Membuat Simpul Baru:

Saat melakukan operasi insert pada bagian depan, kita pertama-tama membuat simpul baru yang akan ditambahkan ke linked list.

2. Inisialisasi Simpul Baru:

Simpul baru ini diinisialisasi dengan data yang ingin ditambahkan dan referensi next yang menunjuk ke simpul yang saat ini menjadi kepala (head) linked list.

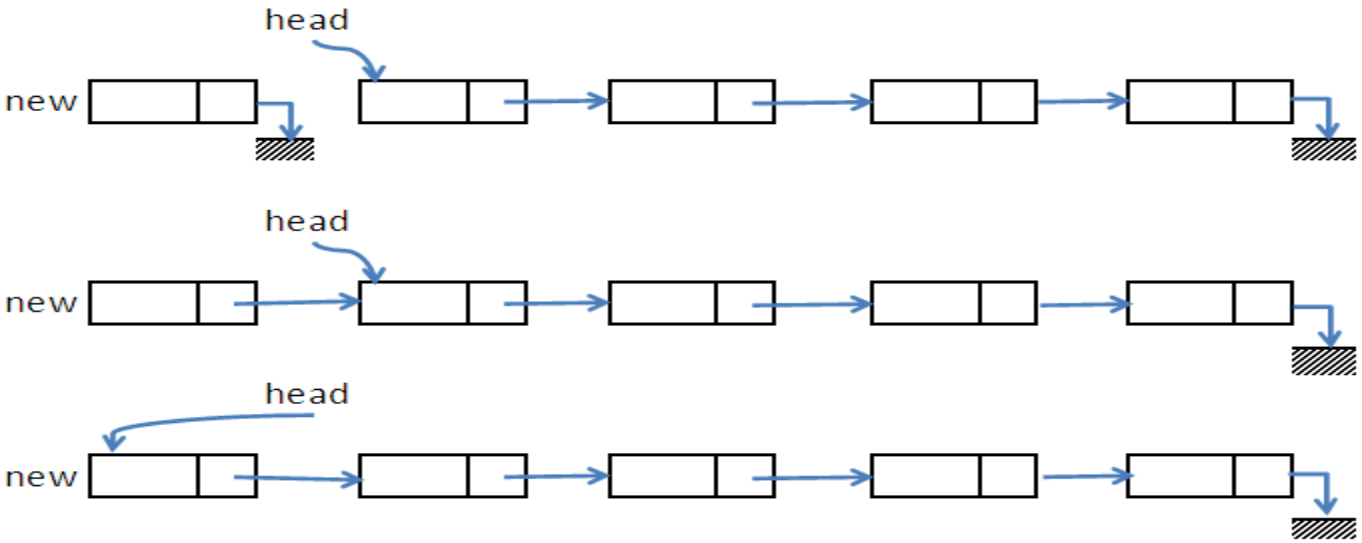
3. Menyesuaikan Pointer Head:

Setelah simpul baru dibuat, kita mengatur referensi kepala (head) linked list agar menunjuk ke simpul baru ini. Dengan kata lain, simpul baru sekarang menjadi elemen pertama dalam linked list.

4. Penyesuaian Ukuran Linked List:

Terakhir, kita bisa menyesuaikan ukuran linked list (jika diperlukan) dengan menambahkan 1 pada nilai ukuran linked list.

4.2.1 *Insert* Pada Bagian Depan *List* (Lanj.)



4.2.2 Insert di Tengah

1. Membuat Simpul Baru:

Saat melakukan operasi insert pada bagian tengah, kita membuat simpul baru yang akan ditambahkan ke linked list.

2. Menentukan Posisi Insert:

Menentukan posisi (indeks) di mana kita ingin menambahkan simpul baru. Posisi ini tidak boleh melebihi ukuran (length) linked list.

3. Iterasi ke Posisi Sebelum:

Iterasi dari kepala linked list hingga posisi sebelum simpul yang akan di-insert. Ini dilakukan untuk mendapatkan simpul sebelum posisi yang ditargetkan.

4.2.2 Insert di Tengah (Lanj.)

4. Inisialisasi Simpul Baru:

Simpul baru diinisialisasi dengan data yang ingin ditambahkan dan referensi next yang menunjuk ke simpul setelah posisi sebelumnya.

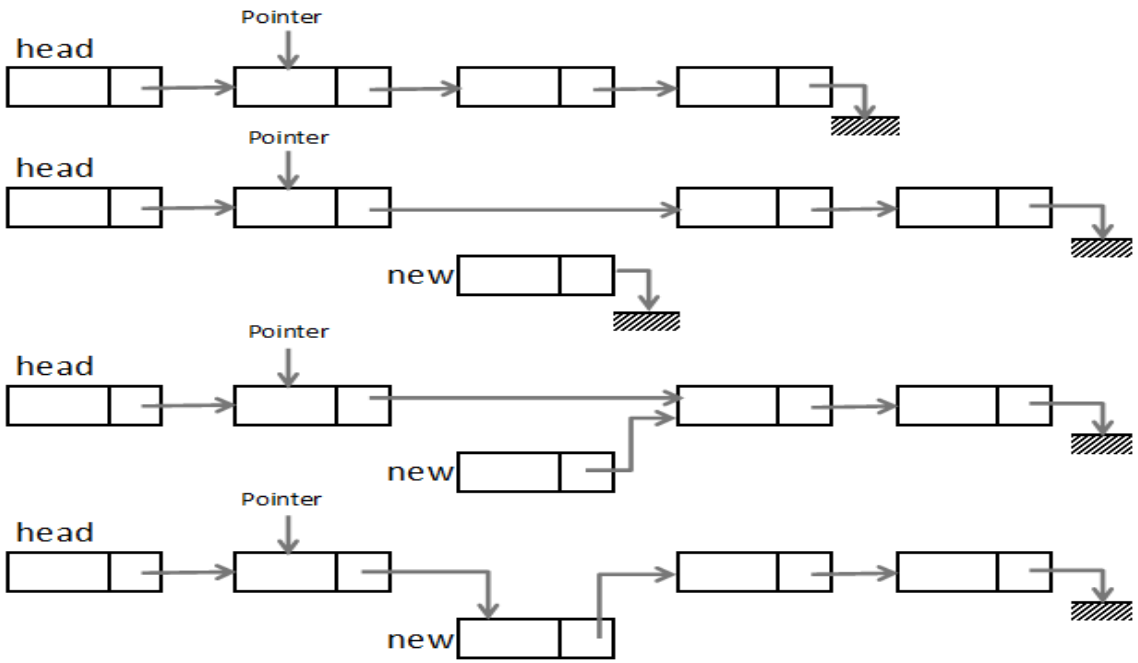
5. Menghubungkan Simpul Baru:

Referensi next pada simpul sebelumnya diatur untuk menunjuk ke simpul baru, sehingga simpul baru tersambung di bagian tengah linked list.

6. Penyesuaian Ukuran Linked List:

Jika diperlukan, kita bisa menyesuaikan ukuran linked list dengan menambahkan 1 pada nilai ukuran linked list.

4.2.2 Insert di Tengah (Lanj.)



4.2.2 Insert di Tengah (Lanj.)

```
class singleLL:
    # Pembuatan kelas Node
    class _Node:
        def __init__(self, element, nextNode=None)
            self.element = element
            self.nextNode = nextNode

    # Pembuatan Single Linked List
    def __init__(self):
        self._head = None
        self._size = 0

    # Pembuatan Fungsi View All
    def __str__(self):
        result = ''
        pointer = self._head
        while pointer != None:
            result = result + str(pointer.element)
            pointer = pointer.nextNode
        return result
```

```
# Add First pada List yang masih kosong
def add_first(self, element):
    newNode = self._Node(element)
    newNode.nextNode = self._head
    self._head = newNode
    self._size += 1

# Menambahkan Node di Tengah Linked List
def add_middle(self, index, element):
    if index < 0 or index > self._size:
        raise IndexError("Index out of range"
    if index == 0:
        self.add_first(element)
    else:
        newNode = self._Node(element)
        current_node = self._head
        for _ in range(index - 1):
            current_node = current_node.nextN
        newNode.nextNode = current_node.nextN
        current_node.nextNode = newNode
        self._size += 1
```

```
    def __len__(self):
        return self._size


def main():
    mylist = singleLL()
    mylist.add_first(50)
    mylist.add_first(30)
    mylist.add_first(10)

    print("Original Linked List:")
    print(str(mylist))
    print(len(mylist))

    # Menambahkan node di tengah (indeks 1)
    mylist.add_middle(1, 20)

    print("\nLinked List setelah menambahkan node di tengah:")
    print(str(mylist))
    print(len(mylist))
```

Output

```
Original Linked List:
10 30 50
3

Linked List setelah menambahkan node di tengah:
10 20 30 50
4
```

4.3 *Insert* Pada Bagian Akhir *List*

1. **Membuat Simpul Baru:**

Saat melakukan operasi insert pada bagian akhir, kita membuat simpul baru yang akan ditambahkan ke linked list.

2. **Inisialisasi Simpul Baru:**

Simpul baru diinisialisasi dengan data yang ingin ditambahkan dan nilai next yang secara awal diatur menjadi **None**, karena simpul baru akan menjadi elemen terakhir.

3. **Iterasi ke Akhir Linked List:**

Iterasi dari kepala linked list hingga mencapai elemen terakhir (simpul yang next-nya **None**).

4.3 *Insert* Pada Bagian Akhir *List (Lanj.)*

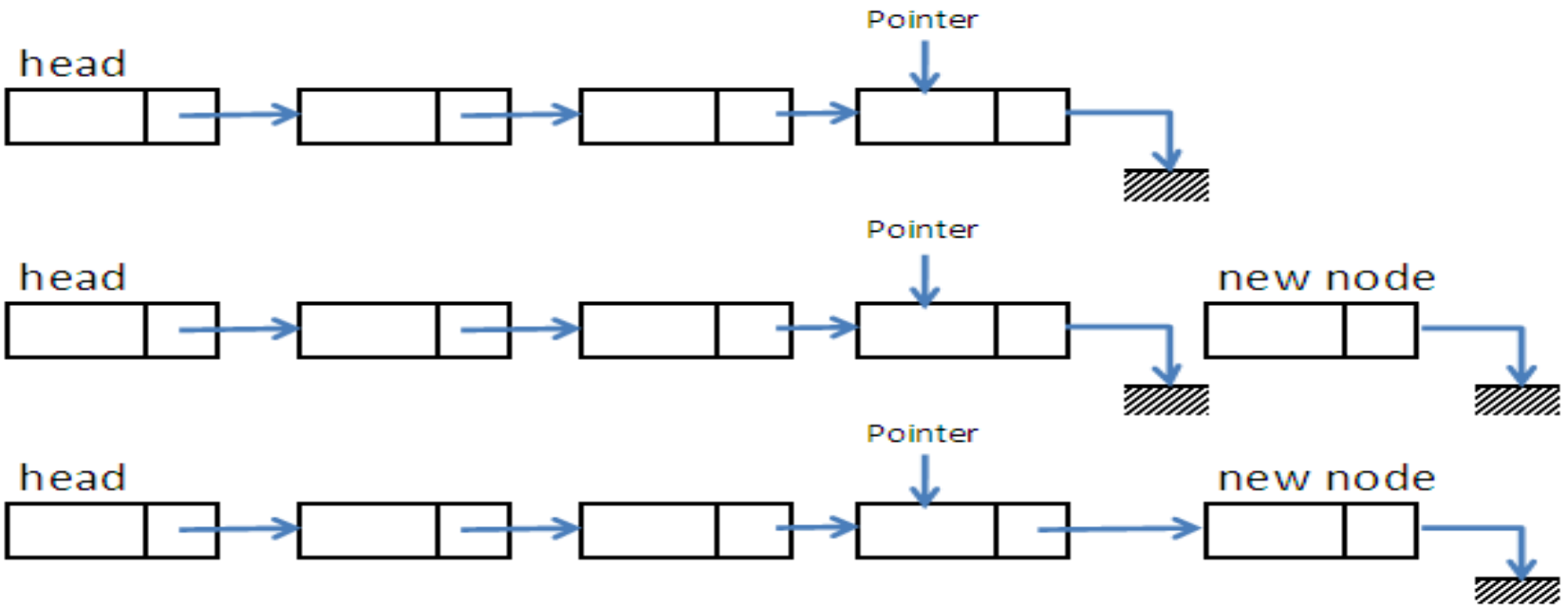
4. **Menghubungkan Simpul Baru:**

Setelah mencapai elemen terakhir, mengatur referensi next pada simpul terakhir untuk menunjuk ke simpul baru, sehingga simpul baru terhubung di bagian akhir linked list.

5. **Penyesuaian Ukuran Linked List:**

Jika diperlukan, kita dapat menyesuaikan ukuran linked list dengan menambahkan 1 pada nilai ukuran linked list.

4.3 *Insert* Pada Bagian Akhir *List* (Lanj.)



4.1. *Delete Operation*

- Pada bagian depan / *delete head* (*WARNING!!!: don't lose the head!*)
- Pada bagian tengah
- Pada bagian akhir / *delete tail*

4.2. *Delete Head*

1. Pengecekan Kepala Tidak Kosong:

Sebelum menghapus elemen pertama, kita perlu memeriksa apakah linked list tidak kosong. Jika linked list kosong, operasi delete head tidak dapat dilakukan.

2. Penyesuaian Referensi Kepala:

Jika linked list tidak kosong, kita akan mengubah referensi kepala (head) linked list agar menunjuk ke simpul kedua. Dengan kata lain, simpul pertama dihapus dari linked list.

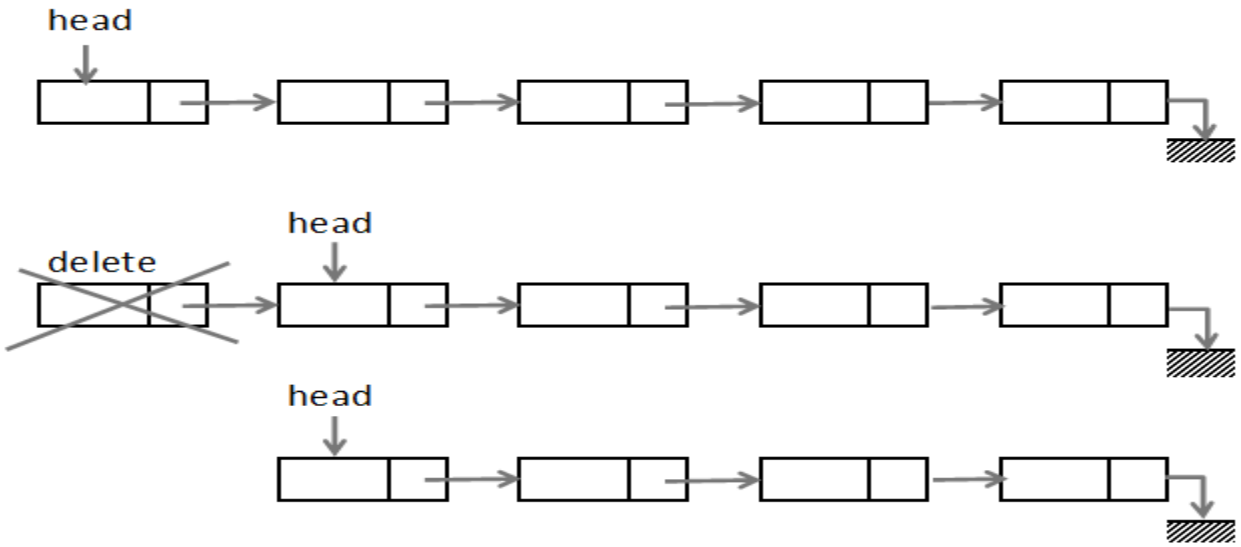
3. Penghapusan Simpul Pertama:

Simpul pertama yang semula menjadi elemen pertama linked list dihapus. Python akan secara otomatis mengurus pengelolaan memori untuk simpul yang tidak lagi terpakai.

4. Penyesuaian Ukuran Linked List:

Jika diperlukan, kita dapat menyesuaikan ukuran linked list dengan mengurangkan 1 dari nilai ukuran linked list

4.2. *Delete Head* (Lanj.)



4.3. *Delete* di Tengah

1. Menentukan Posisi Hapus:

Menentukan posisi (indeks) di mana kita ingin menghapus elemen. Posisi ini tidak boleh melebihi ukuran (length) linked list.

2. Iterasi ke Posisi Sebelum:

Iterasi dari kepala linked list hingga posisi sebelum simpul yang akan dihapus. Ini dilakukan untuk mendapatkan simpul sebelum posisi yang ditargetkan.

3. Penyesuaian Referensi Next:

Mengubah referensi next pada simpul sebelumnya untuk menunjuk langsung ke simpul setelah simpul yang dihapus. Dengan kata lain, simpul yang dihapus dilewati dan tidak lagi terhubung di linked list.

4.3. *Delete* di Tengah

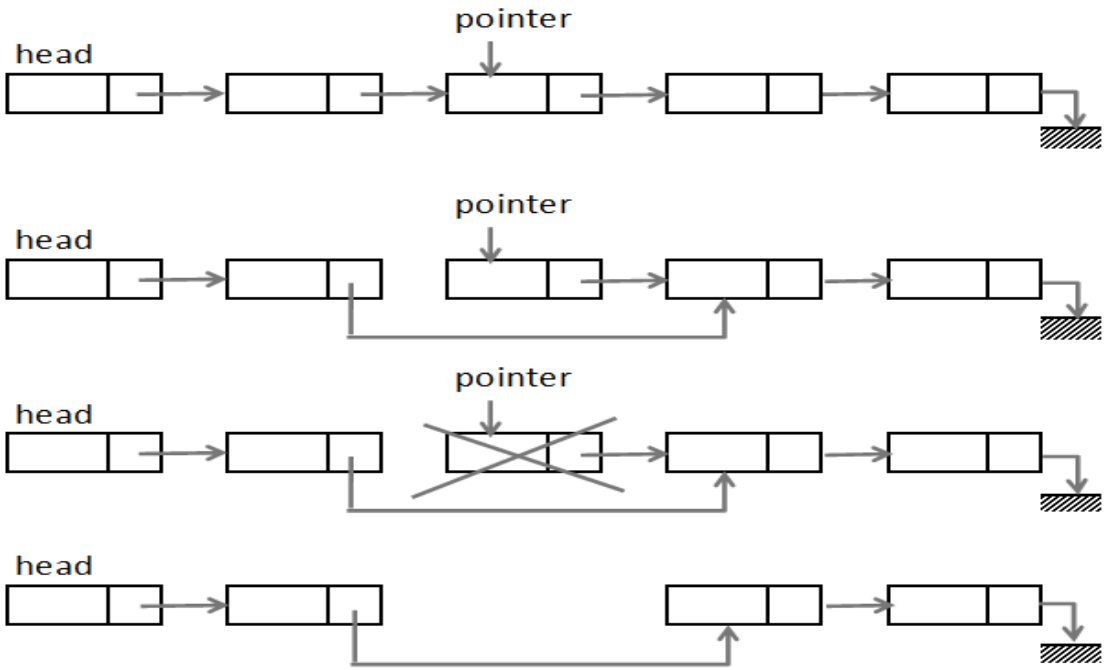
4. Penghapusan Simpul:

Simpul pada posisi tertentu dihapus dari linked list. Python akan secara otomatis mengurus pengelolaan memori untuk simpul yang tidak lagi terpakai.

5. Penyesuaian Ukuran Linked List:

Jika diperlukan, kita dapat menyesuaikan ukuran linked list dengan mengurangkan 1 dari nilai ukuran linked list.

4.3. *Delete* di Tengah (Lanj.)



4.4. *Delete* Pada Akhir *List*

1. Pengecekan Kondisi Awal:

•Memeriksa apakah linked list kosong. Jika linked list kosong, operasi delete di akhir tidak dapat dilakukan.

2. Pengecekan Jumlah Elemen:

•Jika linked list hanya memiliki satu elemen, maka elemen tersebut dihapus, dan referensi kepala diatur menjadi **None**.

3. Iterasi ke Sebelum Akhir:

•Jika linked list memiliki lebih dari satu elemen, iterasi dilakukan hingga mencapai elemen sebelum elemen terakhir. Hal ini dilakukan untuk memperbarui referensi next pada elemen sebelumnya.

4.4. *Delete* Pada Akhir *List* (Lanj.)

4. Penghapusan Elemen Akhir:

•Setelah mencapai elemen sebelum elemen terakhir, referensi next pada elemen tersebut diatur menjadi **None**, sehingga elemen terakhir terputus dari linked list.

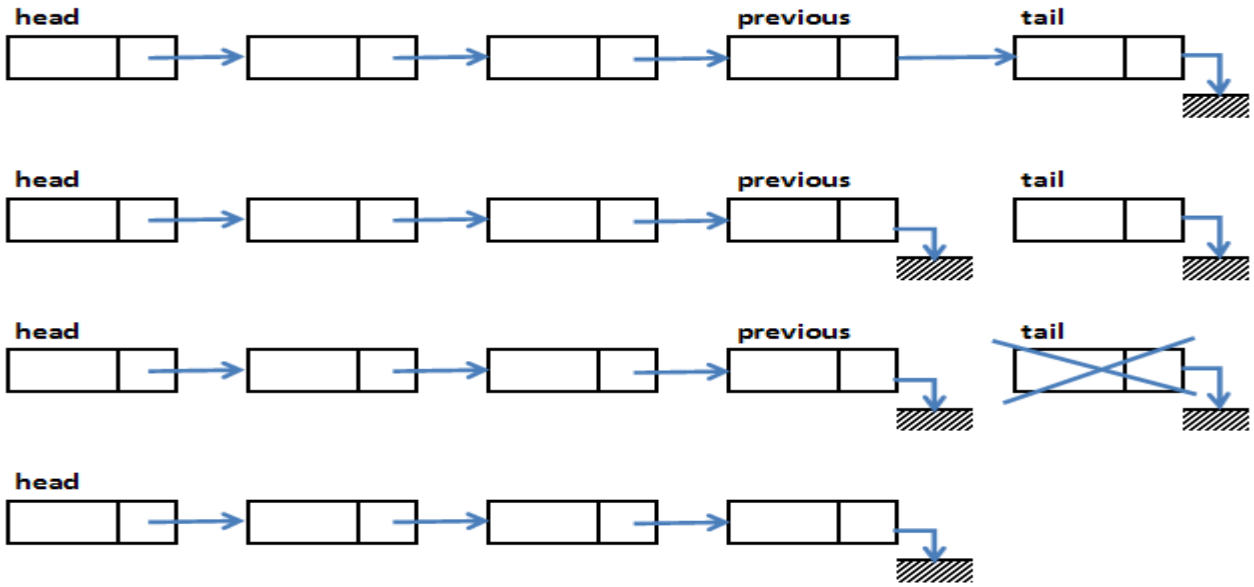
5. Pengelolaan Memori:

•Python secara otomatis mengurus pengelolaan memori untuk elemen yang dihapus.

6. Penyesuaian Ukuran Linked List:

•Jika diperlukan, ukuran linked list dikurangi 1 setelah operasi delete di akhir.

4.4. *Delete* Pada Akhir *List (Lanj.)*



4.4 Contoh Delete pada Akhir List

```
class singleLL:
    # Pembuatan kelas Node
    class _Node:
        def __init__(self, element, nextNode=None):
            self.element = element
            self.nextNode = nextNode

    # Pembuatan Single Linked List
    def __init__(self):
        self._head = None
        self._size = 0

    # Pembuatan Fungsi View All
    def __str__(self):
        result = ''
        pointer = self._head
        while pointer is not None:
            result += str(pointer.element) + ' '
            pointer = pointer.nextNode
        return result
```

```
    # Add First pada List yang masih kosong
    def add_first(self, element):
        newNode = self._Node(element)
        newNode.nextNode = self._head
        self._head = newNode
        self._size += 1

    # Menghapus Node di Akhir Linked List
    def delete_last(self):
        if self._head is None:
            raise IndexError("Linked list is empty")
        elif self._size == 1:
            self._head = None
        else:
            current_node = self._head
            while current_node.nextNode.nextNode is not N
                current_node = current_node.nextNode
            current_node.nextNode = None
        self._size -= 1

    def __len__(self):
        return self._size
```

```
def main():
    mylist = singleLL()
    mylist.add_first(50)
    mylist.add_first(30)
    mylist.add_first(10)

    print("Original Linked List:")
    print(str(mylist))
    print(len(mylist))

    # Menghapus node di akhir
    mylist.delete_last()

    print("\nLinked List setelah menghapus node di akhir:")
    print(str(mylist))
    print(len(mylist))
```

Output

```
Original Linked List:
10 30 50
3

Linked List setelah menghapus node di akhir:
10 30
2
```

Contoh Phyton Lengkap

```
class Simpul:
  def __init__(self,dat):
    self.dat = dat
    self.next = None

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
    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
      ptr.next = simpulBaru
```

```
def hapus(self,dihapus):
  ptr = self.head
  prev = None
  while ptr and (ptr.dat != dihapus):
    prev = ptr
    ptr = ptr.next
  if (ptr.dat == dihapus):
    if (prev==None):
      self.head = ptr.next
    else:
      prev.next = ptr.next
    del ptr

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

Contoh Phyton Lengkap (Lanj.)

```
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()
```

Ringkasan

- Single *Linked List* sangat tergantung pada *Head*, *Head* jangan sampai hilang.
- Penambahan, penghapusan, pemindahan node tidak boleh sampai membuat sebagian *link* hilang karena *link* terputus.
- Penggunaan variabel *pointer Tail* dapat mempermudah proses-proses pada *single linked list.*
- *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)