Bab 3. Linked List

OBJEKTIF :

1. Mahasiswa mampu mengetahui Linked List, macam-macam Linked List, dan ADT Linked List.
2. Mahasiswa mampu mengimplementasikan ADT Linked List menggunakan Bahasa Pemrograman Python.

3.1 Linked List

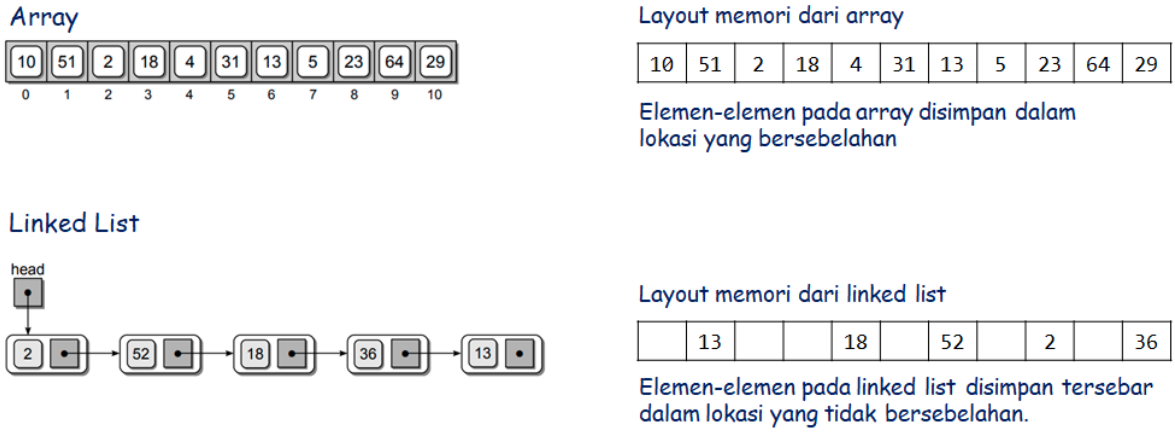
Array adalah wadah untuk barisan data yang paling dasar dan digunakan untuk menyimpan sebuah koleksi data. Array menyediakan akses langsung ke elemen individu. Tetapi array terbatas dalam fungsionalitasnya. Tipe list Python, yang juga merupakan wadah untuk barisan data, adalah tipe barisan abstrak yang diimplementasikan menggunakan struktur array. list mengekstensi fungsionalitas dari array dengan menambahkan operasi-operasi lain dan dapat secara otomatis menyesuaikan ukurannya ketika elemen-elemen ditambahkan atau dikurangi.

Array dan list dapat digunakan untuk mengimplementasikan banyak tipe data abstract. Keduanya menyimpan data dalam urutan linear dan menyediakan akses yang mudah ke elemen-elemennya. Pencarian biner (*binary search*) dapat digunakan dengan kedua struktur tersebut ketika data-data disimpan secara terurut sehingga memungkinkan pencarian cepat. Tetapi terdapat beberapa kekurangan dari penggunaan array dan list. Pertama, operasi *insertion* (penyisipan) dan *deletion* (penghapusan) memerlukan elemen-elemen digeser untuk membuat ruang atau menutup ruang kosong. Proses ini dapat memakan waktu, khususnya untuk barisan data yang besar. Kedua, ukuran dari array adalah tetap dan tidak dapat diubah. Sedangkan pada list, walaupun list dapat membesar dan mengecil, proses ini memerlukan ruang memori. Karena elemen-elemen dalam list disimpan dalam sebuah array, ekspansi memerlukan pembuatan array baru yang lebih besar yang ke dalamnya elemen-elemen dari array sebelumnya harus disalin. Kekurangan lain dari array dan list adalah elemen-elemen dari array dan list disimpan dalam ruang memori yang berurutan. Setiap kali array atau list dibuat, program harus mencari dan mengalokasikan ruang memori yang cukup untuk menyimpan keseluruhan isi array atau list tersebut. Untuk array atau list yang besar, hal ini dapat sulit dilakukan atupun tidak memungkinkan untuk program mengalokasikan ruang memori yang cukup untuk menyimpan elemen-elemen array atau list tersebut. Terlebih pada list Python yang dapat menjadi lebih besar saat eksekusi program karena setiap ekspansi memerlukan hingga dua kali ruang memori yang lebih besar daripada yang dibutuhkannya.

Pada bagian ini, kita akan membahas struktur data linked list, yang merupakan struktur data alternatif dari array dan list. Linked list meningkatkan cara konstruksi dan pengelolaan data yang terdapat di array dan list dengan hanya memerlukan alokasi memori yang lebih kecil dan tidak adanya pergeseran elemen untuk *insertion* dan *deletion*. Tetapi linked list mempunyai kekurangan dibandingkan array dan list karena elemen-elemennya tidak bisa diakses secara langsung. Oleh karena ini, tidak semua persoalan penyimpanan data cocok menggunakan linked list.

Perbedaan mendasar antara linked list dengan array atau list adalah linked list tidak menyimpan data-data dalam ruang memori yang bersebelahan namun data-data dapat tersebar dalam lokasi-lokasi acak dalam memori. Lokasi-lokasi tersebar tersebut dihubungkan dengan *link* untuk membentuk rangkaian data linear. Oleh karena ini, linked list lebih efisien karena tidak membutuhkan alokasi ruang memori yang besar di awal, karena ruang memori hanya

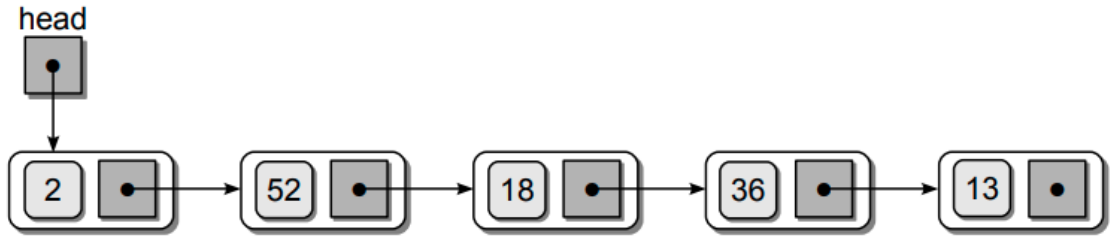
berkembang setiap data ditambahkan ke linked list. Gambar berikut mengilustrasikan perbedaan linked list dengan array:



Terdapat beberapa macam linked list. Linked list paling sederhana adalah *singly* linked list. Istilah linked list umumnya mengacu ke *singly* linked list. Variasi lainnya dari linked list termasuk: *circularly* linked list, *doubly* linked list, dan *circularly doubly* linked list. Pada bagian ini kita akan membahas *singly* linked list.

3.2 ADT Linked List

Structure linked terdiri dari koleksi *object-object* yang disebut dengan *node*, dimana masing-masing *node* berisi data dan referensi (pointer) ke *node* berikutnya. Referensi ke *node* berikutnya ini disebut dengan *link*. *Linked list* adalah struktur *linked* yang setiap *node*-nya terhubung dalam barisan untuk membentuk list linear. Gambar berikut mencontohkan sebuah *linked* list berisi lima *node*:



Definisi ADT Linked List

Linked list menyimpan data-data dalam *node-node* yang terhubung satu sama lain yang membentuk urutan linear. Setiap *node* menyimpan data dan menyimpan referensi ke sebuah *node* lain. Operasi-operasi yang dapat dilakukan terhadap linked list adalah sebagai berikut:

- `LinkedList()` : *Constructor* untuk membuat *linked list*.
- `length()` : Mengembalikan banyaknya *node* dalam *linked list*.
- `contains(nilai)` : Mengembalikan `True` jika `nilai` ada di *linked list* dan mengembalikan `False` jika `nilai` tidak ada di *linked list*. Diakses menggunakan operator `in`.
- `addFirst(data)` : Menambahkan sebuah *node* di awal *linked list*.
- `remove(data)` : Menghapus *node* yang menyimpan `data`. *Node* yang pertama yang ditemukan yang dihapus. Jika terdapat *node-node* lain setelahnya yang menyimpan `data`, *node-node* tersebut tidak dihapus.
- `iterator()` : Meng-*traverse* semua *node* dalam *linked list*.

Program berikut mencontohkan penggunaan *linked list*:

```
from linkedlist import LinkedList
```

```
def main():
    # Buat sebuah linked list.
    myList = LinkedList()

    # Isi linked list dengan data menggunakan
    # method addFirst(data).
    myList.addFirst(34)
    myList.addFirst(45)
    myList.addFirst(22)
    myList.addFirst(21)

    # Melakukan pencarian terhadap nilai yang dimasukkan pengguna
    # Pencarian dilakukan dengan memanggil method contains melalui operator in.
    nilai = int(input('Masukkan nilai yang dicari: '))
    if nilai in myList:
        print(f'{nilai} ditemukan dalam linked list.')
    else:
        print(f'{nilai} tidak ditemukan dalam linked list.')

    # Menghapus data dari linked list dari nilai yang dimasukkan pengguna
    # Penghapusan data dilakukan dengan memanggil method remove(data)
    nilai = int(input('Masukkan nilai untuk dihapus: '))
    try:
        myList.remove(nilai)
        print(f'{nilai} dihapus dari linked list.')
    except ValueError:
        print('Data tidak ditemukan. Tidak ada yang di-remove.')

    # Mencetak data pada linked list dengan memanggil method iterator.
    # Pemanggilan method iterator dengan menggunakan loop for terhadap linked
list.
    for data in myList:
        print(data)

main()
```

Implementasi ADT Linked List

Untuk mengimplementasikan ADT Linked List, kita memerlukan dua class:

1. Sebuah class yang merepresentasikan node dari linked list dan
2. Sebuah class yang merepresentasikan linked list itu sendiri.

Class yang merepresentasikan node kita sebut sebagai class Node dan class yang merepresentasikan linked list kita sebut sebagai class LinkedList. Class Node, yang digunakan sebagai representasi sebuah node, mempunyai *field* untuk menyimpan data node tersebut dan *field* referensi ke node selanjutnya. Sedangkan class LinkedList adalah struktur linked list yang menggunakan class Node sebagai penyimpan data dan mengimplementasikan *method-method* dari operasi-operasi pada linked list.

Class Node

Class node mempunyai dua *field*: *field* pertama untuk menyimpan data dan *field* kedua sebagai referensi (pointer) ke node selanjutnya. Kita menamakan class untuk node ini sebagai `_Node`. Dinamakan dengan diawali *underscore* karena class ini bersifat *private* dan hanya digunakan oleh class yang mengimplementasikan Linked List. Class `_Node` tidak memiliki *method* apapun selain *method constructor* untuk membuat *object* `_Node`.

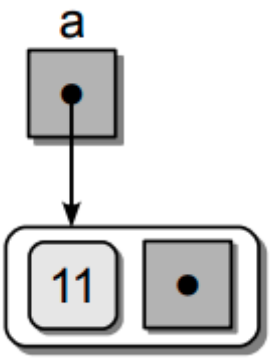
Class `_Node` dapat diimplementasikan seperti berikut:

```
class _Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Misalkan, kode berikut:

```
a = _Node(11)
```

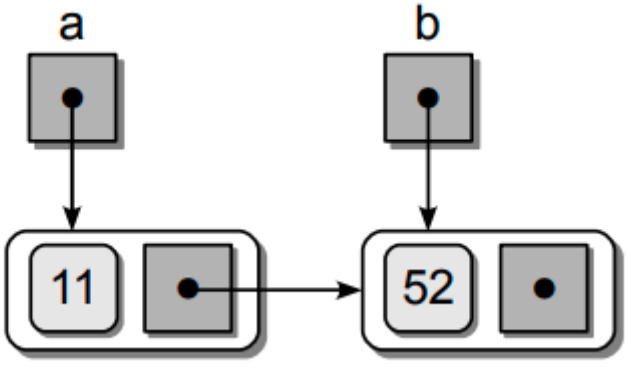
akan menghasilkan node seperti gambar berikut:



Untuk membuat keterhubungan antar node, kita menugaskan *field* `_next` dari suatu node ke node berikutnya. Sebagai contoh, kode berikut membuat node baru, `b`, dan menugaskan *field* `_next` dari node `a` ke `b`:

```
b = _Node(52)
a._next = b
```

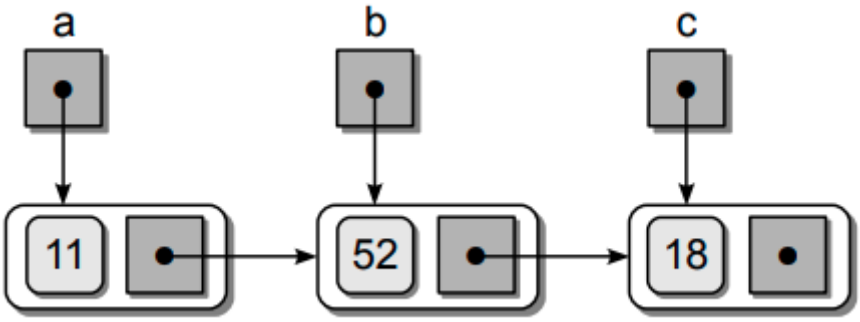
Sehingga, sekarang terdapat dua node dengan keterhubungan seperti berikut:



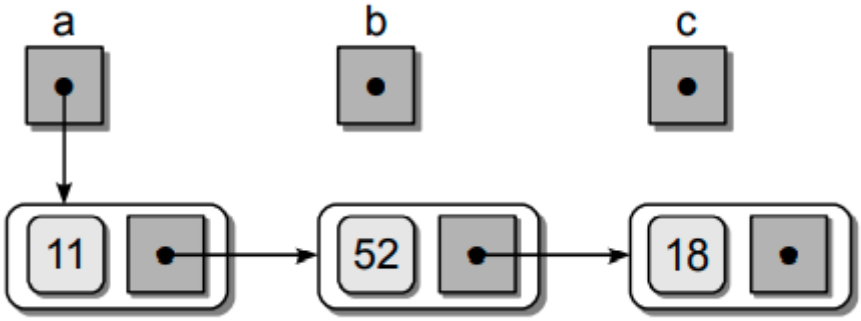
Kita dapat melanjutkan penambahan node dengan cara yang sama:

```
c = _Node(18)
b._next = c
```

Sehingga setelah kode di atas dieksekusi kita akan mempunyai node-node yang terhubung seperti berikut:



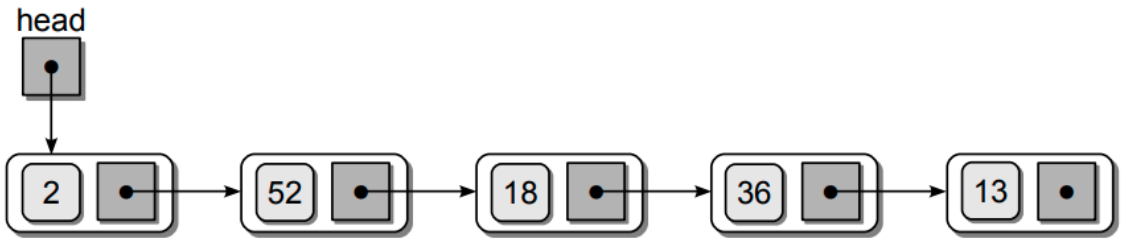
Kita dapat menghapus referensi ke node kedua dan node ketiga dengan menugaskan `b` dan `c` ke `None`. Sehingga, kita akan mempunyai node-node seperti berikut:



Hasil dari node-node dan keterhubungannya seperti gambar di atas adalah struktur linked list. Kita hanya memerlukan satu referensi ke node pertama karena node kedua dan ketiga yang sebelumnya direferensikan oleh `b` dan `c` tetap dapat diakses menggunakan referensi `a`. Sebagai contoh, misalkan kita ingin mencetak nilai dari ketiga node tersebut. Kita dapat mengakses node kedua dan ketiga melalui *field* `next` dari node pertama:

```
print(a.data)
print(a.next.data)
print(a.next.next.data)
```

Node pertama dalam linked list harus direferensikan oleh sebuah variabel untuk memberikan pintu masuk ke linked list. Variabel ini disebut sebagai ***pointer head*** atau ***referensi head***. Ilustrasi linked list beserta *referensi head*-nya dapat dilihat pada gambar berikut:



Ketika sebuah linked list dibuat pertama kali, linked list tersebut akan tidak mempunyai node sama sekali. Linked list tanpa node disebut linked list kosong. Linked list kosong ditandai dengan referensi *head* bernilai `None`.

Class `LinkedList`

Berikut adalah kerangka dari class `LinkedList` beserta class `_Node` yang sebelumnya telah kita tulis:

```
class LinkedList:
    # ... implementasi method-method

class _Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Selanjutnya kita akan mengimplementasikan *method-method* di dalam class `LinkedList`.

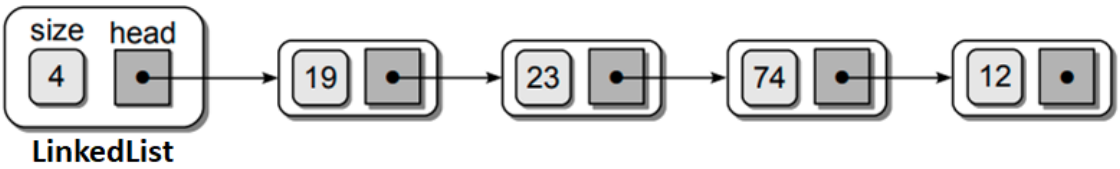
Constructor `LinkedList()`

Pada *constructor* `LinkedList()`, kita mendefinisikan dua *field*: *field* pertama adalah referensi ke node pertama dalam linked list yang kita namakan dengan `_head` dan *field* kedua adalah ukuran dari linked list (banyaknya node dalam linked list) yang kita namakan dengan `_size`.

Ketika linked list dibuat dengan *constructor*, kita akan mempunyai sebuah linked list kosong. Sehingga, *field* `_head` kita inisialisasikan dengan `None` yang berarti menunjuk ke *null*, dan *field* `_size` kita inisialisasikan dengan nilai 0. Kode berikut adalah implementasi *constructor* linked list:

```
def __init__(self):
    self._head = None
    self._size = 0
```

Untuk mengisi linked list ini dengan node-node kita menggunakan *method* `addFirst()` yang akan kita implementasikan berikutnya. Contoh linked list ini setelah pengisian node-node menggunakan *method* `addFirst()` dapat dilihat pada gambar berikut:



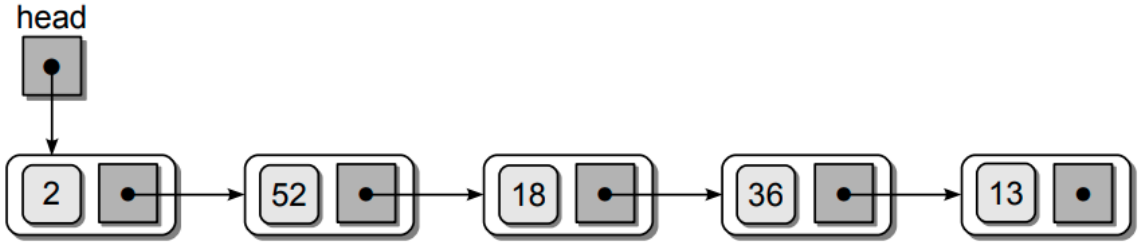
Method `length()`

Untuk mengimplementasikan *method* `length()`, kita hanya perlu mengembalikan nilai dari *field* `_size`. Kode berikut adalah implementasi *method* `length()`:

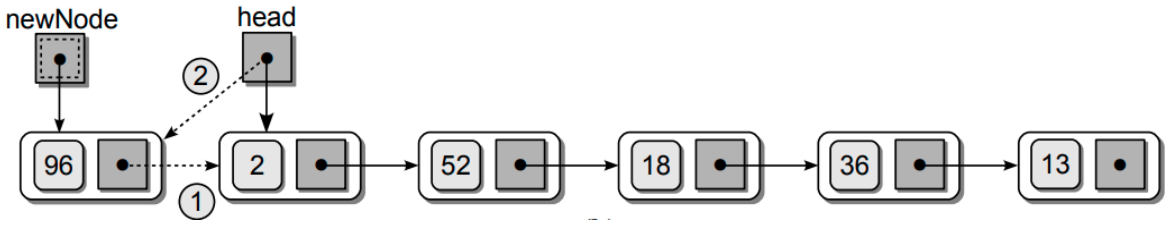
```
def __len__(self):
    return self._size
```

Method `addFirst(data)`

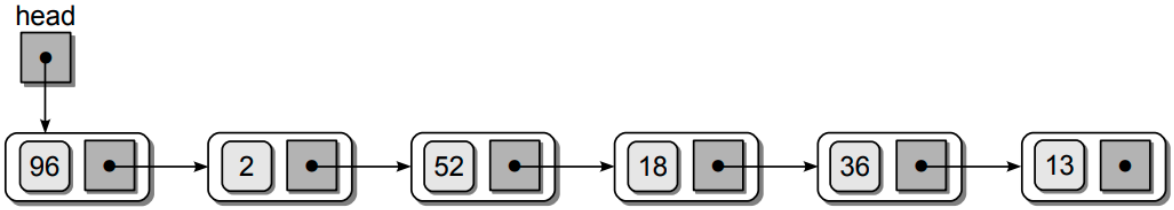
Method `addFirst()` menambahkan node pada awal linked list. Misalkan kita mempunyai linked list dengan node-node seperti berikut:



dan kita ingin menambahkan nilai 96 ke linked list tersebut. Pertama, kita harus membuat sebuah node baru yang menyimpan nilai 96 dan menugaskan *field* `_next` dari node tersebut ke node pertama dalam linked list. Lalu, kita mengubah *field* `_head` pada linked list untuk mereferensikan node yang baru. Proses ini diilustrasikan seperti berikut:



Hasil akhir dari proses penambahan nilai 96 adalah linked list seperti gambar berikut:



Setelah proses penambahan node, kita juga perlu menginkrementasikan *field* `_size`.

Kode implementasikan algoritma penambahan node di atas pada *method* `addFirst()` dapat dituliskan seperti berikut:

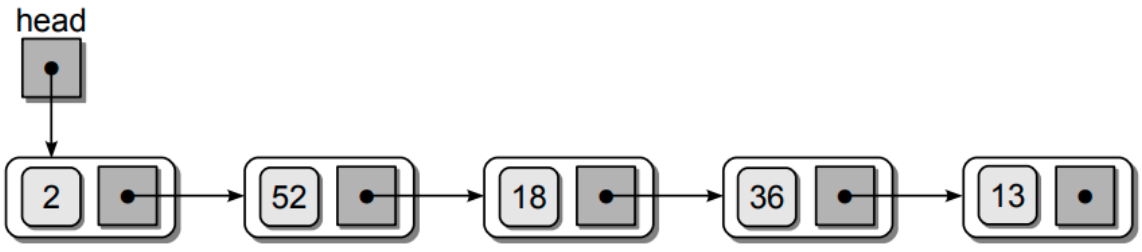
```
def addFirst(self, data):
    newNode = _Node(data)
    newNode.next = self._head
    self._head = newNode
    self._size += 1
```

Method `contains(data)`

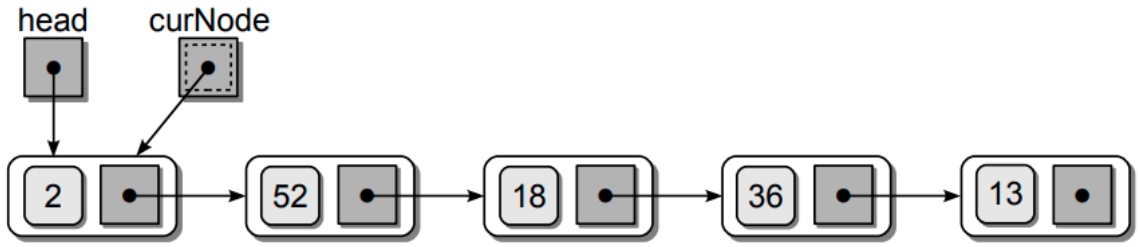
Method `contains()` digunakan untuk mencari data dalam linked list. *Method* ini dipanggil melalui operator `in`. *Method* ini diimplementasikan untuk mengembalikan `True` jika data yang dicari ditemukan dalam linked list dan mengembalikan `False` jika data yang dicari tidak ditemukan.

Pencarian data pada linked list dilakukan dengan meng-*traverse* node per node dari node pertama hingga node dengan data yang dicari ditemukan. Untuk meng-*traverse* node-node pada linked list kita membutuhkan sebuah *loop* dan sebuah referensi bantu. Referensi bantu ini yang bergerak dari node pertama ke node berikutnya setiap iterasi sampai dengan node dengan data yang dicari ditemukan. Kita menamakan referensi bantu ini dengan `curNode` singkatan dari *current node* (yang berati node sekarang).

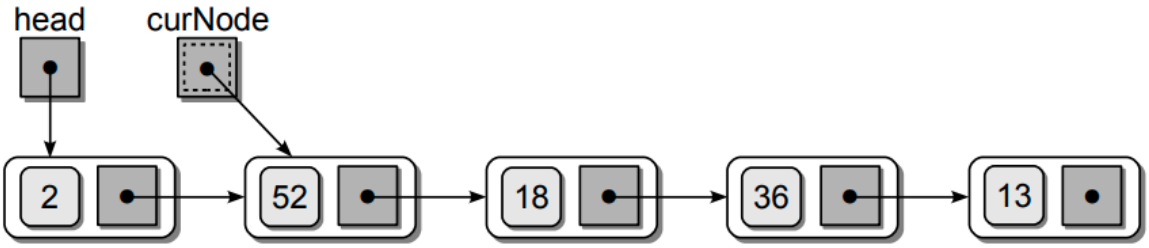
Misalkan kita mencari nilai 18 pada linked list berikut:



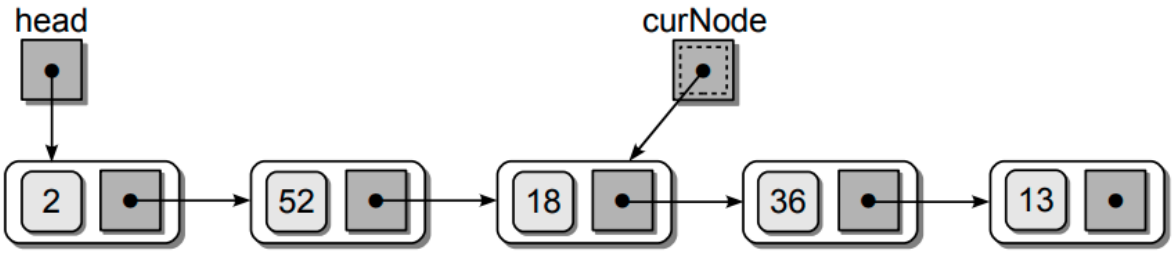
Proses pencarian dimulai dengan menugaskan referensi bantu `curNode` ke referensi *head* dari linked list:



Lalu, kita menguji apakah data dalam node mempunyai nilai yang dicari. Karena nilai pada node yang ditunjuk `curNode` bukan bernilai 18, maka kita memajukan `curNode` ke node berikutnya:

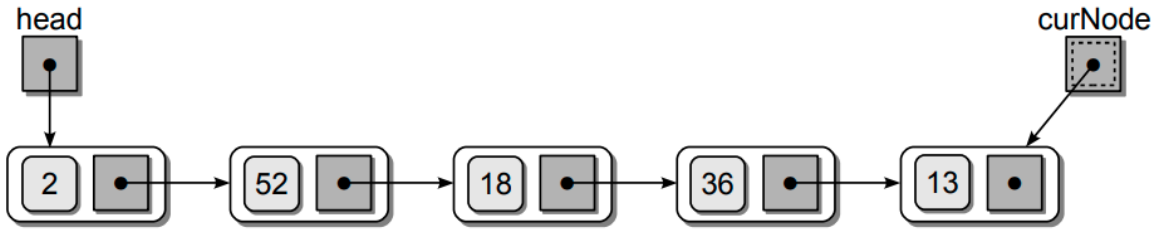


Karena node yang ditunjuk `curNode` sekarang tidak bernilai 18, kita memajukan `curNode` ke node berikutnya:

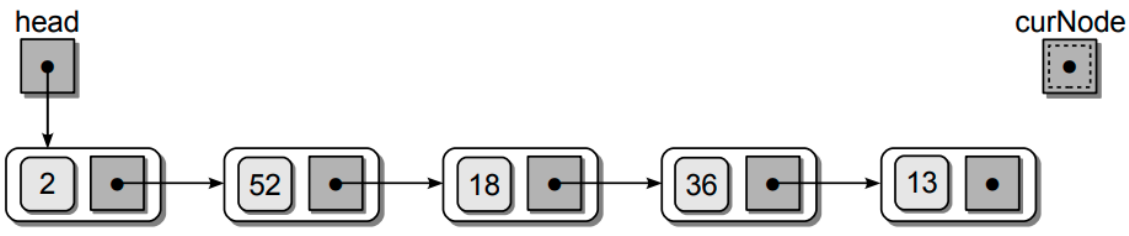


Karena node yang ditunjuk `curNode` sekarang bernilai 18, maka kita mengembalikan `true`.

Sekarang misalkan kita mencari data yang tidak terdapat dalam linked list. Ini akan menyebabkan `curNode` terus maju sampai ke node terakhir:



Dan karena pada node terakhir *field* `next` dari node tersebut menunjuk ke `None`, maka setelah menguji data yang dicari dengan data pada node terakhir `curNode` akan dimajukan sehingga menunjuk ke `None`



Dari contoh proses pencarian di atas, kita dapat menyimpulkan, proses traversing yang mencari data dalam linked list dapat berakhir berdasarkan dua kondisi:

- Ketika `curNode` mereferensikan `None`. Ini berarti semua node telah dikunjungi dan tidak ditemukan node dengan data yang dicari.
- Ketika `curNode` mereferensikan sebuah node yang menyimpan data yang dicari.

Berdasarkan dua kondisi tersebut, kita dapat menuliskan proses pencarian data menggunakan *loop* `while` seperti berikut:

```
while curNode is not None and curNode.data != data:
    curNode = curNode.next
```

Dan untuk mengetahui apakah node dengan data yang dicari ditemukan kita hanya perlu menguji apakah `curNode` mereferensikan `None` atau tidak. Sehingga *method* `contains()` dapat kita tuliskan seperti berikut:

```
def __contains__(self, data):
    curNode = self._head
    found = False
    while curNode is not None and curNode.data != data:
        curNode = curNode.next
    if curNode is not None:
        found = True
    else:
        found = False
    return found
```

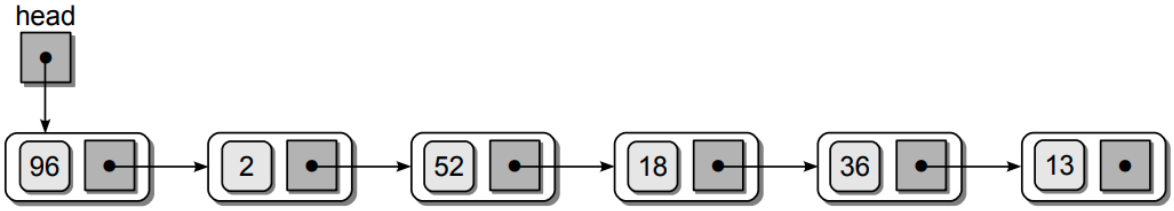
Pada kode di atas, kita membuat sebuah variabel `found` sebagai variabel yang menyimpan nilai kembali dari *method* ini. Ketika *loop* selesai dieksekusi, `curNode` akan bernilai `None` jika data yang dicari tidak ditemukan dan akan bernilai bukan `None` jika data yang dicari ditemukan. Sehingga setelah *loop* kita menuliskan kondisional yang menetapkan `found` sebagai `True` jika `curNode` tidak `None` dan menetapkan `found` sebagai `False` jika `curNode` bernilai `None`.

Perlu juga diperhatikan bahwa urutan kondisi pada *loop* `while` di atas tidak bisa ditukar. Kita harus menguji apakah `curNode` tidak mereferensikan `None` terlebih dahulu sebelum menguji *field* `_data` dari node yang direferensikan `curNode`. Jika kita mencoba mengakses *field* `_data` ketika `curNode` mereferensikan `None`, maka ini akan menyebabkan *error run-time*.

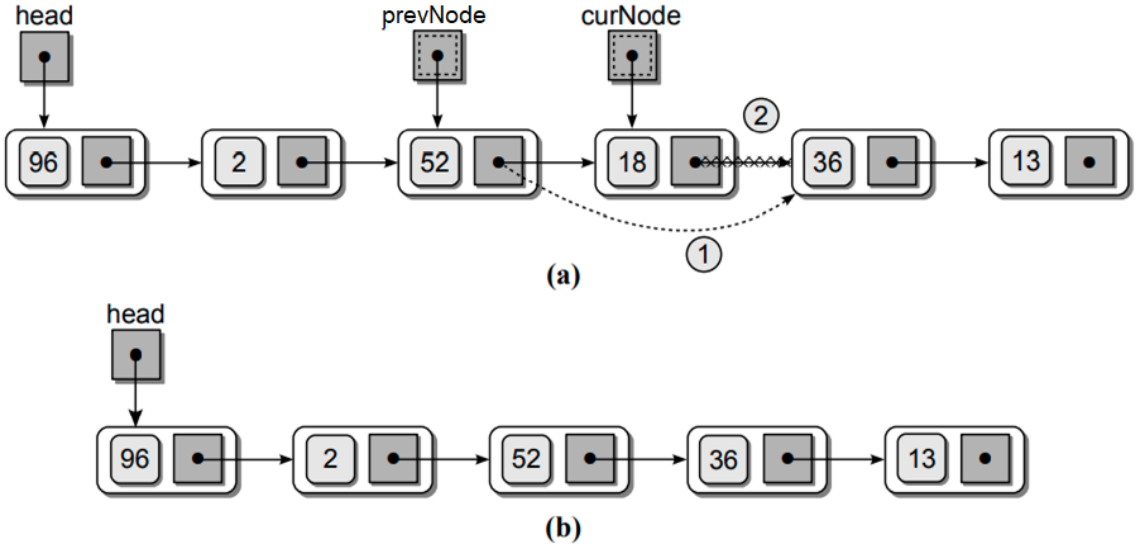
Method `remove(data)`

Method `remove()` menghapus node yang pertama kali ditemukan dalam linked list yang menyimpan data yang sama dengan data yang diberikan. Untuk menghapus node yang berisi data yang ingin dihapus, kita pertama harus mencari node tersebut. Cara pencarian node dapat kita lakukan seperti pada method `contains()` sebelumnya. Tetapi kita perlu cara untuk mengakses node sebelum node yang dicari untuk mengubah *link* `next`-nya. Oleh karena ini, kita membuat referensi bantu kedua yang kita gunakan untuk mengikuti referensi bantu `curNode` satu node dibelakangnya. Referensi bantu kedua ini kita namakan dengan `prevNode` yang berarti *previous node* atau node sebelumnya.

Misalkan kita ingin menghapus node yang menyimpan nilai 18 pada linked list berikut:



Untuk melakukannya, kita meng-*traverse* linked list ini satu per satu dari node pertama dengan menggunakan *loop* untuk menggerakkan `curNode` untuk maju node per node setiap iterasinya dan juga menggerakkan `prevNode` untuk mengikuti `curNode` satu node dibelakangnya. Gambar berikut mengilustrasikan proses penghapusan node:

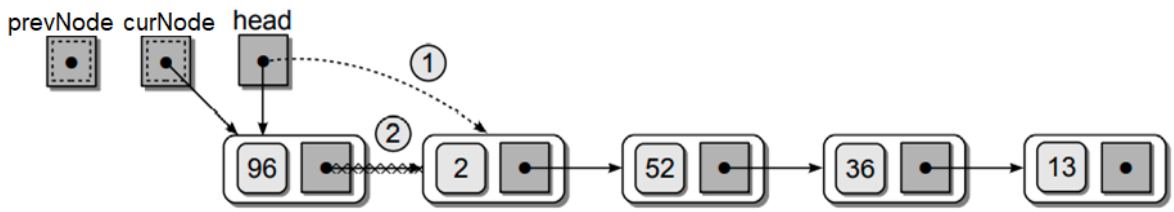


Pada Gambar (a), ketika `curNode` menemukan node yang ingin dihapus, *link* `next` pada node sebelumnya yang ditunjuk oleh `prevNode` diubah untuk menunjuk ke node setelah node yang ditunjuk `curNode`. Lalu, *link* `next` dari node yang ditunjuk `curNode` dihapus dengan menugaskannya ke `None`. Gambar (b) adalah hasil dari linked list setelah penghapusan node.

Terdapat tiga kondisi yang harus kita perhatikan saat penghapusan node:

1. Node yang ingin dihapus tidak ada dalam linked list;
2. Node yang ingin dihapus adalah node pertama dalam linked list;
3. Node yang ingin dihapus berada setelah node pertama

Kondisi 3 adalah kondisi yang kita gunakan untuk contoh proses penghapusan node sebelumnya. Kondisi 1 ditandakan dengan `curNode` menunjuk ke `None` setelah proses *traverse* seluruh node. Jika ini terjadi, kita tidak perlu melakukan apapun. Untuk kondisi 2, yaitu saat node yang ingin dihapus adalah node pertama dalam lnked list, kita perlu menguji apakah `curNode` menunjuk ke node yang sama seperti yang ditunjuk oleh `head`. Jika ya, kita menetapkan `head` untuk menunjuk ke node berikutnya dalam linked list, seperti terlihat pada gambar berikut:



Method `remove()` dapat dituliskan seperti berikut:

```
def remove(self, data):
```

```
    curNode = self._head
    prevNode = None
    while curNode is not None and curNode.data != data:
        prevNode = curNode
        curNode = curNode.next
    if curNode is None:
        raise ValueError('Data tidak ditemukan.')
    else:
        self._size -= 1
        if curNode is self._head:
            self._head = curNode.next
        else:
            prevNode.next = curNode.next

    return curNode.data
```

Baris 7, adalah jika terjadi kondisi 1, yang berarti jika node yang berisi data yang ingin dihapus tidak ditemukan. Jika terjadi kondisi 1, kita meng-*raise* ekspesi `ValueError`. Baris 11 adalah kondisi 2, yaitu kondisi dimana node yang ingin dihapus adalah node pertama. Jika kondisi 2 terjadi, kita menetapkan `head` untuk menunjuk node setelah node pertama. Baris 13 adalah kondisi 3, ketika node yang ingin dihapus berada setelah node pertama. Jika kondisi 3 terpenuhi, kita mengubah *link* `next` dari node sebelum node yang dihapus ke node berikutnya dari node yang dihapus.

Method `iterator()`

Pembuatan iterator untuk linked list mirip dengan pembuatan iterator pada ADT-ADT lain yang sebelumnya telah kita lakukan. Kita memerlukan sebuah class iterator. Pada class iterator kita mendefinisikan sebuah *field* berupa variabel referensi yang menunjuk ke node yang sedang diproses. *Field* ini kita namakan sebagai `_curNode`. Variabel referensi `_curNode` digunakan sama seperti saat kita menggunakannya untuk melakukan proses *traversing* linked list pada *method* `contains()` maupun *method* `remove()`. Hanya saja pada class iterator ini, kita tidak menuliskan *loop* `while` karena class ini dimaksudkan untuk digunakan dalam *loop* `for`.

Kode untuk class iterator yang kita namakan sebagai class `_LinkedListIterator` dapat dituliskan seperti berikut:

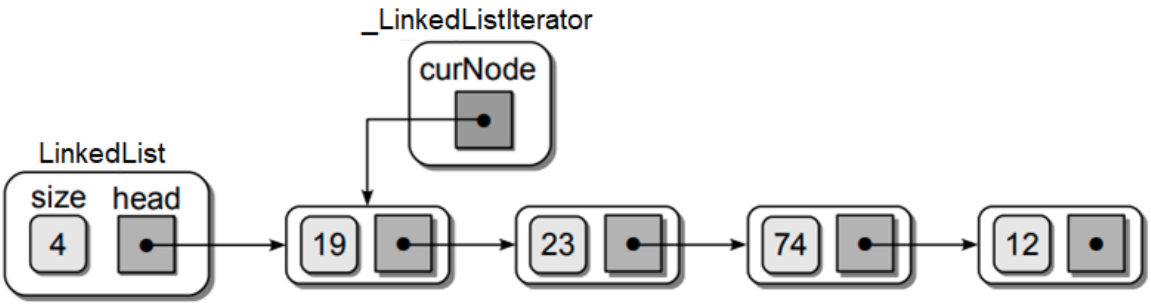
```
class _LinkedListIterator:
    def __init__(self, listHead):
        self._curNode = listHead

    def __iter__(self):
        return self

    def __next__(self):
        if self._curNode is None:
            raise StopIteration
        else:
            data = self._curNode._data
            self._curNode = self._curNode.next
            return data
```

Pada class `_LinkedListIterator` di atas, dalam *method* `__next__` kita menghentikan iterasi dengan meng-*raise* eksepsi `StopIteration` ketika `_curNode` menunjuk ke `None` yang berarti `_curNode` telah mengunjungi semua node dalam linked list. Lalu, pada klausa `else` kita menuliskan kode untuk mengembalikan data pada node yang ditunjuk `_curNode` dan memajukan `_curNode` ke node berikutnya.

Kita juga perlu menuliskan *method* `__iter__` pada class `LinkedList` yang mengembalikan *object* dari class `_LinkedListIterator`:

```
def __iter__(self):
    return _LinkedListIterator(self._head)
```

Gambar berikut mengilustrasikan *object* `LinkedList` dan `LinkedListIterator` saat awal dari *loop* `for`:



Kode Lengkap Implementasi ADT Linked List

Implementasi lengkap dari ADT Linked List dapat dilihat pada kode berikut:

Module *`linkedlist.py`*

```
# Implementasi ADT LinkedList
class LinkedList:
    # Constructor Linked List (dimulai dengan linked list kosong)
    def __init__(self):
        self._head = None
        self._size = 0

    # Method length() mengembalikan banyaknya elemen dalam linked list.
    # Method ini diakses menggunakan fungsi len().
    def __len__(self):
        return self._size

    # Method addFirst(data) menambahkan data ke awal linked list.
    # Method ini tidak mengembalikan nilai.
    def addFirst(self, data):
        newNode = _Node(data)
        newNode.next = self._head
        self._head = newNode
        self._size += 1

    # Method contains(data) digunakan untuk mencari data dalam linked list.
    # Method ini diakses menggunakan operator in dan
    # mengembalikan nilai Boolean: True jika data ditemukan dan False jika
    # data tidak ditemukan.
    def __contains__(self, data):
```

```
        curNode = self._head
        found = False
        while curNode is not None and curNode.data != data:
            curNode = curNode.next
        if curNode is not None:
            found = True
        else:
            found = False
        return found

    # Method remove() menghapus data dari linked list.
    # Method ini mengembalikan data yang dihapus.
    def remove(self, data):
        curNode = self._head
        prevNode = None
        while curNode is not None and curNode.data != data:
            prevNode = curNode
            curNode = curNode.next
        if curNode is None:
            raise ValueError('Data tidak ditemukan.')
        else:
            self._size -= 1
            if curNode is self._head:
                self._head = curNode.next
            else:
                prevNode.next = curNode.next

        return curNode.data

    # Method iterator untuk meng-traverse linked list
    # Method ini mengembalikan object dari class `_LinkedListIterator`
    # dengan argumen referensi head.
    def __iter__(self):
        return _LinkedListIterator(self._head)

# Definisikan class _Node sebagai penyimpanan data dan referensi
# ke node selanjutnya.
class _Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Class iterator untuk ADT Linked List
class _LinkedListIterator:
    def __init__(self, listHead):
        self._curNode = listHead

    def __iter__(self):
        return self

    def __next__(self):
        if self._curNode is None:
            raise StopIteration
        else:
            data = self._curNode.data
            self._curNode = self._curNode.next
            return data
```

Menguji Linked List

Kita dapat menguji implementasi linked list menggunakan program berikut:

```
from linkedlist import LinkedList

def main():
    # Buat sebuah linked list.
    myList = LinkedList()

    # Isi linked list dengan data menggunakan
    # method addFirst(data).
    myList.addFirst(34)
    myList.addFirst(45)
    myList.addFirst(22)
    myList.addFirst(21)

    # Melakukan pencarian terhadap nilai yang dimasukkan pengguna
    # Pencarian dilakukan dengan memanggil method contains melalui operator in.
    nilai = int(input('Masukkan nilai yang dicari: '))
    if nilai in myList:
        print(f'{nilai} ditemukan dalam linked list.')
    else:
        print(f'{nilai} tidak ditemukan dalam linked list.')

    # Menghapus data dari linked list dari nilai yang dimasukkan pengguna
    # Penghapusan data dilakukan dengan memanggil method remove(data)
    nilai = int(input('Masukkan nilai untuk dihapus: '))
    try:
        myList.remove(nilai)
        print(f'{nilai} dihapus dari linked list.')
    except ValueError:
        print('Data tidak ditemukan. Tidak ada yang di-remove.')

    # Mencetak data pada linked list dengan memanggil method iterator.
    # Pemanggilan method iterator dengan menggunakan loop for terhadap linked
list.
    for data in myList:
        print(data)

main()
```

Output program di atas:

```
Masukkan nilai yang dicari: 34
34 ditemukan dalam linked list.
Masukkan nilai untuk dihapus: 22
22 dihapus dari linked list.
21
45
34
```

REFERENSI

[1] Necaise, Rance D. 2011. Data structures and algorithms using Python .