2_4-LinkedList

1 Linked List

Struktur Data ***

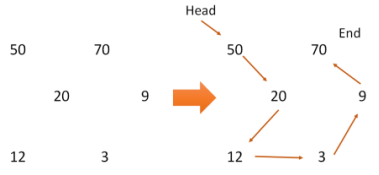
Bahasa pemrograman python telah menyediakan type data yang dinamis, yaitu **List**. Ukuran dari variabel yang bertipe data **list** dapat diatur sesuai dengan keinginan programmer selama program dijalankan, tidak harus mempunyai ukuran tetap di awal. Tipe data ini juga menyediakan method menambah data pada saat diperlukan, sehingga tipe data ini bersifat dinamis.

Akan tetapi tidak semua pemrograman menyediakan type data seperti ini, oleh karena itu terdapat suatu struktur data yang dapat dibuat oleh programmer yang bersifat dinamis, yaitu **Linked List**.

Lima hal utama yang terdapat pada struktur data linked list: 1. Section 1.1 2. Section 1.3 3. Section 1.5 4. Section 1.7 5. Section 1.13

1.1 Node

Data di dalam memory berada di alamat yang berbeda-beda, jika dibutuhkan agar data-data tersebut dapat terhubung satu sama lain, maka informasi tambahan yang menunjukkan alamat data berikutnya sangatlah diperlukan, seperti yang terlihat pada Gambar 1 berikut.



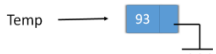
Gambar1. Data-data yang terhubung di dalam memory

Oleh karena itu data berikutnya dapat diketahui dengan cari mengikuti informasi *link* yang terdapat pada informasi tambahan tersebut. Lokasi data pertama dari **linked list** haruslah diketahui secara eksplisit, sehingga dari data pertama tersebut, data kedua data ditemukan, data ketiga, dan seterusnya. Informasi yang menunjuk pada lokasi data pertama atau **head of the list** tersebut disebut dengan **external reference**. Begitu juga dengan data terakhir, harus ada informasi yang menunjukkan bahwa data tersebut adalah data terakhir di dalam linked list, tidak ada lagi data berikutnya.

Untuk membuat struktur data linked list, terlebih dahulu dibuat **node-node** penyusun linked list tersebut. **Node** ini harus memiliki setidaknya dua informasi, yaitu informasi mengenai data atau nilai, dan informasi mengenai node berikutnya. Oleh karena itu **node** dibuat menjadi sebuah tipe data baru yang bertipe **class**, dengan dua informasi yaitu *data* dan *next*.

Terdapat beberapa method penting pada class **node** ini, antara lain: - *constructor*, yang akan dijalankan setiap instansiasi class - *getData*, untuk mengetahui informasi data yang terdapat pada

node tersebut - *getNext*, untuk mengetahui informasi node berikutnya, jika tidak ada node berikutnya maka nilai balik berupa *None* - *setData*, untuk merubah informasi data yang terdapat pada node tersebut - *setNext*, untuk menentukan node berikutnya yang ditunjukan oleh informasi *next* dari node tersebut



Contoh Node dapat dilihat pada Gambar 2. **Gambar 2. Node dengan nilai '93'**

Pada Gambar 2 tersebut, terdapat dua informasi yang terkandung di dalam node, yaitu data dari node adalah 93, dan node tersebut menunjuk ke *Nil* atau *Ground* atau tidak ada node yang ditunjuk.

1.2 Code

Berikut adalah pembuatan class Node yang merupakan representasi dari sebuah node. *Property* atau *state* yang terdapat pada class Node ini : 1. data : berisi nilai dari node 2. next : berisi informasi berikutnya yang ditunjuk oleh node. Proses intansiasi, property next ini diset `None` yang merupakan representasi Nil atau Ground, berarti tidak ada node yang ditunjuk oleh node ini

```
In [1]: class Node:
            def __init__(self, init_data):
                self.data = init_data
                self.next = None
            def getData(self):
                return self.data
            def getNext(self):
                return self.next
            def setData(self, newdata):
                self.data = newdata
            def setNext(self, new_next):
                self.next = new_next
```

Berikut adalah contoh penggunaan class Node.

```
In [3]: a=Node(93)
        b=Node(20)
        print(a.getNext())
        print(b.getNext())
        a.setNext(b)
        print(a.getNext())

None
None
<__main__.Node object at 0x0000005979C08550>
```

Pada code diatas, terdapat obyek `a` dan obyek `b` yang memiliki tipe data class `Node`. Pada saat instansiasi, property data kedua obyek ini bernilai `93` dan `20`, serta property `next` bernilai **None**. Pada baris kelima, terdapat perintah `a.setNext(b)`, yang berarti property `next` dari obyek

`a` akan menunjuk ke obyek `b`. Sehingga ketika dilakukan perintah `print(a.getNext())` akan menunjukkan ke suatu class Node.

Section 1

1.3 Linked List Class

Linked list merupakan kumpulan dari node-node yang terhubung satu sama lain. Untuk mengakses node-node yang terdapat pada linked list tersebut, haruslah diketahui terlebih dahulu lokasi node pertama dari suatu linked list, sehingga diperlukan pointer tambahan untuk menunjukkan keberadaan node pertama.

1.4 Code

Berikut adalah class untuk linked list, dimana pada class tersebut terdapat pointer yang menunjukkan node pertama dari suatu linked list (*head*). Terdapat dua buah method utama pada class LinkedList ini, antara lain: 1. *constructor*, `__init__`, yang merupakan method yang dijalankan pada saat pembuatan obyek. Karena obyek baru pertama kali dibuat, maka linked list masih kosong, sehingga pointer *head* masih bernilai *None*. 2. Method `isEmpty`, untuk pengecekan apakah linked list memiliki node ataukah tidak. Jika pointer *head* masih menunjuk pada *None*, maka linked list masih tidak memiliki node, sehingga return value adalah True.

```
In [0]: class LinkedList:
            def __init__(self):
                self.head = None
            def isEmpty(self):
                return self.head==None
```

Berikut ini adalah cara penggunaan class LinkedList yang telah dibuat

```
In [0]: mylist=LinkedList()

In [0]: print(mylist.head)
        mylist.isEmpty()

In [0]: print(mylist.head)

In [0]: mylist.isEmpty()
```

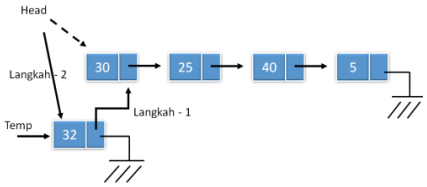
Obyek mylist yang telah dibuat pada code diatas yang bertipe *linked list*, adalah linked list yang masih kosong, belum ada node dalam list tersebut, seperti yang ditunjukkan pada Gambar 3.

Gambar 3. Obyek mylist berbentuk Linked List yang belum memiliki Node

Section 1

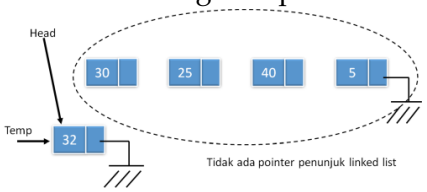
1.5 Penambahan Data pada Linked List

Secara default, penambahan data baru diletakkan pada awal linked list atau yang terdapat pada pointer *head*. Penambahan data baru ini, dilakukan dengan dua tahapan : - Tautkan node baru ini ke node awal dari linked list - modifikasi head dari linked list agar menunjuk pada node baru ini,



seperti yang ditunjukkan pada Gambar 4 berikut : **Gambar 4. Penambahan node baru secara benar pada linked list**

Urutan tahapan ini **tidak boleh dibalik**, karena jika dibalik maka linked list yang awal tidak lagi dapat ditemukan seperti yang terdapat pada Gambar 5 berikut :



Gambar 5. Penambahan node baru secara tidak tepat pada linked list

1.6 Code

Berikut adalah penambahan method `add()` pada class LinkedList. Seperti yang dijelaskan sebelumnya, penambahan node baru harus dilakukan dengan urutan yang tepat, yaitu:

```
temp=Node(item) #temp adalah node baru yang akan ditambahkan
temp.setNext(self.head) #pointer Next dari node temp menunjuk pada node yang ditunjuk oleh poi
self.head=temp #pointer head menunjuk pada node temp yang sudah tersambung dengan linked list
```

```
In [4]: class LinkedList:
            def __init__(self):
                self.head = None
            def isEmpty(self):
                return self.head==None
            def add(self, item):
                temp = Node(item)
                temp.setNext(self.head)
                self.head = temp
```

Berikut adalah contoh penggunaan class LinkedList yang sudah dibuat

```
In [5]: mylist=LinkedList()

In [6]: print(mylist.head)
        mylist.isEmpty()

None


Out[6]: True

In [7]: mylist.add(34)
        print(mylist.head)
```

```
<__main__.Node object at 0x0000005979C03AC8>
```

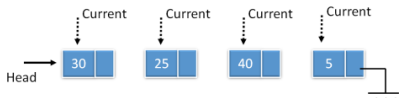
```
In [8]: mylist.add(45)
        print(mylist.head)
        print(mylist.head.data)

<__main__.Node object at 0x0000005979C03550>
45
```

```
In [9]: mylist.add(21)
```

Setelah penambahan node terakhir diatas, maka mylist akan berisi = [21, 45, 34]
Section 1

1.7 Traversal Linked List

Untuk mengetahui ukuran dari list, diperlukan tahapan **traverse**, yaitu menelusuri setiap node yang terdapat pada linked list, seperti pada Gambar 6 sebagai berikut :



Gambar 6. Traversal Linked List

Pada proses penelusuran atau *traversal* dibutuhkan pointer bantuan. Pointer bantuan yang ditunjukkan pada Gambar 6 tersebut, adalah pointer current yang bergerak dari node awal sampai dengan node akhir. Proses traversal ini dibutuhkan untuk beberapa hal, seperti untuk menghitung jumlah node yang terdapat pada Linked List, untuk mencari node pada linked list, untuk menampilkan seluruh node dari linked list, untuk menyisipkan node setelah atau sebelum node yang sudah terdapat pada linked list, dan untuk menghapus suatu node.

1.8 Code

Implementasi pertama yang dibuat adalah pembuatan method size(), untuk menghitung jumlah node. Pada method size terdapat beberapa tahapan : 1. pointer bantuan current berada pada node yang ditunjuk oleh head (yaitu node pertama) 2. pointer current bergerak, dengan perintah `current=current.getNext()`, sekaligus dilakukan increment variabel count, yang merepresentasikan jumlah node 3. pergerakan atau traversal ini akan berakhir ketika pointer current menunjuk pada None, yang merepresentasikan, tidak ada lagi node yang terdapat pada Linked list

```
In [10]: class LinkedList:
             def __init__(self):
                 self.head = None
             def isEmpty(self):
                 return self.head==None
             def add(self, item):
                 temp = Node(item)
                 temp.setNext(self.head)
                 self.head = temp
```

```
        def size(self):
            current = self.head
            count = 0
            while current != None:
                count = count + 1

                current = current.getNext()
            return count
```

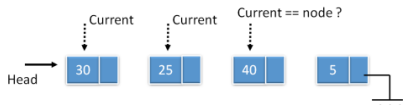
Berikut contoh penggunaan method `size()` pada class LinkedList

```
In [4]: mylist=LinkedList()
        mylist.add(45)
        mylist.add(34)
        mylist.add(70)
        print(mylist.size())
```

```
3
```

1.8.1 Pencarian Node pada Linked List

Untuk pencarian node di dalam linked list juga perlu dilakukan *traverse* linked list seperti sebelumnya, hanya saja setiap berada pada suatu node, maka dilakukan pencocokan apakah node tersebut adalah node yang dicari, seperti Gambar 7 berikut:



Gambar 7. Pencarian Node pada Linked List

Seperti halnya pada method sebelumnya, terdapat pointer current yang bergerak dari awal sampai akhir, hanya saja diperlukan pencocokan antara data yang terdapat pada current dengan data yang dicari. Implementasi pencarian dapat dilihat pada method berikut

1.9 Code

Berikut penambahan method `search()` pada class LinkedList. Method `search()` ini hampir sama dengan method `size`, hanya saja jika ditambahkan apakah node yang ditunjukkan oleh pointer current adalah node yang dicari, dengan perintah:

```
if current.getData() == item: #dimana item adalah node yang dicari
```

maka method ini akan menghasilkan nilai True.

```
In [11]: class LinkedList:
            def __init__(self):
                self.head = None
            def isEmpty(self):
                return self.head==None
            def add(self, item):
```

```
            temp = Node(item)
            temp.setNext(self.head)
            self.head = temp
        def size(self):
            current = self.head
            count = 0
            while current != None:
                count = count + 1
                current = current.getNext()
            return count
        def search(self,item):
            current = self.head
            found = False
            while current != None and not found:
                if current.getData() == item:
                    found = True
                else:
                    current = current.getNext()
            return found
```

Berikut adalah contoh penggunaan method `search()`

```
In [12]: mylist=LinkedList()
         mylist.add(45)
         mylist.add(34)
         mylist.add(70)
         mylist.add(84)
         mylist.size()
         mylist.search(34)

Out[12]: True
```

1.9.1 Menampilkan seluruh data pada Linked list

Proses traversal juga dapat digunakan untuk menampilkan data pada seluruh node yang terdapat pada Linked List.

1.10 Code

Berikut method `display()` untuk menampilkan data dari seluruh node pada linked list

```
In [5]: class LinkedList:
            def __init__(self):
                self.head = None
            def isEmpty(self):
                return self.head==None
            def add(self, item):
                temp = Node(item)
                temp.setNext(self.head)
```

```
        self.head = temp
    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count + 1
            current = current.getNext()
        return count
    def search(self,item):
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()
        return found
    def display(self):
        current = self.head
        while current != None:
            print(current.getData())
            current = current.getNext()
```

Berikut adalah contoh penggunaan method `display()`

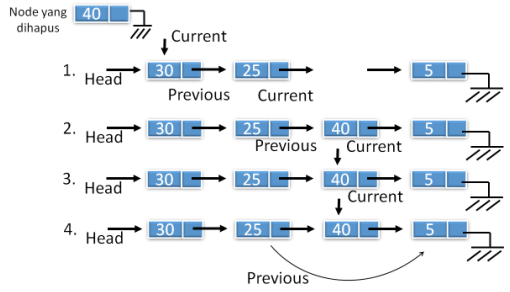
```
In [6]: mylist=LinkedList()
        mylist.add(45)
        mylist.add(34)
        mylist.add(70)
        mylist.add(84)

In [7]: mylist.display()

84
70
34
45
```

1.10.1 Penghapusan Node pada Linked List

Penghapusan data (remove data) dilakukan dengan dua tahapan yaitu : - traverse linked list, untuk mencari node mana yang akan dihapus - remove data, menghapus node dari linked list

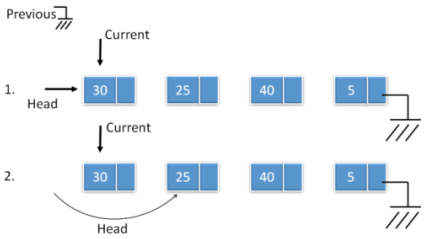
Untuk traverse linked list sama hal nya dengan method sebelumnya, yaitu terdapat pointer **current** yang menelusuri node awal sampai dengan node akhir untuk mencari node yang akan dihapus. Ketika node sudah ditemukan, penghapusan tidak dapat dilakukan langsung begitu saja, karena di dalam node terdapat informasi lain, yaitu next yang menunjukkan lokasi node berikutnya. Sebelum dilakukan penghapusan perlu dilakukan terlebih dahulu pentautan antara

node sebelum node yang dicari dengan node sesudah node yang dicari, sehingga node yang dicari dapat dihapus. Oleh karena itu diperlukan pointer tambahan selain **current**, yaitu pointer **previous**. Pointer ini bergerak satu langkah sebelum pointer current, sehingga ketika pointer **current** sudah menemukan data yang dicari, maka pointer **previous** ini dapat ditautkan dengan node sesudah **current**. Ilustrasi ketika data yang akan dihapus berada di tengah *linked list* ditunjukkan pada Gambar 8.



Gambar 8. Penghapusan Node pada Linked List

Jika data yang dihapus berada di node awal dari linked list (yang ditunjukkan dengan nilai previous masih *None*, maka yang dilakukan pointer *head* langsung menunjuk pada node setelah node yang akan dihapus tersebut, seperti Gambar 9 berikut:



Gambar 9. Penghapusan Node awal pada Linked List

1.11 Code

Berikut adalah penambahan method `remove()` untuk menghapus node yang diinginkan pada linked list

```
In [13]: class LinkedList:
             def __init__(self):
                 self.head = None
             def isEmpty(self):
                 return self.head==None
             def add(self, item):
                 temp = Node(item)
                 temp.setNext(self.head)
                 self.head = temp
             def size(self):
                 current = self.head
                 count = 0
                 while current != None:
                     count = count + 1
                     current = current.getNext()
                 return count
             def search(self,item):
```

```
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()
        return found
    def display(self):
        current = self.head
        while current != None:
            print(current.getData())
            current = current.getNext()
    def remove(self, item):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

Berikut adalah contoh penggunaan method `remove()`

```
In [14]: mylist=LinkedList()
         mylist.add(45)
         mylist.add(34)
         mylist.add(70)
         mylist.add(84)
         mylist.add(97)
         mylist.display()
```

```
97
84
70
34
45
```

```
In [15]: mylist.remove(34)
```

```
In [16]: mylist.display()
```

```
97
84
70
45
```

1.12 Latihan

Tambahkan dua buah method sebagai berikut: - insertPrevious, yaitu menambahkan node baru sebelum node tertentu - insertNext, yaitu menambahkan node baru setelah node tertentu

Section 1

1.13 Ordered List

Proses pencarian pada linked list sebelumnya dilakukan dengan cara mencari node satu persatu sampai node terakhir. Proses pencarian ini akan menjadi lebih cepat jika data sudah dalam keadaan terurut, sehingga pencarian dapat dihentikan ketika ditemukan node dengan data lebih rendah atau lebih tinggi. Class Ordered List akan memudahkan pencarian suatu node, karena data yang terdapat pada class ini sudah dalam keadaan terurut.

Method yang terdapat pada ordered list, sama halnya dengan class linkedlist, hanya saja terdapat perbedaan pada method untuk **add data** (karena node yang terbentuk harus dalam keadaan terurut), dan method search data.

1.14 Code

Berikut method untuk class ordered list

```
In [19]: class OrderedLinkedList:
             def __init__(self):
                 self.head = None
             def isEmpty(self):
                 return self.head==None

             def size(self):
                 current = self.head
                 count = 0
                 while current != None:
                     count = count + 1
                     current = current.getNext()
                 return count

             def display(self):
                 current = self.head
                 while current != None:
                     print(current.getData())
                     current = current.getNext()
             def remove(self, item):
                 current = self.head
```

```
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
    def search(self,item):
        current = self.head
        found = False
        stop=False
        while current != None and not found and not stop:
            if current.getData() == item:
                found = True
            else:
                if current.getData() > item:
                    stop = True
                else:
                    current = current.getNext()
        return found
    def add(self, item):
        current=self.head
        previous = None
        stop = False
        while current != None and not stop:
            if current.getData() > item:
                stop = True
            else:
                previous = current
                current = current.getNext()

        temp = Node(item)
        if previous == None:
            temp.setNext(self.head)
            self.head = temp
        else: # ditautkan antara previous dengan current
            temp.setNext(current)
            previous.setNext(temp)
```

Berikut adalah contoh penggunaan class ordered Linked List

```
In [20]: myList=OrderedLinkedList()

In [21]: myList.add(9)
```

```
In [22]: myList.add(14)

In [23]: myList.display()

9
14


In [24]: myList.add(100)
         myList.display()

9
14
100
```

Section 1