Pengenalan Struktur Data

A. Apa itu Struktur Data?

Struktur Data adalah sebuah organisasi dan manajemen data yang dibuat sedemikian rupa sehingga memungkinkan pengaksesan dan modifikasi yang efisien. Lebih tepatnya, struktur data adalah koleksi dari sekumpulan nilai/data, hubungan diantara data-data tersebut, dan operasi-operasi yang dapat diaplikasikan pada data tersebut.[1]

B. Abstract Data Type (ADT): The Intro

Abstract Data Type, atau biasa disingkat ADT secara sederhana adalah tipe/objek dimana perilaku dari ADT tersebut didefinisikan oleh kumpulan nilai (*value*) dan kumpulan operasi. Abstract Data Type merupakan *interface* "abstrak" yang implementasinya disembunyikan dari user. Implementasi dari sebuah ADT bisa bermacam-macam, namun sebuah ADT selalu mempunyai maksud yang sama.

Contohnya adalah **Integer**. Integer merupakan ADT, yang mempunyai nilai-nilai $\ldots, -2, -1, 0, 1, 2, \ldots$ dan operasi-operasi seperti penjumlahan, pengurangan, perkalian, pembagian dsb. Disini, user tidak perlu tahu bagaimana Integer diimplementasikan pada komputer. User hanya perlu tahu bahwa "***there exist Integer***".[2]

ADT vs Struktur Data

- ADT merupakan ranah logikal (menyediakan interface) dan dapat diimplementasikan dengan berbagai cara.
- Struktur Data adalah implementasi konkret dari ADT.

Contoh Analogi:

> Bayangkan "Kendaraan" sebagai sebuah ADT. "Kendaraan" dapat diimplementasikan dengan berbagai bentuk, bisa mobil, kereta, bus, dll.

[1] Dikutip dan diterjemahkan dari https://en.wikipedia.org/wiki/Data_structure

[2] Dikutip dan disaring dari https://en.wikipedia.org/wiki/Abstract_data_type

Contoh-contoh ADT yang sering dijumpai pada bahasa pemrograman.

Abstract Data Type	Implementasi Struktur Data
List	Dynamic Array, Linked List
Stack	Linked List, Dynamic/Static Array
Queue	Linked List, Dynamic/Static Array
Bitset	Dynamic/Static Array
Priority Queue	Linked List, Heap
Set dan Map	Balanced Binary Search Tree (AVL Tree, Red-Black Tree)
(Unordered) Set dan Map	Hash Table
Graph	Directed/Undirected Graph

C. Jenis Struktur Data

Pada umumnya, struktur data dibedakan berdasarkan bentuk penyimpanannya. Struktur data dikelompokkan menjadi dua jenis, yakni **Struktur Data Linear** dan **Struktur Data Non-Linear**.

C.1 Struktur Data Linear

Suatu struktur data dianggap linear apabila data-datanya disusun dalam bentuk berupa sekuens linear berurutan (*one after another*). Contoh struktur data linear yang sering kita jumpai adalah **Array**. Contoh lainnya yang akan dibahas berupa **Linked list, Stack** dan **Queue**.

C.2 Struktur Data Non-Linear

Struktur data non-linear berbanding terbalik dengan linear. Dalam struktur data non-linear, data disusun dan dibangun tidak secara berurutan/sekuensial melainkan disusun secara hirarkikal menggunakan aturan tertentu. Contoh struktur data non-linear adalah **Tree** dan **Graf**.

C.3 Perbandingan Struktur Data Linear dan Non-Linear

Perbandingan	S.D. Linear	S.D. Non-Linear
Struktur	Data disusun secara sekuensial dan berurutan (*one after another*).	Data disusun tidak secara sekuensial, melainkan secara hirarkikal.
Implementasi	Mudah	Cenderung Sulit
Contoh	Array, Stack, List, Queue	Tree, Graf

D. Operasi Dasar Struktur Data

Beberapa operasi dasar yang ada pada struktur data meliputi operasi-operasi berikut.

- *INSERT* – yakni untuk menambahkan/memasukkan data baru pada struktur data.
- *RETRIEVE* – yakni operasi untuk mendapatkan/melihat/mengambil data pada struktur data.
- *DELETE* – yakni operasi untuk menghapus data pada struktur data.

Seluruh operasi dasar tersebut mempunyai karakteristik dan perilaku yang berbeda-beda tergantung pada struktur datanya masing-masing.

Notasi Big-O

A. Mengenal Kompleksitas Algoritma

Kompleksitas dari sebuah algoritma dapat dihitung dengan berbagai faktor. Untuk saat ini, kita hanya akan membatasinya dengan melihat kompleksitas dari sebuah algoritma melalui waktu yang dibutuhkan (*runtime*). Notasi yang umum digunakan untuk mengukur waktu (*runtime*) adalah "Big-O". Faktor penentunya adalah banyaknya data/input yang biasa dinotasikan dengan N. Notasi "Big-O" disini melambangkan kompleksitas yang diukur berdasarkan kasus terburuk (worst-case) dari sebuah proses/algoritma.

B. Contoh Umum

Berikut beberapa contoh umum penggunaan "Big-O" dalam merepresentasikan kompleksitas waktu dari sebuah algoritma.

Kompleksitas	Sebutan	Contoh
$O(1)$	Constant Time	Waktu yang dibutuhkan selalu konstan, tidak peduli seberapa besar jumlah data, contohnya **Mengakses elemen array**.
$O(N)$	Linear Time	Waktu yang dibutuhkan sebanding dengan jumlah data, contohnya **Linear Search**.
$O(\log N)$	Logarithmic Time	Biasanya pada algoritma yang membagi masalah menjadi masalah yang lebih kecil (sub-problem), contohnya **Binary Search**.
$O(N \log N)$	Linearithmic Time	Contohnya pada **Merge-Sort**.
$O(N^2)$	Quadratic Time	Contohnya pada **Bubble-Sort**.
$O(2^N)$	Exponential Time	Mencari seluruh subset dari sebuah set memerlukan waktu 2^N.
$O(N!)$	Factorial Time	Mencari seluruh permutasi dari sebuah string dengan panjang N.

Dynamic Array

Array mungkin sudah tidak asing lagi, merupakan struktur data paling dasar yang sering digunakan untuk menyimpan data. Array bersifat statis, yang artinya ukurannya selalu tetap dan tidak dapat berubah (grow/shrink). Namun, array memungkinkan kita untuk mengakses elemennya dengan waktu yang konstan $O(1)$. Kita akan menggunakan "Static Array" untuk menyebutkan array statis.

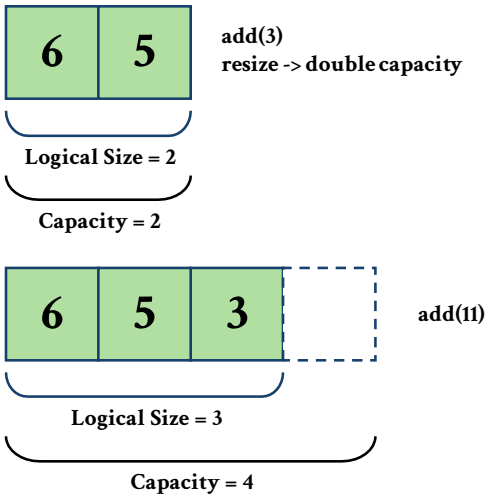
A. Definisi

Sama seperti namanya, **Dynamic Array** memungkinkan kita untuk membuat array yang memiliki kemampuan untuk berubah ukuran (size) sesuai dengan banyaknya data yang dimasukkan dengan tetap mempertahankan karakteristik array, yakni pengaksesan elemen menggunakan *indexing*.

B. Ilustrasi

Dynamic Array biasanya diimplementasikan menggunakan Static Array dengan kapasitas awal tertentu.

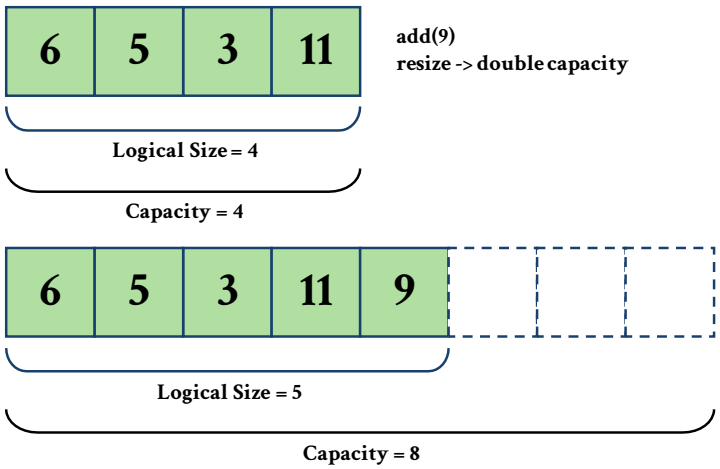
Misalkan, kapasitas awalnya adalah 2 dan sudah diisi dengan dua elemen yakni 6 dan 5. Ketika hendak mengisi lagi elemen 3, dapat dilihat bahwa kapasitas array sudah penuh.



Gambar B.1 Ilustasi Dynamic Array 1

Jika kapasitas array sudah penuh, maka akan mengalokasikan array baru dengan ukuran 2 kali ukuran awal, kemudian menyalin data dari array lama ke array baru. Array barulah yang akan digunakan. Disini, array terbagi menjadi dua area:

- ***Logical Size*** adalah area array yang dapat diakses.
- ***Capacity*** adalah ukuran array yang sebenarnya. Area diluar logical size seharusnya tidak dapat diakses.



Gambar B.2 Ilustrasi Dynamic Array 2

***Notes*:** Dynamic Array berbeda dengan array yang dialokasikan secara dinamis.

C. Static Array vs Dynamic Array

Berikut tabel perbandingan antara Static Array dengan Dynamic Array.

Static Array	Dynamic Array
Ukurannya tetap.	Ukuran dapat berubah sesuai dengan banyaknya data.
Sangat efisien, tidak melibatkan realokasi memori.	Kurang efisien dibandingkan Static Array karena memerlukan alokasi-realokasi memori pada saat run-time.
Bagus digunakan apabila ukuran data diketahui dan tetap.	Bagus digunakan apabila ukuran data tidak tetap.
Tidak efisien secara memori.	Sangat efisien secara memori.

D. Operasi Dasar

Secara umum, Dynamic Array mendukung operasi-operasi dasar seperti berikut.

- **pushBack** – menambahkan data baru dari belakang. Operasi ini mempunyai mekanisme yang sudah dijelaskan pada ilustrasi.
- **popBack** – menghapus data terakhir/paling belakang.
- **back** – mendapatkan data terakhir/paling belakang.
- **front** – mendapatkan data terdepan/data pertama.
- **getAt(i)** – mendapatkan data pada indeks ke-i (dilakukan secara konstan O(1)).
- **setAt(i, value)** – mengubah data pada indeks ke-i dengan nilai baru.
- **isEmpty** – memeriksa apakah array kosong atau tidak.

E. Implementasi ADT: *DynamicArray*

Implementasi Dynamic Array akan dibawa ke Abstract Data Type dengan nama yang sama, yakni ***DynamicArray*** yang menyimpan tipe data **int**. Kapasitas awal array adalah 2 dengan faktor pertumbuhan 2 kali lipat.

Operasi	Kompleksitas Waktu
isEmpty	$O(1)$
pushBack	amortized $O(1)$
popBack	$O(1)$
back	$O(1)$
front	$O(1)$
setAt	$O(1)$
getAt	$O(1)$

E.1 ADT: *DynamicArray*

Struktur ADT Dynamic Array berisi array (variabel **`_arr`**), kemudian informasi ukuran (**`_size`**) dan kapasitas array (**`_capacity`**).

```
typedef struct dynamicarr_t {
    int *_arr;
    unsigned _size, _capacity;
} DynamicArray;
```

E.2 isEmpty

```
bool dArray_isEmpty(DynamicArray *darray) {
    return (darray->_size == 0);
}
```

E.3 pushBack

```
void dArray_pushBack(DynamicArray *darray, int value)
{
    if (darray->_size + 1 > darray->_capacity) {
        unsigned it;
        darray->_capacity *= 2;
        int *newArr = (int*) malloc(sizeof(int) * darray->_capacity);

        for (it=0; it < darray->_size; ++it)
            newArr[it] = darray->_arr[it];

        int *oldArray = darray->_arr;
        darray->_arr = newArr;
        free(oldArray);
    }
    darray->_arr[darray->_size++] = value;
}
```

E.4 popBack

```
void dArray_popBack(DynamicArray *darray) {
    if (!dArray_isEmpty(darray)) darray->_size--;
    else return;
}
```

E.5 back

```
int dArray_back(DynamicArray *darray) {
    if (!dArray_isEmpty(darray))
        return darray->_arr[darray->_size-1];
    else return 0;
}
```

E.6 front

```
int dArray_front(DynamicArray *darray) {
    if (!dArray_isEmpty(darray))
        return darray->_arr[0];
    else return 0;
}
```

E.7 setAt

```
void dArray_setAt(DynamicArray *darray, unsigned index, int value)
{
    if (!dArray_isEmpty(darray)) {
        if (index >= darray->_size)
            darray->_arr[darray->_size-1] = value;
        else
            darray->_arr[index] = value;
    }
}
```

E.8 getAt

```
int dArray_getAt(DynamicArray *darray, unsigned index)
{
    if (!dArray_isEmpty(darray)) {
        if (index >= darray->_size)
            return darray->_arr[darray->_size-1];
        else
            return darray->_arr[index];
    }
}
```

Linked List

A. Terminologi

Sebelum masuk kepada definisi, terdapat istilah yang sering digunakan dalam mendeskripsikan linked list.

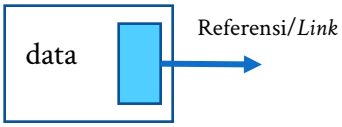
- ***Node*** – Sebuah *node* merepresentasikan satu elemen data yang dapat berisi nilai dan/atau informasi lain yang dibutuhkan serta terdapat hubungan atau *link/reference* ke *node* lain.
- ***Head*** – merupakan *node* pertama/paling depan dari linked list.
- ***Tail*** – merupakan *node* terakhir/paling belakang dari linked list.

B. Definisi

Linked list adalah struktur data yang menyimpan data dalam bentuk linear, dimana tiap-tiap data direpresentasikan oleh *node-node* yang membentuk sekuens secara berurutan.[3] Pada dasarnya, satu *node* dalam linked list terdiri dari:

- Data yang disimpan, dan
- Referensi (*link*) kepada node selanjutnya.

Contoh ilustrasi sebuah *node* dalam linked list.



Gambar B.1 Ilustrasi *Node* pada linked list

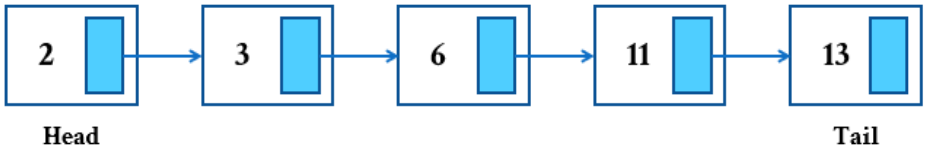
Karakteristik

Linked list merupakan struktur data yang bersifat **dinamis**, yang artinya ukurannya dapat berubah mengikuti banyaknya data yang dimasukkan/ditambahkan, tidak seperti Static Array. Namun, data pada linked list tidak bisa diakses secara random layaknya pengaksesan indeks pada array, melainkan harus melalui proses *traversing* terlebih dahulu.

[3] Disaring dari **wikipedia**

C. Ilustrasi

Linked list dapat diilustrasikan sebagai kumpulan node yang saling terhubung secara sekuensial membentuk rangkaian secara berurutan. Misalkan, terdapat list A dengan kumpulan data $A = [2, 3, 6, 11, 13]$.

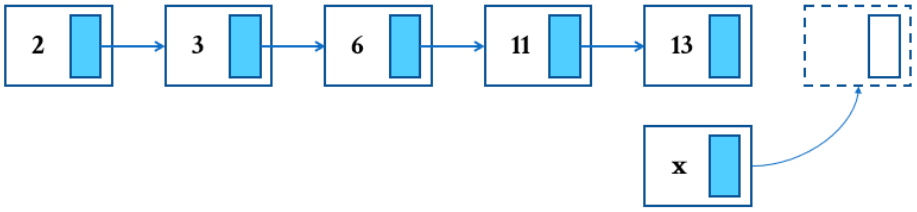


Gambar C.1 Ilustrasi untuk linked list ***A***

D. Operasi Dasar

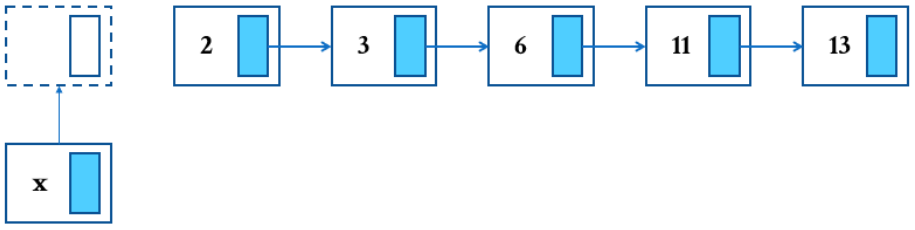
Beberapa operasi dasar yang sering diaplikasikan pada linked list adalah sebagai berikut.

- **pushBack** – operasi untuk menambahkan data baru dari belakang list.



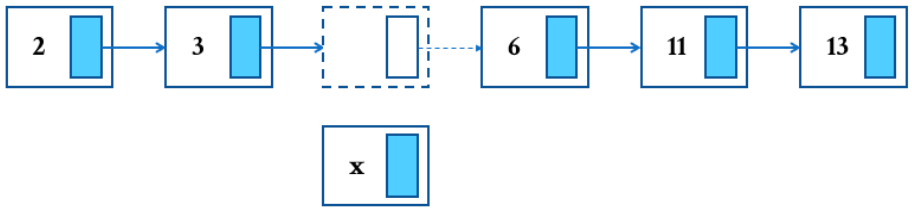
Gambar D.1 Ilustrasi operasi PUSH_BACK

- **pushFront** – operasi untuk menambahkan data baru dari depan list.



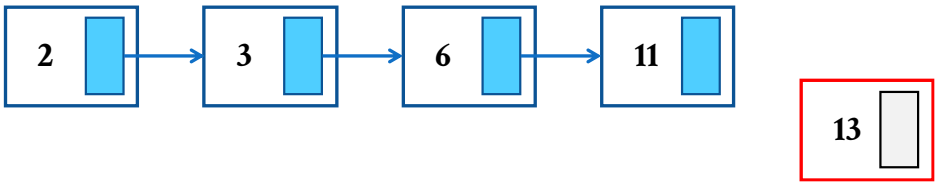
Gambar D.2 Ilustrasi operasi PUSH_FRONT

- **insertAt** – operasi untuk menambahkan data baru pada posisi yang diinginkan.



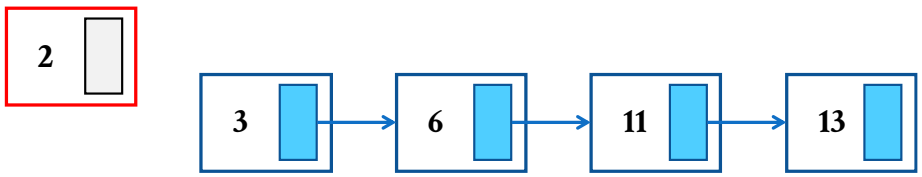
Gambar D.3 Ilustrasi operasi INSERT_AT

- **back** – untuk memperoleh data yang berada pada paling belakang.
- **front** – untuk memperoleh data yang berada pada paling depan.
- **find(x)** – untuk mencari apakah suatu data tertentu ada pada list.
- **popBack** – operasi untuk menghapus data yang berada pada paling belakang.



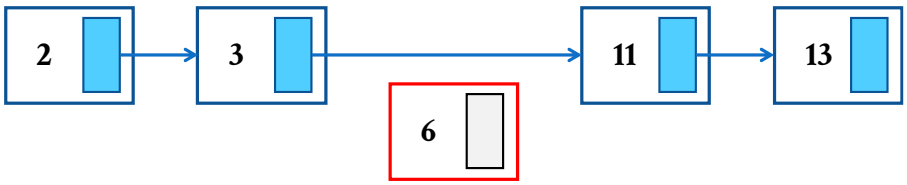
Gambar D.4 Ilustrasi operasi POP_BACK

- **popFront** – operasi untuk menghapus data yang berada pada paling depan.



Gambar D.5 Ilustrasi operasi POP_FRONT

- **remove(x)** – untuk menghapus data tertentu pertama pada list.



Gambar D.6 Ilustrasi operasi REMOVE(6)

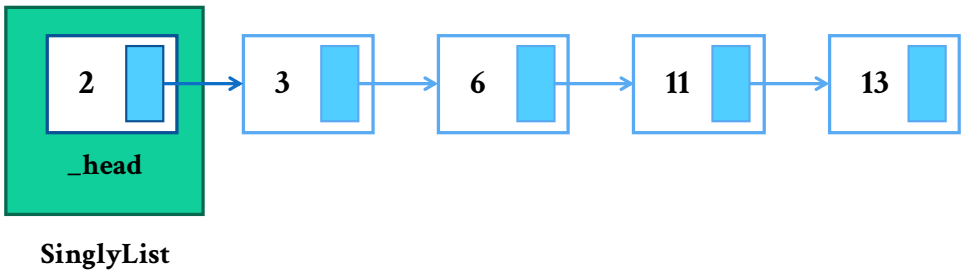
E. Variasi Linked List

-- to do --

F. Implementasi ADT: *SinglyList*

Implementasi lengkap dapat dilihat pada link berikut :
https://github.com/bayulaxana/data_structure_implementation

Representasi dan Implementasi yang dijelaskan dalam modul ini adalah **Singly Linked List** yang menyimpan tipe data **`int`**. Representasi akan dibawa ke dalam bentuk Abstract Data Type (ADT) yang nantinya akan menjadi tipe data baru bernama **`SinglyList`**.



Gambar F.1 Representasi ADT SinglyList

Dalam implementasi ini, kompleksitas waktunya adalah :

Operasi	Keterangan	Kompleksitas Waktu
pushBack	Memasukkan data baru dari belakang.	$O(N)$
pushFront	Memasukkan data baru dari depan.	$O(1)$
insertAt	Memasukkan data baru pada posisi tertentu.	$O(N)$
popBack	Menghapus node paling belakang.	$O(N)$
popFront	Menghapus node paling depan.	$O(1)$
remove(x)	Menghapus node pertama dengan data x.	$O(N)$
find(x)	Mencari data tertentu pada linked list.	$O(N)$
front	Mendapatkan nilai node terdepan.	$O(1)$
back	Mendapatkan nilai node paling belakang.	$O(1)$
getAt	Mendapatkan nilai node pada posisi tertentu.	$O(N)$

isEmpty	Memeriksa apakah list kosong.	$O(1)$

F.1 Representasi Node

Sebuah *node* pada singly linked list dalam bentuk paling dasar berisi **data** dan **referensi** ke *node* selanjutnya. Disini, data dinyatakan pada variabel **`data`** dan referensi berupa pointer **`next`**.

```
typedef struct snode_t {
    int data;
    struct snode_t *next;
} SListNode;
```

Nama dari struktur node tersebut adalah **`SListNode`**.

F.2 ADT: *SinglyList*

Setelah struktur untuk node tersedia, selanjutnya adalah membuat struktur ADT. ADT **SinglyList** akan berisi satu *node* yang berperan sebagai **head** dan satu informasi untuk menyimpan **size** (ukuran List).

```
typedef struct slist_t {
    unsigned _size;
    SListNode *_head;
} SinglyList;
```

F.3 pushBack

Secara umum, langkah untuk melakukan *pushBack* adalah sebagai berikut.

1 buat node baru dan isikan datanya
2 jika list kosong, maka :
3 jadikan node baru sebagai head
4 next dari node baru -> kosong
5 jika tidak kosong:
6 tampung head pada variabel temporary
7 loop variabel temporary hingga posisi paling belakang
8 selesai loop, referensikan next node paling belakang ke node baru
9 next dari node baru -> kosong
10 selesai

Implementasinya adalah sebagai berikut.

```
void slist_pushBack(SinglyList *list, int value)
{
    SListNode *newNode = (SListNode*) malloc(sizeof(SListNode));
    if (newNode) {
        list->_size++;
        newNode->data = value;
        newNode->next = NULL;

        if (slist_isEmpty(list))
            list->_head = newNode;
        else {
            SListNode *temp = list->_head;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = newNode;
        }
    }
}
```

F.4 pushFront

Operasi *pushFront* lebih mudah dilakukan. Caranya adalah sebagai berikut.

1. buat *node* baru dan isikan datanya
2. jika list kosong:
3. maka next dari *node* baru -> kosong
4. jika tidak:
5. next dari *node* baru adalah *head*
6. jadikan *node* baru sebagai *head*

Implementasi *pushFront*.

```
void slist_pushFront(SinglyList *list, int value)
{
    SListNode *newNode = (SListNode*) malloc(sizeof(SListNode));
    if (newNode) {
        list->_size++;
        if (slist_isEmpty(list)) newNode->next = NULL;
        else newNode->next = list->_head;

        newNode->data = value;
```

```
        list->_head = newNode;
    }
}
```

F.5 insertAt

F.6 popBack

F.7 popFront

F.8 remove(x)

F.9 find(x)

F.10 front

F.11 back

F.12 getAt

F.13 isEmpty

Untuk memeriksa apakah list kosong, dapat dilakukan dengan memeriksa *head*-nya.

1. jika *head* kosong (NULL):
2. maka return true
3. jika *head* terdapat nilai:
4. maka return false

Stack

A. Definisi

B. Ilustrasi

C. Operasi Dasar

blablabla

Operasi	Keterangan	Kompleksitas Waktu
push	Memasukkan data baru pada stack.	$O(1)$
pop	Mengeluarkan data paling atas dari stack.	$O(1)$
top	Mendapatkan data paling atas.	$O(1)$
isEmpty	Memeriksa apakah stack kosong atau tidak.	$O(1)$

D. Implementasi ADT: Stack (Array Based)

Soal Latihan

E. Implementasi ADT: Stack (Linked List Based)

E.1 push

E.2 pop

E.3 top

E.4 isEmpty

Queue

A. Definisi

B. Ilustrasi

C. Operasi Dasar

Operasi	Keterangan	Kompleksitas Waktu
push	Memasukkan data baru pada queue.	$O(1)$
pop	Mengeluarkan data paling depan dari queue.	$O(1)$
front	Mendapatkan data paling depan.	$O(1)$
isEmpty	Memeriksa apakah queue kosong atau tidak.	$O(1)$

D. Implementasi ADT: Queue (Array Based)

Soal Latihan

E. Implementasi ADT: Queue (Linked List Based)

E.1 push

E.2 pop

E.3 front

E.4 isEmpty

Priority Queue

A. Definisi

B. Ilustrasi

C. Operasi Dasar

D. Implementasi ADT: Priority Queue (Linked List Based)

Soal Latihan

E. Notes