# Implementasi Struktur Data


**LABORATORIUM PRIDE**

**KELOMPOK KEAHLIAN PROGRAMMING**

**FAKULTAS ILMU TERAPAN**

**UNIVERSITAS TELKOM**

## DAFTAR PENYUSUN

Rizza Indah Mega Mandasari, S.Kom, M.T

Cahyana, S.T., M.Kom

## LEMBAR REVISI

| No | Keterangan Revisi | Tanggal Revisi Terakhir |
|---|---|---|

| | | |
|---|---|---|
| 1 | Revisi Bagian Pertama | |
| 2 | Revisi Bagian Kedua | |
| | | |

LEMBAR PERNYATAAN

Saya yang bertanggung jawab di bawah ini:

Nama : Rizza Indah Mega Mandasari, S.Kom., M.T

NIP : 15880031

Dosen PJMP : Implementasi Struktur Data

Kelompok Keahlian : Interactive System

Menerangkan dengan sesungguhnya bahwa modul ini telah direview dan akan digunakan untuk pelaksanaan praktikum di Semester Genap Tahun Ajaran 2016/2017 di Laboratorium Pride Fakultas Ilmu Terapan Universitas Telkom

Bandung, 29 Desember 2016

Mengetahui,

| | |
|---|---|
| Ketua Kelompok Keahlian | Dosen PJMP |
| Hariandi Maulid, S.T., M.Sc | Rizza I. M. Mandasari, S.Kom., M.T |
| NIP | NIP 15880031 |

# DAFTAR ISI

# DAFTAR GAMBAR

# DAFTAR PROGRAM

No table of figures entries found.

DAFTAR TABEL No table of figures entries found.

# Modul 0 : Running Modul

## 0.1 Tujuan

Setelah mengikuti Running Modul mahasiswa diharapkan dapat:

1. Memahami peraturan kegiatan praktikum.
2. Memahami Hak dan Kewajiban praktikan dalam kegiatan praktikum.
3. Memhami komponen penilaian kegiatan praktikum.

## 0.2 Peraturan Praktikum

1. Praktikum diampu oleh **Dosen Kelas** dan dibantu oleh **Asisten Laboratorium** dan **Asisten Praktikum**.
2. Praktikum dilaksanakan di Laboraturium Komputer Gedung FIT lantai 2 sesuai jadwal yang ditentukan.
3. Praktikan wajib membawa **modul praktikum, kartu praktikum, dan alat tulis**.
4. Praktikan wajib mengisi **daftar hadir** dan **BAP praktikum** dengan bolpoin **bertinta hitam**.
5. Durasi kegiatan praktikum **D3 = 4 jam (200 menit)**.
    a. 15 menit untuk pengerjaan Tes Awal atau wawancara Tugas Pendahuluan
    b. 60 menit untuk penyampaian materi
    c. 125 menit untuk pengerjaan jurnal dan tes akhir
6. Jumlah **pertemuan praktikum**:
    - 11 kali di lab (praktikum rutin)
    - 1 kali responsi (terkait Tugas Besar dan/atau UAS/COTS)
    - 1 kali berupa presentasi Tugas Besar dan/atau pelaksanaan UAS/COTS
7. Praktikan **wajib hadir minimal 75%** dari seluruh pertemuan praktikum di lab. Jika total kehadiran kurang dari 75% maka nilai COTS/UAS/ Tugas Besar = 0.
8. Praktikan yang datang terlambat :
    - <= 30 menit : diperbolehkan mengikuti praktikum tanpa tambahan waktu Tes Awal
    - > 30 menit : tidak diperbolehkan mengikuti praktikum 9. Saat praktikum berlangsung, asisten praktikum dan praktikan:
    - Wajib menggunakan **seragam** sesuai aturan Institusi.
    - Wajib mematikan/ men-silent semua **alat komunikasi**(smartphone, tab, iPad, dsb).
    - Dilarang membuka **aplikasi yang tidak berhubungan** dengan praktikum yang berlangsung.
    - Dilarang mengubah **setting software maupun hardware** komputer tanpa ijin.
    - Dilarang **membawa makanan maupun minuman** di ruang praktikum.
    - Dilarang **memberikan jawaban ke praktikan lain** (pre-test, TP, jurnal, dan post-test).
    - Dilarang **menyebarkan soal pre-test, jurnal, dan post-test**.
    - Dilarang**membuang sampah/sesuatu apapun** di ruangan praktikum.

10. Setiap praktikan dapat mengikuti praktikum susulan maksimal 2 modul untuk satu praktikum.

- Praktikan yang dapat mengikuti praktikum susulan hanyalah praktikan yang memenuhi syarat sesuai ketentuan Institusi, yaitu rawat inap di Rumah Sakit (menunjukkan bukti rawat inap dan resep obat dari RS), tugas dari Institusi (menunjukkan surat dinas dari Institusi), atau mendapat musibah (menunjukkan surat keterangan dari orangtua/ wali mahasiswa).
- Persyaratan untuk praktikum susulan diserahkan sesegera mungkin ke Asisten Praktikum untuk keperluan administrasi.

11. Pelanggaran terhadap peraturan praktikum ini akan ditindak secara tegas secara berjenjang di lingkup Kelas, Laboratorium, Program Studi, Fakultas, hingga Institusi.

## 0.3 Penilaian Praktikum

1. Komponen penilaian praktikum:
    - 20**%** nilai Tugas Pendahuluan
    - 20% nilai Tugas Awal
    - 50% Jurnal
    - 10% Skill
2. Seluruh komponen penilaian beserta pembobotannya ditentukan oleh dosen **PJMP**
3. Penilaian permodul dilakukan oleh **asisten praktikum**, sedangkan nilai Tugas Besar/ UAS diserahkan kepada **dosen kelas,** dilaporkan ke **PJMP.**
4. Baik praktikan maupun asisten tidak diperkenankan meminta atau memberikan **tugas tambahan** untuk perbaikan nilai.
5. Standar **indeks dan range nilai** ditentukan oleh dosen PJMP atas sepengetahuan Ketua Kelompok Keahlian

# Modul 1 : Pengantar Struktur Data

## 1.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal struktur data
2. Menginstal IDE IntelliJ IDEA
3. Mengenal kosep Array dan Arraylist

## 1.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 1.3 Dasar Teori

### 1.3.1 Algoritma dan Struktur Data

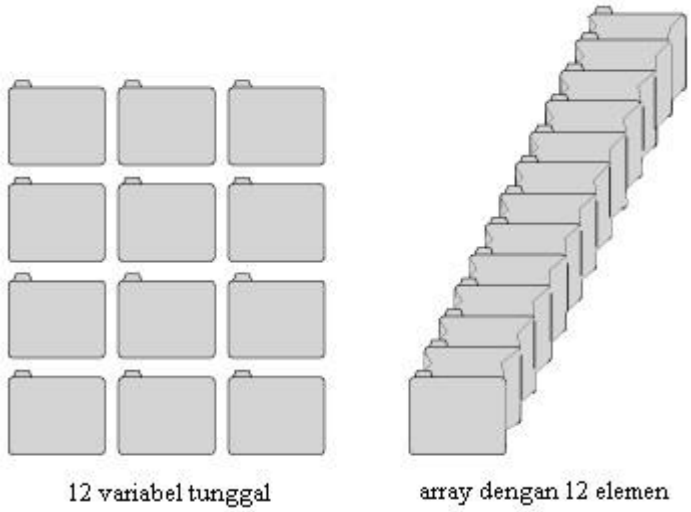
Istilah struktur data tidak dapat terpisah dari algoritma. Jika algoritma adalah suatu langkah atau prosedur yang ditujukan untuk memproses data, maka struktur data adalah bagaimana pengaturan data tersebut dimemori atau disk secara efisien saat akan dilakukan proses pengolahan data tersebut. Saat membuat sebuah algoritma, yang paling ditekankan adalah bagaimana membuat algoritma yang cepat, dan optimum. Dengan pemilhan struktur data yang tepat maka running time terhadap algoritma tersebut akan lebih baik.

*Djikstra Algorithm adalah algoritma yang digunakan untuk menemukan rute terpendek, atau masalah optimasi lintasan terpendek. Tanpa penggunaan struktur data yang baik running time dari Djikstra (impelentasi dengan Adjacency matrix) akan memakan waktu berkali-kali lipat (**$O(N)^2$**) namun dengan pemilihan struktur data yang tepat (implementasikan dengan priority queue) maka, running time yang dapat dimaksimalkan (**$O(N*log^n)$**).*

## 1.4 Array

### 1.4.1 Apa itu Array?

Array merupakan sekelompok data sejenis yang disimpan ke dalam suatu variabel yang mana variabel tersebut memiliki indeks untuk pengaksesan datanya. Berikut contoh ilustrasi antara variabel dengan array.




**Gambar 1.1 Ilustrasi Array**

Jika kita lihat pada gambar yang pertama, kita mendeklarasikan 12 buah variabel dengan nama yang berbeda tentunya untuk tujuan yang sama, misalnya untuk menyimpan total pengeluaran setiap bulan atau menyimpan identitas dari mahasiswa. Sekarang kita lihat gambar yang kedua, kita mendeklarasikan sebuah array dengan 12 element dan fungsinya sama untuk menyimpan total pengeluaran setiap bulan. Jika kita bandingkan tentunya akan lebih mudah jika kita mempergunakan array karena kita hanya mempergunakan satu buah variabel dalam prosesnya nanti, dan juga dalam implementasi ke dalam program kita tidak mungkin akan mendeklarasikan variabel sebanyak seribu buah untuk tujuan yang sama namun dengan menggunakan array untuk mendeklarasikan variabel sebanyak itu akan dapat dilakukan dengan mudah.

### 1.4.2 Deklarasi

Secara umum bentuk pendeklarasian tipe data array pada bahasa Java dapat dilakukan dengan format sebagai berikut.

```
Tipe_Data_Array  Nama_Array  [ukuran]
```

Layaknya variable, sebuah array juga memiliki tipe data dan perlu diperhatikan suatu array hanya bisa memiliki data yang bertipe sama saja. Tipe data dari suatu array bisa berupa integer, float, char, bahkan tipe objek. Tanda kurung [] pada pendeklarasian array di atas digunakan untuk menunjukkan jumlah elemen larik. Jika dideklarasikan berupa int x[10], maka x merupakan array yang berisi 10 elemen dengan tipe data integer. Selain itu, yang perlu diperhatikan lagi bahwa penghitungan elemen dari suatu array dimulai dari 0, bukan 1.

Namun, perlu diperhatikan bahwa pada Java, deklarasi array tidak otomatis menjadikan variabel array tersebut dapat digunakan, karena pada saat deklarasi hanya nama array saja yang terbentuk, sementara objek array belum terbentuk. Saat deklarasi, nilai array masih null, yang

artinya tidak ada objek array tersebut. Agar terbentuk suatu objek array, perlu mengalokasikan array tersebut dengan keyword new. Perhatikan kode berikut.

```
Tipe_Data_Array  Nama_Array  [ukuran]; array-var
= new type[size]
```

Pada kode program diatas, setelah melakukan deklarasi array, dilakukan alokasi pada variabel array tersebut dengan menggunakan new. Type pada bagian new harus memiliki tipe data yang sama dengan tipe data pada saat deklarasi array. Jika tidak dilakukan inisialiasi, otomatis akan diisi dengan nilai default masing-masing tipe data (misal, untuk tipe integer akan diisi nol). Jadi, pembuatan array pada Java membutuhkan dua langkah; pertama deklarasi variabel array, kemudian alokasikan memori untuk variabel tersebut dengan menggunakan keyword new. Karena itu, proses alokasi ini bisa saja dilakukan pada baris program yang terpisah jauh dengan baris program deklarasi array.

### 1.4.3 Inisialisasi Array

Inisialisasi array dapat dilakukan langsung saat deklarasi array, dengan memasukkan serangkaian nilai yang dipisahkan oleh koma pada dalam kurung kurawal. Koma berfungsi memisahkan nilai antar elemen array. Jika melakukan insialisasi pada saat deklarasi, maka keyword new tidak diperlukan. Perhatikan contoh berikut.

```
package praktikum1;


class InitArray {
   public static void main(String args[]) {
int jum_hari[] =
                    {31,28,31,30,31,30,31,31,30,31,30,31};
     System.out.println("Februari memiliki " + jum_hari[1] +
" hari.");


     //deklarasi array terpisah
String nama_hari;
nama_hari = new String[7];
nama_hari[0] = "Senin";
nama_hari[1] = "Selasa";
     }
 }
```

Apakah Outputnya?

……

……

……

Pada program diatas, terdapat dua cara deklarasi dan inisialisasi array. Yang pertama adalah dengan mendeklarasikan dan sekaligus melakukan inisialisasi array (pada array jum_hari), sementara pada array nama_hari, deklarasi dilakukan terpisah dengan inisialisasi. Inisialisasi pada array nama_hari hanya dilakukan pada elemen pertama dan kedua aarray. Selain itu, inisialisasi array juga dapat dilakukan dengan menggunakan perulangan. Perhatikan contoh berikut.

```
package praktikum2;

import.java.util.Scanner;

class DemoArray {
   public static void main(String args[]){
int[] a = new int[100];
      Scanner in = new Scanner (System.in);


      System.out.println("Masukkan banyaknya nilai : ");
int x = in.nextInt(); //menerima input dari console


      for (int i = 0; i < x; i++) {
         System.out.println("Input angka ke - "+(i+1)+ " :" );
a[i] = in.nextInt();
       }


      System.out.println("Angka yang di-input-kan: " );


      for (i = 0; i < x; i++) {
System.out.println(a[i]);
      }
   }
}
```

Apakah Outputnya?

……

……

……

Perhatikan bahwa deklarasi array dapat dilakukan dengan meletakkan kurung siku pada tipe data (bukan hanya pada nama variabel). Untuk menerima input dari console, digunakan Scanner. Karena Scanner merupakan objek dari kelas util dari java (bukan pada kelas yang sedang dijalankan) maka Scanner tersebut harus di-import dengan perintah import setelah nama package. Berikut contoh lain program yang menggunakan array.

```
package praktikum3;

import.java.util.Scanner;

class AnotherArray{
   publlic static void main (String args[]) {
int a[] = new int[100];       int x, sum,
ganjil, genap;
      Scanner in = new Scanner (System.in);
      System.out.println("Masukkan banyaknya nilai: ");
x = in.nextInt();
      sum = ganjil = genap = 0;


      for (int i = 0; i < x; i++) {
        System.out.println("Input angka ke- "+(i+1)+":");
a[i] = in.nextInt();           sum += a[i];           if
(a[i] % 2 == 0)             genap += 1;           else
            ganjil+=1;
      }
      System.out.println("Angka yang di-input-kan: " );
for (i = 0; i < x; i++) {
System.out.println(a[i]);       }
```

Apakah Outputnya?

……

……

……

……

……

……

```
        System.out.println("Total bilangan : " + sum );
        System.out.println("Jumlah bilangan ganjil : " + ganjil );
        System.out.println("Jumlah bilangan genap : "+ genap);

    }
}
```

## 1.5 ArrayList

ArrayList merupakan bagian dari data collection yang disediakan oleh Java API untuk menyimpan grup dari objek yang saling berhubungan. Pengaturan data di dalam ArrayList sama seperti pada array, dimana objek dengan tipe data yang sama disimpan secara berurutan. Namun, kita tidak perlu membuat kode program untuk pengaturan data tersebut, karena sudah disediakan oleh Java. Selain itu, dengan menggunakan ArrayList, alokasi data dapat dilakukan secara dinamis (dapat ditambah dan dikurangi, jika menggunakan array tidak bisa). Adapun cara deklarasi ArrayList adalah sebagai berikut.

```
ArrayLIst<T> nama_variabel = new ArrayList<T>
```

Perintah <T> merupakan tipe data yang digunakan untuk ArrayList. Misal, jika ingin membuat suatu ArrayList dari String, dituliskan ArrayList<String>. Seperti pada array, keyword new digunakan untuk mengalokasikan ArrayList pada memori. Method-method yang akan digunakan dalam ArrayList dapat dilihat pada Tabel 1-1.

**Table 1-1 Method pada ArrayList**

| Method | Keterangan |
|---|---|
| **add** | Menambahkan elemen di ujung AraryList |
| **get** | Mengambil elemen pada indeks tertentu dalam ArrayList |
| **remove** | Menghapus elemen yang pertama kali ditemukan pada ArayList |
| **size** | Mengembalikan besar ArrayList |

Method yang dipelajari pada praktikum ini terbatas untuk add, get dan remove. Penjelasan lebih mendalam mengenai ArrayList akan dipelajari pada mata kuliah Implementasi Struktur Data. Karena ArrayList berada pada kelas lain, maka harus di-import terlebih dahulu. Untuk lebih jelasnya, lihat contoh kode berikut ini.

```
Package praktikum4;

import java.util.ArrayList; // import ArrayList

class DemoArrayList {
   public static void main( String[] args ) {
       ArrayList<String> baju = new ArrayList<String>();
```

Apakah Outputnya?

……

……

```
       baju.add( seragam");
baju.add("gaun");          baju.add(0,"kaos
kaki");

       /*tampilkan isi array*/
       for ( int i=0; i < items.size(); i++ )
System.out.print (baju.get( i ) + " " );

       baju.add("gaun"); //menambahkan gaun diujung array
baju.remove( "gaun" ); // menghapus elemen gaun pertama yang
ditemui
       /*tampilkan isi array*/
       for ( int i=0; i < items.size(); i++ )
System.out.print (baju.get( i ) + " " );
       }
}
```

……

Perintah add akan menambahkan elemen pada ujung array. Namun, jika ditentukan indeks spesifik, seperti pada perintah add (0, "kaos kaki") maka elemen akan diletakkan pada tempat yang ditunjukkan oleh indeks (dalam hal ini, indeks pertama array). Perintah remove hanya menghapus elemen "gaun" yang ditemukan pertama kali.

## 1.6 Generic Types

Generic type adalah sebuah Class atau interface umum yang parameternya dapat diganti type data apapun selain type data primitive. Contohnya dapat dilihat pada code dibawah ini.

```
public class Pair<KeyType, ValueType> {

    private final KeyType key;
private final ValueType value;

    public Pair(KeyType key, ValueType value)
{          this.key = key;          this.value
= value;
    }
    public KeyType getKey() {
return key;
    }
    public ValueType getValue() {
return value;
    }
    public String toString() {
        return "(" + key + ", " + value + ")";
    }
}
```

Dalam penggunaannya Generic Class dapat digunakan seperti contoh dibawah ini:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

**Case!**

Kita dapat menambahkan value ke dalam array selama array tersebut tidak penuh. Selain itu kita dapat menambahkan value ke index tertentu didalam array. Bagaimana jika kita ingin menambahkan value ke dalam index yang telah berisi?

Contoh: add(23,1)

| | |
|---|---|
| 10 | 0 |
| 8 | 1 |
| -11 | 2 |
| 9 | 3 |
| 3 | 4 |
| | 5 |
| | 6 |

# Modul 2 : Singly Linked List

## 2.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep linked list
2. Mengenal konsep singly link list

## 2.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC

## 2.3 Dasar Teori

### 2.3.1 Abstrack Data Type

Abstract data type (ADT) adalah spesifikasi suatu set data dan operasi yang dilakukan pada data tersebut. Spesifikasi ini tidak menyebutkan mengenai bagaimana cara menyimpan data, ataupun implementasinya. Contohnya adalah data mahasiswa dibawah ini:

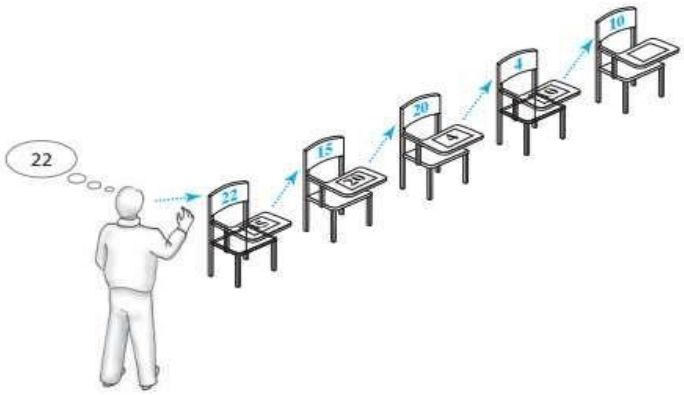
| Abstrak Data Type: Mahasiswa |
|---|
| Data:<br>Nama, NIM, Kelas, IPK, dll. |

| Operation | | |
|---|---|---|
| Pseudocode | UML | Description |
| isEmpty() | + isEmpty() | Task: Cek apakah mahasiswa kosong.<br>Input: -<br>Output: True or false berdasarkan data kosong atau tidaknya. |
| AddFirst(newEntry) | + AddFirst(newEntry) | Task: Menambahkan data mahasiswa didepan list.<br>Input: newEntry object<br>Output: True or false berdasarkan berhasil atau tidaknya penambahan data. |
| Remove(newEntry) | + Remove(newEntry) | Task : Menghapus data mahasiswa<br>Input : newEntry object<br>Output: True or false berdasarkan berhasil atau tidaknya penghapusan data. |
| Clear() | + Clear() | Task : Menghapus seluruh data mahasiswa<br>Input : -<br>Output: True or false berdasarkan berhasil atau tidaknya penghapusan seluruh data. |
| TraverseList | + TraverseList() | Task : Menyusuri seluruh data mahasiswa. |

Semakin detail spesifikasi ADT yang diberikan maka, semakin kompleks dan semakin detail method yang akan digunakan dibahasa pemrograman. Spesifikasi operasi terhadap ADT dapat dikumpulkan menjadi sebuah interface yang mengimplementasikan ADT dalam Bahasa Java. Contoh interface Java yang berisi method untuk ADT diatas dapat dilihat pada code dibawah ini:

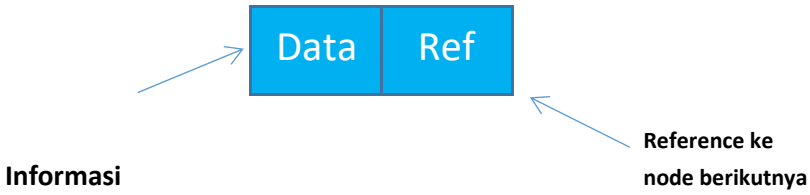
```
public interface ListMahasiswa<T> {

    public boolean isEmpty();
/** Cek apakah list kosong.
     @return true jika kosong, or false tidak */

    public boolean addFirst(T newEntry);
    /** Menyisipkan node didepan list.
     @param newEntry adalah object yang akan ditambahkan
     @return true jika berhasil ditambahkan, or false jika tidak */

    public boolean addLast(T newEntry);
    /** Menyisipkan node didepan list.
     @param newEntry adalah object yang akan ditambahkan
     @return true jika berhasil ditambahkan, or false jika tidak */

    public T findData(String nama);
    /** Menelusuri setiap node didalam list.
     @param nama adalah salah satu data di dalam list
     @return seluruh data dalam node tsb jika ditemukan, null jika tidak
ditemukan */

    public int getPositionOf(String nama);
/** Menelusuri setiap node didalam list.
     @param nama adalah salah satu data di dalam list
     @return posisi node tsb dalam list, -1 jika tidak ditemukan */

    public T remove(int givenPosition);
    /** Remove salah satu node di posisi tertentu, jika memungkinkan.
     // @param posisi dari node tsb
     @return data dalam node yg dihapus, null jika tidak berhasil di-remove
*/
    public void clear();
    /** Remove semua node dilist. */

    public boolean contains(T anEntry);
    /** Tes apakah node tersebut memiliki data sesuai dengan anEntry
     // @param anEntry data yang ingin dicari
     @return true jika ditemukan anEntry, or false jika tidak ditemukan */

    public void traverseList();
    /** Menyelusuri setiap node didalam list dan menampilkan data yang
menjadi isi node. */
}
```

### 2.3.2 Linked List



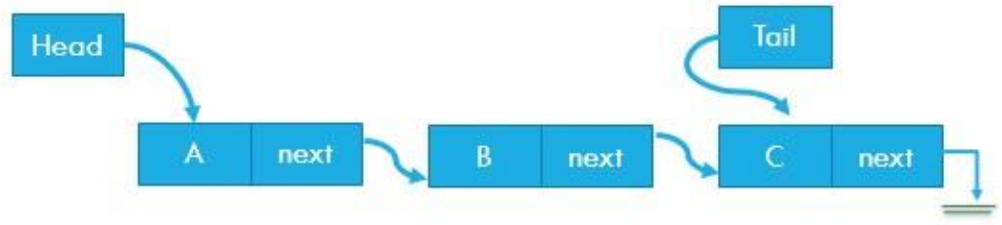

Gambar 2.1 Ilustrasi Linked List (Source: Carrano, 2001)

Linked List (biasa disebut list saja) adalah salah satu bentuk struktur data (representasi penyimpanan) berupa serangkaian elemen data yang saling berkait (berhubungan) tersusun secara linear dengan masing-masing disimpan dalam sebuah Node(simpul). Data yang disimpan dalam sebuah node terdiri atas dua bagian. Pertama adalah bagian yang menyimpan data, sedangkan bagian yang kedua menyimpan referensi atau penunjuk pada node berikutnya. Data pada Linked List dapat berupa data tunggal atau data majemuk.




Gambar 2.2 Linked List

Pada pembahasan kali ini yang akan dibahas adalah singly linked list. Singly linked list merupakan linked list dimana setiap node **hanya** menyimpan referensi terhadap node selanjutya. Sebuah linked list minimal harus memiliki referensi pada Node pertama (head) di list tersebut. Tanpa sebuah referensi yang merujuk ke head, tidak akan ada cara utuk menemukan node dan rangkaiannya.




Gambar 2.3 Singly Linked List dengan 3 Node

Node terakhir di list dikenal dengan istilah tail. Tail dapat ditemukan dengan cara *traversing* linked list tersebut. Traversing (menyusuri) adalah melintasi setiap Node dimulai dari Head menuju node selanjutnya mengikuti referensi dari setiap node hingga ditemukan tail. Tail dapat diidentifikasi sebagai sebuah node yang memiliki Null value untuk referensi selanjutnya. Definisi singkat dari Node dapat dilihat dibawah ini.

```
private class Node {
        private T data; // entry in bag
        private Node next; // link to next node

        private Node(T dataPortion) {
this(dataPortion, null );         }


        private Node(T dataPortion, Node nextNode)
        {
            data = dataPortion;
next = nextNode;         } //
end constructor


        private T getData()
        {
            return data;
} // end getData

    private void setData(T newData)
        {
            data = newData;
        } // end setData


        private Node getNextNode()
        {
            return next;
} // end getNextNode

        private void setNextNode(Node nextNode)
        {
            next = nextNode;
} // end setNextNode

        @Override
        public String toString() {
return this.data.toString();
        }
}
```

Karena Node dibuat menjadi Inner Class, maka generic type T akan sama dengan jenis generik dinyatakan oleh outer class dari Node tesebut. Dengan demikian, kita tidak menulis <T> setelah Node, namun, jika Node bukan inner class tetapi memiliki akses **package** atau **public**, maka harus dituliskan Node <T>. T dapat diganti sesuai dengan ADT yang telah dibuat. (Lihat Modul 1.6)

Untuk implementasinya, kita akan menggunakan singly linked list berisi Node diatas untuk menghubungkan antara Node satu dan Node lainnya. Pada implementasi ini harus “mengingat” Node pertama(firstnode) dari linked list ini.

2.4

```
public class LinkedMahasiswa<T> implements MahasiswaInterface<T>
{      private Node firstNode; // reference to
firstnode     private Node lastNode; // reference to
lastnode      private int numberOfEntries;

    public LinkedMahasiswa()
    {
        firstNode = null ;
        numberOfEntries = 0;
    }

<< Implementasi dari public methods yang dideklarasikan di Interface. >>

    private class Node // private inner class
    {
      < Lihat implementasi Node diatas >
    } // end Node
} // end
```

## Operasi Dasar

Terdapat beberapa macam operasi dasar terhadap ADT pada Linked list. Secara umum operasioperasi tersebut adalah:

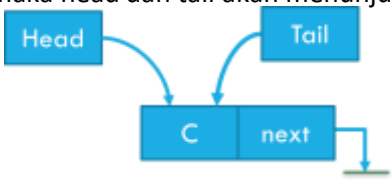
1. Penyisipan Node (Insert)
2. Penghapusan Node (Delete)
3. Penelusuran Node dan menampilkan isi Node (Traversing)
4. Pencarian Node (Searching)
5. Pengubahan isi Node (Update)

### 2.4.1 Penyisipan Node (Insert)

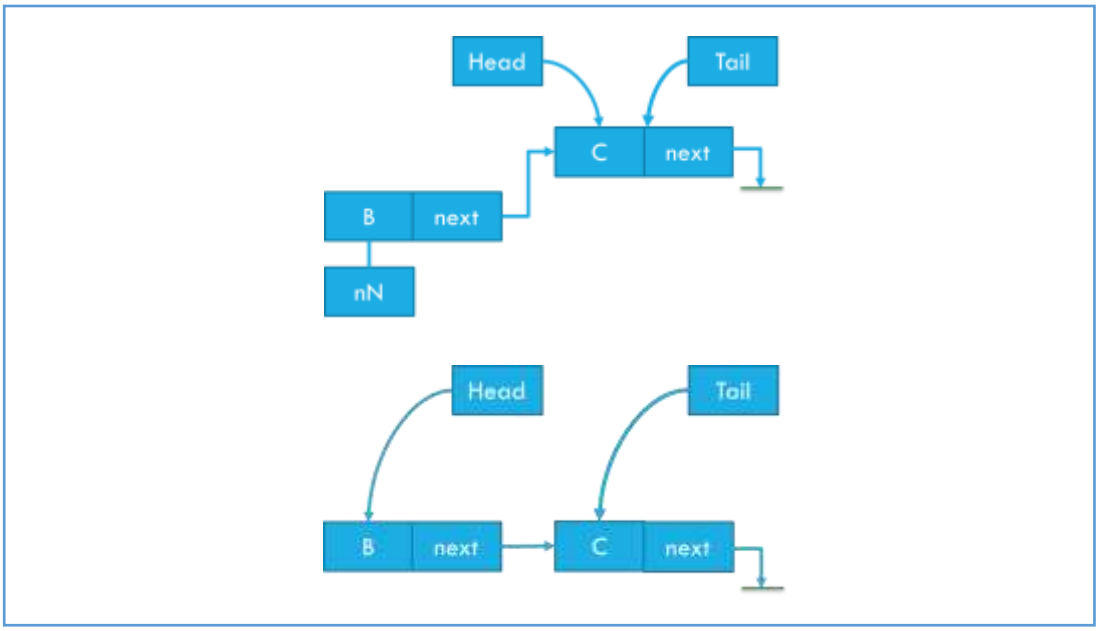
Penyisipan Node pada list dapat dilakukan didepan, dibelakang ataupun diantara Node-node tertentu. Pada penyisipan didepan, langkah-langkah dapat dilihat dibawah ini:

1. Ketika List kosong maka head/ firstnode dan tail/lastnode menunjuk null


2. Ketika disisipkan node baru, maka head dan tail akan menunjuk node tersebut.




3. Ketika kemudian akan disisipkan node baru di depan linked list yang ada, maka newNode next akan menunjuk head. Dan head menunjuk pada newnode.

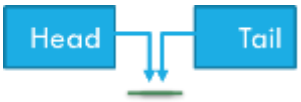



**Gambar 2.4 Insert Depan Singly Linked List**

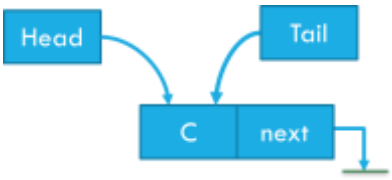
```
public boolean addFirst(T newEntry) {
// add to beginning of chain:
Node newNode = new Node(newEntry);
newNode.setNextNode(firstNode);

        // (firstNode is null if chain is empty)
if (isEmpty()) {
            firstNode = newNode; // new node is at beginning of chain
lastNode = newNode;
        }else{
            newNode.setNextNode(firstNode);
firstNode = newNode;
        }
        numberOfEntries++;
return true;
    }
```

Sedangkan untuk lngkah-langkah pada penyisipan belakang dapat dilihat dibawah ini:

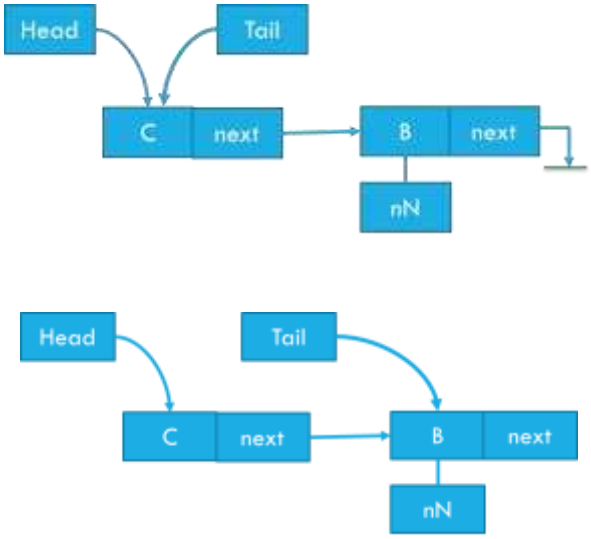
1. Ketika List kosong maka head/ firstnode dan tail/lastnode menunjuk null




2. Ketika disisipkan node baru, maka head dan tail akan menunjuk node tersebut.



3. Ketika kemudian akan disisipkan node baru di depan linked list yang ada, maka tail next akan menunjuk newNode. Dan tail akan menunjuk newNode.




**Gambar 2.5 Insert Belakang Singly Linked List**

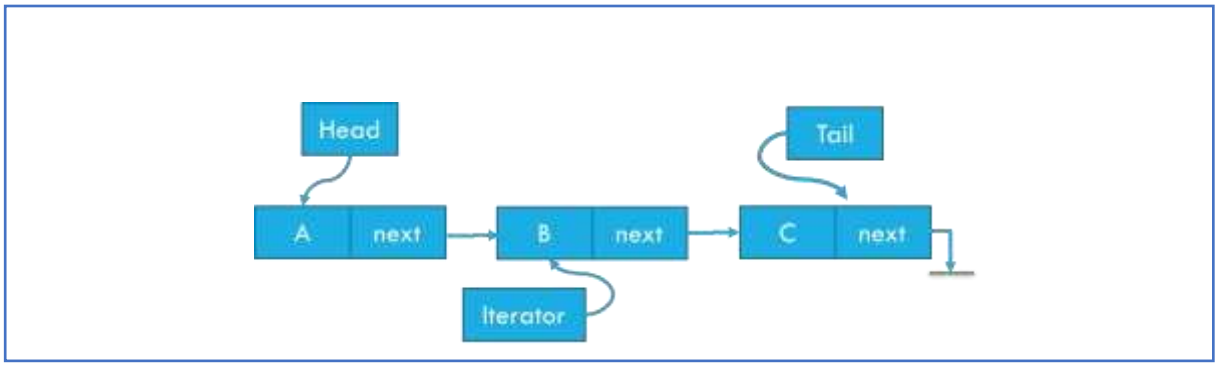
```
    public boolean addLast(T newEntry) {
        Node newNode = new Node(newEntry);


lastNode.setNextNode(newNode);
lastNode = newNode;
numberOfEntries++;          return
true;
    }
```

### 2.4.2 Pencarian (Searching)

Searching merupakan operasi dasar list dengan melakukan aktivitas pencarian terhadap node tertentu. Proses ini berjalan dengan mengunjungi setiap node dan berhenti setelah node yang dicari ketemu. Dengan melakukan operasi searching, operasi-operasi seperti insert after, delete after, dan update akan lebih mudah.

1. Set sebuah iterator pada head.


2. Lakukan perbandingan data yang dikunjungi iterator, bila ditemukan maka kembalikan nilai pada node tersebut, jika tidak ditemukan lakukan step 3.
3. Pindahkan iterator pada node selanjutnya. Lakukan step 2 hingga ditemukan atau iterator telah mencapai akhir list. (iterator mencapai tail)




**Gambar 2.6 Searching pada Link List**

```
public T findData(String nama)
{     boolean found = false;
Node iterator = firstNode;
    while (!found && (iterator != null ))
    {
        Mahasiswa mahasiswa = (Mahasiswa)iterator.data;
if (nama.equals(mahasiswa.getNama())) {
found = true;
            return iterator.data;
        } else {
            iterator = iterator.next;
        }
}
    return null;
} // end
```

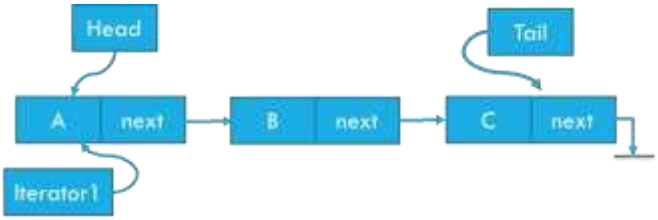
Selain menggunakan cara diatas, pencarian pun dapat dilakukan dengan mengunjungi setiap node, berhenti apabila node tersebut ditemukan dan mengembalikan posisi dari node tersebut. Posisi node tersebut dapat digunakan untuk parameter masukan dalam melakukan penghapusan maupun penyisipan.

```
public int getPositionOf(String nama){
assert (firstNode !=null); //
    Node current = firstNode;


    for (int i = 1; i < numberOfEntries; i++) { //numberOfEntries = size
                                  //dari link list tsb
        Mahasiswa mahasiswa = (Mahasiswa)
current.data;         if
(nama.equals(mahasiswa.getNama()))             return
i;         else
            current = current.next;
    }
    return -1;
}
```

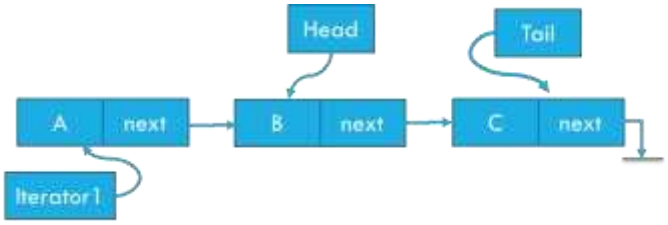
### 2.4.3 Penghapusan (Delete)

Sama seperti penyisipan Node, penghapusan Node pada list dapat dilakukan didepan, dibelakang ataupun diantara Node-node tertentu. Pada penghapusan sebuah Node disebelah depan, langkah-langkah dapat dilihat dibawah ini:

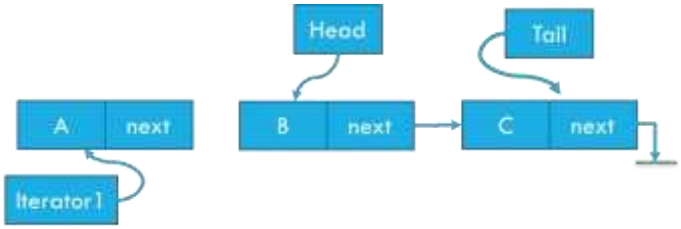
1. Cek apakah list tersebut kosong. Jika tidak kosong lanjutkan menuju step-2.
2. Jika tidak kosong, set Iterator pada Head atau firstNode.




3. Pindahkan Head pada node setelah Head.




4. Set Iterator menjadi null.

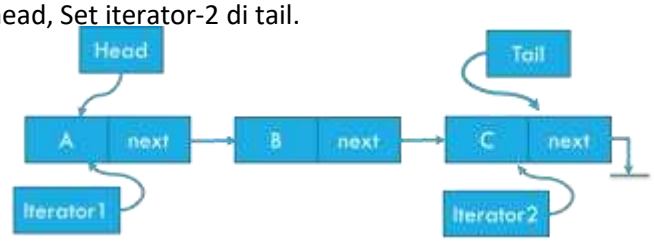



**Gambar 2.7 Singly Link List Hapus Depan**

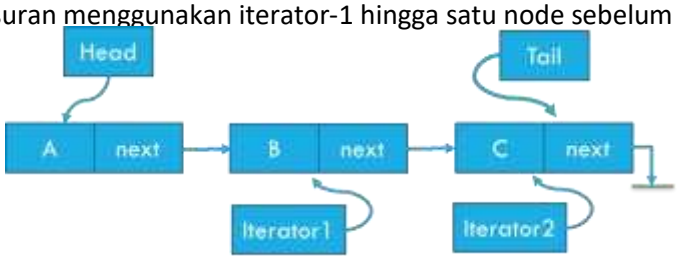
```
public T remove() {
assert !isEmpty();
  T iterator = null; // return value
          iterator = firstNode.getData(); // save entry to be
removed           firstNode = firstNode.getNextNode();           if
(numberOfEntries == 1)
              lastNode = null ; // solitary entry was removed
      }
      numberOfEntries--;
  return iterator; // return removed entry }
```

Sedangkan untuk melakukan hapus belakang, dibutuhkan dua buah iterator. Berikut langkah langkah menghapus node dibelakang.

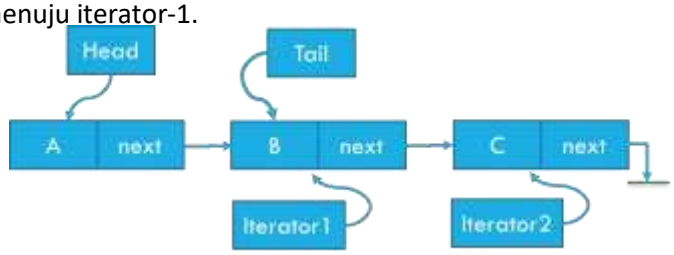
1. Cek apakah list kosong.
2. Set iterator-1 di head, Set iterator-2 di tail.



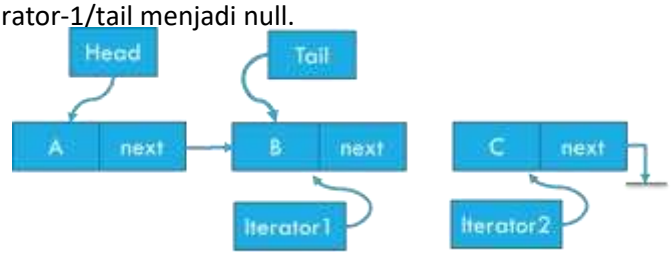

3. Lakukan penelusuran menggunakan iterator-1 hingga satu node sebelum tail.




4. Pindahkan tail menuju iterator-1.




5. Set reference iterator-1/tail menjadi null.




6. Kembalikan nilai iterator-2.

**Gambar 2.8 Singly Link List Hapus Belakang**

```
public T remove() {
    T result = null; // iterator 2 = result, return value
assert !isEmpty();
    Node iterator1 = firstNode;
    Node iterator2 = lastNode;
    While(iterator1.getNextNode() == lastNode){
iterator1 = iterator1.getNextNode;
    }     lastNode = iterator1;
iterator1.setNextNode(null);     result =
lastNode.getData(); // save entry to be removed
numberOfEntries--;
    }      return result; // return
removed entry }
```

### 2.4.4 Traversing dan Display

Traversing merupakan operasi dasar pada list yang menelusuri setiap Node dpada list dimulai dari Head menuju node selanjutnya mengikuti referensi hingga ditemukan tail/node terakhir. Traversing dapat

digunakan untuk melakukan pencarian, menampilkan semua node dalam list, dan menghapus semua node/mengosongkan list tersebut.

```
public void traverseList() {
if (this.firstNode == null) {
return;
    }

    Node node = this.firstNode;

    while (node != null) {
        System.out.println(node + " ");
node = node.getNextNode();
    }
}
```

## 2.5 Latihan

Buatlah ADT pegawai dan Singly Linked List yang mengimplementasikan ADT tersebut. Fungsi utama yang terdapat pada Linked List dengan ketentuan:

1. Input data pegawai
2. Melihat data pegawai
3. Menghapus data pegawai
4. Mengupdate data pegawai.
5. Menghapus seluruh data pegawai.

# Modul 3 : Doubly Linked List

## 3.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep linked list
2. Mengenal konsep doubly linked list

## 3.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 3.3 Doubly Linked List


**Gambar 3.1 Doubly Linked List with 3 Nodes**

Double Linked List adalah linked list yang masing – masing elemen nya memiliki dua reference, yaitu reference yang menunjuk pada elemen sebelumnya (*prev*) dan reference yang menunjuk pada elemen sesudahnya(*next*). Pada double linked list juga terdapat head dan tail, yaitu head (yang menunjuk node pertama pada list) dan tail (yang menunjuk node terakhir list). Definisi singkat dari Node dapat dilihat dibawah ini.

```
private class Node {
        private T data; // entry
        private Node next; // link to next node
private Node prev; // link to next node

        private Node(T dataPortion) {
this(dataPortion, null, null);
        }

        public Node(T data, Node next, Node prev) {
this.data = data;             this.next = next;
this.prev = prev;
        }

        private T getData()
        {
            return data;
} // end getData

        private void setData(T newData)
        {
            data = newData;
        } // end setData

        public Node getNext() {
return next;
        }

        public void setNext(Node next) {
this.next = next;
        }

        public Node getPrev() {
return prev;
        }
```

```
        public void setPrev(Node prev) {
this.prev = prev;
        }
    }
```

## 3.4 Operasi Dasar

Sama seperti Singly Linked List terdapat beberapa macam operasi dasar terhadap Doubly Linked list, secara umum operasi-operasi tersebut adalah:

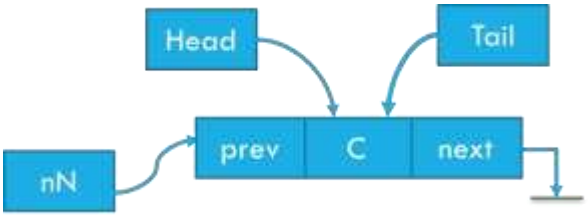
1. Penyisipan Node (Insert)
2. Penghapusan Node (Delete)
3. Penelusuran Node dan menampilkan isi Node (Traversing)
4. Pencarian Node (Searching)
5. Pengubahan isi Node (Update)

### 3.4.1 Penyisipan Node

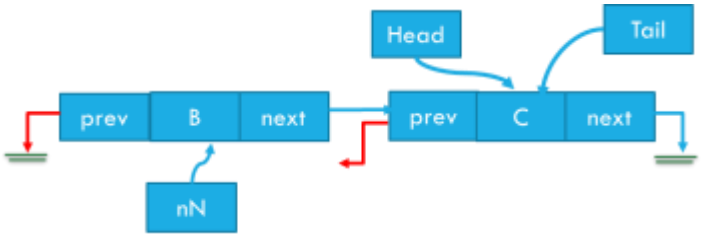
Penyisipan Node pada doubly list dapat dilakukan didepan, dibelakang ataupun diantara Node node tertentu. Berikut merupakan penyisipan node pada bagian awal tertentu.

1. Linked List masih kosong, head dan tail menunjuk ke NULL.


2. Ketika disisipkan node baru, maka head dan tail akan menunjuk node tersebut.

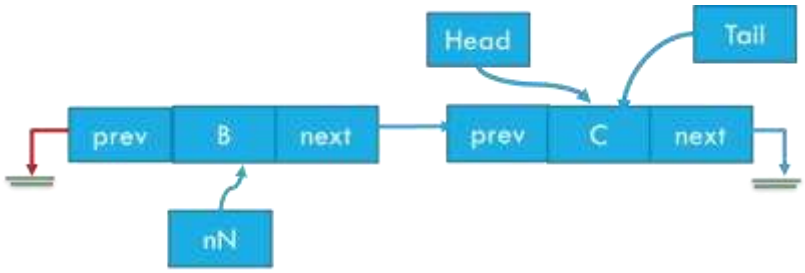



3. Ketika kemudian akan disisipkan node baru di depan linked list yang ada, maka : (1) newNode bagian next akan mengarah ke node yang ditunjuk oleh head

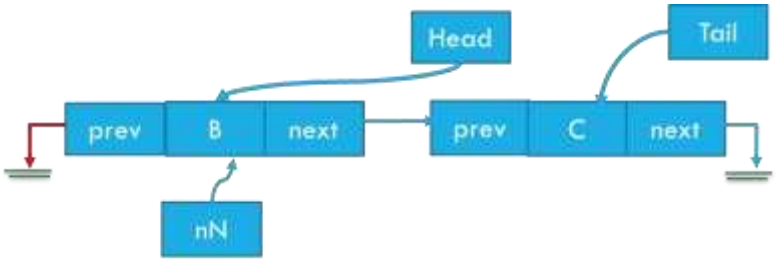



1. Linked List masih kosong, head dan tail menunjuk ke NULL.

(2) Head bagian previous mengarah ke newNode



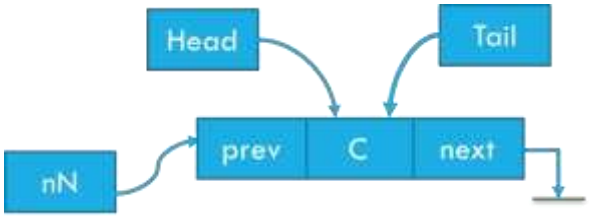

(3) Head pindah ke newNode



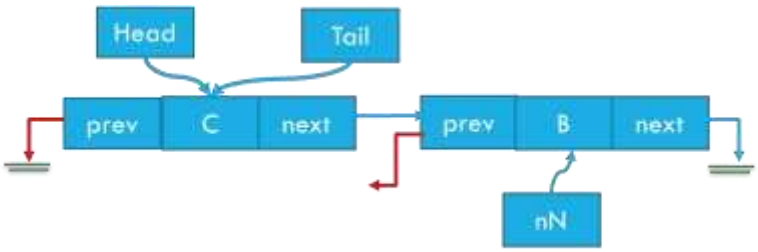

**Gambar 3.2 Insert Depan pada Doubly Linked List**

Sedangkan untuk penyisipan node pada bagian belakang dapat dilihat dilangkah-langkah berikut ini:

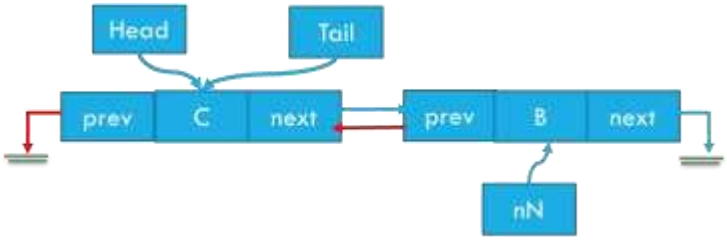

2. Ketika disisipkan node baru, maka head dan tail akan menunjuk node tersebut.



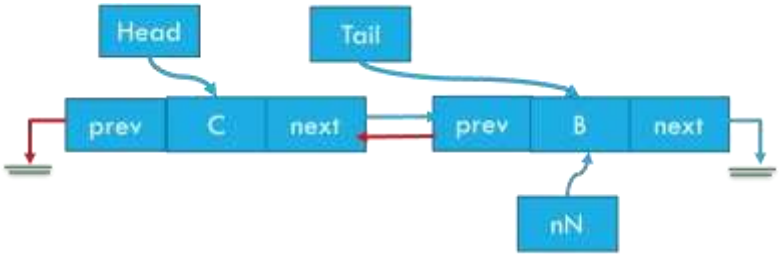

3. Ketika kemudian akan disisipkan node baru di depan linked list yang ada, maka : (i) tail bagian next akan mengarah ke node yang ditunjuk oleh newNode




(ii) newNode bagian previous mengarah ke tail



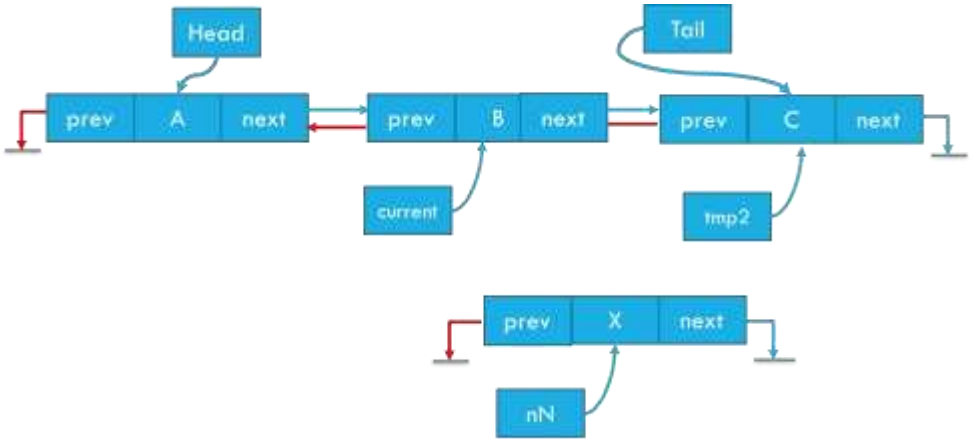

(iii) tail pindah ke newNode



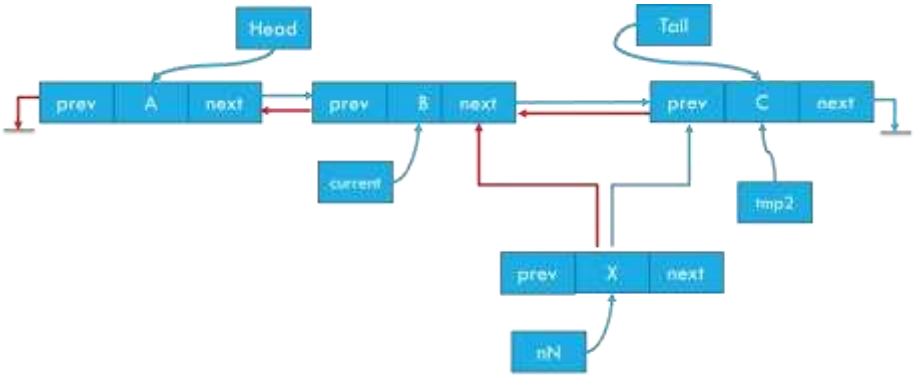

**Gambar 3.3 Insert Belakang pada Doubly Linked List**

Penyisipan node ditengah pada doubly linked list dapat dilihat pada langkah-langkah berikut ini:

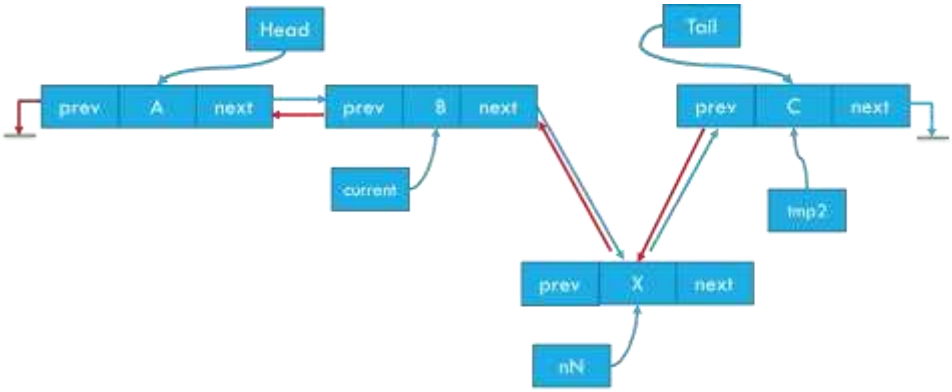
1. Lakukan pencarian node dimana newNode akan disimpan (hasil pencarian ditandai dengan current), set bantuan kedua (tmp2) pada node hasil pencarian next (current next).




2. Set previous newNode menuju current. Set next newNode menuju tmp2.




3. Set next current menuju newNode. Set previous tmp2 menuju newNode.




**Gambar 3.4 Insert Tengah pada Doubly Linked List**

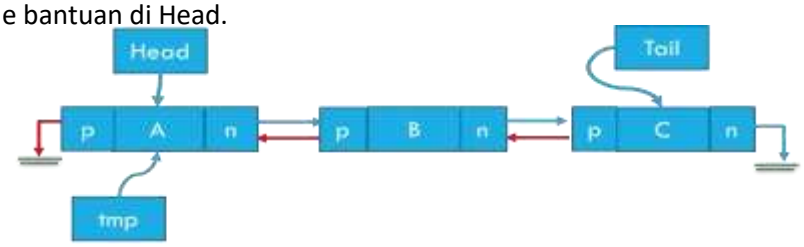
Implementasi dari insert depan, belakang dan tengah dapat dijadikan menjadi satu method dengan parameter masukan posisi dan data yang akan dimasukan. Berikut merupakan contoh dari implementasi insert Node pada suatu posisi.

```
public boolean addAt(int newPosition,T data) {
    Node newNode = new Node(data);
boolean isSuccesful = false;


    if((newPosition >= 1) && (newPosition <= numberOfEntries+1)){
if(isEmpty())//list kosong
        {
            firstNode = newNode;
lastNode = newNode;
        }
        else if(newPosition == 1){ //insert depan
newNode.setNext(firstNode);
firstNode.setPrev(newNode);              firstNode
= newNode;
        }
        else if (newPosition == numberOfEntries +1){ //insert belakang
lastNode.setNext(newNode);              newNode.setPrev(lastNode);
lastNode = newNode;          }else { //insert tengah
         //lihat method getNodeAt di subbab pencarian
            Node before = getNodeAt(newPosition - 1);
Node after = getNodeAt(newPosition + 1);
before.setNext(newNode);
newNode.setPrev(before);
newNode.setNext(after);
after.setPrev(newNode);
        }
        numberOfEntries++;
    }else
        isSuccesful = false;
return isSuccesful;
}
```

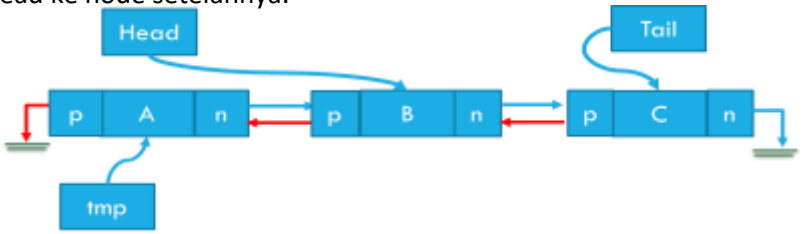
### 3.4.2 Operasi Hapus

Proses hapus dapat dibagi menjadi beberapa bagian, yaitu: hapus depan, hapus belakang, hapus tengah. Proses hapus depan dapat lihat pada langkah-langkah berikut ini:

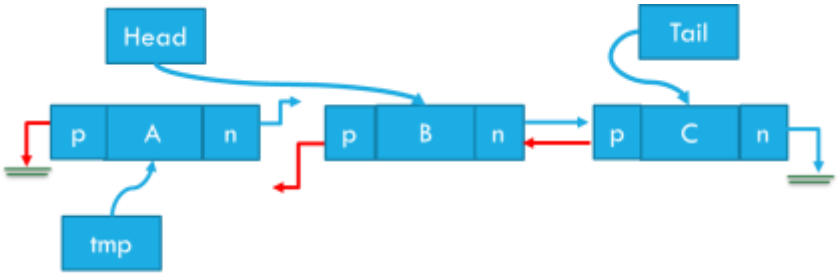
1. Simpan Node bantuan di Head.




2. Pindahkan Head ke node setelahnya.



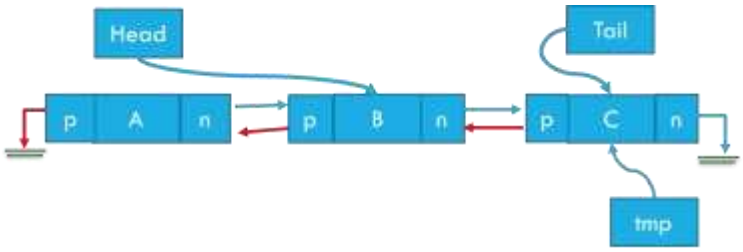
3. Set previous Head menjadi null dan next bantuan menjadi null.



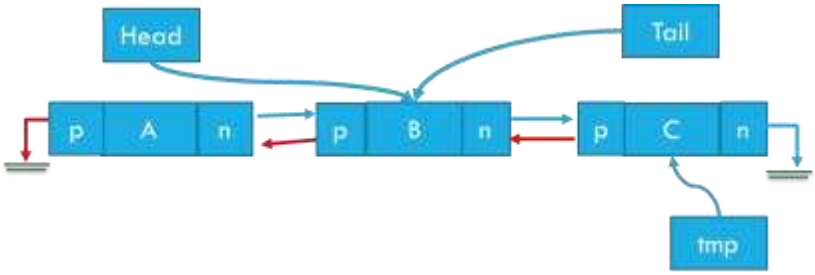

Gambar 3.5 Hapus Depan pada Doubly Linked List

Sedangkan proses hapus belakang dapat lihat pada langkah-langkah berikut ini:

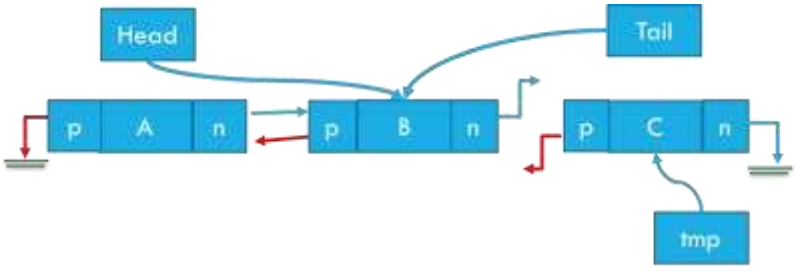
1. Simpan Node bantuan di Tail.




2. Pindahkan Tail di node sebelumnya.



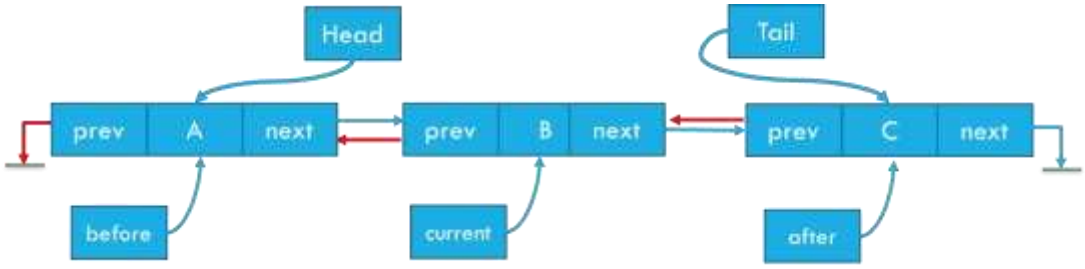

3. Set tail next menjadi null dan previous bantuan menjadi null.



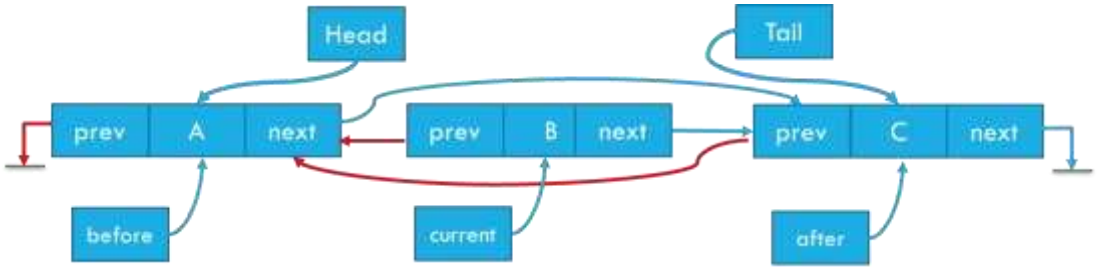

Gambar 3.6 Hapus Belakang pada Doubly Linked List

Sedangkan langkah-langkah untuk hapus tengah dapat dilihat pada gambar berikut:

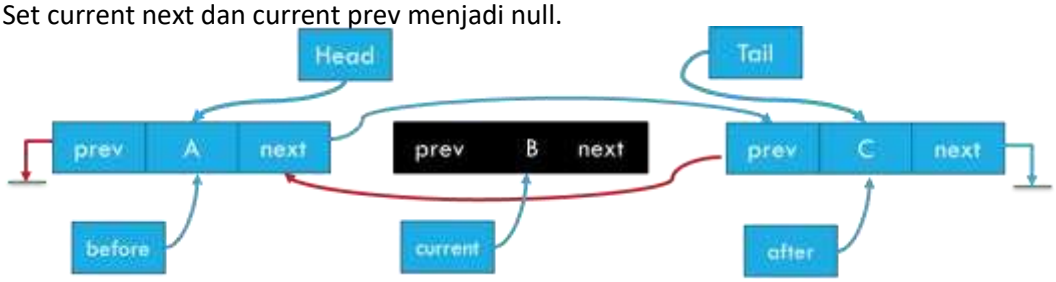
1. Lakukan pencarian untuk mencari node yang akan dihapus. (gunakan method getPosisition dan getNodeAt)
2. Set bantuan (before) pada node sebelum node yang akan dihapus (current). Set bantuan (after) pada node setelah node yangakan dihapus (current).




3. Set next dari before menuju after. Set previous dari after menuju before.




4. Set current next dan current prev menjadi null.




**Gambar 3.7 Delete Tengah pada Doubly Linked List**

Implementasi hapus depan, tengah dan belakang pada doubly linked list dapat dijadikan menjadi satu method dengan parameter masukan posisi dari node tersebut. Contoh implementasi dapat dilihat dibawah ini:

### 3.4.3 Pencarian, Traverse dan Display

Proses pencarian, traverse dan display pada Doubly Linked List pada dasarnya sama dengan proses pada singly linked list, namun proses tersebut pada doubly linked list lebih mudah, karena doubly linked list dapat melakukan iterasi-iterasi maju dan mundur. Sehingga dapat dilakukan dari head maupun dari tail.

```
public T findData(String nama) {
boolean found = false;
    Node iterator = firstNode;
    while (!found && (iterator != null ))
    {
        Mahasiswa mahasiswa = (Mahasiswa)iterator.data;
if (nama.equals(mahasiswa.getNama())) {
found = true;             return iterator.data;
        } else {
            iterator = iterator.next;
        }
}
    return null;
} // end
```

Selain menggunakan cara diatas, pencarian pun dapat dilakukan dengan mengunjungi setiap node, berhenti apabila node tersebut ditemukan dan mengembalikan posisi/data dari node tersebut. Posisi node dapat digunakan untuk parameter masukan dalam melakukan penghapusan maupun penyisipan.

```
private Node getNodeAt(int givenPosition){ //pencarian dengan nilai kembalian data
assert (firstNode != null && givenPosition >=1 && givenPosition <= numberOfEntries);
    Node current = firstNode;


    for (int i = 1; i < givenPosition;i++){
current = current.getNext();
    }
    assert current != null;


    return current;
}
```

```
public int getPositionOf(String ID) {
//pencarian dengan nilai kembalian posisi
    assert (firstNode!=null);
Node current = firstNode;


    for(int i = 1; i < numberOfEntries; i++){
Buku buku = (Buku)current.data;
if(ID.equals(buku.IDBuku)){
return i;          }else{
            current = current.next;
        }
} return -
1;
}
```

## 3.5 Implementasi Doubly Linked List

Implementasi ADT dalam doubly linked list dapat dibagi menjadi beberapa bagian yaitu:

1. Kelas ListDoubly.java yang berisi class Node, inisiasi List dan Implementasi dari method pada interface.

```
package com.company;


public class DoublyLinkedList<T> implements DoublyInterface<T>{
private Node firstNode; // reference to firstnode     private
Node lastNode; // reference to lastnode     private int
numberOfEntries;
     public DoublyLinkedList()
{         firstNode = null;
lastNode = null;
numberOfEntries = 0;
    }
        /* Implementasi dari DoublyInterface disimpan disini */
        /* Set Operasi dapat dilihat pada sub bab 3.3
          /* Class Node dari Subbab 3.2 */
    private class Node {
        private T data; // data entry (Contoh: Object Buku, dapat diganti Object lainnya)
private Node next; // link to next node         private Node prev; // link to next node
         private Node(T dataPortion) {
this(dataPortion, null, null);
        }


        public Node(T data, Node next, Node prev) {
this.data = data;             this.next = next;
this.prev = prev;
        }


        private T getData()
        {
return data;         }
// end getData


        private void setData(T newData)
        {             data
= newData;
        } // end setData


        public Node getNext() {
return next;
        }


        public void setNext(Node next) {
this.next = next;
        }


        public Node getPrev() {
return prev;
        }


        public void setPrev(Node prev) {
this.prev = prev;
        }


        @Override
        public String toString() {
return this.data.toString();
        }
    }
}
```

2. Kelas interface yang berisi set operasi dari ADT.

```
public interface DoublyInterface<T>{

    public boolean addAt(int position,T data);

    public boolean isEmpty();

    public int getPositionOf(String nama);

    public T removeAt(int position);

    public void traverseList(); }
```

3. Kelas Buku.java yang merupakan representasi dari data / object yang akan diolah.

```
package com.company;


public class Buku {
public String IDBuku;
public String Judul;
public int Harga;


    public Buku(String IDBuku, String judul, int harga) {
this.IDBuku = IDBuku;
        Judul = judul;
        Harga = harga;
    }


    @Override
    public String toString() {
return "Buku{" +
                "IDBuku='" + IDBuku + '\'' +
                ", Judul='" + Judul + '\'' +
                ", Harga=" + Harga +
                '}';
    }
}
```

4. Kelas Main.java

```
package com.company; public
class Main {
    public static void main(String[] args) {
Buku buku1 = new Buku("A1", "Awan", 3450);
       Buku buku2 = new Buku("A2", "Atuw", 3550);
Buku buku3 = new Buku("A3", "Atri", 3650);


        DoublyLinkedList<Buku> dBuku = new DoublyLinkedList<>();
        // insert pada Node dengan posisi X
System.out.println("Insert data");          dBuku.addAt(1,buku1);
        dBuku.addAt(dBuku.getPositionOf("Awan"),buku2);
dBuku.addAt(dBuku.getPositionOf("Awan"),buku2);         dBuku.traverseList();
        System.out.println("------");
        System.out.println();


        //removed berdasarkan posisi dari data X
System.out.println("Removed data");
dBuku.removeAt(dBuku.getPositionOf("A2"));         dBuku.traverseList();
    }
}
```

## 3.6 Latihan

Buatlah ADT pegawai dan Doubly Linked List yang mengimplementasikan ADT tersebut. Fungsi utama yang terdapat pada Linked List dengan ketentuan:

1. Input data pegawai
2. Melihat data pegawai
3. Menghapus data pegawai
4. Mengupdate data pegawai.
5. Menghapus seluruh data pegawai.

# Modul 4 : Stack

## 4.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep Stack pada Array
2. Mengenal konsep Stack pada Linked List

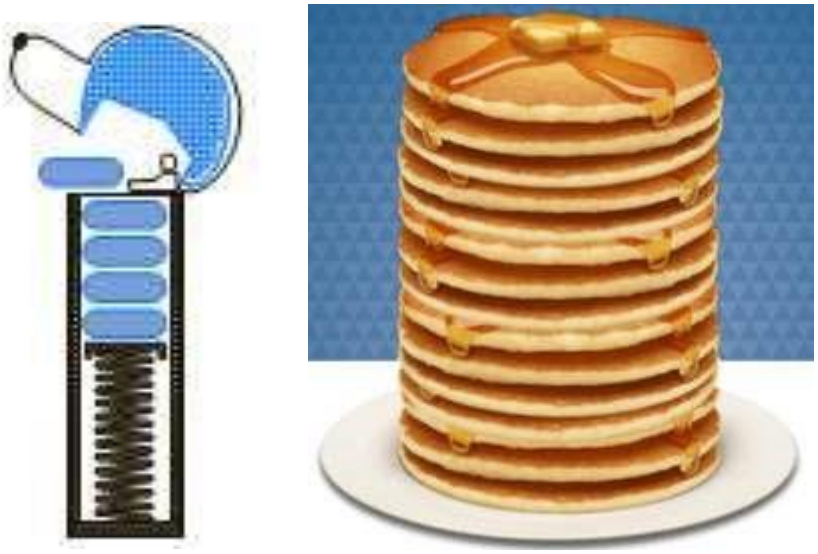
## 4.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 4.3 Stack

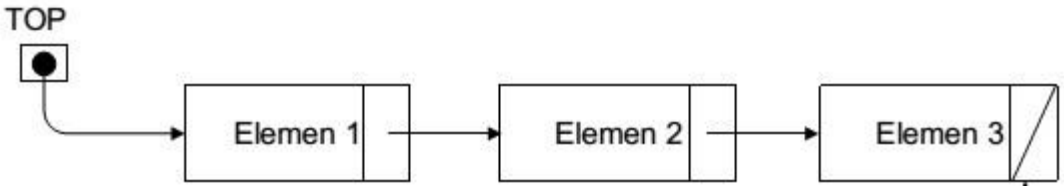
Stack merupakan salah satu bentuk struktur data dimana prinsip operasi yang digunakan seperti tumpukan. Seperti halnya tumpukan, elemen yang bisa diambil terlebih dahulu adalah elemen yang paling atas, atau elemen yang pertama kali masuk, prinsip ini biasa disebut LIFO (Last In First Out). User dapat memasukkan obyek ke dalam tumpukan setiap saat, tetapi hanya dapat mengakses atau menghapus objek yang paling baru dimasukkan atau yang paling atas (disebut "top" dari stack). Nama "stack" berasal dari metafora tumpukan piring disebuah dispenser pegas di kantin. Operasi dasar dari stack adalah "push" dan "pop" piring pada stack.

Ketika akan membutuhkan piring baru dari dispenser, kita "pop" piring dari atas tumpukan, dan ketika akan menambahkan piring, "push" ke dalam tumpukan, dan piring akan menjadi “top” baru dalam tumppukan. Gambar dispenser piring dibawah dapat menggambarkan hal ini.



**Gambar 4.1 (a) Kiri merupakan mekanisme dispenser piring. (b) Tumpukan pancake.**

Pada dasarnya implementasi stack dapat dilakukan menggunakan linked list dan array.




Gambar 4.2 Stack Linked List dengan 3 Element

## 4.4 Implementasi Stack dengan Singly Linked List

Pada pengimplementasian menggunakan linked list, komponen – komponen dalam stack pada dasarnya sama dengan komponen pada single linked list. Hanya saja akses dan operasi pada stack hanya bisa dilakukan pada awal stack saja(top). Operasi-operasi utama pada stack adalah

1. Push() untuk penambahan elemen pada top dari stack,
2. Pop() untuk penghapusan elemen dan mengembalikan nilai yg dihapus, pada top dari stack.

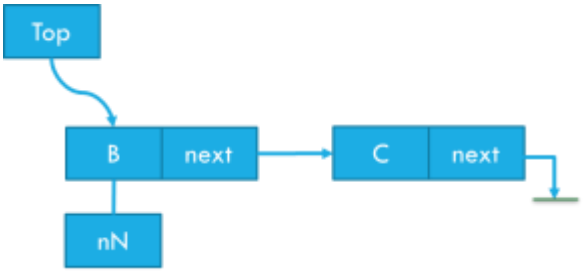
Sebagai tambahan stack pun dapat memiliki operasi tambahan selain kedua operpasi utama tersebut, sperti:

1. Top() mengembalikan nilai top dari stack tanpa menghapusnya.
2. isEmpty() mengembalikan nilai Boolean apakah stack tersebut kosong.
3. Size() mengembalikan banyaknya elemen didalam stack tersebut.

### 4.4.1 Push

Adalah operasi menyisipkan elemen pada tumpukan data. Fungsi ini sama dengan fungsi insert first pada list biasa.


(i) newNode next menuju top



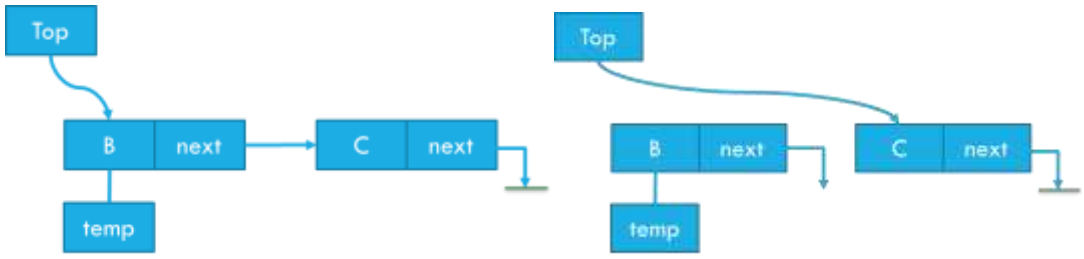

(ii) pindahkan top menuju newNode tersebut.

Gambar 4.3 Operasi Push pada Stack

```
public void push(T element) {
    Node newNode = new Node(element);
if (isEmpty()){          top =
newNode;
    }else{
        newNode.setNext(top);
top = newNode;
    }
size++;
}
```

### 4.4.2 Pop

Operasi pop adalah penghapusan elemen dan mengembalikan nilai yg dihapus, pada top dari stack. Operasi ini mirip dengan operasi delete first dalam list linear, karena elemen yang paling pertama kali diakses adalah elemen paling atas atau elemen paling awal saja.




(i) Simpan bantuan pada top

(ii) Pindahkan top pada node setelah top. Set next dari bantuan menuju null

Gambar 4.4 Operasi Pop pada Stack

```
public T pop() {      Node temp = null;
if(!isEmpty()){          temp = top;
top = top.getNext();
temp.setNext(null);          size--;
    }
    return (T)temp.getData(); }
```

### 4.4.3 Implementasi

Pada dasarnya implementasi stack menggunakan linked list sama seperti implementasi pada single linked list. Pada implementasi tersebut terdiri dari Interface Stack yang dapat pula digunakan untuk Stack pada array, dan kelas Implementasi operasi-operasi pada Stack tersebut. Berikut merupakan Interface Stack didokumentasikan dengan komentar dalam gaya Javadoc. Perhatikan juga penggunaan jenis parameter generik, T, yang memungkinkan stack untuk mengandung element dari tipe-tipe data spesifik.

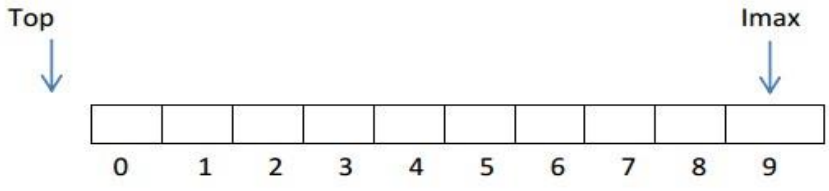
```
public interface StackInterface<T> {
  /**
    * Returns the number of elements in the stack.
    * @return number of elements in the stack
    */
    public int size();

  /**
   * Tests whether the stack is empty.
   * @return true if the stack is empty, false otherwise
   */
    public boolean isEmpty();

  /**
   * Inserts an element at the top of the stack.
   * @param e the element to be inserted
   */
    public void push(T element);

  /**
   * Returns, but does not remove, the element at the top of the stack.
   * @return top element in the stack (or null if empty)
   */
    public T top();
  /**
   * Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
    public T pop();
}
```

Interface pada stack ini dapa digunakan untuk implementasi pada linked list maupun implementasi stack menggunakan array. Dibawah ini merupakan contoh kelas implementasi array menggunakan linked list.

```
public class StackLL<T> implements StackInterface<T> {
private Node top;     private int size;

    public StackLL() {
this.top = null;
this.size = 0;
    }
 // Implement method dari Operasi Stack pada Interface

    private   class   Node{
private         T     data;
private Node next;

        public Node(T datax){
this(datax, null);
        }

        public Node(T data, Node next)
{             this.data = data;
this.next = next;
        }

        public T getData() {
return data;
        }

        public void setData(T data) {
this.data = data;
        }

        public Node getNext() {
return next;
        }

        public void setNext(Node next) {
this.next = next;
        }
    }
}
```

## 4.5 Implementasi Stack menggunakan Array

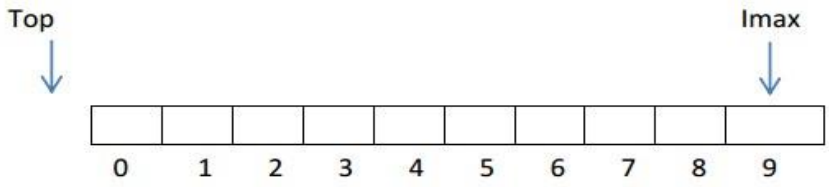
Pada prinsipnya Stack representasi menggunakan Array sama halnya dengan menggunakan linked list. Perbedaannya terletak pada pendeklarasian strukturdatanya, menggunakan array berindeks dengan jumlah tumpukan yang terbatas. Operasi yang terdapat pada Stack menggunakan array sama halnya seperti stack menggnakan Linked list dan Interface yang dibuat pun dapat menggunakan interface yang sama.



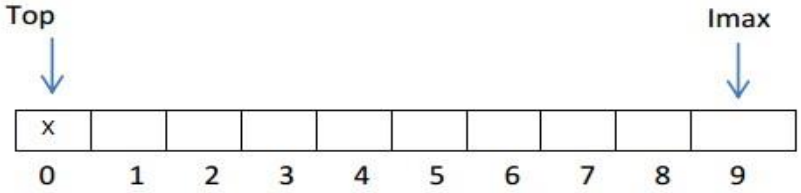
Gambar 4.5 Stack menggunakan Array

### 4.5.1 Push

Push merupakan operasi penyisipan data ke dalam stack, penyisipan dilakukan dengan menggeser indeks dari TOP ke indeks berikutnya.




(i) Set top = -1



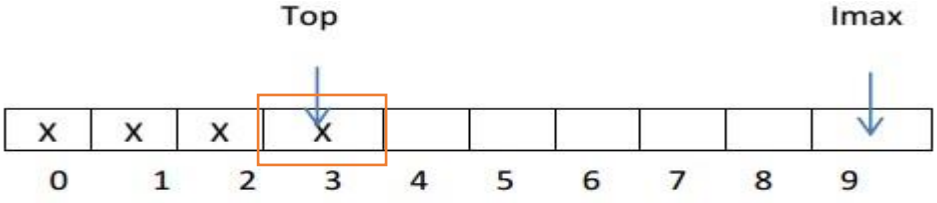

(ii) Top = top + 1

Gambar 4.6 Push pada Stack menggunakan Array

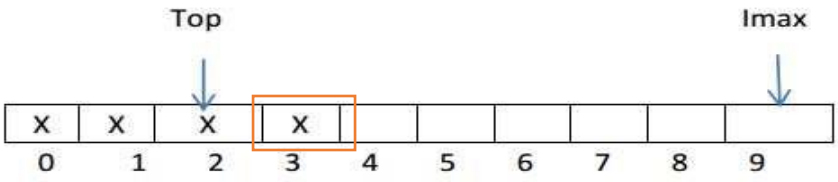
```
public void push(T element) {
if (size() == data.length){
        System.out.println("Stack Overflow");
    }else{          top =
top+1;          data[top] =
element;
    }
}
```

### 4.5.2 Pop

Pop merupakan operasi pengambilan data di posisi indeks TOP berada dalam sebuah stack. Setelah data diambil, indeks TOP akan bergeser ke indeks sebelum TOP dan dilakukan penghapusan. Perhatikan contoh dibawah ini:




(i) Gunakan bantuan untuk mengambil dan mengembalikan nilai dari top. Temp = top.




(ii) Top = Top – 1

**Gambar 4.7 Pop pada Stack menggunakan Array**

```
public T pop() {
if(isEmpty()) {
return null;
    }
else {
        T temp = data[top];
data[top] = null;
top--;          return temp;
    }
}
```

### 4.5.3 Implementasi

Pada implementasi Stack menggunakan Array, dapat menggunakan Interface Stack pada subbab sebelumnya. Berikut merupakan contoh implemtasi Stack menggunakan Array.

```
public class StackArray<T> implements StackInterface<T> {
static final int MAXSIZE = 5;     private T[] data;
private int top = -1;

    public StackArray(){
this(MAXSIZE);
    }
    public StackArray(int maximum){
data = (T[]) new Object[maximum];
    }

        // Implementasi Method pada Interface

    @Override
public int size()

    @Override
    public boolean isEmpty()



    @Override
    public void push(T element)

    @Override
public T pop()

    @Override
public T top()
```

```
}
```

## 4.6 Latihan

Buatlah sebuah program permainan MENARA HANOI dengan mengimplementasikan struktur data stack. MENARA HANOI adalah sebuah game memindahkan disk-disk dari menara ke menara yang lain. Aturannya adalah sebagai berikut.

- Terdapat tiga menara, satu menara sudah berisi tumpukan sejumlah disk
- Disk-disk tersebut berukuran/berbobot berbeda-beda
- Disk dapat ditumpuk jika menara tersebut masih kosong atau disk teratasnya pada menara tersebut ukurannya/bobotnya lebih besar dari disk yang akan ditumpuk.
- Satu menara awal yang berisi tumpukan sejumlah disk, disknya berurutan dari ukuran/bobot paling besar sampai paling kecil (dari terbawah sampai teratas)
- Permainan berakhir jika semua tumpukan disk berada pada satu menara selain menara awal.

# Modul 5 : Queue

## 5.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep Queue pada Array
2. Mengenal konsep Queue pada Linked List
3. Mengenal konsep Priority Queue

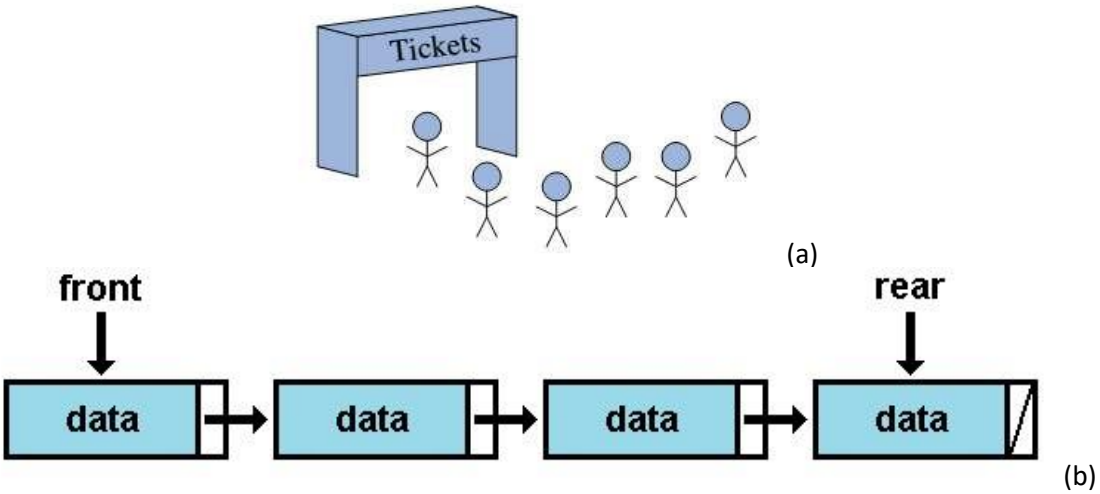
## 5.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 5.3 Queue

Salah satu struktur data dasar yang dikenal sebagai “saudara” dari stack adalah queue. Berbeda dengan stack, queue atau antrian merupakan struktur data yang menganut prinsip FIFO (*First In First Out*), atau yang pertama datang adalah yang pertama kali dilayani. Elemen akan masuk ke dalam queue melalui “belakang” dan akan dihapus pada saat setelah berada di-”depan”. Metafora dari terminology ini adalah pada saat melakukan pembelian tiket bioskop atau pada saat membeli bensin di SPBU. Implementasi queue dbidang *computing* salah satunya adalah implementasi dari antrian digunakan pada printer, juga pada Web server ketika merespon request dari client.

Operasi dasar dari queue terdiri dari dua operasi utama, yaitu enqueue untuk menambahkan element pada belakang antrian (*rear*), dan operasi dequeue untuk menghapus elemen pertama (*front*) dan mengembalikan nilai dari elemen tersebut. Pada dasarnya implementasi dari queue sama seperti stack dapat menggunakan array maupun linked list.




**Gambar 5.1 (a) Antrian Tiket (b) Implementasi dari queue mengunakan linked list**

## 5.4 Implementasi Queue dengan Single Linked List

Pada pengimplementasian menggunakan linked list, komponen – komponen dalam queue pada dasarnya sama dengan komponen pada single linked list. Seperti, istilah head atau elemen pertama pada single linked list, berganti istilah menjadi front. Dan istilah tail yang merupakan element akhir dari single linked list berubah menjadi rear pada queue. Operasi-operasi utama pada queue adalah

1. enqueue() untuk penambahan elemen pada rear dari queue,
2. dequeue() untuk penghapusan elemen dan mengembalikan nilai yg dihapus, pada front dari queue.

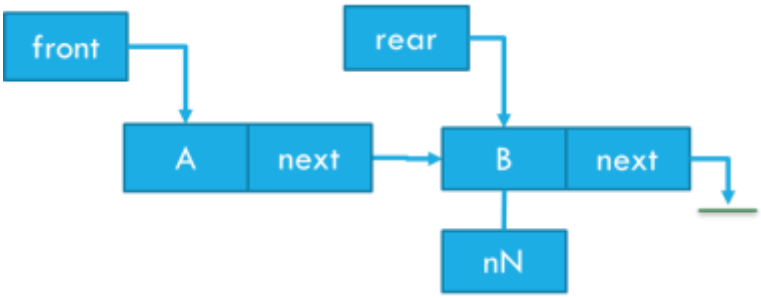
Sebagai tambahan queue pun dapat memiliki operasi tambahan selain kedua operpasi utama tersebut, seperti:

1. first() mengembalikan nilai front dari queue tanpa menghapusnya.
2. isEmpty() mengembalikan nilai Boolean apakah queue tersebut kosong.
3. size() mengembalikan banyaknya elemen didalam queue tersebut.

### 5.4.1 Enqueue

Operasi enqueue pada queue yang diimplementasikan dengan single linked list pada dasarnya sama seperti operasi sisip/insert belakang pada single linked list.


(i) Set rear next menuju newNode.



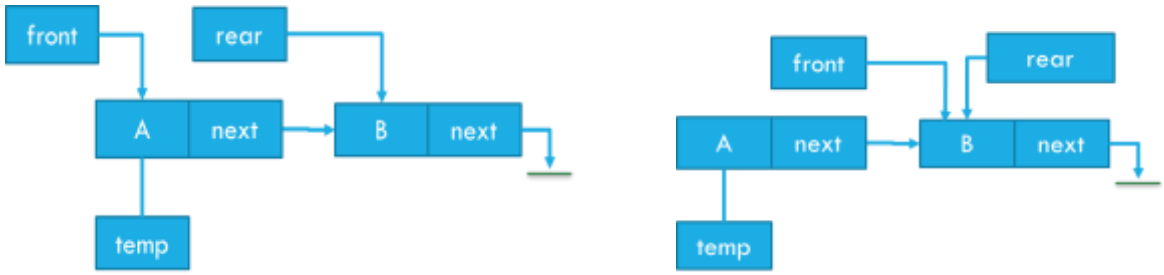

(ii) Set rear di newNode.

**Gambar 5.2 Enqueue pada Queue dengan Implemntasi Single Linked ListEnqueue**

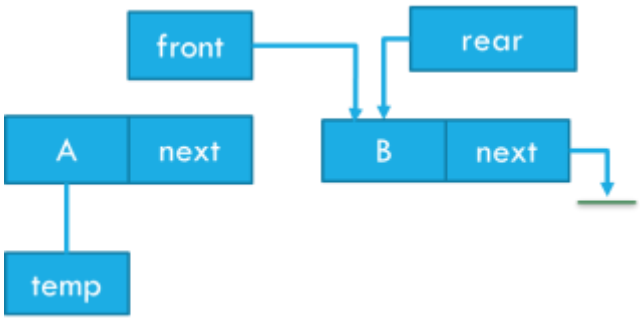
```
public void enQueue(T element) {
Node newNode = new Node(element);
if(isEmpty()){
        front = rear = newNode;
    }else{
        rear.setNext(newNode);
rear = newNode;
    }
}
```

### 5.4.2 Dequeue

Operasi dequeue adalah penghapusan elemen dan mengembalikan nilai yg dihapus, pada frontsebuah queue. Operasi ini mirip dengan operasi remove depan dalam single linked list.




(i) Set temp dipada front, pindahkan front menuju node/elemen selanjutnya.




(ii) Set temp next menuju null

**Gambar 5.3 Dequeue pada Queue dengan Implementasi Linked List**

```
public T deQueue() {
Node temp = null;
if (!isEmpty()){
temp = front;
        front = front.getNext();
temp.setNext(null);
    }
        return temp.data; }
```

### 5.4.3 Implementasi

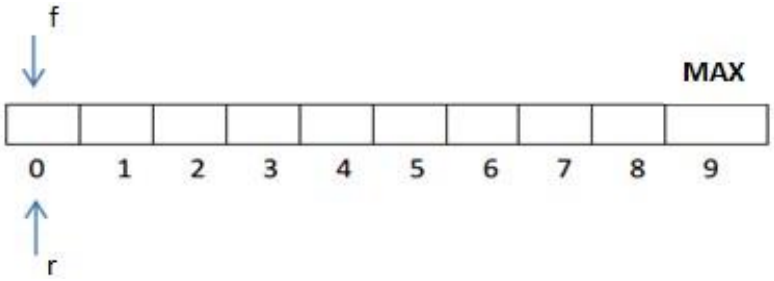
Pada dasarnya implementasi queue menggunakan linked list sama seperti implementasi pada single linked list. Pada implementasi tersebut terdiri dari Interface Queue yang dapat pula digunakan untuk Queue pada array, dan kelas Implementasi operasi-operasi pada Queue tersebut. Berikut merupakan Interface Queue didokumentasikan dengan komentar dalam gaya Javadoc. Perhatikan juga penggunaan jenis parameter generik, T, yang memungkinkan queue untuk mengandung element dari tipe-tipe data spesifik.

```
public interface QueueInterface<T> {
    /** Tests whether the queue is empty. */
public boolean isEmpty();

    /** Returns the number of elements in the queue. */
public int size();

    /** Inserts an element at the rear of the queue. */
public void enQueue(T element);

    /** Removes and returns the first element of the queue (null if empty). */
public T deQueue();

    /** Returns the first element of the queue (null if empty). */
public T first();
}
```

```
public class QueLL<T> implements QueueInterface<T>
{     private Node front;     private Node rear;
private int size;

    public QueLL() {
this.front = null;
this.rear = null;
this.size = 0;
    }
 << Implementasi Queue method dari Operasi Stack pada Interface >>

    private   class   Node{
private         T     data;
private Node next;

        public Node(T datax){
this(datax, null);
        }

        public Node(T data, Node next)
{             this.data = data;
this.next = next;
        }

            }
}
```

## 5.5 Implementasi Queue dengan Array

Pada dasarnya representasi queue menggunakan array sama halnya dengan menggunakan linked list. Perbedaan yang mendasar adalah pada management memori serta keterbatasan jumlah antriannya. . Operasi yang terdapat pada queue menggunakan array sama halnya seperti queue menggnakan Linked list dan Interface yang dibuat pun dapat menggunakan interface yang sama.




Gambar 5.4 Implementasi Queue menggunakan Array

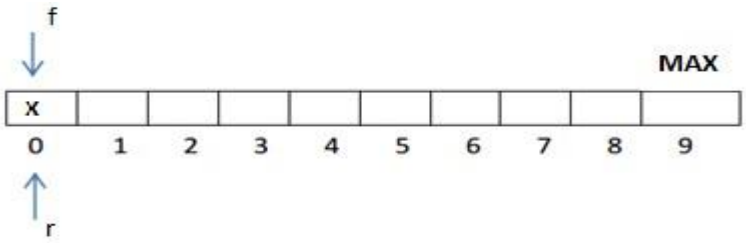
### 5.5.1 Enqueue

Enqueue merupakan operasi penyisipan data ke dalam queue, penyisipan dilakukan dengan menggeser indeks dari rear ke indeks berikutnya.

(i) Queue kosong, set front = 0, size = 0

(ii) Set **rear = (front + size) modulus panjang** dari array.

```
rear = (front + size) % x.length
```

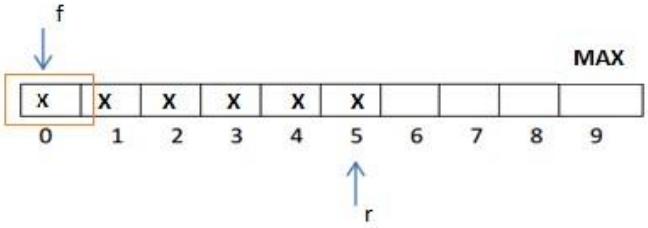



Gambar 5.5 Enqueue pada Queue dengan Implementasi Array

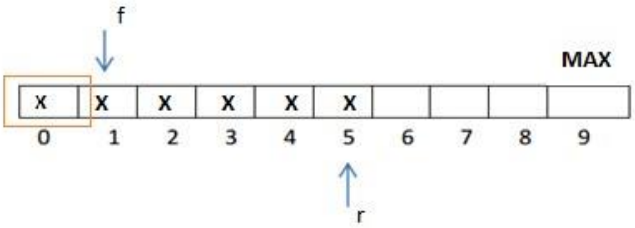
```
public void enQueue(T element) {
if (size == data.length) {
        System.out.println("Queue full");
    }else{
        rear = (front + size) % data.length;
data[rear] = element;          size++;
    }
}
```

### 5.5.2 Dequeue

Dequeue merupakan operasi penghapusan data dalam queue, penghapusan dilakukan dengan menggeser indeks dari front ke indeks berikutnya kemudian mengembalikan nilai yang telah dihapus.




(i) Gunakan bantuan untuk mengambil dan mengembalikan nilai dari front. Temp = front.




(ii) Set **front = (front +1) modulus panjang** dari array.

**Gambar 5.6 Dequeue pada Queue dengan Implementasi Array**

```
public T deQueue() {
if (isEmpty()){
return null;
    }else{
        T temp = data[front];
data[front] = null;
        front = (front+1) % data.length;
size--;          return temp;
    }
}
```

### 5.5.3 Implementasi

Pada implementasi Queue menggunakan Array, dapat menggunakan Interface Queue pada subbab sebelumnya. Berikut merupakan contoh implementasi Queue menggunakan Circular Array.

```
public class QueArray<T> implements QueueInterface<T> {
static final int MAXSIZE = 100;     private T[] data;
private int front = 0;      private int rear;
private int size = 0;

    public QueArray(){
this(MAXSIZE);
    }
    public QueArray(int maximum){
data = (T[]) new Object[maximum];
    }

    @Override
    public boolean isEmpty() {
    }

    @Override
    public void enQueue(T element) {
    }

    @Override
    public T deQueue() {
    }

    @Override
}
```

5.6

Latihan

Pt. Bank Banktut membutuhkan sebuah sistem antrian nasabah. Dengan mekanisme sebagai berikut.

1. Nasabah diminta untuk mengambil no antrian.
2. Dalam no antrian tersebut, harus terdapat, no antrian dari nasabah dan jumlah antrian tersisa sebelum dari  hingga nasabah tersebut terlayani.

# Modul 6 :  Binary Search Tree

## 6.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep Tree
2. Mengenal konsep Binary Search Tree

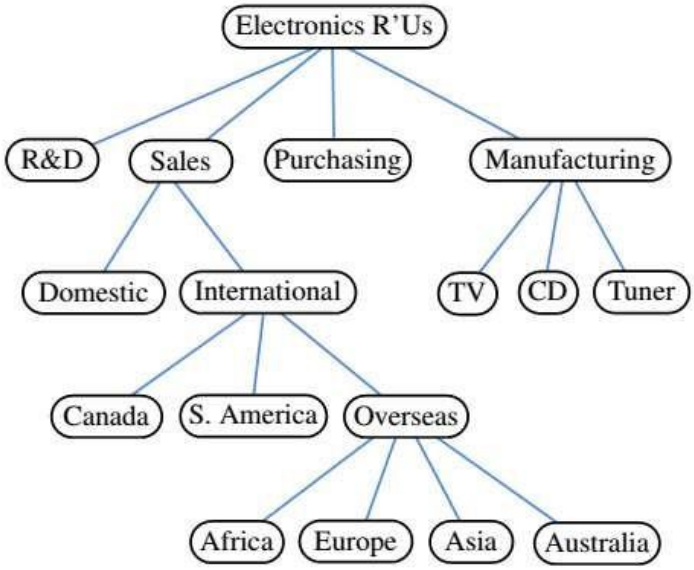
## 6.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 6.3 Tree

Tree adalah tipe data abstrak yang menyimpan elemen secara hierarkis, berbeda dengan stack, queue dan list yang merupakan tipe data astrak yang menyimpan elemen secara linear. Setiap elemen dalam tree memiliki parent (kecuali root) ataupun child sebanyak nol ataupun lebih. Sebuah tree biasanya

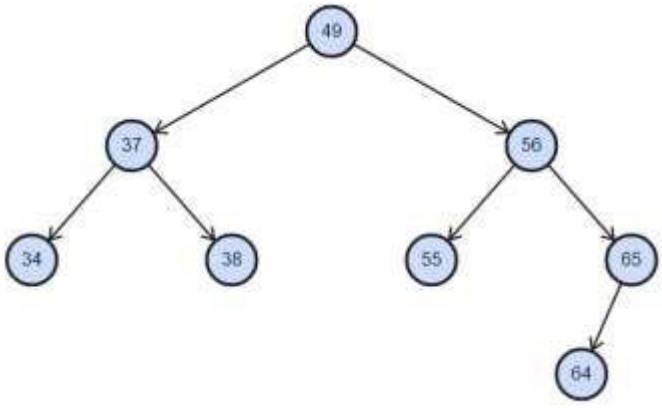
divisualisasikan dengan menempatkan elemen dalam oval atau persegi panjang, dan dengan menggambar hubungan antara parent dan child dengan garis lurus.




Gambar 6.1 Tree (Goodrich, 2014)

## 6.4 Binary Search Tree

Binary Search Tree merupakan tree dimana untuk setiap node X di pohon, nilai-nilai disemua subtree kiri lebih kecil dari node di X dan nilai-nilai dari subtree kanan lebih besar dari nilai di node di X.




Gambar 6.2 Contoh Binary Search Tree

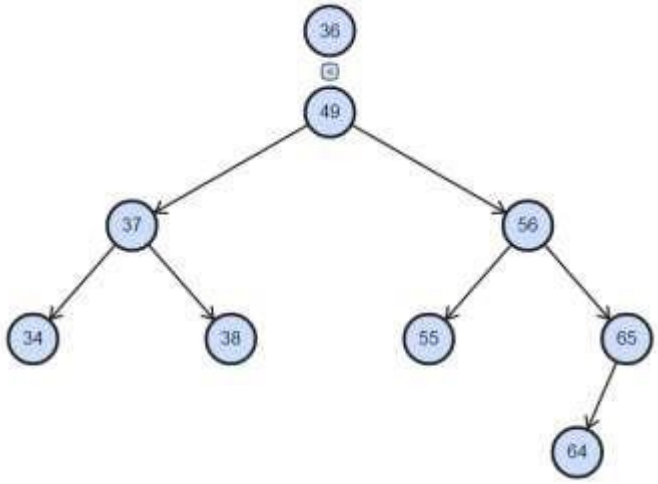
Terdapat beberapa operasi pada binary search tree, yaitu: Insert, Traverse, terdapat tiga macam traversing pada binary search tree, yaitu: preorder, inorder, postorder. Deleted dan Search.

```
public class BSTree implements BSTreeInterface{
public Node root;

    public BSTree(){ root = null;}

    public void addNodeTree(Comparable obj){
    }
    public boolean find(Comparable obj) {
    }

    public void remove(Comparable obj){
    }
    public void printInorder(Comparable obj){
    }
    public void printPostorder(Comparable obj){
    }
    public void printPreorder(Comparable obj){
    }
    class Node{
        public Comparable data;
public Node left;         public
Node right;

        public void addNewNode(Node newnode){
        }

        public void inorder(){
         }

         public void preorder(){
         }

         public void postorder(){
         }
    }
}
```

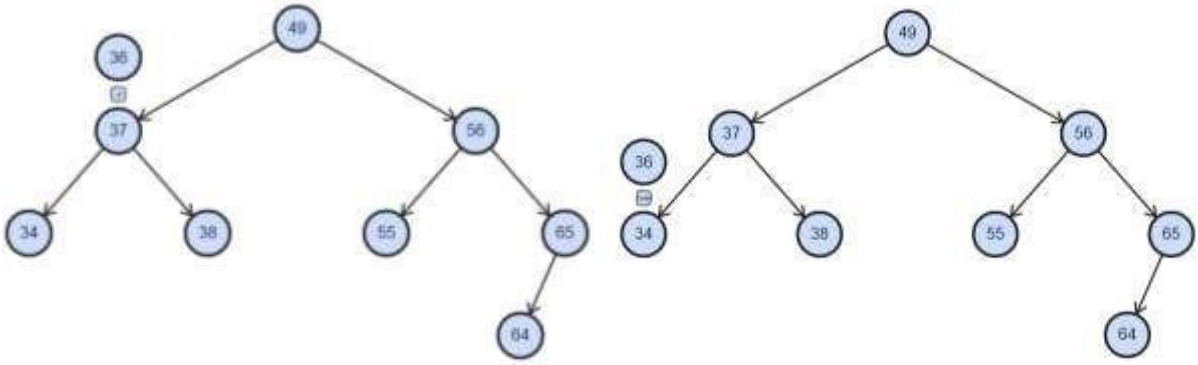
### 6.4.1 Insert pada Binary Search Tree

Pada penyisipan sebuah elemen baru dalam binary search tree, maka elemen tersebut dipastikan akan menjadi leaf dari tree tersebut. Inserting pada binary search tree, dilakukan dengan membandingkan value dari element yang akan dimasukan dengan element dari tree tersebut, berikut merupakan langkah-langkah inserting pada binary search tree.

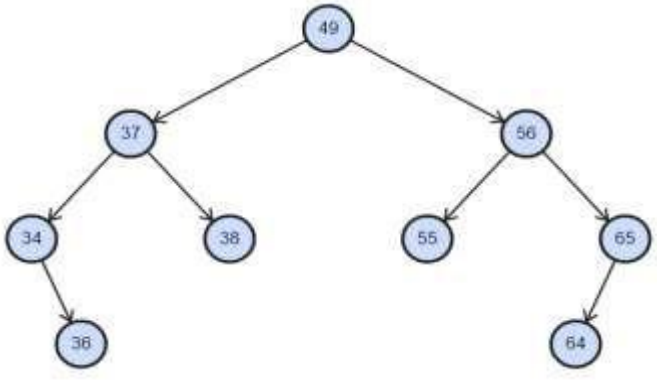
1. Insert x = 36 pada binary search tree, dimulai dari root, jika value dari x lebih kecil dari value dari root, maka x harus dimasukan di left subtree.




2. Sebaliknya jika value dari x lebih besar dari root, maka x harus dimasukan di right subtree.
3. Subtree merupakan juga merupakan tree, sehingga masalah inserting pada element di subtree diselesaikan sama seperti pada step 1 di root. (*Recursive approach*)




(i) x = 36, lebih kecil dari Node = 37 (ii) x = 36, lebih besar dari Node = 34




(iii) x = 36, menjadi right-child dari node = 34

```
public void addNodeTree(Comparable obj){
    Node newnode = new Node();
newnode.data = obj;
newnode.right = null;
newnode.left = null;

    if(root == null)          root = newnode;      else
root.addNewNode(newnode); //merupakan method pada inner class Node }
```

```
public void addNewNode(Node newnode){
int comp = newnode.data.compareTo(data);
if(comp < 0) {          if (left == null)
left = newnode;          else
            left.addNewNode(newnode);
    }
    if(comp > 0){
if(right == null)
right = newnode;
else
            right.addNewNode(newnode);
    }
}
```

### 6.4.2 Traverse pada Binary Search Tree

Traverse pada binary tree terdapat tiga macam traversing pada binary search tree, yaitu: preorder, inorder, postorder. Pada preorder traversal, lakukan kunjungi root, kunjungi left subtree, kunjungi right subtree. Pada inorder traversal, kunjungi left subtree, root, right subtree. Pada postorder traversal, kunjungi left subtree, kunjungi right subtree, kunjungi root.

```
public void printPreorder(){
if (root != null)
        root.preorder(); // method pada inner class Node
    System.out.print(""); }
```

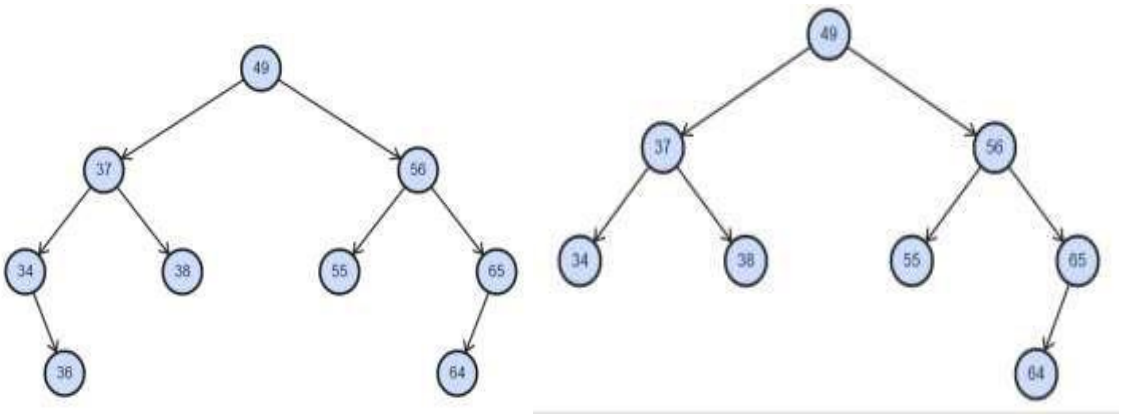
```
public void preorder(){
    System.out.print( data + " "); //kunjungi root
if(left != null)
        left.preorder(); //kunjungi left subtree
if(right != null)
        right.preorder(); //kunjungi right subtree }
```

Traversing pada inorder dan postorder, sama dengan preorder, perbedaannya hanya dengan mengubah urutan dari kunjungan saja.

### 6.4.3 Delete pada Binary Search Tree

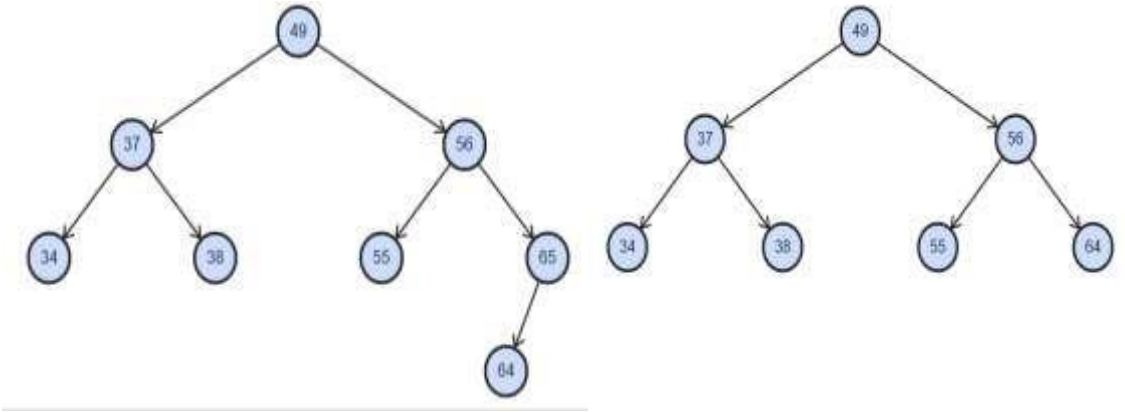
Deleting pada Binary Search Tree, terbagi menjadi tiga scenario, yaitu:

1. Jika dilakukan deleting pada leaf (node yang tidak memiliki child), maka tidak perlu dilakukan modifikasi.



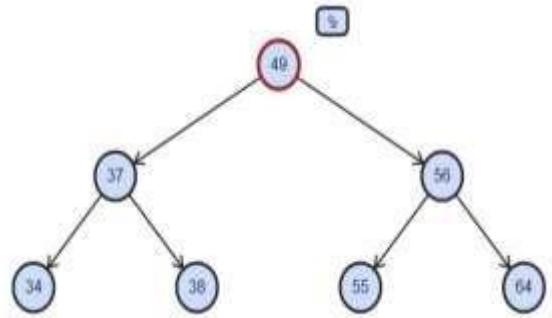

(a) Kiri sebelum di delete, (b) Kanan, setelah di delete

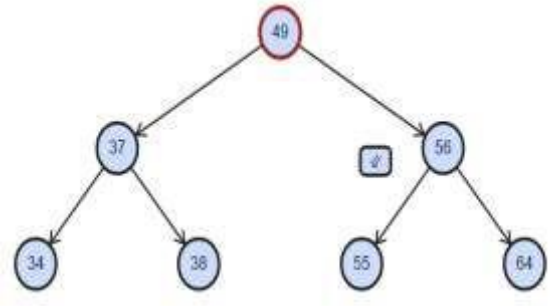
2. Node dengan satu child (baik left child maupun right child), maka child langsung menggantikan posisi parent.



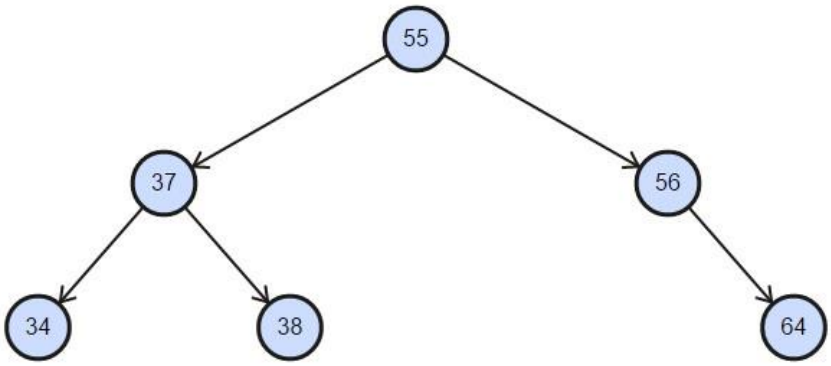

(b) Kiri sebelum di delete, (b) Kanan, setelah di delete

3. Node dengan dua child, maka temukan predesessor inorder ataupun successor inorder dari node tersebut. Predesessor inorder didapatkan dengan mencari nilai maksimum dari left subtree pada node tersebut, dan successor didapatkan dengan mencari nilai minimum dari right subtree pada node tersebut.




(i) Find predesessor / successor in order (proses diatas adalah pencarian successor, mencari nilai minimum dari right subtree pada node tersebut)



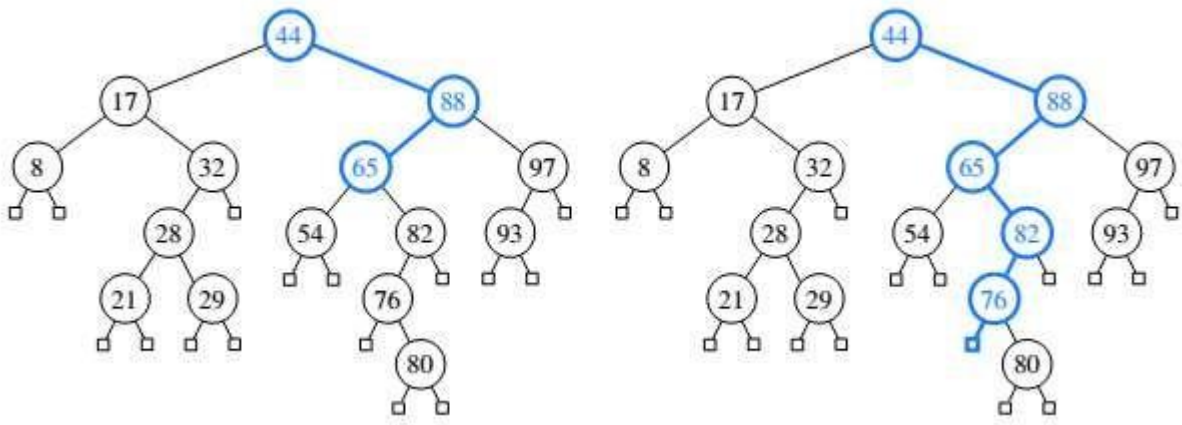

(ii) Swap data predesessor / successor in order dengan node yg akan dihapus

```
public void remove(Comparable obj){
    Node toRemoved = root;
Node parent = null;     boolean
found = false;

        while(!found && toRemoved !=null){
int d = toRemoved.data.compareTo(obj);
if (d == 0)                  found = true;
else {                  parent = toRemoved;
if (d > 0)
                    toRemoved = toRemoved.left;
else                     toRemoved =
toRemoved.right;
            }

        if (!found) return;

        // If one of the children is empty, use the other
if(toRemoved.left == null || toRemoved.right == null){
Node newChild;
            if (toRemoved.left == null)
newChild = toRemoved.right;
else                   newChild =
toRemoved.left;

            if (parent == null)
root = newChild;
            else if (parent.left == toRemoved)
parent.left = newChild;              else if
(parent.right == toRemoved)
parent.right = newChild;              else
return;
         }
            // if node having two child
            // find its successor
            Node temp = toRemoved;
Node temp2 = toRemoved.right;
while(temp2.left != null){
temp = temp2;                  temp2 =
temp2.left;
            }

            // swap data, moved child
toRemoved.data = temp2.data;
if (temp == toRemoved)
temp.right = temp2.right;
else                  temp.left =
temp2.right;
    }
}
```

### 6.4.4 Search pada Binary Search Tree

Salah satu operasi paling yang penting dari Binary Search Tree adalah algoritma pencarian. Operasi pencarian dilakukan dimulai dari root membandingkan nilai yang dicari dengan nilai pada root, apabila nilai tersebut “lebih kecil” maka akan pencarian akan lanjut pada subtree sebelah kiri, jika nilai tersebut “lebih besar” maka pencarian akan dillanjut pada subtree sebelah kanan, dan jika hasil yang didapat sama

maka pencarian akan berakhir dengan sukses. Jika pencarian mencapai leaf, dan tetap tidak didapatkan hasil yang sama maka pencarian berakhir gagal.




(a) Kiri - Pencarian nilai 65, berhasil. (b) Kanan Pencarian nilai 68, tidak ditemukan. (Goodrich, 2014)

```
public boolean find(Comparable obj)
{     Node current = root;     while
(current != null ){
        int d = current.data.compareTo(obj);
if (d == 0)              return true;
else if (d > 0)
            current = current.left;
else             current =
current.right;
    }
    return false;
}
```

# Modul 7 : Heap Tree

## 7.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

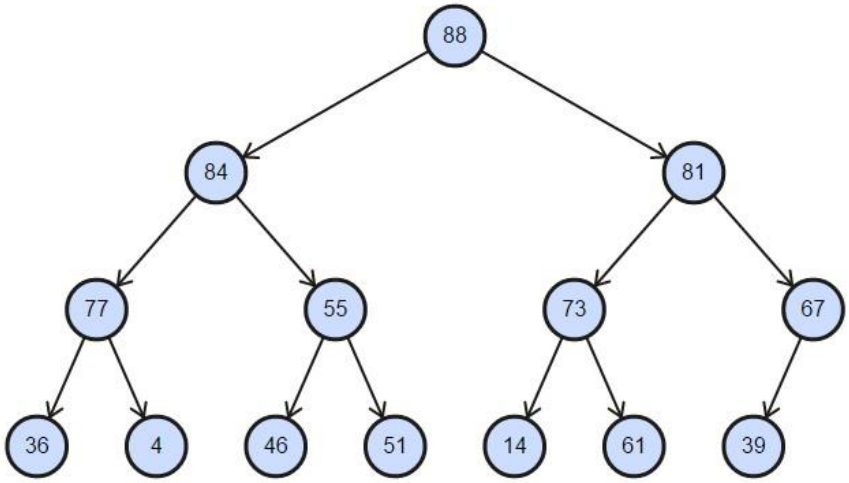
1. Mengenal konsep Tree
2. Mengenal konsep Binary Search Tree

## 7.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 7.3 Heap Tree

Heap tree merupakan sebuah Complete Binary Tree, dimana setiap nodenya diurutkan berdasarkan sebuah prasyarat. Heap dikatakan, Minimum Heap, jika parentnya selalu lebih kecil daripada kedua childrennya. Sedangkan Maximum Heap adalah heap tree jika parentnya selalu lebih besar daripada kedua childrennya.




**Gambar 7.1 Maximum Heap Tree**

```
public interface HeapInterface {     public void add(Comparable newElement);

    public void print();

    public Comparable peek();

    public Comparable remove(Object o);

    public int size();
}
```

```
public class HeapTreeMin implements HeapInterface{

    private ArrayList<Comparable> elements;

    public HeapTreeMin() {
        elements = new ArrayList<Comparable>();
elements.add(null);
    }

    public void add(Comparable newElement) {
            }

    public void print(){
            }

    public Comparable peek() {
            }

    public Comparable remove(Object o) {
            }

    private void fixHeap(int index) { //digunakan pada saat removing
            }

    public int size() {
            }

    private static int getLeftChildIndex(int index) {
return 2 * index;
    }
    private static int getRightChildIndex(int index) {
return 2 * index + 1;
    }
    private static int getParentIndex(int index) {
return index / 2;
    }
    private Comparable getLeftChild(int index) {
return elements.get(2 * index);
    }
    private Comparable getRightChild(int index) {
return elements.get(2 * index + 1);
    }
    private Comparable getParent(int index) {
return elements.get(index / 2);
    }
}
```

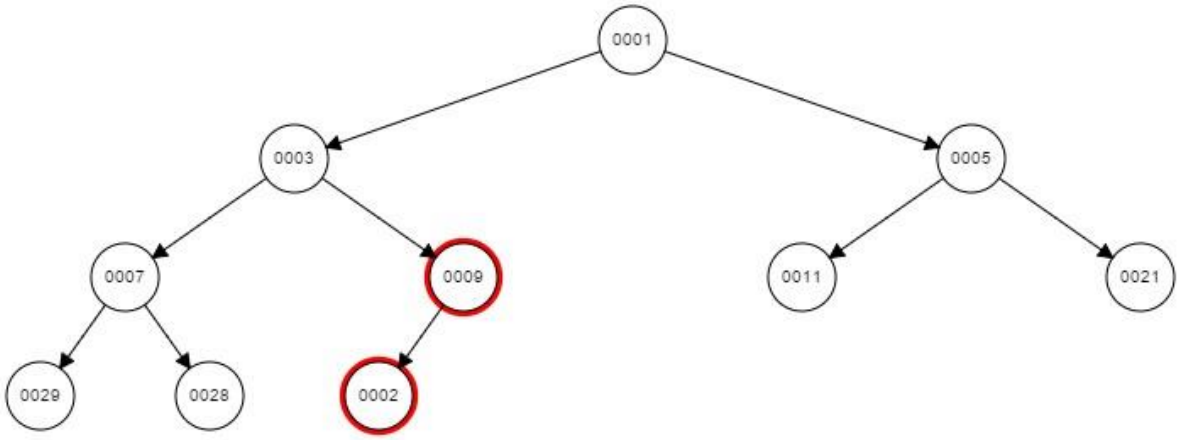
## 7.4 Operasi pada Heap Tree

Terdapat beberapa operasi utama pada heap tree, yaitu: add Node, removing Node, dan Heap sort.

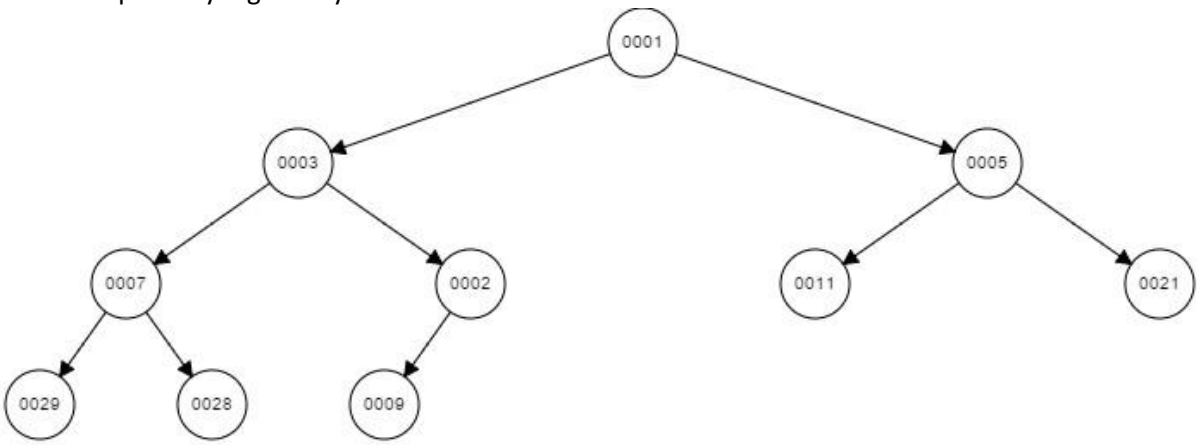
### 7.4.1 Insert Node

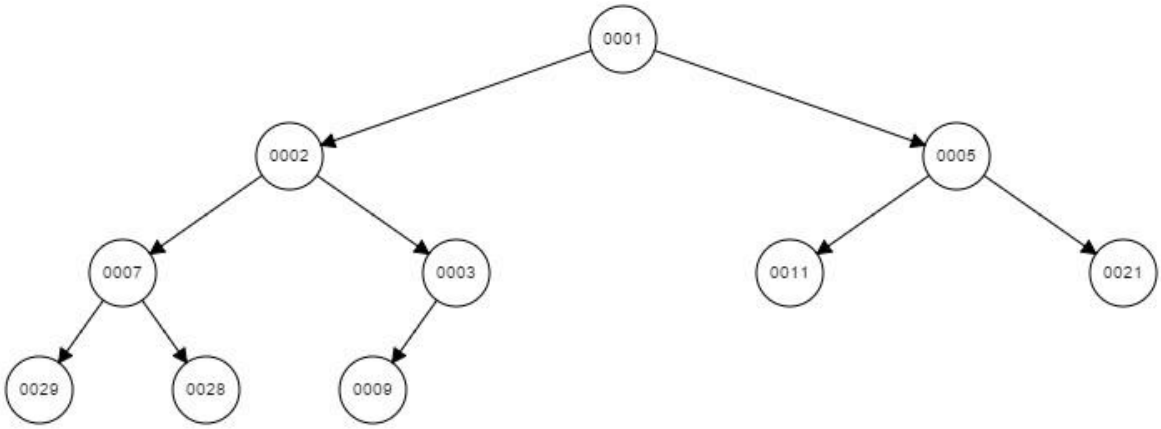
Algoritma insert Node pada heap (minimum) dapat dilihat dibawah ini:

1. Add Node, pada slot terkahir pada tree.




2. Bandingkan nilai node baru tersebut dengan parent-nya, jika parent tersebut lebih kecil dari pada node baru, maka tukar posisi node baru dan parent. Lakukan langkah ini sampai hingga tidak ditemukan parent yang nilainya lebih besar dari node baru.



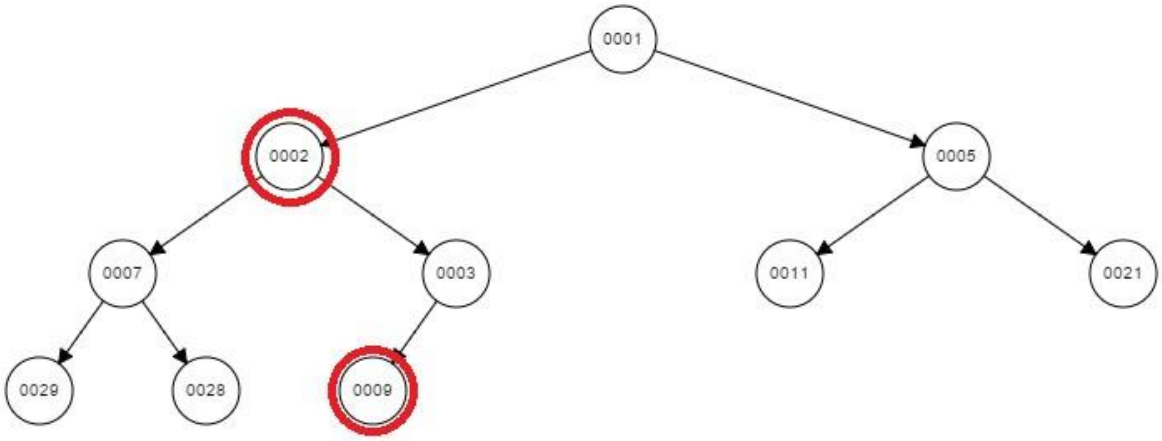



```
public void add(Comparable newElement) {
    // Add a new leaf
elements.add(null);

    int index = elements.size() - 1;

    // Demote parents that are larger than the new element
    while (index > 1 && getParent(index).compareTo(newElement) > 0) {
elements.set(index, getParent(index));          index =
getParentIndex(index);
    }
    // Store the new element in the vacant slot
elements.set(index, newElement); }
```

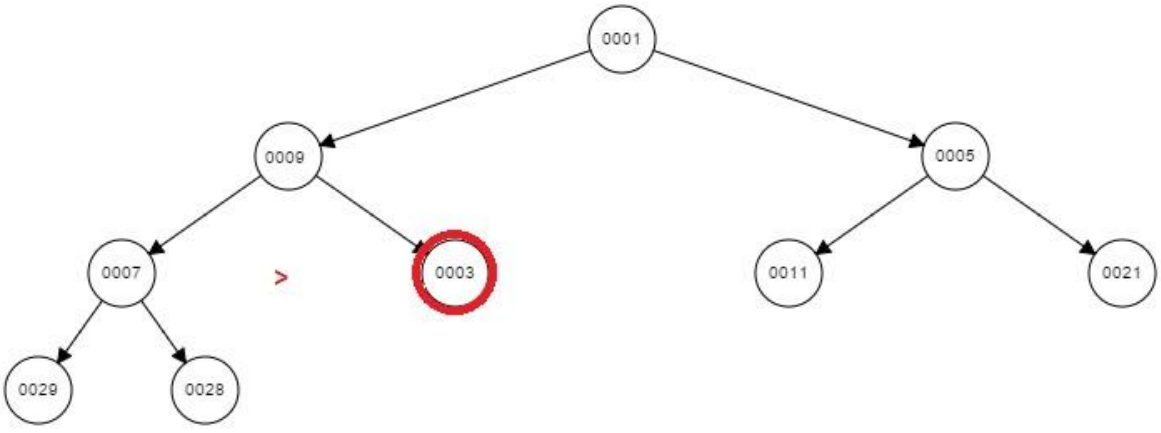
### 7.4.2 Remove

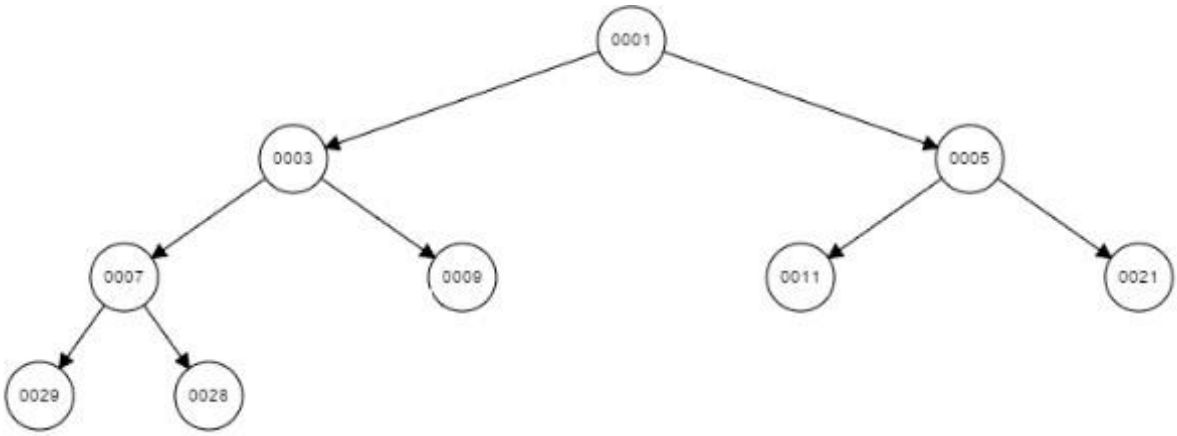
Algoritma remove Node pada heap (Minimum) dapat dilihat dibawah ini:

1. Swap Node pada slot terkahir pada tree dengan node yang akan dihapus.
2. Hapus Node terakhir pada tree.




3. Bandingkan nilai hasil swap Node tersebut dengan parentnya. Jika lebih kecil tukar dengan nilai parentnya. Lakukan juga perbandingan terhadap kedua anaknya, tukar jika lebih kecil.






```
public Comparable remove(Object o) {     int
deletingIndex = elements.indexOf(o);     int
lastIndex = elements.size() - 1;
    Comparable last = elements.get(lastIndex);
Comparable remove = elements.get(deletingIndex);
elements.remove(lastIndex);


    if(deletingIndex >= 1 && deletingIndex < lastIndex){
elements.set(deletingIndex,last);          fixHeap(deletingIndex); // private
method untuk membandingkan setiap node
    }     return
remove;
}
```

```
private void fixHeap(int index) {
    Comparable root = elements.get(index);

    int lastIndex = elements.size() - 1;
    // Promote children of removed root while they are smaller than last     if
(getLeftChildIndex(index) > (lastIndex - 1) && getLeftChildIndex(index) >
(lastIndex - 1)) {
return;
    }

    //int index = 1;
boolean more = true;
while (more) {
        int childIndex = getLeftChildIndex(index);
int comp;
        Comparable temp;

        if (getParent(index) != null) {
            comp = getParent(index).compareTo(elements.get(index));
if (comp > 0) {                  temp = getParent(index);
                elements.set(getParentIndex(index), elements.get(index));
elements.set(index, temp);
            }
        }

        if (childIndex <= lastIndex) {
            // Get smaller child

            // Get left child first
            Comparable child = getLeftChild(index);

            // Use right child instead if it is smaller
if (getRightChildIndex(index) <= lastIndex &&
                                getRightChild(index).compareTo(child) < 0) {
                childIndex = getRightChildIndex(index);
child = getRightChild(index);
            }

            // Check if larger child is smaller than root
if (child.compareTo(root) < 0) {
                // Promote child
                elements.set(index, child);
index = childIndex;
            } else {
                // Root is smaller than both children
more = false;
            }
            elements.set(index, root);
        } else {
            // No children
more = false;
        }
    }
    // Store root element in vacant slot }
```

## 8.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

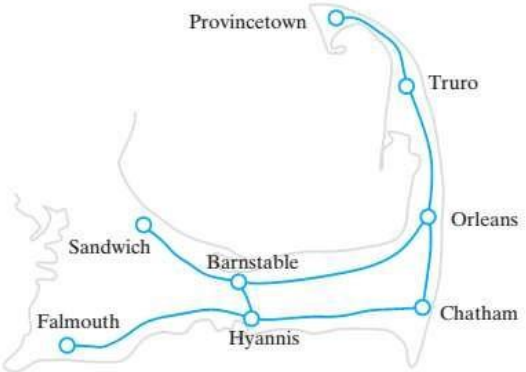
1. Mengenal konsep graph tak berarah
2. Mampu mengimplementasikan konsep graph dalam bahasa pemrograman

## 8.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

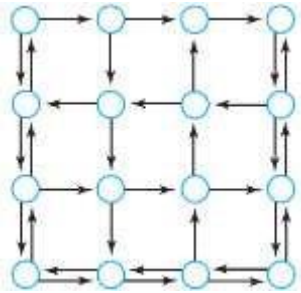
## 8.3 Graph

Graph merupakan himpunan tidak kosong dari node (vertec) dan garis penghubung (edge). Salah satu contoh nyata graph adalah peta sederhana dari daerah Cape Code, Massachusets yang dapat dilihat pada gambar 8-1 dibawah. Lingkaran merepresentasikan kota sedangkan garis merepresentasikan jalan. Didalam graph lingkaran tersebut diberikenal dengan *vertex* sedangkan garis merupakan *edge*.



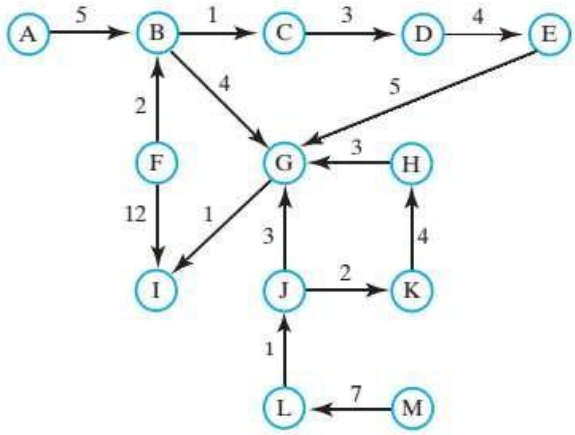

**Gambar 8.1 Graph (Carrano, 2014)**

Karena kita dapat melakukan perjalanan di kedua arah sepanjang jalan pada Gambar 8-1, maka grafik tersebut dikatakan tidak berarah (*undirected graph*). Sedangkan graph di Gambar 8.2 memiliki vertex untuk setiap persimpangan yang masing-masing memiliki arah. Sehingga dikatakan graph berarah atau *directed graph* (digraph).



Gambar 8.2 Directed Graph (Carrano, 2014)

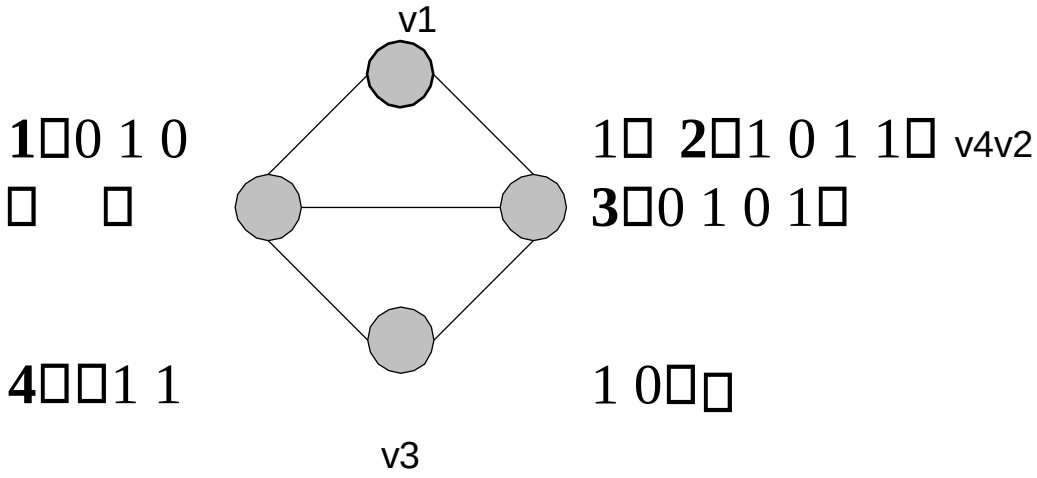
Selain arah, beban atau nilai sering ditambahkan pada edge. Nilai tersebut biasanya merepresentasikan jarak, biaya transportasi, dan lain-lain.




Gambar 8.3 Weigthed Graph

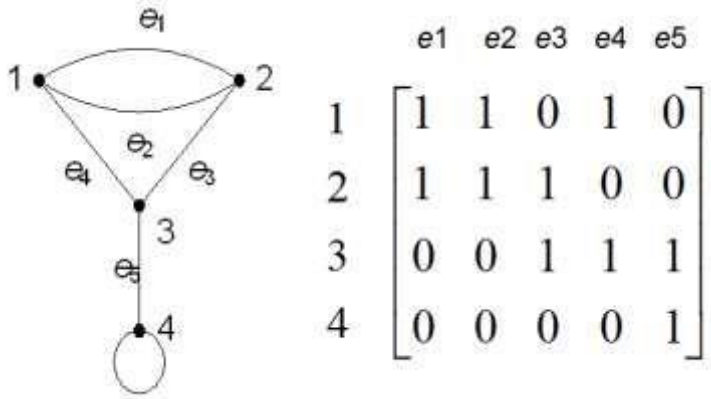
Representasi graph dalam Bahasa pemrograman terdiri dari tiga macam, yaitu:

1. Adjacency Matrix atau matriks ketetanggaan, kolom dan baris pada matriks ini merepresentasikan hubungan antara vertex satu dan lainnya.



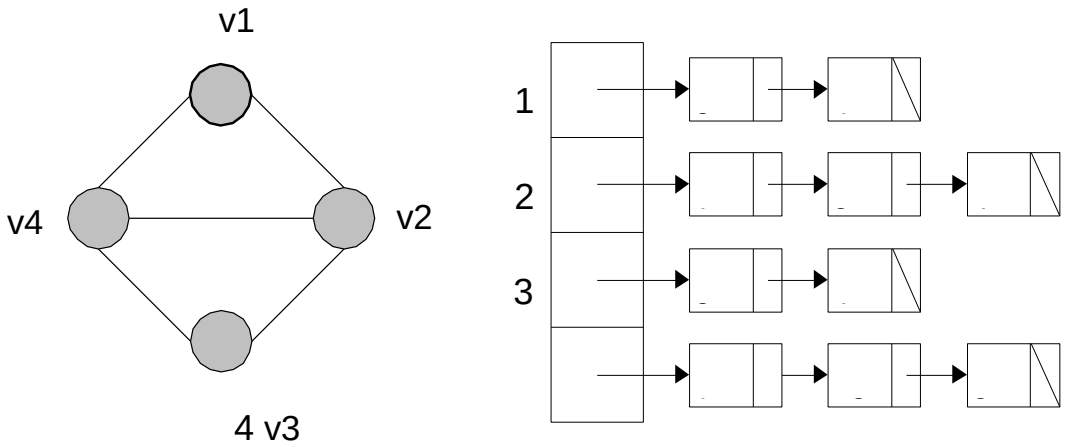

Gambar 8.4 Adjacency Matrix

2. Incidency Matrix, atau matriks bersisian, kolom dan baris pada matriks ini merepresentasikan hubungan antara vertex dan sisi/edge yang menempel pada vertex tersebut




Gambar 8.5 Incidency Matrix

3. Adjacency List, list ketetanggaan, yang merepresentasikan hubungan antara vertex satu dan lainnya.




Gambar 8.6 Adjacency List

## 8.4 Implementasi

Pada praktikum ini, untuk merepresentasikan graph menggunakan adjacency matrix. Representasi struktur graph menggunakan matrix dapat dilihat dibawah ini.

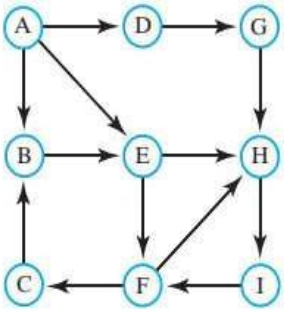
```
public interface GraphInterface {
    public void insertVertex(char labelVertex);
    public void deleteVertex(char labelVertex);

    public void insertEdge(char vertex1, char vertex2);
    public void deleteEdge(char vertex1, char vertex2);

    public void displayDFS();
    public void displayBFS();

    public void display();

}
```

```
public class Graph implements GraphInterface {
final int DEFAULTMAX = 100;


    private int[][] data = new int[DEFAULTMAX][DEFAULTMAX];
private char[] label = new char[DEFAULTMAX]; // Label vertex
private int jumlahVertex;                          // Jumlah vertex


    StackInterface stack = new StackLL();


    public Graph(int maximum) {
data = new int[maximum][maximum];
jumlahVertex = 0;
    }
    private int find(char labelVertex){
for(int i = 0; i <=jumlahVertex; i++){
if (label[i] == labelVertex)
return i;
        }
return -1;
    }


    @Override
    public void insertVertex(char labelVertex) {
if(jumlahVertex == data.length-1) {
System.out.println("FULL");             return;
        }


        if(find(labelVertex) > -1){
            System.out.println("Sudah ada");
return;
        }


        for (int i = 0; i <= jumlahVertex; i++) {
data[i][jumlahVertex] = 0;             data[jumlahVertex][i]
= 0;
        }


        label[jumlahVertex] = labelVertex;


        jumlahVertex++;
    }
                    data[k][j] = data[k][j+1];
            }
        }


        jumlahVertex--;
    }


    @Override
    public void insertEdge(char vertex1, char vertex2) {
int pos1 = find(vertex1);         int pos2 =
find(vertex2);
        if ( (pos1 == -1) || (pos2 == -1) ) {
            System.out.println("Ada vertex yang tidak dikenal" );
return;
        }


        // --- Sisipkan ke
array          data[pos1][pos2]
= 1;          data[pos2][pos1] =
1;      }
```

## 8.5 Traversal Graph

### 8.5.1 Bread-first Search

Dalam Bread-first search, kunjungi semua tetangga dari vertex asal, sebelum mengunjungi tetangga dari tetangga vertex asal. Traversal BFS menggunakan queue sebagai bantuan. Ketika dilakukan dequeue maka semua tetangga yang belum dikunjungi dari vertex tersebut masuk ke dalam queue, begitu selanjutnya hingga seluruh vertex telah dikunjungi. Queue kedua (traversal order) dapat digunakan untuk menyimpan urutan kunjungan yang telah dilakukan.




| frontVertex | nextNeighbor | Visited vertex | vertexQueue (front to back) | traversalOrder (front to back) |
|---|---|---|---|---|
| | | A | A | A |
| A | | | *empty* | |
| | B | B | B | AB |
| | D | D | BD | ABD |
| | E | E | BDE | ABDE |
| B | | | DE | |
| D | | | E | |
| | G | G | EG | ABDEG |
| E | | | G | |
| | F | F | GF | ABDEGF |
| | H | H | GFH | ABDEGFH |
| G | | | FH | |
| F | | | H | |
| | C | C | HC | ABDEGFHC |
| H | | | C | |
| | I | I | CI | ABDEGFHCI |
| C | | | I | |
| I | | | *empty* | |

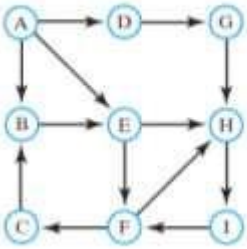
```
public void displayBFS() {
    QueueInterface traversalOrder = new QueLL();
    QueueInterface vertexQueue = new QueLL();

    char[] simpulDikunjungi = new char[data.length];
for (int i = 0; i < jumlahVertex; i++)
simpulDikunjungi[i] = 'x';

    simpulDikunjungi[0] = 'v';
traversalOrder.enQueue(label[0]);
vertexQueue.enQueue(label[0]);
    while(!vertexQueue.isEmpty()){
        char frontVertex = (char)vertexQueue.deQueue();
int posisi = find(frontVertex);          for (int j = 1;
j < jumlahVertex; j++)              if (data[posisi][j]
== 1)
                if (simpulDikunjungi[j] == 'x') {
vertexQueue.enQueue(label[j]);
traversalOrder.enQueue(label[j]);                       simpulDikunjungi[j]
= 'v';
                }
    }
    while (!traversalOrder.isEmpty()){
        System.out.print(traversalOrder.deQueue() + " - ");
    }
}
```

Gambar 8.7 Bread-First Search

### 8.5.2 Depth-first Search

Dalam depth-first search, dilakukan eksplorasi terhadap vertex yang terdalam. Algoritma ini dimulai dengan mengunjungi vertex asal, kemudian tetangga dari vertex tersebut, tetangga dari tetangga, dan sebagainya, maju sejauh mungkin dari vertex awal. Gunakan stack dalam membangun DFS.

Kita mulai dengan memasukan vertex asal ke dalam stack. Ketika top vertex memiliki tetangga yang belum dikunjungi, kunjungi tetangga tersebut dan masukan ke dalam stack. Jika sudah tidak ada lagi tetangga dari vertex tersebut maka dipopkan dari stack. Urutan dari traversal ini dapat dibantu dengan menggunakan queue seperti pada gambar dibawah ini.




| topVertex | nextNeighbor | Visited vertex | vertexStack (top to bottom) | traversalOrder (front to back) |
|---|---|---|---|---|
| | | A | A | A |
| A | | | A | |
| | B | B | BA | AB |
| B | | | BA | |
| | E | E | EBA | ABE |
| E | | | EBA | |
| | F | F | FEBA | ABEF |
| F | | | FEBA | |
| | C | C | CFEBA | ABEFC |
| C | | | FEBA | |
| F | | | FEBA | |
| | H | H | HFEBA | ABEFCH |
| H | | | HFEBA | |
| | I | I | IHFEBA | ABEFCHI |
| I | | | HFEBA | |
| H | | | FEBA | |
| F | | | EBA | |
| E | | | BA | |
| B | | | A | |
| A | | | A | |
| | D | D | DA | ABEFCHID |
| D | | | DA | |
| | G | | GDA | ABEFCHIDG |
| G | | | DA | |
| D | | | A | |
| A | | | *empty* | ABEFCHIDG |

```
public void displayDFS() {
    QueueInterface traversalOrder = new QueLL();
    StackInterface vertexStack = new StackLL();


    char[] simpulDikunjungi = new char[data.length];
for (int i = 0; i < jumlahVertex; i++)
simpulDikunjungi[i] = 'x';


    vertexStack.push(label[0]);


    while(!vertexStack.isEmpty()){
        char topVertex = (char) vertexStack.pop();


        int posisi = find(topVertex);
if(simpulDikunjungi[posisi] == 'x') {
traversalOrder.enQueue(topVertex);
simpulDikunjungi[posisi] = 'v';
        }
        for (int j = 0; j < jumlahVertex; j++) {
if (data[posisi][j] == 1) {                 if
(simpulDikunjungi[j] == 'x') {
vertexStack.push(label[j]);
                }
            }
        }
    }
    while (!traversalOrder.isEmpty()){
        System.out.print(traversalOrder.deQueue() + " - ");
    }
}
```

Gambar 8.8 Depth-First Search

# Modul 9 : Graph

## 9.1 Tujuan

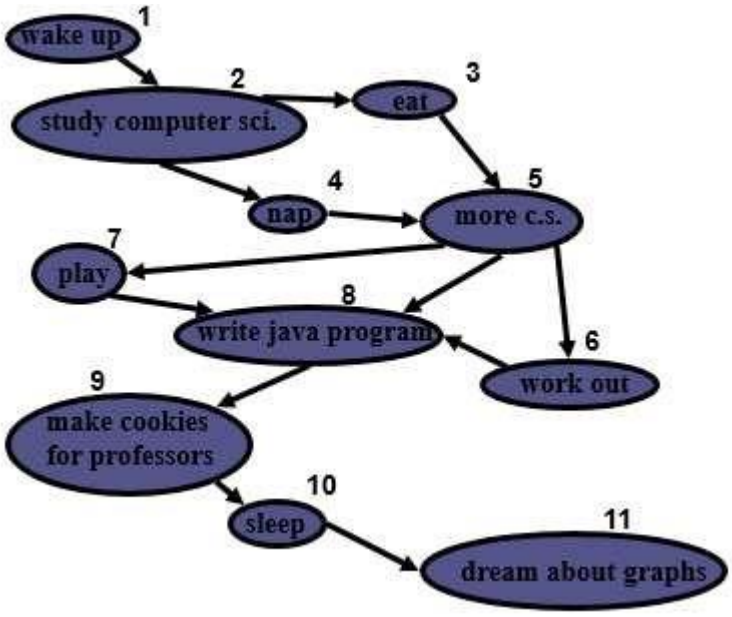
Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep graph tak berarah
2. Mampu mengimplementasikan konsep graph berarah untuk menyelesaikan suatu kasus dalam bahasa pemrograman.

## 9.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

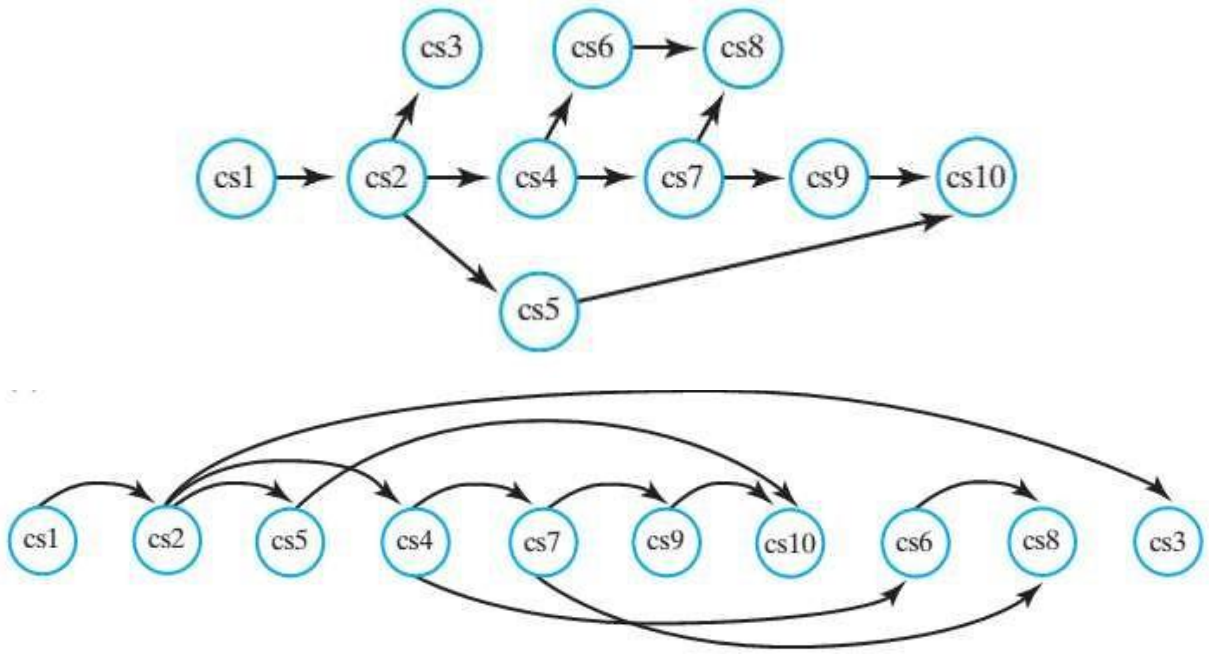
## 9.3 Topological Sort




Gambar 9.1 Graph Keseharian Mahasiswa

Pada gambar 9.1 diatas menggambarkan keseharian mahasiswa. Gambar tersebut adalah gambar graph yang tidak cyclic. Graph yang tidak cyclic dapat digambarkan dengan keterurutan linear, atau dapat disebut topological order. Diberikan urutan partial dari elemen suatu himpunan, dikehendaki agar elemen yang terurut partial tersebut mempunyai keterurutan linier. Contoh dari keterurutan partial banyak dijumpai dalam kehidupan sehari-hari, misalnya:

1. Dalam suatu kurikulum, suatu mata pelajaran mempunyai prerequisit mata pelajaran lain. Urutan linier adalah urutan untuk seluruh mata pelajaran dalam kurikulum.




**Gambar 9.2 Graph Prerequisit Matakuliah Beserta Salah Satu Topological Ordernya**

2. Dalam suatu proyek, suatu pekerjaan harus dikerjakan lebih dulu dari pekerjaan lain (misalnya membuat fondasi harus sebelum dinding, membuat dinding harus sebelum pintu. Namun pintu dapat dikerjakan bersamaan dengan jendela.

3. Dalam pembuatan tabel pada basis data, tabel yang di-refer oleh tabel lain harus dideklarasikan terlebih dulu. Jika suatu aplikasi terdiri dari banyak tabel, maka urutan pembuatan tabel harus sesuai dengan definisinya.

Proses dalam menemukan topological order dari suatu graph dinamakan **topological sort**. Algoritma topological sort memiliki beberapa pendekatan yaitu dengan menggunakan DFS(mencari vertex terdalam) dan Indegree Vertex. Pada topological sort menggunakan DFS, dimulai dengan mencari vertex yang tidak memiliki successor / tetangga. Tandai vertex tersebut dan masukan ke dalam stack. Kemudian lakukan pencarian vertex *u* lainnya yang belum dikunjungi dan tidak memiliki tetangga / tetangga tersebut telah dikunjungi. Masukan *u* dalam stack dan tandai telah dikunjungi. Lakukan proses ini hingga semua vertex telah dikunjungi. Pada saat semua vertex telah dikunjungi, stack penyimpanan berisi urutan dari topological order dari graph tersebut. Algoritma topological order dengan menggunakan DFS dapat dilihat dibawah ini:

```
Algorithm getTopologicalOrder()
vertexStack = a new stack to hold vertices as they are
visited numberOfVertices = number of vertices in the graph
for (counter = 1 to numberOfVertices)
{
 nextVertex = an unvisited vertex whose neighbors, if any, are all visited
Mark nextVertex as visited  vertexStack.push(nextVertex)
}
return vertexStack
```

Pada pendekatan indegree vertex, topological sort dimulai dengan mencari jumlah indegree dari sebuah vertex. Masukan semua vertex yang memiliki 0 indegree, tandai telah dikunjungi. Pop-kan top dari stack masukan ke dalam queue. Cari semua tetangga dari top stack tersebut, dan kurangi satu indegree-nya. Jika terdapat tetangga yang memiliki indegree menjadi 0, masukan ke dalam stack, tandai telah dikunjungi. Pop-kan kembali stack yang berisi vertex, dan ulangi proses pengurangan indegree dari setiap tetangga dari vertex tersebut.

```
public void topologicalSort() {
    int[] indegree = new int[data.length];
    StackInterface zeroIndegree = new StackLL();
QueueInterface toposort = new QueLL();      char[]
simpulDikunjungi = new char[data.length];

    for (int i = 0; i < jumlahVertex; i++) //set all vertex unvisited
simpulDikunjungi[i] = 'x';

    for(int i = 0; i < jumlahVertex; i++) //set all vertex 0 indegree
indegree[i] = 0;

    for(int vertex = 0; vertex < jumlahVertex; vertex++){ //calculate indegree
for(int neighbour = 0; neighbour < jumlahVertex; neighbour++){
if(data[vertex][neighbour] == 1){                  indegree[neighbour]++;
            }
        }
    }
    for (int i = 0; i < jumlahVertex; i++){    //collect all 0 indegree vertex
if (indegree[i] == 0 ) {
            …….                                  //push to stack, mark visited
            …….
        }
    }

    while (!zeroIndegree.isEmpty()){
        …….                                     //pop top of stack
        …….                                     //insert to queue

        for (int neighbour = 0; neighbour < jumlahVertex; neighbour++){
if(data[vertex][neighbour] == 1){        // find all neighbour
                …….                              // decrease indegree
            }
            if (indegree[neighbour] == 0 && simpulDikunjungi[neighbour]=='x') {
                …….                              //push to stack if indegree = 0
                …….                              //mark visited
            }
        }
    }
}
```

# Modul 10 : Hash

## 10.1 Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep hash
2. Mampu mengimplementasikan konsep hash untuk melakukan searching unlinear.

## 10.2 Alat & Bahan

Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC.

## 10.3 Hashing

Hashing adalah teknik yang digunakan untuk menemukan elemen pada suatu struktur data, tanpa perlu melakukan pencarian linear pada struktur data tersebut. Hasil dari hashing diletakkan pada suatu *hash table*, yang biasanya berbentuk array. Indeks hash table ditentukan melalui hashing. Implementasi hashing dapat berupa sets dan map (kadang disebut dictionary). Indeks suatu hash table diperoleh dengan melakukan komputasi pada suatu search key dengan menggunakan **hash function**, hasilnya berupa bilangan integer yang menandakan indeks array dimana suatu elemen akan disimpan.

Hash code yang dihasilkan melalui hash function akan berbeda antara satu objek dengan objek lainnya. Hash code yang dihasilkan harus dapat mencakup seluruh key search yang ada. Kelas Object memiliki method hashCode yang harus di-override oleh suatu kelas yang akan membuat hash code.

1. Hash Code untuk integer.

   Hash code dihitung dari search key, kemudian mampatkan hash code ke dalam jangkauan indeks hash table dengan menghitung modulus dari array.

```
int h = x.hashCode(); if
(h < 0) h = -h;
position = h % buckets.length;
```

2. Hash Code untuk String

   Hash code untuk String merupakan kombinasi dari character value yang menyusun string tersebut. Character value merupakan Unicode dari setiap karakter pembentuk string. Diabawah ini merupakan perhitungan hashCode dari library standar java.

```
final int HASH_MULTIPLIER = 31; //gunakan bilangan prima untuk hasil terbaik
int h = 0;
for (int i = 0; i < s.length(); i++) h
= HASH_MULTIPLIER * h + s.charAt(i);
```

3. Has Code untuk Object

   Untuk menghitung hash code dari class object, semua instance variable dari class tersebut harus dihitung. Contoh jika memiliki class object Buku, yang memiliki variable nama dalam String dan harga dalam int, maka kita harus menghitung hash code untuk nama dan hash code untuk harga dan kemudian ditambahkan hasil perhitungan dari hash code dari setiap variable tersebut.

```
class Buku {
    public int hashCode() {
        int h1 = nama.hashCode(); //String memiliki standar method u/
                                  //menghitung hash code
        int h2 = new Integer(harga).hashCode();
final int HASH_MULTIPLIER = 29;          int h =
HASH_MULTIPLIER * h1 + h2;          return h;
    }
}
```

## 10.4 Collision

Fungsi hash yang baik harus memenuhi syarat dapat menghitung hashcode dengan cepat dan dapat meminimalisir collision. Collision adalah bentroknya banyak data pada satu index karena memiliki key yang sama. Untuk meminimalisir collision, terdapat beberapa metode untuk menentukan lokasi kosong(**probing**), yaitu:

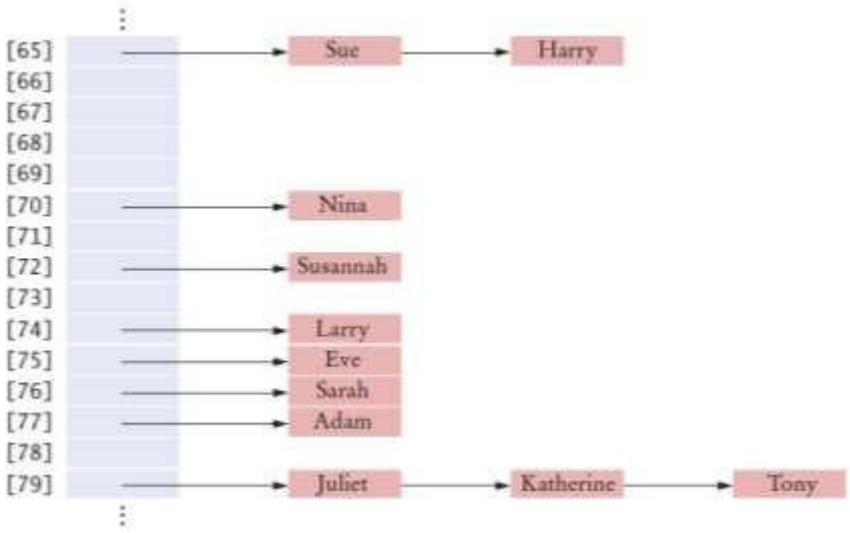
1. Linear Probing, mencari lokasi kosong secara linear. Bila collision terjadi pada index $k$, dimana $k$ merupakan hashCode pada key $x$, maka $k = k + 1$.

2. Quadratic Probing mencari lokasi kosong secara tidak linear, Bila collision terjadi pada index $k$, dimana $k$ merupakan hashCode pada key $x$, maka $k = k + j2$, dimana $j \geq 0$.

3. Double Hash Probing, menggunakan fungsi hash kedua berdasarkan key yang ada untuk mencari tempat kosong. Double Probing dihitung dari kombinasi $h1$ dan $h2$. Pada Double Hash Probe, terdapat beberapa syarat yang harus dipenuhi untuk hash function yang kedua, yaitu:
   a. Hash function yang kedua harus berbeda dengan hash yang pertama
   b. Harus berdasarkan search key
   c. Menghasilkan nonzero value

```
h1(key) = key modulo 29
h2(key) = 7 - key modulo 7
```

Kekurangan dari metode probing adalah memunculkan clustering data. Sehingga memperlambat proses pencarian. Untuk menghindari collision dan clustering, maka dapat digunakan struktur hash Table yang berbeda sehingga setiap lokasi dapat merepresentasikan lebih dari satu nilai/ data. Lokasi ini disebut dengan buket, dan metode ini disebut sparate chaining. Pada subbab selanjutnya akan dibahas mengenai implementasi dari sparate chaining.




**Gambar 10.1 Gambaran Sparate Chaining**

## 10.5 Implementasi

Class HashNode

```
package HashMaps;

public class HashNode<K,V> {
 K key;
 V value;
 HashNode<K, V>next;
 public HashNode()
 {
        this.key=key;
        this.value=value;
 }
}
```

Class Map

```
package com.company;

import java.util.ArrayList;

public class Map<K, V> {

    ArrayList<HashNode<K, V>> bucket = new ArrayList<>();
int numBuckets;     int size;

    public Map(int lengthOfBuckets) {
this.numbuckets = lengthOfBuckets;          for (int
i = 0; i < lengthOfBuckets; i++) {
bucket.add(null);
        }
    }
```

```
    public int getSize() {
return size;
    }
    public boolean isEmpty() {
return size == 0;
    }
    private int getBucketIndex(K key) {
int hashCod = key.hashCode();
return hashCod % numBuckets;
    }
    public V get(K key) {
        int index = getBucketIndex(key);
        HashNode<K, V> head = bucket.get(index);
while (head != null) {              if
(head.key.equals(key)) {                    return
head.value;
            }
            head = head.next;
        }
        return null;
    }
```

```
    public V remove(K key) {
        int index = getBucketIndex(key);
        HashNode<K, V> head = bucket.get(index);
if (head == null) {                 return null;
        }
        if (head.key.equals(key)) {
V val = head.value;                 head
= head.next;
bucket.set(index, head);
size--;                 return val;
        } else {
            HashNode<K, V> prev = null;
while (head != null) {

                if (head.key.equals(key)) {
prev.next = head.next;                          size--;
                    return head.value;
                }
                prev = head;
head = head.next;
            }
size--;
return null;
        }
    }
    public void add(K key, V value) {
int index = getBucketIndex(key);
        System.out.println(index);
        HashNode<K, V> head = bucket.get(index);
HashNode<K, V> toAdd = new HashNode<>();
toAdd.key = key;          toAdd.value = value;
```

```
        if (head == null) { // jika head kosong add pada index
tsb             bucket.set(index, toAdd);             size++;
} else {
            while (head != null) {
                 if (head.key.equals(key)) {
head.value = value; //jika key sama maka update value
size++;                      break;
                }
                head = head.next; //jika key tdk sama maka cek node selanjutnya
            }
            if (head == null) { //jika tidak ada key yang sama maka tambahkan pda
head = bucket.get(index); //awal list                toAdd.next = head;
bucket.set(index, toAdd);                 size++;
            }
}
        if ((1.0 * size) / numBuckets > 0.7) { //jika telah terisi lbh dr 70%
            //do something - dua kalikan panjang dari arraylist
            ArrayList<HashNode<K, V>> tmp = bucket;           bucket = new
ArrayList<>();              numBuckets = 2 * numBuckets; //bucket baru dengan
panjang 2x lipat              for (int i = 0; i < numBuckets; i++) {
bucket.add(null);
            }
            for (HashNode<K, V> headNode : tmp) { //isikan semua data dari awal
while (headNode != null) {                      add(headNode.key,
headNode.value);                    headNode = headNode.next;
                }
            }
        }

    }
}
```

*Sumber: geekforgeeks.com dengan perubahan*