

MODUL PRAKTIKUM
ALGORITMA DAN STRUKTUR DATA
(MII211204)



LABORATORIUM KOMPUTER DASAR
DEPARTEMEN ILMU KOMPUTER DAN ELEKTRONIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS GADJAH MADA
2023

DAFTAR ISI

<i>BAB I PENGENALAN JAVA DAN OBJECT ORIENTED PROGRAMMING</i>	1
1.1 Tujuan Praktikum	1
1.2 Materi	1
1.2.1 Basic Syntax	1
1.2.2 Java Primitive Types	3
1.2.3 Variabel, Konstanta, dan <i>Assignments</i>	3
1.2.4 Komentar	4
1.2.5 <i>Operator</i>	5
1.2.6 <i>Flow Control</i>	6
1.2.7 <i>Input</i>	7
1.2.8 <i>Object Oriented Programming (OOP)</i> pada Java	9
1.2.9 <i>Constructor</i>	11
1.2.10 <i>Encapsulation</i>	11
1.2.11 <i>Inheritance</i>	13
1.2.12 <i>Polymorphism</i>	16
1.2.13 <i>Abstract</i>	18
1.2.14 <i>Interface</i>	20
1.3 Tugas	23
<i>BAB II ARRAY DAN LINKED LIST BESERTA APLIKASINYA</i>	25
2.1 Tujuan Praktikum	25
2.2 Materi	25
2.2.1 <i>Array</i>	25
2.2.2 <i>Linked List</i>	27
2.2.3 <i>Stack</i>	33
2.2.4 <i>Queue</i>	37
2.3 Tugas	44
<i>BAB III TREE & BINARY TREE</i>	46
3.1 Tujuan Praktikum	46
3.2 Materi	46

3.2.1 <i>Tree</i>	46
3.2.2 <i>Binary Tree</i>	47
3.2.3 <i>Binary Search Tree</i>	49
3.3 Tugas	53
<i>BAB IV BALANCED TREE: AVL</i>	54
4.1 Tujuan Praktikum	54
4.2 Materi	54
4.2.1 AVL Tree	54
4.2.2 Insertion.....	55
4.2.3 Deletion	61
4.3 Tugas	70
<i>BAB V SORTED TREE: HEAP TREE</i>	71
5.1 Tujuan Praktikum	71
5.2 Materi	71
5.2.1 Array Implementation	72
5.2.2 Insert.....	73
5.2.3 Remove.....	74
5.2.4 Print	75
5.3 Tugas	76
<i>BAB VI GRAF</i>	77
6.1 Tujuan Praktikum	77
6.2 Materi	77
6.2.1 Graf.....	77
6.2.2 Implementasi Java	79
6.3 Tugas	81
<i>BAB VII SHORTEST PATH PROBLEM DENGAN ALGORITMA DIJKSTRA</i>	82
7.1 Capaian Pembelajaran	82
7.2 Materi	82
7.2.1 Algoritma Dijkstra.....	82
7.2.2 Implementasi Java	87
7.3 Tugas	89
<i>BAB VIII MINIMUM SPANNING TREE DENGAN ALGORITMA PRIM</i>	91

8.1 Tujuan Pembelajaran	91
8.2 Materi	91
8.2.1 Minimum Spanning Tree Problem	91
8.2.2 Algoritma Prim.....	92
8.2.3 Implementasi Java untuk Algoritma Prim.....	96
8.3 Tugas	99
<i>BAB IX DISJOINT SET</i>	100
9.1 Tujuan Pembelajaran	100
9.2 Materi	100
9.2.1 Disjoint Set.....	100
9.2.2 Implementasi	101
9.2.3 Implementasi Java	104
9.3 Tugas	107
<i>BAB X STRING MATCHING</i>	108
10.1 Tujuan Pembelajaran	108
10.2 Materi	108
10.2.1 String Matching	108
10.2.2 Algoritma Naïve	109
10.2.3 Algoritma KMP	110
10.2.4 Implementasi Java	113
10.3 Tugas	114
<i>BAB XI ALGORITMA GEOMETRI – CONVEX HULL</i>	116
11.1 Tujuan Pembelajaran	116
11.2 Materi	116
11.2.1 Convex Hull	116
11.2.2 Algoritma Monotone Chain.....	117
11.2.3 Implementasi Java	120
11.3 Tugas	123

BAB I

PENGENALAN JAVA DAN OBJECT ORIENTED PROGRAMMING

1.1 Tujuan Praktikum

1. Praktikan mulai dapat mengenal bahasa pemrograman Java
2. Praktikan dapat menggunakan bahasa pemrograman Java untuk membuat program sederhana
3. Praktikan dapat memahami konsep *object oriented programming (OOP)* pada Java
4. Praktikan dapat memahami fitur-fitur yang bisa digunakan pada Java beserta keunggulan dari bahasa Java dibanding bahasa pemrograman lain.

1.2 Materi

Java adalah bahasa pemrograman yang dapat membuat seluruh bentuk aplikasi, desktop, *web*, *mobile* dan lainnya, sebagaimana dibuat dengan menggunakan bahasa pemrograman konvensional yang lain. Bahasa Pemrograman Java ini berorientasi objek (*OOP-Object Oriented Programming*) dan dapat dijalankan pada berbagai platform sistem operasi. Perkembangan *Java* tidak hanya terfokus pada satu sistem operasi, tetapi dikembangkan untuk berbagai sistem operasi dan bersifat *open source*. Dengan slogannya. “*Write once, run anywhere*” .

1.2.1 Basic Syntax

Java dapat didefinisikan sebagai sebuah kumpulan objek yang yang saling berkaitan dengan beberapa *method*. Beberapa istilah yang perlu dipahami dalam Java, diantaranya *object*, *class*, *method* dan *instance variable*.

- **Object** – Objek memiliki bentuk dan perilaku, contohnya anjing mempunyai warna, jenis dan mungkin julukan/nama. Begitupun anjing juga mempunyai beberapa perilaku seperti mengibaskan ekor, menggonggong maupun makan. Suatu objek merupakan suatu contoh/*instance* dari suatu *class*.

- **Class** – Sebuah *class* dapat diartikan sebagai cetakan untuk membuat suatu bentuk atau perilaku yang mendukung objek yang akan dibuat
- **Method** – Sebuah method dapat diartikan sebagai perilaku. Sebuah *class* dapat berisi beberapa *method*. Dalam *method* ini berisi tentang logika kita, data yang terolah dan tentunya dapat dieksekusi
- **Instance Variable** – Setiap objek memiliki identitas unik berupa kumpulan *instance variables*. Sebuah bentuk objek dibuat dengan nilai yang dimasukkan ke *instance variable* yang dimilikinya.

Mari kita simak contoh program berikut yang mencetak tulisan Hello World.



```
1  public class HelloWorld {
2      /* This is my first java program.
3      */
4      public static void main(String args[]) {
5          System.out.println("\nHello World"); // prints Hello World
6      }
7 }
```

Sekarang mari praktikkan bagaimana cara menyimpan, *compile* dan eksekusi programnya.

- Simpan file dengan nama: 'HelloWorld.java'.
- Buka *command prompt window* dan menuju directory dimana program itu dijalankan. Asumsikan di C:\.
- Ketik 'javac HelloWorld.java' dan tekan enter untuk *compile code*-nya. Jika tidak terjadi *error* maka *command prompt* akan membawamu menuju baris selanjutnya
- Sekarang ketikkan 'java HelloWorld' untuk menjalankan program.
- 'Hello World' akan muncul di layar.

Yang harus kita perhatikan juga dalam pemrograman java :

- **Case Sensitivity** – Java mempunyai sifat *case sensitive*, Hello dan hello bisa memiliki arti yang berbeda
- **Class Names** – Untuk nama *class*, huruf pertama harus huruf besar, begitupun jika terdiri beberapa kata, huruf depan kata harus huruf besar.

Contoh: class MyFirstJavaClass

- **Method Names** – Untuk nama class awalnya harus huruf kecil, tetapi jika terdiri beberapa kata maka awal kata dalam nama method tersebut harus huruf besar.
Contoh: `public void myMethodName()`
- **Program File Name** – Nama dari program harus sama dengan nama *class* utamanya. Ketika menyimpan filenya, biasanya tersimpan menggunakan nama *class* utamanya dan berakhiran '.java'. Kalau nama class dan nama program beda maka tidak akan bisa dilakukan *compiler*.
Contoh: Misalnya 'MyFirstJavaProgram' adalah nama *class*, maka *file* tersebut harus disimpan dengan nama 'MyFirstJavaProgram.java'
- **public static void main(String args[])** – Program *java* dimulai dari *method* yang berawalan *main()*

1.2.2 Java Primitive Types

Tipe variabel primitif atau *built-in* pada Java ditunjukkan pada Tabel 1.1

Tabel 1.1. Java Primitive Types

Tipe	Ukuran (bits)	Minimum	Maksimum	Contoh
byte	8	-2^7	2^7-1	<code>b = 100</code>
short	16	-2^{15}	$2^{15}-1$	<code>s = 30000</code>
int	32	-2^{31}	$2^{31}-1$	<code>i = 100000000</code>
long	64	-2^{63}	$2^{63}-1$	<code>l = 1000000000000000</code>
float	32	-10^{38}	$10^{38}-1$	<code>f = 1.456</code>
double	64	-10^{308}	$2^{308}-1$	<code>d = 1.456789101234</code>
char	16	0	$2^{16}-1$	<code>c = 'c'</code>
boolean	1	-	-	<code>a = false</code> <code>b = true</code>

1.2.3 Variabel, Konstanta, dan Assignments

- Deklarasi suatu variabel dengan tipe tertentu

```
type identifier;
int option;
```

- Deklarasi beberapa variabel dengan tipe yang sama, dipisahkan dengan koma
`type identifier1, identifier2, ... , identifierN;`
`double sum, difference, product, quotient;`
- Deklarasi variabel dan memberikan (*assign*) nilai awal
`type identifier = initialValue;`
`int magicNumber = 88;`
- Deklarasi banyak variabel dan memberikan (*assign*) nilai awal
`type identifier1 = initialValue1, ... , identifierN = initialValue2;`
`String greetingMsg = "Hi!", quitMsg = "Bye!";`
- Deklarasi Konstanta
`final type identifier = value; //perlu diinisialisasi`
`final double PI = 3.1415926;`
- Memberikan (*assign*) nilai literal (dari *right hand side - RHS*) ke variabel (dari *left hand side - LHS*)
`variable = literalValue;`
`number = 88;`
- Evaluasi ekspresi *RHS* dan memberikan (*assign*) hasilnya ke variabel *LHS*
`sum = sum + number;`

1.2.4 Komentar

Ada dua komentar yaitu *inline comment* dan *block comment*. *Inline Comment* yaitu komentar yang berada dalam satu baris, menggunakan tanda // .

Contoh :

```
// ini komentar
```

Block Comment yaitu komentar yang penjelasannya lebih dari satu baris, menggunakan tanda

```
/*....  
....  
....*/
```

Contoh :

```
/*ini komentar 1  
ini komentar 2
```

Ini komentar 3 */

1.2.5 Operator

Java mendukung beberapa operator aritmatika seperti yang terlihat pada Tabel 1.2

Tabel 1.2. Operator Aritmatika

Operator	Deskripsi	Penggunaan	Contoh
*	Perkalian	expr1 * expr2	$2 * 3 \rightarrow 6$ $3.3 * 1.0 \rightarrow 3.3$
/	Pembagian	expr1 / expr2	$1 / 2 \rightarrow 0$ $1.0 / 2.0 \rightarrow 0.5$
%	Modulo (Sisa Bagi)	expr1 % expr2	$5 \% 2 \rightarrow 1$ $-5 \% 2 \rightarrow -1$ $5.5 \% 2.2 \rightarrow 1.1$
+	Penjumlahan (atau <i>unary positive</i>)	expr1 + expr2	$1 + 2 \rightarrow 3$ $1.1 + 2.2 \rightarrow 3.3$
-	Pengurangan (atau <i>unary negative</i>)	expr1 - expr2	$1 - 2 \rightarrow -1$ $1.1 - 2.2 \rightarrow -1.1$

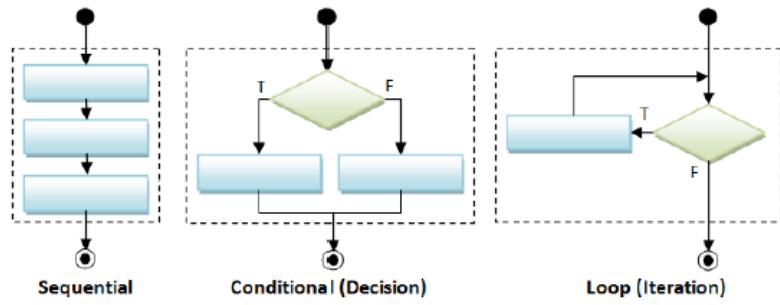
Selain *assignment operator* sederhana biasa (=) yang telah dijelaskan sebelumnya, Java juga menyediakan *compound assignment operator* seperti yang tercantum pada Tabel 1.3. Java juga bisa melakukan operasi increment/decrement, seperti pada C++.

Tabel 1.3. *Compound Assignment*

Operator	Deskripsi	Penggunaan	Contoh
=	<i>Assignment</i> <i>Assign the value of the LHS to the variable at the RHS</i>	var = expr	x = 5
+=	<i>Compound addition and assignment</i>	var += expr sama dengan var = var + expr	x += 5 sama dengan x = x + 5
-=	<i>Compound subtraction and assignment</i>	var -= expr sama dengan var = var - expr	x -= 5 sama dengan x = x - 5
*=	<i>Compound multiplication and assignment</i>	var *= expr sama dengan var = var * expr	x *= 5 sama dengan x = x * 5
/=	<i>Compound division and assignment</i>	var /= expr sama dengan var = var / expr	x /= 5 sama dengan x = x / 5
%=	<i>Compound remainder (modulus) and assignment</i>	var %= expr sama dengan var = var % expr	x %= 5 sama dengan x = x % 5

1.2.6 Flow Control

Terdapat tiga *flow control* dasar, yaitu *sequential*, *conditional* (atau *decision*), dan *loop* (atau *iteration*), seperti diperlihatkan pada contoh di bawah



Sintaks	Contoh
<pre>// if-then if (booleanExpression) { true-block ; }</pre>	<pre>// if-then if (mark >= 50) { System.out.println("Congratulation!"); System.out.println("Keep it up!"); }</pre>
<pre>// if-then-else if (booleanExpression) { true-block ; } else { false-block ; }</pre>	<pre>// if-then-else if (mark >= 50) { System.out.println("Congratulation!"); System.out.println("Keep it up!"); } else { System.out.println("Try Harder!"); }</pre>

1.2.7 Input

Java seperti bahasa lain juga mendukung sistem *input*, *output* dan *error*. Kita dapat melakukan input dengan keyboard via **System.in** (*standard input device*). Sedangkan untuk output dapat dilakukan dengan **System.out** (*standard output device*) dan error dengan **System.err** (*standard error device*) secara otomatis dari *console*. Berikut ini merupakan contoh program untuk menghitung *ideal fat-burning heart rate*.

The screenshot shows an IDE interface with two main panes. The top pane is titled "FatBurningHeartRate.java" and displays the following Java code:

```
1 package fatburningheartrate;
2
3 import java.util.Scanner;
4
5 public class FatBurningHeartRate {
6     public static void main(String args[]) {
7         // TODO code application logic here
8         Scanner input = new Scanner(System.in);
9
10        System.out.println("Enter your age in years: ");           //output
11        double age = input.nextDouble();                          //input
12        System.out.println("Enter your maximum heart rate: "); //output
13        double rate = input.nextDouble();                        //input
14
15        double fb = (rate-age) * 0.65;
16
17        System.out.println("Your ideal fat-burning heart rate is " + fb);
18    }
19
20 }
```

The bottom pane is titled "Output - Run (fatburningheartrate)" and shows the console output:

```
Enter your age in years:
24
Enter your maximum heart rate:
80
- Your ideal fat-burning heart rate is 36.4
-----
BUILD SUCCESS
-----
Total time: 30.744 s
```

1.2.8 Object Oriented Programming (OOP) pada Java

Java dirancang untuk menjadi bahasa yang sederhana, meminimalkan kesalahan, namun tetap tangguh. Hal ini yang membedakan Java dengan bahasa pemrograman yang lain. Suatu aplikasi Java ditulis dalam bahasa Java dan memanfaatkan Java *API* (*Application Programming Interface*). Java *API* berisi koleksi *class* siap pakai yang mempermudah dalam penulisan aplikasi.

Class, Object, Properties, Method

Class adalah merupakan cetakan, *template*, *prototype*, tempat dari object, sedangkan **object** adalah isi dari kelas itu sendiri. Satu *class* dapat mempunyai *object* lebih dari satu atau lebih. Contoh sederhananya seperti suatu cetakan jelly yang bisa menghasilkan banyak jelly. Contoh lain, belalang diibaratkan sebagai suatu objek yang punya nama, mata, kaki, sayap, warna, jenis. Belalang juga dapat terbang dan hinggap. Mata, kaki sayap dan warna belalang dalam dunia pemrograman disebut juga **atribut atau properties**. Sedangkan aktifitasnya yaitu terbang dan hinggap dalam dunia pemrograman disebut sebagai **method**.

Tabel 1.4 Beberapa kata di Java yang tidak bisa digunakan sebagai nama *class*

Reserved Words				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

Modifier

Modifier digunakan untuk menentukan hubungan suatu unsur kelas dengan unsur kelas lainnya. Berdasarkan akses kontrolnya, *modifier* dapat dibagi menjadi 4 tipe, yaitu:

- **Public** : semua unsur yang terdapat dalam suatu *class (method,object,dll)* bisa diakses secara bebas oleh semua *class* lain yang berada dalam satu *package* ataupun tidak.
- **Protected** : semua unsur yang terdapat dalam suatu *class (method,object,dll)* bisa diakses oleh semua *class* lain yang berada dalam satu *package* dan *class* bagian/turunan dari *class* awal meski berbeda *package*.
- **Default** : semua unsur yang terdapat dalam suatu *class (method,object,dll)* bisa diakses oleh semua *class* lain yang berada dalam satu *package*.
- **Private** : semua unsur yang terdapat dalam suatu *class (method,object,dll)* hanya bisa diakses oleh *class* itu sendiri.

Rangkuman perbedaan masing-masing *modifiers* dapat dilihat pada Tabel 1.5.

Tabel 1.5 Modifiers

Visibility	public	protected	default	private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in same package	Yes	Yes	Yes	No
Subclass outside the same package	Yes	Yes	No	No
Non-subclass outside the same package	Yes	No	No	No

Sebelumnya kita telah membahas tipe *access modifiers*, selanjutnya kita akan membahas beberapa tipe *non-access modifiers*, yaitu:

- **static** adalah salah satu jenis *modifier* di Java yang digunakan agar suatu atribut atau pun *method* dapat diakses oleh kelas atau objek tanpa harus melakukan instansiasi terhadap kelas tersebut. Method *main* adalah salah satu contoh *method* yang mempunyai *modifier static*.

- ***final*** adalah salah satu modifier yang digunakan agar suatu atribut atau *method* bersifat final atau tidak bisa diubah nilainya. *Modifier* ini digunakan untuk membuat konstanta di Java.
- ***abstract*** adalah modifier yang digunakan untuk membuat kelas dan method abstrak.

1.2.9 *Constructor*

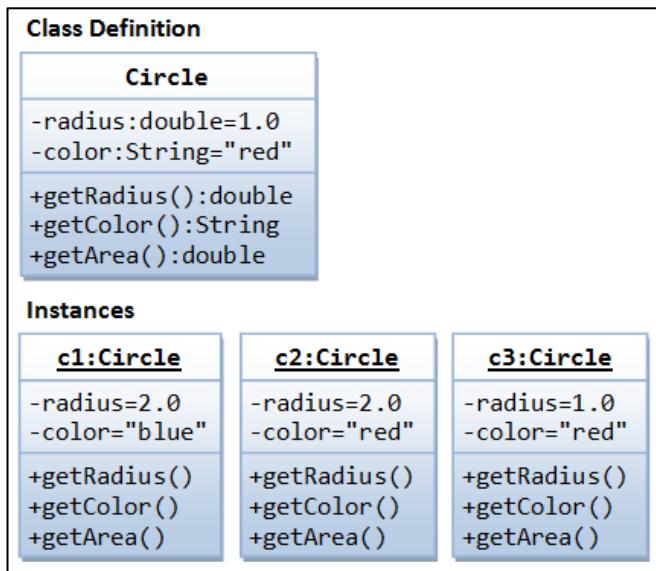
Constructor adalah *method* yang secara otomatis dipanggil/dijalankan ketika suatu class diinstansi atau dengan kata lain *constructor* adalah *method* yang pertama kali dijalankan pada saat sebuah objek pertama kali diciptakan. Jika dalam sebuah *class* tidak terdapat *constructor* maka secara otomatis Java akan membuatkan sebuah default *constructor*.

1.2.10 *Encapsulation*

Encapsulation adalah pembungkus, pembungkus disini dimaksudkan untuk menjaga suatu proses program agar tidak dapat diakses secara sembarangan atau diintervensi oleh program lain. Konsep enkapsulasi sangat penting dilakukan untuk menjaga kebutuhan program agar dapat diakses sewaktu-waktu, sekaligus menjaga program.

Dalam kehidupan sehari-hari enkapsulasi dapat dimisalkan sebagai arus listrik pada generator, dan sistem perputaran generator untuk menghasilkan arus listrik. Kerja arus listrik tidak mempengaruhi kerja dari sistem perputaran generator, begitu pula sebaliknya. Karena didalam arus listrik tersebut, kita tidak perlu mengetahui bagaimana kinerja sistem perputaran generator, apakah generator berputar kebelakang atau ke depan atau bahkan serong. Begitu pula dalam sistem perputaran generator, kita tidak perlu tahu bagaimana arus listrik, apakah menyala atau tidak. Begitulah konsep kerja dari enkapsulasi, dia akan melindungi sebuah program dari akses ataupun intervensi dari program lain yang mempengaruhinya. Hal ini sangat menjaga keutuhan program yang telah dibuat dengan konsep dan rencana yang sudah ditentukan dari awal.

Contoh penggunaan *constructor* dan *encapsulation* dapat dilihat pada pembuatan kelas *Circle* berikut ini.



Berikut ini merupakan *class Circle* yang disimpan sebagai **Circle.java**

```

1 package circle;
2
3 public class Circle { // Save as Circle.java
4     // Private instance variables
5     private double radius;
6     private String color;
7
8     // Constructors
9     public Circle(){
10         radius = 1.0;    // 1st Constructor
11         color = "red";
12     }
13
14     public Circle(double r){
15         radius = r;    // 2nd Constructor
16         color = "red";
17     }
18
19     public Circle(double r, String c){
20         radius = r;    // 3rd Constructor
21         color = c;
22     }
23
24     public double getRadius(){
25         return this.radius;
26     }
27
28     public String getColor(){
29         return this.color;
30     }
31
32     public double getArea(){
33         final double pi = 3.14;
34         return pi*radius*radius;
35     }
36 }
```

Berikut *class* untuk melakukan *testing* yang disimpan sebagai **TestCircle.java**

```

1 package circle;
2
3 public class TestCircle { // Save as TestCircle.java
4     public static void main(String[] args) { // Program entry point
5         // Declare and Construct an Instance of the Circle Class Called c1
6         Circle c1 = new Circle(2.0, "blue"); // Use 3rd Constructor
7         System.out.println("The radius is: " + c1.getRadius()); //use dot operator
8         System.out.println("The color is: " + c1.getColor());
9         System.out.printf("The area is: %.2f%n", c1.getArea());
10
11        // Declare and Construct another Instance of the Circle Class Called c2
12        Circle c2 = new Circle(2.0); // Use 2nd Constructor
13        System.out.println("The radius is: " + c2.getRadius());
14        System.out.println("The color is: " + c2.getColor());
15        System.out.printf("The area is: %.2f%n", c2.getArea());
16
17        // Declare and Construct yet another Instance of the Circle Class Called c3
18        Circle c3 = new Circle(); // Use 2nd Constructor
19        System.out.println("The radius is: " + c3.getRadius());
20        System.out.println("The color is: " + c3.getColor());
21        System.out.printf("The area is: %.2f%n", c3.getArea());
22    }
23}

```

Berikut hasil ketika program tersebut dijalankan

```

Output - Run (circle) x
The radius is: 2.0
The color is: blue
The area is: 12.56
The radius is: 2.0
The color is: red
The area is: 12.56
The radius is: 1.0
The color is: red
The area is: 3.14
-----
BUILD SUCCESS
-----
Total time: 3.947 s

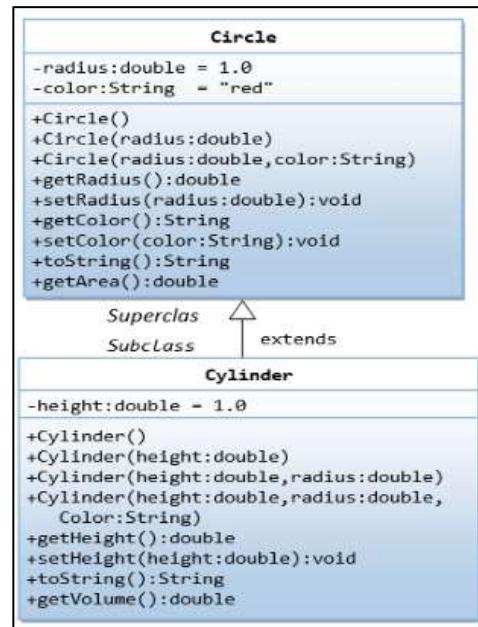
```

1.2.11 Inheritance

Inheritance (penurunan sifat / pewarisan) merupakan ciri khas dari *OOP* yang tidak terdapat pada pemrograman prosedural gaya lama. Dalam hal ini, *inheritance* bertujuan membentuk obyek baru yang memiliki sifat sama atau mirip dengan obyek yang sudah ada sebelumnya (pewarisan). Inheritance dapat diartikan sebagai pewarisan sifat-sifat suatu objek kepada objek turunannya. Obyek turunan dapat digunakan membentuk obyek turunan lagi dan seterusnya. Setiap perubahan pada obyek induk, juga akan mengubah obyek turunannya. Susunan obyek induk dengan obyek turunannya disebut dengan hirarki obyek.

Berikut merupakan contoh penggunaan *inheritance*. Pada contoh ini kita menurunkan sebuah subclass yang bernama *Cylinder* dari superclass *Circle* yang telah

kita buat sebelumnya. Perlu kita catat kita dapat menggunakan class Circle secara berulang-ulang. Sifat penggunaan yang diulang-ulang ini lah merupakan sifat yang penting pada *OOP*. Class Cylinder menurunkan semua *variables* (radius dan color) dan *methods* (*getRadius()*, *getArea()*, dan lainnya) dari *superclass Circle*.



Buatlah class *Cylinder* dan simpan sebagai **Cylinder.java**

```

1 package circle;
2
3 public class Cylinder extends Circle{
4     // private instance variables
5     private double height;
6
7     //Constructors
8     public Cylinder(){
9         super(); //invoke superclass' constructor Circle
10        this.height = 1.0;
11    }
12
13    public Cylinder(double height){
14        super(); //invoke superclass' constructor Circle
15        this.height = height;
16    }
  
```

```

17
18     public Cylinder(double height, double radius){
19         super(radius); //invoke superclass' constructor Circle(radius)
20         this.height = height;
21     }
22
23     public Cylinder(double height, double radius, String color){
24         super(radius, color); //invoke superclass' constructor Circle(radius)
25         this.height = height;
26     }
27
28     //Getter and Setter
29     public double getHeight(){
30         return this.height;
31     }
32
33     public void setHeight(double height){
34         this.height = height;
35     }
36
37     //Return the volume of this Cylinder
38     public double getVolume(){
39         return getArea()*height; // Use Circle's getArea()
40     }
41 }
```

Kemudian tambahkan beberapa baris kode berikut pada **Circle.java**

```

37     public void setRadius(double radius){
38         this.radius = radius;
39     }
40
41     public void setColor(String color){
42         this.color = color;
43     }
44
45     public String toString(){
46         return "This is a Circle";
47     }
```

Selanjutnya, untuk melakukan *testing*, tambahkan baris kode berikut pada **TestCircle.java**

```

22
23     // Declare and Construct an Instance of the Cylinder Class Called c4
24     Cylinder c4 = new Cylinder(500); // Use Cylinder
25     System.out.println("\nThe radius is: " + c4.getRadius()); // Invoke superclass Circle's methods
26     System.out.println("The color is: " + c4.getColor()); // Invoke superclass Circle's methods
27     System.out.printf("The area is: %.2f\n", c4.getArea()); // Invoke superclass Circle's methods
28     System.out.println("The height is: " + c4.getHeight());
29     System.out.printf("The volume is: %.2f\n", c4.getVolume());
30 
```

Berikut ini merupakan hasil ketika program dijalankan.

```

Output - Run (circle) ×
The radius is: 2.0
The color is: blue
The area is: 12.56
The radius is: 2.0
The color is: red
The area is: 12.56
The radius is: 1.0
The color is: red
The area is: 3.14

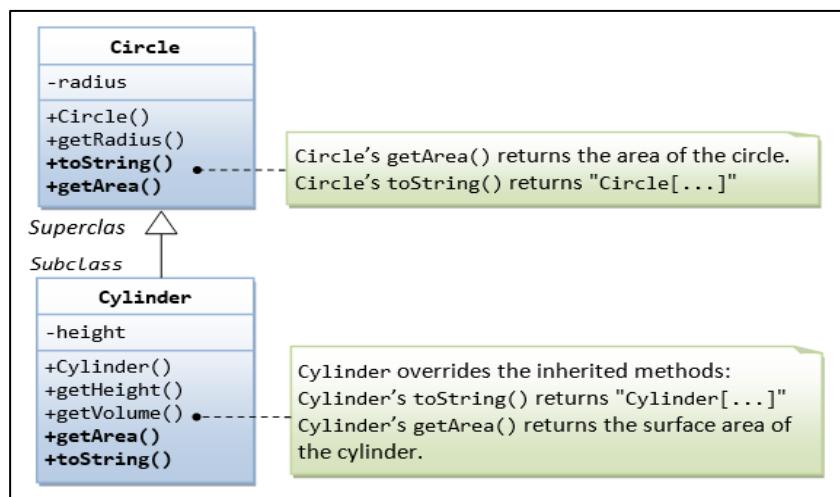
The radius is: 1.0
The color is: red
The area is: 3.14
The height is: 500.0
The volume is: 1570.00
-----
BUILD SUCCESS
-----
Total time: 1.299 s

```

1.2.12 Polymorphism

Polymorphism adalah suatu aksi yang memungkinkan *programmer* menyampaikan pesan tertentu keluar dari hierarki objeknya, dimana objek yang berbeda memberikan tanggapan/respon terhadap pesan yang sama sesuai dengan sifat masing-masing obyek. *Polymorphic* dapat berarti banyak bentuk, maksudnya yaitu kita dapat menimpa (*override*), suatu *method*, yang berasal dari *parent class (super class)* dimana objek tersebut diturunkan, sehingga memiliki *behavior* yang berbeda.

Pada contoh *Circle* dan *Cylinder* sebelumnya: *Cylinder* merupakan subclass dari *Circle*. Kita dapat mengatakan bahwa “*Cylinder is a Circle*” . Hubungan *subclass-superclass* dapat dikatakan sebagai hubungan “*is a*”. Berikut ini merupakan contoh penggunaan *polymorphism*.



Pada class *Circle* ubahlah *modifier* dari *private* menjadi *public*.

```
4 // Change from private to public
5     public double radius;
6     public String color;
7 }
```

Kemudian tambahkan beberapa baris kode berikut pada *Cylinder.java*

```
41     public double getArea(){
42         final double pi = 3.14;
43         return 2*pi*radius*radius + 2*pi*height;
44     }
45
46
47     // Define itself
48     public String toString(){
49         return "This is a Cylinder";    //to be refined later
50     }
51 }
52 }
```

Selanjutnya, tambahkan beberapa baris kode berikut pada *TestCircle.java* dan kajilah hasil yang dikeluarkan dari baris kode 33-38.

```
30 // Declare and Construct an Instance of the Cylinder Class Called c4
31 Circle c5 = new Cylinder(1.1,2.2); // Use Cylinder
32 System.out.println("\nThe radius is: " + c5.getRadius()); // Invoke superclass Circle's methods
33 System.out.println("The color is: " + c5.getColor()); // Invoke superclass Circle's methods
34 System.out.println("The height is: " + c5.getHeight()); //compilation error
35 System.out.printf("The volume is: %.2f\n", c5.getVolume()); //compilation error
36 System.out.printf("The area is: %.2f\n", c5.getArea()); // Run the overridden version!
37 System.out.println(c5.toString()); // Run the overridden version!
```

Pada contoh ini, kita membuat sebuah *instance* dari *class Cylinder* dan meng-*assign* itu ke sebuah *Circle (superclass)*, seperti pada berikut:

```
// Substitute a subclass instance to a superclass reference
Circle c5 = new Cylinder(1.1, 2.2);
```

Kita dapat menggunakan semua *class* yang terdapat pada *Circle* (yang diturunkan pada *Cylinder*), misalnya

```
// Invoke superclass Circle's methods
System.out.println("\nThe radius is: " + c5.getRadius());
System.out.println("The color is: " + c5.getColor());
```

Hal ini disebabkan *instance* pada *subclassnya* memiliki semua sifat dari *superclassnya*. Namun, kita tidak dapat menjalankan *method* yang didefinisikan di *class Cylinder* pada *c5*. Contohnya :

```
//CANNOT invoke method in Cylinder as it is a Circle reference!
//compilation error
System.out.println("The height is: " + c5.getHeight());
System.out.printf("The volume is: %.2f\n", c5.getVolume());
```

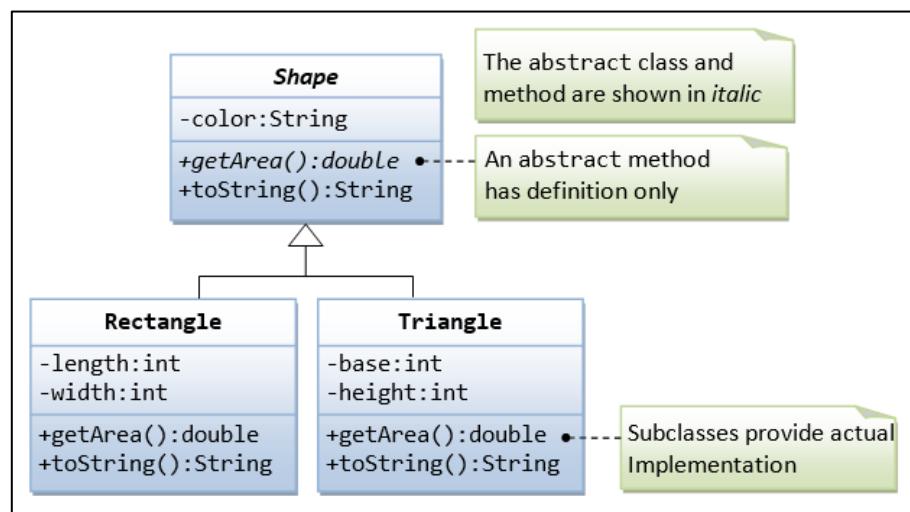
Hal ini dikarenakan `c5` merujuk pada *class Circle*, bukan *class Cylinder*, dimana hal ini menyebabkan tidak diketahui metode yang ada pada *subclass Cylinder*.

Walaupun `c5` merujuk pada *class Circle*, tetapi `c5` mengenai objek *subclass Cylinder*. Sehingga, meskipun `c5` merujuk dan mempertahankan identitas internalnya, *subclass Cylinder* meng-*overrides method getArea()* dan *toString()*. Pada contoh ini `c5.getArea()` atau `c5.toString()` meminta versi *override* yang didefinisikan pada *subclass Cylinder* sebagai ganti dari versi yang didefinisikan di *class Circle*. Hal ini dikarekan `c5` pada faktanya berkaitan dengan *class Cylinder* secara internal.

```
// Run the overridden version!
System.out.printf("The area is: %.2f%n", c5.getArea());
System.out.println(c5.toString());
```

1.2.13 Abstract

Metode *abstract* adalah metode dengan hanya *signature* saja atau representasi (misalnya, nama metode, daftar argumen dan jenis keluaran) tanpa implementasi (yaitu, metodenya). Kita dapat menggunakan *keyword abstract* untuk mendeklasaikan metode abstrak. Berikut ini merupakan contoh penggunaan *abstract*, pada diagram ini ditunjukkan bahwa terdapat suatu *class abstract Shape* dan dua *subclass* yang menunjukkan implementasi dari *class abstract* tersebut, yaitu *Rectangle* dan *Triangle*.



Berdasarkan contoh diatas, kita membuat *class Shape* terlebih dahulu dan simpan sebagai *Shape.java*, pada *class* ini kita dapat mendeklarasikan metode *abstract* *getArea()*, *draw()*, dll.

```

1 package shape;
2
3 abstract public class Shape {
4     //Private member variable
5     private String color;
6
7     //Constructor
8     public Shape (String color){
9         this.color = color;
10    }
11
12     @Override
13     public String toString(){
14         return "Shape of color=\"" + color + "\"";
15     }
16
17     //All shape subclass must implement a method called getArea()
18     abstract double getArea();
19 }
```

Implementasi dari *method* ini tidak dimungkinkan pada *class Shape*, karena ada beberapa faktor yang belum diketahui, yaitu bagaimana menghitung suatu luas bentuk jika bentuknya belum diketahui. *Method* ini dapat dijalankan ketika bentuk sudah diketahui. *Abstract method* ini tidak dapat dipergunakan karena *method* tersebut tidak memiliki implementasi nyatanya. Oleh karena itu, kita harus mengimplementasikan *subclass* yang lain seperti berikut ini.

Berikut ini merupakan *subclass Triangle* yang disimpan sebagai **Triangle.java**

```

1 package shape;
2
3 public class Triangle extends Shape{
4     //Private member variables
5     private int base;
6     private int height;
7
8     //Constructor
9     public Triangle(String color, int base, int height){
10        super(color);
11        this.base=base;
12        this.height=height;
13    }
14
15     @Override
16     public String toString(){
17         return "Triangle[base=" + base + ",height=" + height + "," + super.toString() + "]";
18     }
19
20     //Override the inherited getArea() to provide the proper implementation
21     @Override
22     public double getArea(){
23         return 0.5*base*height;
24     }
25 }
```

Berikut ini merupakan *subclass Rectangle* yang disimpan sebagai **Rectangle.java**

```

1 package shape;
2
3 public class Rectangle extends Shape{
4     //Private member variables
5     private int length;
6     private int width;
7
8     //Constructor
9     public Rectangle(String color, int length, int width){
10        super(color);
11        this.length=length;
12        this.width=width;
13    }
14
15    @Override
16    public String toString(){
17        return "Rectangle[length=" + length + ",width=" + width + "," + super.toString() + "]";
18    }
19
20    //Override the inherited getArea() to provide the proper implementation
21    @Override
22    public double getArea(){
23        return length*width;
24    }
25}

```

Berikut *class* untuk melakukan *testing* yang disimpan sebagai **TestShape.java**

```

1 package shape;
2
3 public class TestShape {
4     public static void main(String[] args){
5         Shape s1 = new Rectangle("red", 4, 5);
6         System.out.println(s1);
7         System.out.println("Area is " + s1.getArea());
8
9         Shape s2 = new Triangle("red", 4, 5);
10        System.out.println(s2);
11        System.out.println("Area is " + s2.getArea());
12    }
13}
14

```

Berikut ini hasil ketika program dijalankan

```

Output - Run (shape) x
[Run] Rectangle[length=4,width=5,Shape of color="red"]
[Run] Area is 20.0
[Run] Triangle[base=4,height=5,Shape of color="red"]
[Run] Area is 10.0
-----
[BUILD SUCCESS]
-----
Total time: 3.077 s

```

1.2.14 Interface

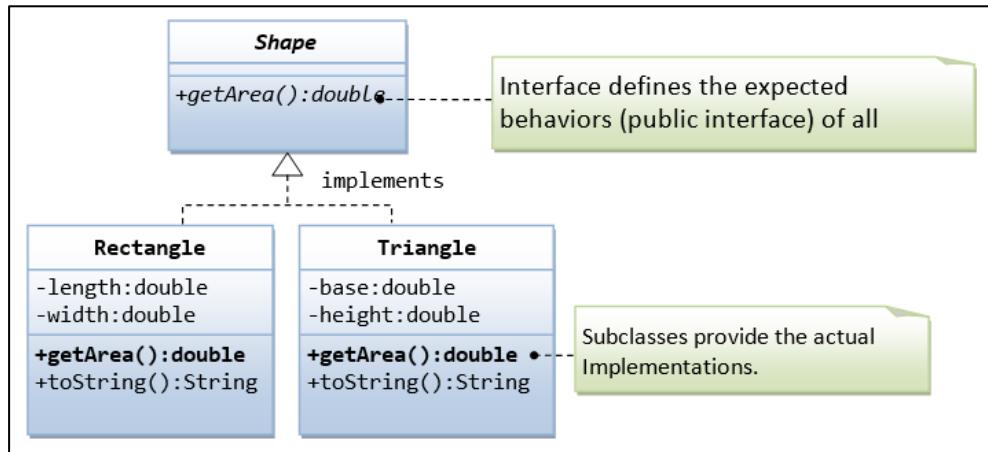
Secara sederhana, *object interface* adalah suatu kontrak atau perjanjian implementasi *method* yang berisi apa yang bisa dilakukan oleh suatu *class*, namun tidak dispesifikasikan bagaimana *class* melakukannya. Bagi *class* yang menggunakan *object interface*, *class* tersebut harus mengimplementasikan ulang seluruh *method* yang ada di dalam *interface*. Dalam pemrograman berorientasi objek, penyebutan

object interface sering disingkat menjadi *interface* saja. *Interface* berupa suatu bentuk, protokol, standar, kontrak, spesifikasi, sekumpulan aturan untuk semua *object* yang mengimplementasikannya. Mirip dengan suatu *abstract superclass*, *interface* tidak dapat langsung dipakai. Jika kita telah mempelajari *abstract class*, maka *interface* bisa dikatakan sebagai bentuk lainnya, walaupun secara konsep teoritis dan tujuan penggunaannya berbeda. Sama seperti *abstract class*, *interface* juga hanya berisi *public abstract methods* (metode dengan *signature* dan tanpa implementasi) dan mungkin konstanta atau parameternya saja (*public static final variables*) jika ada. Isi dari method akan dibuat ulang di dalam *class* yang menggunakan *interface*. Jika kita menganggap *abstract class* sebagai kerangka atau *blue print* dari class-class lain, maka *interface* adalah implementasi *method* yang harus tersedia dalam sebuah objek. *Interface* tidak bisa disebut sebagai kerangka *class*.

Misalnya, jika dianalogikan dengan komputer, *interface* bisa dicontohkan dengan *mouse*, atau *keyboard*. Di dalam *interface mouse*, kita bisa membuat method seperti *klik_kiri()*, *klik_kanan()*, dan *double_klik()*. Jika *class laptop*, menggunakan *interface*, maka *class* tersebut harus membuat ulang method *klik_kiri()*, *klik_kanan()*, *double_klik()*.

Kita harus menggunakan *keyword interface* untuk mendefinisikan *interface* (bukan *keyword class* untuk *class* normal). *Keyword public* dan *abstract* tidak diperlukan. Pada *interface* kita harus membuat *subclass* yang mengimplementasikan *interface* dan memberikan implementasi aktual dari semua metodenya. Tidak seperti *class* normal, di mana kita menggunakan *keyword extends* untuk mendapatkan *subclass*, untuk *interface*, kita menggunakan *keyword implements* untuk menurunkan subkelas. Di Java, *abstract class* dan *interface* digunakan untuk memisahkan antarmuka publik dari suatu kelas dari implementasinya sehingga memungkinkan *programmer* untuk memprogram di *interface* daripada berbagai implementasi.

Diagram ini menunjukkan contoh penggunaan *interface*.



Berdasarkan contoh di atas, kita membuat *class Shape2* terlebih dahulu dan simpan sebagai Shape2.java.

```

1  /*
2   * The interface Shape specifies the behaviors
3   * of this implementations subclasses.
4   */
5
6 package shape2;
7
8 public interface Shape2 {
9     // Use keyword "interface" instead of "class"
10    // List of public abstract methods to be implemented by its subclasses
11    double getArea();
12 }
13
  
```

Kemudian buatlah *subclass Triangle* yang disimpan sebagai Triangle.java

```

1 package shape2;
2
3 // The subclass Triangle need to implement all the abstract methods in Shape
4 public class Triangle implements Shape2 {
5     // Private member variables
6     private int base;
7     private int height;
8
9     // Constructor
10    public Triangle(int base, int height) {
11        this.base = base;
12        this.height = height;
13    }
14
15    @Override
16    public String toString() {
17        return "Triangle[base=" + base + ",height=" + height + "]";
18    }
19
20    // Need to implement all the abstract methods defined in the interface
21    @Override
22    public double getArea() {
23        return 0.5 * base * height;
24    }
25 }
  
```

Selanjutnya, buatlah *subclass Rectangle* yang disimpan sebagai Rectangle.java

```

1 package shape2;
2
3 // The subclass Rectangle needs to implement all the abstract methods in Shape
4 public class Rectangle implements Shape2 {
5     // using keyword "implements" instead of "extends"
6     // Private member variables
7     private int length;
8     private int width;
9     // Constructor
10    public Rectangle(int length, int width) {
11        this.length = length;
12        this.width = width;
13    }
14    @Override
15    public String toString() {
16        return "Rectangle[length=" + length + ",width=" + width + "]";
17    }
18    // Need to implement all the abstract methods defined in the interface
19    @Override
20    public double getArea() {
21        return length * width;
22    }
23}

```

Berikut *class* untuk melakukan *testing* yang disimpan sebagai **TestShape.java**

```

1 package shape2;
2
3 public class TestShape {
4     public static void main(String[] args) {
5         Shape2 s1 = new Rectangle(1, 2); // upcast
6         System.out.println(s1);
7         System.out.println("Area is " + s1.getArea());
8         Shape2 s2 = new Triangle(3, 4); // upcast
9         System.out.println(s2);
10        System.out.println("Area is " + s2.getArea());
11        // Cannot create instance of an interface
12        // Shape2 s3 = new Shape2("green"); // Compilation Error!!
13    }
14}

```

Berikut hasil ketika program dijalankan

```

Output - Run (shape2) ×
Rectangle[length=1,width=2]
Area is 2.0
Triangle[base=3,height=4]
Area is 6.0
BUILD SUCCESS
Total time: 3.000 s

```

1.3 Tugas

1. Buatlah sebuah program yang mendefinisikan sebuah hewan. Hewan tersebut terdiri dari 3 jenis, *karnivora*, *herbivora*, dan *omnivore*. Kemudian jalankan *method eat*, ketika *method eat* pada *karnivora* dijalankan maka akan muncul hasil “Makan daging”, ketika *method eat* pada *herbivora* dijalankan akan muncul “Makan Tumbuhan”, ketika *method eat* pada *omnivore* dijalankan akan muncul “Makan daging dan tumbuhan”. Kemudian, jelaskan mana yang *abstract*, *polymorphisms*, *inheritance* maupun *encapsulation*.

2. Buatlah sebuah miniatur program reservasi hotel. Misalkan terdapat suatu hotel yang terdiri beberapa kamar (minimal 10), lalu ketika kita jalankan suatu *method* (bebas nama *methodnya*) ke dalam suatu kamar, kamar yang status awalnya kosong berubah menjadi terisi dan juga muncul nama pengisinya

Contoh :

```
Sebelum dilakukan hoho.getRoom001("Joni")
Kamar001 : Kosong
Setelah dilakukan hoho.getRoom001("Joni")
Kamar001 : Terisi Joni
```

```
Sebelum dilakukan hoho.getRoom009("Parker")
Kamar009 : Kosong
Setelah dilakukan hoho.getRoom009("Parker")
Kamar009 : Terisi Parker
```

BAB II

ARRAY DAN LINKED LIST BESERTA APLIKASINYA

2.1 Tujuan Praktikum

1. Praktikan dapat memahami *array* dan *linked list*
2. Praktikan mampu mengimplementasikan *array* dan *linked list* dan mengetahui kapan harus menggunakannya
3. Praktikan dapat memahami tentang *stack* dan *queue*
4. Praktikan mampu mengimplementasikan *stack* dan *queue* dan mengetahui kapan harus menggunakannya

2.2 Materi

2.2.1 Array

Array adalah salah satu struktur data dasar yang tersedia, dan merupakan kumpulan dari banyak variabel dengan tipe data yang sama yang ditunjukkan oleh indeks untuk setiap elemen. Kita bisa membayangkan array seperti tabel, di mana setiap elemen disimpan dalam setiap ruang tabel. Terdapat 2 jenis array, yaitu *array* satu-dimensi yang disebut vektor dan *array* dua dimensi yang disebut matriks.

Pada Java, *array* merupakan suatu tipe primitif yang diperlakukan sebagai *object*. Cara membuat suatu array dilakukan dengan menggunakan operator baru.

```
int[] intArray; //mendefinisikan suatu array  
//membuat suatu array dan mengeset array tersebut ke intArray  
intArray = new int[100];
```

Deklarasi diatas dapat disingkat dengan satu statement:

```
int[] intArray = new int[100];
```

Deklarasi ini digunakan untuk membuat *array* dengan elemen kosong (*null*). Untuk memasukkan elemen saat awal deklarasi *array* dapat dilakukan dengan cara berikut:

```
int[] intArray = {0,3,6,9,12,15,18,21,24}; //deklarasi isi array
```

Untuk mengakses elemen dalam *array* membutuhkan tanda kurung siku ([]), mirip dengan bahasa lain. Perhatikan bahwa indeks dalam *array* dimulai dengan 0, artinya elemen pertama ada di indeks 0.

```
temp = intArray[3]; //mendapatkan isi elemen keempat pada array  
intArray[7] = 66; //memasukkan 66 ke dalam sel ke-delapan
```

Ketika suatu *array* telah disimpan, kita dapat menggunakan *array* untuk memproses data. Sebagai contoh dalam kasus pencarian (*search*) dan penggantian (*replacement*). Program ini akan menunjukkan bagaimana suatu *array* disimpan, kemudian ditampilkan nilainya. Misalkan program ini meminta input berupa ukuran *array* dan elemen/nilai yang akan disimpan, kemudian menampilkan seluruh nilai dalam satu baris sebagai output. Kemudian, kode dilanjutkan dengan pencarian data untuk menemukan apakah suatu elemen disimpan dalam *array* atau tidak. Selanjutnya, apabila data ditemukan, maka akan dilakukan penggantian data dengan suatu nilai tertentu, sehingga suatu nilai dalam *array* tersebut akan terganti. Implementasi *array* ini dapat dilihat pada kode SearchReplacement.java berikut.



The screenshot shows a Java code editor window titled "SearchReplacement.java". The code is a demonstration of how to create an array, store elements, and display them. It includes imports for Scanner and IOException, defines a class SearchReplacement with a main method, and uses loops to input array size, elements, and print the array back out.

```
1 //SearchReplacement.java  
2 //Demonstrates how to create an array and stores and display array elements  
3  
4 package searchreplacement;  
5  
6 import java.util.Scanner;  
7 import java.io.IOException;  
8  
9 public class SearchReplacement {  
10    public static void main(String[] args) throws IOException{  
11        int size; //size of arrray that we want  
12        Scanner input = new Scanner(System.in); //set up the input function;  
13        System.out.print("Enter the size of the array: ");  
14        size = input.nextInt(); //insert the desired array  
15        System.out.println("");  
16  
17        int[] arr; //reference  
18        arr = new int[size]; //make array  
19  
20        for(int j=0;j<size;j++){ //for each array space  
21            System.out.print("Enter number " + j +":\t");  
22            arr[j] = input.nextInt(); //insert a number  
23        }  
24        for(int j=0;j<size;j++){ //for each element in array  
25            System.out.print(arr[j] + " "); //print the elements  
26        }  
27        System.out.println(size);  
28        System.out.println("");  
29    }
```

```

30 //SEARCHING
31     int numSearch;           //search a number
32     System.out.print("What number do you want to search? ");
33     numSearch = input.nextInt();
34
35     boolean found = false;    //assume number is not present
36     for(int j=0; j<size; j++){
37         if(numSearch == arr[j]){
38             found = true;      //if the number is found
39             break;            //set found to true
40         }
41     }
42
43     if(found)                //if number is found
44         System.out.println(numSearch + " is available");
45     else                      //if number is not found
46         System.out.println(numSearch + " not found");
47
48 //REPLACEMENT
49     int numToReplace;        //number to be replaced
50     int numReplacing;       //new number
51     System.out.print("What number do you want to replace? ");
52     numToReplace = input.nextInt();
53     System.out.print("What will be the new number? ");
54     numReplacing = input.nextInt();
55     for(int j=0; j<size; j++){ //check every element
56         if(numToReplace == arr[j]){ //if the number is found
57             arr[j] = numReplacing; //replace the number
58         }
59     }
60
61     for(int j=0;j<size;j++){   //display the resulting change
62         System.out.print(arr[j] + " "); //print the element
63     }
64 } //end main()
65 } //end class Array

```

Berikut ini hasil program ketika dijalankan

```

Output - Array (run) X
run:
Enter the size of the array: 5
Enter number: 65
Enter number: 76
Enter number: 1
Enter number: 900
Enter number: 55
65 76 1 900 55
BUILD SUCCESSFUL (total time: 17 seconds)

```

2.2.2 Linked List

Mirip dengan *array*, *linked list* adalah struktur data dasar lain untuk penyimpanan dan pengolahan data. Namun, perbedaannya terletak pada penggunaan memori: jika *array* menyimpan elemen dalam blok memori yang besar, *linked list* menyimpan setiap elemen pada ruang memori yang terpisah yang disebut *node*. Ruang ini

kemudian dihubungkan satu sama lain menggunakan *pointer*, sehingga disebut *linked list*.

Suatu *linked list* memerlukan dua jenis variabel:

- Node

Bagian utama dari *list*, setiap *node* berisi elemen yang disimpan dan *pointer* ke *node* berikutnya dalam *list*.

- Pointer

Kepala dari *list*, *pointer* berisi alamat memori *node* berikutnya.

Pada Java, implementasi *linked list* diawali dengan pembuatan *node* dan *pointer*. Kemudian, dilanjutkan dengan implementasi *linked list class* yang berisi inisialisasi *linked list* dan operasi yang dapat dilakukan oleh *link list*. Operasi-operasi tersebut berupa *method* yang membantu kita untuk memproses dan memanipulasi *linked list*. *Method* pada *linked list* sebagai berikut

- *insertFirst()* dan *insertLast()*

insertFirst() *method* pada *LinkListInit class* berfungsi untuk menambahkan sebuah *node* baru di awal *list*. Sedangkan *insertLast()* *method* pada *LinkListInit class* berfungsi untuk menambahkan sebuah *node* baru di akhir *list*.

- *deleteFirst()* dan *deleteLast()*

deleteFirst () method pada *LinkListInit class* berfungsi untuk menghapus *node* pertama dari *list*. Sedangkan *deleteLast()* *method* pada *LinkListInit class* berfungsi untuk menghapus *node* terakhir dari *list*. Hal ini dilakukan dengan memindahkan *pointer* dari *node* pertama ke *node* selanjutnya untuk *deleteFirst()* dan memindahkan *pointer* dari *node* terakhir ke *node* terakhir kedua untuk *deleteLast*.

- *displayList()*

Method pada *LinkListInit* yang berfungsi untuk menampilkan isi *list*, dari *node* yang pertama hingga *node* terakhir.

Berikut ini merupakan implementasi dari *LinkList.java*. Pada program ini, dua *list* dideklarasikan, *list* pertama berisi bilangan bulat yang dimasukkan dari akhir, sedangkan *list* kedua berisi bilangan bulat yang dimasukkan di awal. *Node* pertama pada *list* pertama dan *node terakhir* pada *list* kedua kemudian dihapus.

The screenshot shows an IDE interface with a Java file named `LinkList.java` open. The code defines a `Node` class and a `LinkListInit` class. The `Node` class has fields `iData` and `next`, and methods `displayLink`. The `LinkListInit` class has methods for initializing the list, checking if it's empty, inserting nodes at the start or end, and deleting the first node.

```
//LinkList.java
//demonstrates linkedlist
package linklist;

import java.util.Scanner;
import java.io.IOException;

class Node{
    public int iData; //data item
    public Node next; //next link in list

    public Node(int id){ //constructor
        iData = id; //initialize data
    }

    public void displayLink(){ //display ourself
        System.out.print("{ " + iData + " } ");
    }
} //end class Node

class LinkListInit{
    private Node first; //ref to first node on list

    public LinkListInit(){ //constructor
        first = null; //no items on list yet
    }

    public boolean isEmpty(){ //true if list is empty
        return (first==null);
    }

    public void insertFirst(int id){ //insert at start of list
        Node newNode = new Node(id); //make new node
        newNode.next = first; //newNode --> old first
        first = newNode; //first --> newNode
    }

    public void insertLast(int id){ //insert at end of list
        if(first==null) //if the list is empty
            insertFirst(id); //create first node
        else{
            Node temp = first; //start from first node
            while(temp.next!=null){ //until we are at the last node
                temp = temp.next; //keep going
            }
            temp.next = new Node(id); //add new node at the end
        }
    }

    public Node deleteFirst(){ //delete first item (assume list not empty)
        Node temp = first; //save reference to node
        first = first.next; //delete it: first --> next node
        return temp; //return deleted node
    }
}
```

```

55
56     public Node deleteLast(){           //delete last node
57         Node temp = first;           //start from first node
58         while(temp.next.next != null){ //till we are at the second last node
59             temp = temp.next;        //keep going
60         }
61         Node toReturn = temp.next;   //store the last node
62         temp.next = null;          //delete reference to last node
63         return toReturn;          //return deleted node
64     }
65
66     public void displayList(){
67         System.out.print("List (first-->last): ");
68         Node current = first;       //start at the beginning of the list
69         while(current!=null){      //until end of list
70             current.displayLink();  //print data
71             current = current.next; //move to next link
72         }
73         System.out.println("");
74     }
75 } //end class LinkListInit
76

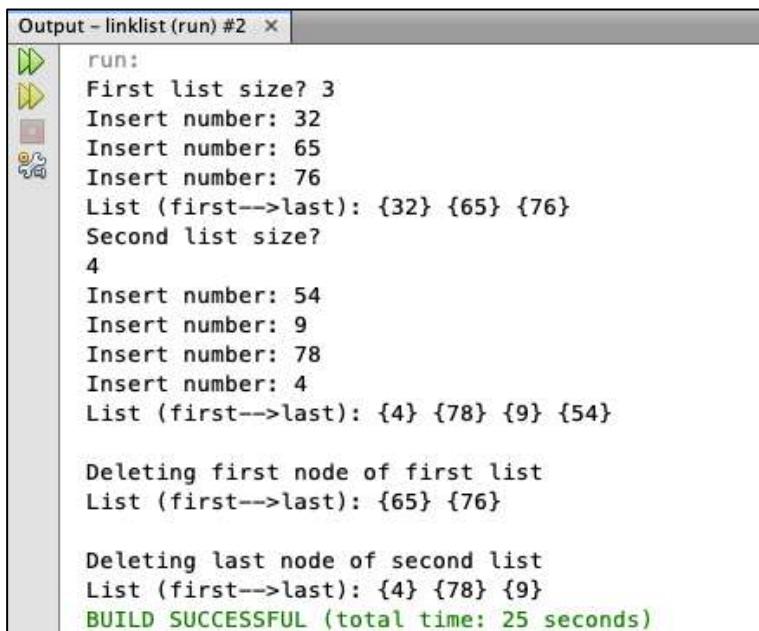
```

```

77 class LinkList {
78     public static void main(String[] args) throws IOException{
79         LinkListInit theList1 = new LinkListInit();           //make new list
80         LinkListInit theList2 = new LinkListInit();           //make new list
81
82         Scanner in = new Scanner(System.in);                  //set scanner
83         int nodeNum1;           //the number of integers for first list
84         int nodeNum2;           //the number of integers for second list
85         int tempNum;            //temporary holder for integer
86
87         System.out.print("First list size? ");
88         nodeNum1 = in.nextInt(); //insert first list size
89
90         for(int i=0; i<nodeNum1; i++){
91             System.out.print("Insert number: ");
92             tempNum = in.nextInt();           //insert number
93             theList1.insertLast(tempNum);    //on the end of list
94         }
95         theList1.displayList();           //display list
96
97         System.out.print("Second list size? ");
98         nodeNum2 = in.nextInt(); //insert second list size
99
100        for(int i=0; i<nodeNum2; i++){
101            System.out.print("Insert number: ");
102            tempNum = in.nextInt();           //insert number
103            theList2.insertFirst(tempNum);  //on the start of list
104        }
105        theList2.displayList();           //display list
106
107        System.out.println("\nDeleting first node of first list");
108        theList1.deleteFirst();           //deleting the first node of first list
109        theList1.displayList();          //first list after deletion
110
111        System.out.println("\nDeleting last node of second list");
112        theList2.deleteLast();           //deleting the last node of second list
113        theList2.displayList();          //second list after deletion
114    } //end main()
115 } //end class LinkList

```

Berikut ini contoh hasil program ketika dijalankan.



The screenshot shows the output window of a Java IDE. The title bar says "Output - linklist (run) #2". The output text is as follows:

```
run:
First list size? 3
Insert number: 32
Insert number: 65
Insert number: 76
List (first-->last): {32} {65} {76}
Second list size?
4
Insert number: 54
Insert number: 9
Insert number: 78
Insert number: 4
List (first-->last): {4} {78} {9} {54}

Deleting first node of first list
List (first-->last): {65} {76}

Deleting last node of second list
List (first-->last): {4} {78} {9}
BUILD SUCCESSFUL (total time: 25 seconds)
```

Implementasi lain dari *linked list* pada Java adalah penggunaan *built-in LinkedList class*. Class ini berisi banyak operasi, termasuk yang telah dilakukan sebelumnya, tanpa harus membuat class baru. Berikut ini implementasi *linked list* menggunakan *built-in*.

The screenshot shows a Java code editor window titled "LinkListClass.java". The code is a demonstration of using the built-in `LinkedList` class. It includes imports for `IOException`, `Scanner`, and `LinkedList`. The `main` method creates two `LinkedList` objects, reads sizes and integers from the user, inserts them into the lists, displays the lists, and then deletes the first and last nodes of both lists before displaying them again.

```
1 //LinkListClass.java
2 //demonstrates the use of the built-in LinkedListClass
3 package linklistclass;
4 import java.io.IOException;
5 import java.util.Scanner;
6 import java.util.LinkedList; //important for LinkedListClass
7
8 public class LinkListClass {
9     public static void main(String[] args) throws IOException{
10         LinkedList theList1 = new LinkedList(); //make new list
11         LinkedList theList2 = new LinkedList(); //make new list
12
13         Scanner in = new Scanner(System.in); //set scanner
14         int nodeNum1; //the number of integers for first list
15         int nodeNum2; //the number of integers for second list
16         int tempNum; //temporary holder for integer
17
18         System.out.print("First list size? ");
19         nodeNum1 = in.nextInt(); //insert first list size
20
21         for(int i=0; i<nodeNum1; i++){
22             System.out.print("Insert number: ");
23             tempNum = in.nextInt(); //insert number
24             theList1.addLast(tempNum); //on the end of list
25         }
26         System.out.println(theList1); //display list
27
28         System.out.print("Second list size? ");
29         nodeNum2 = in.nextInt(); //insert second list size
30
31         for(int i=0; i<nodeNum2; i++){
32             System.out.print("Insert number: ");
33             tempNum = in.nextInt(); //insert number
34             theList2.addFirst(tempNum); //on the start of list
35         }
36         System.out.println(theList2); //display list
37
38         System.out.println("\nDeleting first node of first list");
39         theList1.removeFirst(); //deleting the first node of first list
40         System.out.println(theList1); //first list after deletion
41
42         System.out.println("\nDeleting last node of second list");
43         theList2.removeLast(); //deleting the last node of second list
44         System.out.println(theList2); //second list after deletion
45     }
46 }
```

Berikut ini hasil program ketika dijalankan.

```
Output - linklistclass (run) ×
run:
First list size? 3
Insert number: 51
Insert number: 76
Insert number: 3
[51, 76, 3]
Second list size? 5
Insert number: 12
Insert number: 54
Insert number: 85
Insert number: 3
Insert number: 8
[8, 3, 85, 54, 12]

Deleting first node of first list
[76, 3]

Deleting last node of second list
[8, 3, 85, 54]
BUILD SUCCESSFUL (total time: 27 seconds)
```

2.2.3 Stack

Suatu *stack* merupakan suatu struktur data linear yang memungkinkan akses ke hanya satu item data: item terakhir dimasukkan. Apabila menghapus item, maka item yang diakses adalah item yang dimasukkan terakhir. Contoh *stack* seperti tumpukan baju: baju ditaruh dalam lemari pada tumpukan paling atas dan baju diambil dari tumpukan yang paling atas. Suatu *stack* menggunakan mekanisme penyimpanan *Last-In-First-Out (LIFO)* karena item terakhir yang dimasukkan adalah item pertama yang dihapus. Stack dapat diimplementasikan menggunakan *array* dan *linked list*.

Stack memiliki beberapa operasi (*method*) dasar sebagai berikut.

- *Initialize stack*

Inisialisasi *stack* dengan membuat *stack* kosong atau *stack* yang tidak memiliki elemen di dalamnya

- *Push*

Push merupakan operasi penambahan elemen ke bagian atas *stack*. Algoritmanya adalah sebagai berikut:

- a. Buat data baru yang akan dimasukkan (*push*) ke dalam *stack*
- b. Masukkan data ke dalam *stack*

- c. Ubah *pointer* teratas untuk menunjuk ke data yang baru saja dimasukkan
- *Pop*

Pop adalah operasi menghilangkan elemen dari *stack*. Dengan asumsi *stack* berisi beberapa elemen, algoritma-nya adalah sebagai berikut:

 - a. Ambil dan hapus data teratas dari *stack*
 - b. Ubah pointer teratas untuk menunjuk ke data teratas berikutnya
- *isEmpty*

isEmpty adalah suatu operasi untuk memeriksa apakah *stack* berisi elemen atau tidak. Operasi mengembalikan *true* apabila *stack* kosong dan mengembalikan *false* apabila sebaliknya
- *printStack*

printStack adalah operasi untuk mencetak *stack*

Berikut ini adalah contoh implementasi *stack* menggunakan *array* pada Stack.java. Program menyimpan input (dalam angka) pada suatu *stack*, kemudian menampilkan angka yang telah disimpan.

```

Stack.java X
Source History I O M F S D E L C P R T Z
1 // Stack.java
2 // demonstrates stack
3 package stack;
4 import java.io.IOException; //exception for I/O
5 import java.util.Scanner; //for input
6 import java.util.Arrays;
7
8 class StackInit{ //contains stack methods
9     private final int maxSize; //size of stack array
10    private int[] stackArray; //initialize array
11    private int top; //top of stack
12
13    public StackInit(int s){ //constructor
14        maxSize = s; //set array size
15        stackArray = new int[maxSize]; //create array
16        top = -1; //no items yet
17    }
18
19    public void push(int j){ //put item on top of stack
20        stackArray[++top] = j; //increment top, insert item
21    }
22
23    public double pop(){ //take item from top of stack
24        return stackArray[top--]; //access item, decrement top
25    }
26
27    public boolean isEmpty(){ //true if stack is empty
28        return (top == -1);
29    }
30
31    public void printStack(){
32        System.out.println(Arrays.toString(stackArray));
33    }
34 } // end class StackInit
35

36 public class Stack {
37     public static void main(String[] args) throws IOException {
38         int stackSize; //stack size
39         int stackNum; //number to be inserted in stack
40         Scanner in = new Scanner(System.in);
41
42         System.out.print("How many integer? ");
43         stackSize = in.nextInt(); //insert stack size
44
45         StackInit theStack = new StackInit(stackSize); //make new stack
46
47         for(int i=0; i<stackSize; i++){
48             System.out.print("Enter number: ");
49             stackNum = in.nextInt(); //insert number
50             theStack.push(stackNum); //push element onto stack
51         }
52         theStack.printStack(); //print Stack
53
54         while(!theStack.isEmpty()){ //until it is empty, delete item from stack
55             double value = theStack.pop();
56             System.out.print(value); //display the popped item
57             System.out.print(" ");
58         }
59
60         System.out.println("");
61     } //end main()
62 } //end class Stack

```

Berikut hasil program ketika dijalankan

```
Output - stack (run) ×
run:
How many integer? 4
Enter number: 1
Enter number: 2
Enter number: 6
Enter number: 7
[1, 2, 6, 7]
7.0 6.0 2.0 1.0
BUILD SUCCESSFUL (total time: 12 seconds)
```

Java memiliki *built-in Stack class* yang di-import melalui `java.util.Stack`. Berikut adalah contoh program yang mirip dengan sebelumnya tetapi menggunakan *Stack class*, pada `StackBasic.java`

```
StackBasic.java ×
Source History ...
1 // StackBasic.java
2 // demonstrates the built-in stack
3 package stackbasic;
4 import java.io.IOException;           //exception for I/O
5 import java.util.Scanner;           //for input
6 import java.util.Stack;             //for Stack class
7
8 public class StackBasic {
9     public static void main(String[] args) throws IOException {
10         int stackSize;                  //stack size
11         int stackNum;                 //number to be inserted in stack
12
13         Scanner in = new Scanner(System.in);
14
15         System.out.print("How many integer? ");
16         stackSize = in.nextInt();      //insert stack size
17
18         Stack theStack = new Stack();   //make new stack
19
20         for(int i=0; i<stackSize; i++){
21             System.out.print("Enter number: ");
22             stackNum = in.nextInt();    //insert number
23             theStack.push(stackNum);   //push element onto stack
24         }
25
26         System.out.print(theStack);
27         System.out.println("");
28
29         while(!theStack.isEmpty()){    //until it is empty, delete item from stack
30             Integer value = (Integer) theStack.pop();
31             System.out.print(value);    //display the popped item
32             System.out.print(" ");
33         }
34
35     }
36 }
37 }
```

Berikut ini hasil program ketika dijalankan.

```
Output - StackBasic (run)
run:
How many integer? 5
Enter number: 12
Enter number: 25
Enter number: 76
Enter number: 39
Enter number: 9
[12, 25, 76, 39, 9]
9 39 76 25 12
BUILD SUCCESSFUL (total time: 30 seconds)
```

2.2.4 Queue

Suatu *queue* adalah struktur data linier yang mirip dengan *stack*, tetapi penambahan dan penghapusan item data dilakukan pada ujung yang berlawanan. Item dapat dimasukkan dari suatu ujung dan dihapus dari ujung yang lainnya. Contoh *queue* seperti antrian supermarket: pembeli masuk dari satu ujung antrian dan keluar dari ujung lainnya menuju kasir. Suatu *queue* dikatakan sebagai mekanisme penyimpanan *First-In-First-Out (FIFO)*, karena item pertama yang dimasukkan adalah yang pertama kali dihapus. *Queue* dapat diimplementasikan dalam dua cara: *array* dan *linked-list*.

Queue memiliki beberapa operasi (*method*) dasar sebagai berikut.

- *Initialize queue*

Inisialisasi *queue* dengan membuat *queue* kosong atau *queue* yang tidak memiliki elemen di dalamnya.

- *Enqueue*

Enqueue adalah operasi penambahan elemen baru pada ujung belakang *queue*. Algoritma-nya adalah sebagai berikut:

- a. Buat data baru yang akan dimasukkan (*push*) ke dalam *queue*
- b. Masukkan data baru ke dalam *queue*
- c. Ubah *pointer* terakhir untuk menunjuk ke data yang baru dimasukkan

- *Dequeue*

Dequeue adalah operasi menghilangkan suatu elemen dari ujung depan *queue*. Dengan asumsi *queue* berisi beberapa elemen, algoritma-nya adalah sebagai berikut:

- a. Ambil dan hapus data yang paling depan dari *queue*
- b. Ubah *pointer* pertama untuk menunjuk ke data depan berikutnya
- *isEmpty*
isEmpty adalah suatu operasi untuk memeriksa apakah *queue* berisi elemen atau tidak. Operasi mengembalikan *true* apabila *queue* kosong dan mengembalikan *false* apabila sebaliknya
- *isFull*
Kebalikan dari *isEmpty*, operasi mengembalikan nilai *true* apabila *queue* penuh dan mengembalikan *false* apabila sebaliknya
- *printQueue*
printQueue adalah operasi untuk mencetak *queue*.
Berikut ini adalah contoh implementasi *queue* menggunakan *array* pada *Queue.java*. Program akan mengimplementasikan suatu *queue* menggunakan *QueueInit class* dan beroperasi berdasarkan perintah yang sesuai.

```
1 // Queue.java
2 // demonstrates queue
3 package queue;
4 import java.io.IOException;           //exception for I/O
5 import java.util.Scanner;            //for input
6 import java.util.Arrays;
7
8 class QueueInit{ //contains queue methods
9     private int maxSize;
10    private int[] queueArray;
11    private int front;
12    private int rear;
13    private int nItems;
14
15    public QueueInit(int s){           //constructor
16        maxSize = s;
17        queueArray = new int[maxSize];
18        front = 0;
19        rear = -1;
20        nItems = 0;
21    }
22
23    public void enqueue(int j){        //put item at rear of queue
24        if (rear == maxSize-1)         //deal with wraparound
25            rear = -1;
26        queueArray[++rear] = j;       //increment rear and insert
27        nItems++;                   //one more item
28    }
29
30    public int dequeue(){             //take item from front of queue
31        int temp = queueArray[front++]; //get value and increment front
32        if(front == maxSize)          //deal with wraparound
33            front = 0;
34        nItems--;                   //one less item
35        return temp;
36    }
37
38    public boolean isEmpty(){         //true if queue is empty
39        return (nItems==0);
40    }
41
42    public boolean isFull(){          //true if queue is full
43        return (nItems==maxSize);
44    }
45
46    public void printQueue(){
47        System.out.println(Arrays.toString(queueArray));
48    }
49}
```

```

51  public class Queue {
52      public static void main(String[] args) throws IOException{
53          int queueSize;                                //for queue size
54          int numTemp;                                 //for inserted number
55          int numChoice = 0;                           //for command
56
57          Scanner in = new Scanner(System.in);           //for input
58          System.out.print("Enter queue size: ");
59          queueSize = in.nextInt();
60
61          QueueInit theQueue = new QueueInit(queueSize); //set queue
62
63          while(numChoice != 3){
64              System.out.println("\n 1: Enqueue \t 2 : Dequeue \t 3 : End");
65              System.out.print("Enter command: ");
66              numChoice = in.nextInt();
67              if(numChoice == 1){
68                  if(theQueue.isFull())
69                      System.out.println("Queue is full");
70                  else{
71                      System.out.print("Enter number: ");
72                      numTemp = in.nextInt();
73                      theQueue.enqueue(numTemp);
74                  }
75              }
76              else if(numChoice == 2){
77                  if(theQueue.isEmpty())
78                      System.out.println("Queue is empty");
79                  else{
80                      numTemp = theQueue.dequeue();
81                      System.out.println("Dequeue number: " + numTemp);
82                  }
83              }
84              else if(numChoice != 3){
85                  System.out.println("Wrong command");
86              }
87          } //end main()
88      } //end class Queue
89  }

```

Berikut hasil program ketika dijalankan



```
Output - Queue (run) X
run:
Enter queue size: 5

1: Enqueue    2 : Dequeue    3 : End
Enter command: 1
Enter number: 23

1: Enqueue    2 : Dequeue    3 : End
Enter command: 1
Enter number: 45

1: Enqueue    2 : Dequeue    3 : End
Enter command: 2
Dequeue number: 23

1: Enqueue    2 : Dequeue    3 : End
Enter command: 2
Dequeue number: 45

1: Enqueue    2 : Dequeue    3 : End
Enter command: 3
BUILD SUCCESSFUL (total time: 11 seconds)
```

Queue biasanya digunakan dalam situasi dimana elemen dapat masuk satu per satu dan diproses satu per satu sehingga membutuhkan penggunaan *queue*. Salah satu contohnya adalah antrian perbankan (*banking queue*), dimana orang menunggu dalam satu baris untuk mendapatkan layanan perbankan. Berikut adalah contoh penerapan tersebut pada BankService.java. Program ini memungkinkan untuk menambah pelanggan pada antrian, menghapus pelanggan dari antrian dan mengecek situasi antrian. Perhatikan bahwa program ini menggunakan *LinkedList class* untuk membuat queue.


```

70 public class BankService {
71     public static void main(String[] args) throws IOException {
72         String name_m;           //for getting customer name
73         double idCust_m = 0;      //for getting customer id
74         double balance_m;        //for getting customer balance
75         int queueCom = 0;         //for queue command
76
77         Scanner in = new Scanner(System.in);    //for input
78         BankServiceStart bankQueue = new BankServiceStart(); //create queue
79
80         //creating a bank system
81         System.out.println("Welcome to Rajasa Bank Queueing System! ");
82
83         while(queueCom != 4){ //while we are not done
84             System.out.println("What would you like to do?"); //ask command
85             System.out.println("1: Check queue\t" +
86                                 "2: Add customer to queue\t" +
87                                 "3: Call customer from queue\t" +
88                                 "4: Finish");
89             queueCom = in.nextInt();
90             switch(queueCom){
91                 case 1:
92                     bankQueue.displayQueue();
93                     break;
94                 case 2:
95                     System.out.print("Customer name: ");
96                     name_m = in.next();
97                     idCust_m++;
98                     System.out.print("Balance: ");
99                     balance_m = in.nextDouble();
100                    //create customer object
101                    Customer newCust = new Customer(name_m, idCust_m, balance_m);
102                    bankQueue.enqueue(newCust);
103                    System.out.println(name_m + " has been added to queue!");
104                    break;
105                 case 3:
106                     if(bankQueue.isEmpty())
107                         System.out.println("Queue is empty");
108                     else{
109                         Customer toServe = bankQueue.callCust();
110                         System.out.println("Calling " + toServe.getName());
111                         bankQueue.dequeue();
112                     }
113                     break;
114                 case 4:
115                     System.out.println("Thank you for using the system!");
116                     break;
117                 default:
118                     System.out.println("Incorrect command. Try again.");
119             }
120         }
121     } //end main
122 } //end class BankService

```

Berikut ini contoh hasil *output* ketika program dijalankan.

```
Output - BankService (run) X
Welcome to Rajasa Bank Queueing System!
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
1
Queue is currently empty
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
2
Customer name: Jenny
Balance: 20000
Jenny has been added to queue!
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
2
Customer name: Henry
Balance: 50000000
Henry has been added to queue!
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
1
Name: Jenny
Customer ID: 1.0
Balance: 20000.0
Name: Henry
Customer ID: 2.0
Balance: 5.0E7
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
3
Calling Jenny
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
1
Name: Henry
Customer ID: 2.0
Balance: 5.0E7
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
3
Calling Henry
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
1
Queue is currently empty
What would you like to do?
1: Check queue 2: Add customer to queue      3: Call customer from queue 4: Finish
4
Thank you for using the system!
```

2.3 Tugas

1. Menggunakan *array*, buatlah program yang dapat menjumlahkan semua angka yang dimasukkan oleh *user*. Perhatikan bahwa program akan membaca jumlah angka yang akan dimasukkan dan angka yang dimasukkan. Lakukan hal yang sama seperti pertanyaan di atas, tetapi gunakan *linked list*.
2. Buatlah *array* yang menyimpan 10 angka, kemudian lipat tigakan setiap angka.
3. Buatlah program untuk membalik urutan dalam deretan karakter yang dimasukkan oleh user (misalnya REVELATION -> NOITALEVER)

4. Buatlah program yang memeriksa apakah suatu *string* adalah palindrome (palindrome adalah string yang dibaca dari depan atau belakang, misalnya TACOCAT)
5. Buatlah program yang menghitung biaya parkir yang harus dibayar setiap mobil di tempat parkir. Setiap mobil memiliki informasi model mobil dan waktu parkirnya. Asumsikan tarifnya adalah Rp. 2000 per jam.

BAB III

TREE & BINARY TREE

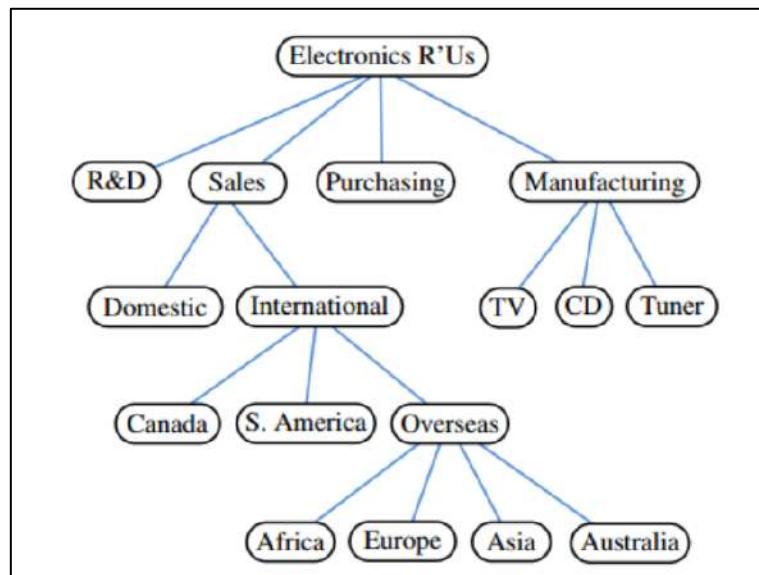
3.1 Tujuan Praktikum

1. Praktikan dapat memahami dan mampu mengimplementasikan *binary tree*
2. Praktikan dapat memahami dan mampu mengimplementasikan *binary search tree*

3.2 Materi

3.2.1 Tree

Tree adalah sebuah *abstract data type (ADT)* yang menyimpan data secara hirarki. Semua elemen pada *Tree* memiliki *parent* dan *child* sejumlah nol atau lebih. Sebuah *Tree* biasanya direpresentasikan menggunakan oval atau persegi panjang yang saling dihubungkan dengan garis (seperti gambar di bawah). Elemen paling atas dari sebuah *Tree* biasanya disebut dengan *root*.



Sebuah *tree T* harus memenuhi beberapa hal berikut:

- Jika *T* tidak kosong, *T* memiliki sebuah *node root*, yang tidak memiliki *parent*.
- Setiap *node v* pada *T*, memiliki sebuah *parent k*, setiap *v* yang memiliki *parent k* merupakan *child* dari *k*.

3.2.2 Binary Tree

Binary tree merupakan *tree* terurut yang memenuhi aturan berikut:

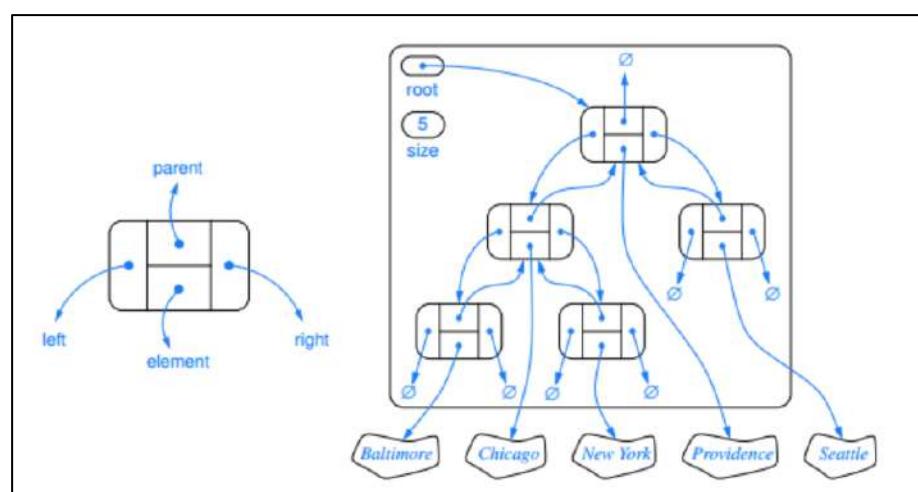
- Setiap *node* maksimal memiliki 2 *child*.
- Setiap *child* merupakan *left-child* atau *right-child*.
- *Left-child* mendahului *right-child* dalam urutan.

Sebagai *ADT*, binary tree mendukung beberapa fungsi berikut:

- *left(p)*: mengembalikan posisi *left child* dari p (atau *null* jika p tidak memiliki *left child*).
- *right(p)*: mengembalikan posisi *right child* dari p (atau *null* jika p tidak memiliki *right child*).
- *sibling(p)*: mengembalikan posisi *sibling* dari p (atau *null* jika p tidak memiliki *sibling*).

Salah satu bentuk implementasi *binary tree* adalah menggunakan *linked structure*, dimana suatu *node* memiliki alamat memori ke elemen yang disimpan pada posisi p dan memiliki alamat memori ke node lain yang terikat sebagai *children* atau *parent* dari p.

- Jika p merupakan *root* dari T, maka *parent* dari p adalah *null*.
- Jika p tidak memiliki suatu *left child* (atau *right child*), *children* yang terkait dengan p adalah *null*
- *Tree* itu sendiri memiliki variabel instan yang menyimpan alamat memori ke *node root* dan variabel yang disebut *size* yang merepresentasikan keseluruhan jumlah *node* dari *tree T*.

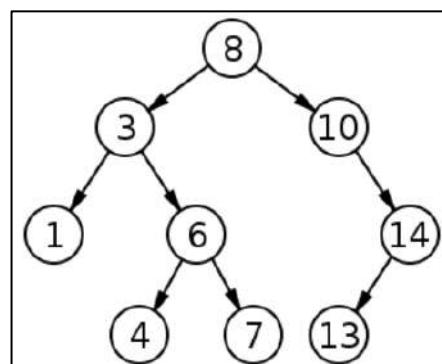


Agar dapat dilakukan update pada elemen binary tree, beberapa fungsi berikut haruslah didukung:

- *addNode(node)*: Menambahkan suatu *node* ke suatu *binary tree*. Jika *tree* kosong, maka suatu *node* menjadi *root*.
- *insertNode()*: Menambahkan/menyisipkan *node* baru ke *binary tree* yang ada.
- *deleteNode(node)*: Menghapus sebuah *node* dari suatu *binary tree*.

Terdapat tiga metode untuk mencetak semua elemen dalam *binary tree*. Urutan pencetakan elemen didasarkan pada urutan *node* yang diproses.

- *preOrder*: *Node root* akan dicetak terlebih dahulu, kemudian pohon cabang sebelah kiri, dan yang terakhir pohon cabang sebelah kanan
- *postOder*: Pohon cabang sebelah kiri akan dicetak terlebih dahulu, kemudian pohon cabang sebelah kanan dan yang terakhir *node root*
- *inOrder*: Pohon cabang sebelah kiri akan dicetak terlebih dahulu, kemudian *node root* dan yang terakhir pohon cabang sebelah kanan



Berdasarkan contoh *binary tree* diatas, berikut hasil pencetakan elemen *binary tree*.

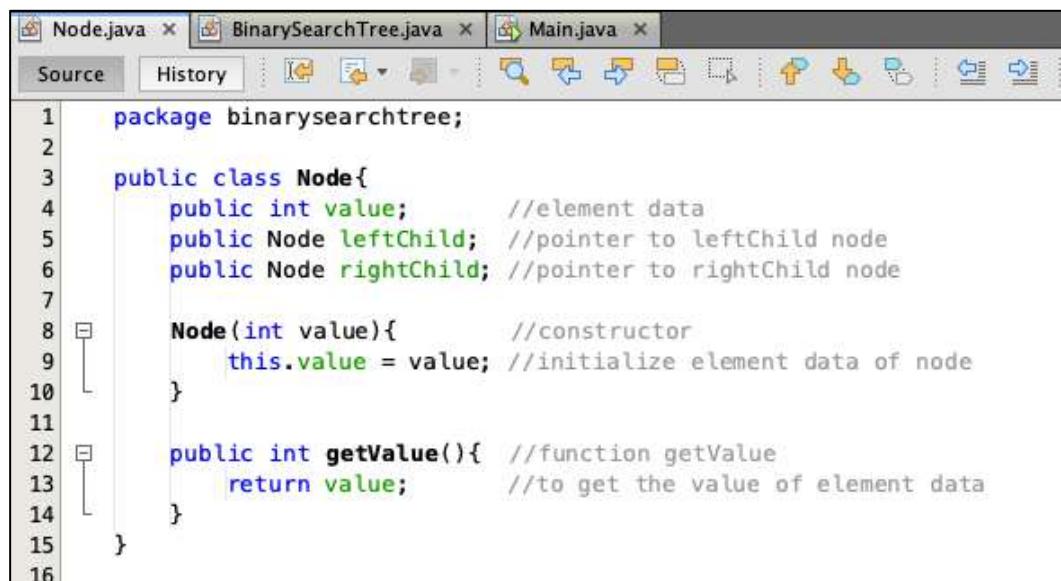
- *PREORDER (Root – Left – Right)* → 8 3 1 6 4 7 10 14 13
- *POSTODER (Left – Right – Root)* → 1 4 7 6 3 13 14 10 8
- *INODER (Left – Root – Right)* → 1 3 4 6 7 8 10 13 14

3.2.3 Binary Search Tree

Binary search tree (BST), atau biasa dikenal dengan nama *sorted binary tree* adalah suatu struktur data yang menyimpan item (seperti nomor, nama, dll.) dalam memori. *BST* mempercepat proses pencarian, penambahan, dan penghapusan data. Agar dapat dilakukan update pada elemen *binary search tree*, beberapa fungsi berikut haruslah didukung:

- *addNode(node)*: Menambahkan suatu *node* ke suatu *binary search tree*. Jika *tree* kosong, maka suatu *node* menjadi *root*.
- *insertNode()*: Menambahkan/menyisipkan *node* baru ke *binary search tree* yang ada.
- *deleteNode(node)*: Menghapus sebuah *node* dari suatu *binary search tree*.
- *searchValue(root, value)*: Suatu *static method* untuk mengecek apakah nilai yang diberikan ada pada *binary search tree* yang ada.

Berikut ini merupakan implementasi *binary search tree*. Pertama buatlah *class* *Node* dan simpan sebagai *Node.java*



```
1 package binarysearchtree;
2
3 public class Node{
4     public int value;          //element data
5     public Node leftChild;    //pointer to leftChild node
6     public Node rightChild;   //pointer to rightChild node
7
8     Node(int value){         //constructor
9         this.value = value; //initialize element data of node
10    }
11
12    public int getValue(){   //function getValue
13        return value;       //to get the value of element data
14    }
15 }
16
```

Kemudian, buatlah class *BinarySearchTree* yang mendefinisikan seluruh *method* pada *binary search tree* dan simpan sebagai *BinarySearchTree.java*.

```

1 package binarysearchtree;
2
3 public class BinarySearchTree {
4     public Node root; //ref to root node on tree
5
6     public void addNode(Node node){ //method to add a node on the tree
7         if(root == null){ //if the root is empty
8             root = node; //set node as the root of the tree
9         }
10        else{ //if the root is not empty
11            insertNode(root, node); //insert a node on the tree
12        } //using function insertNode
13    }
14
15    public void insertNode(Node parent, Node node){ //method to insert a node on the tree
16        if(parent.getValue() > node.getValue()){ //if the value of parent > node
17            if(parent.leftChild == null){ //if the leftChild of parent is null
18                parent.leftChild = node; //set node as the leftChild of parent node
19            }
20            else{ //if the leftChild of parent is not null
21                insertNode(parent.leftChild, node); //call function insertNode to
22            } //insert a node as a child of the leftchild of parent node
23        }
24        else{
25            if(parent.rightChild == null){ //if the value of parent <= node
26                parent.rightChild = node; //set node as the rightChild of parent node
27            }
28            else{ //if the rightChild of parent is not null
29                insertNode(parent.rightChild, node); //call function insertNode to
30            } //insert a node as a child of the rightChild of parent node
31        }
32    }
33
34    int minValue(Node root){ //method to get the minimum value of tree
35        int minv = root.value; //set the value of root of the tree as minv
36        while (root.leftChild != null){ //repeat if the leftChild of root is not null
37            minv = root.leftChild.value; //update minv as the value of the leftChild of root
38            root = root.leftChild; //update root as the leftchild of root
39        }
40        return minv; //return the minimum value of the tree
41    }
42
43    public void deleteNode(int value) { //method to delete node of the tree based on value
44        root = deleteFunc(root, value); //using deleteFunc function
45    }
46
47    public Node deleteFunc(Node root, int value){ //function to delete node of the tree based on value
48        if (root == null){ //if the tree is empty
49            return root; //return root
50        }
51
52        // Otherwise, recur down the tree
53        if (value < root.value) //if the value < the value of root
54            root.leftChild = deleteFunc(root.leftChild, value);
55        else if (value > root.value) //if the value > the value of root
56            root.rightChild = deleteFunc(root.rightChild, value);
57
58        // if value is same as root's value, then
59        // this is the node to be deleted
60        else{
61            // node with only one child or no child
62            if (root.leftChild == null) //if the leftChild of root is null
63                return root.rightChild; //return the rightChild of root
64            else if (root.rightChild == null) //if the rightChild of root is null
65                return root.leftChild; //return the leftChild of root
66
67            // node with two children:
68            // Get the inorder successor (smallest in the right subtree)
69            root.value = minValue(root.rightChild); //set the min value of right subtree as the value of root
70
71            // Delete the inorder successor
72            root.rightChild = deleteFunc(root.rightChild, root.value);
73        }
74    }
75}

```

```

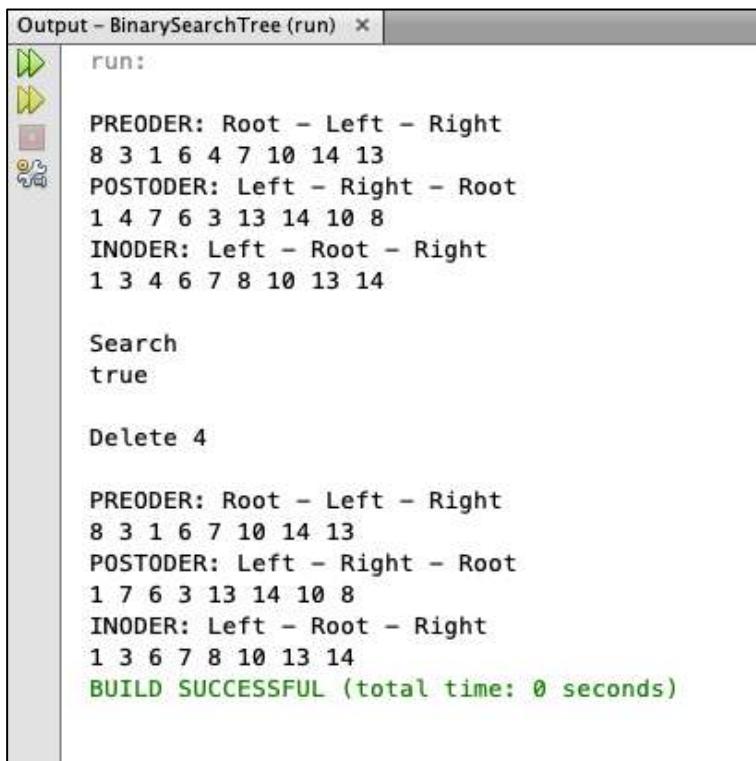
76
77     public static void preorderPrint(Node root) {
78         //method to print all elements in a binary tree in preorder process
79         //The the root node is printed first, then left subtree, and the last is right subtree
80         if (root != null) { //if root is not null
81             System.out.print( root.value + " " ); //Print the value root node
82             preorderPrint( root.leftChild ); //Print the item in left subtree recursively
83             preorderPrint( root.rightChild ); //Print the item in right subtree recursively
84         }
85     }
86
87     public static void postorderPrint(Node root) {
88         //method to print all elements in a binary tree in postorder process
89         //The left subtree is printed first, then right subtree and the last is the root node
90         if (root != null) { //if root is not null
91             postorderPrint( root.leftChild ); //Print the item in left subtree recursively
92             postorderPrint( root.rightChild ); //Print the item in right subtree recursively
93             System.out.print( root.value + " " ); //Print the value root node
94         }
95     }
96
97     public static void inorderPrint(Node root) {
98         //method to print all elements in a binary tree in postorder process
99         //The left subtree is printed first, then the root node, and the last is right subtree
100        if (root != null) { //if root is not null
101            inorderPrint( root.leftChild ); //Print the item in left subtree recursively
102            System.out.print( root.value + " " ); //Print the item in right subtree recursively
103            inorderPrint( root.rightChild ); // Print items di pohon cabang kanan
104        }
105    }
106
107    public static boolean searchValue(Node root, int value){
108        //method to searchValue in binary tree
109        if(root == null){ //if the tree is empty
110            return false; //return false
111        }
112        else{ //if the tree is not empty
113            if(root.getValue() == value){ //if the value of root = the value that we search
114                return true; //return true
115            }
116            else if(root.getValue() > value){ //if the value of root > the value that we search
117                return searchValue( root.leftChild, value ); //search the value in the leftChild of root
118            }
119            else{ //if the value of root < the value that we search
120                return searchValue( root.rightChild, value ); //search the value in the rightChild of root
121            }
122        }
123    }
124 }
125

```

Kemudian, buatlah *main method* dan simpanlah pada Main.java.

```
1 package binarysearchtree;
2
3 public class Main {
4     public static void main(String[] args) {
5         //Create a binary tree
6         BinarySearchTree bt = new BinarySearchTree();
7         bt.addNode(new Node(8));
8         bt.addNode(new Node(3));
9         bt.addNode(new Node(1));
10        bt.addNode(new Node(6));
11        bt.addNode(new Node(4));
12        bt.addNode(new Node(7));
13        bt.addNode(new Node(10));
14        bt.addNode(new Node(14));
15        bt.addNode(new Node(13));
16
17        //Print a binary tree
18        System.out.println("\nPREORDER: Root - Left - Right");
19        BinarySearchTree.preorderPrint(bt.root);
20        System.out.println("\nPOSTORDER: Left - Right - Root");
21        BinarySearchTree.postorderPrint(bt.root);
22        System.out.println("\nINORDER: Left - Root - Right");
23        BinarySearchTree.inorderPrint(bt.root);
24        System.out.println("");
25
26        //Search an element in a binary tree
27        System.out.println("\nSearch");
28        System.out.println(BinarySearchTree.searchValue(bt.root, 3));
29
30        //Delete a node
31        System.out.println("\nDelete 4");
32        bt.deleteNode(4);
33
34        //Print a binary tree
35        System.out.println("\nPREORDER: Root - Left - Right");
36        BinarySearchTree.preorderPrint(bt.root);
37
38        System.out.println("\nPOSTORDER: Left - Right - Root");
39        BinarySearchTree.postorderPrint(bt.root);
40
41        System.out.println("\nINORDER: Left - Root - Right");
42        BinarySearchTree.inorderPrint(bt.root);
43
44    }
45
46 }
```

Berikut ini merupakan hasil *output* ketika program dijalankan.



The screenshot shows the 'Output' window of a Java IDE. The title bar says 'Output - BinarySearchTree (run)'. The window contains the following text:

```
run:  
  
PREORDER: Root - Left - Right  
8 3 1 6 4 7 10 14 13  
POSTORDER: Left - Right - Root  
1 4 7 6 3 13 14 10 8  
INORDER: Left - Root - Right  
1 3 4 6 7 8 10 13 14  
  
Search  
true  
  
Delete 4  
  
PREORDER: Root - Left - Right  
8 3 1 6 7 10 14 13  
POSTORDER: Left - Right - Root  
1 7 6 3 13 14 10 8  
INORDER: Left - Root - Right  
1 3 6 7 8 10 13 14  
BUILD SUCCESSFUL (total time: 0 seconds)
```

3.3 Tugas

1. Implementasikan *binary search tree* menggunakan *linked-structure* dengan semua *method* yang sudah dijelaskan.
2. Implementasikan fungsi *delete* pada *binary search tree*.

BAB IV

BALANCED TREE: AVL

4.1 Tujuan Praktikum

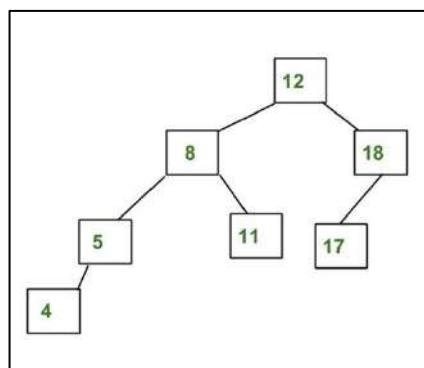
1. Praktikan dapat memahami konsep AVL Tree
2. Praktikan mampu mengimplementasikan AVL Tree

4.2 Materi

4.2.1 AVL Tree

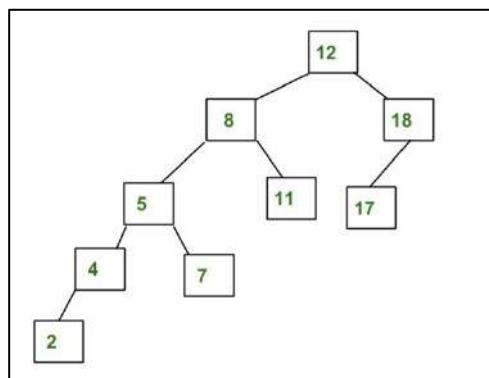
AVL tree adalah Binary Search Tree (BST) yang self-balancing atau seimbang secara otomatis dimana perbedaan tinggi antara anak pohon kiri dan kanan tidak boleh lebih dari satu node.

Contoh Tree yang termasuk AVL Tree



Tree di atas termasuk AVL karena perbedaan tinggi antara pohon kiri dan kanan kurang dari sama dengan 1.

Contoh tree yang BUKAN merupakan AVL Tree



Tree diatas bukan AVL karena perbedaan tinggi pohon di kiri dan kanan lebih dari 1 node.

Kenapa AVL Trees?

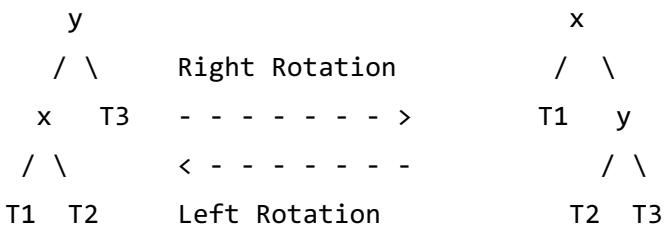
Kebanyakan dari operasi BST (search, max, min, insert, delete, dll) membutuhkan waktu $O(h)$ dimana h adalah tinggi dari BST. Cost dari operasi tersebut bisa menjadi $O(n)$ apabila tree nya tumpang. Apabila kita bisa memastikan bahwa tinggi tree tetap $O(\log n)$ setelah setiap insertion dan deletion, maka kita bisa memastikan bahwa batas atas $O(\log n)$ akan terjadi untuk semua jenis operasi. Tinggi dari AVL Tree adalah selalu $O(\log n)$ dimana n adalah jumlah nodes di tree.

4.2.2 Insertion

Untuk memastikan sebuah tree tetap AVL setelah setiap kali insertion, maka kita harus menambahkan fungsi BST insert untuk melakukan proses re-balancing atau penyeimbangan ulang. Berikut adalah dua operasi dasar yang dilakukan untuk melakukan penyeimbangan ulang BST tanpa melanggar property dari BST ($\text{keys(left)} < \text{key(root)} < \text{keys(right)}$).

- 1) Left Rotation
 - 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on the left side) or x (on the right side)



Keys in both of the above trees follow the following order:

`keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)`

So BST property is not violated anywhere.

Langkah untuk insertion

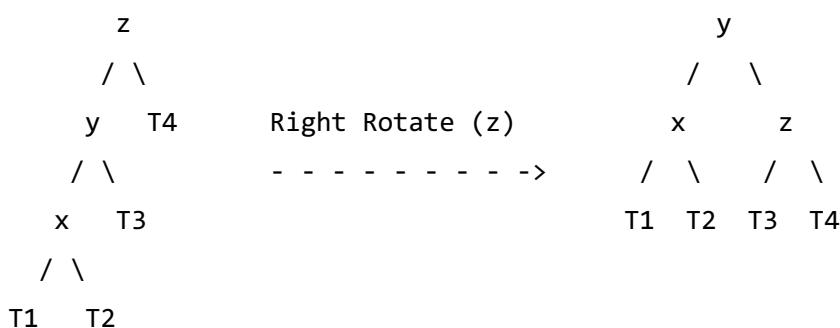
Andaikan node baru yang akan dimasukkan adalah w.

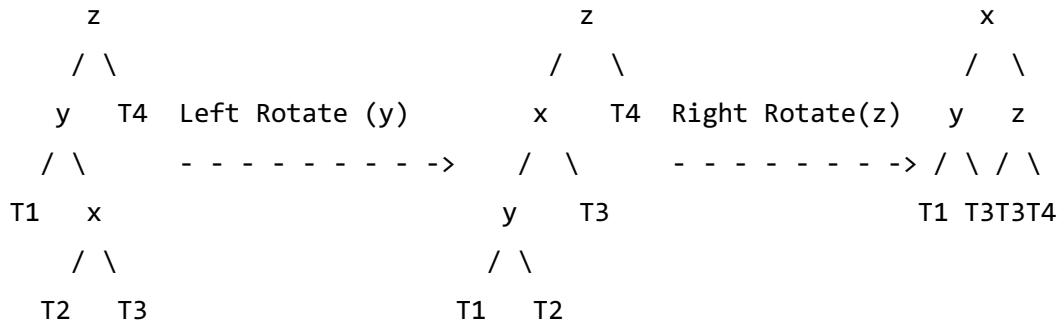
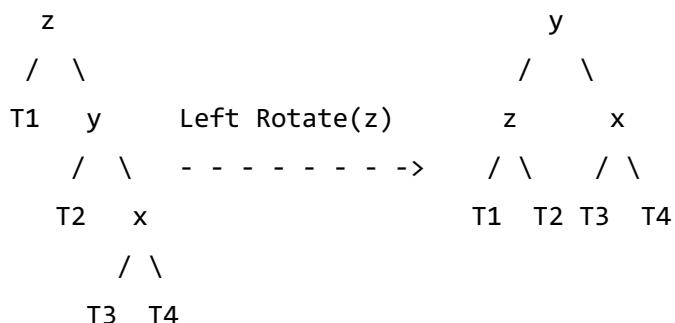
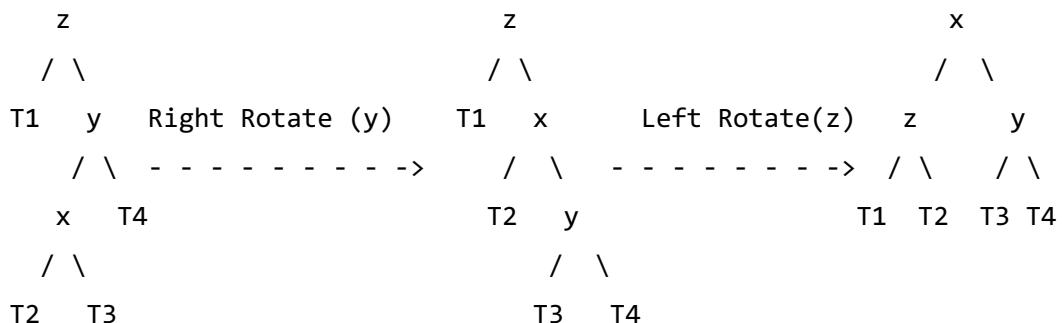
1. Lakukan standar BST insert untuk w.
2. Mulai dari w, jelajah naik dan cari node pertama yang unbalanced. Andaikan z adalah node pertama yang unbalanced, y adalah anak dari z yang berada di jalur dari w ke z dan x adalah cucu dari z yang berada di jalur dari w ke z.
3. Seimbangkan ulang tree dengan melakukan rotasi yang sesuai pada subtree dengan root z. Terdapat 4 kasus yang mungkin yang perlu diperlakukan oleh x, y, dan z dimana mereka dapat memiliki 4 jenis posisi. Berikut adalah 4 kemungkinan posisinya:
 - a. y is left child of z and x is left child of y (Left Left Case)
 - b. y is left child of z and x is right child of y (Left Right Case)
 - c. y is right child of z and x is right child of y (Right Right Case)
 - d. y is right child of z and x is left child of y (Right Left Case)

Berikut adalah operasi yang bisa dilakukan untuk keempat jenis kasus diatas. Dalam semua kasus, kita hanya membutuhkan penyeimbangan ulang subtree dengan root z dan tree yang lengkap menjadi balanced karena tinggi dari subtree (setelah rotasi yang tepat) yang berakar z tetap sama sebelum proses insertion.

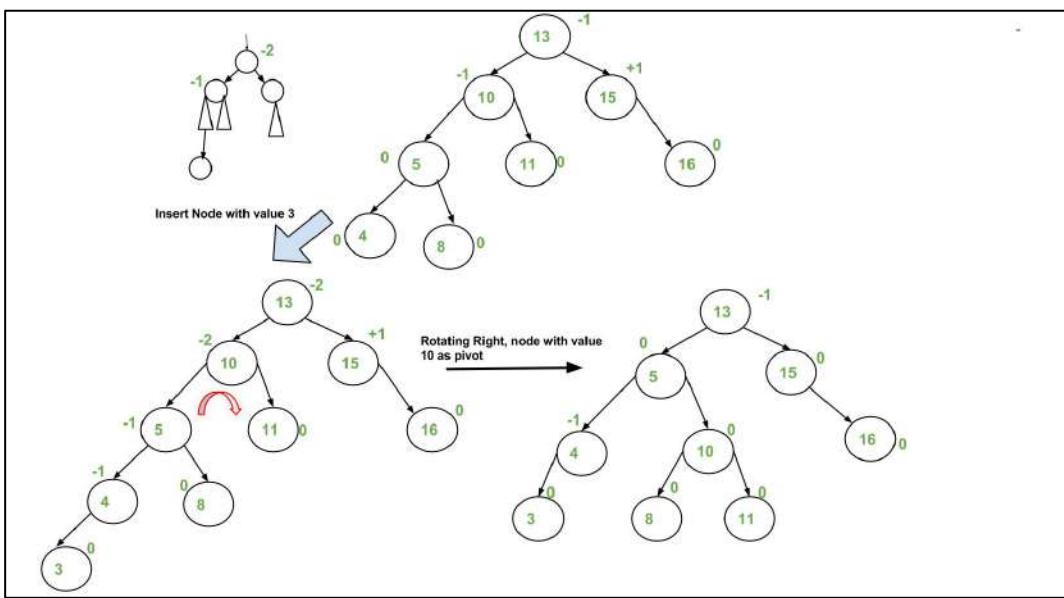
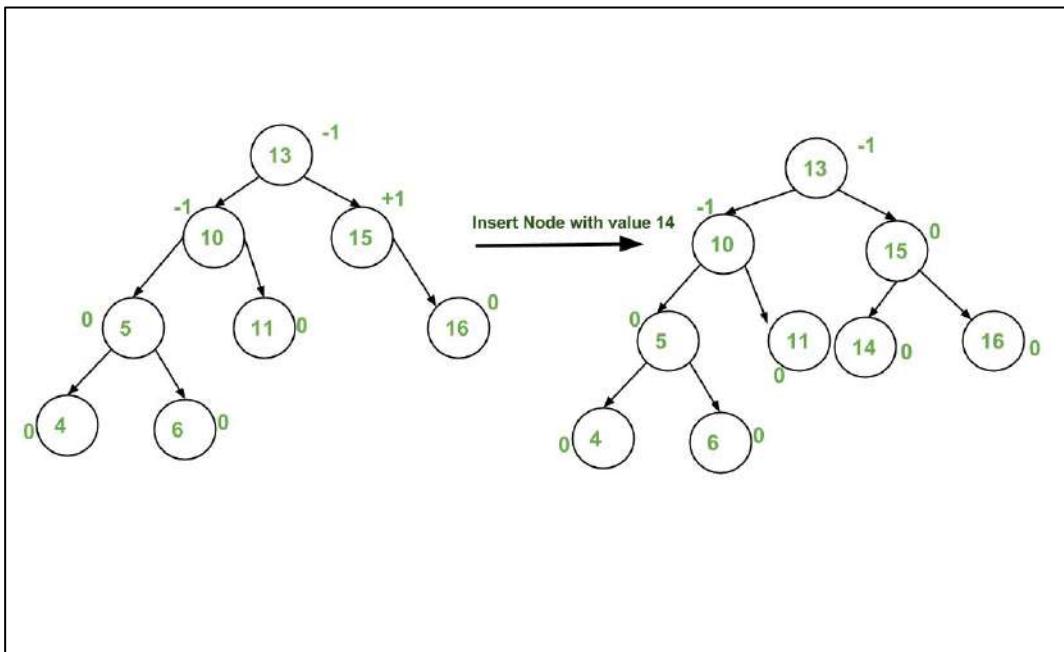
a) Left Left Case

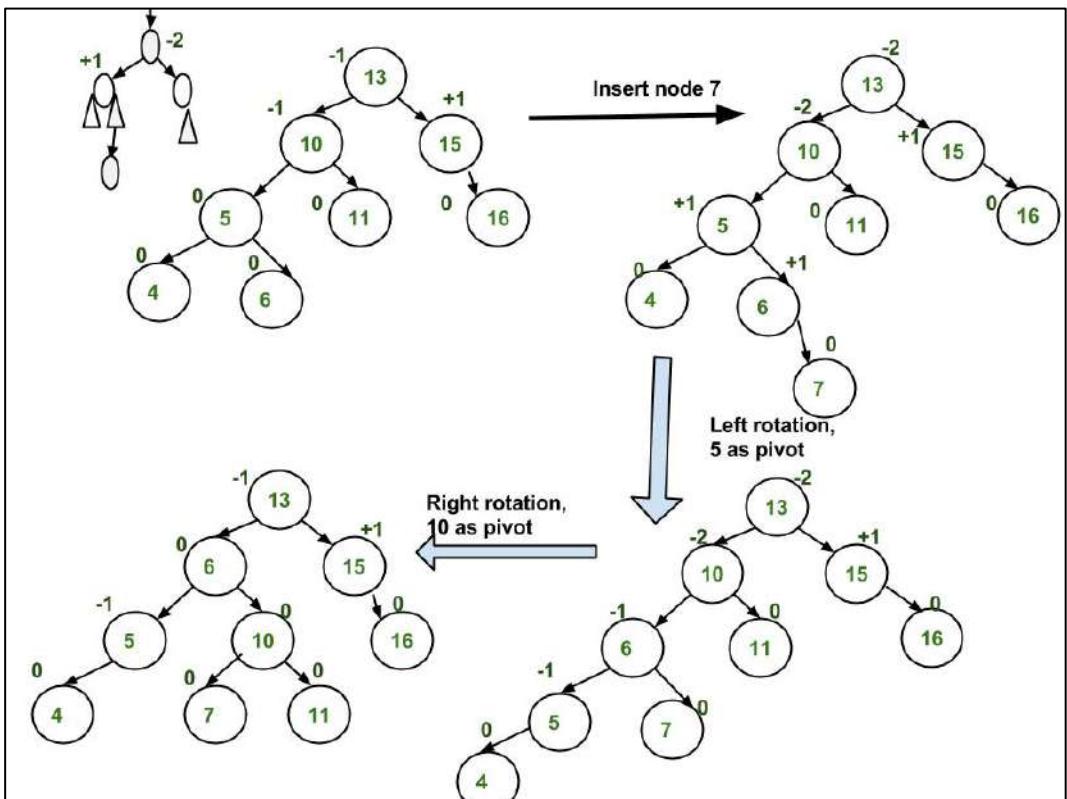
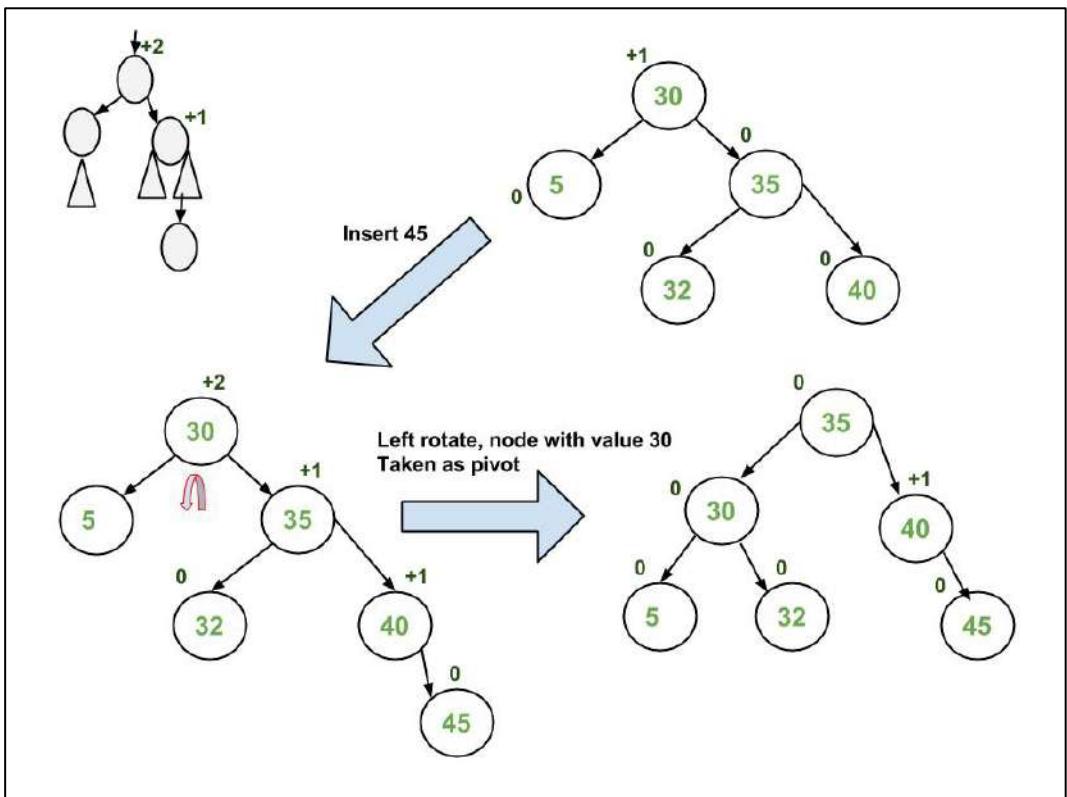
T1, T2, T3 and T4 are subtrees.

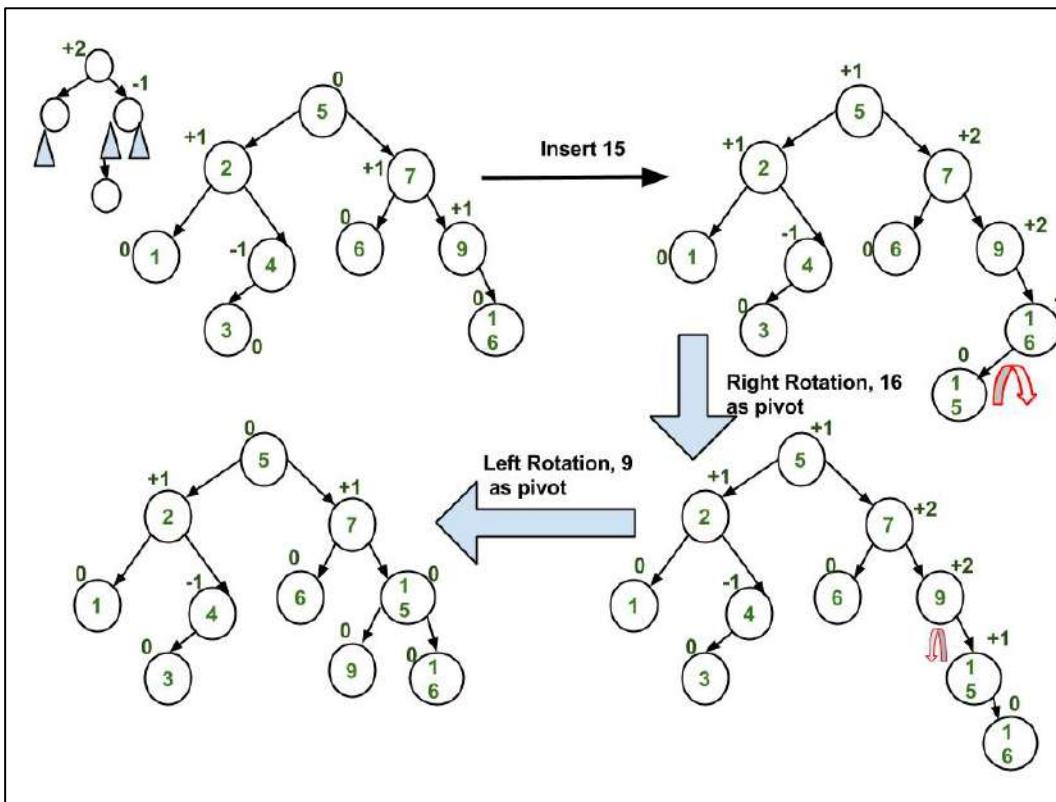


b) Left Right Case**c) Right Right Case****d) Right Left Case**

Contoh Insertion:







Implementasi

Berikut adalah implementasi proses insert pada AVL Tree. Implementasi berikut menggunakan BST insert yang rekursif. Dalam BST insert rekursif, setelah proses insertion, kita mendapat pointers untuk semua ancestors satu per satu dari bawah ke atas. Sehingga kita tidak perlu pointer parent untuk menjelajah naik. Kode rekursif secara otomatis menjelajah naik dan mengunjungi ancestors dari node baru.

1. Lakukan BST insertion normal.
2. Node sekarang haruslah salah satu ancestors dari node baru. Update tinggi dari node sekarang.
3. Dapatkan balance factor (tinggi left subtree – tinggi right subtree) dari tree sekarang.
4. Jika balance factor lebih dari 1, maka node/tree sekarang dalam keadaan unbalanced, kemudian lakukan antara Left Left case atau left Right case. Untuk mengecek apakah left left case atau bukan, bandingkan key baru yang masuk dengan key yang ada di left subtree root.
5. Jika balance factor kurang dari -1, maka node sekarang unbalanced dan antara Right Right case atau Right-Left case. Untuk mengecek apakah Right Right case

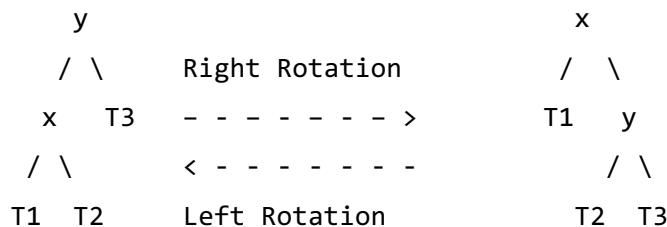
atau bukan, bandingkan key baru yang masuk dengan key yang ada di right subtree root.

4.2.3 Deletion

Untuk memastikan tree tetap AVL setelah setiap deletion, kita harus menambahkan operasi delete standar BST untuk melakukan re-balancing atau penyeimbangan ulang. Berikut adalah dua operasi dasar yang bisa dilakukan untuk melakukan penyeimbangan BST tanpa melanggar aturan BST ($\text{keys(left)} < \text{key(root)} < \text{keys(right)}$).

- 1) Left Rotation
- 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)



Keys in both of the above trees follow the following order:

$$\text{keys(T1)} < \text{key(x)} < \text{keys(T2)} < \text{key(y)} < \text{keys(T3)}$$

So BST property is not violated anywhere.

Anggap w adalah node yang akan di delete.

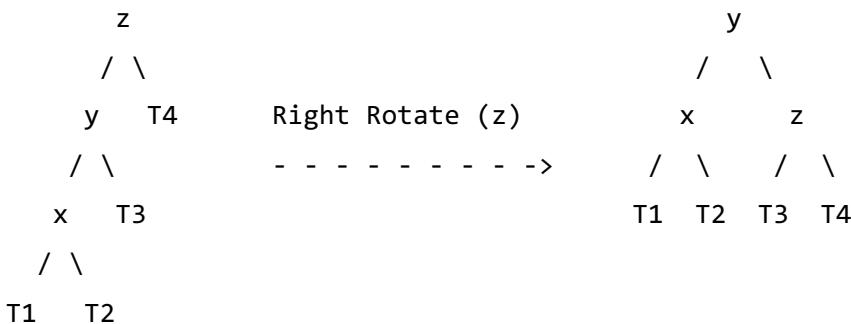
1. Lakukan standar BST delete untuk w.
2. Mulai dari w, jelajah keatas dan temukan node unbalanced pertama. Anggap z adalah node unbalanced pertama, y adalah child diatas z dan x adalah child diatas y. Perlu diperhatikan bahwa definisi x dan y berbeda dari insertion.
3. Re-balance tree dengan melakukan rotasi yang tepat pada subtree dengan root z. Terdapat 4 kemungkinan kasus yang perlu ditanggulangi karena kombinasi x, y dan z ada 4 jenis kombinasi. Berikut adalah 4 kombinasi yang mungkin:
 - a. y is left child of z and x is left child of y (Left Left Case)
 - b. y is left child of z and x is right child of y (Left Right Case)

- c. y is right child of z and x is right child of y (Right Right Case)
- d. y is right child of z and x is left child of y (Right Left Case)

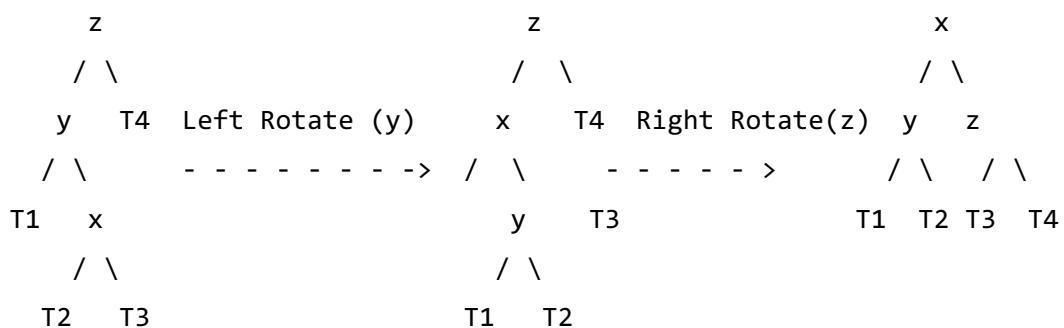
Seperti insertion, berikut adalah operasi yang dilakukan untuk 4 kasus diatas. Perlu diperhatikan, berbeda dari insertion, menyelesaikan z tidak akan membenarkan AVL tree. Setelah memperbaiki z, harus membenarkan ancestorsnya z juga.

a) Left Left Case

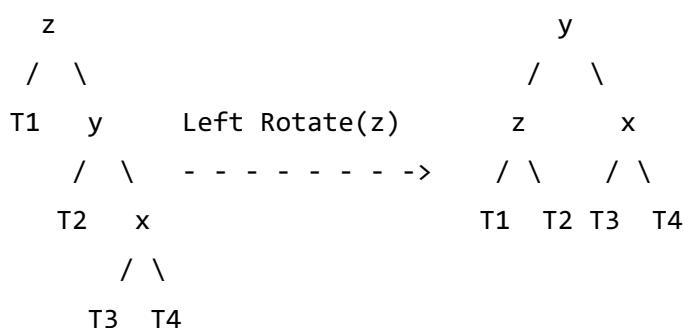
T1, T2, T3 and T4 are subtrees.



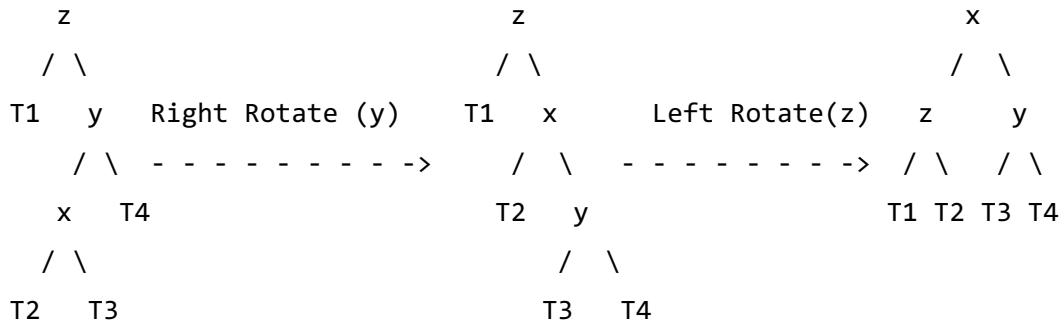
b) Left Right Case



c) Right Right Case

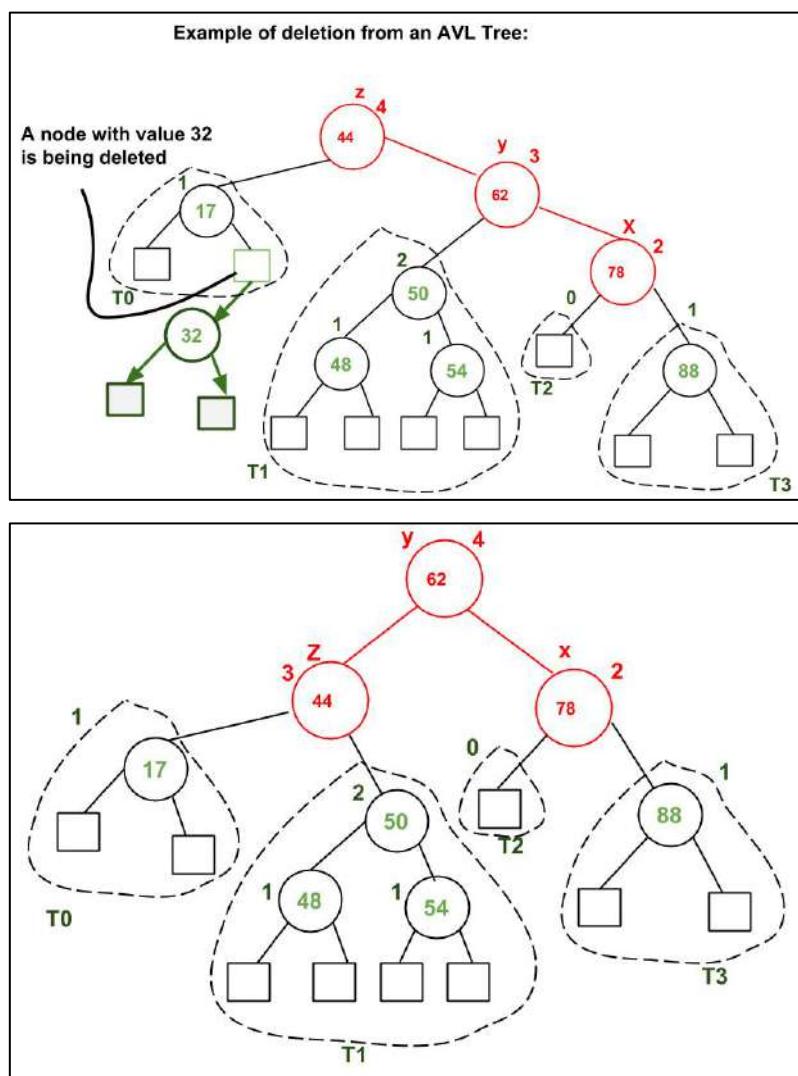


d) Right Left Case



Tidak seperti insertion, dalam deletion, setelah rotasi di z kita mungkin perlu melakukan rotasi ancestor dari z. Sehingga kita perlu lakukan trace atau pelacakan pada jalur sampai mencapai root.

Contoh deletion:



Sebuah node dengan value 32 akan dihapus/delete. Setelah menghapus 32, kita jelajahi naik dan mencari node unbalanced pertama yaitu 44. Kita tandai sebagai z, serta child atasnya sebagai y yaitu 62, dan atanya y yaitu x, bisa jadi 78 atau 50 namun dipilih 78. Sekarang kasusnya adalah Right Right, sehingga dilakukan left rotation.

Implementasi

Berikut adalah implementasi AVL Tree Deletion. Implementasi berikut menggunakan BST rekursif sebagai basis. Dalam recursive BST delete, setelah delete, kita dapat semua pointers dari ancestor satu satu dari bawah keatas. Sehingga kita tidak perlu menjelajah keatas melalui pointer parent. Kode rekursif secara sendiri menjelajah keatas dan mengunjungi semua ancestor dari node yang akan didelete.

1. Lakukan normal BST deletion.
2. Node sekarang harus satu dari ancestors dari node yang dihapus Update tinggi node sekarang.
3. Dapatkan balance factor (left subtree height – right subtree height) dari node sekarang.
4. Jika balance factor lebih dari 1, maka node sekarang adalah unbalanced dan kita pilih antara Left Left case atau Left Right case. Untuk mengecek apakah Left Left case atau Left Right case, dapatkan balance factor dari subtree kiri. Jika balance factor subtree kiri lebih dari sama dengan 0, maka Left Left case, kalau tidak Left Right case.
5. Jika balance factor kurang dari -1, maka node sekarang adalah unbalanced dan kita pilih antara Right Right case or Right Left case. Untuk mengecek apakah Right Right case or Right Left case, dapatkan balance factor dari subtree kanan. Jika balance factor subtree kanan lebih kecil atau sama dengan 0, maka Right Right case, kalau tidak Right Left case.

```

1 // Java program for deletion in AVL Tree
2
3 class Node
4 {
5     int key, height;
6     Node left, right;
7
8     Node(int d)
9     {
10         key = d;
11         height = 1;
12     }
13 }
```

```

15 class AVLTree
16 {
17     Node root;
18
19     // A utility function to get height of the tree
20     int height(Node N)
21     {
22         if (N == null)
23             return 0;
24         return N.height;
25     }
26
27     // A utility function to get maximum of two integers
28     int max(int a, int b)
29     {
30         return (a > b) ? a : b;
31     }
32
33     // A utility function to right rotate subtree rooted with y
34     // See the diagram given above.
35     Node rightRotate(Node y)
36     {
37         Node x = y.left;
38         Node T2 = x.right;
39
40         // Perform rotation
41         x.right = y;
42         y.left = T2;
43
44         // Update heights
45         y.height = max(height(y.left), height(y.right)) + 1;
46         x.height = max(height(x.left), height(x.right)) + 1;
47
48         // Return new root
49         return x;
50     }
```

```

54     Node leftRotate(Node x)
55     {
56         Node y = x.right;
57         Node T2 = y.left;
58
59         // Perform rotation
60         y.left = x;
61         x.right = T2;
62
63         // Update heights
64         x.height = max(height(x.left), height(x.right)) + 1;
65         y.height = max(height(y.left), height(y.right)) + 1;
66
67         // Return new root
68         return y;
69     }
70
71     // Get Balance factor of node N
72     int getBalance(Node N)
73     {
74         if (N == null)
75             return 0;
76         return height(N.left) - height(N.right);
77     }
78
79     Node insert(Node node, int key)
80     {
81         /* 1. Perform the normal BST rotation */
82         if (node == null)
83             return (new Node(key));
84
85         if (key < node.key)
86             node.left = insert(node.left, key);
87         else if (key > node.key)
88             node.right = insert(node.right, key);
89         else // Equal keys not allowed
90             return node;
91
92         /* 2. Update height of this ancestor node */
93         node.height = 1 + max(height(node.left),
94                               height(node.right));
95
96         /* 3. Get the balance factor of this ancestor
97            node to check whether this node became
98            Unbalanced */
99         int balance = getBalance(node);
100
101        // If this node becomes unbalanced, then
102        // there are 4 cases Left Left Case
103        if (balance > 1 && key < node.left.key)
104            return rightRotate(node);
105
106        // Right Right Case
107        if (balance < -1 && key > node.right.key)
108            return leftRotate(node);
109
110        // Left Right Case
111        if (balance > 1 && key > node.left.key)
112        {
113            node.left = leftRotate(node.left);
114            return rightRotate(node);
115        }
116
117        // Right Left Case
118        if (balance < -1 && key < node.right.key)
119        {
120            node.right = rightRotate(node.right);
121            return leftRotate(node);
122        }
123
124        /* return the (unchanged) node pointer */
125        return node;
126    }

```

```

128+ /* Given a non-empty binary search tree, return the
129 node with minimum key value found in that tree.
130 Note that the entire tree does not need to be
131 searched. */
132 Node minValueNode(Node node)
133 {
134     Node current = node;
135
136     /* loop down to find the leftmost leaf */
137     while (current.left != null)
138         current = current.left;
139
140     return current;
141 }
142
143 Node deleteNode(Node root, int key)
144 {
145     // STEP 1: PERFORM STANDARD BST DELETE
146     if (root == null)
147         return root;
148
149     // If the key to be deleted is smaller than
150     // the root's key, then it lies in left subtree
151     if (key < root.key)
152         root.left = deleteNode(root.left, key);
153
154     // If the key to be deleted is greater than the
155     // root's key, then it lies in right subtree
156     else if (key > root.key)
157         root.right = deleteNode(root.right, key);
158
159     // if key is same as root's key, then this is the node
160     // to be deleted
161     else
162     {
163
164         // node with only one child or no child
165         if ((root.left == null) || (root.right == null))
166         {
167             Node temp = null;
168             if (temp == root.left)
169                 temp = root.right;
170             else
171                 temp = root.left;
172
173             // No child case
174             if (temp == null)
175             {
176                 temp = root;
177                 root = null;
178             }
179             else // One child case
180                 root = temp; // Copy the contents of
181                             // the non-empty child
182         }
183         else
184         {
185
186             // node with two children: Get the inorder
187             // successor (smallest in the right subtree)
188             Node temp = minValueNode(root.right);
189
190             // Copy the inorder successor's data to this node
191             root.key = temp.key;
192
193             // Delete the inorder successor
194             root.right = deleteNode(root.right, temp.key);
195         }
196     }

```

```

197
198     // If the tree had only one node then return
199     if (root == null)
200         return root;
201
202     // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
203     root.height = max(height(root.left), height(root.right)) + 1;
204
205     // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
206     // this node became unbalanced)
207     int balance = getBalance(root);
208
209     // If this node becomes unbalanced, then there are 4 cases
210     // Left Left Case
211     if (balance > 1 && getBalance(root.left) >= 0)
212         return rightRotate(root);
213
214     // Left Right Case
215     if (balance > 1 && getBalance(root.left) < 0)
216     {
217         root.left = leftRotate(root.left);
218         return rightRotate(root);
219     }
220
221     // Right Right Case
222     if (balance < -1 && getBalance(root.right) <= 0)
223         return leftRotate(root);
224
225     // Right Left Case
226     if (balance < -1 && getBalance(root.right) > 0)
227     {
228         root.right = rightRotate(root.right);
229         return leftRotate(root);
230     }
231
232     return root;
233 }
234

```

```

235     // A utility function to print preorder traversal of
236     // the tree. The function also prints height of every
237     // node
238     void preOrder(Node node)
239     {
240         if (node != null)
241         {
242             System.out.print(node.key + " ");
243             preOrder(node.left);
244             preOrder(node.right);
245         }
246     }
247
248     public static void main(String[] args)
249     {
250         AVLTree tree = new AVLTree();
251
252         /* Constructing tree given in the above figure */
253         tree.root = tree.insert(tree.root, 9);
254         tree.root = tree.insert(tree.root, 5);
255         tree.root = tree.insert(tree.root, 10);
256         tree.root = tree.insert(tree.root, 0);
257         tree.root = tree.insert(tree.root, 6);
258         tree.root = tree.insert(tree.root, 11);
259         tree.root = tree.insert(tree.root, -1);
260         tree.root = tree.insert(tree.root, 1);
261         tree.root = tree.insert(tree.root, 2);
262
263         /* The constructed AVL Tree would be
264          9
265          / \
266          1 10
267          / \ \
268          0 5 11
269          / / \
270          -1 2 6
271          */
272         System.out.println("Preorder traversal of "+
273                         "constructed tree is : ");
274         tree.preOrder(tree.root);
275
276         tree.root = tree.deleteNode(tree.root, 10);
277
278         /* The AVL Tree after deletion of 10
279          1
280          / \
281          0 9
282          /   / \
283          -1 5 11
284          / \
285          2 6
286          */
287         System.out.println("");
288         System.out.println("Preorder traversal after "+
289                         "deletion of 10 : ");
290         tree.preOrder(tree.root);
291     }
292 }
```

4.3 Tugas

1. Implementasikan AVL pada bab ini dengan data $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ secara terurut. Setelah memasukkan semua data tersebut, buatlah gambar AVL Tree yang terbuat!
2. Dari implementasi Binary Search Tree yang terdapat pada Bab 3, buatlah satu method untuk mengecek apakah Binary Search Tree yang terbentuk merupakan AVL Tree atau bukan.

BAB V

SORTED TREE: HEAP TREE

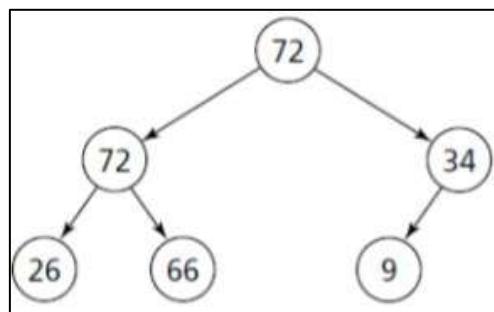
5.1 Tujuan Praktikum

1. Praktikan dapat memahami konsep Heap Tree
2. Praktikan mampu mengimplementasikan Heap Tree

5.2 Materi

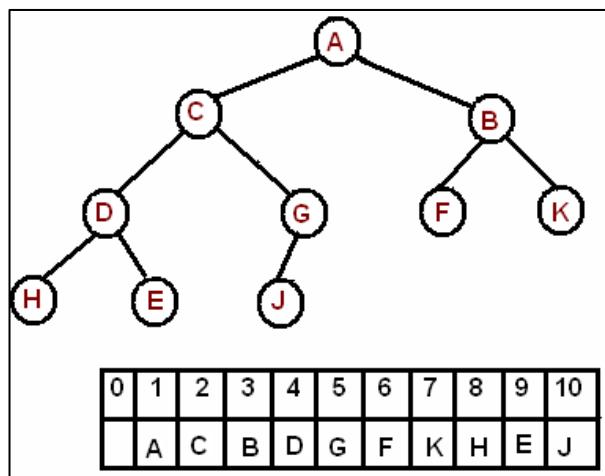
Sebuah heap adalah sebuah binary tree yang memenuhi property sebagai berikut:

- Tree haruslah complete. Maksudnya adalah setiap level dari tree harus terisi (sebuah parent harus memiliki leftchild dan right child), kecuali pada level paling bawah boleh tidak complete. Jika level paling bawah tidak complete maka child yang harus dikosongkan adalah right child.
- Untuk setiap elemen E, semua nilai yang merupakan child dari E harus kurang dari atau sama dengan nilai dari E. Oleh karena itu, root akan menyimpan nilai paling maksimum.



5.2.1 Array Implementation

Dalam pengindex-an pada array, heap tree memiliki aturan untuk menyimpan value seperti berikut:



Dalam kasus ini root diletakkan pada index “1” untuk memudahkan perhitungan index. Dapat dicermati bahwa untuk node selain root berlaku,

- Untuk setiap left child berada pada index $2*k$
- Untuk setiap right child berada pada index $2*k+1$
- Dan parent berada di index $k/2$

Heap tree dibentuk dari node-node yang memiliki key sebagai value. Oleh karena itu dibutuhkan kelas node seperti berikut

```
1 public class Node {  
2     private int iData;  
3  
4     public Node(int key){  
5         iData = key;  
6     }  
7  
8     public int getKey(){  
9         return iData;  
10    }  
11  
12    public void setKey(int id){  
13        iData = id;  
14    }  
15 }
```

Setelah buat kelas node, maka dibuat kelas heap

```

1 - public class Heap {
2     private Node[] heapArray;
3     private int maxSize;
4     private int currentSize;
5
6     public Heap(int mx){
7         maxSize = mx;
8         currentSize = 0;
9         heapArray = new Node[maxSize + 1];
10    }
11
12    public boolean isEmpty(){
13        return currentSize == 0;
14    }
15
16    public boolean isFull(){
17        return currentSize == maxSize;
18    }
19
20    public boolean hasLeftChild(int index){
21        return 2*index <= currentSize;
22    }
23
24    public boolean hasRightChild(int index){
25        return 2*index + 1 <= currentSize;
26    }

```

Setelah dibentuk kelas Heap, maka dibutuhkan method untuk insert, remove, dan untuk menjaga agar property heap selalu dipenuhi.

5.2.2 Insert

Elemen baru selalu ditambahkan pada level heap paling bawah. Property dari heap selalu dijaga dengan cara membandingkan elemen yang ditambahkan dengan parentnya dan kemudian swap position apa bila elemen lebih besar daripada parentnya. Perlu diingat bahwa tujuan dari property ini adalah untuk menjaga agar nilai parent selalu lebih besar daripada childnya.

```

28+    public boolean insert(int key){
29+        if (isFull()){
30+            return false;
31+        }
32+
33        Node newNode = new Node(key);
34        currentSize++;
35        heapArray[currentSize] = newNode;
36        trickleUp(currentSize);
37        return true;
38    }
39
40+    public void trickleUp(int index){
41        int parent = index/2;
42        Node bottom = heapArray[index];
43
44+        while (index>1 && heapArray[parent].getKey() < bottom.getKey()){
45            heapArray[index] = heapArray[parent];
46            index = parent;
47            parent = index/2;
48        }
49
50        heapArray[index] = bottom;
51    }

```

5.2.3 Remove

Pada heap tree jika dikenai dengan method remove() maka akan membuat node dengan value terbesar dihapus. Kemudian root akan digantikan dengan node paling kanan bawah. Kemudian heap tree akan menjaga propertinya dengan aturan yang sudah ada yang akan membawa node paling kanan bawah tersebut ke posisi yang paling sesuai pada heap tree.

```

53 *     public Node remove(){
54         Node root = heapArray[1];
55         heapArray[1] = heapArray[currentSize];
56         currentSize--;
57         trickleDown(1);
58         return root;
59     }
60
61     public void trickleDown(int index){
62         int largerChild;
63         Node top = heapArray[index];
64
65         while (hasLeftChild(index)){ // has at least left child
66             int leftChild = 2*index;
67             int rightChild = leftChild + 1;
68
69             if (hasRightChild(index) &&
70                 heapArray[rightChild].getKey() > heapArray[leftChild].getKey()){
71                 largerChild = rightChild;
72             }
73             else{
74                 largerChild = leftChild;
75             }
76
77             if (top.getKey() >= heapArray[largerChild].getKey()){ // stop looping
78                 break;
79             }
80
81             heapArray[index] = heapArray[largerChild];
82             index = largerChild;
83         }
84         heapArray[index] = top;
85     }

```

Delete pada method ini adalah delete max, sehingga jika method dipanggil maka key pada root akan terhapus karena key pada root pastilah key paling besar/maksimum. Sesudah root dihapus maka method trickle down akan mencari key yang tepat untuk menjadi root pengganti sekaligus untuk menjaga property heap tetap terpenuhi.

5.2.4 Print

Setiap value/key dari heap tree disimpan dalam array. Maka untuk menampilkan heap tree digunakan property heap agar bentuk tree dapat terlihat.

```

87     public void displayHeap(){
88         System.out.print("heapArray : ");
89
90         for (int i=1 ; i<=currentSize ; i++){
91             System.out.print(heapArray[i].getKey() + " ");
92         }
93
94         System.out.println("");
95     }
96 }
```

5.3 Tugas

1. Buatlah heap tree untuk array dengan ukuran 15 dan data key:
Key = {78, 3, 9, 10, 23, 77, 34, 86, 90, 100, 20, 66, 94, 63, 97}
2. Buatlah fungsi heap sort dengan memanfaatkan heap dari bab ini.

BAB VI

GRAF

6.1 Tujuan Praktikum

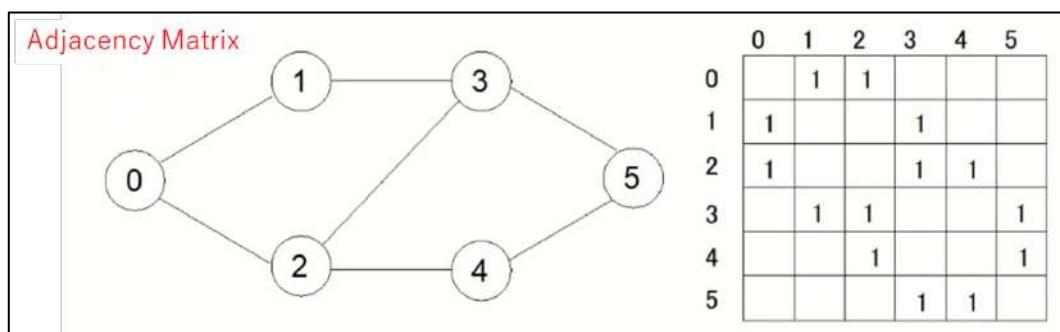
1. Mahasiswa mampu memahami struktur data graf.
2. Mahasiswa mampu merepresentasikan graf dengan adjacency matrix dan adjacency list.
3. Mahasiswa mampu memahami konsep BFS dan DFS pada graf.

6.2 Materi

6.2.1 Graf

Graf merupakan salah satu struktur data yang praktis dan sering digunakan. Sebuah graf terdiri dari simpul (nodes) dan cabang (branches). Sebuah cabang menunjukkan koneksi/hubungan antar node. Dalam bab ini, kita akan mengimplementasikan graf di java.

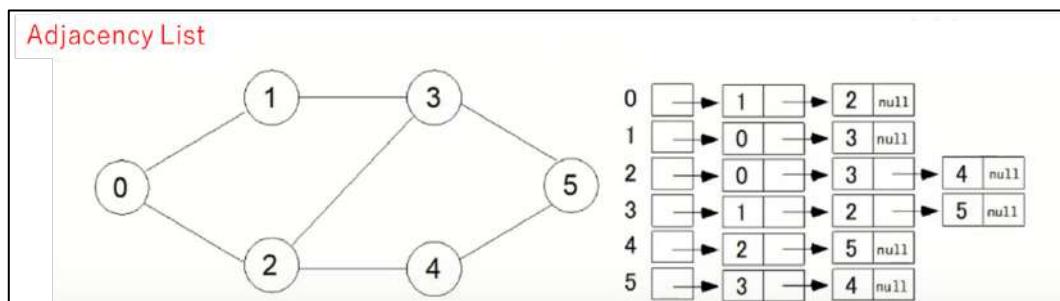
Suatu graf dilambangkan dengan $G = \{V, E\}$. Dimana V adalah himpunan simpul dan E adalah himpunan garis. Graf juga dibagi menjadi graf berarah dan graf tak berarah. Graf berarah adalah graf yang ujung-ujungnya ditambahkan informasi arah. Ada dua cara yang umum digunakan untuk merepresentasikan graf. Salah satunya menggunakan matriks ketetanggaan (adjacency matrix) dan yang lainnya dengan daftar ketetanggaan (adjacency list). Gambar 1 menunjukkan contoh adjacency matrix dari suatu graf dengan 6 simpul dan 7 garis.



Gambar 1. Representasi graf menggunakan adjacency matrix

Dalam hal ini, adjacency matrix adalah matriks persegi berukuran 6x6. Node-0 terhubung ke node-1 dan node-2, sehingga "1" ada di kolom-1 dan kolom-2 baris-0 dari matriks. Demikian pula, Node-1 terhubung dengan Node-0 dan Node-3, sehingga "1" ada di kolom-0 dan kolom-3 dari baris-1 dan seterusnya. Angka "1" ada jika ada sisi yang menghubungkan dua simpul.

Jika graf yang sama direpresentasikan menggunakan adjacency list, maka akan terlihat seperti Gambar 2. Node-0 terhubung node-1 dan node-2, sehingga akan ada linked list dengan koneksi berikut $0 \rightarrow 1 \rightarrow 2$. Karena tidak ada yang lain setelah 2, bagian yang menjadi penunjuk di sebelahnya berisi nol. Hal yang sama berlaku untuk node-1 yang terhubung dengan node-0 dan node-3 ($1 \rightarrow 0 \rightarrow 3$).



Gambar 2. Representasi graf menggunakan adjacency list

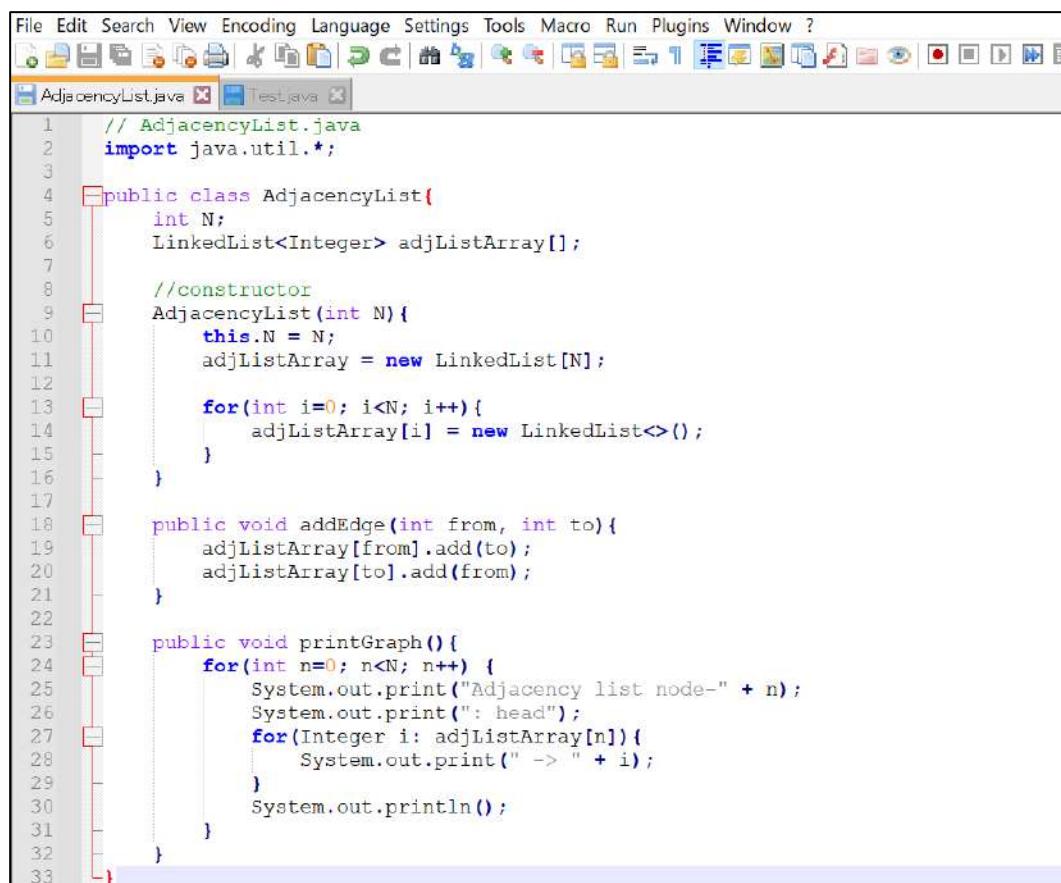
Dalam aplikasi di dunia nyata, seringkali kita perlu menelusuri/mencari graf untuk menemukan beberapa informasi penting (seperti rute terpendek di google map dll). Traversal (pencarian) dalam suatu graf hampir sama dengan yang ada di tree, bedanya, tidak seperti tree, graf dapat memiliki cycle, sehingga kita dapat kembali ke vertex yang sama. Untuk menghindari pemrosesan vertex lebih dari sekali, kita dapat menggunakan array yang dikunjungi dengan tipe boolean. Ada dua jenis traversal graf: BFS (Breadth First Search/Traversal) dan DFS (Depth First Search/Traversal).

BFS sangat berguna untuk menemukan jalur terpendek pada graf tak berbobot. BFS dimulai pada suatu sembarang vertex dari graf dan mengunjungi vertex tetangga terlebih dahulu, sebelum pindah ke tetangga tingkat berikutnya. BFS menggunakan struktur data queue untuk melacak vertex mana yang akan dikunjungi selanjutnya. Setelah mencapai vertex baru, BFS menambahkannya ke queue untuk mengunjunginya nanti.

Sementara itu, DFS sendiri tidak terlalu berguna, tetapi ketika digunakan untuk melakukan tugas lain seperti menghitung komponen yang terhubung (connected components), menentukan suatu graf terhubung atau tidak, atau menemukan titik jembatan/artikulasi maka DFS lebih mudah diaplikasikan. Seperti namanya, DFS memanfaatkan kedalaman terlebih dahulu pada graf tanpa memperhatikan garis mana yang dibutuhkan selanjutnya sampai tidak dapat melangkah lebih jauh di titik mana ia mundur dan melanjutkan.

6.2.2 Implementasi Java

Implementasi java untuk merepresentasikan graf yang ditunjukkan pada Gambar 2 adalah seperti yang ditunjukkan di bawah ini. Di sini, ada dua kelas: AdjacencyList dan Test, seperti namanya, satu digunakan untuk representasi graf menggunakan linked list, sementara yang lain untuk pengujian.



```
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
AdjacencyList.java TestJava.java

1 // AdjacencyList.java
2 import java.util.*;
3
4 public class AdjacencyList{
5     int N;
6     LinkedList<Integer> adjListArray[];
7
8     //constructor
9     AdjacencyList(int N){
10         this.N = N;
11         adjListArray = new LinkedList[N];
12
13         for(int i=0; i<N; i++){
14             adjListArray[i] = new LinkedList<>();
15         }
16     }
17
18     public void addEdge(int from, int to){
19         adjListArray[from].add(to);
20         adjListArray[to].add(from);
21     }
22
23     public void printGraph(){
24         for(int n=0; n<N; n++) {
25             System.out.print("Adjacency list node-" + n);
26             System.out.print(": head");
27             for(Integer i: adjListArray[n]){
28                 System.out.print(" -> " + i);
29             }
30             System.out.println();
31         }
32     }
33 }
```

```
1 // Test.java
2
3 public class Test{
4     public static void main(String args[]){
5         int N = 6;
6         AdjacencyList list = new AdjacencyList(N);
7
8         list.addEdge(0, 1);
9         list.addEdge(0, 2);
10        list.addEdge(1, 3);
11        list.addEdge(2, 3);
12        list.addEdge(2, 4);
13        list.addEdge(3, 5);
14        list.addEdge(4, 5);
15
16        list.printGraph();
17    }
18}
```

Hasil setelah menjalankan class Test adalah sebagai berikut.

```
Adjacency list node-0: head -> 1 -> 2
Adjacency list node-1: head -> 0 -> 3
Adjacency list node-2: head -> 0 -> 3 -> 4
Adjacency list node-3: head -> 1 -> 2 -> 5
Adjacency list node-4: head -> 2 -> 5
Adjacency list node-5: head -> 3 -> 4
```

Tambahkan method berikut ke dalam file AdjacencyList.java untuk mengimplementasikan BFS.

```
public void bfs(int start){
    boolean visited[] = new boolean[N];
    LinkedList<Integer> queue = new LinkedList();

    visited[start] = true;
    queue.add(start);

    while(queue.size() != 0){
        start = queue.poll();
        System.out.print(start+" ");

        Iterator<Integer> i = adjListArray[start].listIterator();
        while(i.hasNext()){
            int n = i.next();
            if(!visited[n]){
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}
```

Panggil method tadi dari Test.java menggunakan list.bfs(start); ganti start dengan nomor vertex dari mana Anda ingin memulai pencarian/kunjungan.

6.3 Tugas

1. Implementasikan representasi graf dengan menggunakan adjacency matrix.
2. Modifikasilah method bfs() agar dapat digunakan dalam adjacency matrix.
3. Pelajari lebih lanjut terkait DFS dan tambahkan method dengan nama dfs() pada kode di atas.

BAB VII

SHORTEST PATH PROBLEM DENGAN ALGORITMA

DIJKSTRA

7.1 Capaian Pembelajaran

1. Siswa mampu merepresentasikan graf dengan edge yang berbobot
2. Siswa mampu menyelesaikan permasalahan shortest path menggunakan algoritma Djikstra

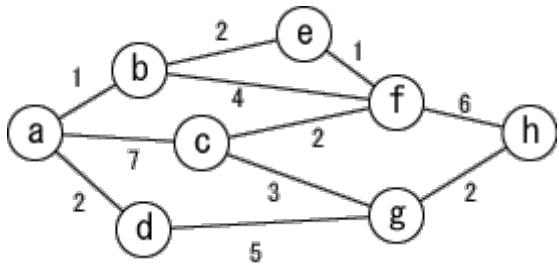
7.2 Materi

7.2.1 Algoritma Dijkstra

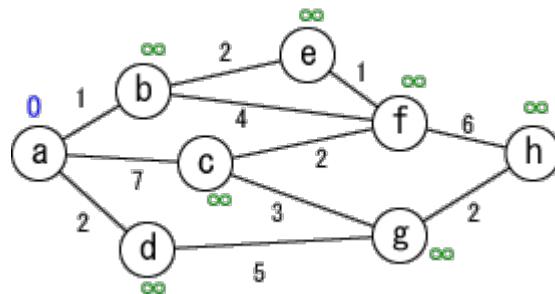
Algoritma Dijkstra adalah metode pencarian jalur terpendek pada sebuah graf di mana pencarian dimulai dengan melihat jarak node-node yang terhubung langsung dengan node awal. Jangkauan pencarian kemudian diperluas secara bertahap menuju node-node yang jaraknya jauh dari node awal. Algoritma ini bekerja sebagai berikut.

1. Jarak ke setiap node diasumsikan "tidak diketahui" di awal dan pertama-tama diatur menjadi bernilai tak terhingga (infinity).
2. Jarak ke starting node diset menjadi 0.
3. Pilih node dengan jarak terpendek dari node-node yang belum dipilih sebelumnya dan catat jarak tersebut sebagai jarak terpendek sementara.
4. Hitung jarak node yang "terhubung langsung" dengan node yang dipilih pada langkah 3. Jika jaraknya lebih pendek dari jarak yang diset sebelumnya, perbarui jaraknya.
5. Algoritma akan berhenti jika seluruh node sudah dipilih. Jika belum, maka kembali ke 3.

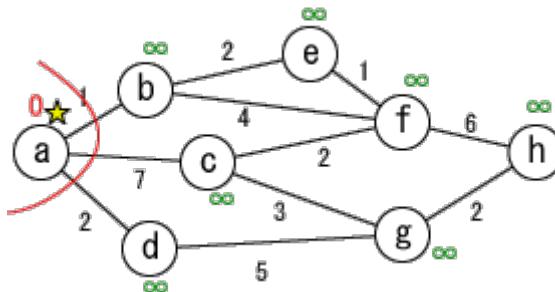
Untuk lebih memahami bagaimana algoritma ini bekerja dalam praktiknya, asumsikan graf berikut:



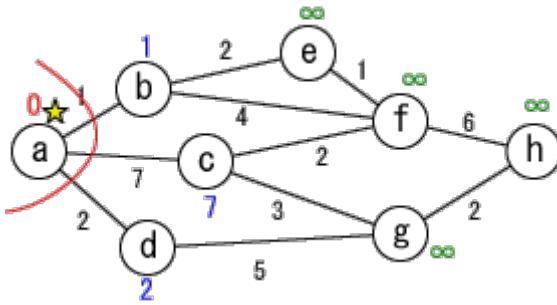
Mari kita cari jarak terpendek ketika node awal adalah node-a dan node tujuan adalah node h.



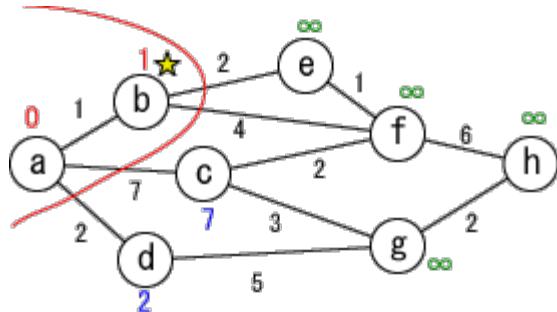
Jarak ke setiap node dalam graf ini dituliskan di atas/di bawah setiap node. Pada awalnya, jarak ini tidak diketahui dan diset seingga bernilai tak hingga (infinity). Karena node-a adalah node awal, jarak ke dirinya sendiri adalah 0.



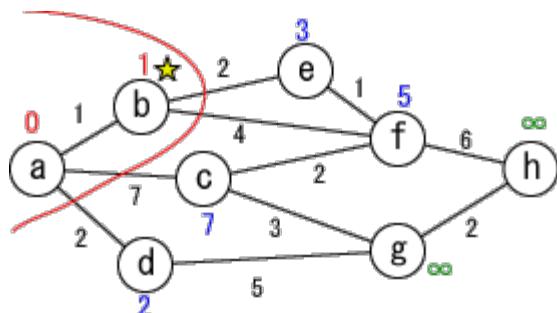
Sekarang, kita harus memilih node dengan nilai jarak terkecil. Node-a akan dipilih karena saat ini jaraknya adalah 0. Jarak ini kemudian ditetapkan sebagai jarak terpendek yang ditemukan sejauh ini. Kurva merah digunakan untuk menandai semua node yang sudah dipilih (saat ini hanya node-a), node di luar kurva ini statusnya "belum dipilih". Node yang baru dipilih ditandai dengan tanda bintang.



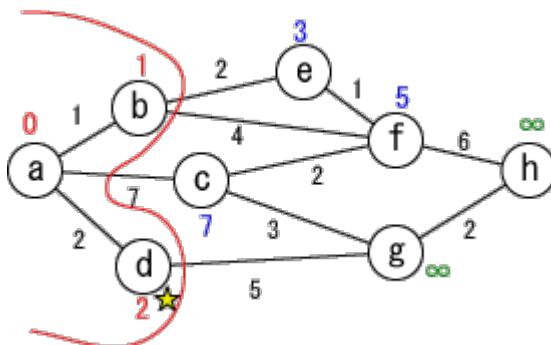
Jarak dari node awal (node-a) ke node yang terhubung langsung dengannya tetapi statusnya "belum dipilih", yakni node-(b, c, d), akan dihitung dan nilainya diperbarui jika jaraknya lebih kecil dari jarak sebelumnya. Jarak dari node-a ke node-(b, c, d) nilainya diperbarui secara berturut-turut dari ∞ menjadi 1, 7, dan 2.



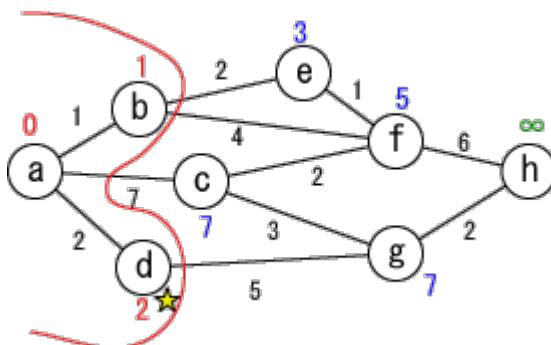
Sekarang, node-b akan dipilih karena ia memiliki jarak paling pendek (1) dibandingkan dengan node-node yang "belum dipilih" lainnya. Jarak ini kemudian diset sebagai jarak terpendek yang ditemukan sejauh ini. Jika ada beberapa node dengan jarak terpendek yang sama, Anda dapat memilih salah satunya. Node-b yang baru dipilih ditandai dengan bintang.



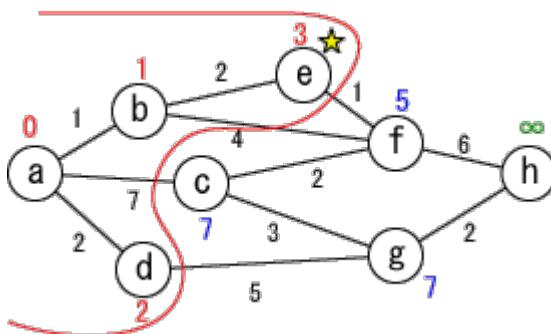
Jarak dari node-a ke node yang terhubung langsung dengan node-b tetapi statusnya masih "belum dipilih", akan dihitung dan diperbarui jika nilainya lebih kecil dari nilai jarak mereka sebelumnya. Node-node tersebut yaitu node-(e, f). Jarak dari node-a ke node-(e, f) diperbarui dari ∞ berturut-turut menjadi 3 dan 5



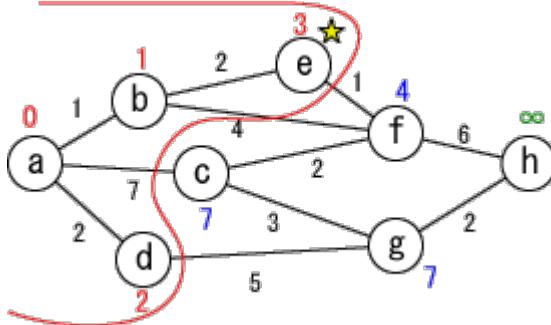
Berikutnya, node-d akan dipilih karena ia memiliki jarak paling pendek (2) dibandingkan dengan node-node yang “belum dipilih” lainnya. Jarak ini kemudian diset sebagai jarak terpendek yang ditemukan sejauh ini. Node-d yang baru dipilih ditandai dengan bintang.



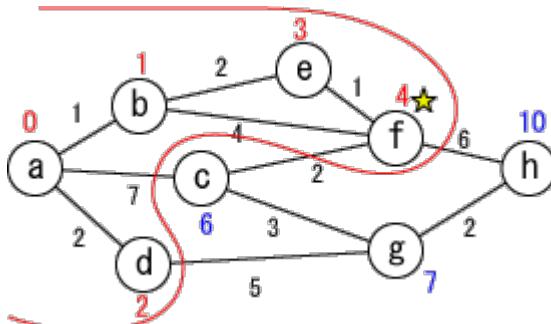
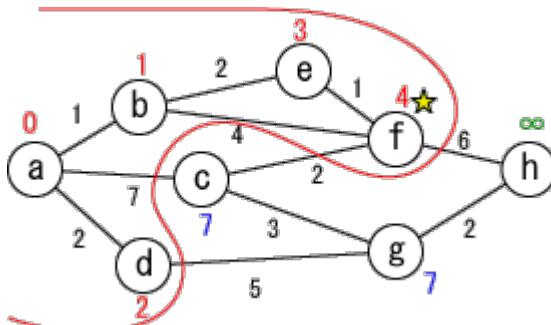
Jarak dari node-a ke node yang terhubung langsung dengan node-d tetapi statusnya masih "belum dipilih", akan dihitung dan diperbarui jika nilainya lebih kecil dari nilai jarak sebelumnya. Node tersebut yaitu node-g. Jarak dari node-a ke node-g diperbarui dari ∞ menjadi 7.



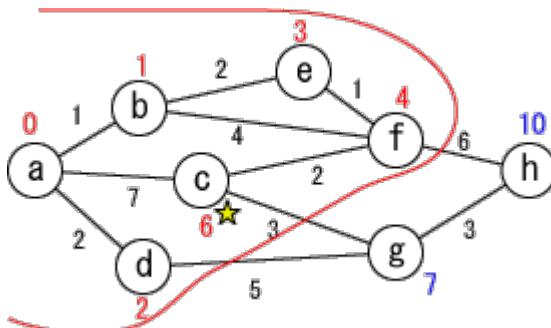
Berikutnya, node-e akan dipilih karena ia memiliki jarak paling pendek (3) dibandingkan dengan node-node yang “belum dipilih” lainnya. Jarak ini kemudian diset sebagai jarak terpendek yang ditemukan sejauh ini. Node-e yang baru dipilih ditandai dengan bintang.



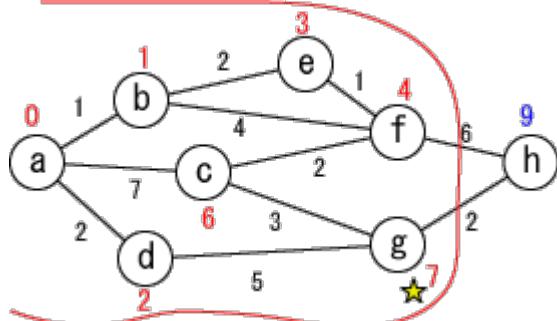
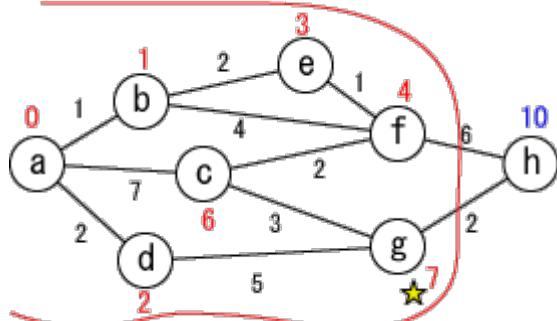
Jarak dari node-a ke node yang terhubung langsung dengan node-e tetapi statusnya masih "belum dipilih", akan dihitung dan diperbarui jika nilainya lebih kecil dari nilai jarak sebelumnya. Node tersebut yaitu node-f. Jarak dari node-a ke node-f diperbarui dari 5 menjadi 4. Proses ini terus dilanjutkan hingga seluruh node dipilih. Ilustrasi gambarnya adalah sebagai berikut.



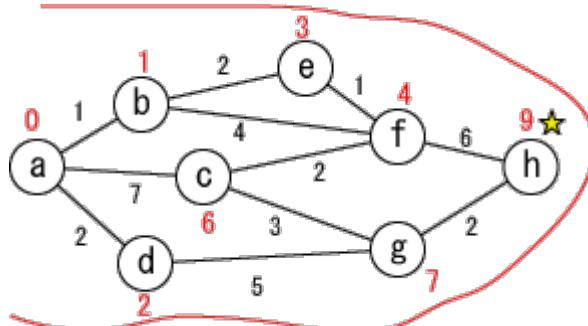
Node-f dipilih dan nilai node-(c, h) diperbarui.



Node-c dipilih namun tidak ada nilai node lain yang diperbarui.



Node-g dipilih dan nilai node-h diperbarui.



Seluruh node selesai dipilih. Jarak terpendek dari node-a ke node-h didapatkan bernilai 9.

7.2.2 Implementasi Java

Implementasi algoritma Dijkstra untuk menghitung jarak terpendek diberikan sebagai berikut. Ada dua class yang digunakan pada kode di bawah: Graph dan GraphTest. Graph digunakan untuk representasi graf dan implementasi algoritma Dijkstra, sementara GraphTest digunakan untuk testing.

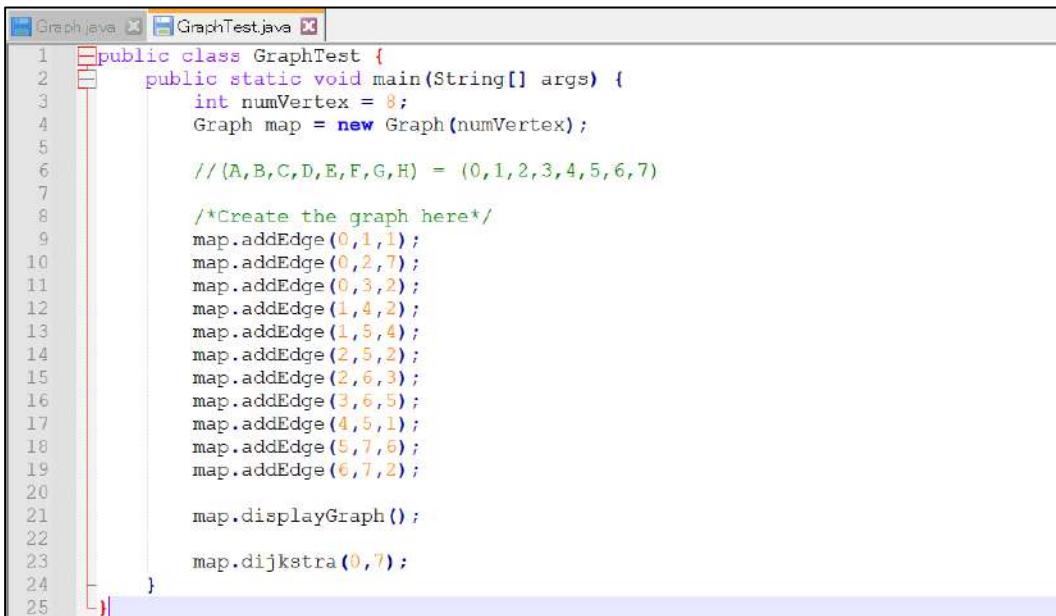
Graph.java

```
Graph.java X GraphTest.java X
1  public class Graph {
2      private int numVertex;
3      private int[][] adjacencyMatrix;
4
5      public Graph(int numVertex) {
6          this.numVertex = numVertex;
7          adjacencyMatrix = new int[numVertex][numVertex];
8      }
9
10     public void addEdge(int from, int to, int len) {
11         adjacencyMatrix[from][to] = len;
12         adjacencyMatrix[to][from] = len;
13     }
14
15     public void displayGraph() {
16         for (int i=0 ; i<numVertex ; i++) {
17             for (int j=0 ; j<numVertex ; j++) {
18                 System.out.print(adjacencyMatrix[i][j] + " ");
19             }
20             System.out.println();
21         }
22     }
23 }
```

```
Graph.java X GraphTest.java X
24  public void dijkstra(int src, int dst) {
25      int[] distance = new int[numVertex];
26      boolean[] fixed = new boolean[numVertex];
27      for (int i=0; i<numVertex; i++) {
28          distance[i] = Integer.MAX_VALUE;
29          fixed[i] = false;
30      }
31      distance[src] = 0;
32      while (true) {
33          int marked = minIndex(distance, fixed);
34          if (marked < 0) break;
35          if (distance[marked]==Integer.MAX_VALUE) break;
36          fixed[marked] = true;
37          for (int j=0; j<numVertex; j++) {
38              if (adjacencyMatrix[marked][j]>0 && !fixed[j]) {
39
40                  int newDistance = distance[marked]+adjacencyMatrix[marked][j];
41
42                  if (newDistance < distance[j]) distance[j] = newDistance;
43              }
44          }
45          if (distance[dst]==Integer.MAX_VALUE) {
46              System.out.println("no route");
47          } else {
48              System.out.println("distance="+distance[dst]);
49          }
50      }
51  }
52 }
```

```
Graph.java X GraphTest.java X
53  public int minIndex(int[] distance, boolean[] fixed) {
54      int idx=0;
55      for (; idx<fixed.length; idx++)
56          if (!fixed[idx]) break;
57      if (idx == fixed.length) return -1;
58      for (int i=idx+1; i<fixed.length; i++)
59          if (!fixed[i] && distance[i]<distance[idx]) idx=i;
60      return idx;
61  }
62 }
```

GraphTest.java



```
1 public class GraphTest {
2     public static void main(String[] args) {
3         int numVertex = 8;
4         Graph map = new Graph(numVertex);
5
6         // {A,B,C,D,E,F,G,H} = {0,1,2,3,4,5,6,7}
7
8         /*Create the graph here*/
9         map.addEdge(0,1,1);
10        map.addEdge(0,2,7);
11        map.addEdge(0,3,2);
12        map.addEdge(1,4,2);
13        map.addEdge(1,5,4);
14        map.addEdge(2,5,2);
15        map.addEdge(2,6,3);
16        map.addEdge(3,6,5);
17        map.addEdge(4,5,1);
18        map.addEdge(5,7,6);
19        map.addEdge(6,7,2);
20
21        map.displayGraph();
22
23        map.dijkstra(0,7);
24    }
}
```

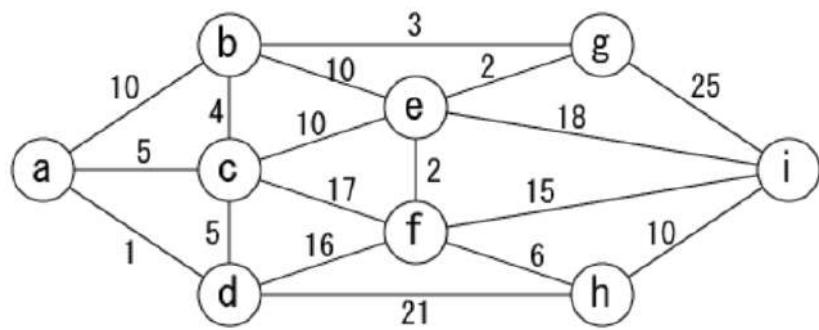
Hasil yang didapatkan setelah menjalankan GraphTest.java adalah sebagai berikut.

```
PS C:\$Users\$alfam\$java> javac Graph.java
PS C:\$Users\$alfam\$java> javac GraphTest.java
PS C:\$Users\$alfam\$java> java GraphTest
0 1 7 2 0 0 0 0
1 0 0 0 2 4 0 0
7 0 0 0 0 2 3 0
2 0 0 0 0 0 5 0
0 2 0 0 0 1 0 0
0 4 2 0 1 0 0 6
0 0 3 5 0 0 0 2
0 0 0 0 0 6 2 0
distance=9
```

Perhatikan bahwa pada implementasi kali ini, representasi graf yang digunakan adalah dalam bentuk adjacency matrix.

7.3 Tugas

1. Hitung jarak terpendek dari node-a ke node-i untuk graf berikut. Hitung secara manual (menggunakan pensil/pena dan kertas) dan cocokkan hasilnya dengan implementasi menggunakan kode di atas. Apakah hasilnya sama?



2. Pahami kode di atas dan modifikasi kode tersebut sehingga path yang dilalui untuk mendapatkan jarak terpendek juga diprint ke layar. Misal, path dengan jarak terpendek dari node-a ke node-h pada graf di 7.2.1 adalah melalui A-D-G-H. Tunjukkan bagian mana dari kode di atas yang anda modifikasi dan bagaimana anda memodifikasinya.

BAB VIII

MINIMUM SPANNING TREE DENGAN ALGORITMA PRIM

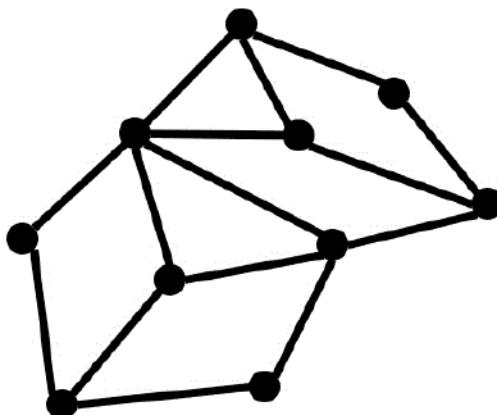
8.1 Tujuan Pembelajaran

1. Mahasiswa mampu mengimplementasikan algoritma Prim untuk menyelesaikan permasalahan minimum spanning tree (MST).

8.2 Materi

8.2.1 Minimum Spanning Tree Problem

Spanning tree adalah graf yang memuat semua vertex dari graf asli dan memiliki edge-edge tertentu sehingga membentuk sebuah tree. Perhatikan bahwa, berbeda dari graf, tree tidak memiliki cycle.



Misalnya, Anda dapat membuat tree yang melewati semua vertex graf di atas dengan memilih edge-edge yang diwarnai ungu seperti di bawah ini. Kumpulan edge berwarna ungu ini adalah salah satu dari spanning tree graf di atas.

Pada graf berbobot, spanning tree dengan total bobot terkecil disebut minimum spanning tree (MST). Permasalahan menentukan MST ini dapat diselesaikan dengan menggunakan dua algoritma yang terkenal, yaitu algoritma Kruskal dan Prim. Pada praktikum kali ini, kita akan fokus pada penyelesaian MST menggunakan algoritma Prim.

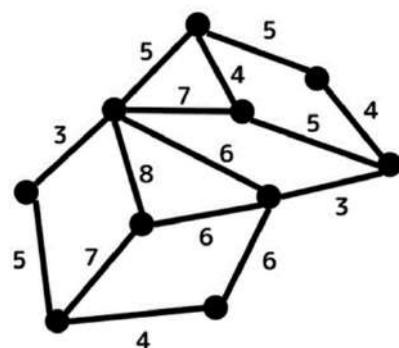
8.2.2 Algoritma Prim

Algoritma Prim bekerja seperti berikut

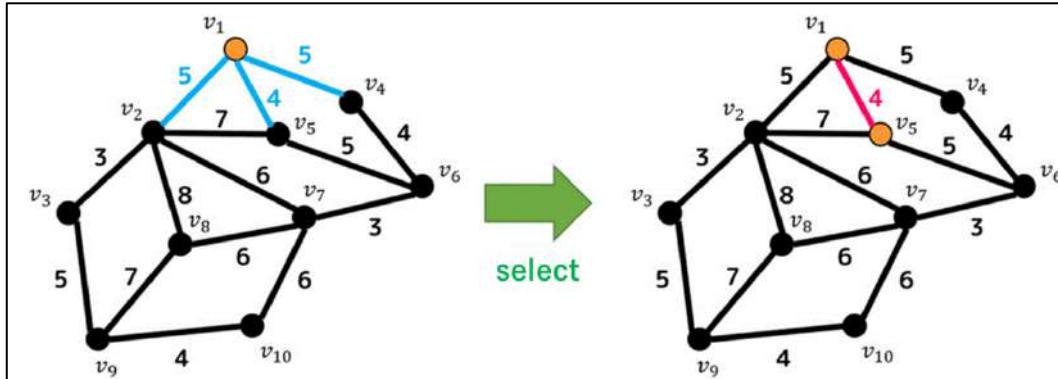
1. Pilih dan tandai vertex awal (vertex manapun OK)
2. Pilih edge dengan bobot terkecil di antara edge-edge yang terhubung ke vertex awal ini (jika ada beberapa edge dengan bobot terkecil, Anda dapat memilih salah satunya) dan tandai vertex baru yang berada diujung berlawanan edge tersebut.
3. Pilih edge baru dengan bobot terkecil dari edge-edge yang terhubung ke vertex-vertex yang telah ditandai DAN edge baru ini tidak boleh membentuk cycle saat dipilih.
4. Ulangi step 3 hingga tidak ada lagi edge yang bisa dipilih

Mari kita aplikasikan algoritma di atas untuk menentukan MST sebuah graf.

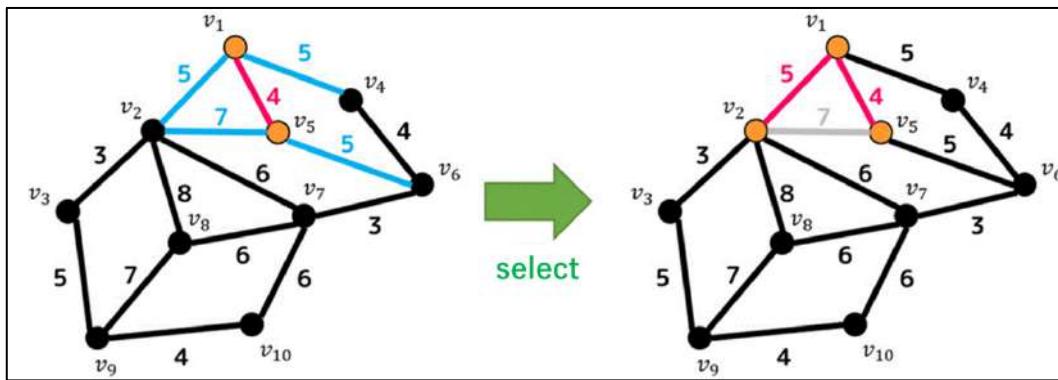
Pertama-tama, asumsikan sebuah graf berbobot seperti berikut:



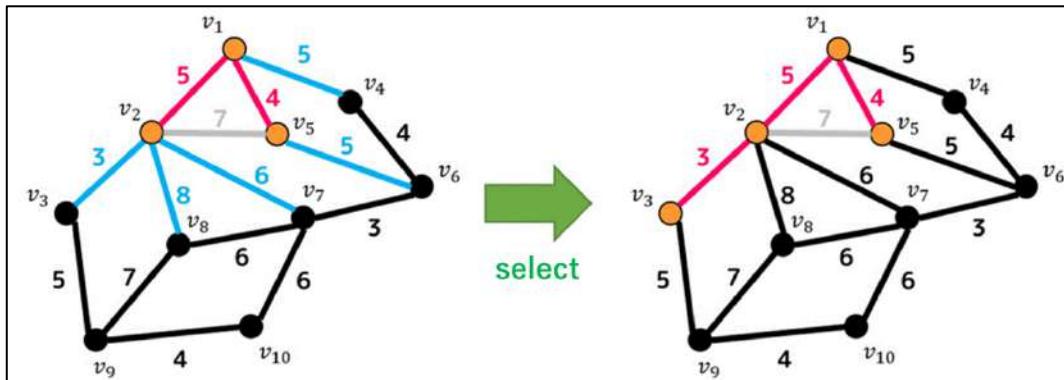
Untuk meminimalisir kesalahan saat menerapkan algoritma Prim, mari kita ingat-ingat dari awal hal berikut: "karena jumlah vertex ada sepuluh, maka kita perlu memilih sembilan edge!". Kita beri nama vertex awal sebagai v1 (vertex teratas pada graf tersebut), dan vertex lainnya diberi nama sesuai dengan gambar di bawah ini. Selanjutnya, pilih edge dengan bobot terkecil di antara edge-edge yang terhubung ke v1 (edge-edge berwarna biru pada gambar di bawah). Karena hanya ada satu edge dengan bobot terkecil, yakni 4, pilih edge ini.



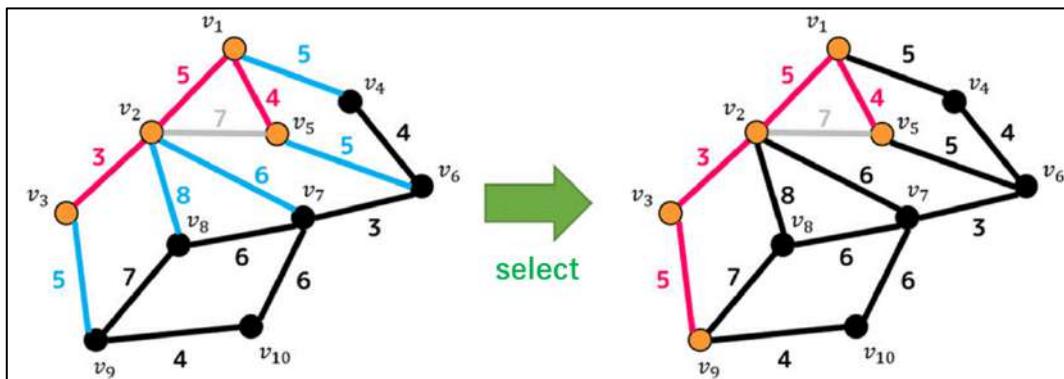
Edge yang telah dipilih ditampilkan dalam warna merah. Edge ini menghubungkan dua vertex, v1 dan v5, dan oleh karena itu, edge-edge yang melekat pada dua vertex ini akan menjadi kandidat untuk pemilihan edge berikutnya. Ada tiga edge dengan bobot terkecil 5 di antara semua kandidat edge, jadi pilih apa pun yang Anda suka di antara ketiga edge ini. Kali ini, kita pilih edge yang menghubungkan v1 dan v2, yaitu yang paling kiri. Dari sini, jika kita memilih edge yang menghubungkan v2 dan v5, maka akan terbentuk sebuah cycle, maka kita harus menghilangkan edge ini dari kandidat edge (edge berwarna abu-abu pada gambar di bawah).



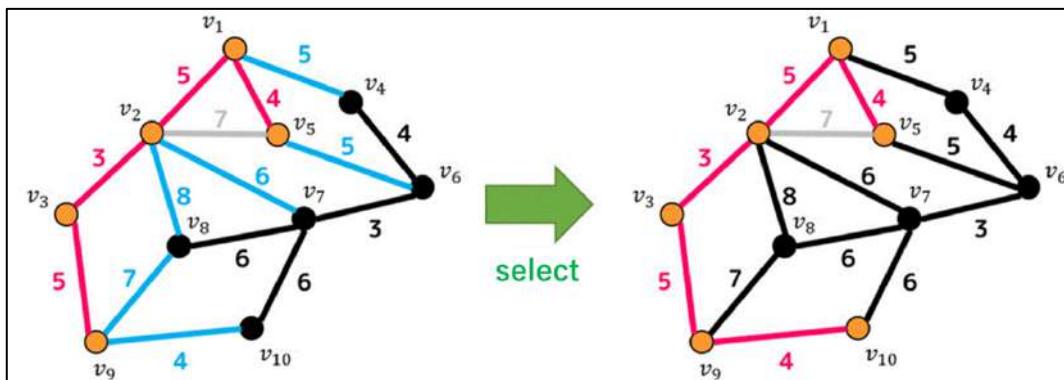
Sekarang sudah ada tiga vertex yang dihubungkan oleh edge-edge yang dipilih, yaitu v1, v2, dan v5. Berikutnya, edge-edge yang melekat pada ketiga vertex tersebut akan menjadi kandidat untuk pemilihan edge berikutnya. Di antara kandidat ini, edge dengan bobot terkecil adalah edge dengan bobot 3. Jadi kita pilih edge ini.



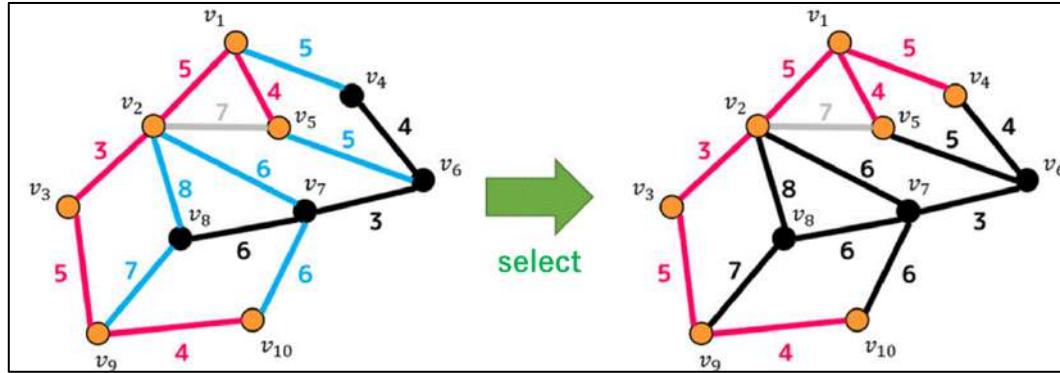
Sekarang ada empat vertex yang terhubung oleh edge-edge yang dipilih, yaitu v_1 , v_2 , v_3 , dan v_5 . Selanjutnya, edge-edge yang melekat pada keempat vertex ini akan menjadi kandidat untuk pemilihan edge berikutnya. Di antara kandidat ini, ada tiga edge dengan bobot terkecil, yakni 5. Jadi, kita pilih salah satunya.



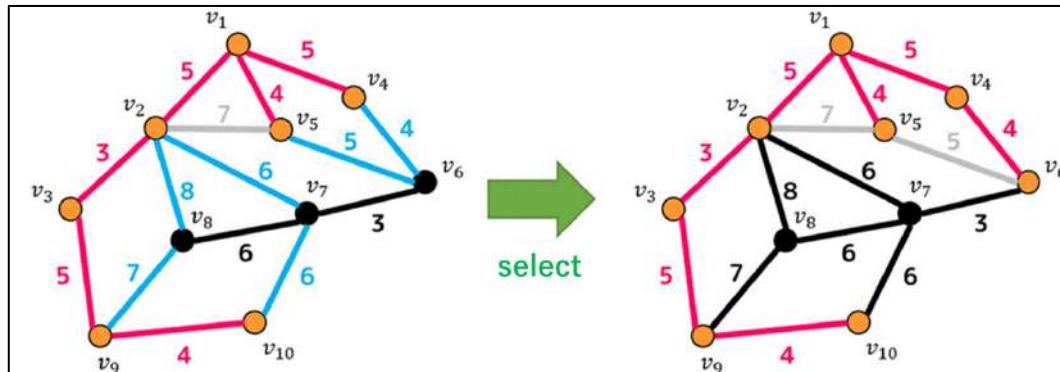
Sekarang ada lima vertex yang terhubung oleh edge-edge yang dipilih, yaitu v_1 , v_2 , v_3 , v_5 , dan v_9 . Sama seperti sebelumnya, edge yang melekat pada kelima vertex ini akan menjadi kandidat untuk pemilihan edge berikutnya. Di antara kandidat ini, edge dengan bobot terkecil adalah edge berbobot 4. Jadi kita pilih edge ini.



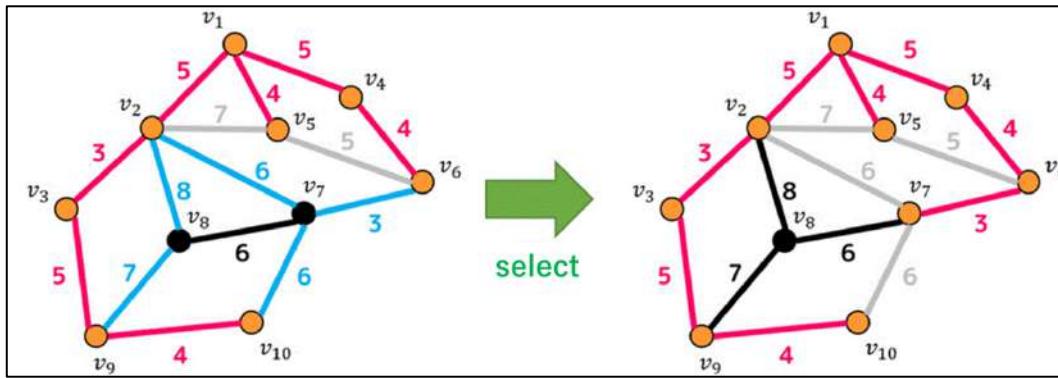
Sekarang ada enam vertex yang dihubungkan oleh edge-edge yang dipilih. Di antara kandidat edge, ada dua edge dengan bobot terkecil, yakni 5. Jadi, kita pilih salah satu dari dua edge ini.



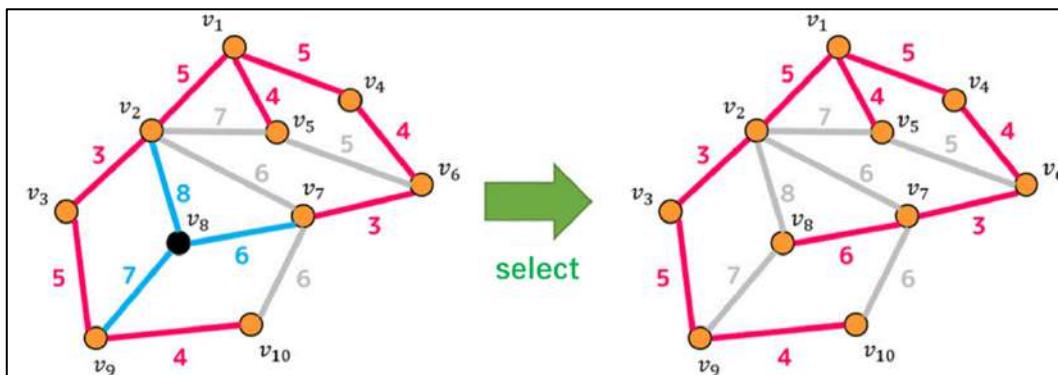
Sekarang ada tujuh vertex yang dihubungkan oleh edge-edge yang dipilih. Di antara kandidat edge, edge dengan bobot terkecil adalah edge berbobot 4. Jadi, kita pilih edge ini. Kemudian, edge yang menghubungkan v5 dan v6 akan membentuk cycle jika dipilih, maka kita harus menghilangkan edge ini dari kandidat edge.



Sekarang ada delapan vertex yang dihubungkan oleh edge-edge yang dipilih. Di antara kandidat edge, edge dengan bobot terkecil adalah edge berbobot 3. Jadi, kita pilih edge ini. Selanjutnya, dua edge (yang menghubungkan v2-v7 dan v7-v10) akan membentuk sebuah cycle jika dipilih, sehingga dua edge ini dihilangkan dari kandidat edge.



Dari jumlah edge berwarna merah, jelas bahwa sejauh ini kita telah memilih total delapan edge, dan ingat bahwa kita hanya perlu memilih satu edge lagi sehingga total tepi edge dipilih menjadi sembilan. Dari kandidat yang tersisa, kita pilih satu dengan bobot terendah, yaitu 6. Dengan demikian, bentuk akhir dari MST graf ini adalah seperti yang ditunjukkan di bawah ini.



Memilih edge yang tersisa akan membentuk cycle, maka kita tidak dapat memilih edge lain lagi dan ini menandai akhir dari algoritma Prim. Di sini kita memperoleh MST dengan total bobot 39.

8.2.3 Implementasi Java untuk Algoritma Prim

Pada implementasi ini, kita mengasumsikan bahwa graf direpresentasikan menggunakan adjacency matrix. Kali ini kita akan menggunakan jumlah vertex yang sudah ditandai untuk menentukan waktu terminasi algoritma Prim. Kita akan melacak vertex mana saja yang sudah ditandai. Untuk tujuan ini, kita akan menggunakan array yang disebut `mstSet` untuk memasukkan vertex-vertex yang sudah ditandai. Setiap vertex pada graf akan diberi nilai `key`. Nilai `key` ini mewakili bobot terkecil edge-edge sebuah vertex yang belum dimuat di `mstSet` dan “tepat satu ujungnya terhubung ke

vertex di mstSet". Perhatikan bahwa, kita mengharuskan kondisi "tepat satu ujungnya terhubung ke vertex di mstSet" untuk menghindari terbentuknya cycle. Karena pada awalnya mstSet kosong, nilai awal key setiap vertex diset ke INFINITE. Vertex dengan nilai key terkecil akan dimasukkan ke mstSet, dengan demikian vertex awal harus diset nilai key-nya menjadi 0 untuk dimasukkan terlebih dahulu ke mstSet. Kemudian, nilai-nilai key ini akan diperbarui selama eksekusi program dan program berakhir ketika semua vertex telah dimasukkan ke mstSet. Program ini bekerja sebagai berikut:

while mstSet does not include all vertices, do:

1. Pick a vertex u which is not in the mstSet and has minimum key value
2. Include u to mstSet
3. Update key values of all adjacent vertices of u

Untuk memperbarui nilai key, kita akan meng-iterasi seluruh vertex yang terhubung dengan u , untuk setiap vertax v yang terhubung dengan u , jika bobot edge $u-v$ lebih kecil dari nilai key v sebelumnya, nilai key tersebut akan diperbarui menjadi bobot $u-v$.

minKey and printMST methods:

```

1 // A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
2 // The program is for adjacency matrix representation of the graph
3
4 import java.util.*;
5 import java.lang.*;
6 import java.io.*;
7
8 class MST {
9     // Number of vertices in the graph
10    private static final int V = 5;
11
12    // A utility function to find the vertex with minimum key
13    // value, from the set of vertices not yet included in MST
14    int minKey(int key[], Boolean mstSet[])
15    {
16        // Initialize min value
17        int min = Integer.MAX_VALUE, min_index = -1;
18
19        for (int v = 0; v < V; v++)
20        {
21            if (mstSet[v] == false && key[v] < min)
22            {
23                min = key[v];
24                min_index = v;
25            }
26        }
27
28        return min_index;
29    }
30
31    // A utility function to print the constructed MST stored in
32    // parent[]
33    void printMST(int parent[], int graph[][], int V)
34    {
35        System.out.println("Edge \tWeight");
36        for (int i = 1; i < V; i++)
37        {
38            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
39        }
40    }
41
42 }

```

primMST method:

The image shows two code snippets from a Java file named MST.java. The first snippet (lines 37-49) initializes arrays for the Minimum Spanning Tree (MST) construction. The second snippet (lines 51-60) initializes keys and the MST set.

```
// Function to construct and print MST for a graph represented
// using adjacency matrix representation
void primMST(int graph[][])
{
    // Array to store constructed MST
    int parent[] = new int[V];

    // Key values used to pick minimum weight edge in cut
    int key[] = new int[V];

    // To represent set of vertices included in MST
    Boolean mstSet[] = new Boolean[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is
    // picked as first vertex
    parent[0] = -1; // First node is always root of MST
}
```

The third snippet (lines 61-87) implements the main logic of the Prim's algorithm, which iteratively picks the minimum key vertex and updates the MST set and key values for its adjacent vertices.

```
// The MST will have V vertices
for (int count = 0; count < V - 1; count++) {
    // Pick thd minimum key vertex from the set of vertices
    // not yet included in MST
    int u = minKey(key, mstSet);

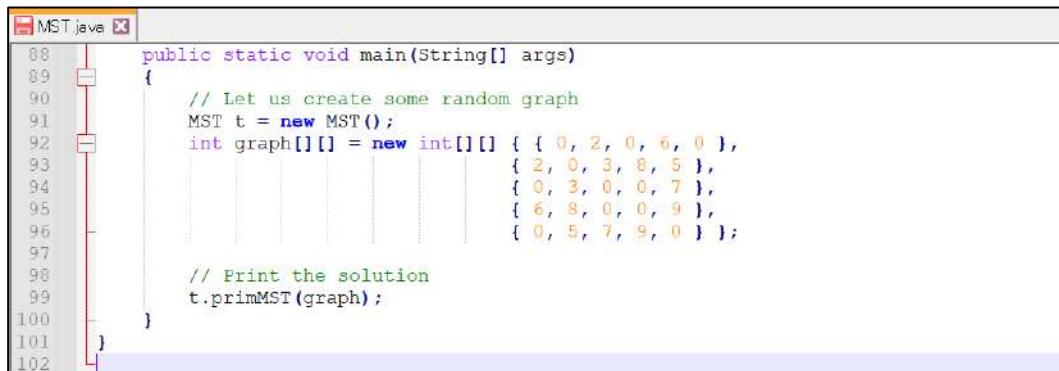
    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of the adjacent
    // vertices of the picked vertex. Consider only those
    // vertices which are not yet included in MST
    for (int v = 0; v < V; v++) {

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
        }
    }

    // print the constructed MST
    printMST(parent, graph);
}
```

main method:



```
MST.java
88  public static void main(String[] args)
89  {
90      // Let us create some random graph
91      MST t = new MST();
92      int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
93                                  { 2, 0, 3, 8, 5 },
94                                  { 0, 3, 0, 0, 7 },
95                                  { 6, 8, 0, 0, 9 },
96                                  { 0, 5, 7, 9, 0 } };
97
98      // Print the solution
99      t.primMST(graph);
100 }
101
102 }
```

8.3 Tugas

1. Uji program di atas dengan graf dari 8.2.2 dan modifikasi program tersebut sedemikian rupa sehingga total cost MST juga diprint. Apakah Anda mendapatkan hasil yang sama dibandingkan dengan hasil perhitungan manual (teori)?
2. Spanning tree lain (bukan minimum) dapat dibuat dari sebuah MST dengan mengubah koneksi edge-nya. Buatlah program yang menghitung jumlah spanning tree yang dapat dibuat dengan mengubah posisi satu edge saja. Kemudian, di antara spanning tree-spanning tree tersebut, output-kan cost spanning tree yang paling kecil. Uji kode Anda pada graf dari 8.2.2. Periksa kebenarannya kode Anda dengan membandingkan hasilnya dengan hasil yang didapatkan melalui perhitungan manual.

BAB IX

DISJOINT SET

9.1 Tujuan Pembelajaran

1. Mahasiswa mampu memahami struktur data disjoint set.
2. Mahasiswa mampu menyelesaikan masalah dengan menggunakan struktur data disjoint set.

9.2 Materi

9.2.1 Disjoint Set

Mari kita mulai dengan masalah berikut: Terdapat beberapa item. Kita diperbolehkan untuk menggabungkan dua item yang dianggap sama. Sewaktu-waktu, kita dapat bertanya apakah dua item dianggap sama atau tidak. Struktur data yang cocok untuk menyelesaikan masalah tersebut secara efektif adalah disjoint set.

Disjoint-set adalah struktur data yang menyimpan sekumpulan elemen yang dipartisi menjadi beberapa subset yang saling asing. Dengan kata lain, disjoint set adalah kumpulan himpunan yang mana tidak ada item yang dapat berada di lebih dari satu himpunan. Struktur data juga sering disebut struktur data **union-find**, karena mendukung operasi union dan find pada himpunan bagian. Mari kita mulai dengan mendefinisikan beberapa method:

- **Find:** Method ini menentukan suatu elemen masuk ke dalam subset yang mana, dan mengembalikan representatif dari subset tertentu. Sebuah item dari subset ini biasanya bertindak sebagai representatif dari set.
- **Union:** Method ini menggabungkan dua subset yang berbeda ke dalam satu subset, dan representatif dari satu subset menjadi representatif dari subset yang lain.

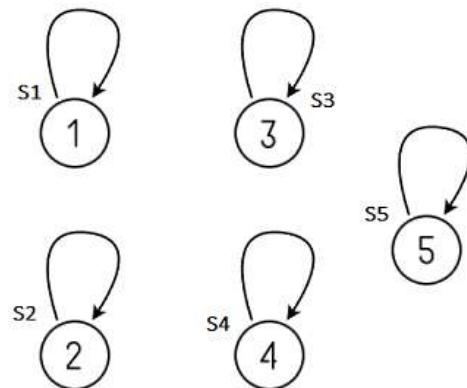
Kita dapat menentukan apakah dua elemen berada dalam subset yang sama dengan membandingkan hasil dari dua operasi Find. Jika dua elemen berada dalam subset yang sama, mereka akan memiliki representatif yang sama; namun jika representatifnya berbeda, mereka bukan merupakan anggota subset yang sama. Jika method union dipanggil pada dua elemen berbeda, gabungkan dua subset berbeda yang memiliki kedua elemen tersebut.

9.2.2 Implementasi

Disjoint-set adalah struktur data yang mana setiap set diwakili oleh tree. Setiap node pada tree mempunyai informasi terkait parent. Representatif dari setiap set adalah root dari tree pada set tersebut.

- **Find:** Ikuti parent node hingga mencapai root.
- **Union:** menggabungkan dua tree menjadi satu dengan menempelkan root dari satu tree ke root dari tree yang lain.

Sebagai contoh, perhatikan lima disjoint set S1, S2, S3, S4, dan S5 yang diwakili oleh sebuah tree, seperti yang ditunjukkan gambar di bawah ini. Setiap set awalnya hanya berisi satu elemen, dan pointer yang mengarah ke parent menunjuk ke dirinya sendiri.

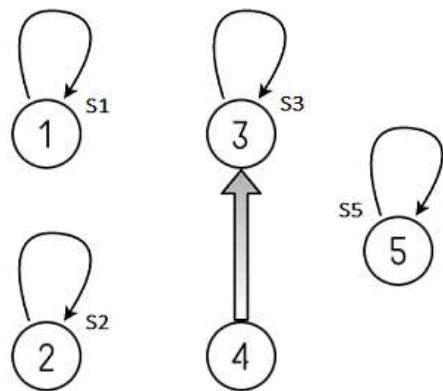


$$S1 = \{1\}, S2 = \{2\}, S3 = \{3\}, S4 = \{4\} \text{ and } S5 = \{5\}.$$

Operasi **Find** pada elemen i akan mengembalikan representatif dari Si, dengan $1 \leq i \leq 5$, seperti $\text{Find}(i) = i$.

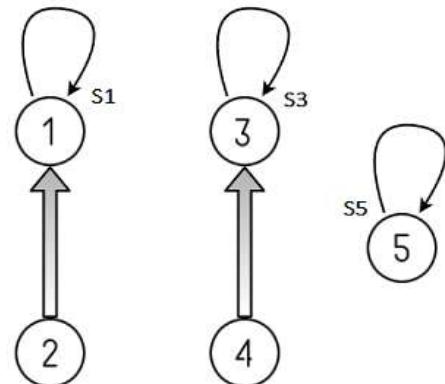
Jika kita melakukan operasi $\text{Union}(S3, S4)$, maka S3 dan S4 akan digabungkan menjadi satu disjoint set, S3. Maka, sekarang kita memiliki $S1 = \{1\}$, $S2 = \{2\}$, $S3 = \{3, 4\}$ dan $S5 = \{5\}$.

Operasi $\text{Find}(4)$ akan mengembalikan representatif dari set S3, yaitu $\text{Find}(4) = 3$.

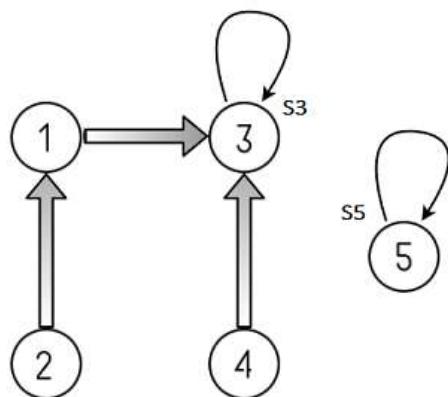


Jika kita melakukan operasi Union(S_1, S_2), maka S_1 dan S_2 akan digabungkan menjadi satu disjoint set, S_1 . Maka, sekarang kita memiliki $S_1 = \{1, 2\}$, $S_3 = \{3, 4\}$ dan $S_5 = \{5\}$.

Operasi Find(2) atau Find(1) akan mengembalikan representatif dari set S_1 , yaitu $\text{Find}(2) = \text{Find}(1) = 1$.



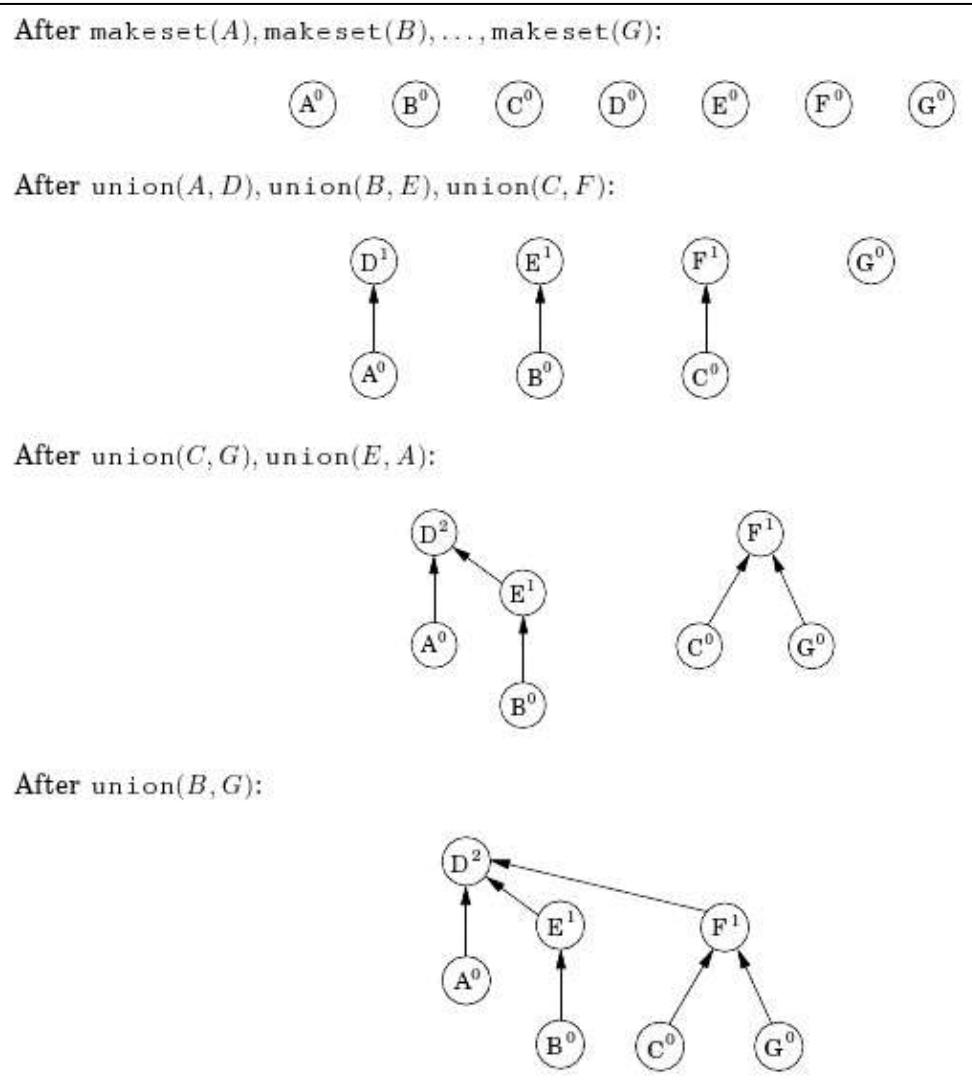
Jika kita melakukan operasi Union(S_3, S_1), maka S_3 dan S_1 akan digabungkan menjadi satu disjoint set, S_3 . Maka, sekarang kita memiliki $S_3 = \{1, 2, 3, 4\}$, dan $S_5 = \{5\}$.



Implementasi yang standar dapat menghasilkan tree yang sangat tidak seimbang. Namun, terdapat dua langkah untuk mengoptimasi:

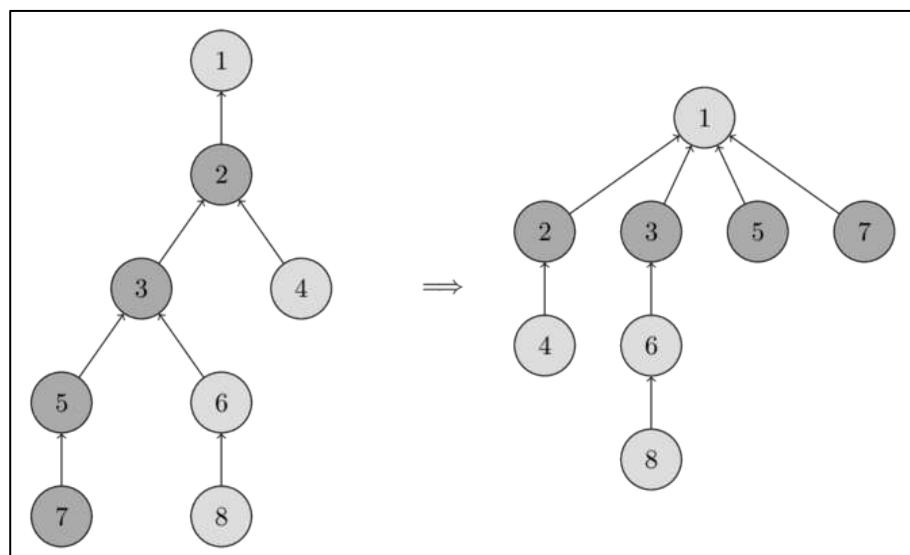
1. Union by rank

Proses ini adalah menempelkan tree yang lebih kecil ke root dari tree yang lebih besar. Karena kedalaman tree mempengaruhi running time, tree dengan kedalaman yang lebih kecil akan ditambahkan di bawah root dari tree yang lebih dalam, yang akan menambahkan kedalaman tree baru hanya jika kedua tree memiliki kedalaman yang sama. Sebuah tree yang memiliki hanya satu elemen didefinisikan memiliki rank nol, dan setiap kali dua tree dengan rank r yang sama digabungkan, tree yang terbentuk memiliki rank $r+1$. Running time pada kasus terburuk akan menjadi $O(\log(n))$ untuk operasi Union atau Find, dan lebih optimal. Berikut ini adalah ilustrasi dari union-by-rank.



2. Path compression

Proses ini adalah membuat tree menjadi lebih datar ketika operasi Find dilakukan pada tree atau set tersebut. Ideanya adalah setiap node yang ada pada tree tersebut langsung disambungkan ke root, sehingga root menjadi parent dari setiap node. Implementasinya adalah ketika operasi Find dijalankan dari satu node hingga root, parent dari node yang dikunjungi akan diubah menjadi root yang ditemukan pada tree tersebut. Tree yang dihasilkan jauh lebih datar, dan mempercepat operasi tidak hanya pada elemen-elemen ini tetapi juga pada elemen yang mereferensikannya, secara langsung atau tidak langsung. Ilustrasi dari operasi path compression dapat dilihat pada gambar berikut. Di sebelah kiri terdapat tree yang asli, dan di sebelah kanan adalah tree yang lebih datar setelah memanggil Find(7), yang memperpendek jalur untuk node 7, 5, 3 dan 2.



Kedua metode ini saling melengkapi, dan running time per operasi menjadi cukup efektif, hanya dalam konstanta yang kecil.

9.2.3 Implementasi Java

Yang pertama adalah mengimplementasikan class **Set**, yang merepresentasikan sebuah set pada struktur data disjoint set.

```

1  public class Set{
2      private int parent;
3      private int rank;
4
5      public Set(int data){
6          this.parent = data;
7          this.rank = 0;
8      }
9
10     public int getParent(){
11         return this.parent;
12     }
13
14     public void setParent(int parent){
15         this.parent = parent;
16     }
17
18     public int getRank(){
19         return this.rank;
20     }
21
22     public void setRank(int rank){
23         this.rank = rank;
24     }
25 }
```

Class Set memiliki dua atribut, yaitu parent dan rank. Atribut parent adalah representatif dari sebuah set jika hanya ada satu node pada set tersebut, dan menjadi parent dari suatu node jika set tersebut memiliki lebih dari satu set, dan atribut rank untuk digunakan adalah proses union-by-rank.

Selanjutnya adalah mengimplementasikan class **DisjointSet**. Terdapat method find dan union. Method find menggunakan teknik path compression dan method union menggunakan teknik union-by-rank.

```

1  public class DisjointSet {
2      private Set[] sets;
3      private int sz;
4
5      public DisjointSet(int numItem){
6          this.sz = numItem;
7          this.sets = new Set[sz + 1];
8          for (int i=1 ; i<=this.sz ; i++){
9              this.sets[i] = new Set(i);
10         }
11     }
12
13     public int find(int item){
14         int parent = this.sets[item].getParent();
15         if (item == parent){
16             return item;
17         }
18         else{
19             parent = find(parent);
20             this.sets[item].setParent(parent); // path compression
21         }
22     }
23 }
```

```

25     public boolean isSameSet(int firstItem, int secondItem){
26         return find(firstItem) == find(secondItem);
27     }
28
29     public void union(int firstItem, int secondItem){ // union by rank
30         int firstItemParent = find(firstItem);
31         int secondItemParent = find(secondItem);
32
33         if (firstItemParent != secondItemParent){
34             int firstRank = this.sets[firstItemParent].getRank();
35             int secondRank = this.sets[secondItemParent].getRank();
36
37             if (firstRank < secondRank){
38                 this.sets[firstItemParent].setParent(secondItemParent);
39             }
39             else if (firstRank > secondRank){
40                 this.sets[secondItemParent].setParent(firstItemParent);
41             }
42             else{
43                 this.sets[secondItemParent].setParent(firstItemParent);
44                 this.sets[firstItemParent].setRank(firstRank + 1);
45             }
46         }
47     }
48 }
50
50     public void print(){
51         for (int i=1 ; i<=this.sz ; i++){
52             System.out.println("Parent of " + i + " = " + find(i));
53         }
54     }
55
56     public void printRank(){
57         for (int i=1 ; i<=this.sz ; i++){
58             System.out.println("Rank of " + i + " = " + this.sets[i].getRank());
59         }
60     }
61 }
```

Setelah mengimplementasikan class Set dan DisjointSet, saatnya mengimplementasikan class DisjointSetMain yang merupakan class main.

```

1  public class DisjointSetMain {
2      public static void main(String[] args) {
3          DisjointSet disjointSet = new DisjointSet(5);
4          disjointSet.print();
5
6          disjointSet.union(3, 4);
7          System.out.println("After union 3 and 4:");
8          disjointSet.print();
9          disjointSet.printRank();
10
11         disjointSet.union(1, 2);
12         disjointSet.union(1, 3);
13
14         System.out.println("Final result:");
15         disjointSet.print();
16         disjointSet.printRank();
17     }
18 }
```

9.3 Tugas

1. Implementasikan sebuah method untuk menghitung banyaknya set yang ada! Sebagai contoh, untuk disjoint-set dengan 5 data, maka $S_1 = \{1\}$, $S_2 = \{2\}$, $S_3 = \{3\}$, $S_4 = \{4\}$, dan $S_5 = \{5\}$, sehingga method tersebut akan mengembalikan **5**. Setelah melakukan operasi Union(S_3, S_4), maka $S_1 = \{1\}$, $S_2 = \{2\}$, $S_3 = \{3, 4\}$ dan $S_5 = \{5\}$, sehingga method tersebut akan mengembalikan **4**.
2. Implementasikan sebuah method untuk menghitung banyaknya item atau elemen dari masing-masing set! Pada contoh di Soal 1, method tersebut akan menuliskan "S1 memiliki 1 elemen, S2 memiliki 1 elemen, S3 memiliki 2 elemen, S5 memiliki 1 elemen".

BAB X

STRING MATCHING

10.1 Tujuan Pembelajaran

1. Mahasiswa mampu memahami konsep pencocokan string (*string matching*).
2. Mahasiswa mampu memahami algoritma naïve untuk pencocokan string.
3. Mahasiswa mampu memahami algoritma KMP untuk pencocokan string.
4. Mahasiswa mampu menyelesaikan masalah yang berkaitan dengan pencocokan string.

10.2 Materi

10.2.1 String Matching

Mari kita mulai dengan masalah berikut: Diberikan sebuah teks (*text*), carilah semua kemunculan dari pola (*pattern*) yang diberikan pada teks tersebut. Sebagai contoh:

1. Untuk teks $T = \text{"ABCABAABCABAC"}$ dan pola $P = \text{"ABAA"}$, pola tersebut hanya muncul satu kali, dimulai dari indeks 3 ("ABCABAABCABAC").
2. Untuk teks $T = \text{"ABCABAABCABAC"}$ dan pola $P = \text{"CAB"}$, pola tersebut muncul dua kali, yaitu dimulai dari indeks 2 dan dari indeks 8 ("ABCABAABCABAC" dan "ABCABAABCABAC").

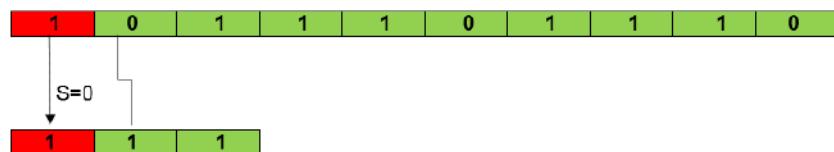
Pencocokan string adalah masalah kehidupan nyata yang sering muncul dalam program text editor seperti MS Word, notepad, notepad++, dll. Algoritma pencocokan string juga digunakan untuk mencari pola tertentu dalam urutan DNA. Tujuannya adalah untuk menemukan semua kemunculan pola $P[0\dots(m-1)]$ dengan panjang m dalam teks $T[0\dots(n-1)]$ dengan panjang n . Ada beberapa algoritma dalam pencocokan string, seperti Algoritma naïve, Algoritma Knuth-Morris-Pratt (KMP), Algoritma Z, Algoritma Rabin-Karp, dll. Pada bab ini, kita hanya akan membahas tentang Algoritma naïve dan Algoritma KMP.

10.2.2 Algoritma Naïve

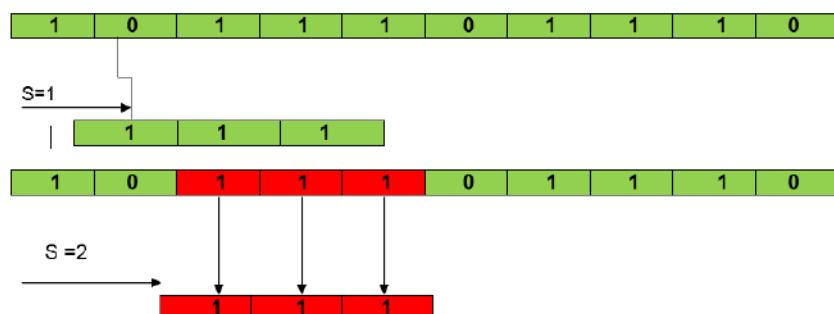
Algoritma ini mencari semua kemunculan pola dengan menggunakan perulangan yang menguji kondisi $P[0\dots(m-1)] = T[s\dots(s+m-1)]$ untuk semua $n-m+1$ macam kemungkinan dari s.

Sebagai contoh, kita memiliki $T = "1011101110"$ dan $P = "111"$. Pada mulanya, s bernilai 0, dan kita akan menguji $P[0\dots2] = T[0\dots2]$. Perhatikan bahwa $P[0] = T[0]$, namun $P[1]$ dan $T[1]$ berbeda, maka update nilai s menjadi 1. Setelah menguji untuk $s=1$, lanjutkan dengan $s=2$, seterusnya hingga $s=n-m=7$. Perhatikan ilustrasi berikut pada contoh yang diberikan.

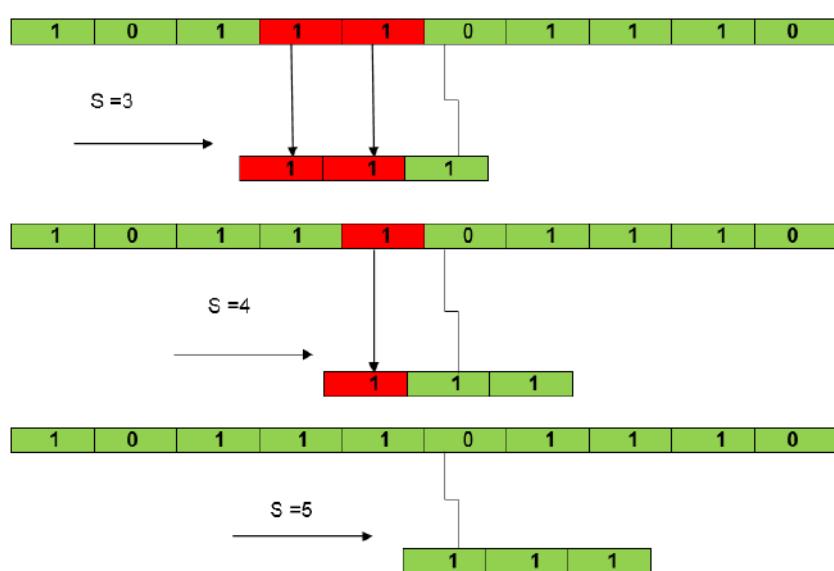
T = Text

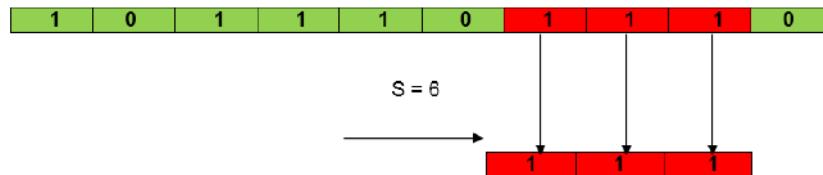


P = Pattern

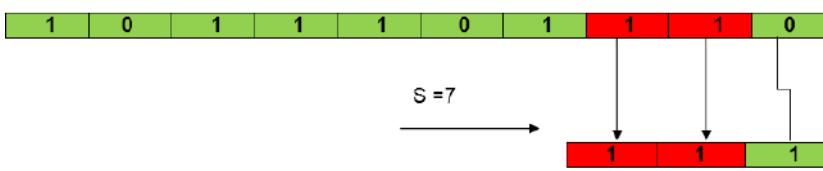


So, S=2 is a Valid Shift





So, $S=6$ is a Valid Shift



10.2.3 Algoritma KMP

Algoritma naïve tidak mengingat informasi apapun mengenai karakter-karakter sebelumnya yang telah cocok. Pada dasarnya algoritma tersebut mencocokkan karakter dengan karakter dari pola yang berbeda berulang kali. Hal ini dapat dioptimalkan dengan memanfaatkan karakter-karakter sebelumnya yang cocok.

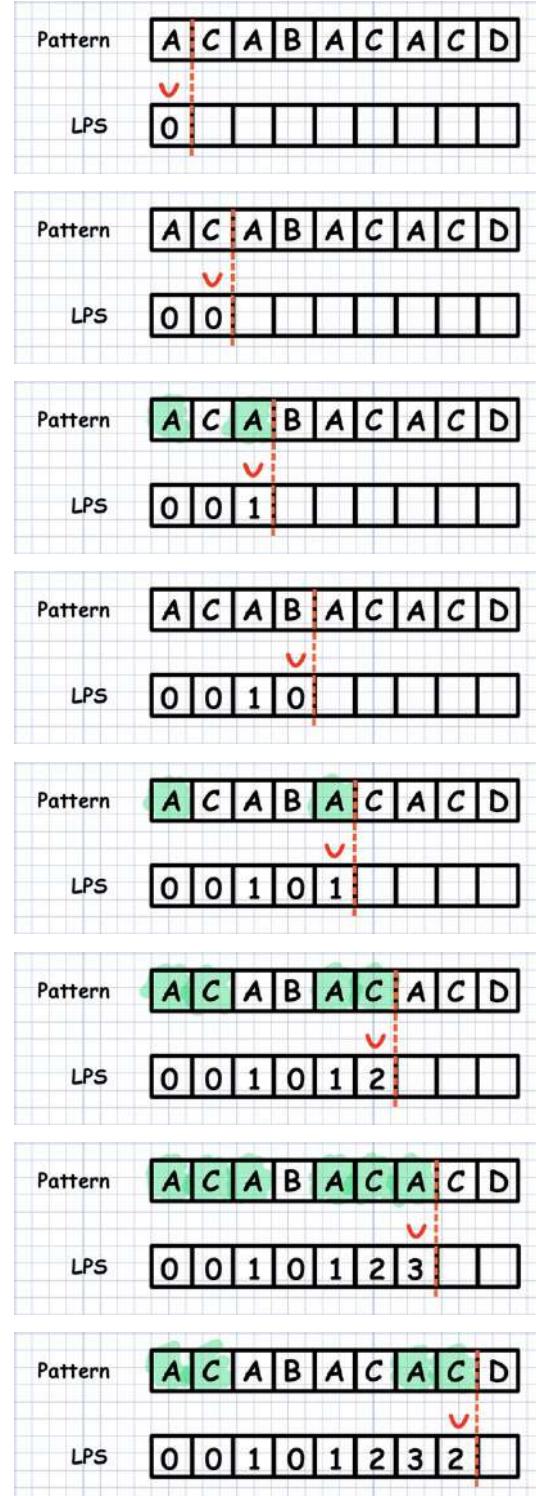
Algoritma KMP Algorithm (atau algoritma pencarian string Knuth, Morris, dan Pratt) memanfaatkan hasil perbandingan karakter-karakter sebelumnya. Algoritma ini dapat mencari pola lebih cepat karena tidak pernah membandingkan ulang karakter dari teks yang telah cocok dengan karakter dari pola. Untuk itu, dibutuhkan tabel perbandingan untuk menganalisa struktur dari pola.

Sebelum lanjut ke cara membuat tabel perbandingan, kita perlu mengetahui definisi dari **proper prefix** dan **proper suffix**:

- **Proper prefix:** String yang dihasilkan dengan menghapus satu atau lebih karakter dari belakang. “S”, “Sn”, “Sna”, dan “Snap” adalah semua proper prefix dari “Snape”.
- **Proper suffix:** String yang dihasilkan dengan menghapus satu atau lebih karakter dari depan. “agrid”, “grid”, “rid”, “id”, dan “d” adalah semua proper suffix dari “Hagrid”.

Tabel perbandingan, atau lebih dikenal dengan tabel LPS, adalah sebuah array yang menyimpan **panjang dari proper prefix terpanjang di dalam (sub)pola yang juga merupakan proper suffix di (sub)pola yang sama** untuk setiap indeks i , dari $i=0$ hingga $m-1$. Sebagai contoh, diberikan sebuah pola "ACABACACD", berikut adalah langkah-langkah untuk membuat tabel LPS dari pola tersebut. Panah merah

menunjukkan elemen LPS saat ini. Garis putus-putus menunjukkan sub-string yang diuji dari pola untuk indeks LPS saat ini. Proper prefix dan proper suffix yang sama ditandai dengan warna hijau.



Pattern	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>A</td><td>C</td><td>A</td><td>B</td><td>A</td><td>C</td><td>A</td><td>C</td><td>D</td></tr> </table>	A	C	A	B	A	C	A	C	D
A	C	A	B	A	C	A	C	D		
LPS	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>2</td><td>3</td><td>2</td><td>0</td></tr> </table>	0	0	1	0	1	2	3	2	0
0	0	1	0	1	2	3	2	0		

Berikut adalah beberapa contoh lainnya:

- Untuk pola “AAAA”, LPS[] adalah [0, 1, 2, 3]
- Untuk pola “ABCDE”, LPS[] adalah [0, 0, 0, 0, 0]
- Untuk pola “AABAACAAABAA”, LPS[] adalah [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
- Untuk pola “AACACAAAAAC”, LPS[] adalah [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]
- Untuk pola “AAABAAA”, LPS[] adalah [0, 1, 2, 0, 1, 2, 3]

Berikut adalah algoritma untuk membuat tabel LPS:

1. Set LPS[0] ke 0 dan asumsikan bahwa pola disimpan pada variabel P.
2. Lakukan iterasi dari $i=1$ hingga $i = P.length - 1$, secara umum iterasi dimulai dari karakter kedua dari P hingga karakter terakhir dari P.
3. Untuk menghitung nilai dari $LPS[i]$, buat variable j yang menyatakan panjang dari suffix terbaik dari $P[0..(i-1)]$. Pada awalnya, $j = LPS[i-1]$.
4. Cek apakah suffix dengan panjang $j+1$ juga merupakan prefix, dengan cara membandingkan $P[j]$ dan $P[i]$.
 - a. Jika $P[i] = P[j]$, maka set $LPS[i] = j+1$, karena kita berhasil menemukan prefix dengan panjang $j+1$ yang juga merupakan suffix dari $P[0..i]$.
 - b. Jika tidak, update nilai j menjadi $LPS[j-1]$ dan ulangi langkah 4a.
 - c. Jika j telah bernilai 0 setelah menjalankan langkah 4b dan masih belum menentukan karakter yang cocok, maka set $LPS[i]=0$.

Sekarang kita beralih ke Algoritma KMP. Misalkan kita memiliki teks T dan pola P. Berikut ini adalah langkah-langkah dari Algoritma KMP:

1. Buatlah string baru S dengan cara menggabungan P dan T dipisahkan dengan tanda '#'. Sebagai contoh, jika T = “ABCABAABCABAC” dan P = “ABAA”, maka S = “ABAA#ABCABAABCABAC”. Kita dapat berasumsi bahwa karakter '#' tidak muncul di T atau di P.
2. Buatlah tabel LPS dari S.
3. Lakukan iterasi dari $i=P.length+1$ hingga $i=S.length-1$.

4. Jika $LPS[i] = P.length$ untuk suatu i , artinya kita menemukan suffix dari $S[0..i]$ dengan panjang $P.length$ yang juga merupakan prefix. Sehingga, kita menemukan kemunculan P yang dimulai dari indeks $i - (P.length + 1) - P.length + 1 = i - 2*(P.length)$.

10.2.4 Implementasi Java

Kita akan membuat class **StringMatcher**, dan terdapat dua method yang public static, naive dan kmp, masing-masing merepresentasikan algoritma yang dibahas di bab ini. Terdapat juga satu method yang private static, computeLPSArray, sebagai method tambahan pada Algoritma KMP.

```

1  public class StringMatcher {
2      public static void naive(String text, String pattern){
3          int textLen = text.length();
4          int patternLen = pattern.length();
5
6          boolean found = false;
7
8          for (int i=0 ; i+patternLen <= textLen ; i++){
9              boolean current_found = true;
10             for (int j=0 ; j<patternLen ; j++){
11                 if (text.charAt(i+j) != pattern.charAt(j)){
12                     current_found = false;
13                     break;
14                 }
15             }
16             if (current_found){
17                 found = true;
18                 System.out.println("Found pattern at index " + i + " using naive.");
19             }
20         }
21         if (!found){
22             System.out.println("Pattern not found using naive.");
23         }
24     }
25
26     private static int[] computeLPSArray(String str){
27         int len = str.length();
28
29         int[] lps = new int[len];
30         lps[0] = 0;
31
32         for (int i=1 ; i<len ; i++){
33             int j = lps[i-1];
34
35             while ((j > 0) && (str.charAt(i) != str.charAt(j))){
36                 j = lps[j-1];
37             }
38             if (str.charAt(i) == str.charAt(j)){
39                 j++;
40             }
41             lps[i] = j;
42         }
43
44         return lps;
45     }

```

```

47     public static void kmp(String text, String pattern){
48         String combined = pattern + "#" + text;
49         int combinedLen = combined.length();
50         int patternLen = pattern.length();
51
52         int lps[] = computeLPSArray(combined);
53
54         boolean found = false;
55         for (int i=patternLen+1 ; i<combinedLen ; i++){
56             if (lps[i] == patternLen){
57                 found = true;
58                 System.out.println("Found pattern at index " + (i - 2*patternLen) + " using KMP.");
59             }
60         }
61
62         if (!found){
63             System.out.println("Pattern not found using KMP.");
64         }
65     }
66 }
```

Terakhir, kita membuat class **StringMatcherMain** yang mengandung method main untuk program pencocokan string (*string matching*).

```

1 import java.util.Scanner;
2
3 public class StringMatcherMain {
4     public static void main(String args[]){
5         Scanner sc = new Scanner(System.in);
6         String text, pattern;
7
8         System.out.println("Input text: ");
9         text = sc.nextLine();
10
11        System.out.println("Input pattern: ");
12        do pattern = sc.nextLine();
13        while (pattern.equals(""));
14
15        StringMatcher.naive(text, pattern);
16        StringMatcher.kmp(text, pattern);
17    }
18 }
```

10.3 Tugas

1. Diberikan sebuah teks = "aaaa...aa" (100.000 buah huruf a) dan sebuah pola = "aaa...aab" (10.000 buah huruf a dan 1 huruf b). Bandingkan running time dari Algoritma naïve dan Algoritma KMP untuk pencocokan string! Lakukan pencocokan string 10 kali, dan selanjutnya hitung rata-rata running time untuk masing-masing algoritma.
2. Diberikan sebuah teks = "aaaa...aa" (100.000 buah huruf a) dan sebuah pola = "aaa...aa" (10.000 buah huruf a). Bandingkan running time dari Algoritma naïve dan Algoritma KMP (**tanpa output apapun**) untuk pencocokan string! Lakukan pencocokan string 10 kali, dan selanjutnya hitung rata-rata running time untuk masing-masing algoritma.

3. Diberikan dua kata S dan T, dengan panjang yang sama (panjang maksimal adalah 100.000). Tugas Anda adalah menentukan apakah T dapat dibuat dengan melakukan beberapa *cycle shift* ke S (*cycle shift* adalah pemindahan karakter pertama string ke akhir string). Misalnya jika S = "erwineko" dan T = "ekoerwin", maka jawabannya haruslah "YA", karena "erwineko" -> "rwinekoe" -> "winekoer" -> "inekoerw" -> "nekoerwi" -> "ekoerwin". Dalam soal ini, Anda harus menggunakan Algoritma KMP untuk menyelesaikan soal ini (pendekatannya mungkin tidak begitu jelas, tetapi Anda harus memikirkan penggunaan Algoritma KMP dalam masalah ini).

BAB XI

ALGORITMA GEOMETRI – CONVEX HULL

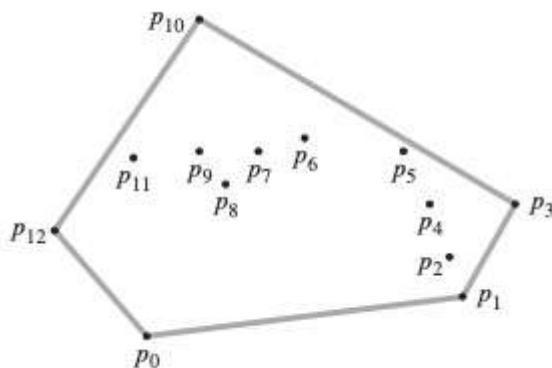
11.1 Tujuan Pembelajaran

1. Mahasiswa mampu memahami konsep convex hull.
2. Mahasiswa mampu memahami algoritma monotone chain untuk convex hull.
3. Mahasiswa mampu menyelesaikan masalah yang berkaitan dengan convex hull.

11.2 Materi

11.2.1 Convex Hull

Diberikan sekumpulan titik Q pada bidang 2-D. **Convex hull** dari Q , dilambangkan dengan $\text{CH}(Q)$, adalah **poligon convex terkecil** P yang mana setiap titik pada Q terletak pada keliling P atau di dalam P . Asumsikan bahwa semua titik di Q berbeda semua dan Q memiliki setidaknya tiga titik yang tidak kolinear (segaris). Secara intuitif, kita dapat menganggap setiap titik di Q sebagai paku yang ditaruh dari papan. Convex hull dibentuk oleh karet gelang yang mengelilingi semua paku. Gambar di bawah menunjukkan sekumpulan titik dan convex hull yang terbentuk, dengan $Q = \{p_0, p_1, \dots, p_{12}\}$ dan convex hull $\text{CH}(Q)$ berwarna abu-abu.

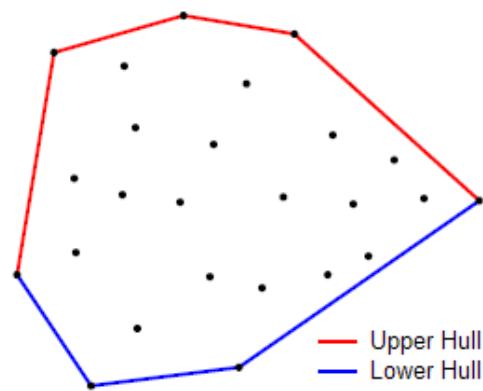


Terdapat beberapa algoritma untuk mencari convex hull, seperti Algoritma Jarvis's march, Algoritma Graham's scan, Algoritma monotone chain, dll. Pada bab ini, kita hanya akan membahas Algoritma monotone chain.

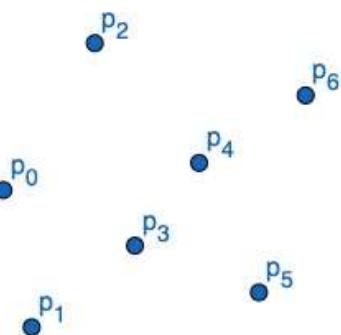
11.2.2 Algoritma Monotone Chain

Algoritma monotone chain membentuk convex hull dari sekumpulan titik pada bidang 2-dimensi. Hal pertama yang dilakukan adalah mengurutkan semua titik berdasarkan koordinat x terkecil, dan koordinat y terkecil jika nilai koordinat x sama, dan selanjutnya membangun lower hull dan upper hull.

Lower hull adalah bagian bawah dari convex hull; dari titik paling kiri ke titik paling kanan dalam urutan berlawanan arah jarum jam (counter-clockwise). Sedangkan upper hull adalah bagian lain dari convex hull selain lower hull. Pada gambar berikut, upper hull adalah bagian convex hull yang berwarna merah, sedangkan lower hull berwarna biru.

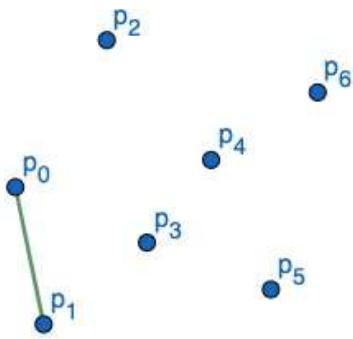


Sebagai contoh, diberikan 7 titik pada bidang 2 dimensi seperti pada gambar berikut. Perhatikan bahwa semua titik sudah diurutkan berdasarkan X terkecil, dan Y terkecil jika X sama.

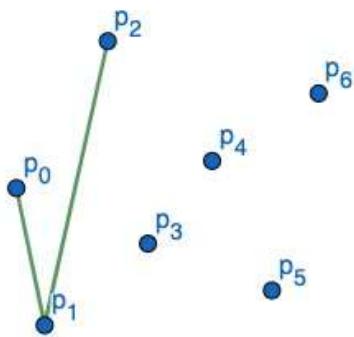


Berikut adalah langkah-langkah algoritma monotone chain untuk membangun lower hull terlebih dahulu:

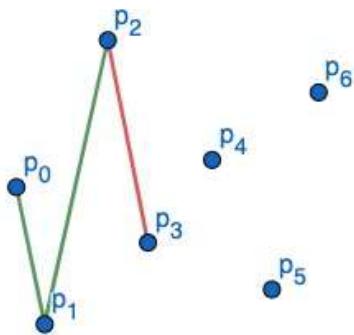
1. Masukkan dua titik paling kiri ke lower hull.

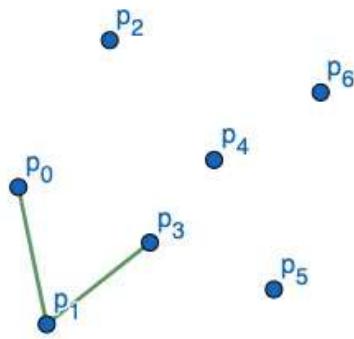


2. Cek apakah P_0, P_1 (dua titik teratas/terakhir pada lower hull), dan P_2 membentuk arah counter-clockwise. Karena iya, maka tambahkan P_2 ke dalam lower hull.

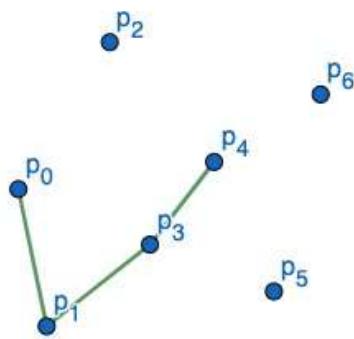


3. Cek apakah P_1, P_2 (dua titik teratas/terakhir saat ini pada lower hull), dan P_3 membentuk arah counter-clockwise. Ternyata tidak, maka hapus P_2 dari lower hull. Selanjutnya, cek apakah P_0, P_1 (dua titik teratas/terakhir saat ini pada lower hull), dan P_3 membentuk arah counter-clockwise. Karena iya, maka tambahkan P_3 ke dalam lower hull.

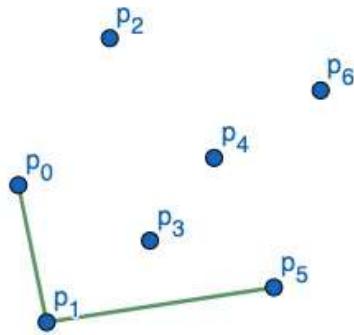




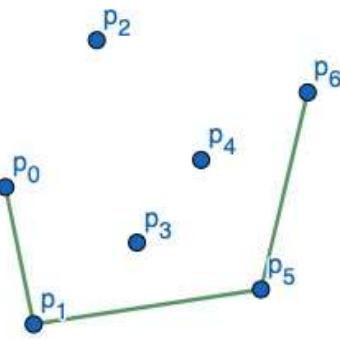
4. Cek apakah P_1 , P_3 (dua titik teratas/terakhir saat ini pada lower hull), dan P_4 membentuk arah counter-clockwise. Karena iya, maka tambahkan P_4 ke dalam lower hull.



5. Dengan cara yang sama, kedua titik P_4 dan P_3 dihapus dari lower hull, dan P_5 ditambahkan ke dalam lower hull.



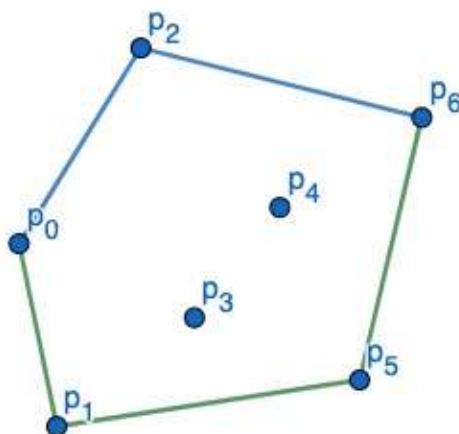
6. Selanjutnya, titik P_6 dimasukkan ke dalam lower hull.



Akhirnya, proses menyusun lower hull selesai. Lower hull memuat titik-titik $\{P_0, P_1, P_5, P_6\}$.

Menyusun upper hull dapat dilakukan dengan cara yang sama, namun prosesnya berjalan dari titik paling kanan ke titik paling kiri. Dalam contoh tadi, proses akan dilakukan dari P_6 sampai P_0 . Setelah selesai, upper hull memuat titik-titik $\{P_6, P_2, P_0\}$.

Pada gambar berikut, lower hull digambarkan dengan garis berwarna hijau, sedangkan upper hull berwarna biru.



Setelah selesai baik lower hull maupun upper hull, kemudian keduanya digabung menjadi satu convex hull namun hilangkan dulu titik terakhir masing-masing lower hull dan upper hull, karena titik terakhir lower hull sama dengan titik pertama upper hull, dan terakhir upper hull sama dengan titik pertama lower hull.

11.2.3 Implementasi Java

Kita akan membuat class **Point** yang mengimplementasikan class Comparable, dan terdapat dua atribut, *x* and *y*, keduanya memiliki tipe data double. Kita juga

melakukan override method compareTo agar bisa mengurutkan titik berdasarkan X terkecil, dan Y terkecil jika X sama.

```
1  class Point implements Comparable<Point> {
2      double x, y;
3
4      public Point(){
5          x = 0.0;
6          y = 0.0;
7      }
8
9      public Point(double _x, double _y){
10         x = _x;
11         y = _y;
12     }
13
14     public int compareTo(Point other) {
15         double EPS = 1e-9;
16         double tmp;
17
18         if (Math.abs(x - other.x) > EPS){
19             tmp = x - other.x;
20             if (tmp > EPS) return 1;
21             else return -1;
22         }
23         else if (Math.abs(y - other.y) > EPS){
24             tmp = y - other.y;
25             if (tmp > EPS) return 1;
26             else return -1;
27         }
28         else {
29             return 0;
30         }
31     }
32
33     public String toString() {
34         return "(" + x + ", " + y + ")";
35     }
36 }
37 }
```

Selanjutnya, kita membuat class **Geometry**. Terdapat tiga method yang static: **cross**, **ccw**, dan **convexHull**. Method cross membutuhkan tiga parameter, yaitu titik O, titik A, dan titik B, dan akan mengembalikan cross product dari vector OA dan OB. Method ccw membutuhkan tiga parameter juga, yaitu titik P, titik Q, dan titik R, dan akan mengembalikan true jika P-Q-R membentuk arah belok kiri (atau arah counter-clockwise).

```

1 import java.util.Arrays;
2
3 public class Geometry {
4     public static double cross(Point O, Point A, Point B) {
5         return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
6     }
7
8     // returns true if pqr turns left (counter clockwise)
9     public static boolean ccw(Point p, Point q, Point r){
10        return cross(p, q, r) > 0;
11    }
12
13    public static Point[] convexHull(Point[] P) {
14        if (P.length > 2) {
15            int n = P.length, upperLength = 0, lowerLength = 0;
16            Point[] lowerHull = new Point[n];
17            Point[] upperHull = new Point[n];
18
19            Arrays.sort(P);
20
21            // Build lower hull first
22            lowerHull[0] = P[0];
23            lowerHull[1] = P[1];
24            lowerLength = 2;
25            for (int i = 2; i < n; i++) {
26                while (lowerLength >= 2
27                      && !ccw(lowerHull[lowerLength - 2], lowerHull[lowerLength - 1], P[i])){
28                    lowerLength--;
29                }
30                lowerHull[lowerLength] = P[i];
31                lowerLength++;
32            }
33
34            // Build upper hull
35            upperHull[0] = P[n-1];
36            upperHull[1] = P[n-2];
37            upperLength = 2;
38            for (int i = n - 3; i >= 0; i--) {
39                while (upperLength >= 2
40                      && !ccw(upperHull[upperLength - 2], upperHull[upperLength - 1], P[i])){
41                    upperLength--;
42                }
43                upperHull[upperLength] = P[i];
44                upperLength++;
45            }
46
47            // Combine lower hull and upper hull
48            Point[] result = new Point[2 * n];
49            int t = 0;
50            for (int i=0 ; i<lowerLength - 1 ; i++){
51                result[t] = lowerHull[i];
52                t++;
53            }
54            for (int i=0 ; i<upperLength - 1 ; i++){
55                result[t] = upperHull[i];
56                t++;
57            }
58
59            if (t > 1) {
60                result = Arrays.copyOfRange(result, 0, t);
61            }
62            return result;
63        } else if (P.length == 2) {
64            return P.clone();
65        } else {
66            return null;
67        }
68    }
69}

```

Terakhir, class **GeometryMain** memuat fungsi main.

```
1 public class GeometryMain{
2     public static void main(String[] args) {
3         Point[] points = new Point[7];
4         points[0] = new Point(3.6, 4.5);
5         points[1] = new Point(0, 4);
6         points[2] = new Point(1.75, 6.75);
7         points[3] = new Point(2.4, 3);
8         points[4] = new Point(5.6, 5.8);
9         points[5] = new Point(0.5, 1.5);
10        points[6] = new Point(4.75, 2.1);
11
12        Point[] hull = Geometry.convexHull(points);
13
14        System.out.println("CONVEX HULL:");
15        for (int i = 0; i < hull.length; i++) {
16            if (hull[i] != null)
17                System.out.println(hull[i]);
18        }
19    }
20}
```

11.3 Tugas

1. Tambahkan method **getConvexHullArea** pada class Geometry yang membutuhkan input convex hull dari sekumpulan titik S, yaitu CH(S), dan akan mengembalikan suatu nilai yang merupakan luas dari wilayah convex hull yang terbentuk.
2. Tambahkan method **getConvexHullLength** pada class Geometry yang membutuhkan input convex hull dari sekumpulan titik S, yaitu CH(S), dan akan mengembalikan suatu nilai yang merupakan keliling dari wilayah convex hull yang terbentuk.