TUGAS MINGGU 5

Binary Tree dan Graph BFS + DFS

GOLONGAN A

Disusun oleh :

Nama : Noga Muktiwati

NIM : E41200415

PROGRAM STUDI TEKNIK INFORMATIKA

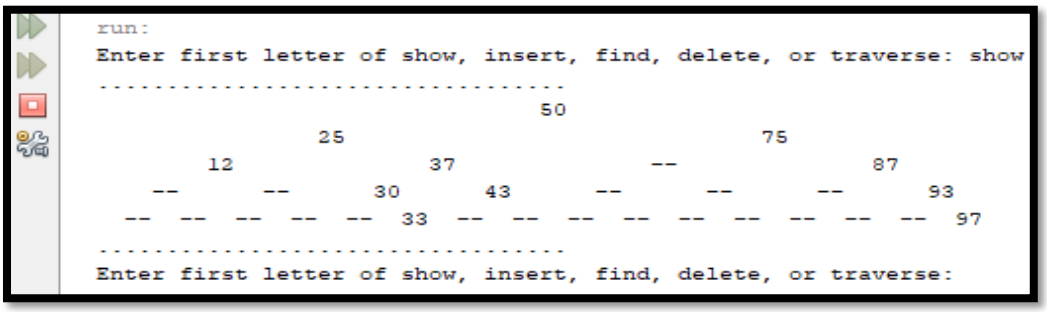
JURUSAN TEKNOLOGI INFORMASI POLITEKNIK NEGERI JEMBER 2020

Tugas Minggu 5

A. Binary Tree

a. **Tuliskan output program pada listing nomor 3 dalam bentuk tree ketika memilih menu Show**

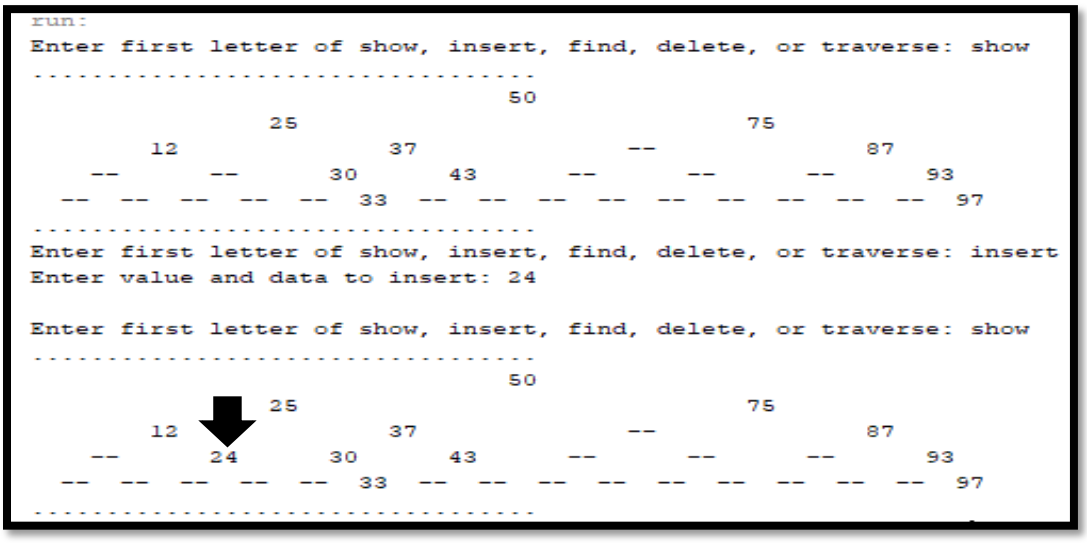
Jawab : **Ouput**



b. **Tambahkan satu node dengan value tertentu! Tampilkan tree, tunjukkan dimana posisi node yang baru ditambahkan, dan jelaskan langkah penambahan node yang diimplementasikan pada method insert!**

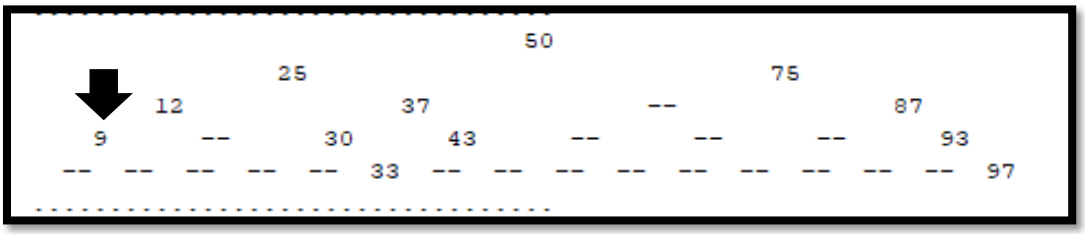
Jawab :

- Menambahkan satu node dengan menginputkan method **"insert"** Dengan **Value = 24**
- Karena **value 24 > node 12** maka 24 masuk child(anak) di sebelah **kanan**



- Coba inputkan **value 9 < node 24,** maka **9** akan menjadi **child (anak) dari node 24** berada di sebelah **kiri**

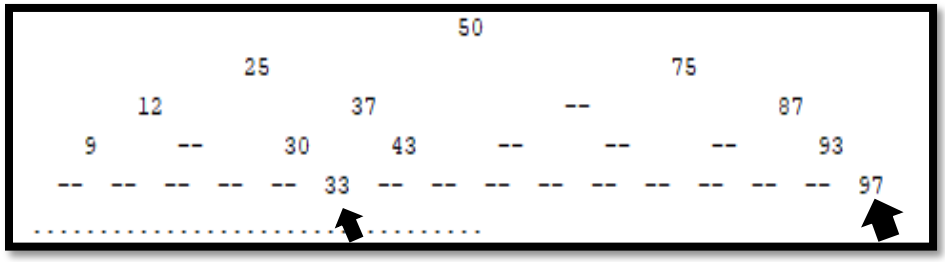
- **OUTPUT**



c. Lakukan percarian salah satu leaf pada tree tersebut! Tuliskan outputnya dan penjelasan langkah pencarian yang diimplementasikan pada method find! Anda dapat menggambarkan ilustrasi pencarian node tersebut pada tree sebagai visualisasi langkah pencarian

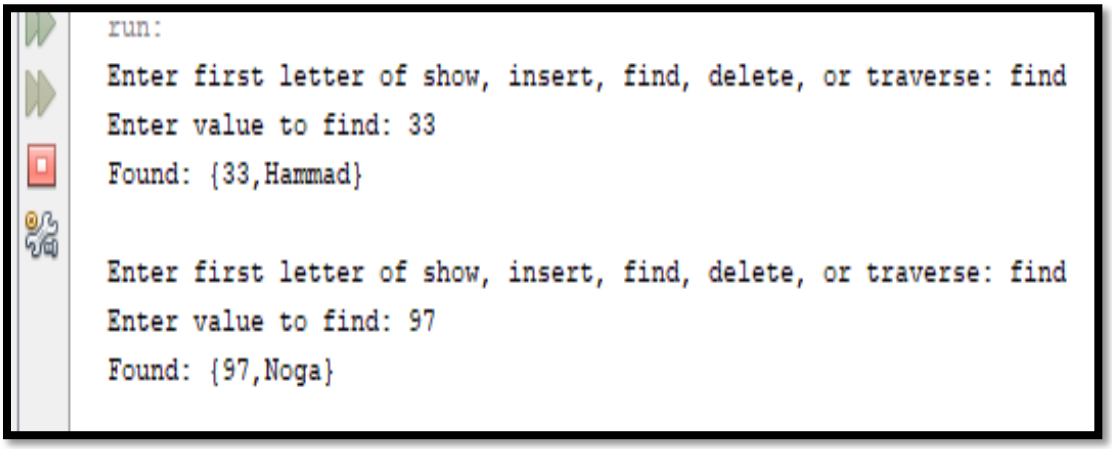
Jawab :

- Leaf adalah node yang **tidak punya children**



- **Leaf :** Node –node dalam tree yang tidak memiliki **successor** (node yang berada dibawah node tertentu)
- Maka menggunakan method **"Find"** di sini kita akan menemukan value **33** dan **97**

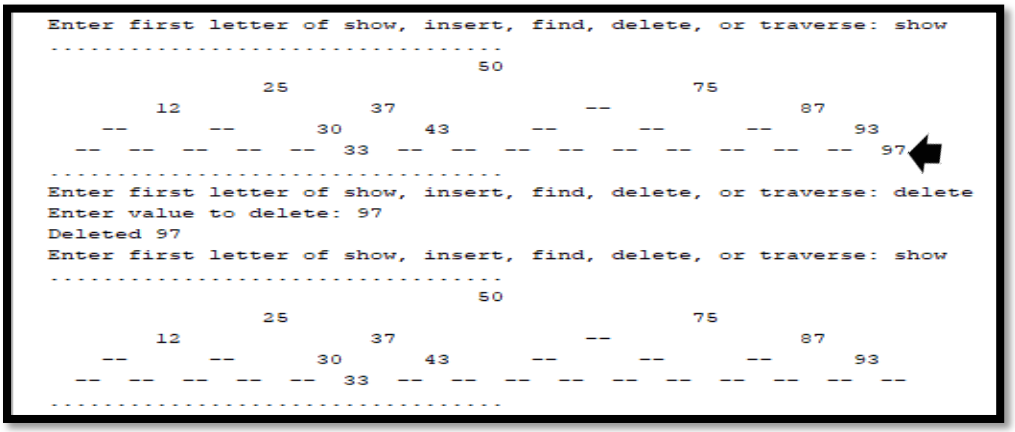
Output :



d. **Proses hapus node pada tree cukup kompleks dibandingkan dengan operasi lain. Langkah pertama yang dilakukan adalah mencari node yang akan dihapus.Terdapat 3 kemungkinan kondisi node yanga akan dihapus yaitu :**

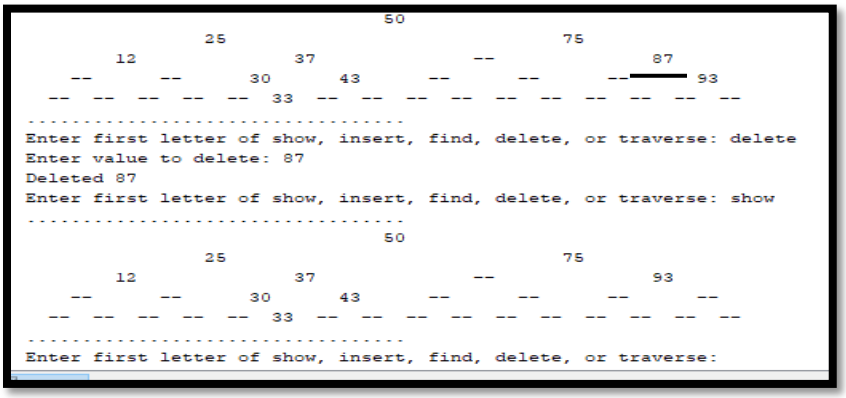
+ Node yang dihapus adalah leaf, yang berarti node tersebut tidak memiliki children.

Deleted = 97



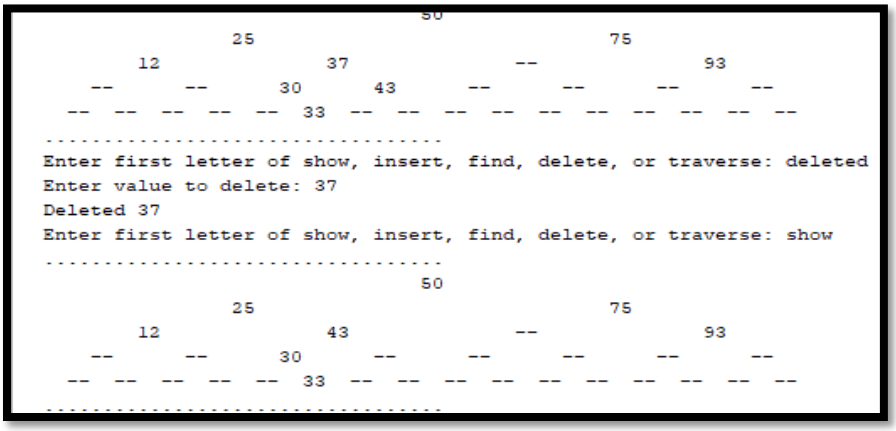
+ Node yang dihapus memiliki satu child.

Deleted = 87 otomatis satu child yang dimiliki node 87 naik



+ Node yang dihapus memiliki dua children. **Deleted =37**

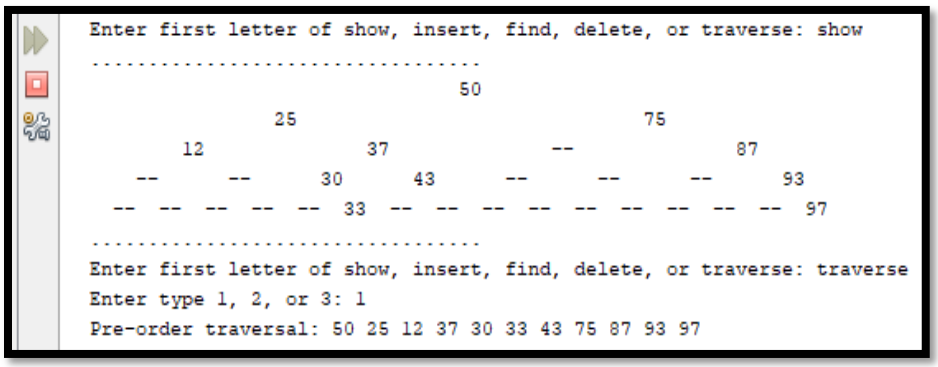
Karena, dari dua anak, **value 43 > 30** maka 43 menggantikan node 37



e. Terdapat tiga jenis operasi traverse, yaitu preorder, inorder, dan postorder. pada program nomor 3, jalankan masing-masing operasi traverse tersebut, tulis outputnya dan jelaskan perbedaan ketiga operasi tersebut!

a. Pre – order

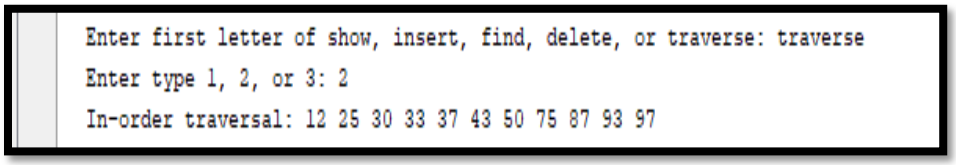
bekerja secara **ROOT - LEFT TREE - RIGHT TREE**
(root = 50) punya 2 children, kemudian **(left tree = 25)**
pada **(node 25)** punya 2 anak yang **(left=12)** yang **(right = 37)**
karena alurnya "**ROOT - LEFT TREE - RIGHT TREE"** maka child sebelah kiri dulu (value 12) baru ke anak sebelah kanan (value 37) dst



Urutan Output : 50, 25,12,37,30,33,43,75,87,93,97

b. In-Order

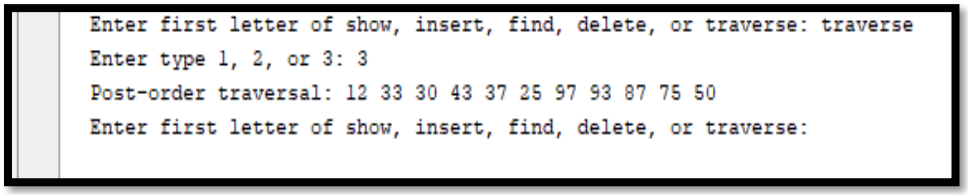
Bekerja secara **LEFT TREE - ROOT - RIGHT TREE**
(**Left tree value=12**) anak dari (**root = 25**) punya child di sebelah **(right tree=30)** dst



Urutan Output : 12, 25, 30, 33, 37, 43, 50, 75, 87, 93, 97

c. Post- Order

Bekerja secara LEFT TREE - RIGHT TREE - ROOT
(**Left tree = 12**) kemudian (**right three=33**) root dari 33 adalah (**root=30**) dst



Urutan Output : 12, 33, 30, 43, 37, 25, 97, 93, 87, 75, 5

B. Graph

a. Graph yang dibentuk pada program tersebut adalah:

Directed Graph

Pada program ini dikatakan directed graph karena Graph yang setiap sisinya memiliki orientasi arah disebut Graph berarah alias ada alur yang harus dipenuhi.

Tidak berlaku arus bolak balik. Apa maksudnya ?

Ini ada gambar ilustrasi dari implementasi kodingnya.

```
16          theGraph.addVertex('A'); //0
17          theGraph.addVertex('B'); //1
18          theGraph.addVertex('C'); //2
19          theGraph.addVertex('D'); //3
20          theGraph.addVertex('E'); //4
21
22          theGraph.addEdge(0, 1); //AB
23          theGraph.addEdge(1, 2); //BC
24          theGraph.addEdge(2, 3); //AD
25          theGraph.addEdge(3, 4); //DE
```

Disini A menduduki urutan ke 0,

B ke 1,

C ke 2,

D ke-3,

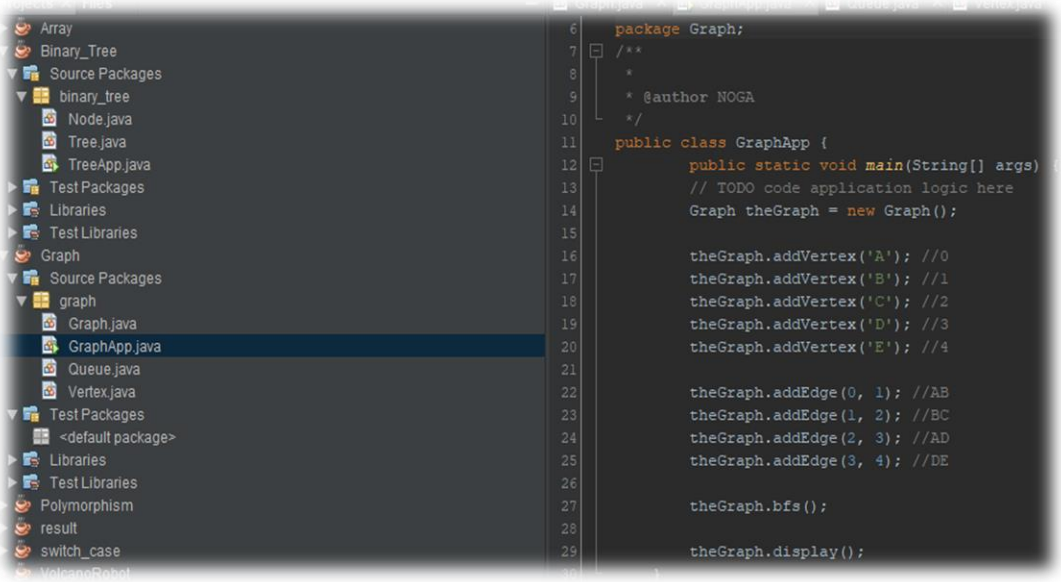
dan E ke- 4.

Urutan panah atau arah arusnya (arah alurnya) 0 ke 1 maka A ke B dan tidak berlaku B ke A tapi B ini menuju ke C kemudian 2 ke 3 maka A menuju D.

Output:

```
Output - Graph (run)
run:
Visit by using BFS algorithm:
A B C D E
Adjecency:
B -- A
C -- B
D -- C
E -- D

BUILD SUCCESSFUL (total time: 0 seconds)
```

b. Panggil method bfs() pada class GraphApp.
Jalankan program, Bagaimana output yang dihasilkan? Jelaskan
Panggil method bfs() dalam class Graph App. Runkan (Shift + F6)



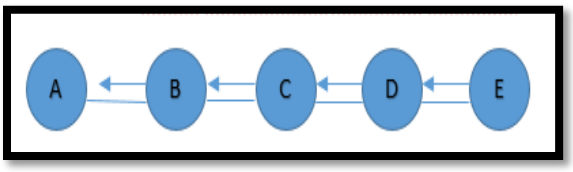
Output yang dihasilkan :

```
run:
Visit by using BFS algorithm:
A B C D E
```

BFS (Breadth – first-search)

Adalah algoritma pencarian, simpul yang dikunjungi disimpan dalam suatu antrian. Antrian ini, digunakan untuk mengacu simpul – simpul yang bertetangga dengannya, akan dikunjungi kemudian sesua sorting (urutan pengantrian) Untuk memperjelas cara kerja algoritma BFS beserta antrian yang digunakannya

BFS di implementasikan menggunakan queue :



Rules :

a. Kunjungi vertex selanjutnya yang berhubungan (jika ada), tandai vertex tersebut, dan insert pada queue.
b. Jika tidak dapat melakukan rule 1 karena tidak ada vertex terhubung yang belum dikunjungi, maka remove vertex dari queue (jika ada), dan jadikan vertex tersebut sebagai current vertex.
c. Jika tidak dapat melakukan rule 2 karena queue telah kosong, maka proses BFS telah selesai

DFS (Deep First Search)

Adalah pencarian meluaskan anak akar pertama yang dipilih dan berjalan dalam dan lebih dalam lagi sampai simpul tujuan ditemukan, atau sampai menemukan simpul yang tidak punya anak. Kemudian, pencarian backtracking (penelusuran balik untuk mendapat jalur yang diinginkan), akan kembali ke simpul yang belum selesai ditelusuri.

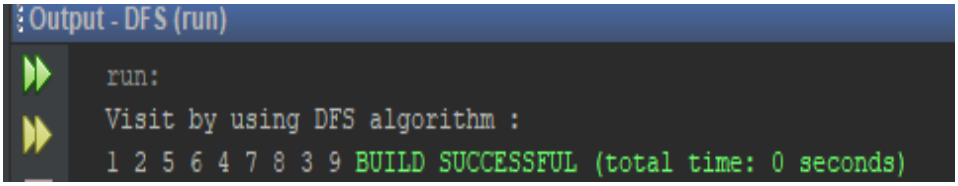
Code program :

Graph.java | GraphApp.java | Queue.java | Vertex.java | DFS.java

```
package dfs;
import java.util.*;
/**
 *
 * @author NOGA
 */
public class DFS {
    int V;
    LinkedList<Integer> adjList[];


    DFS(int v) {
        System.out.println("Visit by using DFS algorithm :");
        V = v;
        adjList = new LinkedList[v];
        for (int i=0; i<v; ++i)
            {
            adjList[i] = new LinkedList();
                }
    }

    void addEdgesToGraph(int v, int u)
    {
        adjList[v].add(u);
    }

    void DFTraversal(int v,int visited[])
    {
        visited[v] = 1;
            System.out.print(v + " ");
        Iterator<Integer> i = adjList[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (visited[n]==0)
                DFTraversal(n, visited);
        }
    }
```

```
    }

    void DFSearch(int v)
    {

        int visited[] = new int[V];

        DFTraversal(v, visited);
                for (int i=1;i<V;i++)
                {
                        if(visited[i]==0)
                        {
                    DFTraversal(i, visited);
                        }
                }
    }

    public static void main(String args[])
    {
        DFS obj = new DFS(10);

        obj.addEdgesToGraph(1,2);
        obj.addEdgesToGraph(1,4);
        obj.addEdgesToGraph(2,5);
        obj.addEdgesToGraph(2,6);
        obj.addEdgesToGraph(4,7);
        obj.addEdgesToGraph(4,8);
        obj.addEdgesToGraph(3,9);
        obj.addEdgesToGraph(3,4);
        obj.addEdgesToGraph(4,3);



        obj.DFSearch(1);
    }
}
```

Output :



TERIMAKASIH ☺