MODUL V

**TREE**

* 1. **TUJUAN**

Tujuan dari praktikum ini adalah:

1. Mahasiswa dapat memahami konsep dan algoritma *tree*.
2. Mahasiswa dapat mengimplementasikan penggunaan *tree* dalam program.
   1. **DASAR TEORI**
      1. ***Tree***

*Tree* atau pohon merupakan struktur data yang tidak linear yang digunakan untuk mempresentasikan data yang bersifat hirarki antara elemen-elemennya. Definisi *tree* yaitu kumpulan elemen yang salah satu elemennya disebut *root* (akar) dan elemen yang lain disebut simpul (*node*) yang terpecah menjadi sejumlah kumpulan yang tidak saling berhubungan satu sama lain yang disebut *sub*-*tree* atau cabang [3].

|  |
| --- |
| max-heap |

Gambar 5.1 Ilustrasi *tree*

*Tree* (pohon) terdiri dari [4]:

1. *Root* (akar) adalah simpul yang tidak memiliki *predecessor*. Untuk pohon yang dicontohkan di atas, maka *root* adalah *node* *object*.
2. *Leaf* (daun) adalah *node* yang tidak memiliki *child* atau *node* yang berada pada hirarki paling bawah.
3. *Height* (tinggi) atau *path* adalah jumlah tingkat dari sebuah *tree*.
4. *Subtree* (anak pohon) adalah beberapa *node* yang tersusun hirarki yang ada di bawah *root*. Tabel berikut menjelaskan beberapa istilah yang terdapat pada *tree*:

**Tabel 5.1** Istilah-istilah pada *Tree*

|  |  |
| --- | --- |
| **Terminologi** | **Definisi** |
| *Predecessor* | *Node* yang berada di atas *node* tertentu. |
| *Successor* | *Node* yang berada di bawah *node* tertentu. |
| *Ancestor* | Seluruh *node* yang terletak sebelum *node* tertentu dan terletak pada jalur yang sama. |
| *Descendant* | Seluruh *node* yang terletak setelah *node* tertentu dan terletak pada jalur yang sama. |
| *Parent* | *Predecessor* satu level di atas suatu *node*. |
| *Child* | *Successor* satu level di bawah suatu *node*. |
| *Sibling* | *Node*-*node* yang memiliki *parent* yang sama. |
| *Subtree* | Suatu *node* beserta *descendant*-nya. |
| *Size* | Banyaknya *node* dalam suatu *tree*. |
| *Height* | Banyaknya tingkatan dalam suatu *tree*. |
| *Root* | *Node* khusus yang tidak memiliki *predecessor*. |
| *Leaf* | *Node*-*node* dalam *tree* yang tidak memiliki *successor*. |
| *Degree* | Banyaknya *child* dalam suatu *node*. |
| *Forest* | Kumpulan dari *tree*. |
| *Depth* | Hasil tingkat *node* maksimum dikurang satu. |

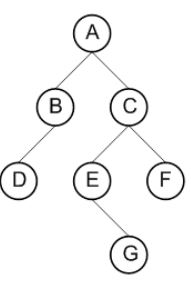
Jenis-jenis dari *tree* [5]:

* 1. *Binary Tree*

*Binary* *tree* adalah *tree* dengan syarat bahwa tiap *node* hanya boleh memiliki maksimal dua *sub-tree* dan kedua *sub*-*tree* tersebut harus terpisah. Sesuai dengan definisi tersebut, maka tiap *node* dalam *binary* *tree* hanya boleh memiliki paling banyak dua *child*.

|  |
| --- |
| Description: H:\binary.JPG  Description: H:\binary.JPG |

Gambar 5.2 Ilustrasi *binary* *tree*



Jenis-jenis *Binary* *Tree*:

1. *Full Binary Tree*

*Binary tree* yang tiap *node*-nya (kecuali *leaf*) memiliki dua *child* dan tiap *sub-tree* harus mempunyai panjang *path* yang sama.

|  |
| --- |
| Description: http://4.bp.blogspot.com/_612DHwdvDBM/Sh_yQLtqtyI/AAAAAAAAAFQ/tFA-r1W6-jM/s320/41.JPG |

Gambar 5.3 Ilustrasi *full binary* *tree*

1. *Complete Binary Tree*

Mirip dengan *full binary tree*, namun tiap *sub-tree* boleh memiliki panjang *path* yang berbeda. *Node* kecuali *leaf* memiliki 0 atau 2 *child*.

|  |
| --- |
| Description: http://4.bp.blogspot.com/_612DHwdvDBM/Sh_ym2tzNJI/AAAAAAAAAFY/cNps0BF_j6w/s320/42.JPG |

Gambar 5.4 Ilustrasi *complete binary* *tree*

1. *Skewed Binary Tree*

Merupakan *binary tree* yang semua *node* nya (kecuali *leaf*) hanya memiliki satu *child.*

|  |
| --- |
| Description: http://4.bp.blogspot.com/_612DHwdvDBM/Sh_zDd91GRI/AAAAAAAAAFg/B4CcyfKPUpo/s320/43.JPG |

Gambar 5.5 Ilustrasi *skewed binary* *tree*

* 1. *Binary Search Tree*

*Binary Search Tree* (BST) adalah jenis pohon terurut yang digunakan untuk menyimpan data sehingga memudahkan pencarian kembali data tersebut.

|  |
| --- |
| Description: H:\BST.JPG |

Gambar 5.6 Ilustrasi  *binary search* *tree*

Operasi pada *Tree*

1. *Create* : Membentuk *binary tree* baru yang masih kosong.
2. *Clear* : Mengosongkan *binary tree* yang sudah ada.
3. *Empty* : Memeriksa apakah *binary tree* masih kosong.
4. *Insert* : Memasukkan sebuah *node* ke dalam *tree*.
5. *Find* : Mencari *root*, *parent*, *left child*, atau *right child* dari suatu jjgjjgjgjgjgjgjjj*node*. (*Tree* tak boleh kosong)
6. *Update* : Mengubah isi dari *node* yang ditunjuk oleh *pointer jg ajgjgjgjjhcurrent*. (*Tree* tidak boleh kosong)
7. *Retrieve* : Mengetahui isi dari *node* yang ditunjuk *pointer current*. hfhfhfhfhffhfhh(*Tree* tidak boleh kosong)
8. *DeleteSub* : Menghapus sebuah *sub*-*tree* (*node* beserta seluruh ghghhhghgjghhh*descendant*-nya) yang ditunjuk *pointer current*. (*Tree* jhjhjhjhjhjhjhhhhtak boleh kosong)
9. *Characteristic*: Mengetahui karakteristik dari suatu *tree*, yakni : *size*, kjjgjgjjgjgjgjggg*height*, serta *average length*-nya. (*Tree* tidak boleh dfdfdfdfdfdfdfdfkosong)
10. *Traverse* : Mengunjungi seluruh *node*-*node* pada *tree*, masing-ghghghghhhhhh masing sekali.
    * 1. ***Tree Traversal***

*Tree traversal* merupakan sebuah kunjungan yang berawal dari *root*, mengunjungi setiap *node* dalam *tree* masing-masing sekali. Kunjungan atau *traversing* dapat dilakukan dengan 3 cara yaitu *preorder*, *inorder* dan *postorder* [5].

1. *Preorder*

Algoritma *preorder* :

1. Mencetak info pada *node* yang dikunjungi.
2. Mengunjungi cabang kiri.
3. Mengunjungi cabang kanan.
4. *Inorder*

Algoritma *inorder* :

1. Mengunjungi cabang kiri.
2. Mencetak info pada *node* yang dikunjungi.
3. Mengunjungi cabang kanan.
4. *Postorder*

Algoritma *postorder* :

1. Mengunjungi cabang kiri.
2. Mengunjungi cabang kanan.
3. Mencetak info pada *node* yang dikunjungi.
   * 1. ***Breadth-first search* (BFS) dan *Depth-First-Search* (DFS)**
4. *Breadth-first search* (BFS)

*Breadth-first search* (BFS)melakukan proses *searching* pada semua *node* yang berada pada level atau hirarki yang sama terlebih dahulu sebelum melanjutkan proses *searching* pada *node* di level berikutnya [3].

|  |
| --- |
| Description: H:\BFS.JPG |

Gambar 5.7 Ilustrasi  *breadth- first- search*

Berdasarkan gambar di atas, maka urutan penelurusannya adalah A-B-C-D-E-F-G-H-I-J-K-L-M.

1. *Depth-First-Search* (DFS)

*Depth-First-Search* (DFS) adalah salah satu algoritma penelusuran struktur graf / pohon berdasarkan kedalaman. Simpul ditelusuri dari *root* kemudian ke salah satu simpul anaknya (misalnya prioritas penelusuran berdasarkan anak pertama (simpul sebelah kiri), maka penelusuran dilakukan terus melalui simpul anak pertama dari simpul anak pertama level sebelumnya hingga mencapai level terdalam. Setelah sampai di level terdalam, penelusuran akan kembali ke 1 level sebelumnya untuk menelusuri simpul anak kedua pada pohon biner (simpul sebelah kanan) lalu kembali ke langkah sebelumnya dengan menelusuri simpul anak pertama lagi sampai level terdalam dan seterusnya [6].

|  |
| --- |
| Description: H:\dfs.JPG |

Gambar 5.8 Ilustrasi  *depth- first- search*

Berdasarkan gambar di atas, maka urutan penelurusannya adalah A-B-E-F-G-C-H-I-J.

* 1. **PERMASALAHAN**

1. Membuat sebuah *Tree* yang menyimpan data mahasiswa berdasarkan urutan abjad berupa nama lengkap. Data mahasiswa yang disimpan adalah nama lengkap, NIM, nama dosen pembimbing, dan nomor telepon dosen pembimbing. Masukkan data awal secara statis.

Membuat menu:

1. Menambah data mahasiswa baru.
2. Melakukan pencarian terhadap data mahasiswa.
3. Menampilkan data secara *preoder*, *inorder*, dan *postorder.*

Menggunakan method “compareToIgnoreCase()”untuk menentukan urutan nama mahasiswa.

Sintaks: “string1.compareToIgnoreCase(string2);”

Apabila *string* 1 urutannya lebih dulu dari *string* 2 berdasarkan abjad, maka nilai kembalian *method* lebih kecil dari 0 (negatif). Jika sebaliknya, nilai kembalian *method* lebih besar dari 0 (positif).

**5.4 HASIL DAN PEMBAHASAN**

**5.4.1 Program Tree**

* + - 1. Algoritma

*Insert*

1. Masukkan sebuah data pertama yang akan masuk menjadi *root.*
2. Masukkan data berikutnya ke dalam *tree* tersebut.
3. Membandingkan data tersebut dengan data yang menjadi *root*, apabila data lebih besar dari *root* maka letaknya ada di sebelah kanan. Jika data lebih kecil, maka letaknya di sebelah kiri *root.*
4. Menentukan *subtree* sama seperti langkah c.

*Preorder*

1. Mendeklarasikan *pointer* ketua pada *root*.
2. Melakukan pengecekan pada *pointer* ketua.
3. Apabila ketua bernilai tidak sama dengan NULL.
4. Mencetak data pada variabel ketua.
5. Secara rekursif, mencetak data pada cabang sebelah kiri.
6. Secara rekursif, mencetak data pada cabang sebelah kanan.

*Inorder*

1. Mendeklarasikan *pointer* ketua pada *root*.
2. Melakukan pengecekan pada *pointer* ketua.
3. Apabila ketua tidak sama dengan NULL.
4. Secara rekursif, mencetak data pada cabang sebelah kiri.
5. Mencetak data pada variabel ketua.
6. Secara rekursif, mencetak data pada cabang sebelah kanan.

*Postorder*

Mendeklarasikan *pointer* ketua pada *root*.

Melakukan pengecekan pada *pointer* ketua.

Apabila ketua tidak sama dengan NULL.

Secara rekursif, mencetak data pada cabang sebelah kiri.

Secara rekursif, mencetak data pada cabang sebelah kanan.

Mencetak data pada variabel ketua.

Pencarian

1. Tentukan data yang akan dicari.
2. Data akan dicari pada *tree* dengan kondisi-kondisi tertentu.
3. Jika data yang dicari lebih besar dari *root*, pencarian akan dilakukan ke sebelah kanan.
4. Jika data yang dicari lebih kecil dari root, pencarian akan dilakukan ke sebelah kiri.
5. Tampilkan data jika telah ditemukan.
   * + 1. *Source code*

|  |
| --- |
| package jurnal;  import java.util.Scanner;  import java.util.jar.JarOutputStream;  class node{  String nama;  int nim;  String dosen;  int tlp;  node left, right;  public node(String a, int b, String c, int d){  nama=a;  nim=b;  dosen=c;  tlp=d;  }  }  public class jurnalmod5 {  node root;  public void Add(String a, int b, String c, int d) {  node temp = new node(a, b, c, d);  if (root == null) {  root = temp;  root.left = root.right = null;  } else {  node ha = null, cek = root;  while (cek != null) {  ha = cek;  if (a.compareToIgnoreCase(cek.nama) < 0) {  cek = cek.left;  } else {  cek = cek.right;  }  }  if (a.compareToIgnoreCase(ha.nama) < 0) {  ha.left = temp;  } else {  ha.right = temp;  }  }  }  public void preOrder(node root) {  if (root != null) {  System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);  preOrder(root.left);  preOrder(root.right);  }  }  public void inOrder(node root) {  if (root != null) {  inOrder(root.left);  System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);  inOrder(root.right);  }  }  public void postOrder(node root) {  if (root != null) {  postOrder(root.left);  postOrder(root.right);  System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);  }  }  public void show(node y) {  System.out.println("Nama: " + y.nama + "| NIM: " + y.nim + "| Nama Dosen: " + y.dosen + "| No. Telp: " + y.tlp);  }  public node Search(String nama, node root) {  node z;  if (nama.compareToIgnoreCase(root.nama) < 0) {  z = Search(nama, root.left);  } else if (nama.compareToIgnoreCase(root.nama) == 0) {  return root;  } else {  z = Search(nama, root.right);  }  return z;  }  public static void main(String[] args) {  Scanner Input = new Scanner(System.in);  Scanner I = new Scanner(System.in);  Scanner mas = new Scanner(System.in);  jurnalmod5 x = new jurnalmod5();  String name, namdos, polih;  int nim, notlp, pil;  String nama;    do {  System.out.println("Pilih Menu : ");  System.out.println("1. Tambah Data ");  System.out.println("2. Cari Data ");  System.out.println("3. tampilkan data ");  System.out.print("Pilih : "); pil = Input.nextInt();  switch (pil) {  case 1:  System.out.println("=============================");  System.out.print("Nama : "); nama = I.next();  System.out.print("NIM : "); nim = Input.nextInt();  System.out.print("Nama Dosen : "); namdos = I.next();  System.out.print("No Telepon : "); notlp = Input.nextInt();  x.Add(nama, nim, namdos, notlp);  System.out.println("=============================");  break;  case 2:  System.out.println("=============================");  System.out.println("nama : "); nama = mas.nextLine();  x.show(x.Search(nama, x.root));  System.out.println("=============================");  break;  case 3:  System.out.println("=============================");  System.out.println();  System.out.println("PreOrder");  x.preOrder(x.root);  System.out.println();  System.out.println("InOrder");  x.inOrder(x.root);  System.out.println();  System.out.println("PostOrder");  x.postOrder(x.root);  System.out.println("=============================");  }  System.out.println();  System.out.print("kembali ke menu (y/Y/except y): "); polih = I.next();  } while (polih.equals("y") || polih.equals("Y"));  }  } |

* + - 1. Hasil *run* :

|  |
| --- |
|  |

Gambar 5.9 Hasil *run* program

Pada **Gambar 5.9** menampilkan proses penambahan data pada sebuah *tree*, yang dimana data-data yang telah ditambahkan akan ditampilkan baik secara *preorder, inorder,* dan *postorder*.

* 1. **ANALISA**

**5.5.1 Program Tree**

|  |
| --- |
| class node{  String nama;  int nim;  String dosen;  int tlp;  node left, right;  public node(String a, int b, String c, int d){  nama=a;  nim=b;  dosen=c;  tlp=d;  }  } |

*Script* “class node{” adalah proses pembuatan *node*. Lalu ada pendeklarasian variabel “nama” dengan tipe data *String*,“nim” dengan tipe data *integer*,“dosen” dengan tipe data *String*,dan“telp” dengan tipe data *integer*, “left” dan “right” dengan tipe data buatan “node”. Di dalam kelas tersebut dibuatkan *method* konstruktor sesuai dengan nama kelasnya dan terdapat parameter “nama”, “nim”, “dosen”, dan “telp”, yang di dalam konstruktor tersebut dideklarasikan variabel “a” sama dengan “nama”, “b” sama dengan “nim”, “c” sama dengan “dosen”, dan “d” sama dengan “telp”.

|  |
| --- |
| public class jurnalmod5 {  node root;  public void Add(String a, int b, String c, int d) {  node temp = new node(a, b, c, d);  if (root == null) {  root = temp;  root.left = root.right = null;  } else {  node ha = null, cek = root;  while (cek != null) {  ha = cek;  if (a.compareToIgnoreCase(cek.nama) < 0) {  cek = cek.left;  } else {  cek = cek.right;  }  }  if (a.compareToIgnoreCase(ha.nama) < 0) {  ha.left = temp;  } else {  ha.right = temp;  }  }  } |

*Script* “public class jurnalmod5 {” merupakan *class* dengan nama “jurnalmod5”. Di dalamnya terdapat pendeklarasian “root” dan juga *method-method* yang akan digunakan dalam program, yaitu “add, preOrder, inOrder, postOrder, show, search”. Untuk *method* “add” memiliki parameter “nama”, “nim”, “alamat” dan “no\_telp”. Di dalamnya terdapat kondisi “if(root==null) ” berfungsi untuk membuat *root* dan baru diletakkan di *root*. Jika kondisi tidak terpenuhi, maka akan dilakukan pemeriksaan kondisi “if(a.compareToIgnoreCase(cek.nama)>0)”, dimana kondisi ini berfungsi untuk membandingkan dua data *string* yang jika kondisi tersebut terpenuhi, maka *node* baru akan diletakkan di sebelah kiri *root*. Jika kondisi tidak terpenuhi, *node* baru akan diletakkan disebelah kanan *root*.

|  |
| --- |
| public void preOrder(node root) {  if (root != null) {  System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);  preOrder(root.left);  preOrder(root.right);  }  } |

*Script* “public void preOrder(node root) {” merupakan *method* “preOrder” yang digunakan untuk membaca *tree* secara *Pre*-*Order*. Di dalamnya terdapat kondisi “if(root!= null)” yang berfungsi untuk membatasi perulangan secara rekursif. *Script* “System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp); System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);” berfungsi untuk menampilkan data-data pada “root”. *Script* “preOrder(root.left);” digunakan untuk memanggil *method* “preOrder” dengan parameter “root.left” untuk mencetak *node* sebelah kiri. *Script* “preOrder(root.right)” digunakan untuk mencetak *node* sebelah kanan.

|  |
| --- |
| public void inOrder(node root) {  if (root != null) {  inOrder(root.left);  System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);  inOrder(root.right);  }  } |

*Script* “public void inOrder(node root) {” merupakan *method* “inOrder” yang digunakan untuk membaca *tree* secara *In*-*Order*. Di dalamnya terdapat kondisi “if(root!= null)” yang berfungsi untuk membatasi perulangan secara rekursif. *Script* “inOrder(root.left);” digunakan untuk memanggil *method* “inOrder” dengan parameter “root.left” untuk mencetak *node* sebelah kiri. *Script* “System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);” berfungsi untuk menampilkan data-data pada “root”. *Script* “inOrder(root.right)” digunakan untuk mencetak *node* sebelah kanan.

|  |
| --- |
| public void postOrder(node root) {  if (root != null) {  postOrder(root.left);  postOrder(root.right);  System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);  }  } |

*Script* “public void postOrder(node root) {” merupakan *method* “postOrder” yang digunakan untuk membaca *tree* secara *Post*-*Order*. Di dalamnya terdapat kondisi “if(root!= null)” yang berfungsi untuk membatasi perulangan secara rekursif. *Script* “postOrder(root.left);” digunakan untuk memanggil *method* “postOrder” dengan parameter “root.left” untuk mencetak *node* sebelah kiri. *Script* “postOrder(root.right)” digunakan untuk mencetak node sebelah kanan. *Script* “System.out.println("Nama: " + root.nama + "| NIM: " + root.nim + "| NamaDosen: " + root.dosen + "| No. Telp: " + root.tlp);” berfungsi untuk menampilkan data-data pada “root”.

|  |
| --- |
| public void show(node y) {  System.out.println("Nama: " + y.nama + "| NIM: " + y.nim + "| Nama Dosen: " + y.dosen + "| No. Telp: " + y.tlp);  } |

*Script* “public void show(node y) {” merupakan *method* “show” yang digunakan untuk menampilkan data yang ada pada program.

|  |
| --- |
| public node Search(String nama, node root) {  node z;  if (nama.compareToIgnoreCase(root.nama) < 0) {  z = Search(nama, root.left);  } else if (nama.compareToIgnoreCase(root.nama) == 0) {  return root;  } else {  z = Search(nama, root.right);  }  return z;  } |

*Script* “public node Search(....” merupakan *method* untuk mencari suatu data berdasarkan nama yang dimana bertipe data *string*. Pendeklarasian variabel “z” dengan tipe data buatan “node”. Terdapat kondisi yang sama seperti pada *method* “add”, dimana terdapat perbandingan untuk kedua *string* dan jika lebih besar dari 0, maka pencarian akan dilakukan ke sebelah kanan. Jika sama dengan 0, dilakukan pengembalian ke “root” atau “return root”. Jika kurang dari 0, maka pencarian akan dilakukan ke sebelah kiri.

|  |
| --- |
| public static void main(String[] args){ |

Potongan *code* di atas merupakan fungsi utama dalam program. Semua *statement* yang berada dalam fungsi main ini yang akan dijalankan. Program dijalankan dengan pembuatan *object* baru lalu pemanggilan *method*.

|  |
| --- |
| Scanner Input = new Scanner(System.in);  Scanner I = new Scanner(System.in);  Scanner mas = new Scanner(System.in);  jurnalmod5 x = new jurnalmod5();  String nama, namdos, polih;  int nim, notlp, pil; |

Potongan-potongan *code* “Scanner”, “jurnalmod5”, “String” dan “int” terdapat “jurnalmod5” yang berguna untuk memasukkan data “x” dimana proses tersebut merupakan proses pembuatan *object* baru.

|  |
| --- |
| do {  System.out.println("Pilih Menu : ");  System.out.println("1. Tambah Data ");  System.out.println("2. Cari Data ");  System.out.println("3. tampilkan data ");  System.out.print("Pilih : "); pil = Input.nextInt();  switch (pil) { |

Potongan *code* “System.out.println("Pilih Menu : ");” adalah proses perulangan yang menampilkan pilihan menu di dalam program. Perulangan dilakukan dengan menggunakan perintah *do*-*while* agar terus mengulang selama *statement* kondisi perulangannya masih terpenuhi. Menu-menu yang terdapat di dalam program dibuat didalam *statement control switch*-*case* karenamerupakan menu yang berbentuk pilihan dan tidak memerlukan kondisi-kondisi khusus untuk dijalankan.

|  |
| --- |
| case 1:  System.out.println("=============================");  System.out.print("Nama : "); nama = I.next();  System.out.print("NIM : "); nim = Input.nextInt();  System.out.print("Nama Dosen : "); namdos = I.next();  System.out.print("No Telepon : "); notlp = Input.nextInt();  x.Add(nama, nim, namdos, notlp);  System.out.println("=============================");  break; |

*Script* “case 1:” *Case* pertama didalam menu adalah untuk melakukan penambahan data. Pada saat menambahkan data, *user* akan memasukkan *input*-an berupa nama mahasiswa, NIM, nama dosen pembimbing, dan No. telepon dosen pembimbing. Setelah *user* selesai memasukkan semua data, data-data tadi akan dimasukkan ke dalam *tree* dengan memanggil *method* “Add” menggunakan objek “x” yang sudah dideklarasikan sebelumnya.

|  |
| --- |
| case 2:  System.out.println("=============================");  System.out.println("nama : "); nama = mas.nextLine();  x.show(x.Search(nama, x.root));  System.out.println("=============================");  break; |

*Script* “case 2:” *Case* kedua dalam menu adalah pencarian data. Proses pencarian data akan menggunakan *method* “show” dengan parameter *method* “search” yang memiliki parameter “(nama, x.root)” agar program langsung menampilkan data setelah dilakukan proses pencarian dari *input*-an *user* saat memilih menu tersebut.

|  |
| --- |
| case 3:  System.out.println("=============================");  System.out.println();  System.out.println("PreOrder");  x.preOrder(x.root);  System.out.println();  System.out.println("InOrder");  x.inOrder(x.root);  System.out.println();  System.out.println("PostOrder");  x.postOrder(x.root);  System.out.println("============================="); |

*Script* “case 3:” *Case* ketiga merupakan case yang hanya memanggil *method* pembacaan *tree* secara *pre*-*oder*, *in*-*oder*, dan *post*-*oder* kemudian ditampilkan di dalam jendela penampil *output* program.

|  |
| --- |
| System.out.println();  System.out.print("kembali ke menu (y/Y/except y): "); polih = I.next();  } while (polih.equals("y") || polih.equals("Y")); |

*Script* “System.out.print("kembali ke menu (y/Y/except y): ");” adalah *script* yang berfungsi untuk menanyakan *user* apakah ingin kembali ke menu, dan pilihan dari *user*-lah yang menjadi kondisi perulangan yang digunakan dalam menu, karea perulangan yang digunakan adalah perulangan *do*-*while*.

* 1. **KESIMPULAN**

Berdasarkan percobaan yang telah dilakukan, maka dapat ditarik kesimpulan sebagai berikut :

* 1. *Tree* memiliki konsep yang hampir sama dengan *linked* *list*. Namun struktur pada *tree* tidak linear seperti *linked* *list*, melainkan bertingkat. *Tree* memiliki sekumpulan elemen yang berawal dari elemen teratas yang disebut *root*, dan kemudian memiliki simpul yang bercabang dan tidak saling berhubungan.
  2. Mengimplementasikan *tree* ke dalam program, terlebih dahulu memahami konsep dasar *tree* yang dimana memiliki tingkatan elemen dan cara untuk memasukkan data yang harus dibandingkan terlebih dahulu, kemudian ditentukan untuk dimasukkan ke cabang kanan atau kiri. Terdapat tiga cara atau metode untuk melakukan penjelajahan pada elemen-elemen *tree* atau yang disebut *Tree* *Traversing* atau Metode *Traversal*, yaitu *Preorder*, *Inorder* dan *Postorder*.