Praktikum DAA: Binary Search Tree

1. Binary Tree

Binary tree adalah tree yang mempunyai maksimal dua buah child (biasanya disebut dengan left child dan right child). Binary tree dapat direpresentasikan secara objek oriented, di mana tree adalah objek yang dibangun dari sekumpulan objek node. Objek node ini memiliki atribut info. Selain itu, terdapat atribut left, right, dan parent, yang secara berturut-turut adalah node yang menjadi left child, right child, dan parent dari sebuah node.

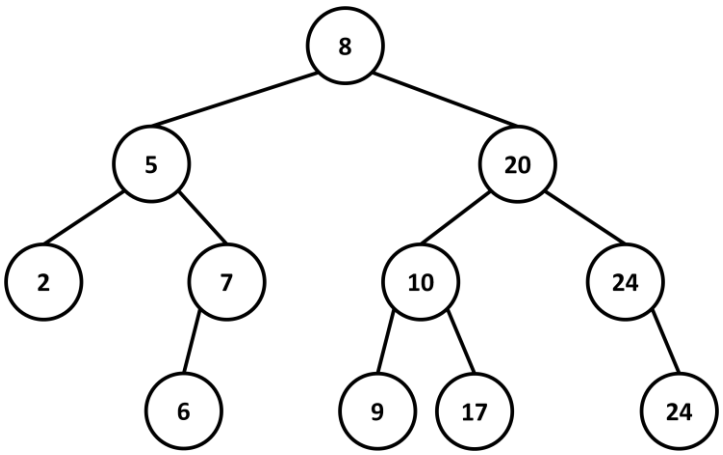
2. Binary Search Tree

Binary search tree adalah binary tree yang memiliki property:

Untuk setiap X yang merupakan sebuah node di dalam binary search tree,

- *Setiap node y yang terdapat pada left subtree dari x memiliki sifat: y.info < x.info.*
- *Setiap node y yang terdapat pada right subtree dari x memiliki sifat: y.info ≥ x.info*.

Contoh BST:



BST merupakan struktur data untuk menyimpan data berupa objek, yang nantinya dapat dilakukan operasi *query* (searching, finding maximum/minimum info, finding successor & predecessor). Keunggulan BST adalah kompleksitas operasi-operasinya yang relatif cepat. Selain operasi query, BST juga memiliki operasi untuk menambah dan mengurangi data (insertion & deletion), serta dapat digunakan untuk mengurutkan data (sorting).

3. Mengimplementasikan Binary Search Tree dengan Tipe Data Generic

Pada modul ini, akan dibuat BST yang dapat menyimpan objek berjenis apapun, yang penting objek tersebut dapat dibandingkan nilainya (pada Java, disebut `Comparable`). Pada Java kita dapat membuat kelas yang menerima input tipe data *Generic*. Tipe data Generic dapat digantikan dengan tipe data apapun pada saat kelas tersebut diinstansiasi.

Pertama-tama buatlah kelas `BinarySearchTree` seperti berikut ini:

```
class BinarySearchTree<T extends Comparable<T>>{

}
```

Kode `<T>` merupakan cara mendeklarasikan sebuah tipe data Generic. Supaya `T` dipastikan merupakan objek `Comparable`, maka kita tuliskan sebagai `<T extends Comparable<T>>`.

Untuk menginstansiasi objek BST, kita perlu menentukan tipe data yang akan menggantikan `T`, contohnya pada kode berikut ini, `T` akan digantikan dengan `Integer`.

```
class TesterBST{
    public static void main(String[] args){
        BinarySearchTree<Integer> bst = new BinarySearchTree<Integer>();

    }
}
```

4. Membuat Inner Class Node

BST dibangun oleh sekumpulan node, maka kita perlu membuat kelas `Node`. Kelas ini menyimpan sebuah info yang bertipe `T`, sesuai dengan yang dideklarasikan pada kelas `BinarySearchTree`. Selain itu, kelas `Node` ini hanya digunakan oleh kelas `BinarySearchTree` saja. Karena itu kita akan membuat kelas ini sebagai *inner class* dari kelas sebelumnya.

Inner class memiliki berbagai keuntungan, salah satunya adalah untuk enkapsulasi. Inner class dan *outer class*-nya dapat saling mengakses atribut maupun method mereka tanpa menggunakan *getter/setter*. Jika inner class dideklarasikan sebagai private, maka kelas-kelas lain tidak dapat mengakses inner class tersebut. Selain itu, jika ada kelas kecil yang hanya digunakan oleh satu kelas saja, kode akan lebih rapih dan mudah dipelihara jika kelas tersebut dijadikan inner class.

Berikut ini contoh membuat kelas `Node` sebagai inner class dari `BinarySearchTree`:

```
class BinarySearchTree<T extends Comparable<T>>{
    private Node root;

    private class Node{
        T info;
        Node left, right, parent;

        Node(T info){
            this.info = info;
        }
    }
}
```

Perhatikan bahwa kelas Node juga menggunakan tipe data Generic T. Karena kelas Node merupakan inner class, maka otomatis T yang digunakan di sini adalah T yang ada pada BinarySearchTree. Tipe data T digunakan untuk mendeklarasikan atribut *info*. Kelas ini dideklarasikan sebagai private, maka hanya outer class-nya yang dapat mengakses dan menggunakan. Atribut dan method pada kelas Node dibiarkan tanpa access modifier, karena kelasnya sudah dibuat private, maka otomatis sudah tidak dapat diakses dari luar.

5. Operasi Insert

Berikutnya kita akan menambahkan method `insert` pada kelas `BinarySearchTree`. Pada method ini, kita perlu membandingan nilai data baru dengan nilai-nilai pada node BST. Karena kita telah memastikan tipe data `T` merupakan object `Comparable`, maka kita dapat menggunakan method `compareTo()`. Selain itu, perhatikan bahwa atribut-atribut pada kelas `Node` dapat langsung diakses oleh method milik kelas `BinarySearchTree` tanpa perlu melalui getter/setter.

Pada slide diberikan pseudocode untuk method insert secara rekursif. Berikut ini contoh implementasi versi iteratifnya:

```
public void insert(T data){
    Node newNode = new Node(data);
    Node curr = this.root;
    Node parent = null;

    if(this.root == null)
        this.root = newNode;
    else{
        while(curr != null){
            parent = curr;
            if(data.compareTo(curr.info) < 0)
                curr = curr.left;
            else
                curr = curr.right;
        }

        newNode.parent = parent;
        if(data.compareTo(parent.info) < 0)
            parent.left = newNode;
        else
            parent.right = newNode;
    }
}
```

6. Inorder pada BST

Berikutnya kita akan meng-override method `toString()`, supaya dapat mencetak isi BST secara inorder. Seperti kita ketahui, urutan inorder pada BST akan menghasilkan nilai terurut. Mula-mula, tambahkan method `toString()` pada kelas `Node` sebagai berikut.

```
private class Node{
    T info;
    Node left, right, parent;

    Node(T info){
        this.info = info;
    }

    @Override
    public String toString(){
        return "[" + this.info.toString() + "]";
    }
}
```

Berikutnya tambahkan method `toString()` pada kelas `BinarySearchTree`. Method ini memanggil method `private inorder()` yang nantinya melakukan traversal pada BST mulai dari root.

```
@Override
public String toString(){
    return inorder(this.root);

}

private String inorder(Node x){
    if(x == null)
        return "";
    if(x.left == null && x.right == null)
        return x.toString();
    else
        return inorder(x.left) + x.toString() + inorder(x.right);
}
```

Ujilah kode anda dengan tester sebagai berikut. Pastikan outputnya merupakan nilai terurut membesar.

```
class TesterBST{
    public static void main(String[] args){
        BinarySearchTree<Integer> bst = new BinarySearchTree<Integer>();
        bst.insert(5);
        bst.insert(14);
        bst.insert(8);
        bst.insert(3);
        bst.insert(11);
        bst.insert(10);
        System.out.println(bst);
    }
}
```

7. Search

Method `search` pada BST menerima parameter berupa nilai yang ingin dicari. method ini mengembalikan objek nilai tersebut jika ditemukan, atau null jika tidak ditemukan. Sama dengan operasi `insert`, `search` dapat diimplementasikan secara rekursif maupun iteratif. Kali ini kita akan membuat versi rekursif karena nantinya method ini juga dipakai pada method `delete`. Buatlah method `search` public untuk diakses dari luar kelas BST, dan method search private untuk keperluan rekursif. Method search private memiliki return value bertipe `Node` karena nanti untuk mengimplementasikan method `delete`, yang diperlukan adalah nodenya. Perhatikan juga bahwa belum tentu data yang dicari ada dalam BST, karena itu pada method search public perlu diperiksa dulu apakah node hasil pencarian null atau bukan, jangan langsung diakses atribut `info`-nya.

```
public T search(T data){
    Node result =  search(data, this.root);
    if(result == null)
        return null;
    else
        return result.info;
}

private Node search(T data, Node curr){
    if(curr == null)
        return null;
    else if(curr.info.compareTo(data) == 0)
        return curr;
    else if(curr.info.compareTo(data) < 0)
        return search(data, curr.right);
    else
        return search(data, curr.left);
}
```

Ujilah method anda dengan menambahkan kode berikut ini pada kelas Tester:

```
System.out.println(bst.search(10));
System.out.println(bst.search(3));
System.out.println(bst.search(12));
```

output yang diharapkan adalah:

```
10
3
null
```

8. Latihan: Operasi min, max, predecessor, & successor

Operasi `min` berguna untuk mengembalikan nilai terkecil pada BST jika BST tidak kosong. Sama seperti pada method `search`, di sini kita juga memiliki dua method. Method public untuk dipakai dari luar kelas, maka return valuenya adalah `T`. Method private, berguna untuk keperluan di dalam kelas `BinarySearchTree`. Method ini nantinya dapat digunakan juga untuk mengimplementasikan operasi `successor` dan `delete`, karena itu method ini mengembalikan `Node<T>`. Lengkapi method-method berikut ini!

```
public T min(){ }
private Node min(Node curr){ }

public T max(){ }
private Node max(Node curr){ }

private Node successor(Node curr){ }
private Node predecessor(Node curr){ }
```

9. Latihan: Operasi delete

Operasi delete berguna untuk menghapus sebuah nilai pada BST. Method ini menerima parameter berupa nilai yang ingin dihapus. Untuk dapat menghapus sebuah node, pertama-tama perlu dicari dulu lokasi node tersebut. Untuk itu kita gunakan method search private yang sebelumnya sudah dibuat. Perhatikan dan lengkapi contoh berikut ini!

```
public boolean delete(T data){
    Node del = search(data, this.root);
    if(del == null)
        return false;    //data doesn't exist
    else{
        delete(del);
        return true;     //successfully deleted data
    }
}

private void delete(Node del){
    //lengkapi...
}
```

10. Latihan: Tipe data selain Integer

Keuntungan menggunakan tipe data Generic adalah kemampuannya diganti dengan tipe data apapun. Cobalah buat `BinarySearchTree` dengan tipe lain selain `Integer`, misalnya `Float`, `String`, dll. Kelas BST yang kita buat dapat digunakan untuk menyimpan objek apapun yang bertipe `Comparable`, bahkan kelas-kelas yang kita buat sendiri.

11. Kesimpulan: Enkapsulasi

Seperti telah disebutkan di awal, kita mengimplementasikan BST dengan memperhatikan konsep enkapsulasi. Perhatikan bahwa method public yang kita miliki semuanya menerima dan mengembalikan tipe data `T`, tidak ada satu pun yang menggunakan `Node`. Dapat dilihat di sini bahwa Node hanya diperlukan untuk proses internal pada operasi-operasi BST, sama sekali tidak perlu diketahui/diakses oleh kelas-kelas lain di luar BST. Dari luar, kelas BinarySearchTree terlihat sangat simple, yaitu hanya memiliki method:

- `insert(T)`
- `delete(T)`
- `search(T)`
- `min()`
- `max()`
- `toString()`