Modul 13 Implementasi Struktur Data
Binary Search Tree (BST)

13.1. Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep dan implementasi rekursif
2. Mengenal konsep dan implementasi Binary Search Tree (BST): insert dan searching
3. Dapat mengimplementasikan *in-order, pre-order* dan *post-order traversal* pada tree

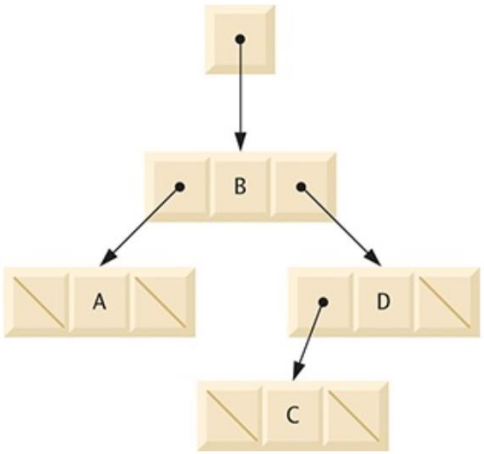
13.2. Alat dan Bahan

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

13.3. Dasar Teori

List, stack dan queue merupakan struktur data linier (dengan kata lain, berurut). Sementara, tree merupakan struktur data non-linier dua dimensi dengan properti khusus. Pada dasarnya, tree dapat dikatakan sebagai bentuk khsus dari graf. Suatu tree akan terdiri atas simpul-simpul (atau node), dan setiap node dapat berisi dua atau lebih link. Pada praktikum kali ini hanya akan dibahas mengenai binary search tree, salah satu bentuk binary tree.

Binary tree (pohon biner) merupakan tree yang setiap *node*-nya hanya memiliki dua *link*; dimana salah satu *link* tersebut, atau keduanya dapat saja tidak berisi null (tidak berisi). *Node* pertama pada tree disebut dengan root. Setiap *link* yang terhubung dengan *root* merujuk ke anak (*child*) dari root. *Left child* dari root merupakan *node* pertama dari *left subtree* (disebut juga sebagai *root node* dari *left subtree*), demikian *juga* right *child* dari *root* merupakan *node* pertama dari *right subtree*.



Gambar 1 Binary Tree (Deitel&Deitel, 2017)

Gambar 1 menunjukkan binary tree dengan node berisi B sebagai root. Anak-anak dari B (yaitu A dan D) disebut sebagai sibling. A dan C, yang tidak memiliki anak, disebut sebagai leaf. Karena pembuatan tree akan banyak menggunakan fungsi rekursif, maka sebelum membahas Binary Search Tree akan dibahas sekilas mengenai Rekursif untuk menyegarkan kembali materi mengenai hal tersebut.

13.3.1. Recursive

Sejauh ini, program yang telah kita buat pada dasarnya merupakan program dengan method yang memanggil satu sama lain dalam struktur hirarki. Namun, pada beberapa kasus, akan lebih baik jika method tersebut memanggil dirinya sendiri. Method yang memanggil dirinya sendiri disebut dengan rekursif, dan kadang kala ada juga method rekursif yang dipanggil secara langsung atau tidak langsung melalui method lain.

Pendekatan rekursif pada dasarnya terdiri atas dua bagian. Bagian **pertama** adalah **base case**(s). Base case ini merupakan bagian paling sederhana dari suatu kasus (masalah). Sebenarnya, method rekursif sebenarnya hanya bisa menyelesaikan bagian ini, jika method ini dipanggil dengan base case-nya, method tersebut akan mengembalikan hasil. Jika method dipanggil dengan kasus yang lebih kompleks, method akan membagi kasus tersebut menjadi dua bagian-bagian dimana method tahu cara memecahkannya dan bagian dimana method tidak tahu cara pemecahannya.

Agar pendekatan dengan cara rekursif ini bisa memecahkan kasus, bagian kedua dari method rekursif harus mirip dengan kasus awal, tapi lebih sederhana, atau merupakan bagian yang lebih kecil dari masalah tersebut. Karena kasus baru ini mirip dengan kasus awal, method tinggal memanggil replika dirinya sendiri untuk menyelesaikan kasus yang lebih kecil tersebut – hal ini disebut sebagai **pemanggilan rekursif** atau **recursion step**. Recursion step biasanya juga memiliki return statement, karena hasil dari tahap ini akan digabungkan dengan bagian method yang diketahui cara penyelesaiannya, kemudian bagian itu akan dikembalikan ke pemanggil awalnya.

Jadi, method rekursif bekerja dengan dua bagian; base case dan recursion step. Recursion step dieksekusi ketika method pemanggil awal masih aktif (belum selesai dieksekusi). Karenanya, bisa jadi akan terbentuk banyak pemanggilan rekursif lagi ketika method membagi-bagi sub-kasus menjadi dua bagian. Agar bagian rekursif ini berhenti, setiap kali method memanggil dirinya sendiri dengan versi yang lebih sederhana dari kasus awal, kasus-kasus yang lebih sederhana ini harus dapat *disatukan pada base case*. Cara pemecahan masalah dengan membagi-bagi masalah menjadi masalah yang lebih kecil, dan kemudian menyatukannya kembali disebut dengan *divide and conquer*.

Faktorial adalah salah satu kasus yang sering dijadikan contoh penyelesaian dengan cara rekursif. Seperti yang telah diketahui bersama, faktorial dari suatu bilangan bulat akan menghitung perkalian bilangan tersebut dari satu sampai bilangan tersebut. Jadi, n! (n faktorial) dapat dijabarkan sebagai berikut:

$$n * (n-1) * (n-2) \ldots * 1$$

Dengan 1! sama dengan 1 dan 0! didefinisikan sebagai 1. Sebagai contoh, 5! Adalah 5*4*3*2*1 = 120. Menghitung factorial dari suatu bilangan bulat n (dengan n>= 0) dapat dilakukan dengan cara iterative:

```
factorial = 1;
for (int counter = number; counter >= 1; counter--) {
   factorial *= counter;
}
```

Gambar 2 Iterative factorial (Deitel&Deitel, 2017)

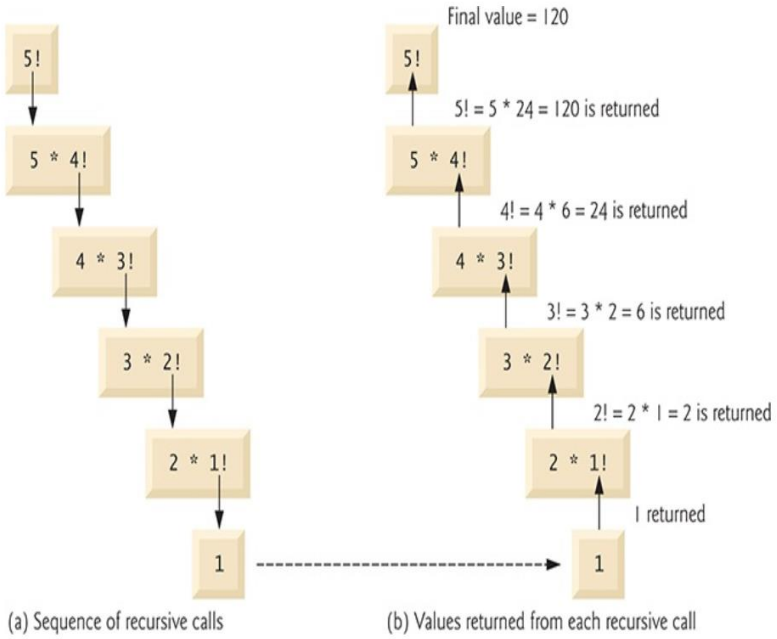
Sementara itu, cara rekursif bagi factorial diperoleh dari:

$$n! = n * (n-1)!$$

Untuk contoh 5!, diperoleh 5! = 5*4!, yaitu:

$$5! = 5 * 4 * 3 * 2 * 1$$
$$5! = 5 * (4 * 3 * 2 * 1)$$
$$5! = 5 * 4!$$

Lebih jauh, pemanggilan rekursif bagi 5! dapat dilihat pada Gambar 3.



Gambar 3 Recursive step of 5! (Deitel&Deitel, 2017)

Hal yang harus diperhatikan saat membuat suatu program rekursif adalah base case dan recursion step-nya. Jika base case tidak diberikan, atau jika salah menentukan recursion step, maka bisa saja terjadi infinite recursion (perulangan tidak berhenti), seperti yang dapat terjadi pada pembuatan perulangan.

Program berikut menghasilkan factorial untuk bilangan 0 sampai 21. Recursive call terlihat dari pemanggilan method factorial di dalam method tersebut (bagian else dari method factorial).

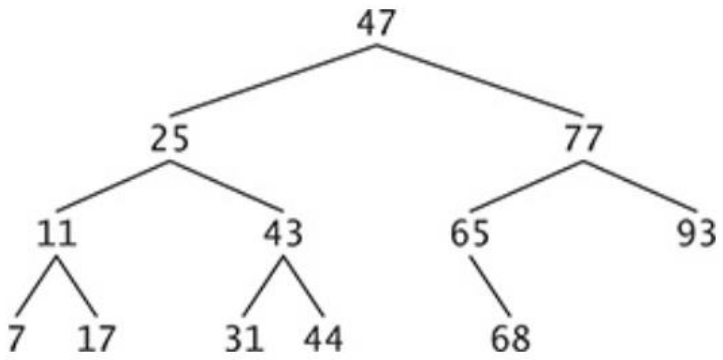
```
public class Factorial {
    public static long factorial(long number) {
        if (number <= 1) { // test for base case
            return 1; // base cases: 0! = 1 and 1! = 1
        } else { // recursion step
            return number * factorial(number - 1);
        }
    }

    public static void main(String[] args) {
        // calculate the factorials of 0 through 21
        for (int counter = 0; counter <= 21; counter++) {
            System.out.printf("%d! = %d%n", counter, factorial(counter));
        }
    }
}
```

13.3.2. Binary Search Tree

Binary search tree (BST) merupakan tree dengan karakteristik khusus; nilai dari node-node yang berada pada subtree kiri lebih kecil daripada nilai pada node parent dari subtree tersebut, dan nilai

dari node-node yang berada pada subtree kanan harus lebih besar daripada nilai pada node parent-nya. Gambar 4 mengilustrasikan suatu BST dari 12 nilai integer.



Gambar 4 Binary Search Tree (Deitel&Deitel, 2017)

Berikut merupakan contoh program (Deitel&Deitel, 2017) untuk Binary Search Tree (BST). Karena BST adalah suatu Tree, maka pertama-tama perlu dibuat kelas untuk membuat node bagi tree nya (kelas TreeNode). Kelas ini merupakan kelas generic, dan karena BST yang terbentuk merupakan suatu pohon yang terurut secara *in-order*, maka data yang dimasukkan pada node harus bisa dibandingkan, sehingga kelas ini menggunakan interface Comparable.

Setiap *node* pada tree ini memiliki *link* ke anak kiri (leftNode) dan anak kanan (rightNode). Proses penyisipan data pada BST ini menggunakan metode rekursif yang terlihat dari pemanggilan method insert di dalam insert.

```
public class TreeNode<E extends Comparable<E>> {
    private TreeNode<E> leftNode;
    private E data; // node value
    private TreeNode<E> rightNode;

    // constructor initializes data and makes this a leaf node
    public TreeNode(E nodeData) {
        data = nodeData;
        leftNode = rightNode = null; // node has no children
    }

    public E getData() {
        return data;
    }

    public TreeNode<E> getLeftNode() {
        return leftNode;
    }

    public TreeNode<E> getRightNode() {
        return rightNode;
    }

    // locate insertion point and insert new node; ignore duplicate values
```

```
    public void insert(E insertValue) {
        // insert in left subtree
        if (insertValue.compareTo(data) < 0) {
            // insert new TreeNode
            if (leftNode == null) {
                leftNode = new TreeNode<E>(insertValue);
            } else { // continue traversing left subtree recursively
                leftNode.insert(insertValue);
            }
        }
        // insert in right subtree
        else if (insertValue.compareTo(data) > 0) {
            // insert new TreeNode
            if (rightNode == null) {
                rightNode = new TreeNode<E>(insertValue);
            } else { // continue traversing right subtree recursively
                rightNode.insert(insertValue);
            }
        }
    }
}
```

Setelah mendeklarasikan *node* pada kelas TreeNode, maka node tersebut dibentuk menjadi BST pada kelas Tree. Untuk program ini, BST yang terbentuk akan memiliki nilai yang unik. Jika akan dimasukkan nilai yang sama dengan yang telah ada pada BST sebelumnya, nilai tersebut otomatis tidak akan masuk. Pada kelas ini juga terdapat penelusuran secara *pre-order*, *in-order* dan *post-order*. Selain itu, terdapat juga method *searching* (Carrano, 2012) pada BST. Perhatikan, *method-method* tersebut juga merupakan *method* rekursif (pada bagian *helper* nya).

```
public class Tree<E extends Comparable<E>> {
    private TreeNode<E> root;

    // constructor initializes an empty Tree of integers
    public Tree() {
        root = null;
    }

    // insert a new node in the binary search tree
    public void insertNode(E insertValue) {
        if (root == null) {
            root = new TreeNode<E>(insertValue); // create root node
        } else {
            root.insert(insertValue); // call the insert method
        }
    }

    // begin preorder traversal
    public void preorderTraversal() {
        preorderHelper(root);
    }
```

```java
    // recursive method to perform preorder traversal
    private void preorderHelper (TreeNode<E> node) {
        if (node == null) {
            return;
        }

        System.out.printf("%s ", node.getData()); // output node data
        preorderHelper(node.getLeftNode()); // traverse left subtree
        preorderHelper(node.getRightNode()); // traverse right subtree
    }

    // begin inorder traversal
    public void inorderTraversal() {
        inorderHelper(root);
    }

    // recursive method to perform inorder traversal
    private void inorderHelper(TreeNode<E> node) {
        if (node == null) {
            return;
        }

        inorderHelper(node.getLeftNode()); // traverse left subtree
        System.out.printf("%s ", node.getData()); // output node data
        inorderHelper(node.getRightNode()); // traverse right subtree
    }

    // begin postorder traversal
    public void postorderTraversal() {
        postorderHelper(root);
    }

    // recursive method to perform postorder traversal
    private void postorderHelper (TreeNode<E> node) {
        if (node == null) {
            return;
        }

        postorderHelper(node.getLeftNode()); // traverse left subtree
        postorderHelper(node.getRightNode()); // traverse right subtree
        System.out.printf("%s ", node.getData()); // output node data
    }

    public void searchBST(E key){
        boolean hasil = searchBSTHelper(root, key);
        if(hasil)
            System.out.println("Data ditemukan");
        else
            System.out.println("Data tidak ditemukan");
    }

    //Carrano
    public boolean searchBSTHelper(TreeNode<E> node, E key){
        boolean result = false;

        if(node!=null){
            if(key.equals(node.getData()))
                result = true;
```

```
            else if(key.compareTo(node.getData())<0)
                result = searchBSTHelper(node.getLeftNode(),key);
            else
                result = searchBSTHelper(node.getRightNode(),key);
        }
        return result;
    }
}
```

Pada kelas Main, dibangun BST dengan sepuluh *node*. Nilai yang dimasukkan merupakan bilangan bulat dari 0 – 100. Searching dilakukan untuk nilai 10. Jika data ditemukan (kebetulan dari nilai random dimasukkan nilai 10) maka akan muncul pernyataan "Data ditemukan". Jika tidak, yang akan muncul adalah "Data tidak ditemukan".

```
public class Main {

    public static void main(String[] args) {
        Tree<Integer> tree = new Tree<>();
        SecureRandom randomNumber = new SecureRandom();

        System.out.println("Inserting the following values: ");

        // insert 10 random integers from 0-99 in tree
        for (int i = 1; i <= 10; i++) {
            int value = randomNumber.nextInt(100);
            System.out.printf("%d ", value);
            tree.insertNode(value);
        }

        System.out.printf("%n%nPreorder traversal%n");
        tree.preorderTraversal();

        System.out.printf("%n%nInorder traversal%n");
        tree.inorderTraversal();

        System.out.printf("%n%nPostorder traversal%n");
        tree.postorderTraversal();
        System.out.println();

        tree.searchBST(10);
    }
}
```

Reference

Carrano, F., M., *Data Structures and Abstraction With Java,* Prentice Hall, (2012)
Deitel, P and Deitel H., *Java How to Program: Early Objects*, 11th Ed, Pearson. (2017)