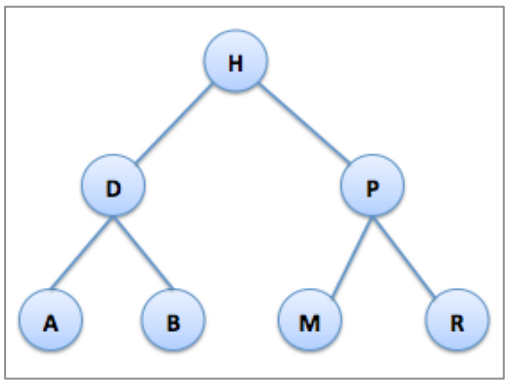
Membuat Binary Search Tree Menggunakan STL Vector C++

Pengantar

Binary Tree adalah struktur data tree yang hanya memiliki tepat dua anak (*child*), yang sering disebut sebagai anak kiri (*left child*) dan anak kanan (*right child*). Salah satu jenis binary tree adalah binary search tree (BST) yang banyak digunakan untuk penelusuran data tree yang memiliki sifat bahwa data yang lebih kecil atau sama dengan parent-nya diletakkan sebagai anak kiri, sedangkan data yang lebih besar dari parent-nya diletakkan sebagai anak kanan.

Misalkan data yang akan disimpan ke dalam BST berupa karakter **H D P A B M R**. Maka struktur BST dari data tersebut seperti gambar berikut:



Tentu saja, implementasi dari struktur data tree dapat dilakukan dengan berbagai cara. Salah satu struktur data yang dapat digunakan untuk implementasi BST ini adalah menggunakan vektor, dimana elemen vektor berupa node dari BST. Artikel ini membahas implementasi BST menggunakan STL (Standard Template Library) dalam C++, yaitu **std::vector**.

Awal Program

Program C++ yang dibuat, tentunya diawali dengan memasukkan beberapa header file yang berisi class yang diperlukan, sebagai berikut:

```
#include <iostream>
#include <vector>
using namespace std;
```

Struktur Data

Untuk membuat BST, dibutuhkan node BST yang minimal berisi tiga komponen, yaitu satu komponen untuk menyimpan nilai node, dan dua komponen untuk menyimpan indeks yang mengarah ke anak kiri dan indeks yang mengarah ke anak kanan. Terkait dengan contoh di atas, maka node dapat didefinisikan dengan tipe data berikut:

```
struct node {
   char value;                // nilai node
   int left, right;           // indeks ke anak kiri dan kanan
};
typedef struct node NODE;
```

Definisi Class

Struktur BST yang memiliki beberapa fungsi, selanjutnya dikemas dalam bentuk class seperti kode di bawah ini. Perhatikan bahwa nilai indeks -1 merepresentasikan pointer NULL.

```
class BST {
   vector<NODE> tree;
   int root;
   public:
      BST() { root=-1; }
      bool isEmpty() { return tree.empty(); }
      int getRoot() { return root; }
      void makeRoot(char aValue);
      void leftChild(unsigned int curr, char aValue);
      void rightChild(unsigned int curr, char aValue);
      void insert(char aValue);
      void inOrderTraversal(unsigned int curr);
      void preOrderTraversal(unsigned int curr);
      void postOrderTraversal(unsigned int curr);
};
```

Class BST berisi dua atribut data, yaitu: (1) tree berupa vektor dimana setiap elemen vektor berisi nilai node (**value**), indeks ke anak kiri (**left**), dan indeks ke anak kanan (**right**); (2) indeks untuk **root**. Dengan struktur seperti ini, maka gambar BST di atas dapat direpresentasikan dalam bentuk tabel vektor sebagai berikut:

index	value	left	right
0	H	1	2
1	D	3	4
2	P	5	6
3	A	-1	-1
4	B	-1	-1
5	M	-1	-1
6	R	-1	-1

Implementasi Fungsi Anggota Membuat dan Mengisi BST

Untuk dapat menyusun implementasi setiap fungsi anggota class BST, dapat dibayangkan terlebih dahulu aplikasi dari class BST tersebut sebagai berikut:

```
int main() {
   BST myTree;
   myTree.makeRoot('H');
   myTree.insert('D'); myTree.insert('P');
   myTree.insert('A'); myTree.insert('B');
   myTree.insert('M'); myTree.insert('R');

   int root=myTree.getRoot();
   myTree.inOrderTraversal(root);
   myTree.preOrderTraversal(root);
   myTree.postOrderTraversal(root);
   return 0;
}
```

Instruksi `BST myTree;` adalah mendefinisikan suatu obyek BST yang diberi nama myTree. Oleh karena itu, pada saat instruksi tersebut dijalankan, berarti dibuat sebuah BST dimana root dari tree tersebut adalah NULL. Ingat bahwa NULL direpresentasikan dengan nilai -1. Oleh karena itu, default constructor dari class BST seperti yang tertulis pada definisi class, yaitu:

```
BST() { root=-1; }
```

Instruksi `myTree.makeRoot('H');` adalah membuat BST pertama kali dimana nilai 'H' sebagai root-nya (node paling atas dari tree). Oleh karena itu, implementasi fungsi anggota makeRoot adalah:

```
void BST::makeRoot(char aValue) {
   NODE t={aValue, -1, -1};
   tree.push_back(t);
   root=0;
}
```

Dengan demikian, fungsi anggota `insert(elemen);` dapat dilakukan terhadap BST yang sudah memiliki root (sudah dijalankan fungsi `makeRoot(elemen);`). Hal ini terlihat pada instruksi awal fungsi anggota `insert(elemen);` berikut:

```
void BST::insert(char aValue) {
   if (isEmpty()) {
      cout << "Tree is not made yet. Please makeRoot first...\n";
      return;
   }
   unsigned int curr = root;
   while (curr<tree.size()) {
      if (aValue<=tree[curr].value) {
         if (tree[curr].left == -1) {
            leftChild(curr, aValue);
            break;
         } else curr = tree[curr].left;
      } else {
         if (tree[curr].right == -1) {
            rightChild(curr, aValue);
            break;
         } else curr = tree[curr].right;
      }
   }
}
```

Pernyataan:

```
   if (isEmpty()) {
      cout << "Tree is not made yet. Please makeRoot first...\n";
      return;
   }
```

diperlukan untuk memeriksa apakah BST tersebut sudah memiliki root (tree kosong atau tidak). Jika kosong, maka proses dihentikan (pernyataan `return;`).

Selanjutnya dilakukan penelusuran (traversal) yang dimulai dari root melalui instruksi:

```
   unsigned int curr = root;
```

Penelusuran dilakukan selama indeks dari node masih dalam rentang ukuran vektor, yaitu dengan instruksi pengulangan:

```
   while (curr<tree.size()) {
```

Pada setiap kunjungan ke node dalam BST, dilakukan pemeriksaan terhadap nilai node. Jika data yang akan dimasukkan lebih kecil atau sama dengan nilai node yang ada, maka penelusuran dilanjutkan ke anak kiri. Sebaliknya, jika data lebih besar dari node yang ada, maka penelusuran dilanjutkan ke anak kanan. Demikian seterusnya sampai ditemukan node yang terakhir (leaf). Dengan demikian, data baru ini akan dibuat sebagai anak kiri atau anak kanan dari node yang terakhir tersebut.

BST Traversal

Yang dimaksud dengan BST Traversal ini adalah menelusuri atau mengunjungi setiap node dengan aturan tertentu. Ada tiga jenis traversal yang dikenal, yaitu: (1) In Order, (2) Pre Order, dan (3) Post Order.

In Order Traversal

Prosedur *in order traversal* ini melakukan kunjungan secara rekursif: (a) pergi ke kiri sampai leaf, (b) tulis isi node, dan (c) pergi ke kanan sampai leaf. Hal ini dilakukan secara rekursif sampai semua node ditulis. Dengan demikian, implementasi fungsi anggota *in order traversal* sebagai berikut:

```
void BST::inOrderTraversal(unsigned int curr) {
   if (tree[curr].left != -1)
      inOrderTraversal(tree[curr].left);
   cout << tree[curr].value << endl;
   if (tree[curr].right != -1)
      inOrderTraversal(tree[curr].right);
}
```

Pre Order Traversal

Prosedur *pre order traversal* ini melakukan kunjungan secara rekursif: (a) tulis isi node, (b) pergi ke kiri sampai leaf, dan (c) pergi ke kanan sampai leaf. Hal ini dilakukan secara rekursif sampai semua node ditulis. Dengan demikian, implementasi fungsi anggota *pre order traversal* sebagai berikut:

```
void BST::preOrderTraversal(unsigned int curr) {
   cout << tree[curr].value << endl;
   if (tree[curr].left != -1)
      preOrderTraversal(tree[curr].left);
   if (tree[curr].right != -1)
      preOrderTraversal(tree[curr].right);
}
```

Post Order Traversal

Prosedur *post order traversal* ini melakukan kunjungan secara rekursif: (a) pergi ke kiri sampai leaf, (b) pergi ke kanan sampai leaf, dan (c) tulis isi node,. Hal ini dilakukan secara rekursif sampai semua node ditulis. Dengan demikian, implementasi fungsi anggota *post order traversal* sebagai berikut:

```
void BST::postOrderTraversal(unsigned int curr) {
   if (tree[curr].left != -1)
      postOrderTraversal(tree[curr].left);
   if (tree[curr].right != -1)
      postOrderTraversal(tree[curr].right);
   cout << tree[curr].value << endl;
}
```