University M'hamed Bougara Boumerdes

Institute of Electrical and Electronics Engineering (ex: INELEC)

Algorithms and Data structures laboratory

LAB N°4

Friday, March 18, 2022

Superviser:	**Students:**	
Dr. A. Zitouni	Madaoui Zakaria	G3 Computer
	Maallem Nassim	G3 Computer

Objectives:

- Write a C++ program that implements a binary search tree using an array and a pointer implementation
- The program should provide a way to:
    - o Initialize the BST
    - o Insert a new integer into the tree
    - o Traversing the BST using in-order traversal
    - o Traversing the BST using pre-order traversal
    - o Traversing the BST using post-order traversal

Task 1: Array implementation

A binary search tree is a special type of binary trees, such that for any given node, the value stored in its left child is less or equal to its value, and the value stored in its right child is larger than its value.

In order to implement this tree using arrays, we need to declare a struct called **node** that keeps track of the indexes of the parent node , left and right children. In addition, we need another struct called **tree** that has an array of nodes as well as it keeps track of the size of the tree in order to determine the index of the next newly added element.

Since trees are recursive by nature, we implemented recursive functions to insert elements to the tree and traverse it in the 3 different methods. And this keeps the code very simple, as it is written as stated by the definition.

- **Code:**

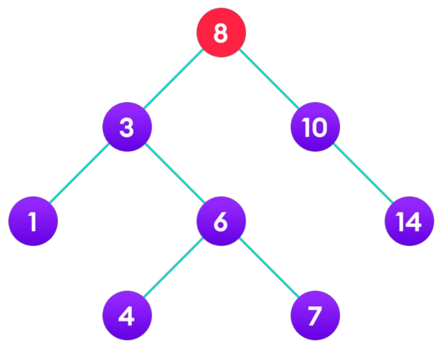
```
#include<iostream>
#define MAX_NODES 27

using namespace std;

typedef struct{
    int val;
    int left;
    int right;
    int parent;
} node;

typedef struct{
    int size;
    node nodes[MAX_NODES];
} tree;

//prototypes
void    BST_init(tree &t);
void    BST_insert(tree &t , int val);
void    BST_traverse_inOrder(tree &t);
void    BST_traverse_preOrder(tree &t);
void    BST_traverse_postOrder(tree &t);
void    printTabs(int tabs);
void    printTree(tree &t,int root, int depth);
```

```cpp
int main(){
    tree t;
    BST_init(t);

    BST_insert(t, 8 );
    BST_insert(t, 3 );
    BST_insert(t, 1 );
    BST_insert(t, 6 );
    BST_insert(t, 7 );
    BST_insert(t, 9);
    BST_insert(t, 4 );

    int choice = -1 , tempVal;
    while(choice != 0){
        cout    <<  "Choose an option:\n"
                <<  "1 + intVAL to insert a new integer\n"
                <<  "2: print tree in graphical way\n"
                <<  "3: Traverse BST using inorder traversal\n"
                <<  "4: Traverse BST using perorder traversal\n"
                <<  "5: Traverse BST using postorder traversal\n"
                <<  "6: Show tree + all traversals\n"
                <<  "0: to EXIT"
                << endl;

        cin >> choice;
        switch (choice)
        {
        case 1:
            cin >> tempVal;
            cout << "inserting  " << tempVal << endl;
            BST_insert(t,tempVal);
            break;
        case 2:
            cout <<"\n==================================\n";
            printTree(t,0,0);
            cout <<"\n==================================\n";
            break;
        case 3:
            BST_traverse_inOrder(t);
            break;
        case 4:
            BST_traverse_preOrder(t);
            break;
        case 5:
            BST_traverse_postOrder(t);
            break;
        case 6:
            cout << "\n==================================\n";
            printTree(t,0,0);
            cout << endl;
            BST_traverse_inOrder(t);
            BST_traverse_preOrder(t);
            BST_traverse_postOrder(t);
            cout <<"\n==================================\n";
            break;
        }
    }

    return 0;
}
```

```cpp
void printTabs(int tabs){
    for (int i = 0; i < tabs; i++)cout << "\t";
}
void printTree(tree &t,int root, int depth){
    if(t.size == 0){
        cout << "tree is empty !" << endl;
        return;
    }

    if(t.nodes[root].right != -1) printTree(t, t.nodes[root].right, depth + 1);
    printTabs(depth); cout << "[" <<t.nodes[root].val << "]\n" ;
    if(t.nodes[root].left != -1) printTree(t, t.nodes[root].left, depth + 1);

}

//================================ implementations ==================================

void BST_init(tree &t){
    t.size = 0;
    for (int i = 0; i < MAX_NODES; i++)
    {
        t.nodes[i].parent = t.nodes[i].left = t.nodes[i].right = -1;
    }

}

//--------------------------------------------------------------------------------

void BST_insert_rec(tree &t, int root , int val){
    /*The main loop MUST always call this with root = 0 */

    if(t.size == MAX_NODES-1){
        cout << "Can't insert, tree is full !" << endl;
    }
    //case1: tree is empty
    else if(t.size == 0){
        t.nodes[0].val = val;
        t.size++;
    }
    //case2: insert left
    else if(val <= t.nodes[root].val){
        if(t.nodes[root].left == -1){
           int index = t.size;
           t.nodes[index].val = val;      //insert value
           t.nodes[index].parent = root; //set parent
           t.nodes[root].left = index;   //update parent's left node value
           t.size++;
        }else{
            BST_insert_rec(t, t.nodes[root].left , val);
        }
    //case3: insert right
    }else{
        if(t.nodes[root].right == -1){
           int index = t.size;
           t.nodes[index].val = val;      //insert value
           t.nodes[index].parent = root; //set parent
           t.nodes[root].right = index;  //update parent's right node value
           t.size++;
        }else{
            BST_insert_rec(t, t.nodes[root].right , val);
        }
    }
}

void BST_insert(tree &t , int val){
    BST_insert_rec(t, 0 , val);
}

//---------------------------------------------------------------------------------
void BST_traverse_inOrder_rec(tree &t, int root){
    if(t.size == 0){
        cout << "tree is empty !" << endl;
        return;
    }

    if(t.nodes[root].left != -1)  BST_traverse_inOrder_rec(t, t.nodes[root].left);
    cout << "[" <<t.nodes[root].val << "] -> " ;
    if(t.nodes[root].right != -1) BST_traverse_inOrder_rec(t, t.nodes[root].right);
}

void BST_traverse_inOrder(tree &t){
    cout << "inOrder:   ";
    BST_traverse_inOrder_rec(t, 0);
    cout << "END" << endl;
}

//---------------------------------------------------------------------------------
void BST_traverse_preOrder_rec(tree &t, int root){
    if(t.size == 0){
        cout << "tree is empty !" << endl;
        return;
    }
    cout << "[" <<t.nodes[root].val << "] -> " ;
    if(t.nodes[root].left != -1)  BST_traverse_preOrder_rec(t, t.nodes[root].left);
    if(t.nodes[root].right != -1) BST_traverse_preOrder_rec(t, t.nodes[root].right);
}

void BST_traverse_preOrder(tree &t){
    cout << "preOrder:  ";
    BST_traverse_preOrder_rec(t, 0);
    cout << "END" << endl;
}

//---------------------------------------------------------------------------------
void BST_traverse_postOrder_rec(tree &t, int root){
    if(t.size == 0){
        cout << "tree is empty !" << endl;
        return;
    }
    if(t.nodes[root].left != -1)  BST_traverse_postOrder_rec(t, t.nodes[root].left);
    if(t.nodes[root].right != -1) BST_traverse_postOrder_rec(t, t.nodes[root].right);
    cout << "[" <<t.nodes[root].val << "] -> " ;
}

void BST_traverse_postOrder(tree &t){
    cout << "postOrder: ";
    BST_traverse_postOrder_rec(t, 0);
    cout << "END" << endl;
}
```

- **Output:**

Inserting integers into an empty tree in the following order: 8,3,1,6,7,10,14,4 should produce a tree as depicted by the figure below:



By running our program and inserting integers as mentionned before we get an identical output, which is as follows (note that the graphical output prints the upmost root from left and the tree branches towards the right)

```
› g++ arrayTree.cpp -o arrayTree ; ./arrayTree
Choose an option:
1 + intVAL to insert a new integer
2: print tree in graphical way
3: Traverse BST using inorder traversal
4: Traverse BST using perorder traversal
5: Traverse BST using postorder traversal
6: Show tree + all traversals
0: to EXIT
> 6
==================================
                [14]
        [10]
[8]
                        [7]
                [6]
                        [4]
        [3]
                [1]

inOrder:   [1] → [3] → [4] → [6] → [7] → [8] → [10] → [14] → END
preOrder:  [8] → [3] → [1] → [6] → [4] → [7] → [10] → [14] → END
postOrder: [1] → [4] → [7] → [6] → [3] → [14] → [10] → [8] → END
```

Task 2: Pointer implementation

The pointer implementation is even simpler than the array one, since there is no need to keep track of indexes of other nodes, but just pointers that point to them. This removes the limitation of fixed size that the array implementation has and results in a much simpler code.

The previous code was modified as follows to convert from arrays to pointers:

```
#include<iostream>

using namespace std;

typedef struct node{
    int val;
    struct node* left;
    struct node* right;
    struct node* parent;
} node;

//prototypes
void    BST_init(node* &tree);
void    BST_insert(node* &tree , int val);
void    BST_traverse_inOrder(node* tree);
void    BST_traverse_preOrder(node* tree);
void    BST_traverse_postOrder(node* tree);
void    printTabs(int tabs);
void    printTree(node* root, int depth);

int main(){
    node *tree;

    BST_init(tree);

    BST_insert(tree, 8 );
    BST_insert(tree, 3 );
    BST_insert(tree, 1 );
    BST_insert(tree, 6 );
    BST_insert(tree, 7 );
    BST_insert(tree, 10);
    BST_insert(tree, 14);
    BST_insert(tree, 4 );

    int choice = -1 , tempVal;
    while(choice != 0){
        cout    <<  "Choose an option:\n"
                <<  "1 + intVAL to insert a new integer\n"
                <<  "2: print tree in graphical way\n"
                <<  "3: Traverse BST using inorder traversal\n"
                <<  "4: Traverse BST using perorder traversal\n"
                <<  "5: Traverse BST using postorder traversal\n"
                <<  "6: Show tree + all traversals\n"
                <<  "0: to EXIT"
                << endl;

        cin >> choice;
        switch (choice)
        {
        case 1:
            cin >> tempVal;
            cout << "tempVal = " << tempVal << endl;
            BST_insert(tree,tempVal);
            break;
        case 2:
            cout <<"\n==================================\n";
            printTree(tree,0);
            cout <<"\n==================================\n";
            break;
        case 3:
            cout <<"inOrder: ";BST_traverse_inOrder(tree); cout << "END" << endl;
            break;
        case 4:
            cout <<"preOrder: ";BST_traverse_preOrder(tree); cout << "END" << endl;
            break;
        case 5:
            cout <<"postOrder: ";BST_traverse_postOrder(tree); cout << "END" << endl;
            break;
        case 6:
            cout << "\n==========================================================\n";
            printTree(tree,0);
            cout << endl;
            cout <<"inOrder: ";BST_traverse_inOrder(tree); cout << "END" << endl;
            cout <<"preOrder: ";BST_traverse_preOrder(tree); cout << "END" << endl;
            cout <<"postOrder: ";BST_traverse_postOrder(tree); cout << "END" << endl;
            cout <<"\n===========================================================\n";
            break;
        }
    }

    return 0;
}

void printTabs(int tabs){
    for (int i = 0; i < tabs; i++)cout << "\t";
}
void printTree(node* root, int depth){
    if(root == nullptr){
        cout << "tree is empty !" << endl;
        return;
    }

    if(root->right != nullptr) printTree(root->right, depth + 1);
    printTabs(depth); cout << "[" <<root->val << "]\n" ;
    if(root->left != nullptr) printTree(root->left, depth + 1);

}

//================================ implementations ==================================

void BST_init(node* &tree){
    tree = nullptr;
}

//-----------------------------------------------------------------------------------

void BST_insert(node* &root , int val){

    //case1: root is null  (empty tree)
    if(root == nullptr){
        root = new node;
        root->val = val;
        root->left = root->right = root->parent = nullptr;
    }
    //case2: insert left
    else if(val <= root->val){
        if(root->left == nullptr){
            root->left          = new node;
            root->left->val     = val;
            root->left->parent  = root;
        }else{
            BST_insert(root->left , val);
```

```cpp
        }
    //case3: insert right
    }else{
        if(root->right == nullptr){
            root->right          = new node;
            root->right->val     = val;
            root->right->parent  = root;
        }else{
            BST_insert(root->right , val);
        }
    }

}

//-----------------------------------------------------------------------------------
void BST_traverse_inOrder(node* root){
    if(root == nullptr){
        cout << "tree is empty !" << endl;
        return;
    }

    if(root->left != nullptr)  BST_traverse_inOrder(root->left);
    cout << "[" <<root->val << "] -> " ;
    if(root->right != nullptr) BST_traverse_inOrder(root->right);
}

//-----------------------------------------------------------------------------------
void BST_traverse_preOrder(node* root){
    if(root == nullptr){
        cout << "tree is empty !" << endl;
        return;
    }

    cout << "[" <<root->val << "] -> " ;
    if(root->left != nullptr)  BST_traverse_preOrder(root->left);
    if(root->right != nullptr) BST_traverse_preOrder(root->right);
}

//-----------------------------------------------------------------------------------
void BST_traverse_postOrder(node* root){
    if(root == nullptr){
        cout << "tree is empty !" << endl;
        return;
    }

    if(root->left != nullptr)  BST_traverse_postOrder(root->left);
    if(root->right != nullptr) BST_traverse_postOrder(root->right);
    cout << "[" <<root->val << "] -> " ;
}
```

* Note that the output of this code is identical to the previous one.