Practice #8

Trees (Binary Tree)

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Practice #8 TO-DO List

To-Do	Submission	Notes
Binary Tree	Screenshot and source code (BinTree.h, BinTree.cpp)	p.5-12

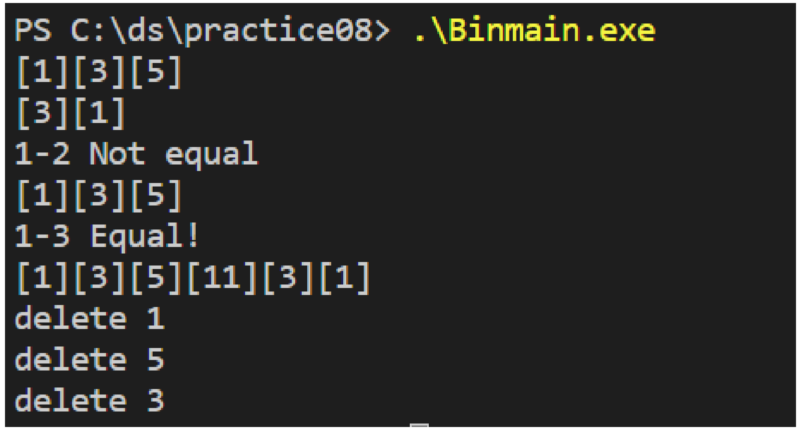
- Upload your screenshot and source codes on LMS by 11pm on 4/21 (Wed).
  - All your screenshots should be merged in one pdf file, screenshot.pdf.
  - Your pdf and all source codes should be compressed into zip file.
- File name: practice08_Your Student ID_Name.zip (only zip, not pdf, docx, c, etc)
  - ex) practice08_20400022_고윤민.zip

Binary Tree

- Implement a binary tree class
  - Write BinTree.h and BinTree.cpp (Binmain.cpp: no need to change)
  - Refer to p.5-12 (in this slides)
  - Implement following member functions
    - IsEmpty(): check empty
    - MakeBT() make binary tree with root, left subtree of root, and right subtree of root
    - Lchild(): return left child
    - Rchild(): return right child
    - insert_node_left(): insert a new node at leftmost leaf
    - insert_node_right(): insert a new node at rightmost leaf
    - delete_tree(): delete node from binary tree
    - inorder(): inorder traversal
    - Print(): print the node
    - equal(): testing for equality of binary trees
    - copy(): copying binary tree

Binary Tree

- Expected results



```
PS C:\ds\practice08> .\Binmain.exe
[1][3][5]
[3][1]
1-2 Not equal
[1][3][5]
1-3 Equal!
[1][3][5][11][3][1]
delete 1
delete 5
delete 3
```

Binmain.cpp

```
#include <iostream>
#include "BinTree.h"
using namespace std;
int main() {
    BinTree* tree1 = new BinTree();
    BinTree* tree2 = new BinTree();
    BinTree* tree3 = new BinTree();
    BinTree* tree4 = new BinTree();

    tree1->insert_node_left(3);
    tree1->insert_node_left(1);
    tree1->insert_node_right(5);
    tree1->Print();

    tree2->insert_node_right(3);
    tree2->insert_node_right(1);
    tree2->Print();
    if (tree1->equal(tree2)) cout << "1-2 Equal!" << endl;
    else cout << "1-2 Not equal" << endl;

    tree3->copy(tree1);
    tree3->Print();
    if (tree1->equal(tree3)) cout << "1-3 Equal!" << endl;
    else cout << "1-3 Not equal" << endl;

    tree4->MakeBT(tree1, 11, tree2);
    tree4->Print();

    delete tree1;
}
```

BinTree.h

```
typedef struct node {
    int data;
    struct node *left_child, *right_child;
} tree_node;

class BinTree{
  private:
    tree_node* root;
  public:
    BinTree();
    ~BinTree();
    bool IsEmpty();
    void MakeBT(BinTree *b1, int item, BinTree *b2);
    tree_node* Lchild(tree_node *bt){return bt->left_child;};
    tree_node* Rchild(tree_node *bt){return bt->right_child;};
    void insert_node_left(int data);
    void insert_node_right(int data);
    // tree_node* modified_search(int data);
    // void insert_node(int data);
    void delete_tree(tree_node* tree);
    void inorder(tree_node* ptr);
    void Print();
    bool equal(tree_node *first, tree_node *second);
    bool equal(BinTree* second);
    tree_node* copy(tree_node* original);
    void copy(BinTree* source);
};
```

In this practice, we don’t implement modified_search() and insert_node().

BinTree.cpp

```
BinTree::BinTree()
{
    root = NULL;
}

BinTree::~BinTree()
{
    delete_tree(root);
    root = NULL;
}

void BinTree::delete_tree(tree_node* ptr)
{
    if (ptr) {
        delete_tree(ptr->left_child);
        delete_tree(ptr->right_child);
        cout << "delete " << ptr->data << endl;
        delete(ptr);
    }
}

bool BinTree::IsEmpty()
{
   return (root == NULL);
}
```

BinTree.cpp

Practice

```
void BinTree::insert_node_left(int num)
{
    tree_node *ptr = new tree_node;
    ptr->data = num;
    ptr->left_child = ptr->right_child = NULL;

    if(IsEmpty()) root = ptr;
    else{
        tree_node *cur = root;
        for(;cur->left_child;cur=cur->left_child);
        cur->left_child = ptr;
    }
}

void BinTree::insert_node_right(int num)
{
    tree_node *ptr = new tree_node;
    ptr->data = num;
    ptr->left_child = ptr->right_child = NULL;

    if(IsEmpty()) root = ptr;
    else{
        tree_node *cur = root;
        for(;cur->right_child;cur=cur->right_child);
        cur->right_child = ptr;
    }
}
```

BinTree.cpp

```
void BinTree::inorder(tree_node* ptr)
{
   if (ptr) {
      inorder(ptr->left_child);
      cout << "[" << ptr->data << "]";
      inorder(ptr->right_child);
   }
}

void BinTree::Print()
{
    inorder(root);
    cout << endl;
}
```

BinTree.cpp

```
bool BinTree::equal(tree_node* first, tree_node* second)
{
      return ((!first && !second)
             ||(first && second && (first->data == second->data) &&
             equal(first->left_child, second->left_child) &&
             equal(first->right_child, second->right_child)));
}

bool BinTree::equal(BinTree* second)
{
   return equal(root, second->root);
}
```

BinTree.cpp

Practice

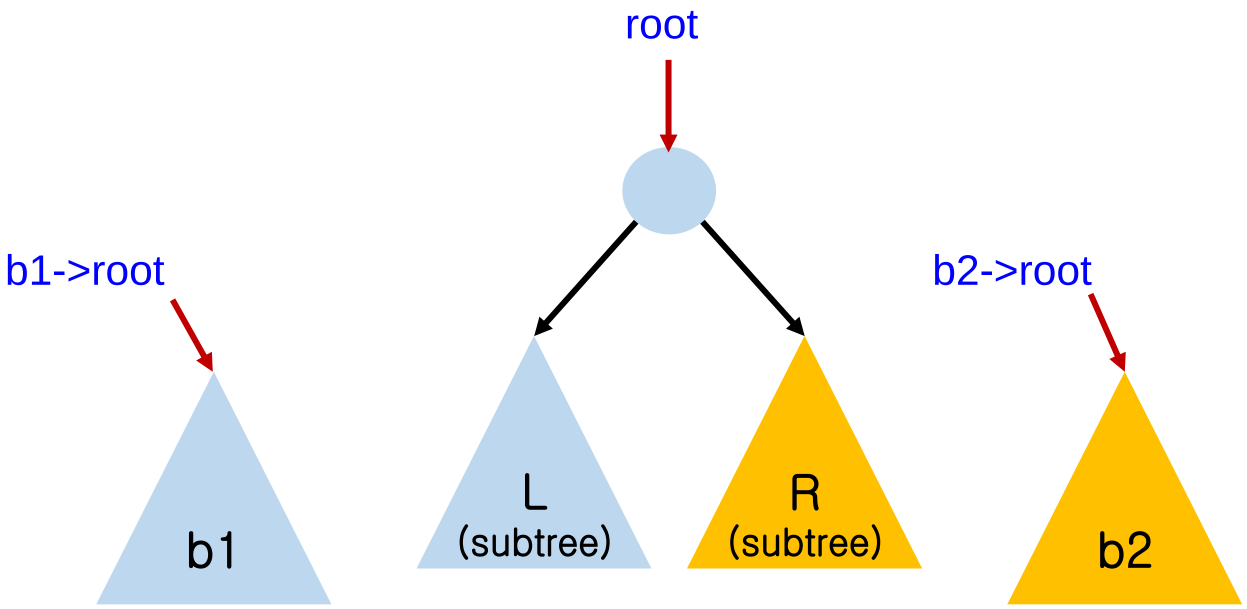
```
tree_node* BinTree::copy(tree_node* original)
{
   if(original){
        tree_node* temp = new tree_node;
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return temp;
   }
   return NULL;
}

void BinTree::copy(BinTree *source)
{
    root = copy(source->root);
}
```

Practice

BinTree.cpp

```
void BinTree::MakeBT(BinTree *b1, int num, BinTree *b2)
{
    root = new tree_node;
    root->left_child = copy(b1->root);
    root->right_child = copy(b2->root);
    root->data = num;
}
```


root
b1->root
b2->root
b1
L
(subtree)
R
(subtree)
b2