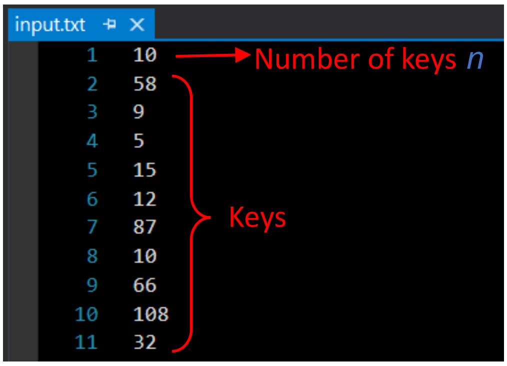
Binary Search Tree

Introduction

This program is an implementation of Binary Search Tree (BST). The main function reads the node keys from input file *input.txt* and construct the corresponding BST. Run *main.cpp* in Visual Studio to construct the Binary Search Tree and do further operations by the instructions shown in the window. Note that all the files (including input file, *.h and *.cpp) have to be placed under the same directory.

Input

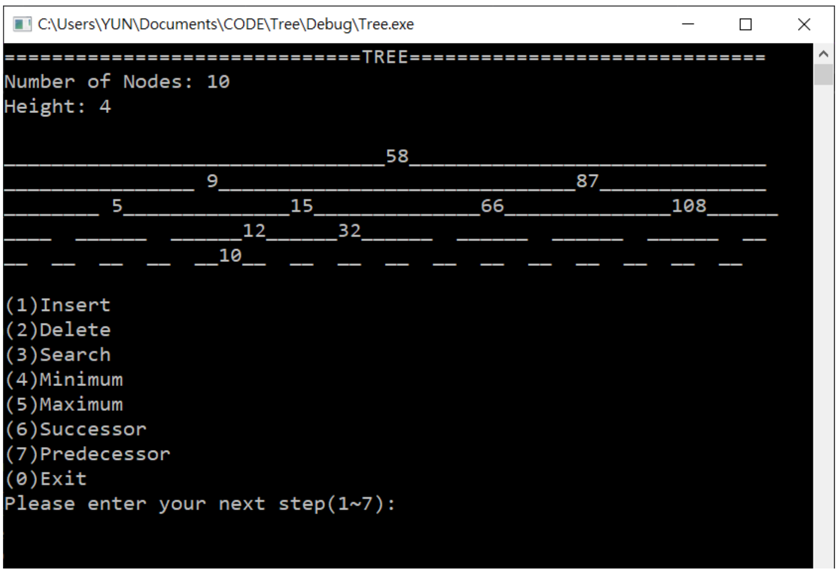
The input file *input.txt* has the number of keys *n* as the first line, and followed by *n* keys in separate lines.



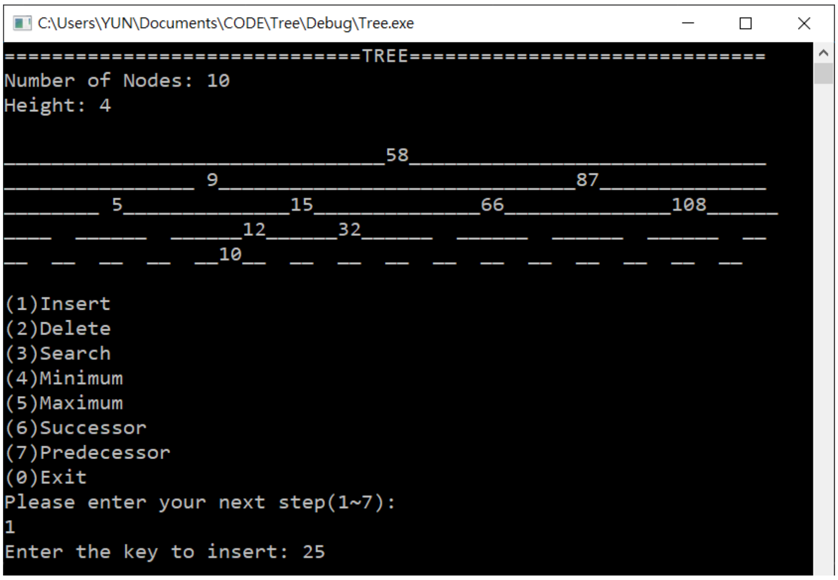
Test

The following are some examples of running *main.cpp*.

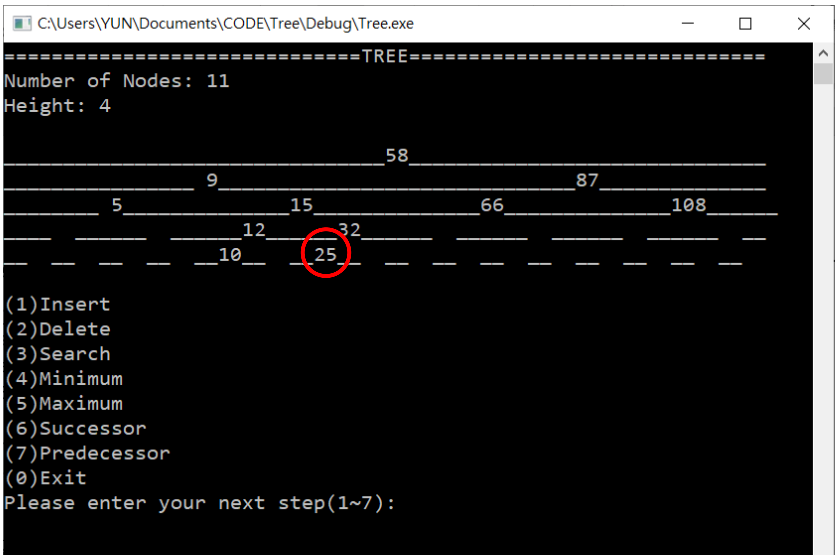
1. Run



2. Insert: key 25



Result:



Concept

1. *TreeNode* Class

   This is a class that interpret tree nodes as objects. Each instance of *TreeNode* contains the following attributes:

Members	Description
TreeNode* *parent*	A pointer that points to its parent. (Default: null pointer)
TreeNode* *leftchild*	A pointer that points to its left child. (Default: null pointer)
TreeNode* *rightchild*	A pointer that points to its right child. (Default: null pointer)
int *key*	The key of the node.

	(Default value: 0)
int *address*	The address of the node in the tree. (Default value: 1→the root)
int *depth*	The depth of the node in the tree. (Default value: 0→the root)
Functions	
printInfo	Prints the information (the members mentioned above) of the node.

2. *Tree* Class

   This is the class of BST as objects, which uses *TreeNode* as its basic structure to construct the tree. Each instance of *Tree* contains the following attributes:

Members	Description
int *height*	The height of the tree. (Default value: 0)
int *num_of_nodes*	The number of nodes in the tree. (Default value: 0)
TreeNode* *root*	A pointer that points to the root. (Default: null pointer)
Functions	
BFS	Does BFS for the sake of printing the tree.
Setup	Setup the address and depth for all nodes, and also the height of the tree. Used when tree is modified in *Delete*.
printTree	Prints the tree, its height, and the number of nodes.
Insert	Insert a new node with a specified key into the tree.
Transplant	Replace one node with another node.
Delete	Deletes the node with a particular key from the tree.
IterativeTreeSearch	Finds the node with a particular key in the tree.
Minimum	Finds the node with the minimum key in a subtree.
Maximum	Finds the node with the maximum key in a subtree.
Successor	Finds the successor of a particular node.
Predecessor	Finds the predecessor of a particular node.

3. Explanation for the main operation functions:

(1) *Insert* function

i. Upon being called, the function receives a new *TreeNode* z* whose key has already been assigned.

ii. If the tree is empty, there will be no root (i.e. *root = nullptr*), so let *z* be the root.

iii. If the tree is not empty:

(a) Starting from the root, compare its key with *z*.

(b) If *z* is smaller, then move on to compare with the root's left child, otherwise compare with its right child.

(c) Keep on comparing until the node to compare is null pointer, then we have found the correct position for *z*.

(d) Setup the address and depth of *z* according to that of its parent.

(2) *Delete* function

To delete a node with a particular key, we use *IterativeTreeSearch* to get that node before calling *Delete*.

i. Upon being called, the function receives the node to delete.

ii. Examine which of the following operations should be done:

(a) Case 1: *z* has only one child or no child

→ Just use *Transplant* to replace *z* with its child (or NIL if no child).

(b) Case 2: *z* has two children

→ Since all the nodes in *z*'s right subtree are larger than *z*, we search for the smallest node among them to replace *z*, so that the rule of BST can be maintained.

iii. After the replacement is done, delete *z* by calling the destructor of *TreeNode*, so as to deallocate memory.

iv. Call *Setup* to adjust the address, depth of the nodes, and the height of the tree after deletion.

(3) *IterativeTreeSearch* function

(The idea is similar to *Insert*.)

i. The function takes two arguments:

(a) *TreeNode* x*: The root of the subtree to search for *k*.

(b) int *k*: The key to search.

ii. Starting from *x*, compare its key with *k*.

iii. If *x* happens to possess the key same as *k*, then return *x*.

iv. If *k* is smaller, then move on to compare with *x*'s left child, otherwise compare with its right child.

v. Keep on comparing in the while-loop until (d) is satisfied. Otherwise, the loop ends when the node to compare is null pointer, which means the key does not exists, then a null pointer is returned.

(4) *Minimum* function

For a given *TreeNode** *x*, the function finds the minimum node in the subtree rooted at *x*. This is done by iterating through every node's left child. Until the next left child is NIL, then we have found the minimum in this subtree.

(5) *Maximum* function

For a given *TreeNode** *x*, the function finds the maximum node in the subtree rooted at *x*. This is done by iterating through every node's right child. Until the next right child is NIL, then we have found the minimum in this subtree.

(6) *Successor* function

i. The function finds the Inorder successor of a given *TreeNode** *x*.

ii. Case 1: *x* has right child

→Find the minimum node in its right subtree with *Minimum*.

iii. Case 2: *x* has no right child

→This means that its successor is located at the upper levels, somewhere where a node is larger than its parent, which means that it is a right child. Thus, starting from *x*'s parent, we iterate through the parents until a node satisfies the condition, and then return that node.

(7) *Predecessor* function

i. The function looks for the Inorder predecessor of a given *TreeNode** *x*.

ii. Case 1: *x* has left child

→Find the maximum node in its right subtree with *Maximum*.

iii. Case 2: *x* has no left child

→This means that its predecessor is located at the upper levels, somewhere where a node is smaller than its parent, which means that it is a left child. Thus, starting from *x*'s parent, we iterate through the parents until a node satisfies the condition, and then return that node.