

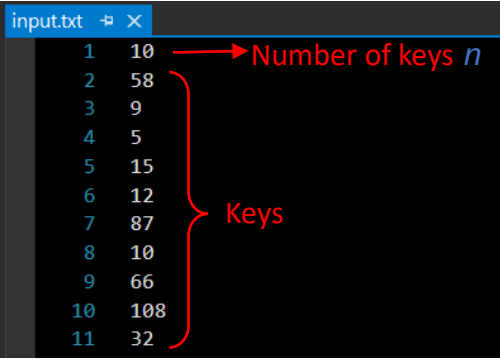
Binary Search Tree

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.



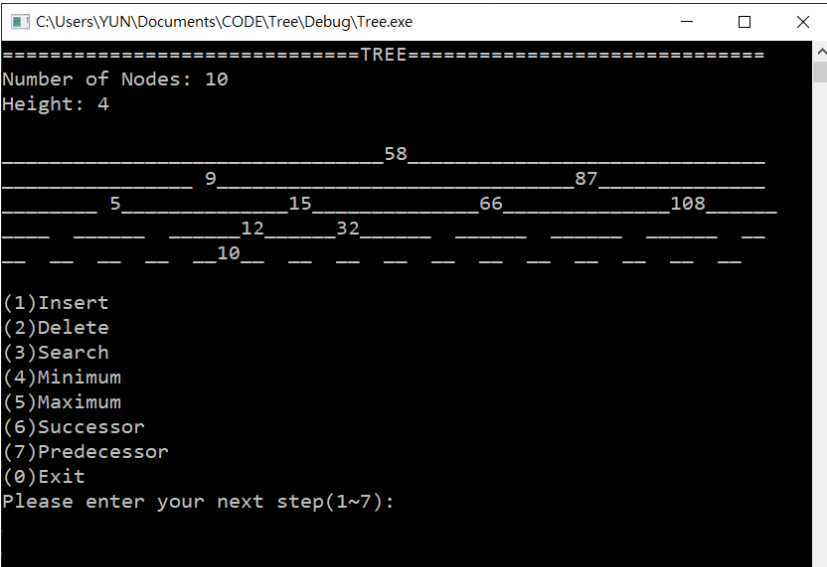
A screenshot of a text editor window titled 'input.txt'. The window contains 11 lines of text. The first line is '10'. The following 10 lines are pairs of numbers: '2 58', '3 9', '4 5', '5 15', '6 12', '7 87', '8 10', '9 66', '10 108', and '11 32'. A red arrow points from the text 'Number of keys n' to the first line '10'. A red bracket on the right side of the window groups the last 10 lines, with the label 'Keys' next to it.

Line	Key
1	10
2	58
3	9
4	5
5	15
6	12
7	87
8	10
9	66
10	108
11	32

Test

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

1. Run



A screenshot of a Windows command prompt window titled 'C:\Users\YUN\Documents\CODE\Tree\Debug\Tree.exe'. The window displays the output of the program. It starts with '=====TREE====='. Below this, it shows 'Number of Nodes: 10' and 'Height: 4'. Then, it displays a binary search tree structure using a level-by-level layout. The root node is 58. Its left child is 9, and its right child is 87. Node 9 has a left child 5 and a right child 15. Node 87 has a left child 66 and a right child 108. Node 5 has a left child 12 and a right child 32. Node 12 has a left child 10. Below the tree structure, there is a list of menu options: (1)Insert, (2)Delete, (3)Search, (4)Minimum, (5)Maximum, (6)Successor, (7)Predecessor, and (0)Exit. The prompt 'Please enter your next step(1~7):' is shown at the bottom.

```
=====TREE=====
Number of Nodes: 10
Height: 4

          58
        /  \
       9    87
      / \  / \
     5 15 66 108
    / \ / \
   12 32
  / 
 10

(1)Insert
(2)Delete
(3)Search
(4)Minimum
(5)Maximum
(6)Successor
(7)Predecessor
(0)Exit
Please enter your next step(1~7):
```

2. Insert: key 25

```
C:\Users\YUN\Documents\CODE\Tree\Debug\Tree.exe
=====TREE=====
Number of Nodes: 10
Height: 4

      58
     /  \
    9    87
   / \  / \
  5 15 66 108
 / \ / \
12 32
/ \
10

(1)Insert
(2)Delete
(3)Search
(4)Minimum
(5)Maximum
(6)Successor
(7)Predecessor
(0)Exit
Please enter your next step(1~7):
1
Enter the key to insert: 25
```

Result:

```
C:\Users\YUN\Documents\CODE\Tree\Debug\Tree.exe
=====TREE=====
Number of Nodes: 11
Height: 4

      58
     /  \
    9    87
   / \  / \
  5 15 66 108
 / \ / \
12 32
/ \
10 25
```

Concept

1. *TreeNode* Class

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

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

	(Default value: 0)
int <i>address</i>	The address of the node in the tree. (Default value: 1→the root)
int <i>depth</i>	The depth of the node in the tree. (Default value: 0→the root)
Functions	
<i>printInfo</i>	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 <i>height</i>	The height of the tree. (Default value: 0)
int <i>num_of_nodes</i>	The number of nodes in the tree. (Default value: 0)
TreeNode* <i>root</i>	A pointer that points to the root. (Default: null pointer)
Functions	
<i>BFS</i>	Does BFS for the sake of printing the tree.
<i>Setup</i>	Setup the address and depth for all nodes, and also the height of the tree. Used when tree is modified in <i>Delete</i> .
<i>printTree</i>	Prints the tree, its height, and the number of nodes.
<i>Insert</i>	Insert a new node with a specified key into the tree.
<i>Transplant</i>	Replace one node with another node.
<i>Delete</i>	Deletes the node with a particular key from the tree.
<i>IterativeTreeSearch</i>	Finds the node with a particular key in the tree.
<i>Minimum</i>	Finds the node with the minimum key in a subtree.
<i>Maximum</i>	Finds the node with the maximum key in a subtree.
<i>Successor</i>	Finds the successor of a particular node.
<i>Predecessor</i>	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 exist, 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 left 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.

. Divide the n elements of the input array into $b_n = 5c$ groups of 5 elements each and at most one group made up of the remaining $n \bmod 5$ elements. 2. Find the median of each of the $d_n = 5e$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements. 3. Use SELECT recursively to find the median x of the $d_n = 5e$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.) 4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the k th smallest element and there are n_k elements on the high side of the partition. 5. If $i \leq k$, then return x . Otherwise, use SELECT recursively to find the i th smallest element on the low side if $i \leq k$.