The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

Lab on BT

Table of Contents

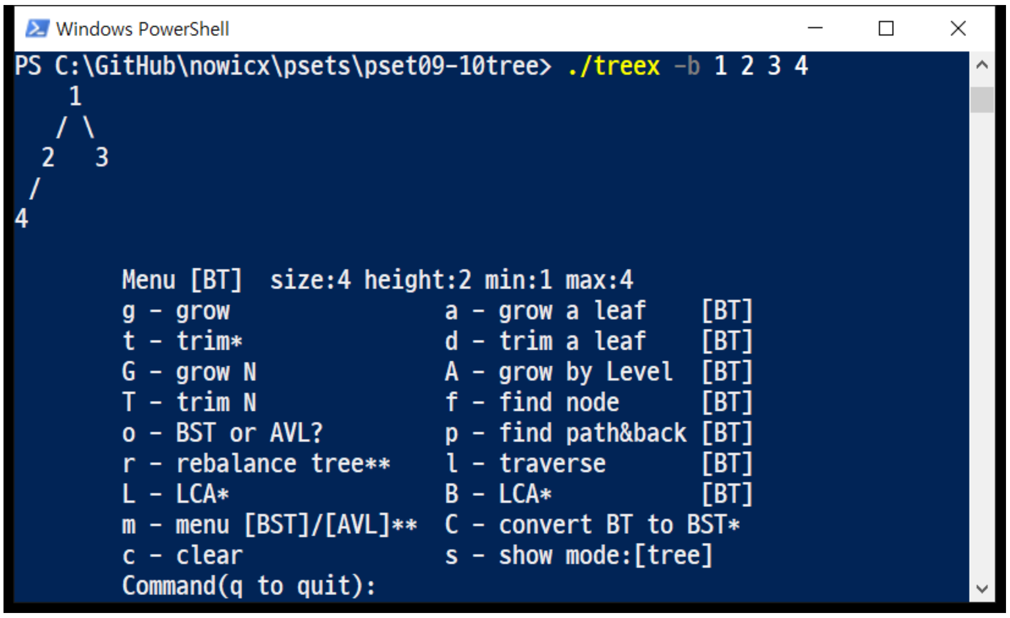
Introduction

This problem set consists of three sets of problems but they are closely related each other. Your task is to complete functions to handle a binary tree(BT), the binary search tree(BST), and AVL tree in tree.cpp, which allow the user test the binary search tree interactively. The following files are provided.

- **treeDriver.cpp**	: tests BT/BST/AVL tree implementation interactively. don't change this file.
- **tree.cpp**	: provided it as a skeleton code for your BST/AVL tree implementations.
- treenode.h	: defines the basic tree structure, and the key data type
- tree.h	: defines ADTs for BT, BST and AVL tree. don't change this file
- treeprint.cpp	: draws the tree on console
- treex.exe	: provided it as a sample solution for your reference.

Your program is supposed to work like treex.exe provided. I expect that your tree.cpp must be compatible with tree.h and treeDriver.cpp. Therefore, you don't change signatures and return types of the functions in tree.h and tree.cpp files.

The function **build_tree_by_args()** in treeDriver.cpp gets the command arguments and builds a **BT, BST or AVL** tree as shown above. If no argument for tree is provided, it begins with BT by default.



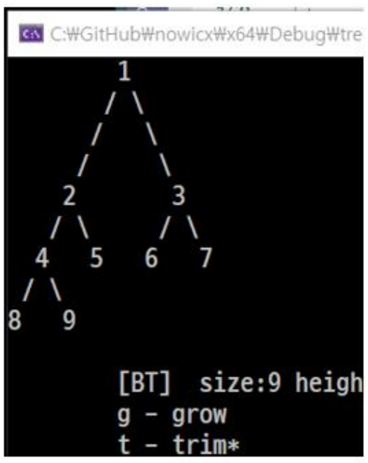
```
PS C:\GitHub\nowicx\psets\pset09-10tree> ./treex -b 1 2 3 4
    1
   / \
  2   3
 /
4

        Menu [BT]  size:4 height:2 min:1 max:4
        g - grow                a - grow a leaf    [BT]
        t - trim*               d - trim a leaf    [BT]
        G - grow N              A - grow by Level  [BT]
        T - trim N              f - find node      [BT]
        o - BST or AVL?         p - find path&back [BT]
        r - rebalance tree**    l - traverse       [BT]
        L - LCA*                B - LCA*           [BT]
        m - menu [BST]/[AVL]**  C - convert BT to BST*
        c - clear               s - show mode:[tree]
        Command(q to quit):
```

With the following three different options you can get three different trees created automatically at the beginning of the tree program execution.

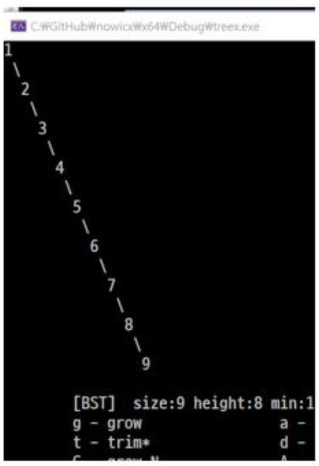
`./treex -b  1 2 3 4 5 6 7 8 9`

`./treex -s  1 2 3 4 5 6 7 8 9`

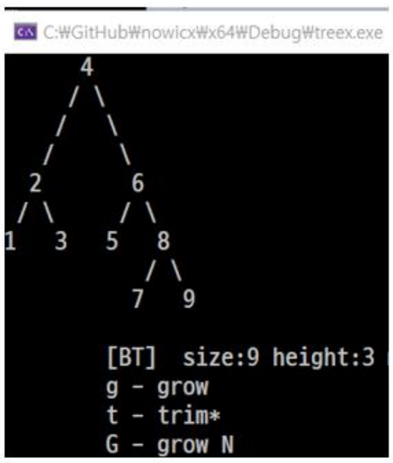
`./treex -a 1 2 3 4 5 6 7 8 9`



```
        1
       / \
      /   \
     /     \
    2       3
   / \     / \
  4   5   6   7
 / \
8   9

        [BT]  size:9 heigh
        g - grow
        t - trim*
```



```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
           \
            7
             \
              8
               \
                9

        [BST]  size:9 height:8 min:1
        g - grow                a -
        t - trim*               d -
```



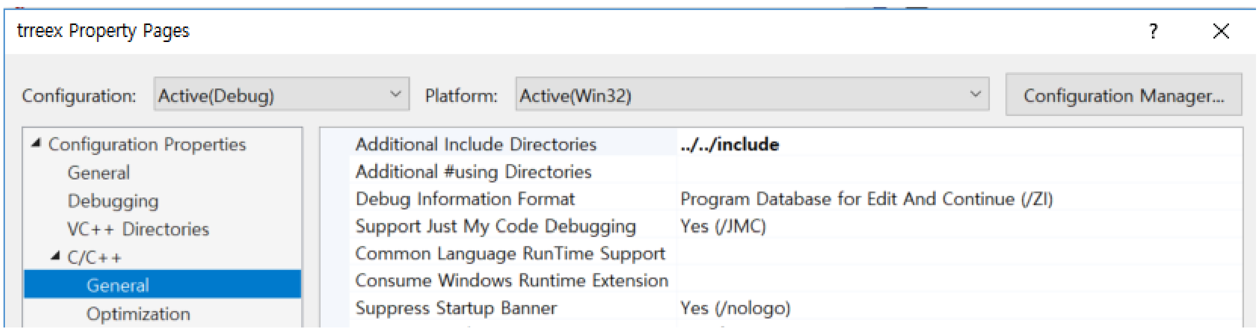
```
      4
     / \
    /   \
   /     \
  2       6
 / \     / \
1   3   5   8
           / \
          7   9

        [BT]  size:9 height:3
        g - grow
        t - trim*
        G - grow N
```

JumpStart

For a jump-start, create a project called tree first. As usual, do the following:

- Add ~/include at
  - Project Property → C/C++ → General → Additional Include Directories
- Add ~/lib at
  - Project Property → Linker → General → Additional Library Directories
- Add nowic.lib at
  - Project Property → Linker → Input → Additional Dependencies
- Add /D "DEBUG" at
  - Project Property → C/C++ → Command Line

In my case for example:

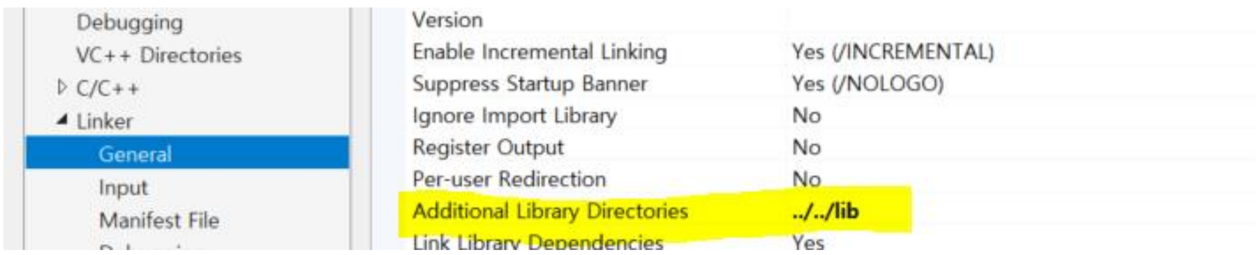


treex Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

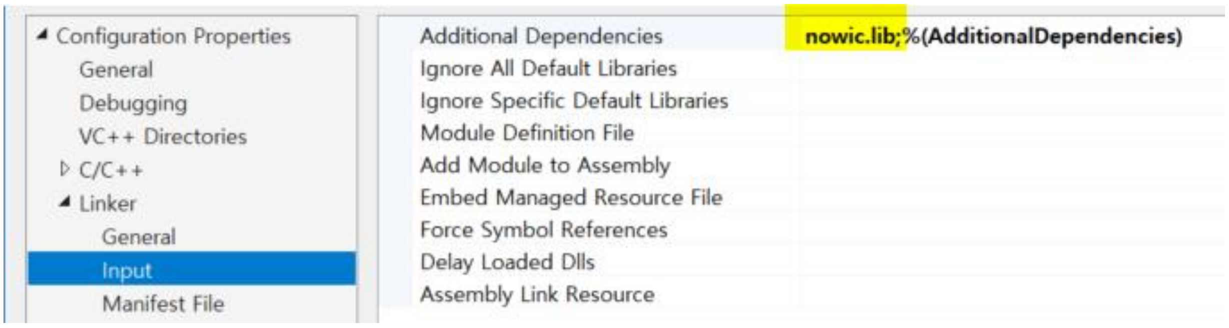
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization

Additional Include Directories	../../include
Additional #using Directories	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Support Just My Code Debugging	Yes (/JMC)
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)



- Debugging
- VC++ Directories
- C/C++
- Linker
  - General
  - Input
  - Manifest File

Version	
Enable Incremental Linking	Yes (/INCREMENTAL)
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	../../lib
Link Library Dependencies	Yes



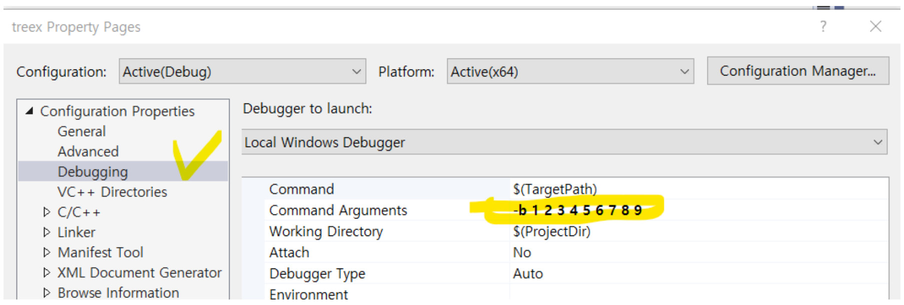
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File

Additional Dependencies	nowic.lib;%(AdditionalDependencies)
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Add ~.h files under Project 'Header Files' and ~.cpp files under project 'Source Files'. Then you may be able to build the project.

Step 0: An easy way to create a tree for debugging

Quite often we want to create a same tree every time for debugging purpose initially. To have a tree to begin with, you may specify the initial keys for the tree in
Project Properties → Debugging → Command Argument



treex Property Pages

Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information

Debugger to launch:

Local Windows Debugger

Command	$(TargetPath)
Command Arguments	-b 1 2 3 4 5 6 7 8 9
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	

Step 1.1: BT basic operations

Some functions in the tree.cpp are already implemented. You are required to implement the following ones. Feel free to make any extra helper functions, especially for recursion, as necessary. Ideally, all your code for this Problem Set goes into tree.cpp.

The menu items with [BT] usually works for all three types of the tree. But it may be slow. For example: find(), findPath(), findPathBack(), maximum() and minimum(), and so forth. Test the following functions and make sure that they are working correctly and get familiar with the development environment. There is a good reference for the tree recursion in Korean.

- clear()
- size()
- height()
- minimumBT(), maximumBT();
- containsBT(), findBT()
- inorder(), preorder(), postorder()

Step 1.2: levelorder() – iteration version

This traversal visits every node on a level before going to a lower level. This search is referred to as breadth-first search (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth. This will require space proportional to the maximum number of nodes at a given depth. This can be as much as the total number of nodes / 2.

Algorithm (Iteration):

- Create empty queue and push root node to it.
- Do the following while the queue is not empty.
  - Pop a node from queue and print/save it.
  - Push left child of popped node to queue if not null.
  - Push right child of popped node to queue if not null.

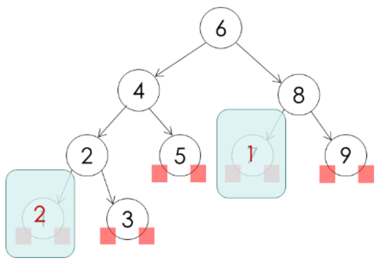
```
// level order traversal of a given binary tree using iteration.
void levelorder(tree root, vector<int>& vec) {
   Visit the root.
   if it is not null, push it to queue.
   while queue is not empty
     queue.front() - get the node from the queue
     visit the node (save the key in vec).
     if its left child is not null, push it to queue.
     if its right child is not null, push it to queue.
     queue.pop() - remove the node in the queue.
   }
}
```

Once you understand and implement the level order traversal, you will find that the next function growBT() is another variation of levelorder() function.

Step 1.3: growBT() – add a node by level order

This function inserts a node with the key and returns the root of the binary tree.

For example, if a tree shown below, the first empty node add is the node 8's left child. The next empty node is the node 2's left child.



The idea is to do iterative level order traversal of the given tree using queue. You may use the following algorithm if necessary.

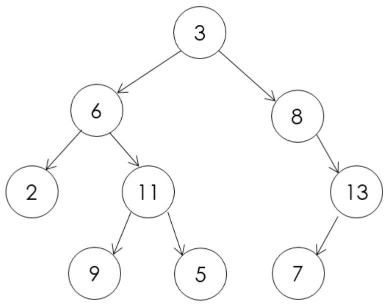
```
First, push the root to the queue.
Then, while the queue is not empty,
    Get the front() node on the queue
    If the left child of the node is empty,
       make new key as left child of the node. - break and return;
    else
       add it to queue to process later since it is not nullptr.
    If the right child is empty,
       make new key as right child of the node. - break and return;
    else
       add it to queue to process later since it is not nullptr.
    Make sure that you pop the queue finished.
    Do this until you find a node whose either left or right is empty.
```

Your code may begin as shown below:

```
tree growBT(tree root, int key) {
  if (root == nullptr)
    return new TreeNode(key);
  queue<tree> q;
  q.push(root);
  while (!q.empty()) {
    // your code here
  }
  return root; // returns the root node
}
```

Step 1.4: findPath() & findPathBack()

findPath(): Given a binary tree with unique keys, return the path from root to a given node x. For example: the path for the node 2 is [3, 6, 2], the node 9 → [3, 6, 11, 9], the node 13 → [3, 8, 13].



Intuition:
Push the current node to the vector (path). If the current node is x, return true. Go down the tree left

and right to search x, recursively. If x is found, return true. If not found, remove the current node. This algorithm comes from the concept of preorder().

Algorithm:

- If **root = nullptr**, return false. [base case]
- Push the root's key into **vector**.
- If **root's key = x**, return true.
- Recursively, look for x in root's left or right subtree.
    - If it node with **x** exist in root's left or right subtree, return true.
      else remove root's key from **vector** and return false.

findPathBack(): Using the similar algorithm, you can find a path back to the root.

Intuition:
If the current node is x or x is found while searching the tree left and right, recursively, then push the current node to vector. If x is not found, return false. This algorithm comes from the concept of posorder(). It traces back to the root after it finds the node x.

Algorithm:

- If **root = nullptr**, return false. [base case]
- If **root's key = x** or if it node **x** exists in root's left or right subtree during recursive search,
    - Push the root's key into **vector**. (recursive back-trace happens here)
    - Return true.
- Else
    - return false.

Note: Two functionalities should be coded independently. One function should NOT call the other one and reverse it.

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
- On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
- Signed: _____________ Student Number: _________________
- Make sure your code **compiles** and **runs** right before you submit it.
- If you only manage to work out the problem partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

- **Lab - Binary Tree:** Step 1.1 ~ 1.4
    - tree.cpp

References

1. Recursion :
2. Recursion: