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 BST

Table of Contents

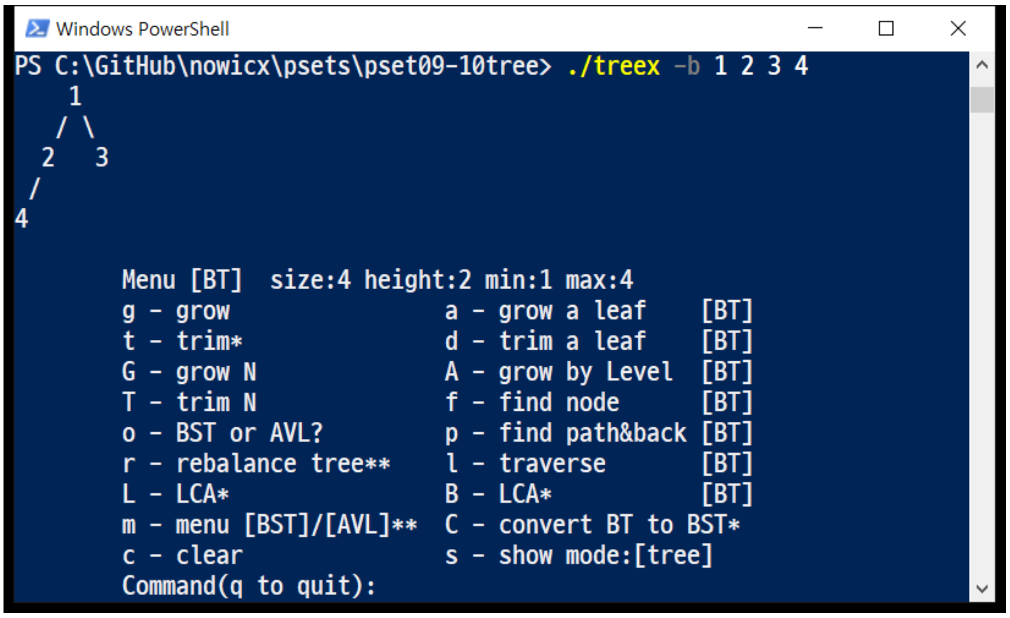
Introduction

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.

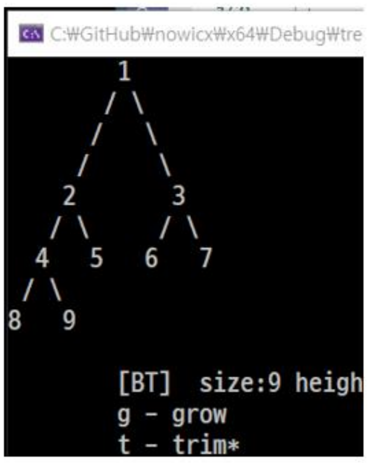
The function **build_tree_by_args()** in treeDriver.cpp gets the command arguments and builds a **BT, BST or AVL** tree as shown above. If no argument for tree is provided, it begins with BT by default.



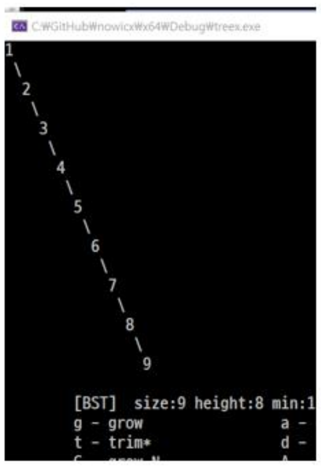
```
PS C:\GitHub\nowicx\psets\pset09-10tree> ./treex -b 1 2 3 4
     1
    / \
   2   3
  /
4

          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.

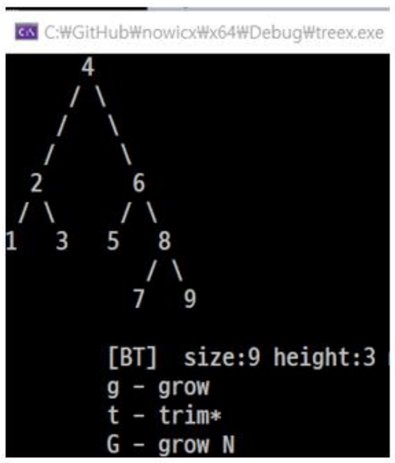
```
./treex -b  1 2 3 4 5 6 7 8 9

./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 -
```



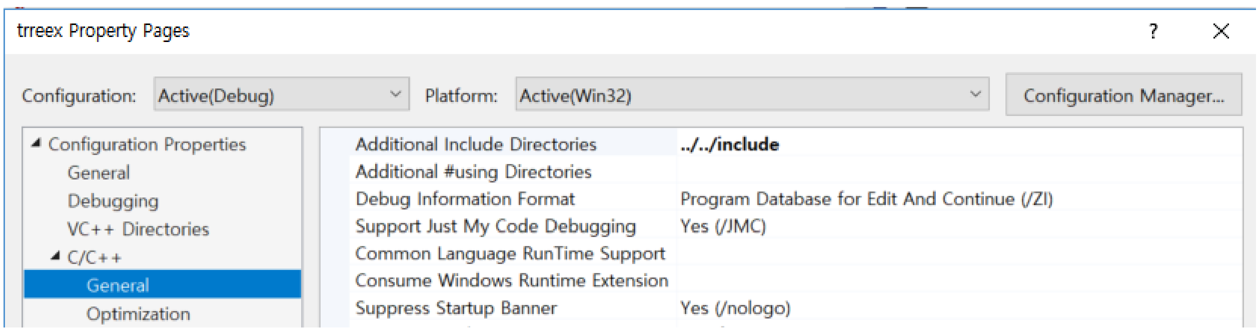
```
      4
     / \
    /   \
   /     \
  2       6
 / \     / \
1   3   5   8
           / \
          7   9

        [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:

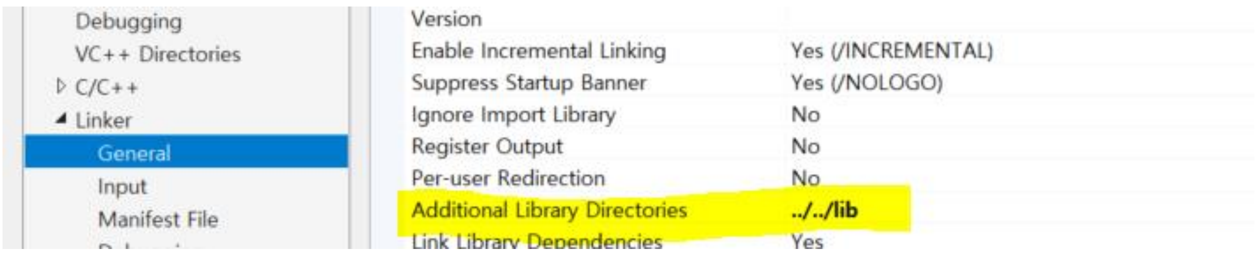


treex Property Pages

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

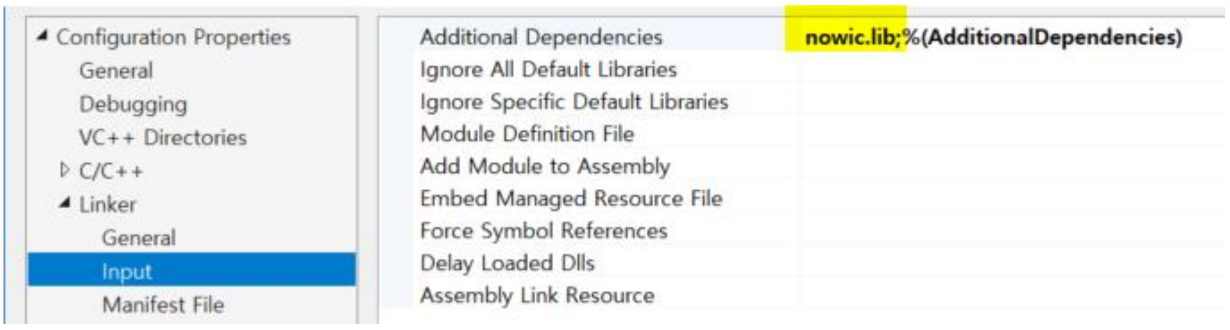
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization

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



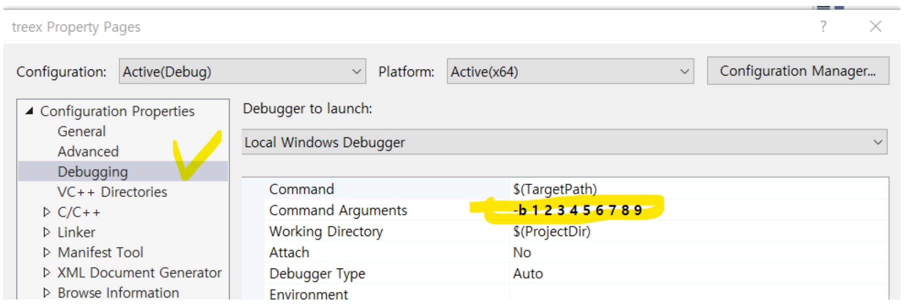
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File

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 2.1: Binary search tree operations:

There are a few additional functions for BST. You check and code them if not working

- pred(), succ() – returns predecessor node and successor node of a tree.
- isBST() – returns true if the tree is a binary search tree, otherwise false.
- minimum(), maximum()
- contains()
- find()

Step 2.2: grow() & trim()

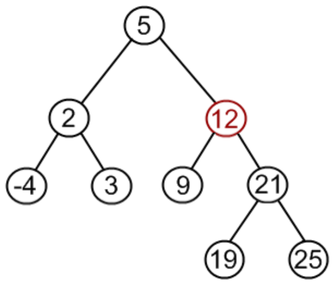
In this step, implement the basic functions grow() and trim(). Since we covered grow() during class, let us go the trim() function once more. The trim (or delete) operation on binary search tree is more complicated than insertion (or grow) and search. Basically, in can be divided into two stages:

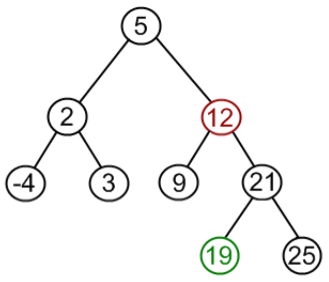
- Search for a node to remove **recursively**; for example:
```
if (key < node->key)
    node->left = trim(node->left, key);
```
- Eventually, this **node→left** will be set by the return value when **trim(node→left, key)** is done.
- If the node is found, run trim algorithm **recursively**.

When we trim a node, three possible cases arise. Once it trims the node, it must return nullptr (if it is a leaf) or the node that is replaced by. This returned pointer is eventually set to either **key→left** or **key→right** of the parent node, as in **node→left = trim(node→left, key);**

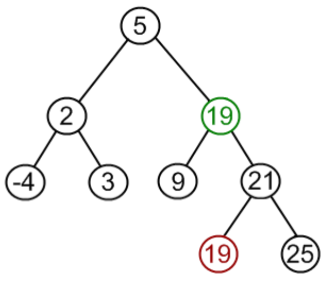
- Node to trim is leaf (or has no children):
  - Simply remove from the tree. Algorithm sets corresponding link of the parent to **nullptr** and disposes the node. **Therefore, it returns nullptr.**
- Node to trim has only one child:
  - Copy the node to a temp node.
  - Set the node to the child.
  - Free the temp node(or the original node to be trimmed).
  - Recursive trim() links this child (with it's subtree) directly to the parent of the removed node. **This is done by returning this child (node).**
- Node to trim has two children:
  - Get heights of two subtrees to determine which one use, either predecessor or successor, (Read "Note" at the end of this section for detail.)
  - Copy the key value of the **successor or predecessor** to the node.
  - Call **trim("Node to trim"'s right child, succ()'s key)** if the successor was chosen.
    Call **trim("Node to trim"'s left child, pred()'s key)** if the successor was chosen.

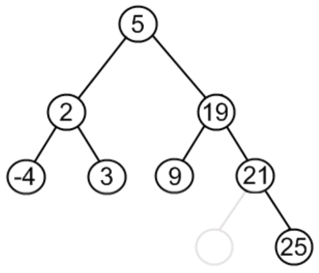
Example: Remove 12 from a BST.





1. Find successor (since the right subtree is higher than that of the left subtree) of the node to be trimmed. In current example it is 19.
2. Replace 12 with 19. Notice, that only values are replaced, not nodes.
   Now we have two nodes with the same value.





3. Remove the original node 19 from the left subtree by calling **trim()** recursively. What will be input parameters to delete the node 19? Of course, the key should be 19 which is successor. **How about the node to pass?** It should be **node→right**.
   This step may be done simply calling another trim(): node→right = trim(node→right, 19);

Note: Which one to use, Successor or Predecessor?

Once you make the trim option successfully using the successor of the node, now you are ready to improve it. When you keep on deleting a node using the successor, the tree tends to be skewed since the node will be trimmed from the right subtree.

You must make your **trim()** function such that it checks the heights of the left subtree and right subtree and decide whether you use either the predecessor or the successor in your trim operation to balance the tree if possible.

Step 2.3: growN() & trimN()

It performs a user specified number of insertion(or grow) or deletion(or trim) of nodes in the tree. The growN() function which is provided for your reference inserts a user specified number N of nodes in the tree. If it is an empty tree, the value of keys to add ranges from 0 to N-1. If there are some existing nodes in the tree, the value of keys to add ranges from max + 1 to max + 1 + N, where max is the maximum value of keys in the tree.

Implement the trimN() function which deletes a user specified number N of nodes in the tree. The nodes to trim are randomly selected from the tree. Therefore, we need to get the keys from the tree since they are not necessarily consecutive. For example, key values in a tree can be 5, 6, 10, 20, 3, 30.

If **<u>a user specified number N</u>** of nodes to trim is less than the **<u>tree size (which is not N)</u>**, you just trim N nodes. If the N is larger than the tree size, set it to the tree size. At any case, you should trim all nodes one by one, but randomly. You may have your own implementation, but here is a suggestion:

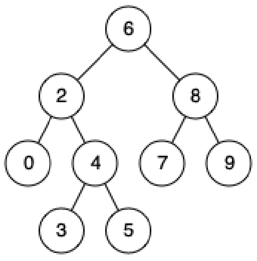
- Step 1:
  1. Get a list of **all keys** from the tree first.
  2. Invoke **inorder()** to fill a vector with keys in the tree.
     (Use a vector object in C++ **to store all keys**. The vector size grows as needed.)
  3. Get **the size of the tree** using size()
  4. Compare the tree size and the vector size returned from inorder() for checking.
- Step 2:
  1. Shuffle the vector with keys. – shuffle()
  2. If you don't take this step, you end up deleting nodes sequentially from the root of the tree which is not our intention.
- Step 3:
  1. Invoke **trim() N times** with a key from the array in sequence. Recall that, Inside a for loop, **trim() may return a new root of the tree**.

Hint: How to get all the keys in a tree.

Use one of tree traversal functions that returns keys in a vector from the tree. You may take a look into a function called inorder() in tree.cpp.

Step 2.4: LCA for BST

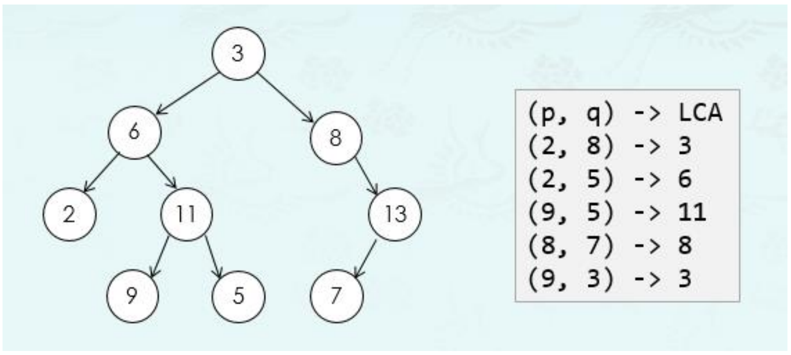
The lowest common ancestor (LCA) is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).



For example, given binary tree shown below, the LCA of nodes 2 and 8 is 6. The LCA of nodes 2 and 4 is 2 since a node can be a descendant of itself according to the LCA definition. Notice that

- All of the nodes' values will be unique
- p and q are different and both values will exist in the BST.

Intuition: Lowest common ancestor for two nodes p and q would be the last ancestor node common to both of them. Here last is defined in terms of the depth of the node. The below diagram would help in understanding what lowest means.



```
(p, q) -> LCA
(2, 8) -> 3
(2, 5) -> 6
(9, 5) -> 11
(8, 7) -> 8
(9, 3) -> 3
```

Note: One of p or q would be in the left subtree and the other in the right subtree of the LCA node.

Algorithm:

1. Start traversing the tree from the root node.
2. If both the nodes p and q are in the right subtree, then continue the search with right subtree starting step 1.
3. If both the nodes p and q are in the left subtree, then continue the search with left subtree starting step 1.
4. If both step 2 and step 3 are not true, this means we have found the node which is common to node p's and q's subtrees. and hence we return this common node as the LCA.

Time Complexity: $O(N)$, where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.

Space Complexity: $O(N)$. This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed BST could be N.

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

- PSet10 for BT – tree.cpp,
  BT menus including 'clear'
- PSet11 – tree.cpp,
  BT & BST menu items should work together.
- PSet12 – tree.cpp, treeprint.cpp,
  BT, BST and AVL menu items should work together.

Due and Grade points

Grade points:

- Step 1.1 ~ 1.5: 1 point per step
- Step 2.2 ~ 2.5: 1 point per step
- Step 3 AVL: 5 points

References

1. Recursion :
2. Recursion: