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.

BST and AVL Tree

Table of Contents

Introduction

This problem set consists of two sets of problems but they are closely related each other. Before you jump to the second problem, it is a good idea to finish the first problem set or BST problem set.

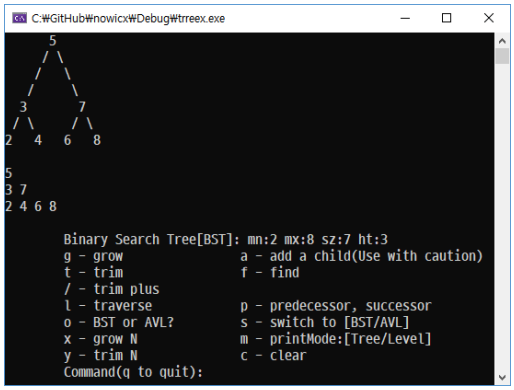
Your task is to complete the binary search tree(BST) program in tree.cpp, which allows the user test the binary search tree interactively.

Files provided

- **treeDriver.cpp** : tests 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 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.

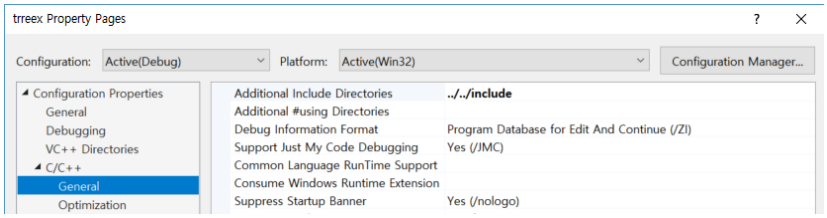
JumpStart



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 /wd4996, and optionally, add /D "DEBUG" at
  - Project Property → C/C++ → Command Line

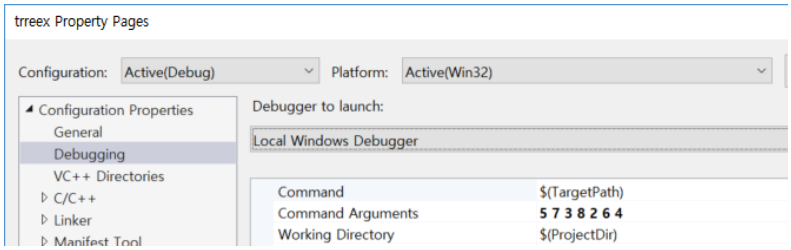
In my case for example:



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

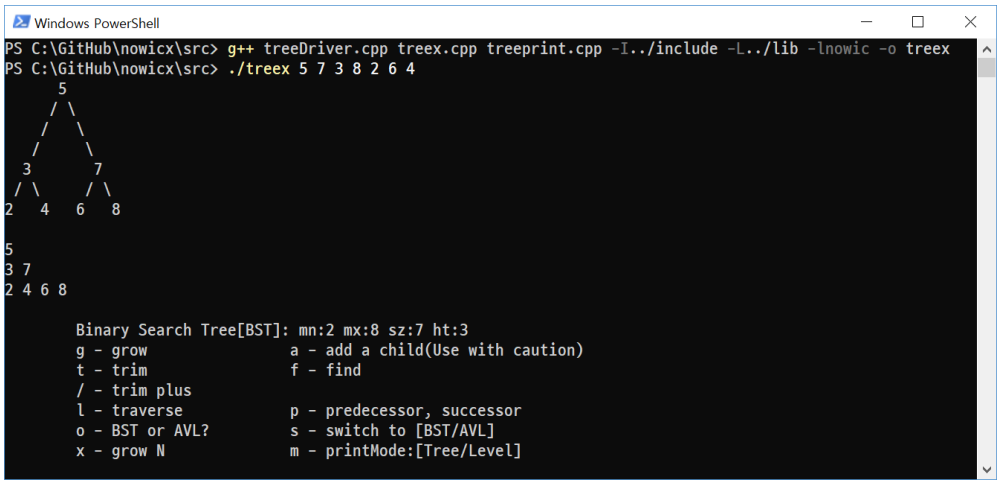
To have a tree to begin with, you may specify the initial keys for the tree in
Project Properties → Debugging → Command Argument

5 7 3 8 2 6 4



The function **build_tree_by_args()** in treeDriver.cpp gets the command arguments and builds a **BST or AVL** tree. You may set the function parameter AVLtree = true to create AVL tree, false for BST. Then, you will see the tree as shown above when you rerun the project.

You may run the program with some tree nodes created initially:



Note that many functions and files are designed to handle the functionalities and menu items both BST and AVL tree.

Step 1: BST ADT – 2 point

Many 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.

- clear()
- size()
- height()
- minimum(), maximum();
- contains()
- inorder(), preorder(), postorder() – use std::vector
- levelorder() – use std::vector and std::queue
- pred(), succ() – returns predecessor node and successor node of a tree.
- isBST(), _isBST() – returns true if the tree is a binary search tree, otherwise false.

Step 2: BST trim() – 1 point

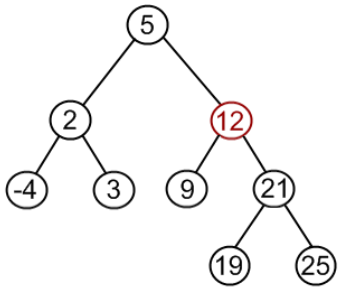
The trim (or delete) operation on binary search tree is more complicated than insertion (or grow) and search. Basically, in can be divided into two stages:

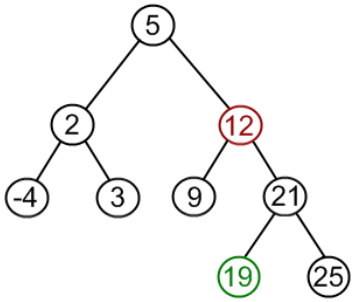
- Search for a node to remove **recursively**; for example:
```
if (key < node->key)
    node->left = trim(node->left, key);
```
- Eventually, this **node→left** will be set by the return value when **trim(node→left, key)** is done.
- If the node is found, run trim algorithm **recursively**.

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);**

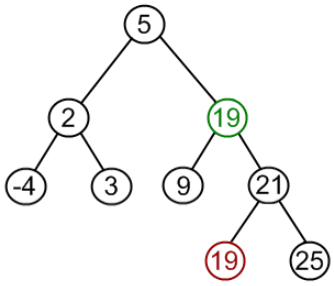
- Node to be trimmed is leaf (or has no children):
  - Simply remove from the tree. Algorithm sets corresponding link of the parent to **nullptr** and disposes the node. **Therefore, it returns nullptr.**
- Node to be trimmed has only one child:
  - Copy the node to a temp node.
  - Set the node to the 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 be trimmed has two children:
  - Find successor of the node.
  - Copy the key value of the **successor** to the node.
  - Delete the successor by calling **trim()** with succ() **recursively.** Therefore it **returns whatever this trim() returns.**

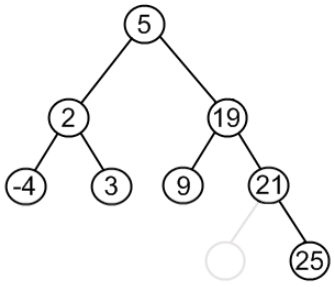
Example: Remove 12 from a BST.





1. Find successor (minimum element in the right 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**.

Step 3: BST Trim plus – 1 point

Once you make the trim option successfully using the successor of the node, now you are ready to improve it. Copy your trim() function into **trimplus()** and implement Trim+ functionality in there.

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. Improve your **trimplus()** 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 4: BST trimN – 1 point

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 **a user specified number N** of nodes to trim is less than the **tree size (which is not N)**, 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: Get a list of **all keys** from the tree first.
    1. 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.)
    2. Get **the size of the tree** using size()
    3. Compare the tree size and the vector size returned from inorder() for checking.
- Step 2: Shuffle the vector with keys. – shuffle()
  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: Invoke **trim() N times** with a key from the array in sequence.
  Inside a for loop, **trim() may return a new root of the tree**. Make sure that you should do something with this new root to pass it to the subsequent call of trim().

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 5: AVL Tree ADT – 3 Points

The second part of this problem set is to complete AVL tree, a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. This height difference is called the balance factor. Most of the BST operations (e.g., find, maximum, minimum, grow, trim... etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed binary tree. If we make sure that height of the tree remains O(Log n) after every insertion and deletion, then we can guarantee an upper bound of

O(Log n) for all these operations. The height of an AVL tree is always O(Log n) where n is the number of nodes in the tree.

You may start the program treeDriver.cpp with AVL tree by setting the function **build_tree_by_args()** parameter **AVLtree = true** to create AVL tree.

growAVL() & trimAVL()

Implement **growAVL()** and **trimAVL()**.

To make sure that the given tree remains AVL after every insertion or deletion, we must argument the standard BST grow/trim operations to perform some re-balancing. Also you must implement the function rebalance() invoked in growAVL() and trimAVL().

The function rebalance() rebalances the AVL tree during grow and trim(insert or delete) operations. It checks the rebalanceFactor at the node and invokes rotateLL(), rotateRR(), rotateLR(), and rotateRL() as needed.

One way to check your AVL tree is formed right is to compare the number of nodes and the height of tree. It follows the following formula:

$$h \leq 1.44\, log_2 n$$

For example, the height of the tree is less than 14.4 for n = 1024 nodes. Even though you increase the number of nodes in double or n = 2048, its heights should increase only by one.

growN() & trimN()

For "grow N" and "Trim N" options, you may use growAVL() and trimAVL() function as the code is provided with you. This implementation, however, has a problem for the large data set of N since grown() and trimAVL() invoke rebalance() N times, where it is a very expensive operation for a large N.

A way to avoid calling rebalance() N times is to trim (or grow) N items using BST functions since AVL tree is also BST. After finishing all trimming (or growing) N times, then invoke rebalanceTree() at the root once. Also we need to make sure that rebalanceTree() work efficiently.

```
// inserts N numbers of keys in the tree(AVL or BST), based
// on the current menu status.
// If it is empty, the key values to add ranges from 0 to N-1.
// If it is not empty, it ranges from (max+1) to (max+1 + N).
// For AVL tree, use BST trim() and rebalanceTree() once at root.
tree growN(tree root, int N, bool AVLtree) {
    int start = empty(root) ? 0 : value(maximum(root)) + 1;
    int* arr = new (nothrow) int[N];
    assert(arr != nullptr);
    randomN(arr, N, start);

    for (int i = 0; i < N; i++) root = grow(root, arr[i]);
    if (AVLtree) root = rebalanceTree(root);
    delete[] arr;
    return root;
}
```

```
// removes randomly N numbers of nodes in the tree(AVL or BST).
// It gets N node keys from the tree, trim one by one randomly.
// For AVL tree, use BST trim() and rebalanceTree() once at root.

tree trimN(tree root, int N, bool AVLtree) {
    vector<int> vec;
    inorder(root, vec);
    shuffle(vec.data(), vec.size());

    int tsize = size(root);
    assert(vec.size() == tsize);   // make sure we've got them all

    int count = N > tsize ? tsize : N;

    for (int i = 0; i < count; i++) root = trim(root, vec[i]);

    if (AVLtree) root = rebalanceTree(root);
    return root;
}
```

Once you finish this far, all menu items should work except 'w – switch to AVL or BST' that invokes rebalanceTree().

Step 6 – rebalanceTree() – 2 Points

We use growAVL() and trimAVL() every time we insert or delete a node into an existing AVL tree. In this case we use **rebalance()**, not rebalanceTree() which makes balances of a tree.

In this Step, implement rebalanceTree() that reconstructs a AVL tree from an existing BST in O(n). It is faster than rebalancing all nodes in BST in place. There could be many ways. Let us start a skeleton code and I propose three methods below:

```
// reconstructs a new AVL tree from BST in O(n).
tree rebalanceTree(tree root) {
  if (root == nullptr) return nullptr;

  // you may use inorder() to get an array of keys or nodes
  // if you use an array of nodes, you just reconstructs AVL tree using nodes.
  // if you use an array of keys, the root should be cleared (or deallocated)
  // your code here                    // O(n)

  return buildAVL(v.data(), v.size()); // O(n)
}
```

Method 1: The first method we can think of is to apply a series of re-balance operations as needed while going down the tree from the root. It is possible, however, it is too costly since it has to invoke the expensive rebalance() too many times. Once again we could repeatedly invoke rebalanceTree() until isAVL() returns true. This solution is **unacceptable.** However, this kind of operation is not usable for the large trees.

```
while (!isAVL(node))                // O(n) ~ O(n log n)
    node = _rebalanceTree(node);    // O(n log n)
```

Method 2: One efficient way to do it is to use one of main feature of BST algorithm and functions which are already available. Here is an algorithm:

1. Use inorder traversal and get **keys** into a sorted array.

2. Build balanced tree from that array (that can be done by picking root from middle of the array and recursively splitting the problem). Balanced tree satisfies AVL definition.

It recreates the whole tree again and deallocates the original tree. Both operations can be easily done in O(n) time. Skeleton codes for Method 2 & 3 are provided. You can get the **an array of keys** using the following inorder() and vector's data() function.

```
void inorder(tree t, std::vector<int>& v); // traverses tree in inorder & returns keys
```

```
vector<int> v;
inorder(root, v);
tree new_root = buildAVL(v.data(), v.size());
```

```
// rebuilds an AVL tree with a list of keys sorted.
// v - an array of keys sorted, n - the array size
tree buildAVL (int* v, int n) {
    if (n <= 0) return nullptr;
    // construct and set a TreeNode and initial values, use the [mid] element of v.

    // your code here - recursive buildAVL() calls for left & right
                    // from 0 to mid-1 (or mid number of nodes)
                    // from mid+1 to the end (how many nodes?)
  return root;
}
```

Method 3: It is the same as the method 2 except this one gets an array of nodes instead of keys of nodes. Then it does utilize all the nodes as they are and reconnect them according to algorithm. A function prototype of inorder() added in tree.h already returns a vector<tree> type with all nodes from the tree. In the following version, it does not allocate a new memory space, it just uses nodes listed in a vector. You can get **an array of nodes** using the following inorder() and vector's data() function.

```
void inorder(tree t, std::vector<tree>& v);    // traverses tree in inorder & returns nodes
```

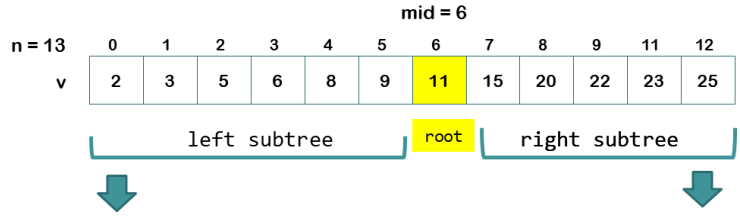
```
vector<tree> v;
inorder(root, v);
tree new_root = buildAVL(v.data(), v.size());
```

```
// rebuilds an AVL tree using a list of nodes sorted, no memory allocations
// v - an array of nodes sorted, n - the array size
tree buildAVL(tree* v, int n)
    if (n <= 0) return nullptr;

    // the first center node becomes the root.
    // all leaf nodes must have null eventually.

    // recursive calls for left & right
                    // from 0 to mid-1 (or mid number of nodes)
                    // from mid+1 to the end (how many nodes?)

    return root;
}
```

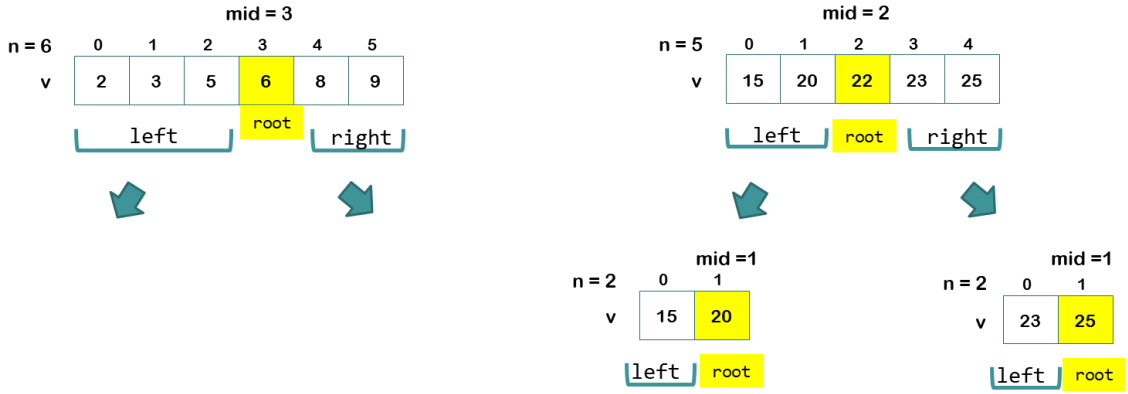
Hint: It is easier that you implement Method 2 first, then you go for Method 3 if you wish.

An example of buildAVL() function

We have that v an array of elements from BST and n is the size of v. The array v can be obtained using inorder() in either keys or nodes. Let's suppose we have the data as the first arguments in buildAVL().



Once you have arguments shown above, then use the middle element as a root. The first half of array goes to form the left subtree and the second half goes for right subtree, recursively.



Step 7 – testing

Make sure that you test "grow N", "trim N", "rebalance tree", and "switch to [BST/AVL]" with N = 100,000 ~ 500,000 or more. Based my experiments, these operations would take a second or less or a few seconds at most in my notebook.

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: _____ Section: _____ 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

- Upload tree.cpp to pset09 folder for BST
- Upload tree.cpp to pset10 folder for AVL tree

Due and Grade points

1st Due: 11:55 pm, May. 23 – BST
2nd Due: 11:55 pm, May. 23 – AVL tree

Grade points:

- 5 for BST
- 5 for AVL