

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.

Pset on AVL

Table of Contents

Introduction	1
JumpStart	2
Step 0: An easy way to create a tree for debugging	3
Step 3: AVL Tree.....	3
Step 3.1 growAVL() & trimAVL() – 1 point	4
Step 3.2 – reconstruct() – 2 points	4
1. Reviewing growN() & trimN()	4
2. Reconstruct().....	5
3. An example of buildAVL() function	7
Step 4 show mode – 1 points	8
Submitting your solution	8
Files to submit.....	8
References	8

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\pset10-12tree> ./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 [BT]/[AVL]** C - convert BT to BST*
c - clear         s - show mode:[tree]
Command(q to quit): 

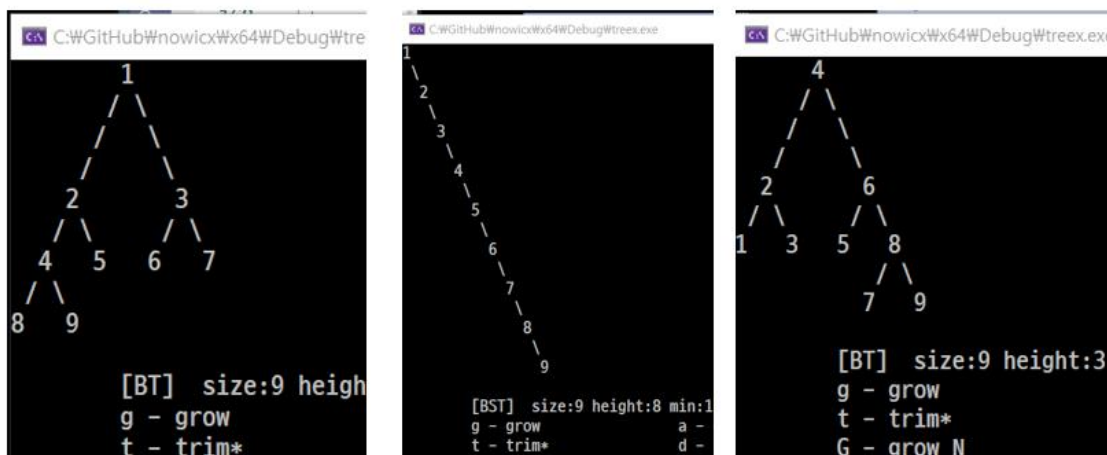
```

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



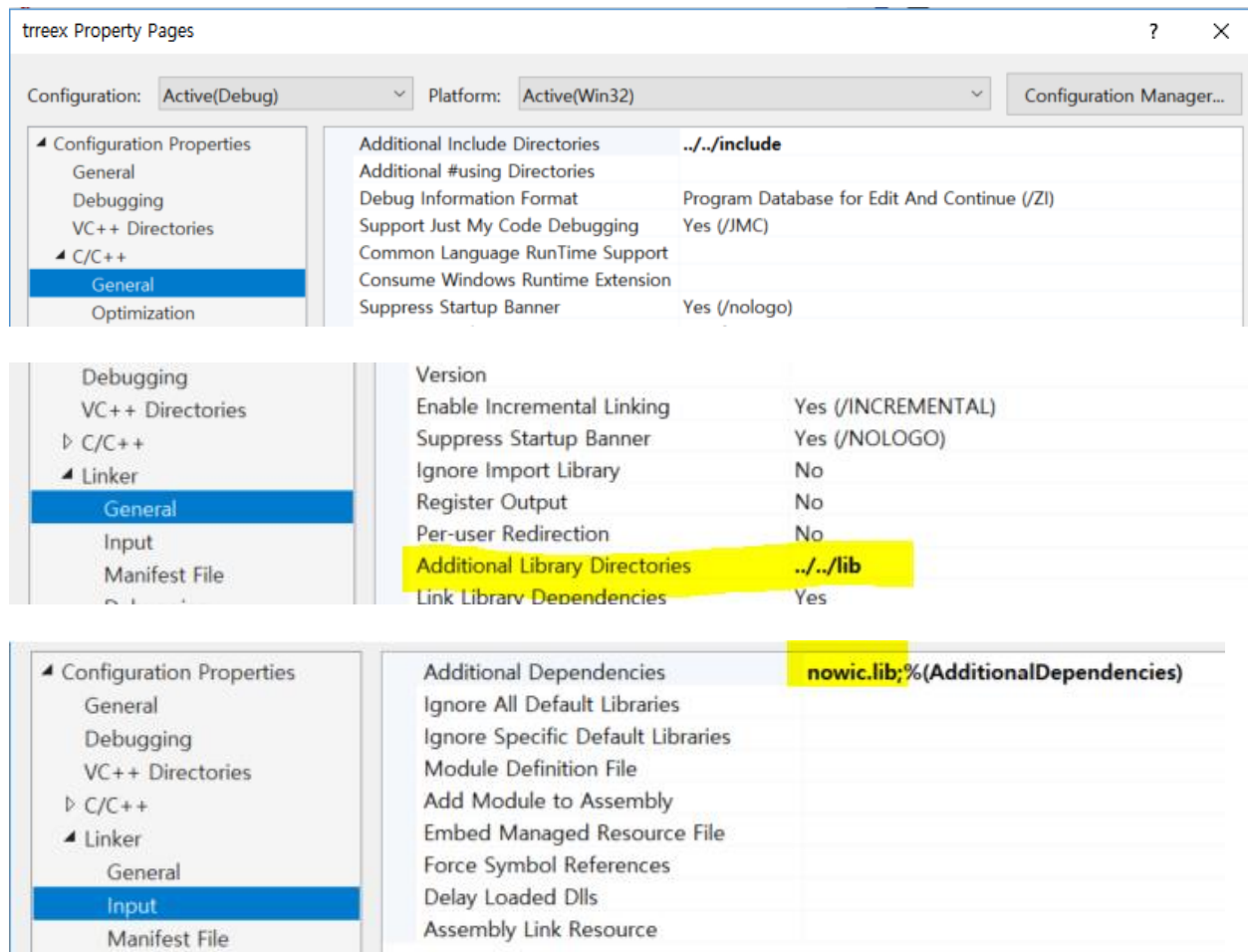
JumpStart

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:

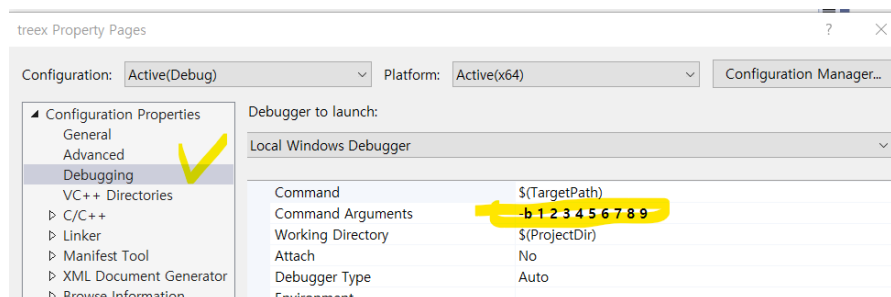


Add ~.h files under Project 'Header Files' and ~.cpp files under project 'Source Files'. Then you may be able to build the project.

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



Step 3: AVL Tree

This part of the 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 `-a` as a command line argument to create AVL tree.

Step 3.1 growAVL() & trimAVL() – 1 point

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.

Step 3.2 – reconstruct() – 2 points

Let's think about `growN()` and `trimN()` for AVL tree which seem working fine as they are.

1. Reviewing growN() & trimN()

Two functions `growN()` and `trimN()` provided for "grow N" and "Trim N" options work fine for small N. These two functions use `growAVL` and `trimAVL` function every time it inserts or deletes a node in the tree as shown below. Surely, this would **not** be acceptable for AVL tree for a large N since it keeps on calling `rebalance()` function which is a very expensive operation.

```
tree growN(tree root, int N, bool AVLtree) {
    DPRINT(cout << ">growN N=" << N << endl);
    int start = empty(root) ? 0 : value(maximum(root)) + 1;

    int* arr = new (nothrow) int[N];
    assert(arr != nullptr);
    randomN(arr, N, start);

    // use its own grow() function, respectively. it is too slow for AVL tree.
    if (AVLtree)
        for (int i = 0; i < N; i++) root = growAVL(root, arr[i]); //// UNACCEPTABLE CODE ////
    else
```

```

    for (int i = 0; i < N; i++) root = grow(root, arr[i]);

    delete[] arr;
    DPRINT(cout << "<growN size=" << size(root) << endl);
    return root;
}

```

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 `reconstruct()` at the root once. Also we need to make sure that `reconstruct()` work efficiently as shown below.

```

// 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 grow() and reconstruct() 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 = reconstruct(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 reconstruct() once at root.
tree trimN(tree root, int N, bool AVLtree) {

    // left out

    return root;
}

```

When implementing `trimN()`, you must make sure that the code handles negative numbers or numbers that are larger than N . You have nothing to delete if non-positive numbers are entered, but remove all nodes if user's input is N or larger.

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

2. Reconstruct()

We don't want to use `growAVL()` and `trimAVL()` for a large of N operations. Instead we want to use BST functions as they are, then invoke `reconstruct()` once for all to make the tree rebalanced.

In this Step, the purpose is to implement `reconstruct()` that reconstructs a AVL tree from an existing BST in $O(n)$. It is faster than rebalancing all nodes in BST in place. We are going to implement two methods.

For small trees or nodes are less than or equal to 10, then we just get keys from BST and recreate a new AVL tree. This is called a **recreation method**.

For larger trees or nodes are more than 10, then we get the nodes from the existing BST and rearrange them as an AVL tree. This is called a **recycling method**.

There could be many ways. Let us start a skeleton code and I propose three methods below:

```
// reconstructs a new AVL tree in O(n), Actually it is O(n) + O(n).
// Use the recreation method if the size is less than or equal to 10
// Use the recycling method if the size is greater than 10.
// recreation method: creates all nodes again from keys
// recycling method: reuses all the nodes, no memory allocation needed
tree reconstruct(tree root) {
    DPRINT(cout << ">reconstruct " << endl);
    if (empty(root)) return nullptr;

    cout << "your code below" << endl;
    if (size(root) > 10) {    // recycling method
        // use new inorder() to get nodes sorted
        // O(n), v.data() - the array of nodes(tree) sorted
        // root = buildAVL(v.data(), (int)v.size()); // O(n)
    }
    else {                  // recreation method
        // use inorder() to get keys sorted
        // O(n), v.data() - the array of keys(int) sorted
        // clear root
        // root = buildAVL(v.data(), (int)v.size()); // O(n)
    }
    DPRINT(cout << "<reconstruct " << endl);
    return root;
}
```

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. This solution is **unacceptable**.

Method 2 (Recreation Method): 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 characteristics that returns **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 **deallocates the original tree** and recreates the whole tree again. 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
```

```
// 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;
    // create a new root 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 (Recycling Method): It is the same as the method 2 except this gets an array of nodes instead of keys of nodes. Then it utilizes 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 sorted by keys from the tree. This algorithm does not recreate new nodes, but it just uses existing nodes of the tree. 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
```

```
// 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;

    // set the mid node as a root.

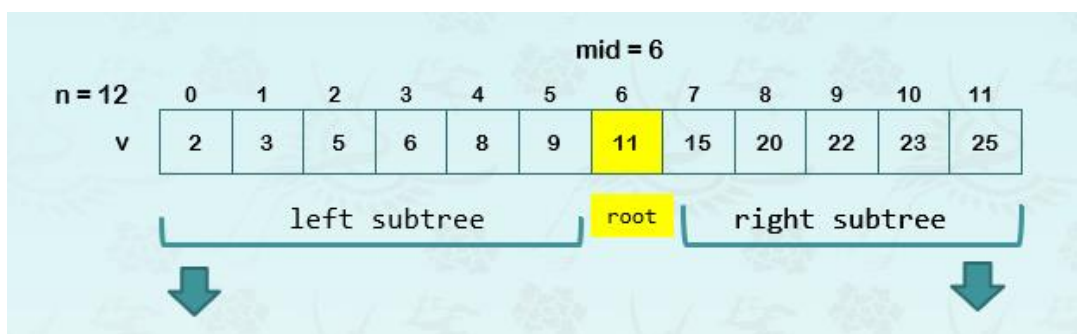
    // recursive calls for left & right
    // from 0 to mid-1 (or mid number of nodes)
    // from mid+1 to the end (how many nodes?)

    return root;
}
```

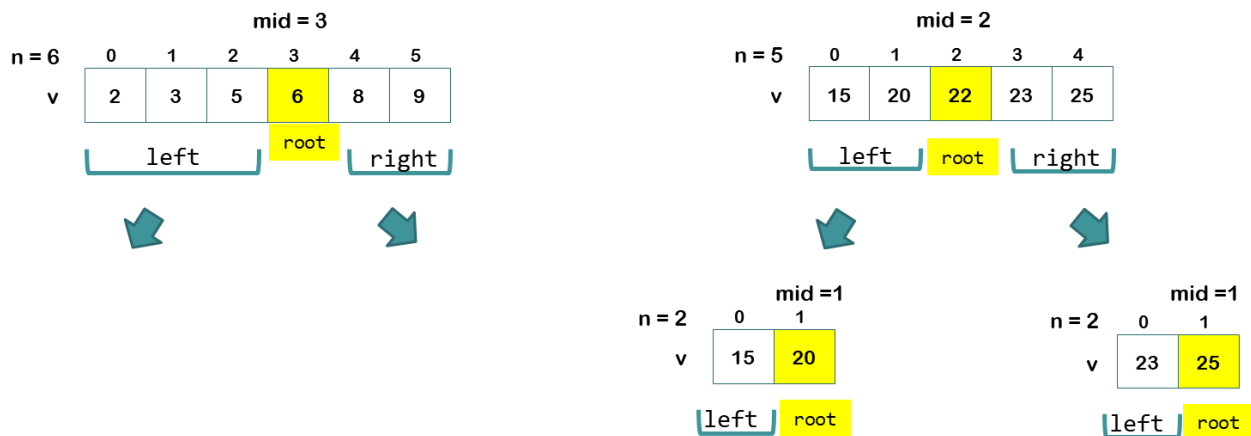
Hint: It will be interesting to see the time difference between the last two methods even though both of them have the time complexity of $O(n)$.

3. 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 4 show mode – 1 points

Improve "show tasty" mode. If the tree grows big, there are too many levels and too many nodes to see them all. It currently shows only first 3 rows and last 3 rows if there are more than nodes. Improve this functionality such that it shows **show_n** nodes only in each level.

Modify the following function in **treeprint.cpp**. Additionally, create a helper function for it as well when necessary in treeprint.cpp.

```
void treeprint_levelorder_tasty(tree root)
```

There is a helper function called `show_vector()` in `tree.cpp` available at your convenience.

```
void show_vector(std::vector<int> vec, int show_n = 20);
```

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

- Pset AVL Tree: Step 3.1 ~ 3.2, Step 4
 - tree.cpp, treeprint.cpp

References

1. [Recursion](#) :
2. [Recursion](#):