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 BT, BST and AVL Tree

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.

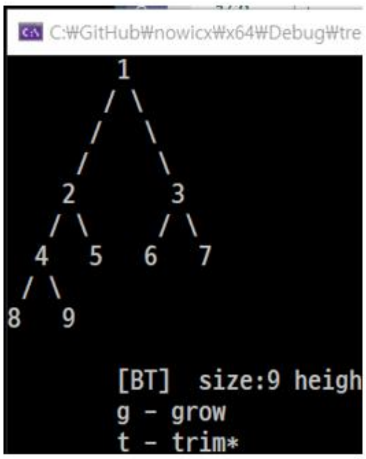
```
PS C:\GitHub\nowicx\psets\pset10-12tree> ./treex -b 1 2 3 4
    1
   / \
  2   3
 /
4

        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 [BST]/[AVL]**  C - convert BT to BST*
        c - clear               s - show mode:[tree]
        Command(q to quit): 
```

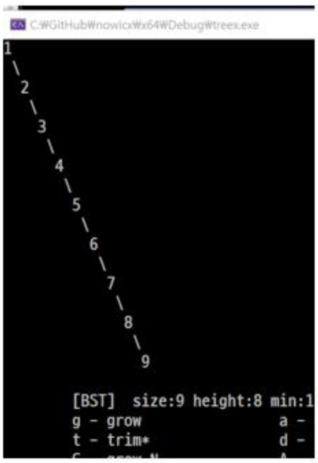
With the following three different options you can get three different trees created automatically at the beginning of the tree program execution.

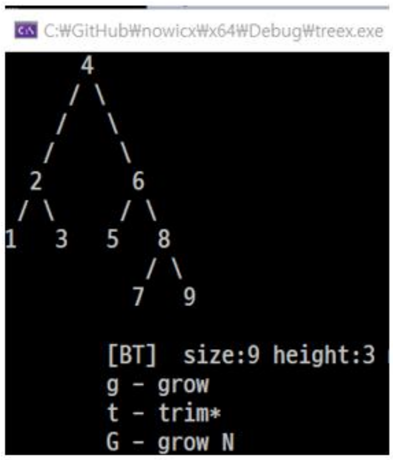
./treex -b 1 2 3 4 5 6 7 8 9

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

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





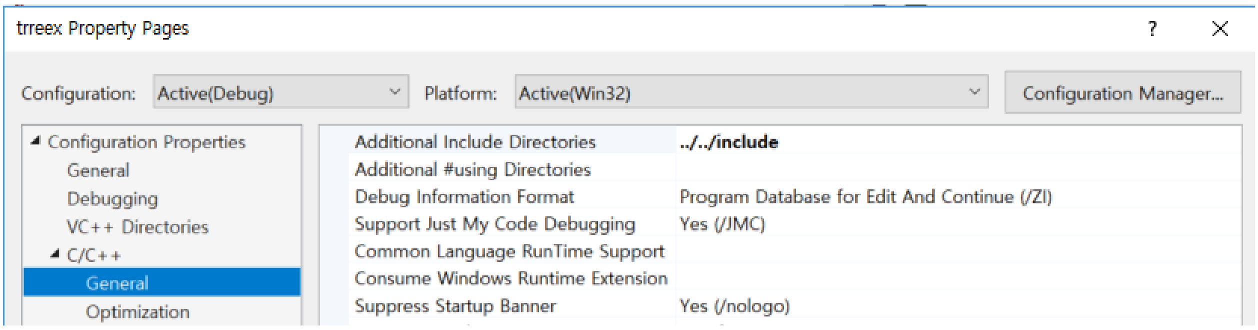


JumpStart

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

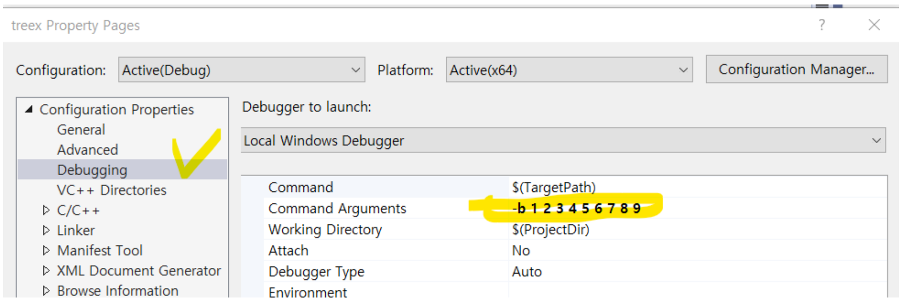
In my case for example:



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 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.

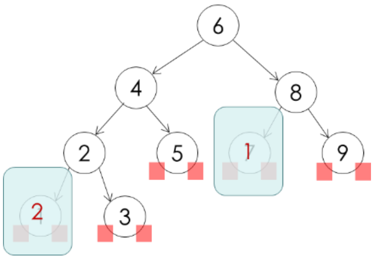
```
// level order traversal of a given binary tree using iteration.
void levelorder(tree root, vector<int>& vec) {
   Visit the root.
   if it is not null, push it to queue.
   while queue is not empty
     queue.front() – get the node from the queue
     visit the node (save the key in vec).
     if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

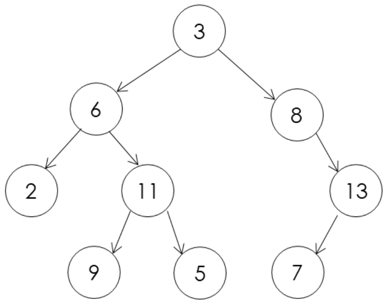
```
First, push the root to the queue.
Then, while the queue is not empty,
    Get the front() node on the queue
    If the left child of the node is empty,
       make new key as left child of the node. - break and return;
    else
       add it to queue to process later since it is not nullptr.
    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:

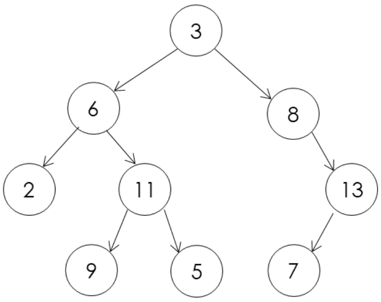
- If **root = nullptr**, return false. [base case]
- If **root's key = x** or if it node **x** exists in root's left or right subtree during recursive search,
    - Push the root's key into **vector**. (recursive back-trace happens here)
    - Return true.
- Else
    - return false.

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

Step 1.5: LCA in BT

This step implements a function called LCA_BT which finds the lowest common ancestor (LCA) of two given nodes in a given binary tree using iteration and recursion algorithms.

The 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)." Two nodes given, p and q, are different and both values will exist in the binary tree. For example,



The lowest common ancestor for the two nodes (2, 8) would be 3. Likewise, LCA(2, 5) = 6, LCA(9, 5) = 11, LCA(8, 7) = 8, LCA(9, 3) = 3.

Intuition (Iteration): A brute-force approach is to traverse the tree and get the path to node p and q. Compare the path and return the last match node of the path.

Algorithm (Iteration):

- Find path from root to p and store it in a vector.
- Find path from root to q and store it in another vector.
- Traverse both paths till the values in vector are same. Return the common element just before the mismatch.

For example, to find LCA(2, 5), use findPath() function to get two paths for p and q. Then you may get them for this example as shown below:

- Path to 2: 3 6 2
- Path to 5: 3 6 11 5

Therefore the lowest common ancestor will be the last element of the same sequences part of two Paths. In this case, 3 and 6 are the common ancestor, but the least one will be 6 since it is closest from two nodes (2, 5).

Recursive algorithm is also shown below:

Intuition (Recursion): Traverse the tree in a depth-first manner. The moment you encounter either of the nodes p or q, return the node. The LCA would then be the node for which both the subtree recursions return a non-NULL node. It can also be the node which itself is one of p or q and for which one of the subtree recursions returns that particular node.

Algorithm (Recursion):

- Start traversing the tree from the root node.
- If the current node is nullptr, return nullptr. [base case]
- If the current node itself is one of p or q, we would return that node. [base case]
- [recursive case]
    - Search for the left side and search for the right side recursively.
    - If the left or the right subtree returns a non-NULL node, this means one of the two nodes was found below. Return the non-NULL node(s) found.
    - If at any point in the traversal, both the left and right subtree return some node, this means we have found the LCA for the nodes p and q.

Time Complexity: O(n), Space Complexity: O(n)

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()

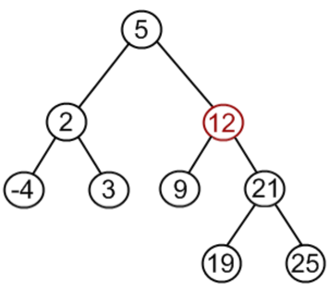
In this step, implement the basic functions grow() and trim(). Since we covered grow() during class, let us go the trim() function once more. The trim (or delete) operation on binary search tree is more complicated than insertion (or grow) and search. Basically, in can be divided into two stages:

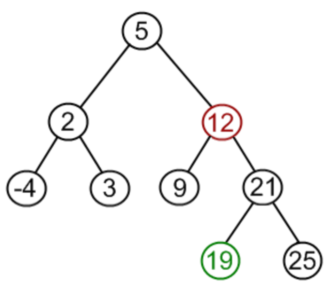
- Search for a node to remove **recursively**; for example:
```
if (key < node->key)
    node->left = trim(node->left, key);
```
- Eventually, this **node→left** will be set by the return value when **trim(node→left, key)** is done.
- If the node is found, run trim algorithm **recursively**.

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);**

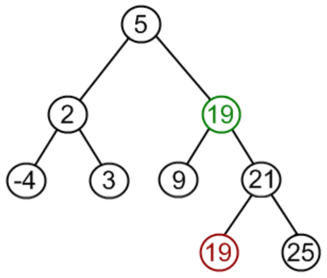
- Node to trim is leaf (or has no children):
    - Simply remove from the tree. Algorithm sets corresponding link of the parent to **nullptr** and disposes the node. **Therefore, it returns nullptr.**
- Node to trim has only one child:
    - Copy the node to a temp node.
    - Set the node to the child.
    - 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.

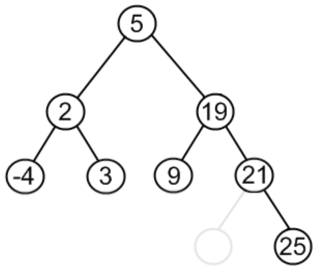
Example: Remove 12 from a BST.





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 **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:
  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:

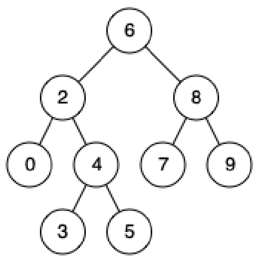
1. Invoke **trim() N times** with a key from the array in sequence. Recall that, Inside a for loop, **trim() may return a new root of the tree**.

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

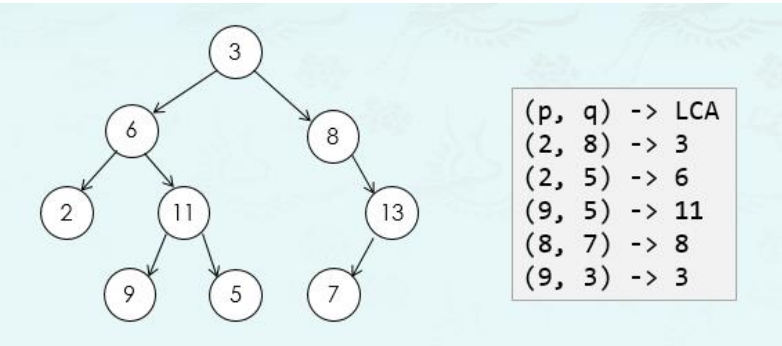
The lowest common ancestor (LCA) is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).



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.



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

Algorithm:

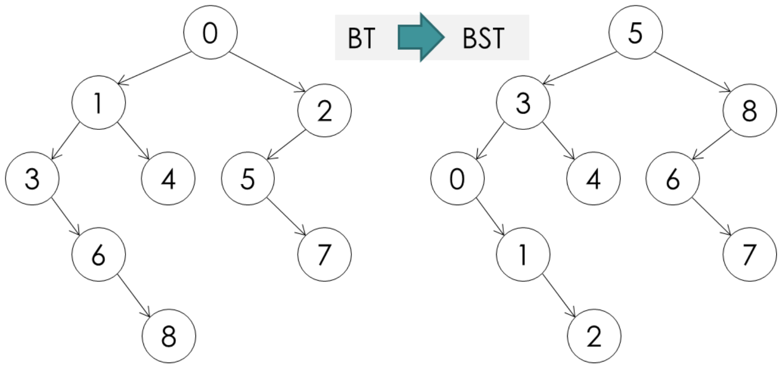
1. Start traversing the tree from the root node.
2. If both the nodes p and q are in the right subtree, then continue the search with right subtree starting step 1.
3. If both the nodes p and q are in the left subtree, then continue the search with left subtree starting step 1.
4. If both step 2 and 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.

Step 2.5: Convert BT to BST in place

In this step, we want to convert a binary tree to a binary search tree while keeping its tree structure as it is. An example is shown below:



Algorithm: Use either **vector or set in STL**, not an array, **because of a pedagogical purpose**.

- Step 1 – store keys of a binary tree into a container (vector or set in STL).
- Step 2 – sort the vector using any sorting technique. STL set is already sorted.
- Step 3 – Now, do the inorder traversal of the tree and copy the elements from the container to the nodes of the tree one by one.

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:

$$h \le 1.44\, log_2 n$$

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 ASL 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 indorder() 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.

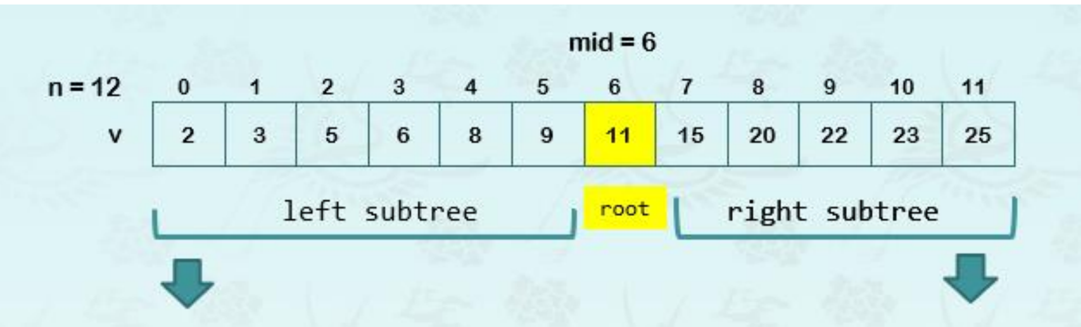
```
void inorder(tree t, std::vector<tree>& v);  // traverses tree in inorder & returns nodes
```

```
// 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;
}
```

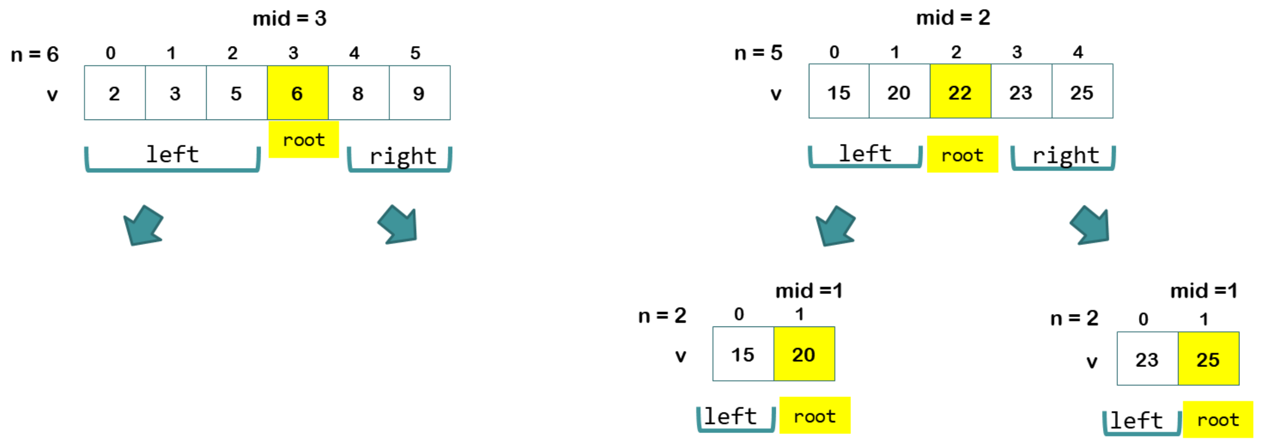
Hint: It will be interesting to see the time difference between the last two methods even though both of them have the time complexity of O(n). .

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 3.4 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

- **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

- Binary Tree: Step 1.1 ~ 1.5
- Binary Search Tree: Step 2.2 ~ 2.5
- AVL Tree: Step 3.1 ~ 3.4

References

1. Recursion :
2. Recursion: