Your work for this must be finished and shown to your lab TA by the end of this lab session or the start of your next session.

Lab4 uses the Binary Search Tree (BST) data structure. Even if you have no previous experience with BST's everything you need to know about them for this lab is explained below or in the linked videos.

This video covers "binary tree" basics https://www.youtube.com/watch?v=KQ4BA5tGkMU.

First, a BST is a binary tree. That means it's either empty (so it has no nodes at all), or it has one root node and every node has at most two children, which are usually called "left" and "right".

- root – the top node in a tree (the only node with no parent)
- parent (of child) – the immediate ancestor of child (where parent—child path exists)
- siblings – nodes with the same parent
- descendant (of child)– a node in the left or right subtrees of child
- ancestor (of child) – a node in the path from root to parent (where parent—child path exists)
- leaf – a node with no children.(also called an external node; all others nodes are internal)
- edge – the connection between parent and child nodes (don't worry about this one; in bst.cc there are no "edges")
- path – a sequence of nodes and edges connecting a node with a descendant of the node
- path-length – given a path, the path-length is the number of edges in the path, which is equal to (number of nodes) - 1

Most of the functions in this lab will work on any binary tree (discussed above) but we're using BST's.

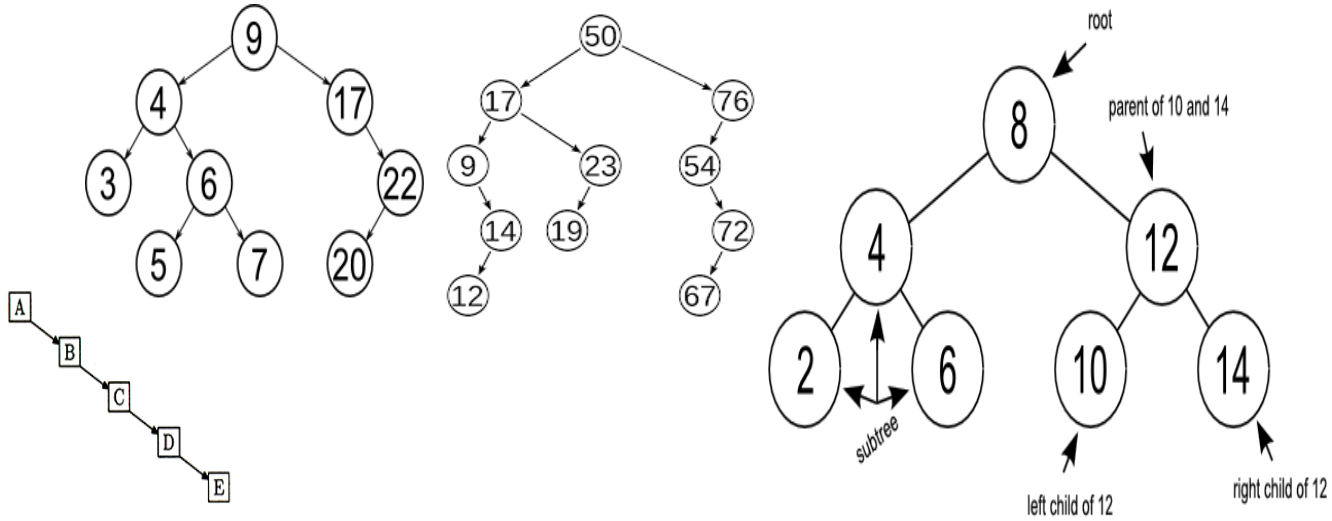
This video covers "binary search tree" basics https://www.youtube.com/watch?v=sf_9w653xdE.

BST nodes will have at least three data-members (there may be more, but it must have these):
- a "key" (sometimes called "data" or "value" or something more meaningful, like "student_id")
- a pointer to its "left" child (a value of NULL means there is no left child; NULL = 0x00)
- a pointer to its "right" child (again, if it's equal to NULL, there is no right child)

Exception: there is a way of storing the nodes of a tree in an array, and in that case the indices of the children are computed from the index of the parent, and vice-versa; they are not explicitly implemented as pointers (as you'll see the "heaps" lab).

Search Tree Property: the key is GT every key in the left subtree and LT every key in the right subtree
This property is true for all BST nodes. (The 4 trees below are BST's; the property is true in all nodes.)

sources: en.wikipedia.org en.wikipedia.org



sources: en.wikipedia.org http://msoe.us/taylor/tutorial/cs2852/bstTreeNames.png

Tool to insert/delete BST nodes (integer keys only) https://www.cs.usfca.edu/~galles/visualization/BST.html
Scroll to the bottom for the controls.
It must run quite slowly if you want to see the sequence of steps it uses when deleting a node with 2 children.

Your work for this must be finished and shown to your lab TA by the end of this lab session or the start of your next session.

Q1. Download the lab files from the course web page (in lab4.zip) and extract them into a new sub-directory ("lab4") of your "cs221" directory. Make that your current working directory then run "make" to compile. There are unit tests which fail (more than 40 of them). You must change the bst.cc code so they pass. If you run bst with no command line arguments, then the unit tests will be checked.

The commands needed to do everything specified in the above paragraph (except downloading the code and extracting its contents) are given below. This assumes you are currently in your home directory, which is the directory you are in after successfully logging in to your account.

```
cd cs221
mkdir lab4
```

Put the extracted files into cs221/lab4 now

```
cd lab4
make
./bst
```

You also have the ability to write your own testing/debugging code and run that instead.
To do that, put your code in runMain() and supply your test keys as command line arguments:

```
./bst 5 3 2 1 6 8 4 7 9
```

Only bst.cc needs to be changed to complete this lab. All the functions you're implementing or completing are described below. Do them in the given order. What you learn in the first relatively easy ones should help with the more complicated ones at the end.

Q2(a). Since this is a recursive structure (a binary tree is either empty, or it is a root node, together with at most two children, both of which are binary trees), recursive functions will work best. If you need a jump-start for this function, watch the first 5 minutes of https://www.youtube.com/watch?v=sqVefIEttT0
Regarding his code: it's considered "better" to deal with the base case first, i.e.:

```
if (! root) return 0; // should be the first line
```

In general, if you first eliminate the exceptions, returning whatever is appropriate for them, then what remains is the general case. In recursive code, the exceptions are the base cases. The above code returns when its condition is true, so program can only get to the next line if root is not NULL (see Note below).
Consider this corresponding example from a previous lab: to count the nodes in a linked_list

```
int count_nodes(Node* head){
  if (! head) return 0;  // BASE CASE
  return 1 + count_nodes(head->next);
}
```

The above is more concise, looks cleaner, and is easier to read and understand than something like:

```
int count_nodes(Node* head){
  if (head != NULL){
    return 1 + count_nodes(head->next);
  }
  else {
    return 0;
  }
}
```

Note: Usually (! root) is used instead of (root == NULL). They are exactly the same thing.
If "root" has value 0x00, then (! root) is (! 0x00) which is simply "not false", which is "true" of course.
If "root" has any other value, then (! root) will evaluate as false.

```
/**
 * Returns the number of nodes in the tree rooted at root.
**/
int numNodes( Node* root );
```

(Qb). The algorithm for numLeaves() will be very similar to numNodes(), with a couple of important differences. For one thing, the base case will be different.

```
/**
 * Returns the number of leaves in the tree rooted at root.
**/
int numLeaves( Node* root );
```

Q2(c). The height of any tree is defined as the greatest path-length from root-to-leaf (see page 1 for path-length definition). A tree with only one node has height = 0 and an empty tree has height = -1.

The height of a node is defined as the greatest path-length from the node to a leaf.
If you think of the node as the root of a subtree, the height of the node is the height of that subtree.

```
/**
 * Returns the height of node x.
**/
int height( Node* x );
```

Q2(d). The depth of a node relative to an ancestor is the length of the path from the ancestor to the node. This concept is so easy that a description of it tends to be complicated. So an example instead:

On page 1, in the tree with root→key = 9, the leaf with node→key = 5 has root-to-node path 9--4--6--5

node→key = 9 has path from root 9 and its depth relative to itself is 0 (clearly the base case)
node→key = 4 has path from root 9--4 so its depth relative to root = path-length(9--4) = 1
node→key = 6 has path from root 9--4--6 so its depth relative to root = path-length(9--4--6) = 2
node→key = 5 has path from root 9--4--6--5 so its depth relative to root = path-length(9--4--6--5) = 3

```
/**
 * Returns the depth of node x in the tree rooted at root.
**/
int depth( Node* x , Node* root ');
```

Q2(e). You saw "`Node*&`" in linked_list.cc and you might recall it means that the variable is passed by reference. The function in linked_list had to be able to change the value in the "head" variable itself. But traversals are not generally destructive; the value of "rootNode" itself won't be changed by your code.

It's often useful to read parameter declarations from right-to-left, so "`Node*& rootNode`" reads as "rootNode is a reference to a pointer to a Node", and "`int level`" as "level is an int".

The "Visitor" variable "v" is also passed by reference. It's a reference to a Visitor object, and likely it's a little too much to expect you to figure out the needed statement on your own in lab4, so here it is:

```
v.visit(rootNode, level);
```

To be clear: in each of these traversals, the above code is what "visits" the node itself.

Three ways to traverse a tree (i.e. to visit all nodes in the tree) are pre-order, in-order, and post-order. They are described below, and also in this video https://www.youtube.com/watch?v=xoU69C4lKlM. The first minute explain all you really need to know, but it's instructive to watch him build the BST.

An analogy for in-order traversal is arithmetic notation ("3+4"); operator "+" is in-between the two operands. In-order traversal happens by first traversing the left subtree, then "visiting" the node itself, and finally traversing the right subtree. In-order traversal "visits" the nodes in ascending order.

```
/**
 * Traverse a tree rooted at rootNode in-order, use 'v' to visit each node.
**/
void in_order( Node*& rootNode, int level, Visitor& v );
```

Your work for this must be finished and shown to your lab TA by the end of this lab session or the start of your next session.

Q2(f). An analogy for pre-order traversal is the way we might invoke an add() function, i.e. "add(3,4)". First there is the operator, then the two operands. Pre-order traversal happens by first "visiting" the node itself, then traversing the left subtree, and finally traversing the right subtree. The first node "visited" by a pre-order traversal will always be the root of the tree. (That fact can be useful if you're given the output of the three traversals and asked to reconstruct the tree itself.)

```
/**
 * Traverse a tree rooted at rootNode pre-order, use 'v' to visit each node.
**/
void pre_order( Node*& rootNode, int level, Visitor& v );
```

Q2(g). An analogy for post-order traversal is Reverse Polish Notation (see the wiki article for more info http://en.wikipedia.org/wiki/Reverse_Polish_notation#Postfix_algorithm). Some very early calculators and computers used this. You'd enter the first number in R0 (register zero) and push it onto the stack, then enter the second number in R0. The "plus" instruction would add whatever was on the top of the stack to whatever was in R0 (putting the result in R0, and decrementing the stack pointer). Post-order traversal happens by first traversing the left subtree, and then traversing the right subtree, and finally "visiting" the node itself. (Another fact that might be useful, if you're given some traversal outputs and asked to reconstruct the tree itself, is which node is always visited last by post-order traversal.) *

```
/**
 * Traverse a tree rooted at rootNode post-order, use 'v' to visit each node.
**/
void post_order( Node*& rootNode, int level, Visitor& v );
```

Q3. Almost finished. All that remains is the missing portion of delete_node(). The easy cases are done: the node to be deleted (case 1) is a leaf, or (case 3) only has a left child, or (case 4) only a right child, but the difficult one, (case 2) the node has two children, is up to you.

This video is extremely helpful https://www.youtube.com/watch?v=wcIRPqTR3Kc

Note that she opts to use the successor of the target node, but we are using the predecessor, which is the rightmost node of the left subtree. The visualizer (second last link on page 1) uses the predecessor.

```
/**
 * Deletes a node containing 'key' in the tree rooted at 'root'.
**/
bool delete_node(Node*& root, KType key);
```

One final question to ask yourself: which function(s) would have to change if the Nodes were part of a binary tree that wasn't also a binary search tree?

Hint: which function(s) relied on the fact that the keys in the left-child and its descendants were less than the key in the node, or that the keys in the right-child and its descendants were greater that it?