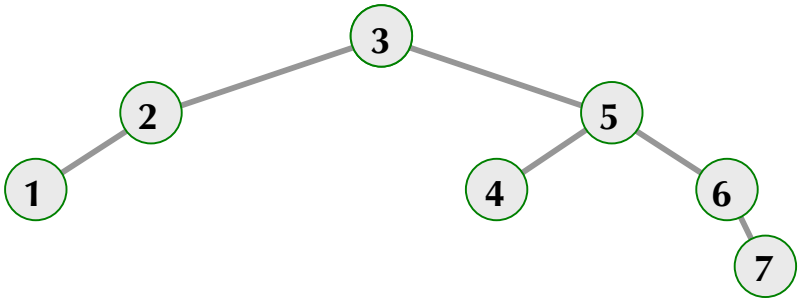
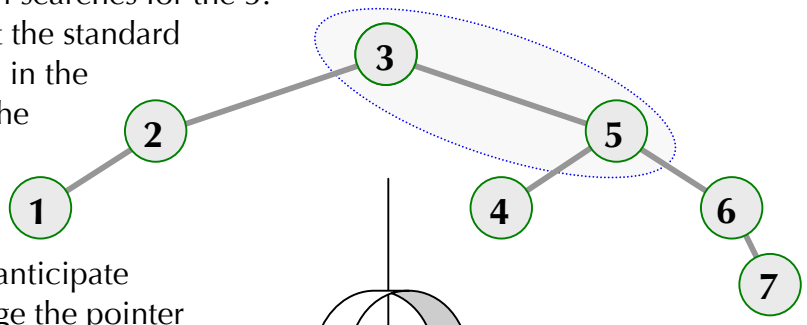
Section Handout

Discussion Problem 1: Tree Rotations

Given the pattern of references in a typical binary search tree, you can often improve the average search time using a simple technique known as **rotation**. Often, references to any particular element in a binary search tree are clustered in time, in the sense that a request for some particular element is likely to be followed by many other requests for the same element in the near future. Rotations can be used to bubble frequently accessed nodes toward the root of the tree, so subsequent searches can succeed in less time.

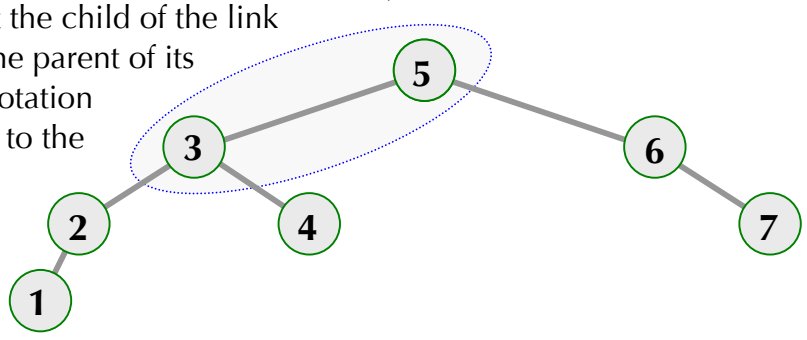


Suppose that a binary search tree has grown to store the first seven integers, as has the tree drawn to the right. Suppose that a program then searches for the 5. Clearly the search would succeed, but the standard implementation would leave the node in the same position, and later searches for the same element would take the same amount of time.



A more novel implementation would anticipate another search for 5, and would change the pointer structure in the vicinity of the 5 by performing what is called a **left-rotation**. A left rotation pivots around the link between a node and its parent, so that the child of the link rotates counterclockwise to become the parent of its parent. In our example above, a left-rotation would change the structure according to the figure on the right. Note that the restructuring is a local operation, in that only a constant number of pointers need to be updated. The key observation is that 5 is brought one level closer to the root, the former parent of 5 becomes 5's left child and in the process inherits what **used** to be the left child of 5 as its right child. In general, these two nodes might occur anywhere in a binary search tree. Notice that the binary search tree property is maintained.

```
rotateLeft(&node3);
```



a) Write a function **`rotateLeft`** which, given the address of the pointer to the parent (perhaps the 3 in the first illustration above), changes a constant number of pointers so that the right child of the referenced node (the 5 in the above illustration) becomes the parent. You may assume that both the referenced parent and its right child are both non-**`NULL`**.

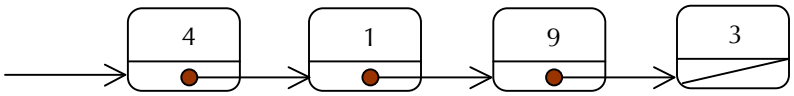
```
struct node {
   int value;
   node *left;
   node *right;
};

static void rotateLeft(node **parentp);
```

b) Now, using the **`rotateLeft`** operation you just wrote, along with its symmetric counterpart **`rotateRight`** (which you can assume works properly without actually writing it), implement **`pullToRoot,`** which takes a pointer to a binary search tree and bubbles the specified value up to the root. You may assume that the value is actually present, although a particularly clever implementation would leave the tree unaltered if the value is missing.

```
static void pullToRoot(node **rootp, int value);
```

Discussion Problem 2: Binary Tree Synthesis

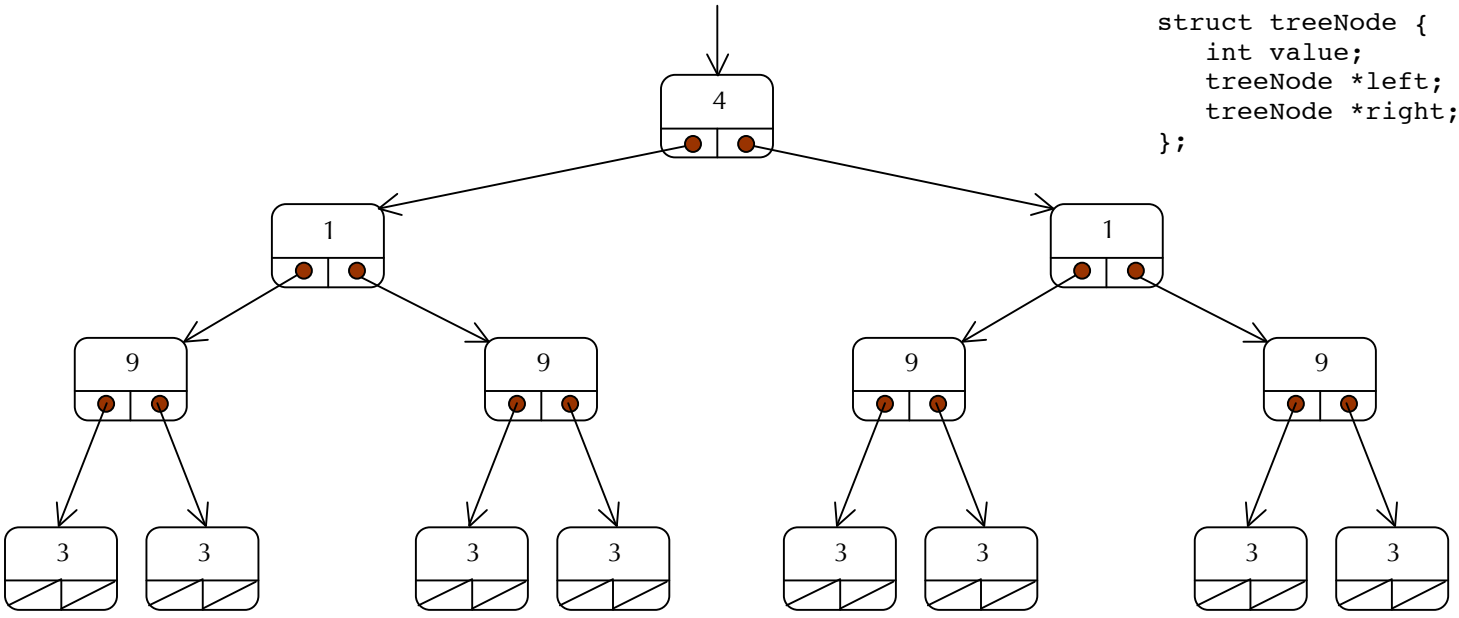
`listToBinaryTree` takes a singly linked list of numbers and constructs an independent binary tree where the nth item of the singly linked list occupies all positions at the nth level of the binary tree structure. So, if given the address of the list's front node




```
struct listNode {
   int value;
   listNode *next;
};
```

you would synthesize the following tree and return the address of its root:

```
struct treeNode {
   int value;
   treeNode *left;
   treeNode *right;
};
```



Had there been additional elements beyond the 3, then all of the 3s in the tree would have had two non-**NULL** children, and so forth. The product of the function you'll write will be a complete, balanced, binary tree (though not binary search) where every path from the root to a leaf sees the same sequence of numbers as seen in the original list.

For the discussion problem, you're to write two version of this **listToBinaryTree** function: one recursive function, and one iterative one. The recursive function is very short and very clean and gives birth to the tree in a depth-first manner. The second version is iterative, uses a single **Queue< treeNode **>** as an auxiliary data structure, and builds up the tree of interest in a breadth-first manner. That means that the node housing the 4 is allocated first, then the two nodes housing the 1s are allocated, initialized and planted as children of the 4 node, and then all four of the nodes holding 9s are placed, and then the eight nodes surrounding 3s would be placed.

There are advantages to both versions, so it's instructive to be familiar with each.

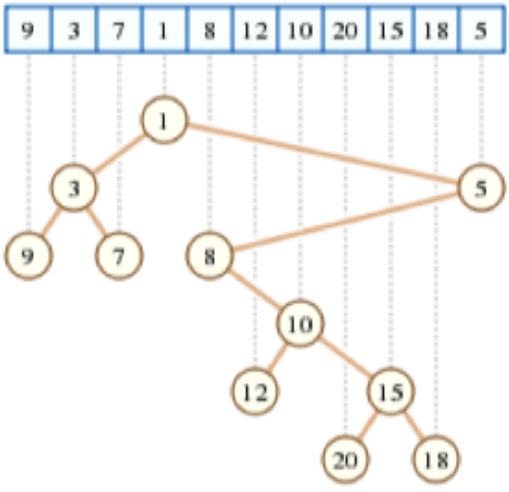
a. Implement **listToBinaryTree** recursively. Do not destroy or otherwise modify the original list. Just use it as a read-only sequence of numbers used to recursively synthesize the corresponding binary tree.

```
static treeNode *listToBinaryTree(const listNode *head) {
```

b. Now implement the iterative version, which should make use of a single **Queue<treeNode **>**. The **Queue<treeNode **>** is used to line up the locations of the **treeNode ***s that need to be considered during the next iteration. You have this and the next page for your solution. (Hint: your **Queue** template has a **size** method that comes in handy.)

```
static treeNode *listToBinaryTree(const listNode *head) {
```

Lab Problem 1: Cartesian Trees



A Cartesian tree is a binary tree structure derived from an array of numbers such that the tree respects the min-heap property (value at the parent is less than the values of the two children) and an inorder traversal of the tree produces the original array sequence. The picture presented on the right (credit to Wikipedia) illustrates how an integer array and the corresponding Cartesian tree are related.

Write a function called **arrayToCartesianTree**, which accepts a reference to a **Vector<int>** of **unique** positive integers, synthesizes the corresponding Cartesian tree, and returns its root.

The problem relies on the existence of the following type definition (which already exists within the provided **cartesian-tree.h** file):

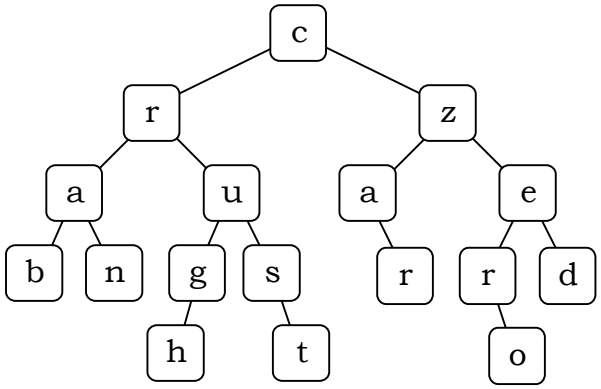
```
struct node {
   int value;
   node *left, *right;
};
```

The starter project includes several files, only one of which you need to change. You should place your implementation of inside **cartesian-tree.cpp**, looking only at **cartesian-tree.h** and **cartesian-tree-test.cpp** if you absolutely need to. Both **cartesian-tree.cpp** and **cartesian-tree-test.cpp** must be included in the project/solution when you compile, run, and test. (Don't worry about freeing the trees.)

Lab Problem 2: Finding Words In Character Trees

You're given the following node definition for a general binary tree of characters.

```
struct node {
   char ch;
   node *left, *right;
};
```



You're interested in writing a function that determines whether or not a particular word can be found along any path from the root to the fringe. So, given the binary tree on the right, your function should be able to confirm that presence of words like **"crab"**, **"ran"**, **"rug"**, "**crust"**, **"rust"**, **"us"**, **"ugh"**, **"czar"**, **"zero"**, and **"zed"**. **"rug"** and **"us"** are noteworthy, because they neither start at the root nor end along the fringe—they're completely internal. (**"arc"** doesn't count, because it's going the wrong way.)

Write a function called **wordExists**, which when given the root of a character tree and a string confirms whether the string can be found along some downward path from the root to any single leaf. In particular, it returns **true** when the supplied text is present and **false** otherwise. You should return as soon as an answer can be determined, which means you should use recursive backtracking to prune the search.

```
bool wordExists(const node *tree, const string& str);
```

Update the **character-tree.cpp** file with your implementation of the **wordExists** function. (You should ignore the **character-tree-test.cpp** file unless you're stumped and want to understand why things aren't working. And don't worry about freeing these trees either.)