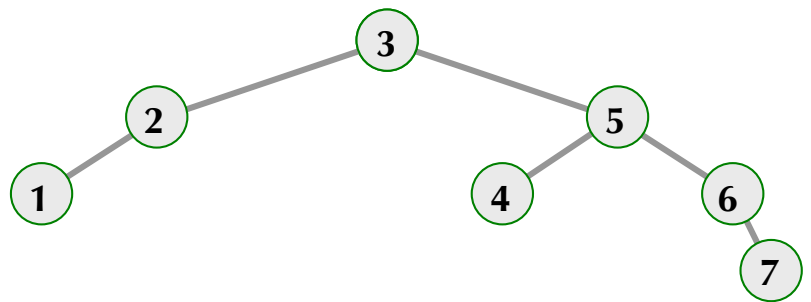


Section Handout

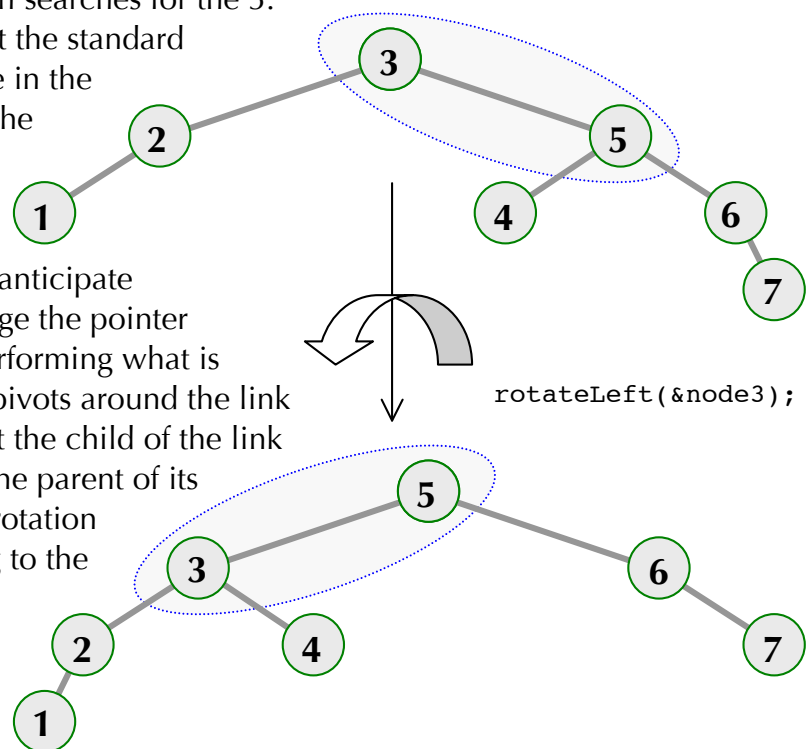
Discussion Problem 1: Tree Rotations

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 an access to a particular element is likely to be followed by many other queries for the same element in the near future. Rotations can be used to bubble frequently access 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.



- 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;
};
```

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

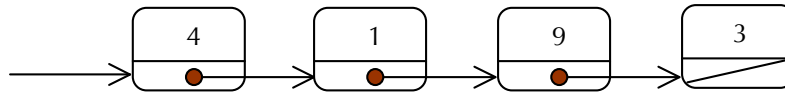
- b) Now, using the **rotateLeft** operation you just wrote, along with its symmetric counterpart **rotateRight** (which you can assume works properly without actually writing it), write a function called **pullToRoot**, which takes a pointer to a binary search tree and bubbles the specified value up 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.

```
/**
 * Function: pullToRoot
 * Usage: pullToRoot(&myTree, 7)
 * -----
 * pullToRoot recursively draws the specified value up
 * to the root of the specified binary search tree. You may
 * assume that the value is in the tree.
 */
```

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

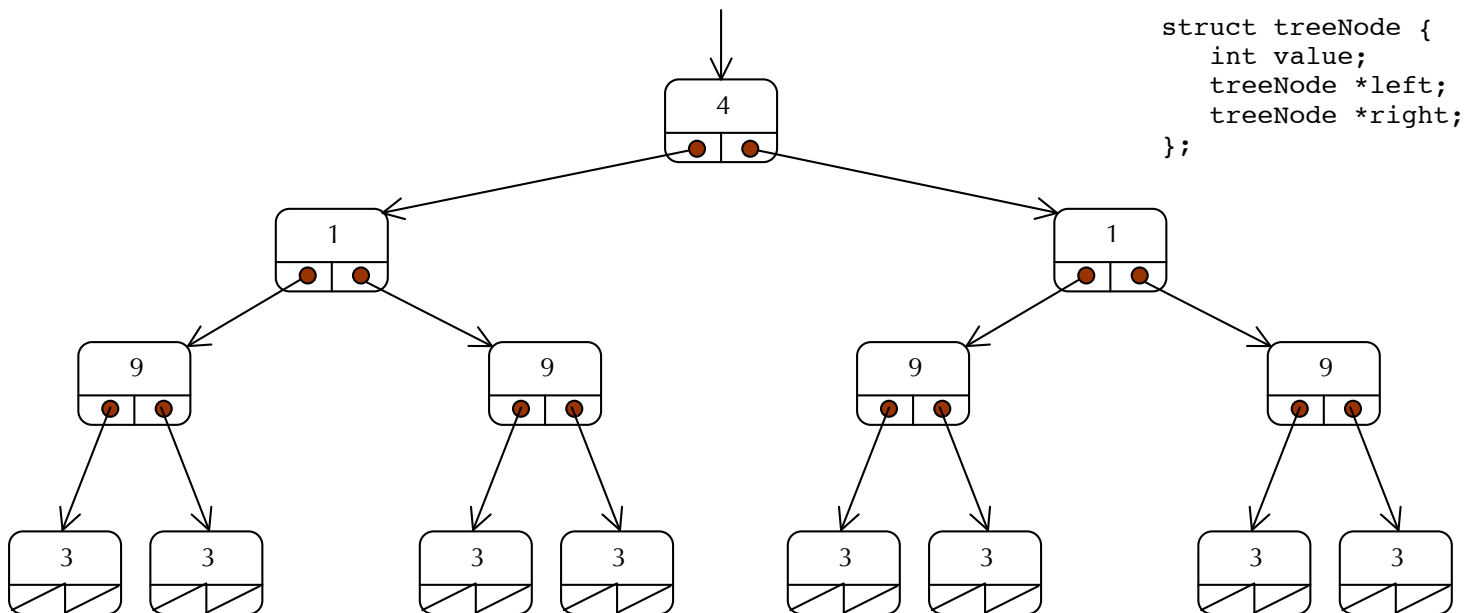
Discussion Problem 2: Binary Tree Synthesis

listToBinaryTree takes a singly linked list of numbers and constructs an independent binary tree where the n th item of the singly linked list occupies all positions at the n th 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, 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.

You're to write two version of this **listToBinaryTree** function: one very recursive function, and one iterative version. 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 and pragmatic to be familiar with each.

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

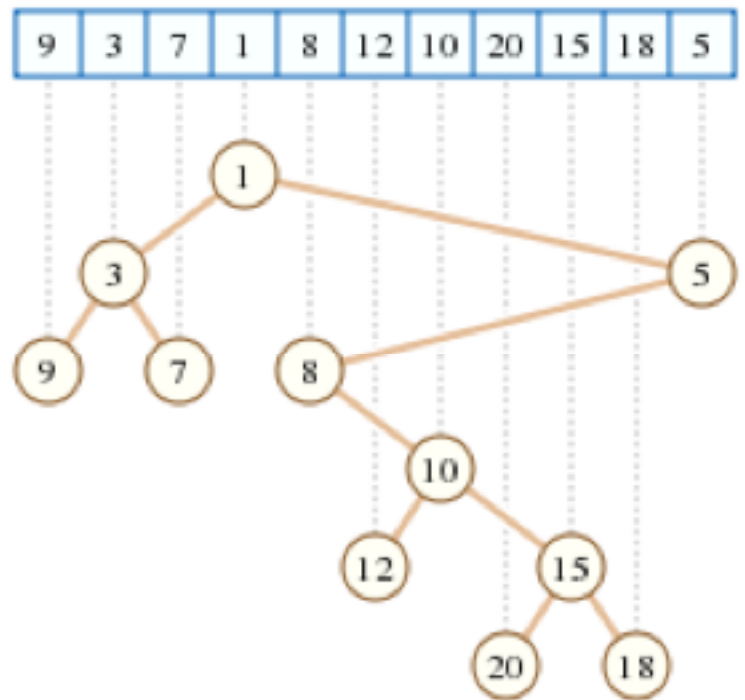
```
treeNode *listToBinaryTree(listNode *head) {
```

- 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.)

```
treeNode *listToBinaryTree(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 present 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):

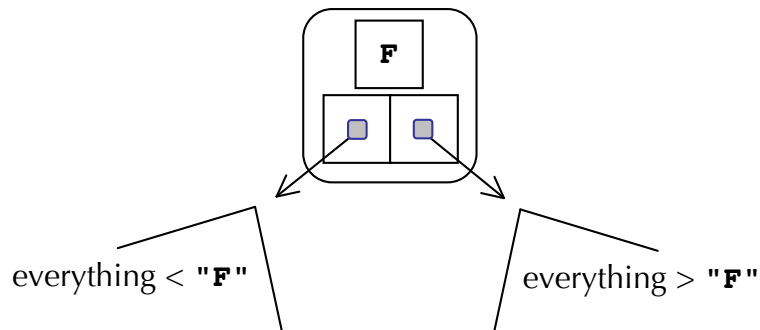
```
struct node {
    int value;
    node *left, *right;
};
```

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.)

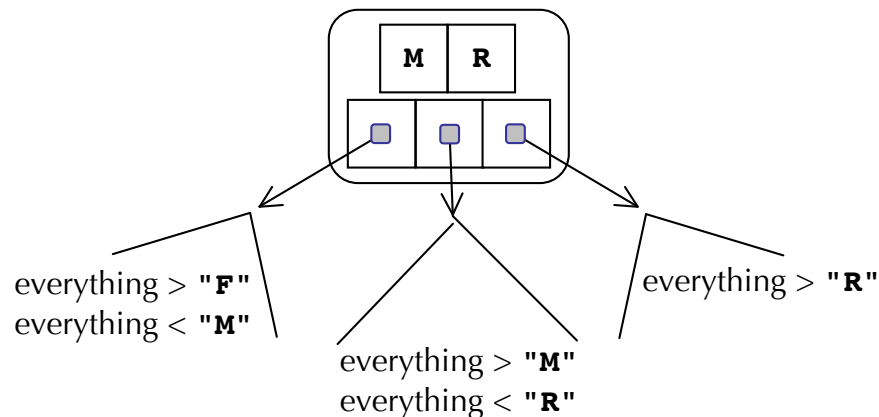
If you're truly up for the challenge, figure out how to implement this in linear time—i.e. in one pass, from left to right, over the integer sequence. But the approach whereby you find the minimum element, create a node around it, and then recur on either side is also acceptable.

Lab Problem 2: Exponential Trees

Exponential trees are similar to binary search trees, except that the **depth** of the node in the tree dictates how many elements it can store. The root of the tree is at depth 1, so it contains 1 string and two children. The root of a tree storing strings might look like this:



If completely full, a node at depth 2—perhaps the right child of the root above—might look like this:



Generally speaking, a node at depth **d** can accommodate up to **d** strings. Those **d** strings are stored in sorted order within a **Vector<string>**, and they also serve to distribute all child elements across the **d + 1** sub-trees.

Exponential trees can be generalized to store any one type, but we'll stick to strings and avoid the template business. We will, however, commit you to the following data structure:

```
struct expnode {
    int depth;           // depth of the node within the tree
    Vector<string> values; // stores up to depth keys in sorted order
    expnode **children; // set to NULL until node is saturated.
```

```
};
```

The lab project includes the definition of a class called **ExponentialTree**, which is a simple string set that's backed by the exponential tree structure we're describing here. The constructor, destructor, and contains method has already been implemented. Your job is to provide a working implementation of the **add** method. You'll want to update the **exponential-tree.cpp** file to include your code, and to use **exponential-tree-test.cpp** to exercise the full functionality. You may need to update **exponential-tree.h** if you elect to add some helper methods.

Some rules:

- Each node must keep track of its **depth**, because the depth alone decides how many elements it can hold, and how many sub-trees it can support.
- The string values are stored in the **values** vector, which maintains all of the strings it's storing in sorted order. We use a **Vector<string>** instead of an exposed array, because the number of elements stored can vary from **0** to **depth**.
- **children** is a dynamically allocated array of pointers to sub-trees. The **children** pointer is maintained to be **NULL** until the **values** vector is full, at which point the **children** pointer is set to be a dynamically allocated array of **depth + 1** pointers, all initially set to **NULL**. Any future insertions that pass through the node will actually result in an insertion into one of **depth + 1** sub-trees.