You may not use any internet devices. You will be graded on functionality—but good style saves time and helps graders understand what you were attempting. You have 180 minutes. We hope this exam is an exciting journey.

Note: DO NOT WRITE in this booklet. Only work in the answer booklet will be graded.
1) Holiday Lights  30 Points

It's the holiday season, which means one thing; holiday lights! You have unboxed your string of lights to hang, but have one problem; though they are multicolored, they are not quite in the color order that you would like. Eager to put your CS 106X skills to work, you try to devise an algorithm to rearrange the light bulbs to be in your preferred order.

You first think about modeling your lights as a linked list, where each node represents one light of a certain color, and contains a link to the next light in the sequence. We could thus define the following struct for a single node in our linked list:

```c
struct LightNode {
    char colorChar;    // color; e.g. 'R' for Red, 'B' for Blue
    LightNode *next;   // pointer to the next light
};
```

Here's one visual example of such a linked list:

![Linked List Diagram]

You may always assume that each node has a set color. Your task is to, for a given linked list of lights, and color string such as "WWGBR" (for White-White-Green-Blue-Red), to try and rearrange the nodes to match the specified color order. **The overall approach we will use is to work from left to right with both the color string and linked list, pulling out the next instance of the light you need from the rest of the list to your right, and moving it to your current place in the list.** For instance, using the linked list illustrated above, here is how the list is modified at each step if we were trying to match to the color string "WRB":

Initially:           R-Y-W-B-W-R-Y

// move first W that is to our right (index 2) to index 0
Step 1 (W, index 0): W-R-Y-B-W-R-Y

// the light we are on is already R - no change
Step 2 (R, index 1): W-R-Y-B-W-R-Y

// move first B that is to our right (index 3) to index 2
Step 3 (B, index 2): W-R-B-Y-W-R-Y

Notice how all untouched nodes remain in the same relative order. Here is another example, again with the linked list illustrated above, but for the color string "RRW":

Initially:           R-Y-W-B-W-R-Y

// move first W that is to our right (index 2) to index 0
Step 1 (W, index 0): W-R-Y-B-W-R-Y

// the light we are on is already R - no change
Step 2 (R, index 1): W-R-Y-B-W-R-Y

// move first W that is to our right (index 3) to index 2
Step 3 (W, index 2): W-R-W-Y-B-R-Y

// the light we are on is already R - no change

// move first R that is to our right (index 5) to index 1

// move first W that is to our right (index 3) to index 2

You should match the specified order as much as possible, even if you may not be able to match it completely, and you should stop as soon as you are not able to find a match. For example, here is how the list is modified at each step for the string "RWBBY":


// the light we are on is already R - no change

// move first W to our right (index 2) to index 1

// move first B to our right (index 3) to index 2

// there is no other B to our right; STOP

// even though there is a yellow, we have already stopped
Step 5 (Y, index 4): NONE

Note that the color order string may be any length, including greater than the length of the linked list. If the color string is empty, for instance, the list should be unmodified. If the string is longer than the length of the list, just rearrange lights as long as you can to fit the color string, until you can no longer do so. **Regardless, though, if you cannot match the entire color string, or if you do match the entire string and still have nodes left over, you should stop and leave the rest of the list unmodified.**

To make it easier to approach, this problem is broken down into two parts and asks you to write two functions.

**Constraints:** For full credit, obey the constraints listed below for all parts of this problem. A violating solution can get partial credit.

- Do not modify the `colorChar` field of any existing nodes.
- Do not create any new nodes by calling `new LightNode(...)`.
- Do not create any data structures (arrays, vectors, sets, maps, etc.)
- You **may** define private helper functions if you like.
A) A handy helper function we will implement first is a function called `findNextLight` that, given a reference to a pointer to the head of a linked list of `LightNode`s, and a color character (such as 'W' or 'B'), removes and returns the leftmost node of that color. In other words, it should rewire the linked list to no longer contain that node, and then return a pointer to that removed node with its next pointer set to `nullptr`. If no node can be found with that color, you should return `nullptr`.

Constraints: For full credit, obey the constraints listed at the top of the problem, as well as the following. A violating solution can get partial credit.

- Your solution should be at worst $O(N)$ time and must make only a single pass over the linked list.
B) Now we can tackle the original problem. Write a function named `rearrangeLights` that takes as parameters the head of a linked list of `LightNode` s by reference, and a color string such as "WWBG", and modifies the linked list to match that color string, as described previously. Your function should return `true` if it was able to match the entire color string, or `false` otherwise. It is fine (and encouraged) for your `rearrangeLights` function to call your `findNextLight` function from part A.
Your friends have organized a gathering to take that all-important holiday photo to share. However, they need to make sure that everyone is placed correctly according to their height; specifically, they say, it would be great to be able to quickly figure out who the $N$-th tallest person in the group is. Eager to put your CS106X skills to work, you volunteer to help!

It turns out that this type of problem is called the *Order Statistics Problem*. Specifically, in a set $S$ of sorted values, the *$k$th order statistic* is the $k$-th smallest value in the set. As an example, the $0$th order statistic is the minimum value, the $1$st order statistic is the second-smallest value, and the $(n-1)$st order statistic is the maximum value.

One way to model this problem is using an *augmented binary search tree*. *Augmented* means that we store some additional information in each tree node in addition to its left pointer, right pointer, and value. For this problem, it turns out that it's helpful to also have each node store the *number of nodes in its left subtree* and the *number of nodes in its right subtree*. We could thus define the following struct for a single node in our tree:

```c
struct TreeNode {
    TreeNode *left;     // pointer to left sub-tree
    TreeNode *right;    // pointer to right sub-tree
    int value;          // value at this node
    int numLeft;        // count of nodes in left sub-tree
    int numRight;       // count of nodes in right sub-tree
}
```

Here's one visual example of such a tree, with each node labeled with its $k$ value:
Complete the following two tasks below to apply this technique to solve the order statistics problem for your friends' heights. Note that this problem has two parts and asks you to write two functions. The following constraints apply for both parts of the problem; a violating solution can get partial credit.

- The functions below must be **recursive** and must not use any **loops**.
- Do not create any **data structures** (arrays, vectors, sets, maps, etc.)
- You **may** define **private helper** functions if you like.
A) It turns out that, for some reason (you're not quite sure why), your friends already have a sorted BST of your friend's heights lying around. However, it has not been labeled to add this augmented information about the subtree sizes (i.e. the numLeft and numRight fields of each node are unspecified, and you should not make any assumptions about their initial values). Write a recursive function named labelBST that fills in these two fields in each node with their correct values. Your function should accept one parameter, which is a pointer to the root node of a BST (in valid BST order) to label. Your function should not return anything. For example, after calling labelBST on the given example tree above, the numLeft and numRight fields should store the values shown in the diagram.

Constraints: For full credit, obey the constraints listed at the top of the problem, as well as the following. A violating solution can get partial credit.

- Your solution should be at worst $O(N)$ time and must make only a single pass over the tree.
B) Now that we have a labeled BST, we can tackle the order statistics problem. Write a recursive function named `findOrderStatistic` that finds the $k$-th order statistic in an augmented binary search tree as described above. Your function should accept three parameters: a pointer to the root node of the BST to search (you may assume it has been properly labeled, as described in part A, is in valid BST order, and is a balanced BST), the integer $k$ representing which order statistic to find, and an integer `value` passed by reference where the order statistic value should be stored. Your function should return true if it found the $k$-th order statistic, and store its value in the reference parameter. If it could not find that order statistic, it should return false, and not update the value of the reference parameter. Here are some examples given the tree above:

- the call of `findOrderStatistic(root, 0, value)` would return true and store the value 3 in the reference parameter.
- the call of `findOrderStatistic(root, 4, value)` would return true and store the value 14 in the reference parameter.
- the calls of `findOrderStatistic(root, -1, value)` and `findOrderStatistic(root, 7, value)` would return false and not update the value in the reference parameter, since these values of $k$ are outside the bounds of the tree.

Constraints: For full credit, obey the constraints listed at the top of the problem, as well as the following. A violating solution can get partial credit.

- Your solution must make only a single traversal through the tree.
- Your solution should be at worst $O(\log N)$ time.
This question consists of two parts. Answer the following questions below.
A) for the graph shown above, write the order that Dijkstra's Algorithm would visit vertexes if it were looking for a path from vertex A to vertex I. Also write the path it would return. Assume that any "for-each" loop over neighbors returns them in ABC order.
B) for the graph shown above, write a valid topological sort of the vertexes. If there are multiple valid sort orders, any will be fine.
As you finish up your fall quarter, your local water utility contacts you in need of urgent help. As temperatures are dropping, they are concerned about their underground pipes freezing over, causing other pipes to take on too much water and bursting! Knowing you are in CS 106X, they are eager for your assistance.

They have already modeled their network of water pipes as a WaterGraph. A WaterGraph, they assure you, is the exact same as the BasicGraph you have already seen, except instead of Vertex and Edge, there are WaterVertex and WaterPipe. These both have the same fields as their cousins Vertex and Edge, with the following additional fields that help model the flow of water:

<table>
<thead>
<tr>
<th>Additional WaterVertex fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double flow</td>
<td>The units of water flowing through this vertex. &gt;= 0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional WaterPipe fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double flow</td>
<td>The units of water flowing through this pipe. &gt;= 0.</td>
</tr>
<tr>
<td>double capacity</td>
<td>The maximum units of water this pipe can carry without bursting. &gt;= 0.</td>
</tr>
</tbody>
</table>

Note that the values above are doubles, so you should use double calculations to ensure exact values. A WaterGraph is guaranteed to be directed and acyclic. Each vertex can take in water from 0 or more pipes, and output it to 0 or more pipes. A vertex’s total input flow, which is the sum of the flow of all its incoming pipes, is thus always equal to its total output flow, which is the sum of the flow of all its outgoing pipes. The exceptions are for source vertexes, which only have outgoing edges to supply water, and sink vertexes, which only have incoming edges to take in water. Flow output is split evenly among all output pipes, meaning that if a vertex has incoming flow 9 and three outgoing pipes, each outgoing pipe will carry flow 3. Here is a visual example of a WaterGraph:
If a pipe freezes, then it **no longer carries water (its flow becomes 0)**. This means **three** things:

1. Its destination vertex loses that flow of water.
2. Its source vertex re-divides its water to flow through its remaining outgoing pipes (if any).
3. These above changes propagate through the graph and may alter additional flows as a result.

As an example, using the diagram above, let's say the pipe going from vertex 0 to vertex 1 freezes over. The graph would change to the following:
Notice how vertex 0 redirects all its water to its only outgoing pipe, which increases node 2 to have a flow of 16. As a result, node 2 partitions this increased flow equally into its two outgoing pipes, each getting flow 8, and so on. This continues propagating through the graph. Additionally, notice how node 1 has lost that flow of water, which causes changes elsewhere in the graph as well.

We must also keep an eye out for **bursting pipes**. A pipe bursts if its flow exceeds its capacity. In the example above, the pipe in red has burst because the increased flow causes it to carry 4 units of water, when its capacity is 3.

Your task is to write a function called `recalculateFlow` that calculates what would happen if a pipe freezes over. Your function should take two parameters; a reference to the `WaterGraph` to look at, and a pointer to a `WaterPipe` that you are pretending has frozen over, and return `nullptr` if no pipes burst as a result, or a pointer to a pipe that would burst as a result. You should traverse the graph and update any relevant nodes and edges to reflect that this pipe is frozen. (Hint: you **should NOT** need to use topological sort here; think about other ways of following a flow update through the graph regardless of topological ordering). If, during the course of the updates, you have determined **for certain** that a pipe would burst, you should immediately stop updating and return a pointer to that pipe. If multiple pipes might burst, it is fine to return a pointer to any one of them. It is also ok to leave the remainder of the graph un-updated. For instance, in the diagram above, vertexes 3 and 5 (and 5's incoming pipe) are not fully updated because we did not reach them before we reached the burst pipe.

To help you approach this problem, consider that there are three ramifications of a pipe freezing over:

1. The edge should be removed from the passed-in graph (it is ok to modify the passed-in graph)
2. Its destination vertex loses this flow of water
3. Its source vertex must re-divide its flow among the remaining outgoing pipes.
You should tackle step 2 before step 3; update the destination vertex and follow this new flow through the graph, then update the source vertex and follow that new flow through the graph. If you update the source vertex first to reflect increased water flow, you might think a pipe would burst, but at this point this would be uncertain. Your algorithm might go on to find out later that water flow from elsewhere also decreased, bringing the water flow back down to a manageable level!

**Constraints:** For full credit, obey the constraints listed below. A violating solution can get partial credit.

- Your solution should be at worst $O(V+E)$ time, where $V$ is the number of vertexes and $E$ is the number of edges.
- Do not create any data structures (arrays, vectors, sets, maps, etc.)
- You may define private helper functions if you like.
This question consists of three parts. Answer the following questions below.
A) Why is inheritance useful in software development? Give a specific example.
B) What is the difference between inheritance and composition? When might you prefer one over the other?
C) Explain how polymorphism impacts the execution of the main function below. As part of your answer, describe the expected behavior of main() without polymorphism, including what output would be printed, and then describe the expected behavior of main() with polymorphism, including what output would be printed.

```cpp
class Shape {  
public:  
    virtual void draw() {  
        performCalculations();  
        drawOnScreen();  
    }  

    virtual void performCalculations() {  
        cout << "Shape calculating" << endl;  
    }  

    virtual void drawOnScreen() {  
        cout << "Drawing shape" << endl;  
    }  
};

class Circle : public Shape {  
public:  
    virtual void performCalculations() {  
        cout << "Circle calculating" << endl;  
    }  

    virtual void drawOnScreen() {  
        cout << "Drawing circle" << endl;  
    }  
};

class Oval : public Circle {  
public:  
    virtual void performCalculations() {  
        Circle::performCalculations();  
        cout << "Oval calculating" << endl;  
    }  
};

class Square : public Shape {  
public:  
    virtual void drawOnScreen() {  
        cout << "Drawing square" << endl;  
    }  
};

int main() {  
    Vector<Shape *> shapes;
```
Shape *square = new Square();
Shape *oval = new Oval();
Shape *circle = new Circle();
Shape *shape = new Shape();

shapes.add(square);
shapes.add(oval);
shapes.add(circle);
shapes.add(shape);

for (Shape *s : shapes) {
    s->draw();
}

return 0;
Define a new class called **UndoStack** that extends **ArrayStack** through inheritance. You should provide the same member functions as the superclass, as well as the following new public member function:

```cpp
virtual void undo();
```
Your subclass represents a stack of integers that allows the user to "undo" the most recent single push or pop action that has been performed on the stack. That is, for a single undo, if the most recent modification made to the stack was to push an element, you should remove that element from the stack; if the most recent modification made to the stack was to pop an element, you should put that element back onto the top of the stack. **You should be able to undo as many actions as have been taken.** (An undo itself does not count as an action; in other words, if the last action was an undo, and you undo again, you would not undo-the-undo; rather, you would undo the second-most-recent action taken.) Your code must work with the existing `ArrayStack` as shown, unmodified.

For example, if the following calls are made on an empty `UndoStack`, the resulting stack contents are shown at right:

```plaintext
UndoStack stack;          // bottom --> top
stack.push(10);           // {10}
stack.push(33);           // {10, 33}
stack.push(24);           // {10, 33, 24}
stack.undo();             // {10, 33}                (undo last push)
stack.undo();             // {10}                    (undo second-last push)
stack.push(45);           // {10, 45}
stack.push(58);           // {10, 45, 58}
stack.pop();              // {10, 45}
stack.undo();             // {10, 45, 58}            (undo last pop)
stack.undo();             // {10, 45}                (undo previous push)
stack.push(58);           // {10, 45, 58}
stack.push(99);           // {10, 45, 58, 99}
stack.push(77);           // {10, 45, 58, 99, 77}
stack.undo();             // {10, 45, 58, 99}        (undo last push)
stack.undo();             // {10, 45, 58}            (undo second-last push)
stack.undo();             // {10, 45}                (undo third-last push)
stack.undo();             // {10}                    (undo fifth-last push)
```

Note that the last `undo()` undoes the `push(45)` command from earlier, because that was the most recent command that was not already undone.

If the stack has just been created and the client tries to call `undo()`, there has not ever been any element pushed or popped from the stack, so you should throw an error using the `error(message)` function.

Write the `.h` and `.cpp` parts of the class separately with a line between to separate them. The majority of your score comes from implementing the correct behavior and using inheritance properly. You should also appropriately utilize behavior inherited from the superclass and not re-implement behavior that already works properly in the superclass.

Recall that subclasses are not able to directly access private members of the superclass.
Simulate the behavior of a HashSet of integers as described and implemented in lecture. Assume the following:

- the hash table (array of buckets) has an initial capacity of 10
- the hashing uses separate chaining to resolve collisions
- the hash function returns the integer key, mod the number of buckets
- rehashing occurs at the end of an add where the load factor is \( \geq 0.75 \) and doubles the number of buckets. You should rehash by going through each bucket by increasing index, from front to back.

Draw a diagram in the text area to show the final state of the hash table after the following operations are performed. Leave a bucket empty if an array element is unused. **Also write the size, capacity, and load factor of the final hash table.**

You do not have to redraw an entirely new hash table after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to redraw the table at various important stages to help earn partial credit in case of an error. If you draw various partial or in-progress diagrams or work, please **clearly indicate your final answer.**

```java
HashSet<int> set;
set.add(3);
set.add(5);
set.add(17);
set.add(10);
set.add(75);
set.add(34);
set.add(36);
set.add(25);
set.remove(8);
set.remove(5);
set.add(24);
if (set.contains(25)) {
    set.add(76);
    set.remove(24);
} else {
    set.add(35);
}
if (set.size() > 7) {
    set.add(26);
}
set.remove(75);
```