Solutions to Final Exam

Problem 1: Holiday Lights (30 Points)

Part A:

```c++
LightNode *findNextLight(LightNode*& head, char lightChar) {
    LightNode *curr = head;
    LightNode *prev = nullptr;
    while (curr != nullptr) {
        if (curr->lightChar == lightChar) {
            // splice this one out
            if (prev == nullptr) {
                head = curr->next;
            } else {
                prev->next = curr->next;
            }
            curr->next = nullptr;
            return curr;
        }
        prev = curr;
        curr = curr->next;
    }
    return nullptr;
}
```

Part B:
There are many possible solutions – we have included iterative and recursive versions below.

Recursive:

```c++
bool rearrangeLights(LightNode*& head, string colorString) {
    if (colorString.size() == 0) return true;
    if (!head) return false;
    if (head->lightChar == colorString[0]) {
        // This light is ok - work on the remaining ones
        return rearrangeLights(head->next, colorString.substr(1));
    }
    // search for light we need
    LightNode *node = findNextLight(head->next, colorString[0]);
    if (!node) return false;
    node->next = head;
    return true;
}
```
Iterative:

```cpp
bool rearrangeLights(LightNode*& head, string colorString) {
    if (colorString.size() == 0) return true;
    else if (!head) return false;

    LightNode *curr = head;
    LightNode *prev = nullptr;
    while (colorString.size() > 0) {
        // Find the light we need and move it
        if (curr->lightChar != colorString[0]) {
            LightNode *node = findNextLight(curr->next, colorString[0]);
            if (!node) return false;
            node->next = curr;
            if (prev != nullptr) {
                prev->next = node;
            } else {
                head = node;
            }
            prev = node;
        } else {
            // Just advance to the next light
            prev = curr;
            curr = curr->next;
        }
        colorString.erase(0, 1);
    }
    return true;
}
```

Problem 2: Order Statistics (30 Points)

Part A:

```cpp
void labelBST(TreeNode *node) {
    if (node) {
        labelBST(node->left);
        labelBST(node->right);
    }
}
```
if (node->left == nullptr) {
    node->numLeft = 0;
} else {
    node->numLeft = node->left->numLeft
    + node->left->numRight + 1;
}

if (node->right == nullptr) {
    node->numRight = 0;
} else {
    node->numRight = node->right->numLeft
    + node->right->numRight + 1;
    }
}

Part B:

```cpp
bool findOrderStatistic(TreeNode *node, int orderStatistic, int& value) {
    if (node == nullptr) {
        return false;
    } else if (orderStatistic == node->numLeft) {
        value = node->value;
        return true;
    } else if (orderStatistic < node->numLeft) {
        return findOrderStatistic(node->left, orderStatistic, value);
    } else { // orderStatistic > node->numLeft
        return findOrderStatistic(node->right, orderStatistic - node->numLeft - 1, value);
    }
}
```

Problem 3: Graphs (20 Points)

Part A:
Vertex visit order: {A, D, E, B, C, F, H, G, I}
Path returned: {A, E, C, F, H, G, I}

Part B:
There are multiple valid topological sorts:
{G, D, H, E, A, B, C, F}
{G, D, H, E, B, A, C, F}
{D, G, H, E, A, B, C, F}
{D, G, H, E, B, A, C, F}
{D, G, H, E, B, C, A, F}
{D, G, H, E, B, C, F, A}
Problem 4: Frozen Pipes (35 Points)
There are many possible solutions; here is one of them.

WaterPipe *recalculateFlow(WaterGraph& graph,
    WaterPipe *frozenPipe) {
    double frozenPipeFlow = frozenPipe->flow;

    // Step 1: propagate the lack of flow
    updatePipeWithFlowChange(graph, frozenPipe,
        -1.0 * frozenPipeFlow);

    // Step 2: propagate the extra flow that can’t
    // traverse this pipe
    WaterNode *frozenPipeStart = frozenPipe->start;
    double additionalFlowPerPipe = frozenPipeFlow /
        (graph.getArcSet(frozenPipeStart).size() - 1.0);

    for (WaterPipe *e : graph.getArcSet(frozenPipeStart)) {
        if (e != frozenPipe) {
            WaterPipe *burstPipe = updatePipeWithFlowChange(
                graph, e, additionalFlowPerPipe);
            if (burstPipe) {
                graph.removeArc(frozenPipe);
                return burstPipe;
            }
        }
    }
    graph.removeArc(frozenPipe);
    return nullptr;
}

/* A helper function that updates the graph to reflect that
 * this pipe should have its flow changed by this amount.
 */
WaterPipe *updatePipeWithFlowChange(const WaterGraph& graph,
    WaterPipe *pipe, double flowChange) {

    // Update this edge
    pipe->flow += flowChange;
    if (pipe->flow > pipe->capacity) {
        // Burst pipe!
        return pipe;
    }
}
// Propagate the change to its destination
WaterNode *pipeEnd = pipe->finish;
pipeEnd->flow += flowChange;

double flowChangePerPipe = flowChange / 
    (1.0 * graph.getArcSet(pipeEnd).size());
for (WaterPipe *currEdge : graph.getArcSet(pipeEnd)) {
    WaterPipe *burstPipe = updatePipeWithFlowChange(graph,
        currEdge, flowChangePerPipe);
    if (burstPipe) {
        return burstPipe;
    }
}
return nullptr;

Problem 5: Inheritance and Polymorphism (20 Points)

Part A:
We were looking for answers that mention how inheritance can reduce code duplication by being able to extend the existing behavior in other classes and automatically be able to utilize their behavior. Other possible benefits include being able to enforce certain behavior in other types via superclasses and subclasses, or write cleaner code by having more specific code only in the subclasses.

There are many possible examples of inheritance. One example is from the CS106XCell assignment; some functionality, such as getValue() and setValue(), were specified in the Expression superclass, and for that reason were automatically available in all Expression subclasses without having to copy-paste the code into each one.

Part B:
Inheritance represents an “is-a” relationship, meaning that one object “is an” instance of another, such as Dog being an instance of an Animal. Thus, inheritance means that one object is extending the behavior of another. Composition represents a “has-a” relationship, meaning that one object “has an” instance of another within it, such as a Room having an instance of a Lamp. Thus, composition means that one object is utilizing the behavior of another in its implementation.

It is best to use inheritance for two types when one is connected to another, and all functionality in one type makes sense in another type; for instance, everything an Animal can do, a Dog can do. It is best to use composition for two types when one is connected to another, but not all existing functionality in one type makes sense in another type; for instance, it doesn’t make sense to turn a Room on and off like you do a Lamp.

Part C:
Because of polymorphism, each object in the shapes Vector is able to execute its own subclass-specific implementation of draw(), even though they are all represented as pointers to Shapes. Here is the output of the code, with polymorphism:

```
Shape calculating
Drawing square
Circle calculating
Oval calculating
Drawing circle
Circle calculating
Drawing circle
Shape calculating
Drawing shape
```

Without polymorphism, we would expect that, because all elements in the Vector are pointers to Shapes, they would all execute Shape’s draw() method, and not execute the specific subclass behavior. Here is the output of the code, without polymorphism:

```
Shape calculating
Drawing shape
Shape calculating
Drawing shape
Shape calculating
Drawing shape
Shape calculating
Drawing shape
```

Problem 6: UndoStack (25 Points)

There are many possible solutions, including using a struct, as below, or having two stacks, one for action types and one for values.

```
UndoStack.h

struct Action {
   int value;
   std::string type;  // "PUSH" or "POP"
};

class UndoStack : public ArrayStack {
public:
   virtual int pop();
   virtual void push(int n);
   virtual void undo();

private:
   Stack<Action> pastActions;
```
UndoStack.cpp

```cpp
int UndoStack::pop() {
    int pop = ArrayStack::pop();
    pastActions.push({pop, "POP"});
    return pop;
}

void UndoStack::push(int n) {
    ArrayStack::push(n);
    pastActions.push({0, "PUSH"});
}

void UndoStack::undo() {
    if (pastActions.isEmpty()) {
        error("no action to undo");
    } else {
        Action lastAction = pastActions.pop();
        if (lastAction.type == "PUSH") {
            // important: don't count undo as action
            // (alt approach: immediately remove added action)
            ArrayStack::pop();
        } else if (lastAction.type == "POP") {
            // important: don't count undo as action
            // (alt approach: immediately remove added action)
            ArrayStack::push(lastAction.value);
        }
    }
}
```

Problem 7: Hashing (20 Points)

Note: we did not mark off if the order of elements in the linked lists did not match exactly.

Size = 8
Capacity = 20
Load Factor = 8/20 = 0.4