Practice CS106B Final Exam

This exam is closed-book and closed-computer. You may have a double-sided, 8.5" × 11" sheet of notes with you when you take this exam. You may not have any other notes with you during the exam. You may not use any electronic devices (laptops, cell phones, etc.) during the course of this exam. Please write all of your solutions on this physical copy of the exam.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem specifically requires an efficient solution.

This exam is "self-contained" in the sense that if you're expected to reference code from the lectures, textbook, assignments, or section handouts, we'll explicitly mention what that code is and provide sufficient context for you to use it. There's a reference sheet at the back of the exam detailing the library functions and classes we've discussed so far.

SUNetID: _____

Last Name: _____

First Name: _____

Section Leader: _____

I accept both the letter and the spirit of the Honor Code. I have not received any unpermitted assistance on this test, nor will I give any. I do not have any advance knowledge of what questions will be asked on this exam. My answers are my own work. Finally, I understand that the Honor Code requires me to report any violations of the Honor Code that I witness during this exam.

(signed) _____

You have three hours to complete this exam. There are 48 total points.

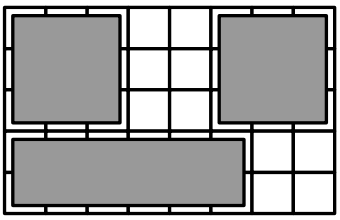
Question	Points	Grader
(1) Recursive Problem-Solving	/ 12	
(2) Linear Structures	/ 12	
(3) Tree Structures	/ 12	
(4) Graphs and Graph Algorithms	/ 12	
	/ 48	

Problem One: Recursive Problem-Solving (12 Points)

Moving Day! ***(Recommended time: 45 minutes)***

It's moving day! You've packed all your things into boxes, you've rented a moving truck, and all that's left to do is get everything to fit in the truck. The challenge is figuring out how exactly you're going to do that.

For simplicity, we'll assume that the length and width of each box is a whole number of feet, and that the length and width of the truck is also a whole number of feet. We can represent the available space in the truck as `Grid<bool>` where each cell in the grid represents one square foot of space in the truck, with `true` representing a filled location and `false` an empty location. For example, below is one possible arrangement and its corresponding array representation:



Truck Layout

T	T	T	F	F	T	T	T
T	T	T	F	F	T	T	T
T	T	T	F	F	T	T	T
T	T	T	T	T	T	F	F
T	T	T	T	T	T	F	F

Array Representation

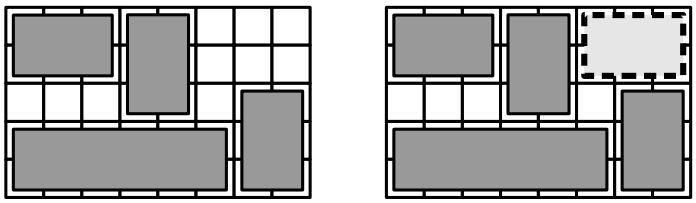
Your job is to write a method

```
bool canPlaceAllBoxes(int width, int height, const Vector<Box>& boxes);
```

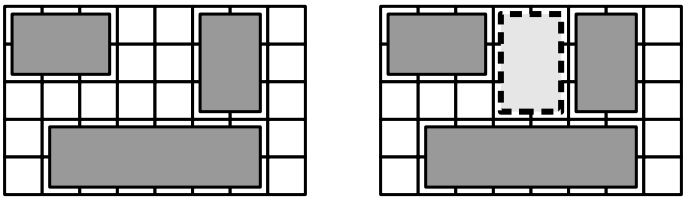
that accepts as input the dimensions of the truck and a list of all the boxes you need to place, then returns whether it's possible to fit all the boxes into the truck. (The `Box` type is defined on the next page.) In the course of solving this problem, you should assume the following:

- Boxes cannot be stacked on top of one another.
- The box must have its sides parallel to the sides of the truck, but may be rotated 90° left or right.

To elaborate on that last point, a 3' × 2' box could fit into the indicated space here:



It could also fit into this space:



To make things a bit easier, you can assume you have access to a function

```
bool fitsAt(int width, int height, int row, int col, const Grid<bool>& truck);
```

that takes as input the width and height of a box and a position in the truck, then returns whether a box of those dimensions would fit at that position.

Syntax refresher: you can select an element from a grid by writing *grid*`[`*row*`][`*col*`]`. You can query the number of rows and columns by writing *grid*`.numRows()` and *grid*`.numCols()`, and you can create a grid of a certain size with `Grid<bool>` *grid*`(`*numRows*`,` *numCols*`);`. Those elements will be uninitialized.

You can assume the truck is initially empty. You don't need to worry about efficiency.

```
struct Box {
    int width;
    int height;
};

/* This function is provided to you. You don't need to implement it. */
bool fitsAt(int width, int height, int row, int col, const Grid<bool>& truck);

bool canPlaceAllBoxes(int width, int height, const Vector<Box>& boxes) {
```

(extra space for your answer to Problem One, if you need it.)

Problem Two: Linear Structures (12 Points)

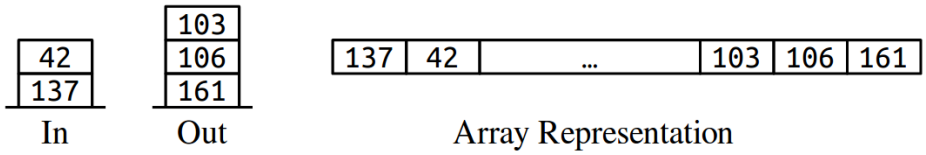
One Queue, Two Stacks, One Array ***(Recommended time: 45 minutes)***

In lecture, we talked about one possible implementation of a queue called a ***two-stack queue***. In this representation, elements are stored in two stacks, an ***in*** stack and an ***out*** stack. To enqueue an element, you simply push the element in question onto the in stack. There are two cases for dequeuing:

1. If the out stack is nonempty, pop the top element and return it.
2. Otherwise, pop each element from the in stack and push it onto the out stack. Then, pop the top of the out stack and return it.

This approach is extremely efficient – each operation works in amortized time O(1).

You've seen how to implement stacks using dynamic arrays. In this particular case, there's a clever trick you can use to store both stacks *in the same dynamic array*. The idea is to have the in stack start at the left side of the array and grow rightward and to have the out stack start at the right side of the array and grow toward the left. For example, here's how we'd represent one combination of an in stack and an out stack:



Unless the array is completely at capacity, there will be a gap between the elements making up the in stack and the elements making up the out stack.

Your task in this problem is to implement a `TwoStackQueue` type using the above technique. You must do all of your own memory management in this problem, so you'll need to dynamically allocate and deallocate your memory as necessary.

You're welcome to add any data members, helper types, or private member functions to this class, but you must not modify the public interface provided to you.

```
class TwoStackQueue {
public:
    TwoStackQueue();
    ~TwoStackQueue();

    void enqueue(int value); // Enqueues a value
    int  dequeue();          // Dequeues a value; triggers an error if empty.

    int  size() const;    // Returns the size of the queue
    bool isEmpty() const; // Returns whether the queue is empty.

private:
    /* Add any private data members, member functions, or member types here. */




};
```

```
TwoStackQueue::TwoStackQueue() {

}
TwoStackQueue::~TwoStackQueue() {

}
```

```
void TwoStackQueue::enqueue(int value) {
```

```
int TwoStackQueue::dequeue() {
```

```
int TwoStackQueue::size() const {




}
bool TwoStackQueue::isEmpty() const {




}
                    /* Extra space for any remaining functions */
```

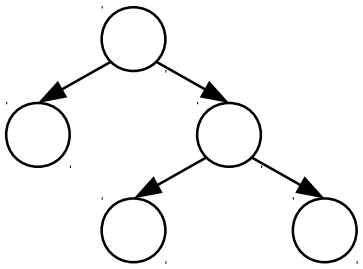
Problem Three: Tree Structures (12 Points)

Encoding Trees, Literally ***(Recommended time: 45 minutes)***

This question has three parts.

On Assignment 6, you saw Huffman encoding as one possible way that you could compress *strings* of text. This question explores how you might go about compressing *binary trees*.

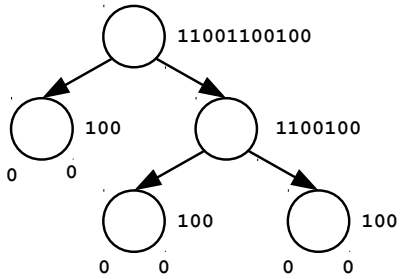
Suppose that you have a binary tree, like this one:



You are interested in transmitting the *shape* of this binary tree to someone else (that is, the structure of the tree rather than the values it contains). You want to do so using the fewest number of bits possible. It turns out that it is possible to encode the shape of any n-node binary tree using only $2n + 1$ bits. The idea is as follows:

- An empty tree with no nodes is encoded with the bit `0`.
- A nonempty tree is encoded as a `1` bit, followed by the bitwise encoding of the root's left child, followed by the bitwise encoding of the root's right child.

For example, the above binary tree has encoding `11001100100`. You can see this here:



Given these binary trees:

The left tree has encoding `1101000`, and the right tree has encoding `110011000`.

Feel free to tear out this page as a reference.

Suppose that each node in a binary tree is represented using the following `struct`:

```
struct Node {
    Node* left;
    Node* right;
};
```

Your first task is to implement the following pair of functions:

```
void compressTree(Node* root, obitstream& out);

Node* decompressTree(ibitstream& in);
```

The `compressTree` function accepts as input a pointer to the root of a binary tree and an `obitstream` where the compressed representation should be written, then writes out a compressed version of the binary tree to the output stream. The `decompressTree` function takes as input an `ibitstream` containing a compressed representation of the binary tree, then rebuilds the encoded tree and returns a pointer to its root. As a hint, these functions can be implemented beautifully using recursion.

As a reminder, you can write a bit to an `obitstream` by calling `out.writeBit(bit)`, where `bit` is either `0` or `1`. You can read one bit from an `ibitstream` by calling `in.readBit()`.

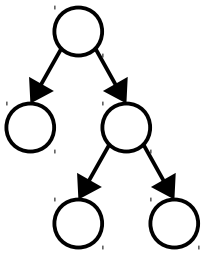
```
void compressTree(Node* root, obitstream& out) {
```

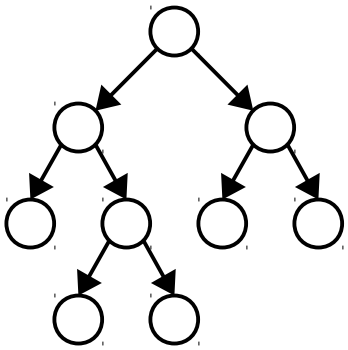
```
Node* decompressTree(ibitstream& in) {
```

The above encoding scheme is very efficient when compressing arbitrary binary trees. However, if we know more about the structure of the binary tree being encoded, we can use even fewer bits.

Suppose, for example, that we want to encode the *shape* of a Huffman encoding tree. It turns out that it is possible to do so using only *n* bits of storage, where *n* is the number of nodes in the tree. The reason for this is that Huffman encoding trees are *pure binary trees* – every node has either 0 or 2 children, and never just one child.

Describe (either in plain English or in code) a way of encoding the shape of a pure binary tree with *n* nodes using exactly *n* bits. You can assume that the tree is nonempty (that is, there is at least one node). You should justify why your system uses exactly *n* bits, but you don't need to formally prove this. To help us better understand your system, write out how your binary encoding system would encode these two pure binary trees:



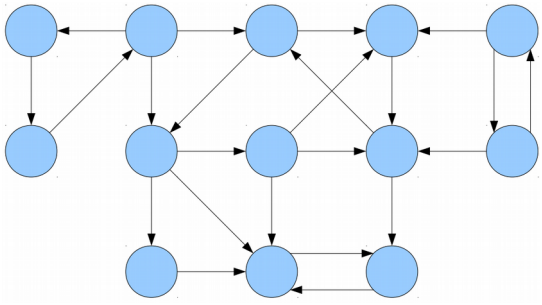


Problem Four: Graphs and Graph Algorithms (12 Points)

Traffic Planning ***(Recommended time: 45 minutes)***

This problem has two parts.

You're on the transportation planning board for a newly-planned city. The board has just put together a recommendation for the city's transportation grid, which includes a number of one-way streets. You're a bit nervous about the grid, though. What if the one-way streets are set up in a way that causes them to trap people somewhere? For example, imagine the city's transportation grid is like the one shown here:



If you start at the upper-left corner of this transportation network, it's possible to travel down to the bottom row, but once you're there there's no way back where you started! Using what you've learned in CS106B this quarter, you decide to check for yourself whether the transportation grid is safe.

For the purposes of this problem, we'll say that a transportation network is a ***good network*** if, starting at any position in the network, it's possible to get to each other position. It's a ***bad network*** if there's a pair of nodes *A* and *B* where either there's no way to get from *A* to *B* or no way to get from *B* to *A* (or both!)

You could in principle solve this problem by running a depth-first search starting at each node in the transportation network to see whether each other node is reachable, but that approach doesn't scale very well as the number of nodes increases. There's a much faster algorithm you can use instead:

1. Choose any node in the graph.
2. Run a depth-first search starting at that node.
3. Reverse all the edges in the graph, then run another depth-first search starting at that node.
4. Return whether every node in the graph was discovered by both of the depth-first searches.

Before we ask you to code up this algorithm, use the space below to explain, as concisely as is possible, why this algorithm correctly checks whether the transportation network is a good network. Specifically:

- Explain why, if this algorithm reports the network is good, it means that starting *anywhere* in the network, you can get to any other location in the network. Be specific.
- Explain why, if this algorithm reports that the network is not good, it means that there's some node in the network that can't reach every other location. Make specific reference to which steps in the algorithm would ensure this. Be specific.

Now, we'd like you to code up this algorithm. We've chosen to represent the transportation grid using the following type:

```
using Graph = Map<string, Set<string>>;
```

Here, keys are nodes, and each node's associated value is the set of nodes immediately adjacent to it. Your task is to write a function

```
bool isGoodNetwork(const Graph& roadNetwork);
```

that takes as input a road network and uses the algorithm described on the preceding page to determine whether or not it's a good network. Here are some notes on this problem:

- You can assume that the transportation grid has at least one node.
- Some of the roads in the transportation grid are one-way. This means that just because there's an edge from some node *A* to some node *B*, you cannot say that there's an edge from *B* back to *A*.
- The `Map::keys()` function returns a `Vector` of all the keys in the map. You can use this to choose a node out of the graph.

```
bool isGoodNetwork(const Graph& roadNetwork) {
```

(Extra space for your answer to Problem Four, if you need it.)

C++ Library Reference Sheet

Lexicon

```
Lexicon lex; Lexicon english(filename);
lex.addWord(word);
bool present = lex.contains(word);
bool pref = lex.containsPrefix(p);
int numElems = lex.size();
bool empty = lex.isEmpty();
lex.clear();
```

Map

```
Map<K, V> map = {{k1, v1}, … {kn, vn}};
map[key] = value; // Autoinsert
bool present = map.containsKey(key);
int numKeys = map.size();
bool empty = map.isEmpty();
map.remove(key);
map.clear();
Vector<K> keys = map.keys();
```

Stack

```
stack.push(elem);
T val = stack.pop();
T val = stack.top();
int numElems = stack.size();
bool empty = stack.isEmpty();
stack.clear();
```

Queue

```
queue.enqueue(elem);
T val = queue.dequeue();
T val = queue.peek();
int numElems = queue.size();
bool empty = queue.isEmpty();
queue.clear();
```

Set

```
Set<T> set = {v1, v2, …, vn};
set.add(elem);
set += elem;
bool present = set.contains(elem);
set.remove(x); set -= x; set -= set2;
Set<T> unionSet = s1 + s2;
Set<T> intersectSet = s1 * s2;
Set<T> difference = s1 - s2;
T elem = set.first();
int numElems = set.size();
bool empty = set.isEmpty();
set.clear();
```

Vector

```
Vector<T> vec = {v1, v2, …, vn};
vec.add(elem);
vec += elem;
vec.insert(index, elem);
vec.remove(index);
vec.clear();
vec[index]; // Read/write
int numElems = vec.size();
bool empty = vec.isEmpty();
vec.subList(start, numElems);
```

TokenScanner

```
TokenScanner scanner(source);
while (scanner.hasMoreTokens()) {
   string token = scanner.nextToken();
   …
}
scanner.addWordCharacters(chars);
```

string

```
str[index]; // Read/write
str.substr(start);
str.substr(start, numChars);
str.find(c); // index or string::npos
str.find(c, startIndex);
str += ch;
str += otherStr;
str.erase(index, length);
```

ifstream

```
input.open(filename);
input >> val;
getline(input, line);
```

GWindow

```
GWindow window(width, height);
gw.drawLine(x0, y0, x1, y1);
pt = gw.drawPolarLine(x, y, r, theta);
```

GPoint

```
double x = pt.getX();
double y = pt.getY();
```

General Utility Functions

```
int getInteger(optional-prompt);
double getReal(optional-prompt);
string getLine(optional-prompt);
int randomInteger(lowInclusive,
                  highInclusive);
double randomReal(lowInclusive,
                  highExclusive);
error(message);
x = max(val1, val2); y = min(val1, val2);
stringToInteger(str); stringToReal(str);
integerToString(intVal);
realToString(realVal);
```