Practice CS106B Final Exam Solutions

Problem One: Recursive Problem-Solving (12 Points)

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

Our solution makes use of a bunch of helper functions to handle subtasks like filling in the region taken up by the box and making an empty truck. Fundamentally, the recursion works by going one box a time, try out putting that box at each position and in each orientation, and seeing if any of those choices lead to a valid solution. Here's what it looks like:

```
bool canPlaceAllBoxes(int width, int height, const Vector<Box>& boxes) {
    Grid<bool> truck = makeTruck(width, height);
    return canPlaceEverythingIn(truck, boxes, 0);
}

bool canPlaceEverythingIn(Grid<bool>& truck, const Vector<Box>& boxes,
                          int index) {
    /* Base case: If we placed everything, we're done! */
    if (index == boxes.size()) return true;

    /* Otherwise, the next box has to go somewhere. Try each position and
     * orientation.
     */
    Box next = boxes[index];
    for (int row = 0; row < truck.numRows(); row++) {
        for (int col = 0; col < truck.numCols(); col++) {
            if (fitsAt(next.width, next.height, row, col, truck)) {
                /* See if this works at this position. */
                fillBoxAt(row, col, next.width, next.height, truck, true);
                if (canPlaceEverythingIn(truck, boxes, index+1)) return true;
                fillBoxAt(row, col, next.width, next.height, truck, false);
            }
            /* Rotate 90 degrees. */
            if (fitsAt(next.height, next.width, row, col, truck)) {
                fillBoxAt(row, col, next.height, next.width, truck, true);
                if (canPlaceEverythingIn(truck, boxes, index+1)) return true;
                fillBoxAt(row, col, next.height, next.width, truck, false);
            }
        }
    }
    /* We tried everything, and nothing worked. */
    return false;
}

                    /* … continued on the next page … */
```

```
/* Creates an empty truck of the specified dimensions. */
Grid<bool> makeTruck(int width, int height) {
    Grid<bool> result(height, width); // Confusing, but correct
    for (int row = 0; row < result.numRows(); row++) {
        for (int col = 0; col < result.numCols(); col++) {
            result[row][col] = false;
        }
    }
    return result;
}

/* Fills the specified rectangle in the truck with the given value. */
void fillBoxAt(int row, int col, int width, int height, Grid<bool>& truck,
               bool value) {
    for (int r = row; r < row + height; r++) {
        for (int c = col; c < col + width; c++) {
            truck[r][c] = value;
        }
    }
}
```

Problem Two: Linear Structures (12 Points)

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

Our solution to this problem maintains two sizes, the size of the in stack and the size of the out stack, which it uses to keep track of where to put elements in the in stack, where to remove elements from the out stack, and where to copy things from when resizing the array.

There are a couple of nuances that come up here that don't come up with a single stack. Resizing the array and copying the elements over now requires copying from both the front and the back; simply copying all the elements of the array from their old position to the same position in the new array doesn't work (do you see why?) There are also some beautiful parts, like how we transfer elements from the in stack to the out stack when the out stack is empty.

Here's our solution:

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

    void enqueue(int value);
    int dequeue();

    int size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  inSize;
    int  outSize;

    void grow();
    void transferStacks();
};

TwoStackQueue::TwoStackQueue() {
    allocatedSize = 8; // They said I could pick anything, so I picked 8.
    inSize = outSize = 0;
    elems = new int[allocatedSize];
}

TwoStackQueue::~TwoStackQueue() {
    delete[] elems; // Be free, little array!
}

void TwoStackQueue::enqueue(int value) {
    /* We are out of space if the entire array is used. */
    if (size() == allocatedSize) grow();

    /* Push onto the in stack. */
    elems[inSize] = value;
    inSize++;
}

                        /* … continued on the next page … */
```

```
void TwoStackQueue::grow() {
    int newSize = allocatedSize * 2;
    int* newElems = new int[newSize];

    /* Copy the in stack to the front of the new array. */
    for (int i = 0; i < inSize; i++) {
        newElems[i] = elems[i];
    }

    /* Copy the out stack to the back of the new array. */
    for (int i = 0; i < outSize; i++) {
        newElems[newSize - 1 - i] = elems[allocatedSize - 1 - i];
    }

    /* Replace the old array with this new one. */
    delete[] elems;
    elems = newElems;

    allocatedSize = newSize;
}

int TwoStackQueue::dequeue() {
    if (isEmpty()) error("Nothing to see here, folks. Move along.");

    /* If the out stack is empty, we need to pop everything off the in stack and
     * push it into the out stack.
     */
    if (outSize == 0) {
        transferStacks();
    }

    int result = elems[allocatedSize() - outSize];
    outSize--;
    return result;
}

/* Moves everything from the in stack to the out stack. */
void TwoStackQueue::transferStacks() {
    while (inSize > 0) {
        elems[allocatedSize - 1 - outSize] = elems[inSize - 1];
        inSize--;
        outSize++;
    }
}

int TwoStackQueue::size() const {
    return inSize + outSize;
}

bool TwoStackQueue::isEmpty() const {
    return size() == 0;
}
```

Problem Three: Tree Structures (12 Points)

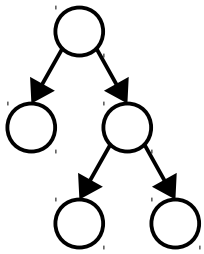
Encoding Encoding Trees ***(Recommended time: 45 minutes)***

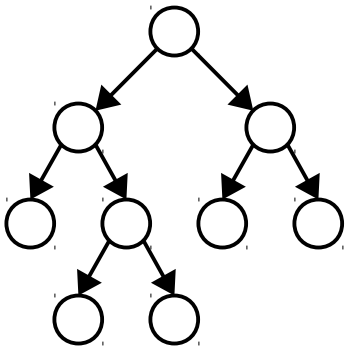
Our code for the first part of this problem works by just going case by case and letting recursion handle all the heavy lifting:

```
void compressTree(Node* root, obitstream& out) {
    if (root == nullptr) {
        out.writeBit(0);
    } else {
        out.writeBit(1);
        compressTree(root->left, out);
        compressTree(root->right, out);
    }
}

Node* decompressTree(ibitstream& in) {
    if (in.readBit() == 0) {
        return nullptr;
    }
    /* Otherwise, the bit was a 1. */
    Node* root = new Node;
    root->left = decompressTree(in);
    root->right = decompressTree(in);
    return root;
}
```

For the second part of the problem, one encoding scheme you could use for a full binary tree would be to write a 0 for a node with no children, then to write a 1 for a node with two children, followed by the encodings of its left and right subtrees. The tree on the left, for example, would have encoding 10100, and the tree on the right would have encoding 110100100.





This system uses a total of n bits for an n-bit tree, since each node contributes a single bit to the total. It can be read or written using a very slight modification on the above encoding and decoding algorithms.

Problem Four: Graphs and Graph Algorithms (12 Points)

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

A depth-first search started at some node in a graph will find all nodes reachable from that node. That means that if we run a DFS starting at some start node and reach every other node in the graph, we know that the start node can reach every other node.

When we reverse the edges in the graph and do a secondary DFS, the set of nodes we reach are precisely the set of nodes that can reach the start node. The reason for this is that each path we discover in the reverse graph is a path that, in the original graph, starts at some node and ends at the start node. This means that if we do a DFS in this reverse graph and find every node, we see that every node in the graph has a path to the start node we chose.

So suppose that both DFSs discover each node. If we want to find a path from some node *A* to some node *B* in the graph, we can obtain one by starting at *A*, taking a path to the start node (which we know exists), then going from that node to node *B*.

On the other hand, if either DFS fails to discover a node, it means either that the start node can't reach everything (if the first DFS fails) or that there's some node that can't reach the start node (if the second DFS fails.)

Some code for implementing this search is shown here:

```
bool isGoodNetwork(const Graph& graph) {
    string startNode = graph.keys()[0];
    return canReachEverything(startNode, graph) &&
           canReachEverything(startNode, reverseOf(graph));
}

bool canReachEverything(const string& node, const Graph& graph) {
    Set<string> visited;
    dfsFrom(node, graph, visited);
    return visited.size() == graph.size();
}

void dfsFrom(const string& node, const Graph& graph, Set<string>& visited) {
    if (visited.contains(node)) return;
    visited += node;

    for (string next: graph[node]) {
        dfsFrom(next, graph, visited);
    }
}

Graph reverseOf(const Graph& graph) {
    Graph result;
    for (string node: graph) {
        for (string endpoint: graph[node]) {
            result[endpoint] += node;
        }
    }
    return result;
}
```