Section Solutions 9

A ***huge*** thanks to Ali Malik for putting together most of the code here!

Week One: Basic Recursion and String Processing

1. Write a function that reverses a string "in-place." That is, you should take the string to reverse as a reference parameter and modify it so that it ends up holding its reverse. Your function should use only O(1) auxiliary space.

Here's one solution:

```
void reverseInPlace(string& str) {
    /* We need to make sure to only loop up to halfway though the string
     * or we will end up with the original string! Try it out on
     * paper if this isn't clear.
     */
    for (int i = 0; i < str.length() / 2; i++) {
        swap(str[i], str[str.length () - 1 - i]); // Question: Why is the -1 here?
    }
}
```

To see that this only uses O(1) storage space, we can account for where all the memory we're using is coming from. We need space for the integer `i`, plus a little stack space for the `swap` function, but aside from that there's no major allocations or storage required.

2. Imagine you have a string containing a bunch of words from a sentence. Here's a nifty little algorithm for reversing the order of the words in the sentence: reverse each individual string in the sentence, then reverse the entire resulting string. (Try it – it works!) Go and code this up in a way that uses only O(1) auxiliary storage space.

You can come up with all sorts of different answers to this problem depending on how you define what a word boundary looks like. We've decided to do this by just finding whitespace, but a more intelligent implementation might opt to do more creative checking.

```
/* Essentially the same function as before, except that we specify our own start
 * and end indices so we can reverse parts of a string rather than the whole
 * string at each point. The end index is presumed to be one past the end of the
 * substring in question.
 */
void reverseInPlace(string& str, int start, int end) {
    for (int i = 0; i < (end - start) / 2; i++) {
        swap(str[i], str[end - 1 - i]);
    }
}

void reverseWordOrderingIn(string& sentence) {
    /* Scan across the sentence looking for words. The start variable holds the
     * index of the start point of the current word in the sentence. The variable
     * i holds the index of the character we're currently scanning.
     */
    int start = 0;
    for (int i = 0; i < sentence.length(); i++) {
        if (sentence[i] == ' ') {
            reverseInPlace(sentence, start, i);
            start = i + 1;
        }
    }

    /* We need to account for the fact that there might be a word flush up at the
     * end of the sentence.
     */
    reverseInPlace(sentence, start, sentence.length());

    /* Now reverse everything. */
    reverseInPlace(sentence, 0, sentence.length());
}
```

Again, we can see that we're using O(1) space because we're only using a few temporary variables to hold indices.

3. Write a recursive function to find a good collection point. See if you can solve this with a solution that runs in time O(log *n*). As a hint, think about binary search. You can assume that all elements in the array are distinct.

The key insight here is to look at the two middle elements of the array to see which direction they slope downhill. Imagine that the water would flow to the left. Then, if we look in the first half of the array, we know there has to be a good collection point somewhere to the left, since if the water flows downhill it has to collect somewhere over there. Using this insight, we can modify our binary search to look like this:

```
/* Given an index into the ridge, returns whether the given index is a good
 * collection point. That happens if both of the position's neighbors are higher
 * than the position itself.
 */
bool isGoodPoint(const Vector<double>& heights, int index) {
    /* Handle boundary cases by pretending the boundaries are infinitely high. */
    double left  = (index == 0? INFINITY : heights[index - 1]);
    double right = (index == heights.size() - 1? INFINITY: heights[index + 1]);

    return heights[index] < left && heights[index] < right;
}

/* Return the index of a good collection point in the interval [start, end). Note
 * that start is inclusive and that end is exclusive.
 */
int findCollectionHelper(const Vector<double>& heights, int start, int end) {
    /* Base case: If the midpoint is a collection point, we're done. */
    int mid = start + (end - start) / 2;
    if (isGoodPoint(heights, mid)) return mid;

    /* If we are on a downward facing slope (left is greater than curr)
     * then search to the right of here
     */
    if(mid > 0 && heights[mid - 1] > heights[mid]) {
        return findCollectionHelper(heights, mid + 1, end);
    }
    /* Otherwise we are on upward facing slope; collection point is to the left
     * so we can limit our search to the left range.
     */
    else {
        return findCollectionHelper(heights, start, mid);
    }
}

int findCollectionPoint(const Vector<double>& heights) {
    return findCollectionHelper(heights, 0, heights.size());
}
```

Week Two: Container Classes

1. Write a function that, given a `Map<string, int>` associating string values with integers, produces a `Map<int, Set<string>>` that's essentially the reverse mapping, associating each integer value with the set of strings that map to it. (This is an old job interview question from 2010.)

Here's one possible implementation. Note the use of the map autoinsertion feature.

```
Map<int, Set<string>> reverseMap(const Map<string, int>& map) {
    Map<int, Set<string>> result;

    for (string oldKey : map) {
        result[map[oldKey]] += oldKey;
    }

    return revMap;
}
```

2. How are `Map` and `HashMap` implemented internally? What's one advantage of `Map` over `HashMap`? One advantage of `HashMap` over `Map`?

The `Map` is internally layered on top of a balanced BST. The `HashMap` is layered on top of a hash table. Because `Map` is backed by a BST, it stores its elements in sorted order, so there's predictable behavior when we iterate over it (we always get things back in sorted order). This contrasts with `HashMap`, where iteration order will visit things in the order in which the elements appear in the underlying hash table, something that's much harder to predict.

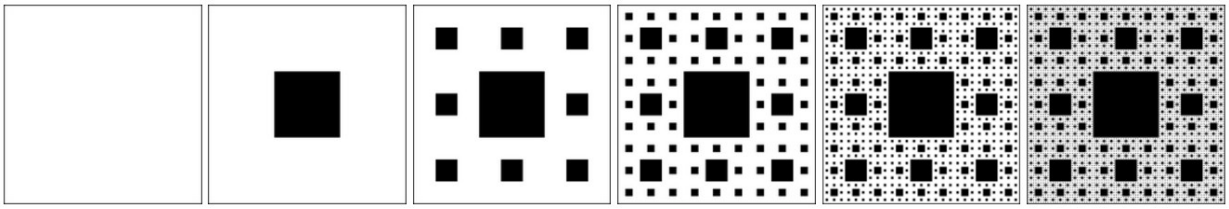
The `HashMap` uses a hash table internally, which is generally faster than a binary search tree (expected runtime O(1) for all major operations, compared with O(log *n*) runtime).

3. A ***compound word*** is a word that can be cut into two smaller strings, each of which is itself a word. The words "keyhole" and "headhunter" are examples of compound words, and less obviously so is the word "question" ("quest" and "ion"). Write a function that takes in a Lexicon of all the words in English and then prints out all the compound words in the English language.

```
bool isCompoundWord(const string& word, const Lexicon& dict) {
    /* Try splitting the word into two possible words for every possible position
     * where you can make the split.
     */
    for (int i = 1; i < word.length(); i++) {
        /* The two words are formed by taking the substring up to the splitting
         * point and the substring starting at the splitting point.
         */
        if (dict.contains(word.substr(0, i)) && dict.contains(word.substr(i))) {
            return true;
        }
    }
    return false;
}

void printCompoundWords(const Lexicon &dict) {
    for (string word: dict) {
        if (isCompoundWord(word, dict)) cout << word << endl;
    }
}
```

Week Three: Graphical Recursion and Recursive Problem-Solving

1. The ***Sierpinski carpet*** is a fractal image in the shape of a square. An order-0 Sierpinski carpet is just an empty square. An order-(n+1) Sierpinski carpet can be formed by subdividing a square into nine smaller squares in a 3 × 3 grid, filling the central square black, then recursively drawing order-n Sierpinski carpets in each of the remaining grid cells. Here are Sierpinski carpets of orders 0, 1, 2, 3, 4, and 5:



Write a function to draw an order-n Sierpinski carpet in a given window.

Here's one possible solution that translates the recursive definition of the Sierpinski carpet into code:

```
/* Draws a Sierpinski carpet of the given order in the specified window, assuming
 * the upper-left corner is at (x, y) and the total width and height are given by
 * size.
 */
void drawCarpetRec(GWindow& window, int order, double x, double y, double size) {
    /* Base case: There's nothing to do if we have an order-0 carpet. */
    if (order == 0) return;

    /* Recursive case: Split into a 3x3 grid. */
    double subSize = size / 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
             /* Recursively draw smaller grids in all squares but the center. */
             if (i != 1 || j != 1) {
              drawCarpetRec(window, order - 1,
                            x + i * subSize,
                            y + j * subSize,
                            subSize);
                }
            }
        }
        /* Draw square in middle sub grid */
        window.fillRect(x + subSize, y + subSize, subSize, subSize);
    }
}

void drawCarpet(GWindow& window, int order, double size) {
    drawCarpetRec(window, order, 0, 0, size);
}
```

2. Imagine you have a $2 \times n$ grid that you'd like to cover using 2×1 dominoes. The dominoes need to be completely contained within the grid (so they can't hang over the sides), can't overlap, and have to be at 90° angles (so you can't have diagonal or tilted tiles). There's exactly one way to tile a 2×1 grid this way, exactly two ways to tile a 2×2 grid this way, and exactly three ways to tile a 2×3 grid this way (can you see what they are?) Write a recursive function that, given a number n, returns the number of ways you can tile a $2 \times n$ grid with 2×1 dominoes.

If you draw out a couple of sample tilings, you might notice that every tiling either starts with a single vertical domino or with two horizontal dominoes. That means that the number of ways to tile a $2 \times n$ (for $n \geq 2$) is given by the number of ways to tile a $2 \times (n - 1)$ grid (because any of them can be extended into a $2 \times n$ grid by adding a vertical domino) plus the number of ways to tile a $2 \times (n - 2)$ grid (because any of them can be extended into a $2 \times n$ grid by adding two horizontal dominoes). From there the question is how to compute this. You could do this with regular recursion, like this:

```
int numWaysToTile(int n) {
    /* There's one way to tile a 2 x 0 grid: put down no dominoes. */
    if (n == 0) return 1;

    /* There's one way to tile a 2 x 1 grid: put down a vertical domino. */
    if (n == 1) return 1;

    /* Recursive case: Use the above insight. */
    return numWaysToTile(n - 1) + numWaysToTile(n - 2);
}
```

With a little bit of thought we can note that this function is begging for memoization: we're going to be making a lot of subcalls on the exact same subproblems. Here's one way to address this, given that we know we're going to make recursive calls on values 0, 1, 2, 3, …, n.

```
int numWaysToTile(int n) {
    Vector<int> memo;
    for (int i = 0; i <= n; i++) {    // 0 through n, inclusive
        memo += -1;
    }
    return numWaysRec(n, memo);
}

int numWaysRec(int n, Vector<int>& memo) {
    /* There's one way to tile a 2 x 0 grid: put down no dominoes. */
    if (n == 0) return 1;

    /* There's one way to tile a 2 x 1 grid: put down a vertical domino. */
    if (n == 1) return 1;

    /* If we haven't yet computed this, compute it and stash it for later. */
    if (memo[n] == -1) {
        memo[n] = numWaysRec(n - 1, memo) + numWaysRec(n - 2, memo);
    }
    return memo[n];
}
```

Week Four: Recursive Enumeration

1. Given a positive integer *n*, write a function that finds all ways of writing *n* as a sum of nonzero natural numbers. For example, given *n* = 3, you'd list off these options:

   3 2 + 1 1 + 2 1 + 1 + 1

The key insight here is that some positive number has to come first in our ordering, so we can just try all possible ways of breaking off some initial bit and see what we find.

```
void printSumsOf(int n) {
    /* Handle edge cases. */
    if (n < 0) error("Can't make less than nothing from more than nothing.");

    printSumsRec(n, {});
}

/* Print all ways to sum up to n, given that we've already broken off the numbers
 * given in soFar.
 */
void printSumsRec(int n, const Vector<int>& soFar) {
    /* Base case: Once n is zero, we need no more numbers. */
    if (n == 0) {
        printAsSum(soFar);
    } else {
        /* The next number can be anything between 1 and n, inclusive. */
        for (int i = 1; i <= n; i++) {
            Vector<int> soFarCopy = soFar;
            soFarCopy += i;
            printSumsRec(n - i, soFarCopy);
        }
    }
}

/* Prints a Vector<int> nicely as a sum. */
void printAsSum(const Vector<int>& sum) {
    /* The empty sum prints as zero. */
    if (sum == 0) {
        cout << 0 << endl;
    } else {
        /* Print out each term, with plus signs interspersed. */
        for (int i = 0; i < sum.size(); i++) {
            cout << sum[i];
            if (i + 1 != sum.size()) cout << " + ";
        }
        cout << endl;
    }
}
```

2. Solve the previous problem assuming that order doesn't matter, so 1 + 2 and 2 + 1 would be treated identically. See if you can find a way to do this that doesn't generate the same option more than once.

There are many ways that we can do this, but one nice option would be to generate the terms in the sum in nonincreasing order. That is, given the option of writing either 1 + 2 or 2 + 1, we'd choose to print 2 + 1. To do this, we'll use a strategy similar to the one from before, except that we'll cap the maximum value we can use in future recursive calls to ensure things are in nonincreasing order.

```
void printSumsOf(int n) {
    /* Handle edge cases. */
    if (n < 0) error("Can't make less than nothing from more than nothing.");

    /* Initially, the maximum value we can use is n. */
    printSumsRec(n, n, {});
}

/* Print all ways to sum up to n, given that we've already broken off the numbers
 * given in soFar, without using any numbers greater than maxVal.
 */
void printSumsRec(int n, int maxVal, const Vector<int>& soFar) {
    /* Base case: Once n is zero, we need no more numbers. */
    if (n == 0) {
        printAsSum(soFar);
    } else {
        /* The next number can be anything between 1 and maxVal, inclusive. */
        for (int i = 1; i <= maxVal; i++) {
            Vector<int> soFarCopy = soFar;
            soFarCopy += i;

            /* Cap all future terms at the one we just added in. */
            printSumsRec(n - i, i, soFarCopy);
        }
    }
}
```

3. Write a function that, given a list of distinct strings and a number *k*, lists off all ways of choosing *k* elements from that list, given that order *does* matter. For example, given the objects A, B, and C and *k* = 2, you'd list

A, B A, C B, A B, C C, A C, B

This one is half combinations, half permutations. We use the permutations strategy of asking "what is the next term in our ordered list?" at each step, and the combinations strategy of cutting off our search as soon as we have enough terms.

```
void listKOrderings(const Vector<string>& choices, int k) {
    /* Quick edge case check: if we want more items than there are options, there
     * are no orderings we can use.
     */
    if (k > choices.size()) {
        listOrderingHelper(choices, k, {});
    }
}

void listOrderingHelper(const Vector<string>& choices, int k
                        const Vector<string>& soFar) {
    /* Base case: If no more terms are needed, print what we have. */
    if (k == 0) {
        cout << soFar << endl;
    }
    /* Recursive case: What comes next? Try all options. */
    else {
        for(int i = 0; i < choices.size(); i++) {
            Vector<string> next = soFar;
            Vector<string> remaining = choices;

            /* Remove this choice from set of future choices */
            remaining.remove(i);

            /* Add this to our choices made soFar */
            next += choices[i];

            /* Recurse on k -1 since we have used up one choice*/
            listOrderingHelper(remaining, k - 1, next);
        }
    }
}
```

Week Five: Recursive Backtracking, Big-O and Sorting

1. One of the problems from the "Container Classes" section of this handout discussed compound words, which are words that can be cut into two smaller pieces, each of which is a word. You can generalize this idea further if you allow the word to be chopped into even more pieces. For example, the word "longshoreman" can be split into "long," "shore," and "man," and "whatsoever" can be split into "what," "so," and "ever." Write a function that takes in a word and returns whether it can be split apart into two *or more* smaller pieces, each of which is itself an English word.

The main insight here is that a word can be broken apart into two or more words if it can be split into two pieces such that the first piece is a word, and the second piece is either (1) a word or (2) itself something that can be split apart into two or more words.

```
void printMultCompoundWords(const Lexicon& dict) {
    for (string word : dict) {
        if (isMultCompoundWord(word, dict)) cout << word << endl;
    }
}

bool isMultCompoundWord(const string& word, const Lexicon& dict) {
    /* In an unusual twist, our base case is folded into the recursive step. We
     * will try all possible splits into two pieces, and if one of them happens
     * to be a pair of words, we stop.
     */

    /* Try all ways of splitting things. */
    for (int i = 1; i < word.length(); i++) {
        /* Only split if the first part is a word. */
        if (dict.contains(word.substr(0, i))) {
            string remaining = word.substr(i);

            /* We're done if the remainder is either a word or a compound word. */
            if (dict.contains(remaining) || isMultCompoundWord(remaining, dict)) {
                return true;
            }
        }
    }

    /* Nothing works; give up. */
    return false;
}
```

2. You are standing on the upper-left corner of a grid of nonnegative integers. You're interested in moving to the lower-right corner of the grid. The catch is that at each point, you can only move up, down, left, or right a number of steps exactly equal to the number you're standing on. For example, if you were standing on the number three, you could move exactly three steps up, exactly three steps down, exactly three steps left, or exactly three steps right. (You can't move off the board). Write a function that determines whether it's possible to get from the upper-left corner (where you're starting) to the lower-right corner while obeying these rules.

The core idea here is that if we are already in the lower-right corner, great! We're done. Otherwise, we need to take some step to get closer to that point, so we can try all of them.

With a bit of thought, you might realize that what we're doing here is a graph search problem! So we'll approach this as a depth-first search and keep track of which positions we've visited before to speed things up.

```
bool canMoveToBottom(const Grid<int>& grid) {
    /* Where we've been before. */
    Grid<bool> visited(grid.numRows(), grid.numCols(), false);

    /* (0, 0) is upper left corner */
    return canMoveToBottomHelper(0, 0, grid, visited);
}

bool canMoveToBottomHelper(int row, int col,
                           const Grid<int>& grid,
                           Grid<bool>& visited) {
    /* If we are out of bounds or been here before, stop. We either are in an
     * impossible state, or we're somewhere we're repeating ourselves.
     */
    if (!grid.inBounds(row, col) || visited[row][col]) return false;

    /* If we are at the final spot, we've made it! */
    if (row == grid.numRows() - 1 && col == grid.numCols() - 1) return true;

    /* Mark this spot as visited so we don't end up in an infinite loop. */
    visited[row][col] = true;

    /* See how much we can move. */
    int delta = grid[row][col];

    /* Try moving in each direction by the allowed amount */
    return    canMoveToBottomHelper(r + delta, c,         grid, visited)
           || canMoveToBottomHelper(r - delta, c,         grid, visited)
           || canMoveToBottomHelper(r,         c + delta, grid, visited)
           || canMoveToBottomHelper(r,         c - delta, grid, visited);
}
```

3. The pancake sorting problem from the midterm asked you to see whether it was possible to sort a stack of pancakes within *k* flips, for some number *k*. This problem is a lot easier to solve if you don't have the upper limit. Write a function that takes in a stack of pancakes and sorts it, provided that the only legal move you can make is to put a spatula under one of the pancakes, then flip it and all the pancakes above it upside down.

Notice that this question doesn't say "find the best series of flips." We just need something that works, period, and that's actually not too bad. We can find the biggest pancake, flip it to the top of the stack, then flip it to the bottom. Then, we find the second-largest, flip that to the top, then flip that to the bottom, etc. We don't even need to do this recursively! Here's one solution along those lines.

```
void sortPancakes(Stack<double>& pancakes) {
    for (int height = pancakes.size() - 1; height > 0, height--) {
        /* Find how deep the largest pancake is. */
        int depth = largestPancakeIn(pancakes, height);

        /* Transfer that many pancakes to a Queue, pulling them off the stack. */
        Queue<double> spatula;
        for (int i = 0; i <= depth; i++) {
            spatula.enqueue(pancakes.pop());
        }

        /* Transfer them back, which puts them back upside-down. */
        while (!spatula.isEmpty()) {
            pancakes.push(spatula.dequeue());
        }
    }
}

/* Finds the index of the biggest pancake in the top of the stack. The stack is
 * returned unmodified.
 */
int largestPancakeIn(Stack<double>& pancakes, int height) {
    double largest = -1;
    int largestIndex; // Will be overwritten

    /* Transfer everything into a temporary Stack. */
    Stack<double> temp;
    for (int i = 0; i < height; i++) {
        double pancake = pancakes.pop();
        if (pancake > largest) {
            largest = pancake;
            largestIndex = i;
        }
        temp.push(pancakes.pop());
    }

    /* Transfer them back. */
    while (!temp.isEmpty()) {
        pancakes.push(temp.pop());
    }

    return largestIndex;
}
```

Week Six: Dynamic Arrays

1. The `int` type in C++ can only support integers in a limited range (typically, -2^{31} to $2^{31} - 1$). If you want to work with integers that are larger than that, you'll need to use a type often called a ***big number*** type (or "bignum" for short). Those types usually work internally by storing a dynamic array that holds the digits of that number. For example, the number 78979871 might be stored as the array 7, 8, 9, 7, 9, 8, 7, 1 (or, sometimes, in reverse as 1, 7, 8, 9, 7, 9, 8, 7). Implement a bignum type layered on top of a dynamic array. Your implementation should provide member functions that let you add or multiply together two bignums. *(Hint: start with addition, then use that to implement multiplication)*.

Sorry, I wasn't able to get this ready in time to release this handout. We'll try to correct this for future quarters. ☹

2. Implement a version of the `Grid` type that supports creating a grid of a certain size, reading from grid locations, and writing to grid locations. Do all your own memory management.

Sorry, I wasn't able to get this ready in time to release this handout. We'll try to correct this for future quarters. ☹

Week Seven: Linked Lists

1. Write a function that, given a pointer to a singly-linked list and a number k, returns the kth-to-last element of the linked list (or a null pointer if no such element exists). How efficient is your solution, from a big-O perspective? As a challenge, see if you can solve this in $O(n)$ time with only O(1) auxiliary storage space.

There are a couple of ways we could do this. One option would be to sweep across the list from the front to the back, counting how many nodes there are, then calculate the index we need. That's shown here:

```
struct Node {
    string value;
    Node* next;
};

int listLength(Node* list) {
    int count = 0;
    for (Node* curr = list; curr != nullptr; curr = curr->next) {
        count++;
    }
    return count;
}

Node* kthToLastSimple(Node* list, int k) {
    /* Find length of the list */
    int len = listLength(list);

    /* If the list is too small, just return nullptr*/
    if (len < k) return nullptr;

    /* Move len - k + 1 steps into the list */
    Node* curr = list;
    for (int i = 0; i < len - k; i++) {
        curr = curr->next;
    }
    return curr;
}
```

Another option, which is a bit less obvious but is quite beautiful, is to walk down the list with two concurrent pointers, one of which is *k* steps ahead of the other. As soon as the lead pointer falls off the list, the pointer behind it is *k* steps from the end. Do you see why?

```
Node* kthToLast(Node* list, int k) {
    /* Set up two pointers, one leader and one follower. */
    Node* leader   = list;
    Node* follower = list;

    /* March the leader k steps forward. If we fall of the list in this time,
     * there is no kth-to-last node.
     */
    if (leader == nullptr) return nullptr;
    for (int i = 0; i < k; i++) {
        leader = leader->next;
        if (leader == nullptr) return nullptr; // Why do we have two checks?
    }

    /* Keep walking the leader and follower forward. As soon as the leader walks
     * off the follower is in the right spot.
     */
    while (true) {
        leader = leader->next;
        if (leader == nullptr) return follower;

        follower = follower->next;
    }
}
```

2. Write an implementation of insertion sort that works on singly-linked lists.

The singly-linked list requirement here suggests that our pattern of repeatedly swapping elements back in the sequence is not going to be easy to implement – we'll keep losing track of where our preceding element is. There are many ways we could deal with this. One would be to build the sorted list in reverse, then fix up the pointers at the end to reverse it. Another, shown below, works by taking the element, moving it to the front of the list, then swapping it forward until it's in the right position.

```
void listInsertionSort(Node* &list) { // Question: why by reference?
    Node* sortedList = nullptr;

    Node* curr = list;
    while(curr != nullptr) {
        /* Need to store the pointer to the next cell in the list, since after
         * we do the insert operation we'll lose track of where the next cell
         * is.
         */
        Node* next = curr->next;
        sortedInsert(curr, sortedList);
        curr = next;
    }
    list = sortedList;
}

/* Adds the given node into a sorted, singly-linked list. */
void sortedInsert(Node* toIns, Node*& list) {
    /* See if we go at the beginning. */
    if (list == nullptr || toIns->value < list->value) {
        toIns->next = list;
        list = toIns;
    } else {
        /* Find the spot right before where we go, since that's the pointer we
         * need to rewire.
         */
        Node* curr = list;
        Node* prev = nullptr;

        while (curr != nullptr && curr->value < toIns->value) {
            prev = curr;
            curr = curr->next;
        }

        /* Splice us in. */
        toIns->next = curr;
        prev->next = toIns;
    }
}
```

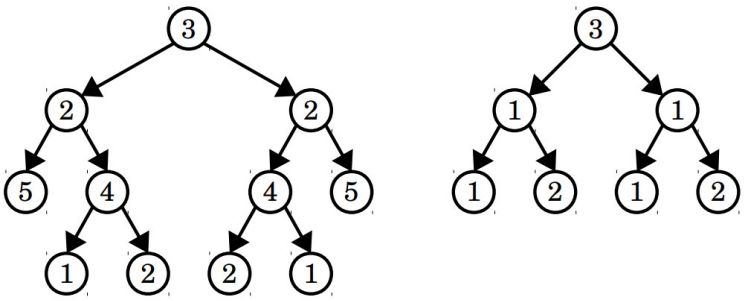
3. Imagine that you have two linked lists that meet at some common point in a Y shape (the head pointer of each linked list would be on the top of the Y, and they merge at a common node). Write a function that finds their intersection point. The "branches" of the Y don't have to have the same lengths, and the elements stored within the linked lists might coincidentally match even before their intersection point. As a challenge, see if you can do this in O(1) auxiliary space.

Suppose the first list appears to have m elements and the second appears to have n, where $m \geq n$. The key observation is that the intersection point can only be in the last n nodes of the two lists – any earlier and both lists would have to be longer than length n. (Do you see why?) This suggests a nice algorithm: make a pass over each list to compute their lengths, skip down so that we're n steps from the end of both lists, then walk forward until they meet.

```
Node* findIntersection(Node* list1, Node* list2) {
    /* Find the difference in length between the two lists */
    int len1 = listLength(list1);
    int len2 = listLength(list2);

    /* Swap the lists so that list1 is longer. */
    if (len1 < len2) {
        swap(list1, list2);
        swap(len1,  len2);
    }

    /* Advance the first pointer forward until the paths to the end of both lists
     * appear equal.
     */
    for (int i = 0; i < len1 - len2; i++) {
        list1 = list1->next;
    }

    /* March both lists forward until they intersect. */
    while (true) {
        if (list1 == list2) return list1;

        list1 = list1->next;
        list2 = list2->next;
    }
}
```

Week Eight: Trees and Hashing

1. A binary tree (not necessarily a binary *search tree*) is called a ***palindromic tree*** if it's its own mirror image. For example, the tree on the left is a palindromic tree, but the tree on the right is not:



   Write a function that takes in a pointer to the root of a binary tree and returns whether it's a palindrome tree.

To solve this problem, we'll solve a slightly more general problem: given two trees, are they mirrors of one another? We can then check if a tree is a palindrome by seeing whether that tree is a mirror of itself.

```
bool isPalindromicTree(Node* root) {
    return areMirrors(root, root);
}

bool areMirrors(Node* root1, TreeNode* root2) {
    /* Two empty trees are equal, even when reversed. */
    if(root1 == nullptr && root2 == nullptr) return true;

    /* If one of the trees is empty and the other is not, or the values stored in
     * the roots are different, they aren't mirrors.
     */
    if(root1 == nullptr || root2 == nullptr || root1->value != root2->value) {
        return false;
    }

    /* To see if they're mirrors, we need to check whether the left subtree of
     * the first tree mirrors the right subtree of the second tree and vice-versa.
     */
    return areMirrors(root1->left,  root2->right) &&
           areMirrors(root1->right, root2->left);
}
```

2. (*The Great Tree List Recursion Problem, by Nick Parlante*) A node a binary tree has the same fields as a node in a doubly-linked list: one field for some data and two pointers. The difference is what those pointers mean: in a binary tree, those fields point to a left and right subtree, and in a doubly-linked list they point to the next and previous elements of the list. Write a function that, given a pointer to the root of a binary *search* tree, flattens the tree into a doubly-linked list, with the values in sorted order, without allocating any new cells. You'll end up with a list where the pointer `left` functions like the `prev` pointer in a doubly-linked list and where the pointer `right` functions like the `next` pointer in a doubly-linked list.

This is a beautiful recursion problem. Essentially, what we want to do is the following:

- The empty tree is already indistinguishable from the empty list.
- Otherwise, flatten the left subtree and right subtree, then concatenate everything together.

To implement that last step efficiently, we'll have our recursive function hand back two pointers: one to the front of the flattened list and one to the back.

```
struct Range {
    Node* first;
    Node* last;
};

Node* treeToList(Node* root) {
    return flatten(root).first;
}

Range flatten(Node* root) {
    /* If the tree is empty, it's already flattened. */
    if (root == nullptr) return { nullptr, nullptr };

    /* Flatten the left and right subtrees. */
    Range left  = flatten(root->left);
    Range right = flatten(root->right);

    /* Glue things together. */
    root->left = left.last;
    if (left.last != nullptr) left.last->right = root;

    root->right = right.first;
    if (right.first != nullptr) right.first->left = root;

    /* Return the full range. */
    return {
        left.first == nullptr? root : left.first ,
        right.last == nullptr? root : right.last
    };
}
```

3. Suppose you insert the numbers 1, 2, 3, 4, 5, …, n into one hash table, then insert the numbers n, n-1, n-2, …, 3, 2, 1 into another hash table. Assuming the hash tables are implemented the same way, is it *guaranteed* that the internal structure of the two hash tables will be the same? Is it *possible* that their internal structure will be the same? Is it *never* going to be the case that the internal structure will be the same?

No, this is not guaranteed to happen. As an example, suppose that the hash table has n buckets and never resizes things. Then if two numbers hash into the same bucket, the ordering of those numbers in the bucket will be reversed in the first case and the second.

It is *possible* the two internal structures will be the same. For example, if there are no hash collisions between the elements then the same elements will end up in the same buckets, and there won't be the possibility that elements will end up out of order.

Week Nine: Graphs and Graph Algorithms

1. Explain how question (2) from the section on Recursive Backtracking in this handout is essentially a graph search problem, then explain how to solve it using breadth-first search.

We can envision what we're doing there as a graph search problem as follows: each cell in the grid is a node, and there's a (directed) edge from one cell to another if, starting at the first cell, you can move from the first cell to the second (that is, they're the appropriate distance apart). The question is then whether there's a path from the upper-left corner to the lower-right corner.

Here's an solution using BFS, which pairs with the DFS solution from earlier:

```
bool canMoveToBottom(const Grid<int>& grid) {
    /* Where we've been before. */
    Grid<bool> visited(grid.numRows(), grid.numCols(), false);

    /* Our worklist! */
    Queue<GridLocation> worklist;
    toVisit.enqueue({0, 0});
    visited[0][0] = true;

    /* Like love, hope that we find a way. */
    while (!worklist.isEmpty()) {
        auto curr = worklist.dequeue();

        /* Are we there yet? */
        if (curr.row == grid.numRows() - 1 && curr.col == grid.numCols()) {
            return true;
        }

        /* Try all hops can make. */
        exploreFrom(curr, grid[curr.row][curr.col], worklist, visited, +1,  0);
        exploreFrom(curr, grid[curr.row][curr.col], visited, -1,  0);
        exploreFrom(curr, grid[curr.row][curr.col], visited,  0, +1);
        exploreFrom(curr, grid[curr.row][curr.col], visited,  0, -1);
    }

    /* Oh fiddlesticks. We can't get there. */
    return false;
}

void exploreFrom(const GridLocation& loc, int stepSize,
                 Queue<GridLocation>& worklist,
                 Grid<bool>& visited,
                 int dx, int dy) {
    /* See if the target is both in bounds and not yet visited. */
    GridLocation target = { loc.row + stepSize * dx, low.col + stepSize * dy };

    if (visited.inBounds(target) && !visited[target.row][target.col]) {
        worklist.enqueue(target);
        visited[target.row][target.col] = true;
    }
}
```

2. Imagine you have a graph representing a social network. Your friends are the people one hop away from you. Someone would be considered a "friend of a friend" if they were two hops away from you (and also not zero or one hops away from you), and someone would be considered a "friend of a friend of a friend" if they were three hops away from you (and also not zero, one, or two hops away from you). Write a function that, given the graph and a number *k*, returns everyone who is a *k*th-order friend of yours.

Any time you hear something about measuring distances in a graph, you should think "breadth-first search" or another algorithm that works similarly to it. It's a great tool to use!

This modified version of BFS keeps track of the distances to each node in the queue. We stop searching as soon as we get more than *k* hops away.

```
struct Entry {
    string name;
    int distance;
};

Set<string> kthOrderFriends(const Map<string, Set<string>>& graph,
                            const string& person, int k) {
    if (!graph.containsKey(person)) error("Person not found?");

    /* Worklist and visited set. */
    Queue<Entry> worklist;
    Set<string> visited;

    worklist.enqueue({ person, 0 });
    visited += person;

    /* Result set, filled in with everyone at distance k. */
    Set<string> result;

    /* Do the BFS! */
    while (!worklist.isEmpty()) {
        auto curr = worklist.dequeue();

        /* If they're at the proper distance, record them. */
        if (curr.distance == k) result += curr.name;

        /* Add their unvisited friends who aren't too far away. */
        if (curr.distance < k) {
            for (string friend: graph[curr.name]) {
                if (!visited.contains(friend)) {
                    visited += friend;
                    worklist.enqueue({ friend, curr.distance + 1 });
                }
            }
        }
    }
    return result;
}
```

Thanks for reading this far!