Chris Gregg
March 20th, 2017

CS106X Final Exam

This is an open-note, open-book exam. You can refer to any course handouts, textbooks, handwritten lecture notes, and printouts of any code relevant to any CS106X assignment. You may not use any laptops, cell phones, or internet or electronic devices of any sort. You will be graded on functionality, but good style helps graders understand what you were attempting. You do not need to `#include` any libraries and you do not need to forward declare any functions. You have three (3) hours to complete the exam. We hope this exam is an exciting journey.

All answers to this exam must go in the space provided. Think ahead before writing anything down. You may use scratch paper, but we will only count answers put in the space provided. Please do not write on the backs of the exam pages except for scrap work.

Honor Code Pledge:

I accept the letter and the spirit of the honor code. I have neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

Your name: _____

Your Stanford SUNet ID (e.g., cgregg): _____

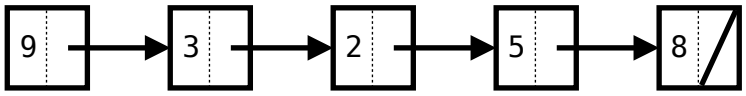
(signed)_____

Question	Points
1. Big Number Addition (Linked Lists)	10
2. Hashing	10
3. Macro Keystroke Sequences	20
4. Binary Search Trees (AVL)	8
5. Graphs	15
6. Inheritance and Polymorphism	8
7. Bonus	3
Total	71

Question 1: Big Number Addition (Linked Lists) (10 Points)

The largest unsigned int can only hold 32-bits of information (2^{32}, or roughly 4 billion numbers), and the largest unsigned long can only hold 64-bits of information (2^{64}, or $\sim 10^{19}$). However, there are times when you might want larger integers, and there are libraries that define a "`BigNum`" as an infinite-precision integer. In a `BigNum`, each digit is represented in a list, and arithmetic functions are written to manipulate the list.

For this problem, you will be working with a `BigNum` class that holds the decimal digits for an integer in a linked list. The list is stored in what is called "little endian" format, meaning that the least significant digit is first, and the most significant digit is last. For example, the following linked list holds the number "85239":



For this problem, write an `add()` method for the `BigNum` class:

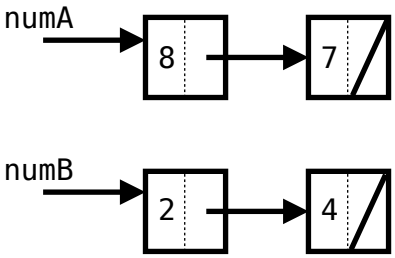
```
// Adds two BigNums together and modifies numA
//      to hold the resulting sum.
// numA and numB each point to the head of a BigNum list
// pre-condition: numA must be equal to or bigger than numB
void add(BigNum *numA, BigNum *numB);
```

The BigNum struct is defined as follows:

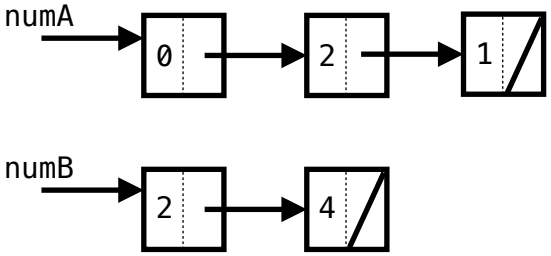
```
struct BigNum {
    int data; // a decimal digit
    BigNum *next; // pointer to the next digit, little endian
    BigNum(int digit) { // constructor
        data = digit;
        next = NULL;
    }
};
```

Example (78 + 42 = 120):

Before function call



After `add(numA, numB);`



Please answer Question 1 on this page.

```
// Adds two BigNums together and modifies numA
//      to hold the resulting sum.
// numA and numB each point to the head of a BigNum list
// pre-condition: numA must be equal to or bigger than numB

void add(BigNum *numA, BigNum *numB) {
   // note: many ways to write this function
   // add the first two digits, and carry if necessary
    int carry = 0;
    int digitSum;
    while (numB != nullptr) {
        digitSum = numA->data + numB->data + carry;
        if (digitSum > 9) {
            carry = digitSum / 10;
            digitSum = digitSum % 10;
        } else {
            carry = 0;
        }
        numA->data = digitSum;
        numB = numB->next;
        if (numB != nullptr) {
            numA = numA->next;
        }
    }
    // we may still have a carry and we may
    // need to cycle through all remaining digits in numA
    while (carry > 0) {
        if (numA->next == nullptr) {
            numA->next = new BigNum(carry);
            carry = 0;
        } else {
            numA = numA->next;
            numA->data = numA->data + carry;
            if (numA->data > 9) {
                carry = numA->data / 10;
                numA->data = numA->data % 10;
            } else {
                carry = 0;
            }
        }
    }
}
```

Question 2: Hashing (10 Points)

Simulate the behavior of a hash map of ints (keys) to strings (values) as described and implemented in lecture. Assume the following:

- the hash table array has an initial capacity of 10
- the hash table uses linked list chaining to resolve collisions, and key/value pairs are inserted at the head of each linked list.
- the hash function is defined as follows.

```
int hash(int key) {
    return key % tableCapacity;
}
```

- the `remove()` function does nothing if the key to remove is not present in the table.
- adding a key/value pair for which the key already exists in the table simply updates the value for that key.
- rehashing occurs at the end of an add where the load factor is ≥ 0.75 and doubles the capacity of the hash table.

Draw an array diagram to show the final state of the hash table after the following operations are performed. Leave a box empty if an array element is unused. All values must be in the proper buckets and in the proper order to receive full credit. Also write the final size, capacity, and load factor of the 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 circle your final answer.

```
map.put(6, "looks");
map.put(5, "job");
map.put(88, "you");
map.put(8, "bee");
map.put(1, "one");
map.put(15, "hug");
map.put(29, "got");
map.put(15, "try" );
map.put(7, "a");
map.put(19, "correct");
map.put(2, "not");
map.put(26,"it");
map.put(14, "completely");
map.put(44, "good");
```

```
map.put(13, "question");
map.remove(7);
map.remove(123);
map.remove(15);
map.put(32,"whole");
map.put(12,"this");
if (map.containsKey(15)) {
    map.remove(29);
}
for (int x = 1; x < 13; x*=2) {
    map.remove(x);
}
map.put(47, "like");
```

Please draw **and circle** your completed table on the next page. You may leave out quotation marks, and your finished table might look something like this:

0

1 7, example 101, answer

2

3 8, the 26, hashing 9, here

4 83, is

5 49, wrong

Please draw and **circle your completed hash table** for question 2 on this page. You can get partial credit for temporary tables.

0:
1:
2:
3:
4: (44,good)
5: (5,job)
6: (26,it), (6,looks)
7: (47,like)
8: (88,you)
9: (29,got)
10:
11:
12: (12,this), (32,whole)
13: (13,question)
14: (14,completely)
15:
16:
17:
18:
19: (19,correct)

Size: 12
Capacity: 20
Load Factor: 0.6

Problem 3, Macro Keystroke Sequences (20 points)

Sometimes when writing computer applications, we want to register a certain sequence of user keystrokes to perform some action. Often, text editors have "macro" functionality, where a small number of keystrokes will print out a larger string. Here, we will simplify that to say that we want to be able to register a sequence of lower-case letters to print out some message to the console in real-time every time those letters are typed in order. For example, say we have the sequence "abc" registered to print out "Hello world" and the sequence "xyz" to print out "Goodbye world". Then if the user types the letters "abcxyz", "Hello world" and "Goodbye world" will be printed in that order, when the 'c' and 'z' are typed respectively.

To do this, we will make use of an expanded version of a binary tree, called a *trie*, where each node has 26 children. You have the following definition:

```
struct TrieNode {
    string message;
    TrieNode *children[26]; // children[0] is 'a', children[1] is 'b'...
}
```

Using this, a sequence of characters can be registered with a given message by modifying your trie such that following the children associated with each letter in the sequence in order ends with a `TrieNode` storing the given message. You can assume that two messages will never be registered to the same sequence of characters. Note that this is also how the Stanford `Lexicon` class is built. Your task then is to write a recursive function which registers a given message to a given sequence of characters.

Part 3a. (10 points) Write the following function, `register()`, which stores a sequence of characters and the message associated to those characters in a trie.

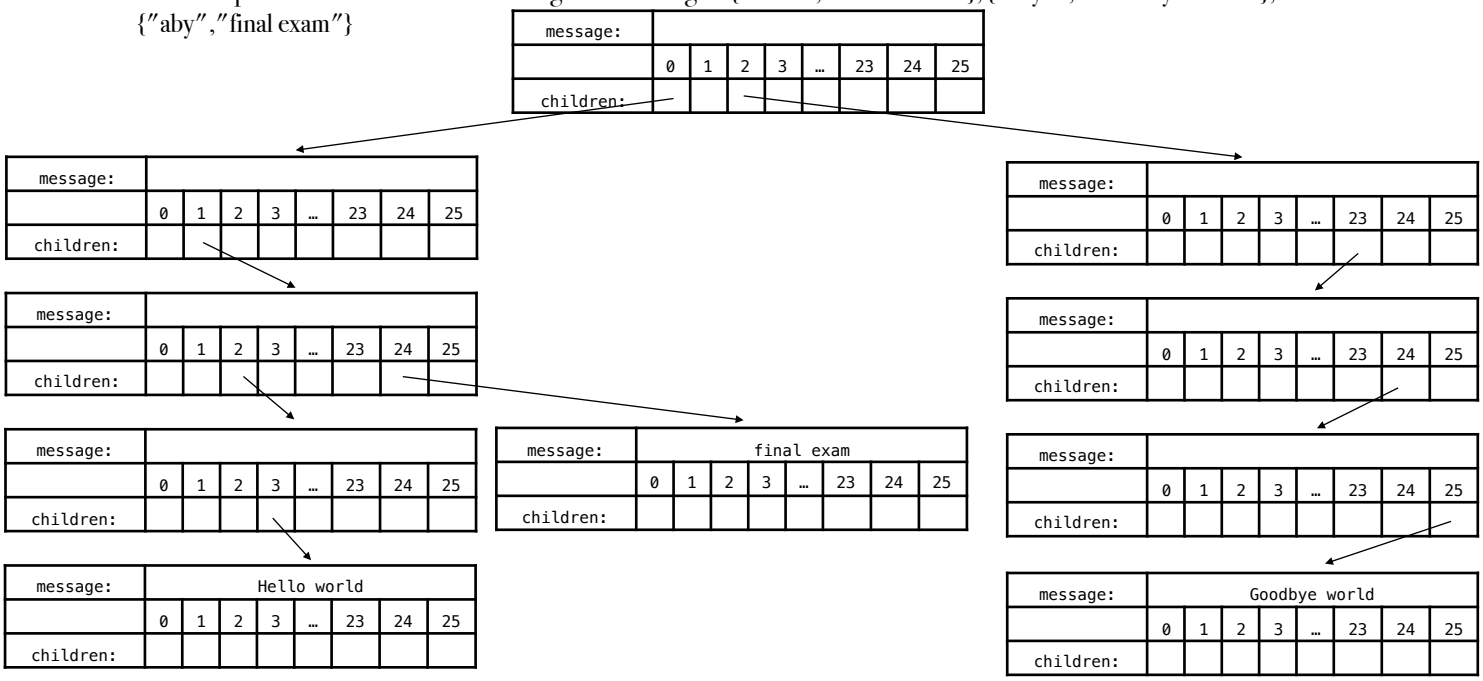
```
void register(string message, string sequence, TrieNode &*root);
```

Hints:

1. You can convert a `char` character to its relative index value by subtracting `'a'` from the value. E.g., to convert the letter `'c'` to the index value 2 ('a' = 0, 'b' = 1, 'c' = 2, etc.), you can write the following:
   ```
   int index = 'c' - 'a'; // index is now 2
   ```

2. The message should only be stored in the trie once, in the `TrieNode` where the last character of the sequence is stored.

3. The example trie below holds the following three messages: {"abcd","Hello world"}, {"cxyz", "Goodbye world"}, and {"aby","final exam"}



Answer problem 3a below. You may create helper functions as necessary.

```
void register(string message, string sequence, TrieNode *&root) {
    registerHelper(message, message, sequence, root);
}

void register(string message, string sequence, TrieNode *&root) {
    if (root == nullptr) {
        root = new TrieNode();
    }
    if (sequence.isEmpty()) {
        root->message = message;
        return;
    }
    register(message, sequence.substr(1), root->children[sequence[0] - 'a']);
}
```

Problem 3b. (10 points) Now that we have our trie, we need to make use of it. To do this, we will write a `keyPressed` function for macros that gets called whenever the user types a letter (which you can assume is lowercase) and immediately prints out any non-empty messages appropriately. There are two important implications of this. First, this means that you will, in essence, be traversing your trie step by step, instead of all at once. Second, you will be maintaining several traversals of the trie at once. For example, if "abc" has been typed so far, then "abc", "bc", and "c" are all valid starts of character sequences (traversals) and so must all be kept track of individually.

Consider the following example sequences (see below for the associated trie):

"ac" registers to "Hello"
"abab" registers to "Goodbye"
"bb" registers to "final exam"

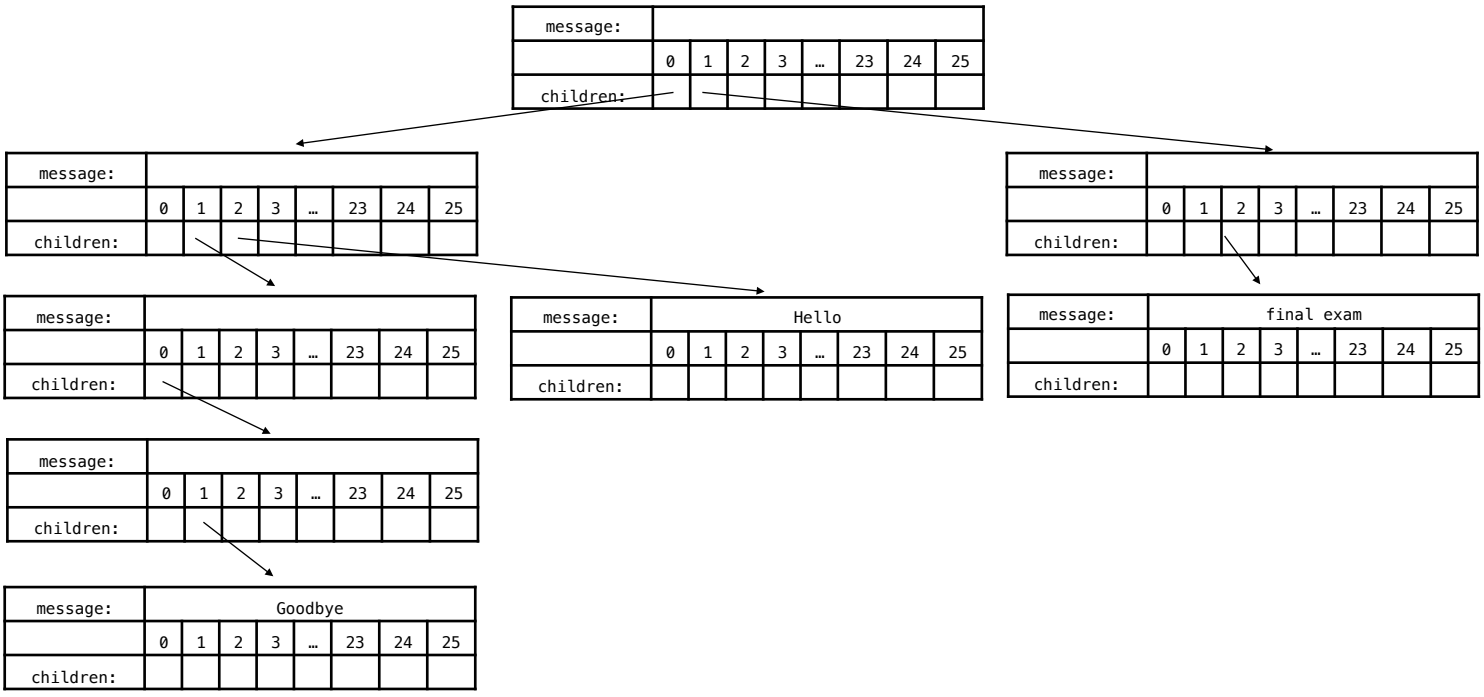
The user has previously typed "ab".

Since "ab" and "b" are both valid prefixes of sequences in the trie, we need to keep track of both as traversals down the trie that are "in progress", which we can do with pointers into the trie representing what node the traversal is currently on. Now, if the user types the letter "a" next, we need to update each of these in-progress traversals accordingly. The "ab" then becomes "aba". However, the "b" becomes "ba" which hits a `nullptr` in our trie (the 'a' child of the node corresponding to the sequence "b"). Because it hit a `nullptr`, we know that it cannot be a valid prefix for a sequence, so we no longer need to keep track of it as a progress traversal. Additionally, we need to keep track of the new "a" sequence, because this "a" could be the start of a new traversal, so we need to add just "a" to our list of in progress traversals.

Assume the user then types "b". We update "aba" to become "abab", and at this point, the program prints out "Goodbye". Additionally, all of the other valid traversals get reset, even if some are in progress (when a macro is printed, it resets the algorithm).

As such, you will use a variable `Vector<TrieNode *> actives` that you can use to keep track of active traversals of the trie. Thus, whenever `keyPressed` is called, you can advance each active traversal by whatever character was entered. If any traversal hits a `nullptr`, it must not be a valid start of a registered sequence, so we can remove it from the `actives` vector.

If a match is found, the entire actives Vector should be cleared (you may use the `.clear()` function of the Vector container to do this).

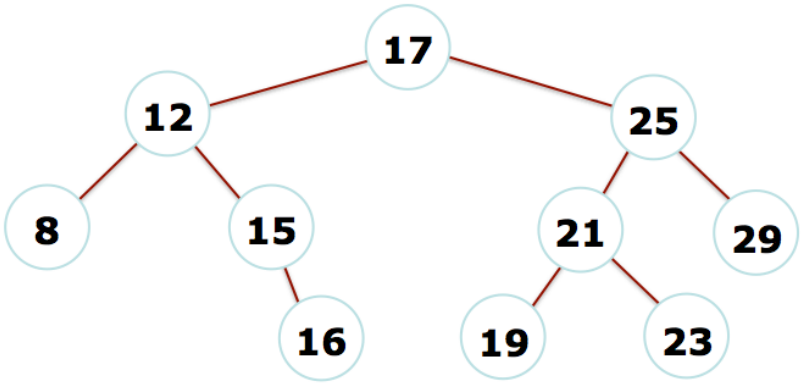


Answer problem 3b below.

```
void keyPressed(Vector<TrieNode *> &actives, TrieNode *root, char typed) {
    actives.add(root);
    for (int i = 0; i < actives.size(); i++) {
        actives[i] = actives[i]->children[typed - 'a'];
        if (actives[i] == nullptr) {
            actives.remove(i);
            i--;
        } else if (!actives[i]->message.isEmpty()) {
            cout << actives[i]->message;
            actives.clear();
            return;
        }
    }
}
```

Problem 4: AVL Trees (8 points)

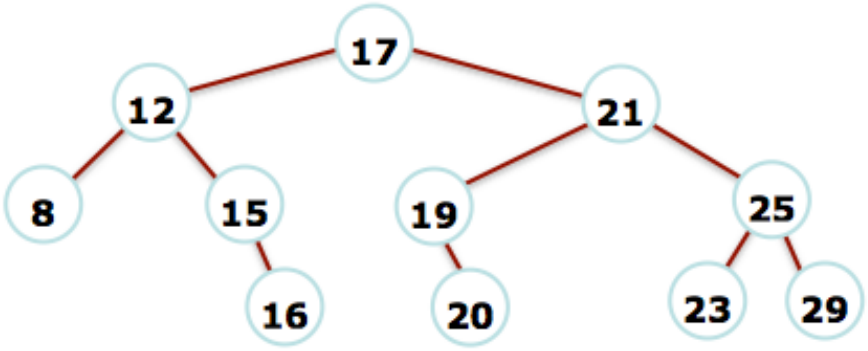
Using the proper AVL rotation(s), insert "**20**" into the AVL tree below and balance it using the algorithm we discussed in class. **Draw your answer in the box below.** (4 points)



A. Is this a single rotation, or a double rotation? How do you know? (2 points)

Single rotation at 25 because 20 is in the left subtree of the left child of 25, which is the first unbalanced node.

B. Draw your answer in the box below (don't forget to answer part A!) (6 points)



Problem 5, Graphs: Network Flow (15 points)

We briefly discussed how traffic on the Internet gets from your computer to the Web site that you're trying to read and back, and this problem will investigate a different method than the one discussed in class. It turns out that there are many different ways of routing traffic between two points on the Internet – after all, the system was designed to withstand a nuclear attack. The best way to implement that design was to provide for many different ways for data to get from one place to another. That way, if some of these paths were disrupted, there would be other ways for the data to get through. As a result, when you look at the computers on the Internet, you see a very large directed graph of interconnected nodes (individual computers). This means that any network traffic between two points can go over any of a number of different paths.

The whole Internet thus is just a large graph, and routing information between two computers is just a graphing problem. This means that you can use the handy `basicGraph` to model how network traffic is sent between computers. Our goal in this problem is to use a simplified model of the Internet built with the `basicGraph` to find the best path to use to route information from one computer to another.

Part (a) (10 points):
Our first step is to build a queue of all of the acyclic paths (ones with no cycles) between two nodes. Unlike in the Trailblazer project, by "path" we refer to a collection of `edge`s that connect the first point to the second point. The order of the edges doesn't matter – because each path is acyclic, it could be constructed from its edges without regard to their order.

What should we use to contain a collection of edges? We have a variety of collection ADTs that we could use – stacks, queues, trees, lists, etc. For this problem, we'll use a `stack` of edges to represent a path. In order for this to work, you can assume that you can copy a stack simply by assigning one stack to another:

```
Stack<Edge *> newStack = oldStack;
```

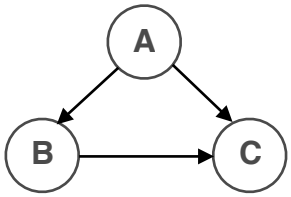
It makes a complete copy of the given stack and returns to you a newly allocated stack with the same contents. You'll find this helpful in extending your paths as you search through the graph.

Your task is to write this function:

```
Queue<Stack<Edge*>> findAllPaths(BasicGraph &graph, Vertex* start, Vertex* end);
```

This function returns a queue like the one we just described, which contains all of the acyclic paths between `startNode` and `endNode`. Each path must be represented by a stack of edges.

As an example, consider this graph:



There are two paths to get from node A to node C in this graph, so your function should build a queue with the following two elements in any order:

- A stack whose elements are the A→B edge and the B→C edge
- A stack whose only element is the A→C edge

That queue has all possible paths from A to C that don't contain a cycle, so you could return it as the solution.

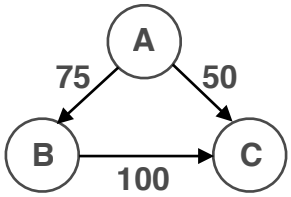
Write your solution to Problem 5, part (a) here. You must use this prototype, though you are free to write helper functions as needed.

```
Queue<Stack<Edge*>> findAllPaths(BasicGraph &graph, Vertex* start, Vertex* end) {
    Queue<Path> queue;
    Path stack;
    recursiveFindAllPaths(graph, start, end, queue, stack);
    return queue;
}

void recursiveFindAllPaths(BasicGraph &graph, Vertex* start, Vertex* end,
                           Queue<Path> &queue, Path &path) {
    if (start->visited) {
        return;
    }

    if (start == end) {
        Path newPath = path;
        queue.enqueue(newPath);
        return;
    }

    start->visited = true;
    Set<Edge *>allEdges = graph.getEdgeSet();

    for (Edge *edge : allEdges) {
        if (edge->start == start){
            path.push(edge);
            recursiveFindAllPaths(graph, edge->end, end, queue, path);
            path.pop();
        }
    }
    start->visited = false;
}
```

Question 5, Part (b) (5 points):

We can rate paths by their throughput, or the amount of network traffic that can be sent over that edge. We will define the best path as the path with the highest throughput.

To determine the throughput of a path, we note that the amount of traffic that we can send over a path is limited by the smallest-weight edge on the path. That edge is a bottleneck for the entire path. The best path, then, is the one with the largest minimum weight – the one whose smallest- weight edge is larger than the smallest-weight edge on any other path. For example, consider the graph from above, but with weights added for each edge:



If we are trying to find the best path from A to C, we know from part (a) that we have two paths: A→C and A→B→C. The smallest-weight edge on the A→C path has weight 50, but the edge with the smallest weight edge on the A→B→C path has weight 75, so we can send more traffic along that path than along the A→C path.

Given the queue of paths from part (a), write a function `bestPath` that returns the path with the highest throughput. Furthermore, we expect our network to guarantee a minimum throughput. If you find that no path exists such that the minimum throughput is met, throw an error.

You may use the integers INT_MAX and/or INT_MIN in your solution.

Write your solution to Problem 5, part (b) here:

```
// Note: Students could use DBL_MIN instead of INT_MIN, and double bestThroughput
Stack<Edge*> bestPath(Queue<Stack<Edge*>> paths, int guaranteedThroughput) {
    int bestThroughput = INT_MIN;
    Stack<Edge*> bestPathSoFar;
    while(!paths.isEmpty()) {
        Stack<Edge*> path = paths.dequeue();
        int pathThroughput = minWeight(path);
        if (pathThroughput > bestThroughput) {
            bestPathSoFar = path;
            bestThroughput = pathThroughput;
        }
    }
    if (bestThroughput < guaranteedThroughput) {
        throw ("Network not able to meet guaranteed throughput!");
    }
    return bestPathSoFar;
}
// helper function (could inline)
int minWeight(Stack<Edge*>path) {
    int min = INT_MAX;
    while (!path.isEmpty()) {
        Edge *edge = path.pop();
        if (edge->cost < min) {
            min = edge->cost;
        }
    }
    return min;
}
```

Problem 6. Inheritance and Polymorphism (8 points)

Consider the following classes:

```
class Stanford {
public:
    virtual void a() {
        cout << "S A" << endl;
        c();
    }

    virtual void c() {
        cout << "S C" << endl;
    }
};

class Harvard : public Stanford {
public:
    virtual void b() {
        a();
        cout << "H B" << endl;
    }

    virtual void c() {
        cout << "H C" << endl;
    }
};

class Yale : public Stanford {
public:
    virtual void b() {
        a();
        cout << "Y B" << endl;
    }

    virtual void c() {
        cout << "Y C" << endl;
        Stanford::c();
    }

    virtual void d() {
        cout << "Y D" << endl;
        c();
    }
};

class Berkeley : public Yale {
public:
    virtual void a() {
        cout << "B A" << endl;
    }

    virtual void c() {
        cout << "B C" << endl;
        Stanford::c();
    }
};
```

Now assume that the following variables are defined:

```
Stanford*    var1   =   new   Harvard();
Yale*        var2   =   new   Berkeley ();
Stanford*    var3   =   new   Yale();
Stanford*    var4   =   new   Berkeley();
Stanford*    var5   =   new   Stanford();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "x / y / z" to indicate three lines of output with "x" followed by "y" followed by "z". If the statement does not compile, write "compiler error". If a statement would crash at runtime or cause unpredictable behavior, write "crash".

Statement	Output
`var1->a();`	S A / H C
`var1->b();`	Compiler Error
`var1->c();`	H C
`var2->a();`	B A
`var2->b();`	B A / Y B
`var2->c();`	B C / S C
`var2->d();`	Y D / B C / S C
`var3->a();`	S A / Y C / S C
`var3->b();`	Compiler Error
`var4->a();`	B A
`var5->a();`	S A / S C
`((Yale*) var1)->a();`	S A / H C
`((Harvard*) var1)->b();`	S A / H C / H B
`((Berkeley*) var2)->d();`	Y D / B C / S C
`((Yale*) var3)->b();`	S A / Y C / S C / Y B
`((Yale*) var4)->a();`	B A
`((Yale*) var4)->b();`	B A / Y B
`((Harvard*) var5)->b();`	Undefined / CRASH

Problem 7. Bonus (2 points)

7a. (1 point) What subway system models the skip list data structure almost perfectly?

New York City

7b. (1 point) What animal do computer scientists use to remotely turn on their microwave ovens?

Python (http://starecat.com/i-figured-out-how-to-turn-on-my-microwave-using-python/)

7c. (1 point) Chris wants to replace the "flood fill" part of the recursion assignment with something different. What did he say he was going to replace it with?

Mandelbrot Set

(alternate: Fractal from midterm)