CS 106X Sample Final Exam #1

This sample exam is intended to demonstrate an example of some of the kinds of problems that will be asked on the actual final exam. We do not guarantee that the number of questions in this sample exam will match the number on the real exam, nor that every kind of problem shown here will exactly match the kinds of problems shown on the final exam, though the real exam will be generally similar overall. Also, to save paper, this exam does not provide blank pages for writing space. The real exam will provide more blank space for you to write your answers.

1. Linked Lists (read)

Consider the following linked list of `ListNode` objects, with a pointer named `front` that points to the first node:

```
front -> [10] -> [20] -> [40] -> [30] -> [90] -> [80] -> [70] -> [60] -> [100] -> [0] /
```

Draw the state of the linked list after the following code runs on it. If a given node is removed from the list, you don't need to draw that node, only the ones that remain reachable in the original list.

```
void linkedListMystery1(ListNode*& front) {
    ListNode* curr = front;
    while (curr != nullptr) {
        if (curr->next != nullptr && curr->data > curr->next->data) {
            curr->data++;
            ListNode* temp = curr->next;
            curr->next = curr->next->next;
            temp->next = curr;
        }
        curr = curr->next;
    }
}
```

---

2. Linked Lists (write)

Write a function **expand** that manipulates a linked list represented by a chain of `ListNode` structures as shown in lecture and section *(see reference sheet)*. The function accepts two parameters: a reference to a `ListNode` pointer to the front of the list, and an integer parameter *k*. Your function's behavior is to replace each node of the list with *k* new nodes, each containing a fraction of the original node's value. Specifically, if the original node's value was *v*, each new node's value should be *v* / *k*. Suppose a variable named `list` points to the front of a list storing the following values:

```
{12, 34, -8, 3, 46}
```

The call of `expand(list, 2);` would change the list to store the following elements. Notice how 12 becomes two 6es, and 34 becomes two 17s, and -8 becomes two -4s, and so on. The value 3 doesn't divide evenly by 2, so each new element is the result of truncated integer division of 3 / 2, which yields 1. The 46 becomes two 23s.

```
{6, 6, 17, 17, -4, -4, 1, 1, 23, 23}
```

If we had instead made the call of `expand(list, 3);` on the original list, the list would store the following elements:

```
{4, 4, 4, 11, 11, 11, -2, -2, -2, 1, 1, 1, 15, 15, 15}
```

If the list is empty, it should remain empty after the call. If the value of *k* passed is 0, you should remove all elements from the list. If the value of *k* passed is negative, do not modify the list and instead throw an integer **exception**.

For full credit, obey the following restrictions in your solution:

- Do not **modify the `data` field** of any nodes; you must solve the problem by changing links between nodes and adding newly created nodes to the list. This means that the list's original nodes must be discarded as they are replaced by the new expanded nodes your code is creating.
- Do not use any auxiliary **data structures** such as arrays, vectors, queues, maps, sets, strings, etc.
- Do not **leak memory**; if you remove nodes from the list, free their associated memory.
- Your code must run in no worse than **O(*N*) time**, where *N* is the length of the list. Furthermore, for full credit you must not make any unnecessary passes over the list.

3. Binary Search Trees (read)

(a) Write the binary search tree that would result if these elements were **added** to an empty **binary search tree** (a simple BST, not a re-balancing AVL tree) in this order:

- Dodo, Eaglet, Rabbit, Cat, Alice, Jabberwock, Hatter, Tweedledee, Queen, Bill

(b) Write the elements of your above tree in the order they would be visited by each kind of **traversal**:

- Pre-order: _____
- In-order: _____
- Post-order: _____

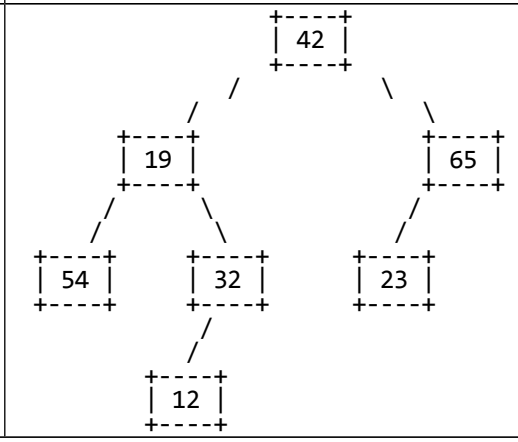
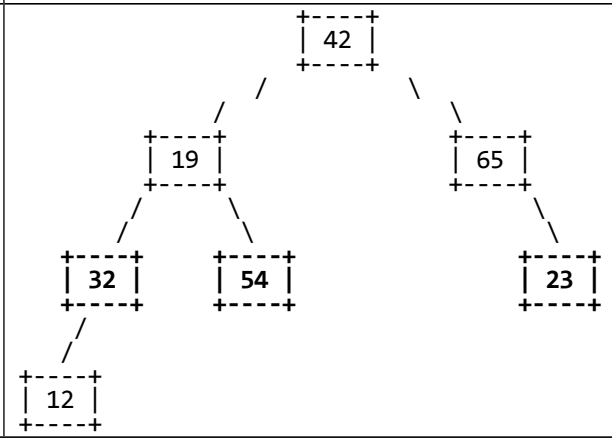
(c) Now draw what would happen to your tree from the end of (a) if the following values were **removed**, in this order (assuming that the BST remove function follows the algorithm shown in the class lecture slides):

- Hatter, Cat, Rabbit, Dodo

4. Binary Trees (write)

Write a function **`swapChildrenAtLevel`** that manipulates a binary tree of `TreeNode` structures as shown in class representing an unordered binary tree (see reference sheet). Your function should accept two parameters: a reference to a pointer to the root of a tree, and an integer *k*, and should swap the left and right children of all nodes at level *k*. In other words, after your function is run, any node at level (*k*+1) that used to be its parent's left child should now be its parent's right child and vice versa. For this problem, the overall root of a tree is defined to be at level 1, its children are at level 2, etc. The table below shows the result of calling this function on a particular tree.

```
TreeNode* tree = ...;
...
swapChildrenAtLevel(tree, 2);
```

Level	Before Call	After Call
1		
2		
3		
4		

If *k* is 0 or less than 0, your function should throw an integer **exception**. If the tree is empty or does not have any nodes at the given level or deeper, it should not be affected by a call to your function.

For efficiency, your function should not traverse any parts of the tree that it does not need to traverse. Specifically, you should not access any nodes lower than level (*k*+1), because there is nothing there that would be changed.

You may define private helper functions to solve this problem. Do not create any data structures such as arrays, vectors, etc. You should not construct any new tree node objects or change the `data` of any nodes, though you can declare pointers if you like. Your solution must be **recursive**.

5. Hashing (read)

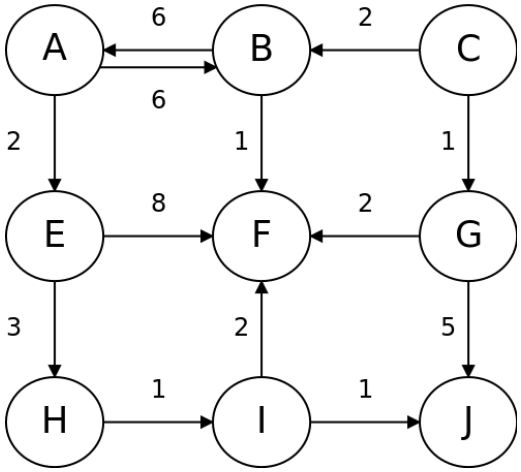
Simulate the behavior of a hash map as described and implemented in lecture. Assume the following:

- the hash table array has an initial capacity of **10**
- the hash table uses **separate chaining** to resolve collisions
- the **hash function** returns the absolute value of the integer key, mod the capacity of the hash table
- **rehashing** occurs at the *end* of an add where the load factor is ≥ **0.5** 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. Also write the size, capacity, and load factor of the final 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**.

```
HashMap map;
map.put(8, 11);
map.put(26, 9);
map.put(16, 9);
map.put(26, 88);
map.put(196, 44);
map.put(38, 11);
map.put(32, 77);
map.remove(26);
map.remove(9);
map.put(100, 44);
if (map.containsKey(196)) {
    map.remove(44);
}
map.put(56, 56);
map.put(-48, 34);
```

6. Graphs (read)



For the graph shown above, answer the following questions:

a) Is the graph directed or undirected? _____

b) Is the graph weighted or unweighted? _____

c) Is the graph connected or unconnected? _____
If it is not connected, give an example of why not.

d) Is the graph cyclic or acyclic? _____
If it is cyclic, give an example of a cycle.

e) What is the degree of each vertex? _____
If it is directed, what is the in-degree and out-degree of each vertex?

f) Write a complete *adjacency list* and *adjacency matrix* representation of the graph.

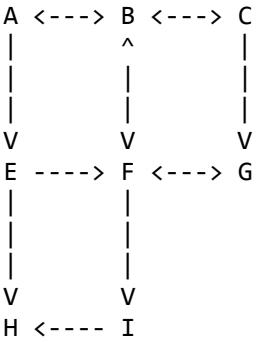
g) Write the order that a *depth-first search* (DFS) would visit vertexes if it were looking for a path from vertex A to vertex I. Also write the path it would return. Assume that any "for-each" loop over neighbors returns them in ABC order.

h) Write the order that a *breadth-first search* (BFS) would visit vertexes if it were looking for a path from vertex C to vertex H. Also write the path it would return. Assume that any "for-each" loop over neighbors returns them in ABC order.

7. Graphs (write)

Write a function **containsBidiPath** that accepts two parameters: a reference to a `BasicGraph`, and a reference to a `Vector` of pointers to vertexes representing a path; and returns `true` if that path is can be traveled in the graph *in both directions*, or `false` if not. In other words, you would return `true` if every vertex in the path is found in the graph, and where there is an edge between each neighboring pair of vertexes in the path along the way from start to end, and also back from the end to the start.

For example, if your function were passed given the graph below and the path {A, B, F, G}, you would return `true` because all of those vertexes are part of the graph and you can travel that path from A to G and back from G to A. If you were passed the same graph and the path {A, E, F, I}, you would return `false` because that path does not go back from I to A. If you were passed the path {A, X, Z, B}, you would return `false` because X and Z are not in the graph. You should not construct any auxiliary data structures while solving this problem, but you can construct as many simple variables as you like. You should not modify the state of the graph passed in. You may assume that the graph's state is valid, and that the path and its elements are not null.



8. Inheritance and Polymorphism (read)

Consider the following classes; assume that each is defined in its own file.

```
class Eddie : public Kurt {
public:
    virtual void b() {
        a();
        cout << "E B" << endl;
    }

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

class Kurt {
public:
    virtual void a() {
        cout << "K A" << endl;
        c();
    }

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

class Chris : public Jerry {
public:
    virtual void b() {
        a();
        cout << "C B" << endl;
    }

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

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

class Jerry : public Kurt {
public:
    virtual void a() {
        cout << "J A" << endl;
    }

    virtual void c() {
        cout << "J C" << endl;
        Kurt::c();
    }
};
```

Now assume that the following variables are defined:

```
Kurt*  var1 = new Jerry();
Jerry* var2 = new Chris();
Kurt*  var3 = new Eddie();
Kurt*  var4 = new Chris();
Kurt*  var5 = new Kurt();
```

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();`	_____
`var1->b();`	_____
`var1->c();`	_____
`var2->a();`	_____
`var2->b();`	_____
`var2->c();`	_____
`var3->a();`	_____
`var3->b();`	_____
`var4->a();`	_____
`var5->a();`	_____
`((Jerry*) var1)->a();`	_____
`((Jerry*) var1)->b();`	_____
`((Chris*) var2)->d();`	_____
`((Eddie*) var3)->b();`	_____
`((Jerry*) var4)->a();`	_____
`((Jerry*) var4)->b();`	_____
`((Jerry*) var5)->b();`	_____

9. Object-Oriented Programming and Inheritance (write)

You have been asked to extend an existing class `Dice` representing a set of 6-sided dice that can be rolled by a player.

Member	Description
`private: int* diceValues`	an array of all dice values rolled
`private: int count`	length of `diceValues` array
`Dice(int count)`	constructs a dice roller to roll the given # of dice; all dice initially have the value of 6
`virtual int getCount() const`	returns # of dice as passed to constructor
`virtual int getValue(int index) const`	returns the die value (1-6) at the given 0-based index
`virtual void roll(int index)`	rolls the given die to give it a new random value (1-6)
`virtual int total() const`	returns sum of all current dice values in this dice roller
`virtual string toString() const`	returns string of dice values, e.g. `"{4, 1, 6, 5}"`
`ostream& operator <<(ostream& out, Dice& dice)`	prints the given dice in its `toString` format

Define a new class called `RiggedDice` that extends `Dice` through inheritance. Your class represents dice that let a player "cheat" by ensuring that every die always rolls a value that is greater than or equal to a given minimum value.

You should provide the same member functions as the superclass, as well as the following new public behavior.

Constructor/member function	Description
`RiggedDice(int count, int min)`	constructs a rigged dice roller to roll the given # of dice, using the given minimum value for all future rolls; all dice initially have the value 6 (if the minimum value is not between 1-6, throw an integer exception)
`virtual int getMin() const`	returns minimum roll value as passed to constructor

For all inherited behavior, `RiggedDice` should behave like a `Dice` object except for the following differences. You may need to override or replace existing behavior in order to implement these changes.

- Every time a die is **rolled**, you must ensure that the value rolled is greater than or equal to the minimum value passed to your constructor. Do this by re-rolling the die if the value is too small, as many times as necessary.
- The rigged dice should return a **total** that lies and claims to be 1 higher than the actual total. For example, if the sum of the values on the dice add up to 13, your rigged dice object's `total` returned should be 14.
- When a rigged dice object's `toString` is called or when it is printed, it should display that the dice are rigged, then the dice values, then the minimum dice value, such as, `"rigged {4, 3, 6, 5} min 2"`

Also **make `RiggedDice` objects comparable to each other by defining a `<` operator**. `RiggedDice` are compared by total dice value in ascending order, breaking ties by minimum roll value in ascending order. In other words, a `RiggedDice` with a lower total dice value is considered to be "less than" one with a higher total. If two objects have the same total, the one with a lower min value passed to its constructor is "less than" one with a higher min value. If the two objects have the same total and the same min, they are considered to be "equal." Though a proper implementation would define other operators such as `<=`, `>`, `>=`, `==`, and `!=`, we ask you to define just this one operator.

Write the .h and .cpp parts of the class separately with a line to separate them. The majority of your score comes from implementing the correct behavior. You should also appropriately utilize the behavior you have inherited from the superclass and not re-implement behavior that already works properly in the superclass. You may assume that the superclass already checks all arguments passed to its constructor and members to make sure that they are valid.