CS 106X Sample Final Exam #2

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. Array List Implementation (write)

In lecture, we discussed the implementation of a class called **`ArrayIntList`**, an implementation of a list of integers using an internal array. It was our own `Vector` of `int`s. Add a member function to this class called **`stretch`** that accepts an integer *k* as a parameter and that replaces each integer in the original list with *k* copies of that integer. For example, if an `ArrayIntList` variable called `list` stores this sequence of values:

```
{18, 7, 4, 4, 24, 11}
```

And the client makes the following call of `list.stretch(3);` , the list should be modified to store the following values. Notice that there are three copies of each value from the original list because `3` was passed as the parameter value.

```
{18, 18, 18, 7, 7, 7, 4, 4, 4, 4, 4, 4, 24, 24, 24, 11, 11, 11}
```

If the value of *k* is less than or equal to 0, the list should be empty after the call.

Remember that an array list has an internal "unfilled" array whose **capacity** might be larger than its size. Note that this member function might require more capacity than your list's array currently has. If so, you must handle this by **resizing** to a larger array if necessary. You should not create any auxiliary arrays unless it is absolutely necessary to do so to solve the problem. If the list's existing internal array already has enough capacity, you should perform the modification in place without using any auxiliary data structures.

Constraints: For full credit, obey the following restrictions in your solution. A violating solution can get partial credit.

- Do not call any **member functions** of the `ArrayIntList`. e.g. **Do not call `add`**, `insert`, `remove`, or `size`. You may, of course, refer to the private member variables inside the list.
- You should not create any auxiliary arrays unless it is absolutely necessary to do so to solve the problem. For example, if the list's existing internal array already has enough capacity to fit the stretched items, do not create any second array; perform the modifications entirely in place.
- You may create a **single auxiliary array** if necessary to store the elements in the list *(see previous bullet)*. Outside of this, do not use any other auxiliary **data structures** such as vectors, queues, maps, sets, strings, etc.
- Do not **leak memory**; if you cease using any dynamically allocated (via `new`) memory, free it.
- Your code must run in no worse than **O(*N*) time**, where *N* is the length of the list.

You should write the member function's body as it would appear in `ArrayIntList.cpp`. You do not need to write the function's header as it would appear in `ArrayIntList.h`. Write only your member function, not the rest of the class.

2. Linked Lists (read)

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

```
front -> [1] -> [2] -> [4] -> [3] -> [5] -> [7] -> [11] -> [0] -> [6] -> [1] -> [1] /
```

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 linkedListMystery2(ListNode*& front) {
    ListNode* curr = front;
    ListNode* prev = nullptr;
    while (curr->next != nullptr) {
        if (curr->data % 2 == 0 && prev != nullptr) {
            prev->next = curr->next;
        } else {
            curr->data--;
        }
        prev = curr;
        curr = curr->next;
    }
}
```

3. Linked Lists (write)

Write a function **`combineDuplicates`** that manipulates a list of `ListNode` structures class from lecture and section *(see reference sheet)*. The function modifies the list by merging any consecutive neighboring nodes that contain the same element value into a single node whose value is the **sum** of the merged neighbors. For example, suppose a pointer named **`list`** points to the front of a list containing the following values. Below is the result of a call of `combineDuplicates(list);` on the list. The underlined areas represent the neighboring duplicate elements that are merged in the final result.

```
{3, 3, 2, 4, 4, 4, -1, -1, 4, 12, 12, 12, 12, 48, -5, -5}        list


{3, 3, 2, 4, 4, 4, -1, -1, 4, 12, 12, 12, 12, 48, -5, -5}        combineDuplicates(list);

{6, 2, 12, -2, 4, 48, 48, -10}                                    result
```

If the list is empty or contains no duplicates, it should be unchanged by a call to your function.

Constraints: For full credit, obey the following restrictions in your solution. A violating solution can get partial credit.

- **It is okay to modify the `data` field** of existing nodes, if you like.
- You **may not create any new nodes** by calling `new ListNode(...)`.
  You may create as many `ListNode*` pointers as you like, though.
- 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.
- Your code must solve the problem by making only a **single traversal** over the list, not multiple passes.

4. Binary Search Trees (read) ABCDEFGHIJKLMNOPQRSTUVWXYZ

(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. *(See alphabet guide at top-right if needed.)*

- Cersei, Arya, Jamie, Littlefinger, Danaerys, Tyrion, Ned, Stannis, Varys, Ramsay, Bran, Hodor

(b) Examine your tree from (a) and answer the following questions about it.

- Is the overall tree balanced? Circle one. **Yes** **No**
- If the tree is **balanced**, briefly explain how you know this by writing your written justification next to the tree. If the tree is ***not*** **balanced**, circle and/or clearly mark **<u>all</u>** node(s) that are unbalanced.

(c) Now draw below what would happen to your tree from the end of (a) if all of the following values were **removed**, in this order *(using the BST remove algorithm shown in lecture)*:

- Littlefinger, Cersei, Jamie

5. Binary Trees (write)

Write a member function **`hasPath`** that interacts with a tree of `TreeNode` structures as seen in class from lecture, representing an unordered binary tree *(see reference sheet)*.

The function accepts integers *start* and *end* as parameters and returns `true` if a path can be found in the tree from *start* down to *end*. In other words, both *start* and *end* must be element data values that are found in the tree, and *end* must be below *start*, in one of *start*'s subtrees; otherwise the function returns `false`. If *start* and *end* are the same, you are simply checking whether a node exists in the tree with that data value. If the tree is empty, your function should return `false`.

For example, suppose a `TreeNode` pointer named `tree` points to the root of a tree storing the following elements. The table below shows the results of several various calls to your function:

```
           67
        /      \
    80           52
  /    \        /
16      21     99
       /
    45
```

Call	Result	Reason
`hasPath(tree, 67, 99)`	`true`	path exists: 67 → 52 → 99
`hasPath(tree, 80, 45)`	`true`	path exists: 80 → 21 → 45
`hasPath(tree, 52, 99)`	`true`	path exists: 52 → 99
`hasPath(tree, 67, 45)`	`true`	path exists: 67 → 80 → 21 → 45
`hasPath(tree, 16, 16)`	`true`	node exists with `data` of 16
`hasPath(tree, 99, 67)`	`false`	nodes do exist, but in wrong order
`hasPath(tree, 80, 99)`	`false`	nodes do exist, but there is no path from 80 to 99
`hasPath(tree, 67, 100)`	`false`	end of 100 doesn't exist in the tree
`hasPath(tree, -1, 45)`	`false`	start of -1 doesn't exist in the tree
`hasPath(tree, 42, 64)`	`false`	start/end of -1 and 45 both don't exist in the tree

Your function **should not modify** the tree's state.

For full credit, your solution should be **efficient**. Specifically, you should not traverse over the same nodes or subtrees multiple times. You also should not explore regions of the tree if you do not need to. For example, if you find a path between *start* and *end*, your algorithm should stop without exploring the rest of the tree.

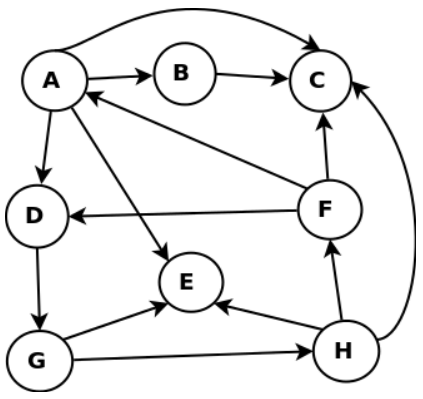
Constraints: For full credit, obey the following constraints in your solution. A violating solution can get partial credit.

- Do not modify the tree's state in any way.
  For example, do not change the `data` field of any existing nodes of the tree, nor its `left` or `right` pointers.
- Do not **leak memory**. If you remove a node from the tree, free its memory.
- Do not create any **data structures** (arrays, vectors, sets, maps, etc.).
- Do not construct **new node** objects or modify the tree in your code. You can declare pointers if you like.
- For full credit, your solution should be at worst O(*N*) time, where *N* is the number of elements in the tree.
  You must also solve the problem using a **single pass** over the tree, not multiple passes.
- You <u>may</u> define **private helper** functions if you like.
- Your solution must be **recursive**.

6. Graphs (write)

Write a function named **`findLongestPath`** that accepts as a parameter a reference to a `BasicGraph`, and returns a `Vector` of `Vertex` pointers representing the longest possible "simple" path between any two vertexes in that graph. A simple path is a non-cycle path that does not repeat any vertexes. Your algorithm does not consider edge weight/cost, only path length.

The diagram below shows an example graph that might be passed to your algorithm. In the following graph, the longest possible simple path is D -> G -> H -> F -> A -> B -> C, which has length 7. So when passed the graph below, your function would return a `Vector` containing pointers to those vertexes in that order. If there is a tie and there are multiple longest paths of exactly the same length, your function can return any one of those equally longest paths.



For homework you learned and wrote several algorithms for efficiently finding shortest paths and minimum-weight paths. It turns out that the task of finding *longest* paths, like you're doing in this problem, doesn't have a known clever algorithm. You need to literally try generating all possible simple paths in the graph and discover which one is the longest through brute force. So you should come up with an algorithm that can enumerate every valid simple path in the graph and find the longest such one. Note that the graph might be cyclic, as in the graph above that contains cycles such as A -> D -> G -> H -> F -> A.

Although you must try every possible path, you should still write code that is **efficient**. If you explore large numbers of paths and possibilities multiple times, or continue exploring paths that are certain to be dead-ends, you may lose points.

You may assume that the graph's state is valid, and that it is **directed** and **unweighted**, and that it contains no self-edges (e.g. from *V1* to *V1*). You may also assume that there is at most one directed edge from any vertex *V1* to any other vertex *V2*. You may define **private helper** functions if so desired, and you may construct auxiliary **collections** as needed to solve this problem. You should not modify the contents of the graph such as by adding or removing vertexes or edges from the graph, though you may modify the state variables inside individual vertexes/edges such as `visited`, `cost`, and `color`.

If the graph does not contain any vertexes and/or edges, your method should return an empty `Vector`.

7. Hashing (read)

Simulate the behavior of a **hash map** of integers 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(19, 9);
map.put(4, 4);
map.put(44, 19);
map.remove(9);
map.put(23, 54);
map.put(73, 54);
map.put(83, 9);
map.put(99, 4);
map.remove(4);
map.put(0, 0);
map.put(-2, -88);
if (!map.containsKey(73)) {
    map.put(66, 77);
}
map.put(333, 0);
```

8. Inheritance and Polymorphism (read)

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

```
class Golbez : public Cecil {
public:
  virtual void m2() {
    cout << "Golbez m2" << endl;
    m1();
  }
};

class Rosa : public Cecil {
public:
  virtual void m3() {
    cout << "Rosa m3" << endl;
    m2();
  }

  virtual void m4() {
    cout << "Rosa m4" << endl;
  }
};

class Cecil : public Kain {
public:
  virtual void m1() {
    cout << "Cecil m1" << endl;
    Kain::m1();
  }

  virtual void m3() {
    cout << "Cecil m3" << endl;
    m2();
  }
};

class Kain {
public:
  virtual void m1() {
    cout << "Kain m1" << endl;
  }

  virtual void m2() {
    m1();
    cout << "Kain m2" << endl;
  }
};
```

Now assume that the following variables are defined:

```
Kain*  var1 = new Cecil();
Cecil* var2 = new Rosa();
Kain*  var3 = new Golbez();
Cecil* var4 = new Golbez();
```

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 other unpredictable behavior, write "**crash**".

Statement	**Output**
`var1->m1();`	
`var1->m2();`	
`var1->m3();`	
`var2->m3();`	
`var3->m2();`	
`var3->m4():`	
`var4->m3();`	
`((Cecil*) var1)->m3();`	
`((Rosa*) var1)->m4();`	
`((Rosa*) var2)->m4();`	
`((Golbez*) var3)->m4();`	

8. Object-Oriented Programming and Inheritance (write)

You have been asked to extend a pre-existing class named `Calculator` that performs various calculations on integers. The `Calculator` class includes the following members:

Member	Description

`private:`
`    int m_seed;`
	private data of the calculator; its random number seed

`Calculator(int seed)`	constructs a `Calculator` with the given seed for random numbers
`virtual int fib(int k)`	returns the *k*th Fibonacci number (assumes $k \geq 1$)
`virtual int getSeed() const`	returns the random number seed passed to the constructor
`virtual bool isPrime(int n)`	returns `true` if *n* is a prime number
`virtual int kthPrime(int k)`	returns the *k*th prime number (assumes $k \geq 1$)
`virtual int rand(int max)`	returns a random value between 0 and *max*, inclusive
`virtual string toString() const`	returns a string representation of the calculator

The class correctly computes its results, but it does so inefficiently. In particular, it often computes the same prime numbers more than once. Suppose that the call of `kthPrime(30)` is made 100 times by client code; there is no reason to compute that same value 100 different times. Instead the calculator could compute it once and store its value, so that the 99 calls after the first simply return the "remembered" value. This idea is called "memoizing".

You are to **define a new class called `MemoryCalculator` that extends `Calculator` through inheritance**. A `MemoryCalculator` should behave like a `Calculator` except that it implements "memoizing" to speed up the computation of primes. Your memo calculator should guarantee that the value of `kthPrime(k)` is computed only once for any given value *k*. Your class should still rely on the `Calculator` class to compute prime numbers when necessary. You are simply guaranteeing that the computation is not performed more than once for any particular value of *k*. You should not make any assumptions about how large *k* might be or about the order in which the function is called with different values of *k*. The `isPrime` member function calls `kthPrime`, so it does not need to be memoized. You also do not need to memoize the Fibonacci (`fib`) computation for this problem.

You should provide the same member functions as the superclass, extended/overridden as necessary to modify their behavior as previously described. You should also provide the following new public members that will allow a client to find out how many values have been directly computed versus how many calls have been handled through memoization:

Member	Description
`MemoryCalculator(int seed)`	constructs a `MemoCalculator` with the given seed for generating random numbers
`virtual int getComputeCount()`	returns the number of prime numbers that were actually computed by a call to the superclass's `kthPrime` method
`virtual int getMemoCount()`	returns number of previous calls that were handled through memoization (rather than actually computing the value)

You must also **write operators `==` and `!=` for comparing `MemoryCalculator` objects for equality**. Two `MemoCalculator`s are equal if they have exactly the same state, including their random number seed, compute count, memo count, and exactly what numbers (if any) have been memoized. If any of the state differs, the two objects are considered to be unequal.

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.