CS 106B Practice Final Exam #8

(based on Autumn 2015 final exam)

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 (write)

Write a member function **`partitionSort`** to be added to the `LinkedIntList` class from lecture *(see reference sheet)*. Your function should assume that the linked list's elements are already sorted by absolute value, and rearrange the list into sorted order. Suppose a `LinkedIntList` variable named `list` stores the values below. Notice that the values are in order of absolute value; that is, they would be in sorted order if you ignored the sign of each value. The call of `list.partitionSort();` should reorder the values into sorted, non-decreasing order (including sign).

```
{0, 0, -3, 3, -5, 7, -9, -10, 10, -11, -11, 11, -11, 12, -15}    list

                                                                  after
{-15, -11, -11, -11, -10, -9, -5, -3, 0, 0, 3, 7, 10, 11, 12}    list.partitionSort();
```

Because the list is sorted by absolute value, you can solve this problem very efficiently. Your solution is required to run in O(*N*) time where *N* is the length of the list. If the list is empty or contains only one element, it should be unchanged by a call to your function. The behavior of your function is undefined if the initial list is not already sorted by absolute value; you do not need to handle that case.

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

- **Do not modify the `data` field** of existing nodes.
- **Do not create any new nodes** by calling `new ListNode(...)`.
  You may create as many `ListNode*` pointers as you like, though.
- Do not call any **member functions** of the `LinkedIntList`. For example, do not call `add`, `remove`, or `size`.
  You may, of course, refer to the private member variables inside the `LinkedIntList`.
  Note that the list does *not* have a `size` or `mysize` field.
- 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.

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

---

2. Heaps (read)

In lecture and homework, we discussed the implementation of a priority queue using a vertically-ordered tree called a *heap*. Recall that a heap "bubbles" elements up and down as they are added and removed to maintain its vertical ordering.

Given the following string/priority pairs:

- A:6, B:10, C:11, D:7, E:4, F:5, G:12, H:2, I:8, J:3, K:1, L:9

a) Draw the tree representation of the **binary heap** that results when all of the above elements are **enqueued** (added in the given order) with the given priorities to an initially empty heap. This is a "min-heap", that is, priorities with lesser integer values are higher in the tree. Circle the final tree that results from performing the additions. Also show the final array representation of the heap, assuming that the first (root) element is put into index 1.

b) After adding all the elements, perform **2 dequeue operations** (remove-min operations) on the heap. Circle the tree that results after the two elements are removed. Also show the final array representation of the heap.

Please show your work. You do not have to draw an entirely new tree after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to show the tree at various important stages to help earn partial credit in case of an error.

---

3. 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.)*

- Gideon, Darlene, Romero, FSociety, MrRobot, Elliot, Shayla, Angela, Ollie, Tyrell, Trenton

(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 **all** 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)*:

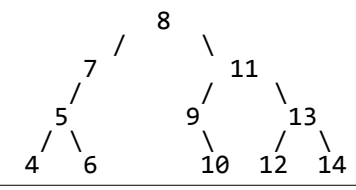
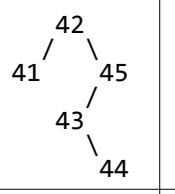
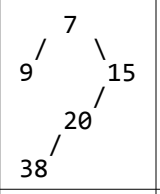
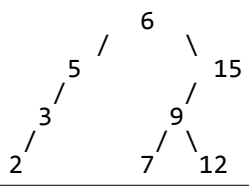
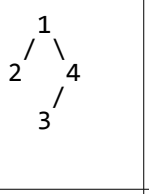
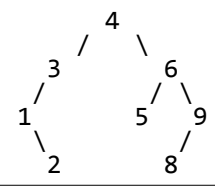
- Angela, Darlene, Gideon

4. Binary Trees (write)

Write a member function **`isConsecutive`** that could be added to the `BinaryTree` class from lecture *(see reference sheet)*.

Your function should examine the tree and return `true` if there is some integer value *k* such that an in-order traversal of the tree's elements would yield a sequence of consecutive integers *k*, *k*+1, *k*+2, ... where the integers differ by exactly 1 from their neighbors in the sequence; otherwise the function should return `false`. Recall that an in-order traversal visits nodes in left-center-right order. For example, suppose `BinaryTree` variables named `tree1`, `tree2`, ..., `tree6` store the trees of elements below.

- The call of `tree1.isConsecutive()` would return `true` because an in-order traversal of `tree1` produces the sequence of integers 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, which are consecutive integers from 4 through 14.
- The call of `tree2.isConsecutive()` would return `true` because an in-order traversal of `tree2` produces the sequence of integers 41, 42, 43, 44, 45, which are consecutive integers from 41 through 45.
- The call of `tree3.isConsecutive()` would return `false` because an in-order traversal of `tree3` produces the sequence of integers 9, 7, 38, 20, 15, which is not a sequence of consecutive integers.
- The call of `tree4.isConsecutive()` would return `false` because an in-order traversal of `tree4` produces the sequence of integers 2, 3, 5, 6, 7, 9, 12, 15, which is not a sequence of consecutive integers. *(It is ascending, but not consecutive.)*
- The call of `tree5.isConsecutive()` would return `false` because an in-order traversal of `tree5` produces the sequence of integers 2, 1, 3, 4, which is not a sequence of consecutive integers. *(The integers could be rearranged into a consecutive sequence, but an in-order traversal does not produce a consecutive sequence.)*
- The call of `tree6.isConsecutive()` would return `false` because an in-order traversal of `tree6` produces the sequence of integers 1, 2, 3, 4, 5, 6, 8, 9, which is not a sequence of consecutive integers *(missing a 7)*.

tree1	tree2	tree3	tree4	tree5	tree6
					
true	true	false	false	false	false

If the tree is empty or contains only a single element, your function should return `true`.

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. Once your code knows for sure whether the tree could or could not be consecutive, your algorithm should stop without exploring further.

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

- **Do not create any data structures (arrays, vectors, sets, maps, etc.).**
- Do not modify the tree's state in any way. Don't modify `root`, or a node's `data`, `left`, or `right` pointers.
- Do not **leak memory**. You should not be allocating dynamic memory or creating new node objects anyway.
- 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 may define **private helper** functions, but you may not call any other member functions of the tree class.
- Your solution must be **recursive**.

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

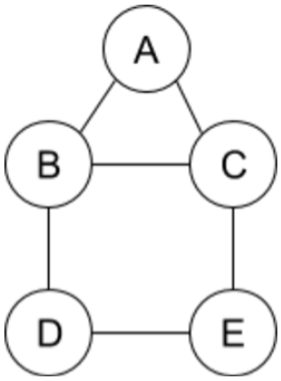
5. Graphs (write)

Write a function named **`findEulerPath`** that accepts as a parameter a reference to a `BasicGraph`, and tries to find an Euler path in the graph, returning it as a `Vector` of `Vertex` pointers. An *Euler path* is a valid path that uses every edge in the graph exactly once. *(A real-world example where you might want an Euler path: A snow plow wants to remove the snow from all the roads in the neighborhood without unnecessarily driving over the same road twice.)*

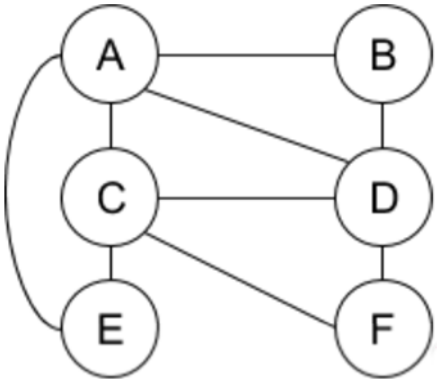
For example, in Graph 1 below at left, you can form an Euler path by starting at vertex C and forming the following path: {C, A, B, C, E, D, B}. If you follow the preceding path you'll find that it uses every single edge from the graph. It revisits some vertexes more than once, which is fine; it is only the edges we may not reuse. In Graph 2 below at right, you can form an Euler path by starting at vertex F and forming the following path: {F, C, D, A, C, E, A, B, D, F}. This second Euler path happens to be a cycle (also called an Euler circuit), which is fine, but the path you find does not need to be a cycle as long as it is a valid path that uses all the edges exactly once.

If the graph contains multiple valid Euler paths, you may return any one of them. Another valid example of an Euler path in Graph 1 below would be {B, D, E, C, A, B, C}. Another valid example of an Euler path in Graph 2 below would be {A, B, D, A, C, D, F, C, E, A}.

graph 1



graph 2



If the graph does not contain an Euler path, or if the graph does not contain any vertexes or edges, your function should return an empty vector. For example, in Graph 1 above at left, if the edge D-E were absent, then the graph would not contain an Euler path.

Implementation hint: This is a classic problem that does not have any known clever algorithm that will work in all cases. You need to perform an exhaustive search and explore all possible paths in the graph to see if you can find an Euler path through brute force. Although you must search exhaustively, you should still write code that is **efficient**. If you explore large numbers of the same paths and options multiple times, or continue exploring paths that are certain to be dead-ends, you may lose points.

You may assume that the graph is **undirected** and **unweighted**, and that it contains no self-edges (e.g. from *V1* to *V1*). Since the graph is undirected, calling `getEdge(v1, v2)` will return the same `Edge` object result as calling `graph.getEdge(v2, v1)` except that the edge will consider its `start` to be the first vertex and its `finish` or `end` to be the second vertex in each case. You may also assume that there is at most one edge from any vertex *V1* to any *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`.

6. Hashing (read)

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

```
HashMap map;
map.put(34, 2);
map.put(14, 99);
map.put(99, 5);
map.put(14, 8);
map.put(82, 59);
map.remove(34);
map.put(74, 18);
if (!map.containsKey(5)) {
   map.put(22, 66);
}
map.put(57, 75);
int x = map.get(14);
x--;
map.put(x, 555);
map.put(59, 888);
map.put(47, 74);
map.remove(75);
map.remove(map.get(82));
map.put(79, 0);
map.put(74, 222);
```

- 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**.

7. Inheritance and Polymorphism (read)

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

```
class Eliza : public Burr {
public:
    virtual void m2() {
        cout << "E2  ";
        Hamilton::m2();
    }

    virtual void m3() {
        Burr::m3();
        cout << "E3  ";
    }
};

class George : public Eliza {
public:
    virtual void m1() {
        cout << "G1  ";
        Burr::m1();
    }

    virtual void m4() {
        cout << "G4  ";
        m2();
    }
};

class Hamilton {
public:
    virtual void m1() {
        cout << "H1  ";
        m2();
    }

    virtual void m2() {
        cout << "H2  ";
    }
};

class Burr : public Hamilton {
public:
    virtual void m1() {
        Hamilton::m1();
        cout << "B1  ";
    }

    virtual void m3() {
        cout << "B3  ";
    }
};
```

Now assume that the following variables are defined:

```
Hamilton* var1 = new Burr();
Hamilton* var2 = new Eliza();
Burr* var3 = new Eliza();
Eliza* var4 = new George();
```

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();`	_____
`var2->m1();`	_____
`var2->m2();`	_____
`var2->m3();`	_____
`var3->m1();`	_____
`var3->m2();`	_____
`var3->m3();`	_____
`var4->m1();`	_____
`var4->m4();`	_____
`((Burr*) var1)->m3();`	_____
`((Eliza*) var2)->m4();`	_____
`((George*) var4)->m4();`	_____
`((Eliza*) var3)->m3();`	_____
`((George*) var2)->m4();`	_____

8. Inheritance / Array List Implementation (write)

In lecture, we discussed the implementation of a class called `ArrayList`, an implementation of a list of integers using an internal array. It was our own `Vector` of `int`s. The syntax reference sheet lists the methods of the `ArrayList` class.

Define a new class called `SortedList` that extends `ArrayList` through inheritance. Your class represents a list of integers that is always stored in sorted non-decreasing order, regardless of the order in which items are added to the list. Your class should provide the same member functions as the superclass, as well as the following new public behavior.

Constructor/member function	Description
`SortedList(bool unique)`	constructs a new sorted list; if the boolean value passed is `true`, the list will be a set that forbids duplicates
`virtual bool isUnique() const`	returns whether this list forbids duplicates, as per the boolean value that was passed to the constructor

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

- When a value is **added** to the list, it must be added in the proper place to maintain sorted order. For example, if the list is `{2, 5, 9}` and the value `4` is added, the list should become `{2, 4, 5, 9}`. This is the case whether the value is added through a call to `add` or a call to `insert`. This also means that the `insert` method ignores its index parameter and instead adds at the proper place to maintain sorted order.
- The **`set` method** does not make sense for a `SortedList` because it could cause the list to fall out of order. Therefore whenever set is called on a `SortedList`, it should throw a string exception.

The following two code samples demonstrate the general behavior of a `SortedList`. Notice that the example at right forbids duplicates from being added because it was constructed with the `bool` value `true`.

```
SortedList list1(false);
list1.add(5);         // {5}
list1.add(3);         // {3,5}
list1.add(4);         // {3,4,5}
list1.add(7);         // {3,4,5,7}
list1.insert(0, 5);   // {3,4,5,5,7}
list1.insert(2, 9);   // {3,4,5,5,7,9}
list1.add(8);         // {3,4,5,5,7,8,9}
list1.add(7);         // {3,4,5,5,7,7,8,9}
list1.add(1);         // {1,3,4,5,5,7,7,8,9}
list1.add(9);         // {1,3,4,5,5,7,7,8,9,9}
```

```
SortedList list2(true);        // unique
list2.add(5);         // {5}
list2.add(3);         // {3,5}
list2.add(4);         // {3,4,5}
list2.add(7);         // {3,4,5,7}
list2.insert(0, 5);   // {3,4,5,7}
list2.insert(2 ,9);   // {3,4,5,7,9}
list2.add(8);         // {3,4,5,7,8,9}
list2.add(7);         // {3,4,5,7,8,9}
list2.add(1);         // {1,3,4,5,7,8,9}
list2.add(9);         // {1,3,4,5,7,8,9}
```

Write the **.h** and **.cpp** parts of the class separately with a line between 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.

Recall that subclasses are **not** able to access private members of the superclass. Part of the challenge of this problem is properly implementing the expected behavior without illegally trying to access the superclass's private members.

You should not create any **auxiliary data structures** (arrays, vectors, queues, maps, sets, strings, etc.) in your code.