CS 106B, Summer 2016
Final Exam, Saturday, August 13, 2016

Your Name: ____________________________________________

Section Leader: ________________________________

Honor Code: I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I submit are my own work.

Signature: ____________________________________________ ← YOU MUST SIGN HERE!

Rules: (same as posted previously to class web site)

- This is an individual exam; you are to complete it yourself without assistance from others.
- You have 3 hours (180 minutes) to complete this exam.
- This test is open-book, but closed notes. You may not use any printed paper resources.
- You may not use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style. Some particular problems have style constraints or other code constraints, so read each problem carefully. We reserve the right to deduct points for solutions that are grossly inefficient or wasteful of resources.
- On code-writing problems, you do not need to write a complete program, nor include statements. Write only the code (function, etc.) specified in the problem statement.
- Please do not abbreviate code, such as writing ditto marks (""") or dot-dot-dot marks (...).
- Unless otherwise specified, you may define helper functions but you may not declare global variables.
- If you wrote your answer on a back page or attached paper, please label this clearly to the grader.
- Follow the Stanford Honor Code on this exam and correct/report anyone who does not do so.

Good luck! You can do it!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
<th>Earned</th>
<th>Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Searching and Sorting (read)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Linked Lists (write)</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Heaps (read)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Binary Search Trees (read)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Binary Trees (write)</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Graphs (write)</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Hashing (read)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>Inheritance/Polymorphism (read)</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>Inheritance / Array Lists (write)</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>TOTAL</td>
<td>Total Points</td>
<td></td>
<td>66</td>
</tr>
</tbody>
</table>

Copyright © Stanford University and Marty Stepp. Licensed under Creative Commons Attribution 2.5 License. All rights reserved.
1. Searching and Sorting (read)

(a) Suppose we are performing a binary search for the value 21 on a sorted vector containing the following elements:

```
// index     0   1   2   3   4   5   6   7   8   9  10  11  12  13
{-18, -3,  6, 11, 14, 19, 25, 29, 31, 34, 38, 62, 81, 96}
```

Write the indexes of the elements that would be examined by the binary search (the mid values in our algorithm's code) and write the value that would be returned from the search. Assume that we are using the binary search algorithm shown in lecture and section.

- Indexes examined: ____________________________________________
- Value Returned: ________________________________

(b) Suppose we are running a selection sort and a merge sort over a vector containing the following elements:

```
// index     0   1   2   3   4   5   6   7   8   9
{15, 56, 24,  5, 39, -4, 27, 10,  7, 31}
```

Show a full trace below of the selection sort and merge sort algorithms over the above vector.
2. Linked Lists (write)

Write a function `swapEnds` that accepts as its parameter a reference to a `ListNode` pointer representing the front of a linked list (see reference sheet). The function modifies the list by swapping the list's first element with its last element. For example, suppose a `ListNode` pointer variable named `front` points to the front of a list storing the following values:

\{11, 88, -7, 99, 22, 0, 4\}

After the call of `swapEnds(front);`, the list should store the following elements (changes are underlined):

\{4, 88, -7, 99, 22, 0, 11\}

If the list is empty or stores only a single element, it should be unchanged by a call to your function. Your function should work properly for a list of any size.

Note that the goal of this problem is to modify the list by modifying pointers. It might be easier to solve it in other ways, such as by changing nodes' data values or by rebuilding an entirely new list, but such tactics are forbidden.

**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 any 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 use any auxiliary data structures such as arrays, vectors, queues, maps, sets, strings, etc.
- Do not leak 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.

Write your answer on the next page.
2. Linked Lists (write)

Writing Space

(for grader only)
3. 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) 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.
(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.)

- Cindy, Anna, James, Lindsey, Daniel, Thomas, Ned, Steve, Victoria, Ryan, Brandon, Hubert

(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):

- Lindsey, Cindy, James
5. Binary Trees (write)

Write a function `trimSmallLeaves` that accepts as its parameter a reference to a `TreeNode` pointer representing the root of a binary tree (see reference sheet). Your function should remove all leaf nodes from the tree where the leaf’s value is smaller than its parent’s value. In other words, at the end of your function, it must be the case that all leaf nodes in the tree have values greater than or equal to their respective parent nodes.

For example, suppose a variable named `root` refers to the root of the tree below at left. If we then make the method call of `trimSmallLeaves(root);`, notice that several leaf nodes have been removed:

- 8, because it is smaller than its parent of 15;
- 11, because it is smaller than its parent of 15;
- 10, because it is smaller than its parent of 13.

But also notice that there are two other nodes that must be removed in this example: the 15 has become a leaf because its children of 8 and 11 were removed, and its value is smaller than its parent of 28, therefore the 15 is removed from the tree. Also, the 13 has become a leaf because its child of 10 was removed, and its value is smaller than its parent of 15, so it must be removed from the tree as well. Therefore we end up with the tree below at right.

<table>
<thead>
<tr>
<th>tree before call</th>
<th>after call of <code>trimSmallLeaves(root);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>28 / 13 \ 15 \ 17 \ 13 \ 8 \ 11 \ 19 \ 10</td>
<td>28 / 13 \ 15 \ 17 \ 19</td>
</tr>
</tbody>
</table>

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 change the data field of any existing nodes of the tree.
- Do not leak memory. If you remove a node from the tree, properly free its memory.
- 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 if you like.
- Your solution must be recursive.

Write your answer on the next page.
5. Binary Trees (write)
   Writing Space

(for grader only)
Write a function named \texttt{largestConnectedSubset} that accepts as a parameter a reference to a \texttt{BasicGraph}, and returns a \texttt{Set} of \texttt{Vertex} pointers representing the largest collection of vertexes that are all mutually reachable from one another. Two vertexes \texttt{v1} and \texttt{v2} are reachable from each other if there exists a path between them.

The diagram below shows an example graph that might be passed to your algorithm. The graph is divided into various clumps of vertexes that are connected by edges. The largest such clump is the one containing vertexes D, E, F, G, and H. So when passed the graph below, your function would return a \texttt{Set} containing pointers to those five vertexes.

If there is a tie and there are multiple clumps that contain exactly the same maximum number of vertexes, your function can return any one of those equal sets. If the graph were entirely connected, your function would return a set containing all vertexes in the graph. If the graph contained zero edges, your function would return a set containing any one vertex. If the graph contained no vertexes at all, your function should return an empty set.

You may assume that the graph's state is valid, and that it is \texttt{undirected} and \texttt{unweighted}, and that it contains no self-edges (e.g. from \texttt{V1} to \texttt{V1}). You may also assume that there is at most one edge from any vertex \texttt{V1} to any other vertex \texttt{V2}. You may define \texttt{private helper} functions if so desired, and you may construct auxiliary \texttt{collections} as needed to solve this problem. Do 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 \texttt{visited}, \texttt{cost}, and \texttt{color}.

\textit{Write your answer on the next page.}
6. Graphs (write)
   Writing Space
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 \( \geq 0.5 \) 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(34, 222);
map.put(14, 99);
map.put(99, 5);
map.put(14, 8);
map.put(82, 59);
map.remove(34);
map.put(74, 118);
if (!map.containsKey(5)) {
    map.put(22, 66);
}
map.put(57, 75);
map.put(59, 888);
map.put(47, 74);
map.remove(75);
map.remove(map.get(82));
int zz = map.get(14);
    zz++;
map.put(zz, 555);
map.put(79, 0);
map.put(74, 222);
```
Consider the following classes; assume that each is defined in its own file.

```cpp
class Mike : public Dustin {
public:
    virtual void m2() {
        cout << "M2 ";
        Eleven::m2();
    }
    virtual void m3() {
        Dustin::m3();
        cout << "M3 ";
    }
};
class Eleven {
public:
    virtual void m1() {
        cout << "E1 ";
        m2();
    }
    virtual void m2() {
        cout << "E2 ";
    }
};
class Lucas : public Mike {
public:
    virtual void m1() {
        cout << "L1 ";
        Dustin::m1();
    }
    virtual void m4() {
        cout << "L4 ";
        m2();
    }
};
class Dustin : public Eleven {
public:
    virtual void m1() {
        Eleven::m1();
        cout << "D1 ";
    }
    virtual void m3() {
        cout << "D3 ";
    }
};
```

Now assume that the following variables are defined:

```cpp
Eleven* var1 = new Dustin();
Eleven* var2 = new Mike();
Dustin* var3 = new Mike();
Mike* var4 = new Lucas();
```

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".

<table>
<thead>
<tr>
<th>Statement</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1-&gt;m1();</td>
<td></td>
</tr>
<tr>
<td>var1-&gt;m2();</td>
<td></td>
</tr>
<tr>
<td>var2-&gt;m1();</td>
<td></td>
</tr>
<tr>
<td>var2-&gt;m2();</td>
<td></td>
</tr>
<tr>
<td>var2-&gt;m3();</td>
<td></td>
</tr>
<tr>
<td>var3-&gt;m1();</td>
<td></td>
</tr>
<tr>
<td>var3-&gt;m2();</td>
<td></td>
</tr>
<tr>
<td>var3-&gt;m3();</td>
<td></td>
</tr>
<tr>
<td>var4-&gt;m1();</td>
<td></td>
</tr>
<tr>
<td>var4-&gt;m4();</td>
<td></td>
</tr>
<tr>
<td>((Dustin*) var1)-&gt;m3();</td>
<td></td>
</tr>
<tr>
<td>((Mike*) var2)-&gt;m4();</td>
<td></td>
</tr>
<tr>
<td>((Lucas*) var4)-&gt;m4();</td>
<td></td>
</tr>
<tr>
<td>((Mike*) var3)-&gt;m3();</td>
<td></td>
</tr>
<tr>
<td>((Lucas*) var2)-&gt;m4();</td>
<td></td>
</tr>
</tbody>
</table>
In lecture, we discussed the implementation of a class called \texttt{ArrayList}, an implementation of a list of integers using an internal array. It was our own \texttt{Vector} of \texttt{int}s. The syntax reference sheet lists the methods of the \texttt{ArrayList} class.

Define a new class called \texttt{StutterList} that extends \texttt{ArrayList} through inheritance. A "stutter" list is one where adding an element value actually causes multiple copies of that value to be added to the list. The stuttered list accepts a parameter to its constructor representing a "stuttering factor"; this represents how many copies of the element should be added each time. For example, a \texttt{StutterList(3)} adds three copies while a \texttt{StutterList(7)} adds seven copies.

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

<table>
<thead>
<tr>
<th>Constructor/member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{StutterList(int factor)}</td>
<td>constructs a new stuttered list; the integer passed should be used as the stuttering factor for that list</td>
</tr>
<tr>
<td>\texttt{virtual int getFactor()} const</td>
<td>returns this list's stuttering factor, the integer value that was passed to the constructor</td>
</tr>
</tbody>
</table>

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

- If 0 or a negative value is passed to your constructor, you should throw an integer \texttt{exception}.
- When a value is \texttt{added} or \texttt{inserted} to the list, the list should actually add \texttt{factor} number of copies of that value. For example, if the stutter factor is 3 and the value 4 is added, your code should actually add 4, 4, 4 to the list. This is the case whether the value is added through a call to \texttt{add} or a call to \texttt{insert}.

The following code sample demonstrates the general behavior of a \texttt{StutterList}. Note that the example list uses a stuttering factor of 3, but this is just an example; any integer can be passed as the stuttering factor.

```cpp
StutterList list(3);
list.add(5);         // {5, 5, 5}
list.add(8);         // {5, 5, 5, 8, 8, 8}
list.insert(1, 4);   // {5, 4, 4, 4, 5, 5, 8, 8, 8}
list.remove(6);      // {5, 4, 4, 4, 5, 5, 8, 8}
list.insert(0, 2);   // {2, 2, 2, 5, 4, 4, 4, 5, 5, 8, 8}
list.add(1);         // {2, 2, 2, 5, 4, 4, 4, 5, 5, 8, 8, 1, 1, 1}
```

Write the \texttt{.h} and \texttt{.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 \texttt{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 \texttt{auxiliary data structures} (arrays, vectors, queues, maps, sets, strings, etc.) in your code.

\textit{Write your answer on the next page.}
9. Inheritance / Array List Implementation (write)

Writing Space