University of Illinois at Urbana-Champaign
Department of Computer Science

Final Examination

CS 225 Data Structures and Software Principles
Sample Exam 1
3 hours permitted

Print your name, netID, and lab section day/time neatly in the space provided below; print your name at the upper right corner of every page.

Name:
NetID:
Lab Section (Day/Time):

- This is a **closed book** and **closed notes** exam. In addition, you are not allowed to use any electronic aides of any kind.
- You should have 12 sheets total (the cover sheet, plus numbered pages 1-22). The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave. The back of the page before the scratch paper contains the interface to the `Array` and `pair` classes, which you might find useful.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.

Problem	Points	Score	Grader
1	16		
2	40		
3	64		
4	15		
5	15		
6	15		
7	15		
Total	180		

1. **[C++ – 16 points].**

   (a) You have a hash table class declared as follows:

```
class HashTable {
private:
   Array<pair<int, int> > table;
public:
   // public function declarations
};
```

   The `Array` will be indexed from `0` through `table.Size() - 1`. The first value in each pair holds `0` if the cell is empty, `1` if the cell is valid, and `-1` if the cell was deleted. The second value in each pair holds the key value in this cell (if the cell is a valid cell). Write the assignment operator for this class. (It's okay to just write the code that would appear in the .cpp file; that is, you can just write the definition and assume the declaration occured in the "`public function declarations`" above section.

(b) Consider the following code:

```
Foo theObject;
cout << theObject(6, 9) << endl;
```

Write a class `Foo` – specifically, write a function object class – so that in the code above, `theObject(6, 9)` produces: the sum of the two arguments. In general, the function call `theObject(x, y)` should return the sum of `x` and `y`.

2. **[Analysis – 40 points].**

   (a) You have an red-black tree composed of `n` nodes that each have *three* pointers. Each node points not only to its two children, but to its parent as well. (The root's `parent` pointer, points to `NULL`, since the root does not have a parent.

   You have a pointer to the node in the red-black tree containing the minimum value in the tree. You want to travel from the minimum node to the root of the tree, printing each value you encounter along the way, and then travel from the root to the node containing the AVL tree's maximum value, again printing each value you encounter along the way. Express the order of growth of the worst-case running time of this procedure, in Big-$\mathcal{O}$ notation, and explain convincingly why your answer is correct.

(b) The algorithm known as *heapsort* was briefly mentioned in lecture once. The algorithm sorts `n` integers by putting them all into a heap, and then calling `DeleteMin` repeatedly, thus pulling the integers out of the heap, in order from lowest to highest. Express the order of growth of the worst-case running time of *heapsort*, in Big-$\mathcal{O}$ notation, and explain convincingly why your answer is correct.

(c) You have a graph where the vertices represent U.S. cities and the edges represent roads that run directly between a given two cities. The graph is stored as an adjacency matrix indexed from `0` through `V - 1`, where `V` is the number of cities in the graph and `E` is the number of road segments between cities. You have a hash table containing the `n` cities – to get the index for a city, you hash on the city's name, and the hash table returns an information record associated with that name, which includes the adjacency matrix index for that city. Given all of that, what you want to do is to take two city names, and determine if they are directly connected by a road segment. Express the order of growth of the average-case running time of this procedure, in Big-$\mathcal{O}$ notation, and explain convincingly why your answer is correct.

(d) You have an ordinary trie, holding `n` different words, all of which are no longer than `L` characters (counting the null character at the end of a word). You want to perform `F` find operations on the trie. Express the order of growth of the worst-case running time of this procedure (that procedure being, "perform `F` find operations"), in Big-$\mathcal{O}$ notation, and explain convincingly why your answer is correct.

3. **[Algorithms – 64 points].**

   (a) Given a singly-linked list, and a pointer `current` to a node in the singly-linked list, explain convincingly how you can implement `InsertBefore` in constant time.

   (b) Imagine you have a sparse array with R rows and C columns, implemented by a "list of arrays". That is, there's a list, and for every row that has at least one non-default value, this list has a node for that row, containing the index of the row, and an array of every possible column in that row. If every row in the sparse array has at least one value, do we save any memory with this approach? Explain why or why not.

(c) Explain convincingly that a single rotation on a binary search tree, results in a tree that is still a binary search tree.

(d) Explain convincingly that for a red-black removal, case 2a works. That is, when the node or "`NULL`" which you've labelled "x" has a black sibling, and of that sibling's two children, the one furthest from "x" is black and the one closest to "x" is red, explain what is done in that case, and why, and explain why we still have a legal red-black tree after this case.

(e) We looked at a depth-first-search-based algorithm for finding the topological sort of a graph. Explain what this algorithm was and why it worked.

(f) Explain the changes made to Dijkstra's algorithn to obtain an implementation of Prim's Algorithm, and explain why once those changes are in place, we do indeed have an implementation of Prim's algorithm.

(g) Consider a hash table where $h(x) = x \bmod 11$ and $h_2(x) = 7 - x \bmod 7$. Insert the values 56, 27, 38, 25, 78, 50, and 45, in that order, into the hash table. Use double hashing to resolve collisions.

	56	27	38	25	78	50	45
$h(x)$	1	5	5	3	1	6	1
$h_2(x)$	7	1	4	3	6	6	4

0	1	2	3	4	5	6	7	8	9	10

(h) If you were to insert the words cot, car, carpet, cow, and coworker into a Patricia tree, not counting the leaves, how many nodes would there be? Explain your answer.

4. **[List Reversal – 15 points].**

   You have the following `ListNode` class:

```
class ListNode {
public:
   int element;
   ListNode* next;
   ListNode* prev;
};
```

   and a doubly-linked list made up of such nodes, with a `ListNode` pointer `head` to the first node and with the first node's `prev` and the last node's `next` equalling `NULL`. We will assume it is publicly accessible, rather than nested in a class, for this problem.

   Write a function `ReverseParts` which has two parameter and returns nothing. The first parameter will be a reference to a `ListNode` pointer. This pointer will point to the head node of a doubly-linked list (and thus would be `NULL` if the list were empty). This list will have the `prev` of the first node and the `next` of the last node both pointing to `NULL`.

   The second parameter will be a positive integer, which we will call n. This function should reverse every set of n values. For example, if the list had been `5->3->1->10-14->22->67->4->NULL` and `n` were `3`, then the resultant list should be `1->3->5->22->14->10->4->67->NULL` (the set of two values at the end of the list likewise have their order reversed). You can assume that `n` will be less than or equal to the list size. Whatever linked list this results in, the `head` parameter should be pointing to the first node of that list when you are done.

```
void ReverseParts(ListNode*& head) {
   // your code goes here
```

(List Reversal, continued)

5. **[Comparison check – 15 points].**

   Given the following class:

```
class TreeNode {
public:
   int element;
   TreeNode* left;
   TreeNode* right;
};
```

   You have a tree built from the above node type. This tree is NOT a Binary Search tree, though it *is* a binary tree, by definition. You want a function `Greater` that takes two parameters. The first will be a pointer to a `TreeNode`. The second should be an integer value. The function should return an array of all values greater than the parameter integer. The `Array<int>` you return needs to be *exactly* the right size for the values it holds. You do not need to keep the array sorted in any particular order. (In this problem, you are allowed to create additional 1-dimensional arrays – you have to, since you are returning one from your function.)

```
Array<int> Greater(TreeNode* ptr, int value) {
   // your code goes here
```

(Verifier, continued)

6. **[Complete Subgraph – 15 points].**

   You have the following `EdgeNode` class:

```
class EdgeNode {
public:
   int index;  // index of target vertex
   EdgeNode* next;  // ptr to next edge
};
```

   Furthermore, you have a variable of type `Array<EdgeNode*>` that is indexed from `1` to the size of the array (given by the `Size()` method in the `Array` class). The array is an adjacency list implementation of a graph, of the kind we first discussed in lecture; the vertices have indices from `1` to `Size()`. You can assume that each edge list in this adjacency list implementation is sorted by target index.

   Recall that a *complete graph* is a graph in which each vertex has every other vertex as neighbors (you can assume no vertex can ever have an edge to itself, whether the graph is defined "complete" or not). You want to write a method `isSubgraphComplete` which has two parameters. The first is a reference to an `Array<EdgeNode*>` as described above. The second is a sorted array of integers, indexed from `1` to the size of that array. Each integer being stored as a value in this array, is a legal index for a cell of the adjacency list (i.e. each integer is a vertex number).

   You want to return `1` if the graph would be complete once you removed the vertices in the second array. For example, if the vertices in the second array were `2`, `3`, and `8`, then you are returning `1` if and only if your graph would be complete once you remove vertices `2`, `3`, and `8` from the graph. (Note that you do not actually have to remove those vertices from the graph!! We're only asking what would happen if you did.)

```
int isSubgraphComplete(Array<EdgeNode*> graph, Array<int> ignored) {
   // your code goes here
```

(Complete Subgraph, continued)

7. **[Trie – 15 points].**

   You have a de la Briandais tree that stores strings made of lowercase letters. Assume that it does NOT use the Patricia tree optimization, and that it is composed of individual nodes of the following list node type:

```
class DeLaNode {
public:
   char letter;
   DeLaNode* subtree;
   DeLaNode* nextInList;
};
```

   To indicate leaves, we just set the null-character cell in that level's array, to point to `leaf` (which we'll assume is a variable we've declared elsewhere).

   You want to write a method `numPrefixes` that has three parameters. The first is a pointer to a `DeLaNode`. The second is a positive integer `n`. The third is an integer holding the depth of this node. You want to return the number of unique prefixes of exactly length `n` that are represented in the trie. For example, if you had already inserted "car", "carpet", and "cotton", then you have two unique prefixes of length three – namely, "car", and "cot".

```
int numPrefixes(DeLaNode* ptr, int n, int level) {
   // your code goes here
```

(Number of Prefixes, continued)

```
class Array:
   Array();   // creates array of size 0
   Array(int low, int hi);   // creates array with index range (low, hi)
   Array(const Array& origVal);  // copy constructor
   ~Array();          // destructor
   const Array& operator=(const Array& origVal);  // assignment operator
   const Etype& operator[](int index) const;
   Etype& operator[](int index);
   void Initialize(Etype initElement);
   void SetBounds(int low, int hi);  // changes bounds of array
   int Size() const;     // returns number of indices in index range
   int Lower() const;    // returns lower bound of index range
   int Upper() const;


template <class Type1, class Type2>
class pair
{
public:

   // ************ Member functions

   // pair
   //    - default constructor
   //    - sets object to hold default values
   //        (on the initializer line, each member variable
   //         is being initialized to the default value
   //         of the appropriate type)
   pair();


   // ************* Member data
   //    - we intend for the data to be public, too

   Type1 first;       // variable of the first type
   Type2 second;      // variable of the second type
};
```

(scratch paper)