University of Illinois at Urbana-Champaign
Department of Computer Science

Final Examination

CS 225 Data Structures and Software Principles
Sample Exam 2
3 hours permitted

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

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 11 sheets total (the cover sheet, plus numbered pages 1-21). 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.
- 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 functions 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	60		
2	40		
3	20		
4	20		
5	20		
6	20		
Total	180		

1. **[Algorithms - 60 points (6 points each)].**

   (a) Explain how the middle node of a singly-linked list can be removed in constant time, given a pointer to that node.

   (b) In a "perfect" skiplist, we expect to never have to traverse forward more than one node on any level. Why is this?

(c) In a perfect binary tree that is also a binary search tree, where is the median value of the tree located? Justify your answer.

(d) You insert the values `1`, `6`, `2`, `5`, `3`, and `4`, in that order, into an empty red-black tree. Counting a "double rotation" as one rotation, how many total rotations are needed over the course of these six insertions?

(e) How do we "mark a vertex known" in the heap implementation of Dijkstra's Algorithm – that is, how do we ensure that a vertex we have selected in one step will never again be selected in a future step – and how long does this process take, for one vertex?

(f) Given a graph that is both undirected, and connected (i.e. given any two vertices, there is a path between them in the graph), can breadth-first-search ever produce a spanning forest instead of a spanning tree? Justify your answer.

(g) Consider a hash table where $h(x) = x \bmod 11$ and $h_2(x) = 5 - x \bmod 5$. Insert the values 80, 58, 30, 5, 21, and 27, in that order, into the hash table. Use double hashing to resolve collisions. We have provided the relevant values of the two hash functions, in the table below.

	80	58	30	5	21	27
$h(x)$	3	3	8	5	10	5
$h_2(x)$	5	2	5	5	4	3

0	1	2	3	4	5	6	7	8	9	10

(h) How many "array nodes" (as opposed to leaves containing info records for our keys, rather than arrays) would there be in a Patricia Tree containing the words `cart`, `car`, `carthage`, `cardio`, and `cartoon`?

(i) Explain convincingly that a red-black tree removal will never require more than three single rotations. (You can assume anything we said in class about the individual cases of the algorithm is true; for example, if some case involves one rotation, you can simply state that such a case involves one rotation, without justifying it.)

(j) For the given graph, run Prim's algorithm, indicating in the table below the distances at each vertex at the end of each step (d_v), and whether or not the vertex has been marked known yet at the end of each step (k_v).

	A	B	C	D	E	F	G
A	0	7	8	9	0	0	0
B	7	0	0	5	1	0	0
C	8	0	0	4	0	6	0
D	9	5	4	0	2	3	11
E	0	1	0	2	0	0	12
F	0	0	6	3	0	0	10
G	0	0	0	11	12	10	0

V	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v
A	∞	0														
B	0	0														
C	∞	0														
D	∞	0														
E	∞	0														
F	∞	0														
G	∞	0														
-	Start		Step 1		Step 2		Step 3		Step 4		Step 5		Step 6		Step 7	

2. **[Analysis – 40 points (10 points each)].**

   (a) You have a pointer `head` to a doubly-linked list of `n` nodes, made up of nodes of the `ListNode` class on page 19. You need to do three things: (1) explain what the following code does, (2) tell us what the the order of growth of the worst-case running time of the following code is (in big-$\mathcal{O}$ notation), and (3) explain your answer to (2) in enough detail to be convincing.

```
// you are given the pointer ‘‘head’’ to a list of size n
ListNode* temp = NULL;
ListNode* latestHead = NULL;
while (head != NULL) {
   ListNode* toEnd = head;
   while (toEnd->next != NULL)
      toEnd = toEnd->next;
   if (toEnd->prev != NULL)
      toEnd->prev->next = NULL;
   else
      head = NULL;
   toEnd->prev = temp;
   if (temp != NULL)
      temp->next = toEnd;
   else
      latestHead = toEnd;
   temp = toEnd;
}
```

(b) Imagine you have a red-black tree, and at each node of this red-black tree, you have an array and an integer. The red-black-tree is arranged by the integer in each node (i.e. that integer is the search key for the tree, i.e. an in-order traversal of the red-black-tree would give you the integers in numerical order). You look up a pair of numbers in this structure, by finding the first number of the pair in the red-black tree, and then once you have found that number in some red-black tree node, you search the array at that node for the second number in the pair. If you have `n` nodes in the tree, and each array is of size `m`, what is the order of growth of the worst-case running time of the above procedure? Express your answer in big-$\mathcal{O}$ notation, and explain your answer in enough detail to be convincing.

(c) Suppose you want to add a vertex to graph of `V` vertices and `E` edges. Assume that this graph is implemented using an adjacency matrix of exactly the right size needed for the graph. Express (using big-$\mathcal{O}$ notation) the order of growth of the worst-case running time of this operation, in terms of `V` and `E`. Explain your answer in enough detail to be convincing.

(d) Explain why the time to lookup a value in a trie does not depend on how many values there are in the trie.

3. **[Range Removal - 20 points].**

   You have the `ListNode` class seen on page `19` of this exam. You want to write a function `RangeRemoval` that will have three parameters. The first is a reference to a pointer to the starting node of a doubly-linked list made up of the above nodes. The `next` variable of the last node and the `prev` variable of the first node are both `NULL`, and the list is sorted from lowest to highest `element`. The second and third parameters are a low and high value defining a range of integers (you can assume the second parameter is less than or equal to the third parameter).

   Your task is to remove all values in the list, whose elements are between the second and third parameters, inclusive, from the parameter list, and to return a sorted list of those removed values. For example, if the list were of size `12` and was as follows:

   ```
   head->0->1->2->4->5->7->8->9->11->12->15->17->NULL
   ```

   and your second and third parameters were `6` and `12`, respectively, then the parameter list becomes:

   ```
   head->0->1->2->4->5->15->17->NULL
   ```

   and you return a pointer to the start of the list:

   ```
    ---->7->8->9->11->12->NULL
   ```

   where the first node's `prev` and the last node's `next` variables are both `NULL` and the values are sorted.

   You are NOT allowed to write to the `element` variable of any of the nodes; you must complete this method by rearranging the nodes themselves. When you are done, the parameter list should still be doubly-linked, the `next` variable of the first node and the `prev` variable of the last node should both point to `NULL`, and the parameter should point to the new front node of the list.

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

(Range Removal, continued)

4. **[Array of Lists - 20 points].**

   You have the use of the `Array`, `ListNode`, and `TreeNode` classes seen on page `19`, as well as a standard `Queue` class with a no-argument constructor that initializes the `Queue` to be empty.

   You want to write a function `levels` that has one parameter, a pointer to `TreeNode`, which points to the root of a binary tree made of `TreeNode` objects. The function should return an `Array` object indexed from `0` to `maxDepth`, where `maxDepth` is the depth of the deepest leaf in the tree. For every index `i` between `0` and `maxDepth`, inclusive, the array cell at index `i` should point to a linked list containing every node of depth `i` in the binary tree, in order from left-to-right. For example, cell `0` of the array would point to a list containing the tree's root node, cell `1` of the array would point to a list containing the root's left and right children, in that order, and so on.

   You are *also* allowed to use one standard `Queue`, which has a no-argument constructor that initializes the `Queue` to be empty.

```
Array<ListNode*> levels(TreeNode* ptr)
   // your code goes here
```

(Array of Lists, continued)

5. **[Dijkstra's Algorithm - 20 points].**

   You have the following classes:

```
class EdgeNode {
public:
   int target;  // index of target vertex
   int weight;  // weight of edge
   EdgeNode* next;  // next node in list
};
```

```
class VertexRecord {
public:
   int distance;
   int known;
   EdgeNode* edgePtr;
};
```

   We can implement graphs using the adjacency list implementation, by having a variable `theGraph` of type `Array<VertexRecord>` that is indexed from `1` to `theGraph.size()`. In this graph, the vertices have indices from `1` to `theGraph.size()`, and all edge weights are positive.

   Your task is to run Dijkstra's algorithm on such a graph. That is, want to write a function `dijkstra` that has two parameters. The first is a reference to a variable `theGraph` as described above, and the second is an integer between `1` and `theGraph.size()` inclusive, which is the index of the source vertex. The `distance` and `known` variables in each `VertexRecord` object are initially not initialized in any way, so you will need to take care of that yourself. You may use those two variables however you see fit, but at the end of your function, each `VertexRecord` should store the minimum distance from the source to that vertex, in the `distance` variable for that vertex. You should use the "table implementation" to implement this algorithm.

```
void dijkstra(Array<VertexRecord> & theGraph, int source) {
   // your code goes here
```

(Dijkstra's Algorithm, continued)

6. **[Optimizing Tries - 20 points].**

   You have the following node class:

```
class TrieNode {
public:
   bool isLeaf;     // true if this is a leaf node; else false,
                    // in which case this node is part of some linked list
   char element;
   TrieNode* subtree;
   TrieNode* next;
};
```

   The above node class is used to implement a de la Briandais tree; each linked list in the tree is composed of the above type of node.

   We will assume that a list of `TrieNode` objects of length 12, takes up less space than the corresponding array of size `27`, but that if you had a list of size `13`, that would take up *more* space than the corresponding array would have needed. Given that assumption, you want to write the method `convertRatio`, which has one parameter, a pointer to the above type of node. That pointer will be a pointer to either a leaf node, `NULL`, or the starting node of some linked list in the de la Briandais tree. You want to return a `pair<int, int>` value, where the first value is how many array-nodes there would be in the tree rooted at `ptr`, if this were a regular trie (i.e. how many linked lists you have in your de la Briandais tree rooted at `ptr`), and the second value is how many of those would use less space if implemented as arrays rather than lists, based on our assumption above. (You can find the `pair` class on page 19 of the exam.)

```
pair<int, int> convertRatio(TrieNode* ptr) {
   // your code goes here
```

(Optimizing Tries, continued)

These nodes and classes are used on the exam:

```
template <typename Etype>
class Array {
   Array();                          // creates array of size 0, with
                                     // lower bound == 0 and upper bound == -1

   void setBounds(theLo, theHigh);   // resizes array to have lower bound
                                     // theLow and upper bound theHigh; any
                                     // values in that index range already,
                                     // remain in the resized array

   int size();                    // returns the number of values

   Etype& operator[](int index)  // returns the cell at parameter index
   // there are other member functions but
   // they are not used on this exam
};

template <typename Etype1, typename Etype2>
class pair {
public:
   Etype1 first;
   Etype2 second;
};


class ListNode {
public:
   int element;
   ListNode* next;
   ListNode* prev;
   ListNode(int elem) {element = elem; next = NULL; prev = NULL;}
};

class TreeNode {
public:
  int element;
  TreeNode* left;
  TreeNode* right;
  TreeNode* parent;
  TreeNode(int elem) {element = elem; left = NULL; right = NULL; parent = NULL; }
};
```

(scratch paper, side 1)

(scratch paper, side 2)