CSCI 104 Practice Final Exam —

Your Name, USC username, Class time, and Student ID:

Question	Points possible	Points
1	16	
2	12	
3	10	
4	12	
5	10	
6	12	
7	12	
8	16	
Total	100	

Do not open the exam until instructed to do so!

Turn off all cell phones!

This packet has 9 pages (including this cover page) and 8 questions. If yours does not, please contact us immediately.

This exam is closed book. You are allowed one 8.5×11 inch note sheet (front and back). You will have 110 minutes to work on this exam.

Please read and reread each question carefully before trying to solve it. In your solutions, you should try hard to write mostly correct C++. Minor syntax errors will (at most) lose small numbers of points, but you should also demonstrate a mastery of most C++ syntax. And of course, you should make sure to not have any memory leaks.

GOOD LUCK

(1) [4+4+4+4=16 points]
For each of the following tasks, describe a data structure that would be particularly well-suited to the task. Give a brief explanation for why this data structure is appropriate. If you need to modify the data structure for your task, explain what modifications you are making and why.

(a) A data structure for handling an emergency room. You will want to (very quickly) identify the patient who has the most life-threatening injury. You will also need to quickly add patients to the waiting list.

(b) An air-traffic controller which keeps track of which landing strip each airplane is currently on. Airplanes are identified by a string of 6 alphanumeric characters, and the operations require very strong *guarantees* on the worst-case running time.

(c) A database of PGP cryptosystem public keys currently in use in a system. Each public key is a string of 400 characters. When users register, they choose a public key. If two users had the same key, this would be a security hazard, so the system should reject a proposed key if it is already in use by someone else. In that case, the user will be forced to choose a different public key.

Your system should be able to support up to 500 million users. Unfortunately, you only have about 2GB (that is, 2 billion bytes) of main memory.

Clearly specify anything you are compromising on to satisfy the above requirements.

(d) (4 pts) A templated `map` which implements `insert, remove,` and `find`. Because your `map` will be used in time-critical applications (TOP-SECRET clearance issues prevent us from telling you precisely what applications), it must have the following guarantees:

(a) The *average-case* running time for each of the three operations must be $O(1)$.

(b) The *worst-case* running time for each of the three operations must be $O(\log n)$.

(2) [12 points]
2-3 Heaps are a significantly more complicated implementation of Heaps (Priority Queues), with the following running time guarantees (where n is the number of items in the Priority Queue):

- `void add(T t)` has running time $O(\log n)$.
- `void remove (T t)` has running time $O(\log n)$.
- `T peek ()` has running time $O(\log n)$
- `void update (T t, int new-priority)` has running time $O(1)$.

Re-analyze the running time of Dijkstra's Shortest Paths Algorithm when using a 2-3 Heap instead of a regular heap. Express the running time as a worst-case time in terms of m (the number of edges) and n (the number of nodes). You don't need to give code (or even pseudo-code) so long as it is clear how the analysis works. If it helps your analysis, you can of course give pseudo-code; we recommend against giving full C++ code for Dijkstra's Algorithm, as it would likely be fairly long.

(3) [10 points]
Using the provided FindMin function, implement selection sort on a linked list. Recall that selection sort first places the smallest item at the front, then the 2nd smallest after that, until the array is sorted. Your function must run in $O(n^2)$ time.

```
struct Item {
 public:
  int getValue();
  Item *next, *prev;
 private:
  int value;
};

//Returns a pointer (in O(n) time) to the smallest item in the Linked List starting at head,
//or NULL if head is NULL.
Item* FindMin(Item *head);

//Returns a pointer to the head of the sorted linked list.
Item* LL_Selection_Sort(Item *head) {
```

(4) [12 points]

You want a data structure supporting the following three operations, running in time $O(\log n)$ (where n as always is the number of items in the data structure):

```
void add (Key k); // adds the key k
void remove (Key k); // removes the key k
unsigned int count (Key k1, Key k2);
// returns how many keys k have been inserted that lie between k1 and k2, inclusive.
// Assumes that Key overloads the comparison operators <, <=, etc.
```

Outline a data structure that implements these three operations. You do not need to (and probably don't want to) write code. You may start from any data structure covered in class, and explain how you would expand/change it. Your analysis should explain why it achieves the running time guarantees that you claim it has.

(5) [10 points]
You put k keys into a hash table with m hash buckets. We assume that the hash function is good enough that you can treat the destination of each key as independently uniformly random. What is the probability that there are no collisions at all, i.e., that all keys end up in different positions of the array? Show your work as you derive your answer.

(6) [12 points]
We have a Bloom Filter with an array of 10 elements (the elements of the set are integers), and using three hash functions

$$\begin{aligned} h_1(x) &= (7x+4) \bmod 10, \\ h_2(x) &= (2x+1) \bmod 10, \\ h_3(x) &= (5x+3) \bmod 10. \end{aligned}$$

We execute the sequence of operations given below. What does the program output? Which of the answers are false positives (the Bloom filter says "Yes", even though the correct answer is "No")? Which are false negatives (the Bloom filter says "No", even though the correct answer is "Yes")? If your final answer is incorrect, you may get more partial credit if you show enough work for us to isolate the mistake.

```
BloomFilterSet<int> bf (10);
bf.add (0);
bf.add (1);
bf.add (2);
bf.add (8);
// Show us what the Bloom Filter's Array looks like at this point.
if (bf.contains (2)) std::cout << "2\n";
if (bf.contains (3)) std::cout << "3\n";
if (bf.contains (4)) std::cout << "4\n";
if (bf.contains (9)) std::cout << "9\n";
```

(7) [12 points]
You have a correctly implemented `Max-Heap` class (indices starting at 1), which provides the two following helper functions (in addition to `insert`, `remove`, `peek`):

(a) `bubbleUp (int i)` takes the element at position i and keeps swapping it with its parent (and grandparent etc.) until it is no larger than the parent at the time.

(b) `trickleDown (int i)` takes the element at position i and keeps swapping it with the larger of its children (and grandchildren etc.) until it is no smaller than both children.

Suppose that you get an instance whose array elements are not in correct heap order, i.e., the heap property does not hold yet. You want to establish the correct arrangement, by using the two helper functions given above. Which of the following two alternatives do this correctly? Explain why they are correct or incorrect. For incorrect ones, give an example of an initial heap which is not correctly processed, and explain what happens when the incorrect variant is executed on it.

```
Heap h;

// Variant 1
for (int i = 1; i <= n; i ++) h.bubbleUp (i);

// Variant 2;
for (int i = 1; i <= n; i ++) h.trickleDown (i);
```

(8) [16 points]

In class, we saw Tries and how to insert and look up words in them. Here, you will implement deletion from a trie, changing the trie as though the string had never been added. We will assume that our strings only consist of lower-case letters.

A friend has already written the following piece of code. She is only sharing the header file with you, but since she's a Trojan, you're sure that it's all correctly implemented.

```
struct TrieNode {
  bool isInSet;        // true if there is a word ending at this node.
  TrieNode *parent;    // the parent in the trie, NULL if the node is the root.
  TrieNode *child[26]; // the (up to) 26 children for the letters. NULL if there is no such child.

  int convert (char letter) // turns a letter into its index in the child array
    { return (int) (letter - 'a'); }
  bool isLeaf () { // returns whether the node is a leaf
    for (int i = 0; i < 26; i ++)
        if (child[i] != NULL) return false;
    return true;
  }
};

class InsertOnlyTrie {
  protected:
    TrieNode *root;

  public:
    InsertOnlyTrie () { root = NULL; }
    ~InsertOnlyTrie (); // correctly implemented destructor

    void add (string s);
      // adds the string to the trie correctly. Is already implemented.

    TrieNode *lookup (string s);
      /* returns the pointer to the node at which the search for s terminates in the trie.
         Is already implemented. If s is not in the trie, returns NULL. */
};
```

On the next page, you should use your friend's class to build a "real" `Trie` class that also provides the following public function.

```
void remove (string s);
      /* If s is not in the trie, the function should not do anything.
         Otherwise, it should remove s from the trie as though it had never been added. */
```

Remember that you cannot alter your friend's code, so you must find a suitable way to build the `Trie` class. For full credit, your solution must not leak any memory.

```
// your code goes here
```