CSCI 104 Practice Midterm Exam 2 —

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

Question	Points possible	Points
1	10	
2	15	
3	15	
4	15	
5	15	
6	15	
7	15	
Total	100	

Do not open the exam until instructed to do so!

Turn off all cell phones!

This packet has 11 pages (including this cover page) and 7 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

General Notes on Data Structures

You are explicitly allowed and encouraged to draw on and use the following data structures/algoirthms.

- All data structures provide the following two constant-time functions:

  ```
  bool empty () const; // returns whether the structure is empty
  unsigned int size() const; // returns the number of elements currently stored in the data structure
  ```

- Maps, referenced as $map < K, V >$. The functions you can assume are the following.

  ```
  void insert (const K & key, const V & value); // does nothing if key is already in the map
  void remove (const K & key); // does nothing if key is not in the map
  V & operator[] (const K & key); /* if key was not in the map, adds it to the map
                                     with a garbage value assigned to it. */
  ```

  All operations take $\Theta(\log n)$ time.

- Vectors, referenced as $vector < T >$. The functions you may assume and their running times are as follows:

  ```
  V & operator[] (const T & val); \\ Takes O(1) worst-case time.
  void push_back (const T & item); \\ Takes O(1) amortized runtime.
  ```

- Stacks, referenced as $stack < T >$. The functions you may assume are the following, all running in time $O(1)$.

  ```
  void push (const T & item);
  void pop ();
  const T & top () const;
  ```

- Queues, referenced as $queue < T >$. The functions you may assume are the following, all running in time $O(1)$.

  ```
  void enqueue (const T & item);
  void dequeue ();
  const T & peekFront () const;
  ```

- If you want to sort a vector, you may just call a function

  ```
  void sort (vector<T> & data);
  ```

  This runs in time $\Theta(n \log n)$

- Of course, you are also allowed to use arrays (static or dynamically allocated, as you see fit), and define your own structs or classes if it helps you.

  If you need these data structures or algorithms — either as member variables or in any form of inheritance — you may just use them without worrying about **#include** or how they are implemented themselves, apart from the information we gave you above.

(1) [10 points]

When you learned about A^* search, it was emphasized that the distance heuristic h must be an *underestimating* heuristic. Show us why, by giving an example of a graph with non-negative edge lengths, and some heuristic h that you get to design (that is not underestimating), such that A^* search does not find the shortest s-t path when using your heuristic h. Explain what the shortest path should be, and what A^* search finds and why.

(2) [5+10=15 points]

Here is the code for a base-b counter. (For this entire problem, you get to assume that $b \geq 2$.)

```
class Counter {
  private:
    int n;
    int b;
    int *p;
  public:
    Counter (int b, int n) {
        this->n = n;   this->b = b;
        p = new int [n];
        for (int i = 0; i < n; i ++) p[i] = 0;
    }
    void increment () {
        int i;
        for (i = 0; i < n && p[i] == b-1; i ++)
            p[i] = 0;
        p[i] ++;
    }
}
int main() {
  int n;
  cin >> n;
  Counter c(2, n);
  for (int i = 0; i < pow(2, n); i++) c.increment();
  return 0;
}
```

(a) Analyze the **worst**-case runtime of `increment()` in terms of `n` using Θ-notation, and explain your answer.

(b) Analyze the **worst**-case runtime of `main()` in terms of `n` using Θ-notation, and explain your answer. **Hint:** figure out how much work is spent changing p[0], p[1], etc, and sum these values together.

(3) [5+10=15 points]

Brutus the Bruin has implemented the following lazy version of Quicksort:

```
void Quicksort (T a[], int l, int r) {
    if (l < r) {
        int m = median-partition(a,l,r);
        Quicksort (a, l, m-1);
    }
}
```

`median-partition(a,l,r)` runs in linear time $\Theta(r-l)$ and finds the *actual median* of the array between l and r, then uses it as a pivot to partition the array between l and r. Notice that Brutus unfortunately only recursively sorts the left subarray, and is forgetting about the right part. So the algorithm isn't particularly useful. Nonetheless, you will analyze here how long it takes.

(a) Set up a recurrence relation for the running time, and describe where the terms came from.

(b) Derive a solution to the recurrence relation using a method of your choice. Show your work.

5

(4) [15 points]

You are given two linked lists u and v as input, and you want to merge them into a single linked list (which will be returned). Specifically, the first node in u should go first, then the first node in v, then the second node in u, etc. If at any point one of the linked lists is empty, append the rest of the other list to the end of your output.

For example:

Sequence u	Sequence v	Output
1 2 3	2 4 6	1 2 2 4 3 6
4 2	4 3 2 1	4 4 2 3 2 1
1 2 3 4	5 6	1 5 2 6 3 4

You do not need to maintain the original lists u and v. That is, as long as you return the correct list from merge, lists u and v do not need to still be in the same state as at the start of the function.

Your algorithm **must** be recursive. At no point may you use the keywords `for`, `while`, `goto`, or `static`.

```
struct Node {
  int value;
  Node* next;
};
Node* merge(Node *u, Node *v) {
```

```
}
```

(5) [15 points]

Imagine that you are writing a simple game for young children to learn to recognize and match shapes. For our purposes, shapes are restricted to squares and circles. Shapes arrive over time and fall on a pile of items. See Figure 1.
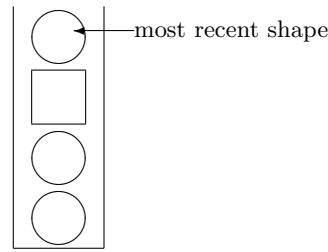

Figure 1: An example of a game state.

At each point, the child may press one of two buttons, 'L' and 'R'. 'L' should be the button to press if there is a square on top, and 'R' if there is a circle on top. More precisely, the rules are as follows:

- If the child presses the correct button ('L' when the top shape is a square, 'R' when it is a circle), the child gets a point, and the top shape disappears.

- If the child presses the wrong button ('L' when the top shape is a circle, 'R' when it is a square), the child loses a point (the points could become negative), and no shape disappears.

- If the child presses a button when the pile is empty, nothing happens.

- The game starts at 0 points.

- Items appear at a regular pace, and their appearance has nothing to do with what buttons are pressed.

- The game may end even when there are still items left.

You will be given a sequence consisting of the letters 'L', 'R', 'S', 'C' (representing the pressing of the two buttons, and the arrival of the two shapes, respectively), and are to determine from it the final score of the child. We promise that the sequence will contain only those four letters, and nothing else.

As an example, for the sequence "SCLLR", the final score will be -1, and there will be a square left in the end. The two presses of the 'L' button are wrong, so each incurs a penalty of 1 point and changes nothing. The press of 'R' is correct, so it earns a point and makes the circle disappear, but the square remains.

Insert your code in the following:

```
#include <iostream>
using namespace std;

int main ()
{
    string s;    // the sequence of characters describing the game's events
    int score = 0;
    cin >> s;

    // your code goes here.



















    cout << "The score is " << score << endl;
    return 0;
}
```

(6) [15 points]

You will be given a directed graph in which nodes are described by their names (strings), and you are supposed to read in the graph structure fast. More specifically, Ms. Trojan already implemented for you the following class:

```
class GraphNode {
   public:
      GraphNode (const string & name);
            // generates a new node without edges, with the given name
            // runs in time O(1)
      void addEdgeTo (GraphNode *otherEndpoint);
            // adds an outgoing edge from this node to otherEndpoint
            // runs in time O(1)
      void addEdgeFrom (GraphNode *otherEndpoint);
            // adds an incoming edge to this node from otherEndpoint
            // runs in time O(1)
      const string & getName () const;
            // returns the name of the node in time O(1)
      // other private and public parts won't be relevant here
};
```

You will be asked to read a directed graph in the following format: the first line contains n and m, the number of nodes and the number of edges. The next n lines each contain the name of one node; names will only be lowercase characters, and nothing else (in particular, no spaces). The next m pairs of lines each describe one edge: the first line of a pair will be the name of the start node of the edge, and the second line the name of the end node. Only valid node names listed above will occur.

You are to generate the graph structure, by creating all the necessary nodes, and correctly adding all the edges between them. Specifically, for each directed edge (u, v), you should add v among the outgoing edges of u, and u among the incoming edges of v.

For full credit, your solution should run in time $O((m + n) \log(m + n))$. If your solution is slower than that, you will get almost no credit. The code skeleton on the next page is supposed to help you. If you don't like it, feel free to cross out any of the code we gave you.

```cpp
#include <iostream>
#include <string>




using namespace std;

int main ()
{
    int n, m;
    cin >> n >> m;







    for (int i = 0; i < n; i ++)
    {
        string nodename;
        cin >> nodename;









    }
    for (int j = 0; j < m; j ++)
    {
        string nodenameStart, nodenameEnd;
        cin >> nodenameStart;
        cin >> nodenameEnd;








    }
    // other stuff using the graph goes here - not yours to worry about
}
```

(7) [15 points]

Your friend Brutus the Bruin is a little weirded out by putting an entire tree in an array when implementing Min-Heaps (or Max-Heaps). He prefers the comfort of his `Node*` class and explicitly building the links, because it avoids the index calculations. Because he isn't such a good programmer, either (maybe that's why he doesn't really understand Heaps?), he asks you to help with his project of implementing a Min-Heap in this way.

He would like your help in writing the function `removeMin`. Your function needs to be correct, but as you may have guessed, keeping the heap at height $O(\log n)$ is much harder, and Brutus is not expecting you to accomplish this.

```
struct Node {
  Node *left, *right, *parent;
  int priority; // smaller means higher priority.
  int element;
    // this is an index of where to find the actual element in an array somewhere
};

void removeMin (Node *root)
/* yours to implement. You don't need to keep the heap balanced, but
of course, you do have to make sure you keep a heap.
If you want helper functions, you may add them. */
```