Data Structures and Objects
CSIS 3700

Spring Semester 2018 — CRN 21212

Second Midterm

Part I. Short answer. 8 points each.

1. The following snippet of code purports to delete all nodes of a linked list by following the links from node to node and deleting all of the nodes:

```
1 ptr = head;
2 for (int i=0;i<count;i++) {
3   delete ptr;
4   ptr = ptr -> next;
5 }
6 head = NULL;
7 count = 0;
```

   Why doesn't this work? Describe (no code necessary) how to delete the list properly. (*Hint*: A small change to the snippet fixes the problem.) **Answer:**
   Once you delete what the pointer points to, you cannot follow it and access what is there.

   The easiest way to fix the problem is to make a copy of the next pointer first, then delete the pointer, then copy the next pointer into **`ptr`**.

2. Consider MergeSort and QuickSort. Normally, QuickSort is faster because MergeSort must copy data to and from a second array. Are there any instances when MergeSort would be faster?

   Answer:
   When the list is already sorted or almost sorted, either in ascending or descending order, QuickSort performs $\Theta(n^2)$ comparisons and uses $\Theta(n)$ space for the control stack due to recursion. In these cases, MergeSort is faster because it always performs $\Theta(n \lg n)$ comparisons.

3. Why doesn't binary search work well on linked lists, even if the data is sorted?

   Answer:
   Binary searches rely on constant-time random access for their $\Theta(\lg n)$ speed; that is, for binary searches to run fast, they must be able to access any element in the list in the same amount of time.

   With a linked list, you must walk to a specific location in the list. The first midpoint requires following $n/2$ pointers, the second midpoint follows an additional $n/4$, the next follows $n/8$,

and so on. The total number of pointers followed is $n - 1$, which is as bad as a sequential search in the worst case.

4. In a queue, why do we need to store the count separately and not just use the distance between head and tail?

   Answer:
   You can use just head and tail indexes for almost all instances. However, if the queue is empty or if it is full, the two indexes are in the same position relative to each other. Thus, by only using the indexes, you cannot distinguish between a full queue or an empty queue.

5. Consider InsertionSort and MergeSort. Which one is better, on average? How do you know this? Are there any instances where the "better" sort actually performs worse than the other?

   Answer:
   On average, MergeSort is better since it performs $\Theta(n \lg n)$ comparisons where InsertionSort averages $\Theta(n^2)$ comparisons. However, if the data is sorted or almost sorted in ascending order, InsertionSort will only perform $\Theta(n)$ comparisons, which is better than MergeSort whose run time is independent of how the data is ordered.

6. What is a circular array? (It's the kind of array used to implement a queue, but you can't use that in your answer.) How is one implemented?

   Answer:
   A circular array is one where the indexes used wrap around when they reach the end of the array. They can be implemented by incrementing the indexes when necessary, using one of these three methods:

```
1 // first method
2 if (++tail == QUEUE_SIZE)
3   tail = 0;
4
5 // second method
6 tail = ++tail % QUEUE_SIZE;
7
8 // third method
9 tail = ++tail & (QUEUE_SIZE - 1);
```

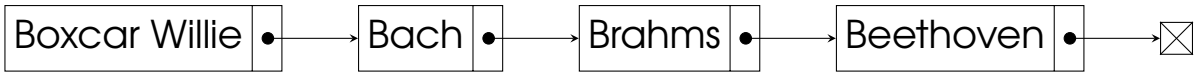
Part II. Make a list, check it twice. 15 points each.

1. Catbert the Evil Human Resources Director wants to keep a list of employees. Show the basic steps necessary — and the order in which they're performed! — to add Asok the Intern between Wally and Loud Howard.

Answer:
Steps:

(a) Create a node and place Asok in the node
(b) Find and point to the predecessor, Wally
(c) Copy Wally's pointer to Loud Howard into Asok's node
(d) Point Wally to Asok
(e) Increment the count

2. One of these things is not like the other. Show the steps to delete it from the list. If you *really* need to know which one doesn't belong, ask.



Answer:
Boxcar Willie is the node to be deleted, since he isn't a classical composer. The steps are:

(a) Point to Boxcar Willie — this is the special case, so you just copy the head pointer into a temporary pointer
(b) Copy Boxcar Willie's pointer into the head pointer
(c) Delete Boxcar Willie
(d) Decrement the counter

Part III. Code trace. 20 points. Show all output.

```
1  #include <iostream>
2  #include <queue.h>
3  #include <linearlist.h>
4
5  using namespace std;
6
7  void print(char &ch) { cout << ch; }
8
9  int main(void) {
10   string team = "Saskatchewan_Squids";
11   Queue<char> q;
12   LinearList<char> list;
13   int i;
14   char c;
15
16   for (i=0;i<team.length();i++)
17     // capital S below
18     if (team[i] == 'S' || team[i] == 't' || team[i] == 'e')
19       q.enqueue(team[i]);
20     else
21       list.insert(0,team[i]);
22
23   while (!q.isEmpty()) {
24     c = q.dequeue();
25     cout << c;
26   }
27
28   list.traverse(print);
29
30   cout << endl << "Done." << endl;
31
32   return 0;
33 }
```

Answer:
The `for` loop iterates over each character of the string.

Uppercase S and lowercase t and e are enqueued, so after the loop the queue contains StES.

The remaining characters are added to the front of the list — always position 0. In effect, the list acts like a stack, with current contents pushed down each time a new character is added to the list. Thus, at the end of the loop, the list contains sdiuq nawhcaksa.

The contents of the queue are output first, then the list is output in order. Thus, the output is:

```
Stessdiuq nawhcaksa
Done.
```

Part IV. Extra credit. 7 points.

Explain how to implement a queue using a linked list, keeping both enqueue and dequeue $\Theta(1)$ operations. Explain any changes you must make to the data structure.

Answer:
There are two common methods. One is to add a pointer to the `LinearList` object that points to the last node in the list. Enqueueing is done by adding to the far end of the list and dequeueing is done by removing the first element.

The other method is to use a circular, doubly-linked list. Enqueueing is done by adding to the far end, finding the last node by following the back pointer from the first node. Dequeueing is done by removing the first element.