Richard To
CS330 HW1 Write Up

7
---

a and b) For the sorting algorithm, I implemented merge sort. I used arrays in my implementation, so I had to create the initial array from the linked list and then convert the sorted array into a linked list.

Merge sort is O(n log n) for the worst, average, and best cases. Converting from a linked list to array would be O(n) since we would need to iterate through the linked list to add the nodes to the array. To transfer from array to linked list again would be O(n) since we would need to iterate through the array to add the nodes back to the linked list, which luckily the addLast method is O(1). But for a big enough n, only the run time for merge sort matter, so it will be O(n log n).

Pseudocode:

```
    arr = []
    while list.hasNext:
        arr.push(list.next)

    sorted_arr = merge_sort(arr)

    while sorted_arr:
        list.add(sorted_arr)

    def merge_sort(arr):

        if arr.len <= 1:
            return arr

        for i = 0; i < arr.len/2; i++:
            left[] = arr[i]

        mid = arr.len / 2
        for i = 0; i < arr.len - mid; i++:
            right[] = arr[mid + i]

        left = merge_sort(left)
        right = merge_sort(right)

        return merge(left, right)

     def merge(left, right):

        result = []

        while !left.empty and !right.empty:
            if left.current >= right.current:
                result.push(left.pop_current())
            else:
                result.push(right.pop_current())

        while !left.empty:
            result.push(left.pop_current())

        while !right.empty:
            result.push(right.pop_current())

        return result
```

c)

I tried to use the Java System.nanoTime() method to do benchmarks, but it gave me inconsistent results. I think this is due to the JVM.

```
Run 0
Count: 1000    Time: 8316000 ns
Count: 10000   Time: 98831000 ns
Count: 100000  Time: 358395000 ns

Run 1
Count: 1000    Time: 284000 ns
Count: 10000   Time: 9781000 ns
Count: 100000  Time: 56893000 ns

Run 2
Count: 1000    Time: 316000 ns
Count: 10000   Time: 3099000 ns
Count: 100000  Time: 104849000 ns
```

8
---

a and b) The best case scenario for the add method is adding nodes to the head of the list. The node should be added to the head in O(1) time. The average and worse cases would be O(n) since the iterator would have to search in order to find the position of the nth node.

Pseudocode:

```
def add(pos, object):
```

```
if(pos == 0){
    if(!isEmpty()){
        head.prev = new Node(object, null, head);
        head = head.prev;
    } else {
        head = tail = new Node(object);
    }
} else if(!isEmpty()){
    InternalIter iter = nodeIterator();
    int curPos = 0;
    Node prev = null;
    Node current = null;
    while(iter.hasNext()){
        prev = current;
        current = iter.next();
        if(curPos == pos){
            Node newNode = new Node(object, prev, current);
            prev.next = newNode;
            current.prev = newNode;
            return this;
        }
        ++curPos;
    }
}
```

c)

```
Best Case Run 0
Count: 1000    Time: 7321000 ns
Count: 10000   Time: 42821000 ns
Count: 100000  Time: 132195000 ns

Best Case Run 1
Count: 1000    Time: 103000 ns
Count: 10000   Time: 1004000 ns
Count: 100000  Time: 60357000 ns

Best Case Run 2
Count: 1000    Time: 91000 ns
Count: 10000   Time: 755000 ns
Count: 100000  Time: 64137000 ns

Worst Case Run 0
Count: 1000    Time: 12102000 ns
Count: 10000   Time: 275556000 ns
Count: 100000  Time: 99095511000 ns

Worst Case Run 1
Count: 1000    Time: 1295000 ns
Count: 10000   Time: 306585000 ns
Count: 100000  Time: 77112483000 ns

Worst Case Run 2
Count: 1000    Time: 1205000 ns
Count: 10000   Time: 260828000 ns
Count: 100000  Time: 71324190000 ns
```