Alexander Shah
Homework 4: Queues and Lists
EN.605.202.81 Section 84

1.

ADT Priority Queue

An empty list of values with data and a priority (integer)

A priority queue is a FIFO queue implemented with a data structure like a stack or array, with a specified integer for the priority of the element in the queue. Elements with higher priority are dequed before lower priority ones. Elements can be any type since the priority and timing when an item is queued determines ordering since items with the same priority are dequeued in FIFO order.

Methods:

Enqueue

Enqueue(element, priority) → None

- **Input:**
  - Element: A data item to store in queue
  - Priority: an integer to specify the priority of the element
- **Preconditions**: None
- **Process**: Add an element to the queue with a specified priority
- **Postconditions**: An element is added to the queue based on priority
- **Output**: None

Dequeue

Dequeue() → Element

- **Input:** None
- **Preconditions**: The queue must not be empty
- **Process**: The element with the highest priority is removed from the queue and returned
- **Postconditions**: The highest priority element is removed from the queue, if multiple elements have the same priority, the first in element is removed
- **Output**: An element

Delete

Delete(element) → None

- **Input:**
  - Element: A data item to remove from queue
- **Preconditions**: The element must exist in the queue
- **Process**: The specified element is removed from the queue
- **Postconditions**: An element is removed from the queue without changing the ordering
- **Output**: None

Empty

Empty() → None

- **Input:** None
- **Preconditions**: The queue must exist
- **Process**: All elements of the queue are removed
- **Postconditions**: The queue is empty
- **Output**: None

isEmpty

isEmpty() → Boolean

- **Input:** None
- **Preconditions**: The queue must exist

- **Process**: Determine if the queue has any elements in it
- **Postconditions**: Queue remains unchanged
- **Output**: Boolean true if there are elements in the queue, otherwise false

Size

Size() → Integer

- **Input:** None
- **Preconditions**: The queue must exist
- **Process**: Calculate the number of elements in the queue
- **Postconditions**: The queue remains unchanged
- **Output**: Integer describing the number of elements in the queue

Peek

Peek() → Element

- **Input:** None
- **Preconditions**: The queue must not be empty
- **Process**: Return the highest priority element without removal
- **Postconditions**: The queue remains unchanged
- **Output**: The element with the highest priority.

End ADT Priority Queue

2.

In order to reverse a singly linked list, we need to keep track of a few extra pointers in order to properly reverse which elements the links point to. We keep track of the previous element, the current element, and the next element. Using an iterative approach through the elements of the list, we temporarily store the current node's next as the next element, assign the current node's next to the previous element (or null if we are at the head), and then update the previous element to be our current element, and our current element to be the next one. Finally we return the head, or the former tail, of our list.

```
Method reverseSinglyLinkedList(head):
   // Determine if we are looking at a single element list, which is already reversed
   if head == null or head.next == null:
      return head

   //otherwise keep track of pointers to swap elements
   prev = null
   current = head
   next = null

   //swap element pointers around until we get through the list
   while current != null

      //where we're going next
      next = current.next

      //swap this element's next pointer to point to previous element
      current.next = prev

      //save this element to be the next one's previous
```

```
        prev = current

        //update current so we can move on to the next element in the list
        current = next

    //return the new head, former tail
    return prev
```

3.

Searching an unordered list we would need to search from beginning to end since the element could be anywhere in the list. In a linked structure we need to access each node by node so the average case would be searching half the elements.

Searching an ordered list, we still need to access each node by node so the average case would be searching half the elements even though we know the list is in order.

Searching an unordered array we would access half the elements on average because an unordered array has to be searched from beginning to end.

In an ordered array the search can occur much faster, since binary search can eliminate half the nodes to be searched with every comparison. This makes the average number of nodes accessed log(n).

4.

In order to swap elements in a singly linked list, we need to consider that elements m and n may both exist, one may not exist, or they could be the same. We must swap the pointers that point to m and n, and the next node m and n point to, by using placeholder pointers.

Method swapMandN(llist, m, n):

```
    //unable to swap elements if they are the same
    if m==n:
        return null

    //pointers to keep track of elements
    prevM = null
    prevN = null
    thisM = llist.head
    thisN= llist.head

    //find M and previous M
    while  thisM != null and thisM.data != m:
        prevM = thisM
        thisM = thisM.next

    //find N and previous N
    while  thisN != null and thisN.data != n:
        prevN = thisN
```

```
    thisN = thisN.next

//one element missing, can't swap
if thisM == null or thisN == null:
    return null

//able to swap elements
temp = null

//swap previous elements' next pointers
temp = prevM.next
prevM.next = prevN.next
prevN.next = temp

//swap current elements' next pointers
temp = thisM.next
thisM.next = tempN.next
thisN.next = temp
```