Homework Four : Queues and Lists

Question One

> Develop an ADT specification for a priority queue. A priority queue is like a FIFO queue except that items are ordered by some priority setting instead of time. In fact, you may think of a FIFO queue as a priority queue in which the time stamp is used to define priority.

ADT - Priority Queue

A list of data items which are ordered by their corresponding priority flag, which is represented by a natural number, with 0 being the highest priority message, and further priority descending as the number ascends. The values are stored as a priority, value pair, with priority inside of priority tiers being based on entrty time.

Methods:

PQInsert(k, v):

input: k, the priority of the key-value pair. v, the value associated with the key preconditions: none process: store an item at the rear of the key tier of the list postconditions: The priority queue will contain an additional key-value pair outputs: nothing

PQPeek():

input: nothing preconditions: none process: returns (but does not destroy) priority queue item with the highest priority item; postconditions: The priority queue is unchanged outputs: returns null if the priority queue is empty, else returns the highest priotity item

PQPop():

input: nothing preconditions: there is at least one entry in the priority queue process: returns and then destroys the highest priority item in the queue postconditions: the queue has one less entry than the last time output: returns highest priority item

PQSize():

input: nothing preconditions: nothing process: counts the number of items in the priority queue postconditions: nothing outputs: the number of items in the priority queue

PQisEmpty():

input: nothing preconditions: nothing process: counts the number of items in the priority queue. Return true if zero, return false else. postconditions: nothing outputs: true if the priority queue is empty, else it returns false

Problem Two

> Write an algorithm to reverse a singly linked list, so that the last element become the first and so on. Do NOT use Deletion - rearrange the pointers

In order to reverse a singly linked list without deletion, we need to be able to traverse the list, while being able to keep some information about the nodes surrounding the currently traversed node, such that we can properly point the list in reverse.

We can write the linked list as head-> n1 -> n2 -> n3 -> n4 -> .. nN -> null, where nN is the tail

To reverse the list, we can see that we have to set head to nN, have n1 point to null, have n2 point to n1, and so on. To accomplish this, we need to be able to track what the previous node we were at was, thus we'll need a prev node which can be kept alongside the linked list the algo then becomes:

```
next = cur.next
cur.next = prev;
prev = cur;
cur = next;
```

We can wrap this up as an iterative function, given an input of head:

```
function reverse_list(head)
    cur = head, prev = null, next;

    while (curr != null) begin
        next = cur.next;
        cur.next = prev;
        prev = cur;
        cur = next;
    end

    return prev; // This returns a new head which points to what was the
tail
endfunction
```

Problem Three

> What is the average number of nodes accessed in search for a particular element in an unordered list? In an ordered list? In an unordered array? In an ordered array? Note that a list could be implemented as a linked structure or within an array

1. An unordered list For an unordered list, you'll on average access N/2 items, where N is the length of the list, because there's no way to be more efficient than a standard linear search
2. An ordered list For an ordered list, if its stored as an array, we can implement a binary search, but if we can only access it in order, there is no efficient manner and we'll on average access N/2 items
3. An unordered array Because it is unordered, there's no way to do a more efficient search, so we'll just access the same N/2 items
4. An ordered array The ordered array can be searched using a binary search, thus we can search it a binary search, which will on average access log(n)

Problem Four

> Write a routine to interchange the mth and nth elements of a singly-linked list. You may assume that the ranks m and n are passed in as parameters. Allow for all the ways that m and n can occur. You must rearrange the pointers, not simply swap the contents.

In order to rearrange the m and nth elements of a linked list with pointers, what that means is that we have to make it such that m point to n+1, n points to m+1, n-1 points to m, and m-1 points to m.

```
// This assumes that we'll be given a pointer to the head
// also assumes that n and m are within the list, and that the list is 0
indexed
function swap(n,m,*head);

    if m==n
        return // changes needed
    // Initialize Helper Structures
    node m_prev, n_prev, n, m;
    n = head;
    m = head;
    // get nodes and their prevs
    repeat(n-1):
        n_prev = n;
        n n.next;
    repeat(m-1):
        m_prev = m;
        m = m.next;

    if m_prev is null
        head = m
    else
        m_prev.next = n;

    if n_prev is null
        head = m;
    else
        n_prev.next = m;

    // swap next pointers
    mnext = n.next;
    nnext = m.next;
    m.next = mnext;
    n.next = nnext;

endfunction: swap
```