Problem Set 4

Justin Ely

615.202.81.FA15 Data Structures

29 September, 2015

---

1)

A priority queue can be implemented with minimal additions to the basic queue ADT. The main changes necessary are: 1) a new method to locate the appropriate insertion index based on the rank of the new data element and 2) the insert method must accept both a new element and it's priority. The Insert will be accessed by the application, where the FindInsertionIndex method can be simply internal, for the Insert method to use.

Other methods, such as ChangePriority (self-explanatory) could be implemented, but were omitted from my ADT because of the potential ambiguities in implementation and uncertainty as to whether or not they belong. Changing priority is difficult to imagine: does the change affect everything at a given priority, or just the first returned element? Does it change the priority of all elements at the specified value, or just the first? Additionally, the order of stacks and queues cannot be changed after they're populated. Given that constraint, I felt that elements added to the priority queue should not be subsequently modified.

```
ADT PriorityQueue is
    Data:  An empty list of values with a reference to the first and last items,
           as well as a priority ranking for each item.
    Methods:
        Empty:
            Input: None
            Precondition: None
            Process: Check if queue is empty
            Postconditions: None
            Output: Return True if queue is empty, otherwise return False
        Delete:
            Input: None
            Precondition: The queue contains at least 1 element
            Process: Remove item from front
            Postcondition: The queue contains 1 fewer element, the front points to
```

```
                    the next element
        Output: The deleted value
    Insert
        Input: Data to be inserted, priority of data
        Precondition: None
        Process: Search for place to insert based on priority, insert new data
        Postcondition: Queue is 1 element longer, priorities
                    remain in sorted order
        Output: None
    Peek:
        Input: None
        Precondition: Queue contains at least 1 element
        Process:  Retrieve highest priority item at the front of the queue
        Postcondition: Queue is unchanged
        Output: Return item with highest priority, at the front of queue
    FindInsertionIndex
        Input: Priority of input data
        Precondition: None
        Process: Search through priorities of queue elements,
                 locating the index to insert new element
        Postcondition:  Queue is unchanged
        Output:  Return index of where to place new element.
```

2)

My linked-list reverse algorithm uses a stack to store successive pointers, then iterates backwards over the initial list, popping the stack to redefine each node's pointer in succession.

```
ListReverse(LinkedList):
    # Create a stack to hold pointers
    S = new Stack()

    #Store NULL and pointer to first element,
    #which will be the last two values needed
    #that can't be taken from the subsequent nodes
    S.push(NULL)
    S.push(LinkedList.FirstPtr)

    #For every node in the list,
    #push pointers for each element to stack
    for (i=0, i<length(LinkedList), i++):
        S.push(LinkedList[i].NextPtr())
```

```
    #Discard final NULL pointer
    S.pop()

    #Set list pointer to point to new first element
    tmpFirstPtr = S.pop()

    #Iterate in reverse order to not rely on changing pointers.
    #For every node in the list,
    #set pointer to the pointer on the top of the stack
    for (i=length(LinkedList) - 1, i>=0, i--):
        LinkedList[i].SetPtr(S.pop())

     #set list pointer to the Nth element pointer
     LinkedList.First Ptr= tmpFirstPtr
```

3)

For unordered collections, whether in a list or an array, the average number of nodes accessed will average out to $\frac{n}{2}$ where n is the number of elements in the structure. Since the structure is unordered, each element needs to be touched until the desired value is found. Sometimes the value will be the first touched, sometimes the last (or not found at all), or anywhere in between. Over time this will average out to half the length of the structure.

For an ordered list, this depends on the specific implementation. If the list is housed in an array such that the valid elements are still continuous (no NULLS or placeholder values), then random access and binary search can be exploited and the number of nodes will be $\sim log(n)$. If instead, each element still needs to be checked either during the search or to verify that a valid value is contained in the array element, then the number of nodes accessed will be closer to $\frac{n}{2}$.

For an ordered array, random access allows a binary search algorithm to be used, which will lead to $\sim log(n)$ accessed nodes.

4)

This algorithm assumes the presence of Next and Prev methods in the list nodes. Next returns the pointer to the next element, while Prev() returns the previous nodes pointer to the current node.

```
swap_nodes(NodeA, NodeB):
    #Get pointers to and from the first node
    to_a = NodeA.Prev()
```

```
from_a = NodeA.Next()

#Get pointers to and from the second node
to_b = NodeB.Prev()
from_b = NodeB.Next()

#temporarily store copies of B pointers
to_b_tmp = to_b
from_b_tmp = from_b

#re-assign b pointers to a pointers
to_b = to_a
from_b = from_a

#re-assign a pointers to saved b pointers
to_a = to_b_tmp
from_a = from_b_tmp
```