**Homework 2 (100 points)**

**Name + NetID: Aima Salman + AXS220380**

1. *Write pseudocode to implement a queue using 2 stacks. (enqueue, dequeue, Qfull, Qempty, display) [10 points]*

```
class Queue
{
        Stack S, S1;

        void enqueue(Obj O)
        {
                push item onto stack S;
        }

        Obj O dequeue()
        {
                if (S1 is empty)
                {
                        pop items off stack S until empty
                                push popped item onto stack S1
                }

                return top item of stack S1
        }

        boolean Qfull()
        {
                if (S AND S1 is full)
                {
                        return true;
                }

                return false;
        }

        boolean Qempty()
        {
                if (S AND S1 is empty)
                {
                        return true;
                }

                return false;
        }

        void display()
```

```
        {
                if (S1 is empty)
                {
                        pop items off stack S1 until empty
                                print (popped item)
                }

                if (S is empty)
                {
                        pop items off stack S until empty
                                push popped item onto stack S1

                        pop items off stack S1 until empty
                                print (popped item)
                }

        }

}
```

2. *Write pseudocode to implement a stack using doubly linked list. (push, pop, stackEmpty, display, top) [10 points]*

```
class Node
{
        node.data = data;
        node.next = null;
        node.prev = null;
        size = 0;
}

class Stack
{
        constructor ()
        {
                set top to null; // head of list
                set size to 0;
        }

        Node top ()
        {
                if (stack is empty)
                {
                        print ("Stack is empty");
                        return null;
                }
```

```
        return first item of linked list;
}

boolean stackEmpty()
{
        if  (first item is null)
        {
                return true;
        }

        return false;
}

void push(Obj newNode)
{
        if (stackEmpty())
        {
                set top to newNode;
                return;
        }


        set top.prev to newNode;

        set newNode.next to top;
        set top to newNode;
        increment size by 1;
}

Node pop()
{
        if (stackEmpty())
        {
                print (“Stack is empty”);
                return null;
        }

        set temp to top;
        set top to top.next;
        set top.prev to null;
        decrement size by 1
        return temp;
}

void display()
{
        if (stackEmpty())
        {
```

```
                print ("Stack is empty");
                return;
        }

        set ptr to top;


        // traverse list until end

        while (ptr is not NULL)

                print each item;
    }
}
```

3. *Write pseudocode to implement a queue using doubly linked list. (enqueue, dequeue, Qempty, display) [10 points]*

```
class Node
{
    node.data = data;
    node.next = null;
    node.prev = null;
}

class Queue
{
    constructor ()
    {
        set head to null;
        set tail to null;
    }

    boolean QEmpty()
    {
        if  (head is null)
        {
            return true;
        }

        return false;
    }

    void enqueue(Obj newNode)
    {
        if (QEmpty())
        {
            set head to newNode;
```

```
                set tail to newNode;
                return;
        }

        else
        {
                // with tail
                set tail.next to newNode;
                set tail to newNode;

                // without knowing tail
                set ptr to head;
                // traverse list until end
                while (ptr is not NULL)
                        continue;
                set ptr.next to newNode;
                set newNode.prev to ptr;
        }
}

Node dequeue()
{
        if (QEmpty())
        {
                print ("Queue is empty");
                return null;
        }

        set temp to head;
        set head to head.next;
        set head.prev to null;
        return temp;
}

void display()
{
        if (QEmpty())
        {
                print ("Queue is empty");
                return;
        }

        set ptr to head;
        to end of list traverse
                print each item in list;
}
}
```

4. *Write pseudocode to implement a circular queue using doubly linked list. (enqueue, dequeue, Qempty, display) [10 points]*

```
class Node
{
        node.data = data;
        node.next = null;
        node.prev = null;
}

class CircularQueue
{
        constructor ()
        {
                set head to null;
                set tail to null;
        }

        boolean QEmpty()
        {
                if  (head is null)
                {
                        return true;
                }

                return false;
        }

        void enqueue(Obj newNode)
        {
                if (QEmpty())
```

```
        {
                set head to newNode;
                set tail to newNode;
                set head.prev and head.next to tail;
                set tail.prev and tail.next to head;
        }

        else
        {
                // with tail
                set newNode.prev to tail;
                set tail.next to newNode;
                set tail to newNode;
                set tail.next to head;

                // without knowing tail
                set ptr to head;
                // traverse list until end
                while (ptr is not NULL)
                        continue;
                set ptr.next to newNode;
                set newNode.prev to ptr;
                set newNode.next to head;
        }
}

Node dequeue()
{
        if (QEmpty())
        {
                print ("Queue is empty");
```

```
            return null;
        }

        // only one node
        if (head.next is null)
        {
            set temp to head;
            set head to null;
            return temp;
        }

        set temp to head;
        set head to head.next;
        set tail.next to head;
        set head.prev to tail;
        return temp;
}

void display()
{
        if (QEmpty())
        {
            print ("Queue is empty");
            return;
        }

        set ptr to head;
        // traverse list until end
        while (ptr is not NULL)
            print each item;
```

```
        }
    }
```

5. *In some circumstances, the normal FIFO operation of a queue may need to be overridden This may occur due to priorities that are associated the elements of the queue that affect the order of processing. In cases such as these, a **priority queue** is used, where the elements are removed based on priority. How will you implement a priority queue? (enqueue, dequeue, Qempty, display) [10 points]*

   To implement a priority queue, it is best to use a linked list to easily manage the items while maintaining a specific order. Items will be numbered from most to least priority, and that will be stored along with their data in each node. These nodes will then be dequeued normally, meaning that the head element will be removed each time. On the other hand, the enqueue function will need to be tweaked slightly to check the placement of the new node in context. This means a pointer will likely have to traverse the list to see where the new node must be placed and adjust the links accordingly. QEmpty would just check if the head node is null, and if it is then return true, else false. Display will traverse the linked list and display each node's data.

6. *Write pseudocode to swap two adjacent elements by adjusting only the links (and not the data) using following data structures [10 Points]*
   a. *Singly linked list*

```
void swap(int swap)
{
    if (head OR head.next is NULL)
    {
        return head;
    }

    if (head.data is equal to swap)
    {
        set temp to head.next;
        set head.next to temp.next;
        set temp.next to head;
        return temp;
    }
```

```
    prev = head
    curr = head.next

    while (curr AND curr.next is not NULL)
    {
        if (curr.value is x)
        {
            set temp to curr.next;
            set curr.next to temp.next;
            set temp.next to curr;
            set prev.next to temp;
            return head;
        }
        set prev to curr;
        set curr to curr.next;
    }
}
```

*b. Doubly linked list*

```
void swap(int swap)
{
    if (head OR head.next is NULL)
    {
        return head;
    }

    if (head.data is equal to swap)
    {
        set temp to head.next;
        set head.next to temp.next;
        set temp.next to head;
```

```
        set temp.prev to null;
        set head.prev to temp;
        return temp;
}

while (curr AND curr.next are not NULL)
{
        if (curr.value is x)
        {
                set ptr to curr.next;
        }

        // adjust previous and next pointers
        if (curr.prev is not NULL)
        {
                set curr.prev.next to ptr;
                set ptr.prev to curr.prev;
                set curr.next to ptr.next;
        }

        if (ptr.next is not NULL)
        {
                set ptr.next.prev to curr;
                set ptr.next to curr;
                set curr.prev to ptr;
        }
        // if swapping at the head
         if (curr is head)
        {
                return ptr;
```

```
            }

            return head
            curr = curr.next
        }
        return head

        // this makes it so that actual swap node is in the middle
        set slow to slow.next;
        set fast to fast.next;

        // swap logic
        set curr to slow.next;
        set curr.next to fast.next;
        set fast.next to curr;
        set slow.next to fast;

        set fast.prev to slow;
        set curr.prev to fast;
        set curr.next.prev to curr;
    }
```

7. *Write pseudocode to implement the 'contains' routine for LinkedList program which checks if a linked list contains a given integer. [5 Points]*

```
boolean contains(int n)
{
        set ptr to head;

        // traverse list until end
        while (ptr is not NULL)
```

```
        {
                if  (ptr's data is equal to n)
                {
                        return true;
                }
        }

        return false;
}
```

8. *The Josephus problem is the following game: N people, numbered 1 to N, are sitting in a circle. Starting at person 1, a hot potato is passed. After M passes, the person holding the potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining person wins. Thus if M = 0 and N = 5, players are eliminated in order and player 5 wins. If M = 1 and N = 5, the order of elimination is 2,4,1,5.*

    a. *Write a pseudocode to solve the Josephus problem for general (integer) values of M and N. Try to make your program as efficient as possible. Make sure you dispose of cells. [7 Points]*

```
// make circular singly linked list
int josephus(int m, int n)
{
        if (n is 0)
                return 0;

        set curr to head

        while (more than one person remains)
        {
                move m nodes through list
                while (ptr is not NULL)
                {
                        if (counter is equal to m)
                        {
                                break;
                        }
                        set prev to prev.next;
```

```
                set curr to curr.next;
            }

            keep track of node prev to curr

            // remove links
            set prev.next to curr.next;
            set curr.next to null;
            set curr to prev.next;

        }

        return last person remaining's number;
    }
```

*b. What is the running time of your program? [3 Points]*
The running time of this program is O(mn) because each elimination has to travel *m* steps, and there are *n* iterations for each new round of those steps. Because these are within each other, they will multiply to get the final time complexity.

9. *What is the running time of the following code?* [5 Points]

```
public static List<Integer> makeList(int N)
{
    ArrayList<Integer> lst = new ArrayList<>();
    for( int i = 0; i < N; i++)
    {
        lst.add(i);
        lst.trimToSize();
    }
}
```

The running time of the code above is O(n^2). The add function takes O(n) time because it must resize the array each time in order to add to the end. This is due to the fact that the array is trimmed after each iteration. Then, the *for* loop will run n times for a total of O(n). These all multiply because they are within each other.

10. *The following routine removes the first half of the list passed as a parameter: [10 Points]*

```
public static void removeFirstHalf(List<?> lst)
{
```

```
    int theSize = lst.size() /2
    for( int i = 0; i < theSize; i++ )
        lst.remove(0);
}
```

a. *Why is theSize saved prior to entering the for loop?*
Size is saved prior to entering the loop because it will change with each iteration of the loop. Since we must do the action for a set amount of time, we need to store the size so that it runs the correct amount.

b. *What is the running time of removeFirstHalf if lst is an ArrayList?*
The running time of the function is O(n^2) when lst is an ArrayList. This is because each resize of the array takes O(n) and the *for* loop will run O(n/2) (simplified to O(n)) times. The two multiply because they are within each other.

c. *What is the running time of removeFirstHalf if lst is a LinkedLIst?*
The running time of the function is O(n) when lst is a LinkedList. This is because each deletion of the list takes O(1) time while the *for* loop will run O(n/2) (simplified to O(n)) times. The two multiply because they are within each other.

d. *Does using an iterator make removeFirstHalf faster for either type of List?*
No, using an iterator does not make it faster for either type of list. In an ArrayList, it is not helpful because the issue is resizing the array, which the iterator cannot help with. Therefore, not impacting the time. Additionally, the iterator also cannot help in a LinkedList because removing from the front is already O(1).

11. *Write a function in pseudocode named removeDuplicates(), which takes a singly linked list sorted in increasing order and deletes any duplicate nodes from the list. The list should only be traversed once and the routine should not call any other routine.*

    *For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.*

    ***Input:*** *the* ***head*** *node of the linked list.*
    ***Output:*** *Your function should return a pointer to the head of linked list with no duplicate element. [10 Points]*

```
Node removeDuplicates(Node head)
{
    if (head OR head.next is null)
    {
        return head;
    }
```

```
        set ptr to head;
        // traverse list until end
        while (ptr is not NULL)
        {
                if (ptr.data is equal to ptr.next.data)
                {
                        // adjust links so that ptr is the ptr.next is skipped
                        set ptr.next to ptr.next.next;
                        // remove links from middle node
                        set ptr.next.next to null;
                }

                else
                {
                        // continue traversing list
                        set ptr to ptr.next;
                }
        }
        return head;
}
```