TABLE OF CONTENTS

Sarah AlJurbua 220410528
Refan Sagga 219410083

Introduction

The project is split into three separate phases (three classes for (stacks, queue, dequeue), Tower of Hanoi, and palindrome checker).

For Stacks, Queue, and Dequeue classes we go over the codes used and the methods implemented. Go over in extreme detail how each method works and mention (if any) alternative codes that could be used in each method and state its big O notations. The methods we used and explained are the basic implementations of each class/ data structure type (such as deleting, inserting, size … etc).

Tower of Hanoi is a mathematical game or puzzle consisting of three rods and a number of disks (in our case it's 3) of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying a set of rules: only one disk may be moved at a time, and no disk may be placed on top of a disk that is smaller than it. This game/ puzzle is mostly solved using recursion but in our solving we used stacks.

For the last part of the project, we coded a palindrome checker. What is a palindrome checker? It is lines of code that checks if the entered string would be the same if it was reversed and read backward such as the word mom, dad, and level. Our code works on strings only (not integers, doubles, or floats).

Design and Implementation

Part 1:

Class Stacks: (LinkedList) (FILO)

```
public class Stacks<L>{//as linkedlist not array list
    class node<L>{
        public L Data;
        public node<L>Next;
        public node(){//default constructor
            Next=null;//pointer set to null
            Data=null;/*data in nodes set to null */ }//end of default constructor
        public node(L GI){//parameterized constructor with GI as value
            Data=GI;//value is assigned to data
            Next=null;/*pointer set to null */ }//end of method node(L GI)
    }//end of parameterized constructor
    private node<L>Top;//top pointer(that points to the top of stack)
    private int Size;//AKA counter, checks size of stack
    public Stacks(){//stacks constructor
        Top=null;//top pointer set to null, initializes data
        Size=0;/*size set to zero as stack is empty is always empty at the beginning */ }//end of stacks constructor
    public boolean isEmpty(){//method that sicks if stack is empty by checking if top is null
        return Top==null; }//end of method isEmpty
    /*another way to check if stack is empty is by using size:
    public boolean isEmpty(){
        return size==0; } */
    public L Pop(){//method used to remove a node
        /*rule of method: if stack not empty
        if(top !=null){ */
        L cs=Top.Data;//take data from top of data as assign it to cs
        Top=Top.Next;//assign top to the data pointed to from the top's next
        Size--;//decrement size since we are removing an element
        return cs;//return data after changes, }else return null;
    }//end of method pop
    public int Size(){//method used to check the size of stack and return it to user
        return Size; }//end of method size
    public L Peek(){//method that returns data at the top of stack without making changes to it
        /*rule of method: if stack not empty
        if(top !=null){ */
        L cs=Top.Data;//takes data from top and assign it to cs
        return cs;// }else return null;
    }//end of method peek
    public void Push(L cs){//Push(item) method, inserts cs of type L
        /* rule of method: stack not full
        if (stack.Full)
            return null;
        else{ */
        node<L>Temporary=new node<L>(cs);//assign cs as temporary(new element)
        Temporary.Next=Top;//make new element(temporary point to the top of stack)
        Top=Temporary;
        Size ++;//size incremints by 1 after adding an element
        // }
    }// end of method push
}//end of class stacks
```

In class stacks, we were asked to include/ implement five methods (Push(item), Pop(), Peek(), isEmpty(), and Size()).

Created two constructors (default and parameterized) for the class node. The default constructor sets Data and Next to null (Data and Next were declared in the Stacks class prior to the constructors). The parameterized constructor sets Data with the value specified in the constructor's parameter and sets Next to null.

Declared Top (pointer) and size (counter), created a default constructor for class stacks that initializes the two variables to null (size to zero).

isEmpty() method is created, it checks if the stack is empty by checking if top is null therefore, it returns top as null if the validation is true. Another way to check is by checking if the size equals zero.

Pop() method is used to remove a node in stacks, and has a rule prior to using/implementing it; the stack can't be empty therefore “ if(top !=null) “ is used. If the stack isn't empty, we take data from top of data and assign it to any variable (cs in our case), then assign top to the data pointed to from the top's next. Size is always decremented in pop method before sending/returning the data/stack to the user after changes.

Size() is a method used to check the size of the stack and return it to the user upon the usage of the method therefore, it returns the variable Size that holds the number for the actual size of the stack.

Peek() method is used to return data at the top of the stack without making changes to it, and it has a rule prior to using this method; stack can't be empty. To check we used “ if(top !=null) “if it’s true (stack is not empty) we take data from top and assign it to a variable (in our case cs). The last step is returning the data to the user.

Push(item) method inserts said item/data of said type into the stack (inserts cs of type L), and has a rule prior to using this method (if an ArrayList); stack can’t be full therefore “ if(stack.isEmpty) → return null; “ is used. If the stack isn't full, we assign a variable (cs in our case) as temporary (new element), make a new element/ node (temporary point to the top of stack) then assign temporary as top (pointer). Finally, increment size by 1.

Class Queue: (LinkedList) (FIFO)

```java
public class Queue<L>{//as linkedlist not array
    class node<L>{
        public L Data;
        public node<L>Next;
        public node(){ //default constructor
            Data=null;//data in nodes set to null
            Next=null;//pointer set to null
        }//end of default constructor
        public node(L GI){//parameterized constructor with GI as value
            Next=null;//pointer set to null
            Data=GI;//value is assigned to data
        }//end of method node(L GI)
    }//end of parameterized constructor
    private node<L>tail,head;//head responsible for deleting elements, tail responsible for adding elements
    private int Size;//AKA counter, checks size of stack
    public Queue(){//Queue constructor
        tail=head=null;//initializes data by setting head and tail to null
        //can be rephrased as: head=null; tail=null;
        Size=0;//size set to zero as queue is empty is always empty at the beginning
    }//end of Queue constructor
    public void Enqueue(L cs){//method used to add a node
        if(tail==null){//if queue is empty
            head=tail=new node<L>(cs);//since queue is empty, added element will be pointed as head and tail
        }//end of if case
        else{//if queue is not empty
            tail.Next=new node<L>(cs);//new node made
            tail=tail.Next;//new node is assigned as tail
        }//end of else case
        Size++;//size increments everytime a node is added
    }//end of method enqueue
    public boolean isEmpty(){/*method that checks if queue is empty by checking if head and tail is null:
        head=tail=null;
        or by checking if size is zero: */
        return Size==0; }//end of method isEmpty
    public L Dequeue(){//method used to remove a node
        if(Size==0)//check if queue is empty
            return null;
        L SE=head.Data;//if not empty, take data from head and assign to SE
        head=head.Next;//change head pointer to data in privious head's next
        Size--;//decrement size since we are removing an element
        if(Size==0){//if size becomes 0, it means we removed the last element in queue
            tail=null; }//so tail will=null since queue is empty now
        return SE;//return data after changes
    }//end of method dequeue
    public int Size(){//method used to check the size of queue and return it to user
        return Size; }//end of method size
}//end of class queue
```

In class Queue, we were asked to include/ implement four methods (Enqueue(item), Dequeue(), isEmpty(), Size()).

Created two constructors (default and parameterized) for the class node. The default constructor sets Data and Next to null (Data and Next were declared in the Stacks class prior to the constructors). The parameterized constructor sets Data with the value specified in the constructor's parameter and sets Next to null.

Declared head and tail (pointers) and size (counter), created a default constructor for class Queue that initializes the variables to null (size to zero).

Enqueue() method is used to add a node and has a rule prior to using it, the queue can’t be full (if an ArrayList) therefore, “ if(tail=null) “ is implemented. If the queue isn't full, added element/ node will be pointed as head and tail. If the queue is NOT empty, a new node is assigned as tail then the size is incremented every time a node is added (regardless of if the queue is empty or not).

isEmpty() method checks if the queue is empty by checking if head and tail are null: head=tail=null; or by checking if the size is zero.

Dequeue() method is used to remove a node (FIFO) but has a rule prior to using it, queue can't be empty; “ if(size==0) return null; “. **If the queue is not empty**, we take data from head and assign it to a variable (SE in our case), then change head pointer to data in previous head's next, then decrement size since we are removing an element/ node. **If the queue is empty**, it means we removed the last element in the queue so tail will equal null since the queue is empty now. Finally, return data/ queue to the user after changes.

Size() method checks the size of the queue and returns it to the user upon the usage of the method therefore, it returns the variable Size that holds the number for the actual size of the stack.

Class Dequeue: (doubly ended queue)

```
public class Dequeue<L>{//as doubly ended queue
    class node<L>{
        public L Data;
        public node<L>Next;
        public node(L GI){//parameterized constructor with GI as value
            Data=GI;//value is assigned to data
            Next=null;//pointer set to null
        }//end of parameterized constructor
        public node(){//default constructor
           Next=null;//pointer set to null
           Data=null;//data in nodes set to null
        }//end of method node
    }//end of default constructor
    private node<L>tail,head;
    private int Size;//AKA counter, checks size of doubly ended queue
    public Dequeue(){
        tail=head=null;//initializes data by setting head and tail to null
        //can be rephrased as: head=null; tail=null;
        Size = 0;//size set to zero as queue is empty is always empty at the beginning
    }//end of method Dequeue
    public boolean isEmpty(){/*method that checks if queue is empty by checking if head and tail is null:
        head=tail=null;
        or by checking if size is zero: */
    return Size==0; }//end of method isEmpty
    public void addFirst(L CS){//adds node to the beggining of queue(head)
        if(head==null){//if queue is empty
            head=tail=new node<L>(CS);//since queue is empty, added element will be pointed as head and tail
        }//end of if case
        else{//if queue is not empty
            node<L>Temporary= new node<L>(CS);//new node made
            Temporary.Next =head;//new node is assigned as head
            head = Temporary;//assign temporary as head
        }//end of else case
        Size++;//size increments everytime a node is added
    }//end of method addFirst
    public void addLast(L cs){//similar to enqueue
        if(tail==null){//if queue is empty
            head=tail=new node<L>(cs);//since queue is empty, added element will be pointed as head and tail
        }//end of if case
        else{//if queue is not empty
            tail.Next=new node<L>(cs);//new node made
            tail=tail.Next;//new node is assigned as tail
        }//end of else case
        Size++;//size increments everytime a node is added
    }//end of method addLast
    public int Size(){//method used to check the size of doubly ended queue and return it to user
        return Size; }//end of method size
    public L removeFirst(){//similar to dequeue
        if(Size==0)//check if queue is empty, can be rephrased as: if(head==null)
```

Big O notation for all methods in this class is O(1), except for method removeLast() because we used a while loop (its **big O notation is O(n)**).

In class Queue, we were asked to include/ implement four methods (addFirst(), addLast(), removeFirst(), removeLast(), isEmpty(), Size()).

Created two constructors (default and parameterized) for the class node. The default constructor sets Data and Next to null (Data and Next were declared in the Stacks class prior to the constructors). The parameterized constructor sets Data with the value specified in the constructor's parameter and sets Next to null.

Declared head and tail (pointers) and size (counter), created a default constructor for class Queue that initializes the variables to null (size to zero).

isEmpty() method checks if the queue is empty by checking if head and tail are null: head=tail=null; or by checking if the size is zero.

addFirst() method adds a node to the beginning of the queue (head) and has a rule prior to implementing the method; queue can't be full therefore, “ if(head==null) “. **If the queue is empty**, the added element/ node will be pointed as head and tail. **If the queue is not empty**, a new node is made (temporary) and assigned as head then size is incremented every time a node is added (regardless of if the queue is empty or not).

addLast() method is similar to enqueue, it is used to add a node, and has a rule prior to using it, the queue can’t be full (if an ArrayList) therefore, “ if(tail=null) “ is implemented. If the queue isn't full, added element/ node will be pointed as head and tail. If the queue is NOT empty, a new node is assigned as tail then the size is incremented every time a node is added (regardless of if the queue is empty or not).

Size() method checks the size of the queue and returns it to the user upon the usage of the method therefore, it returns the variable Size that holds the number for the actual size of the stack.

```
            return null;
        L SE=head.Data;//if not empty, take data from head and assign to SE
        head=head.Next;//change head pointer to data in privious head's next
        Size--;//decrement size since we are removing an element
        if(Size==0){//if size becomes 0, it means we removed the last element in queue
            tail = null; }//so tail will=null since queue is empty now
        return SE;//return data after changes
    }//end of methhod removeFirst
    public L removeLast(){
        if(Size==0)//check if queue is empty, can be rephrased as: if(tail==null)
            return null;
        L SE=tail.Data;//if not empty, take data from head and assign to SE
        if(tail==head){//if true, it means we onlt have one element/node in the queue
            head=tail=null; }//by setting head and tail to null we are removing the last node in the queue
        else{//if it is not the last node in te queue:
        //to delete node from tail we have to traverse through te queue from head all the way to the tail
            node<L>previous=head;//point the previous at the head
            while(previous.Next !=tail){//the while loop doesnt stop till the "previous" node has traversed to the last node befpre the tail
                previous=previous.Next;}//end of while loop
            previous.Next=null;//set the previous' next as null(so deleting the node at the tail)
            tail=previous;//asign previous as te tail
            Size--;//decrement size since we are removing an element
        }//end of else case
        return SE;//return data after changes
    }//end of method removeLast
}//end of Dequeue class
```

removeFirst() method is similar to dequeue, it is a method that is used to remove a node (FIFO) but has a rule prior to using it, queue can't be empty; " if(size==0) return null; ". **If the queue is not empty**, we take data from head and assign it to a variable (SE in our case), then change head pointer to data in previous head's next, then decrement size since we are removing an element/ node. **If the queue is empty**, it means we removed the last element in the queue so tail will equal null since the queue is empty now. Finally, return data/ queue to the user after changes.

removeLast() method removes the node from the end and has a rule prior to implementing the method, queue can't be empty; " if(size==0) return null ", can be rephrased as if(tail==null). If the queue is not empty, we take data from head and assign it to a variable (in our case SE). Check if we have only one node in the queue, if true, we remove the last node by setting the head and tail to null. If not true, to delete a node from tail we have to traverse through the queue from head all the way to the tail, so point a variable (previous in our case) at the head and use a while loop to traverse till it reaches before the last node (prior to tail), set the previous' next as null (so deleting the node at the tail) and assign previous as the tail. Decrement size since we are removing an element/ node and then sending back the data/ queue after changes.

Part 2:

B: Tower Of Hanoi

```
public class Stack{//as linkedlist not array list
    class node{
        public int Data;
        public node Next;
        public node(){//default constructor
            Next=null;//pointer set to null
            Data=0;/*data in nodes set to zero */ }//end of default constructor
        public node(int GI){//parameterized constructor with GI as value
            Data=GI;//value is assigned to data
            Next=null;/*pointer set to null */ }//end of method node(L GI)
    }//end of parameterized constructor
    private node Top;//top pointer(that points to the top of stack)
    private int Size;//AKA counter, checks size of stack
    public Stack(){//stacks constructor
        Top=null;//top pointer set to null, initializes data
        Size=0;/*size set to zero as stack is empty is always empty at the beginning */
    }//end of stacks constructor
    public boolean isEmpty(){//method that checks if stack is empty by checking if top is null
        return Top==null; }//end of method isEmpty
    /*another way to check if stack is empty is by using size:
    public boolean isEmpty(){
        return size==0; } */
    public int Pop(){//method used to remove a node,used int
        /*rule of method: if stack not empty
        if(top !=null){ */
        int cs=Top.Data;//take data from top of data as assign it to cs
        Top=Top.Next;//assign top to the data pointed to from the top's next
        Size--;//decrement size since we are removing an element
        return cs;//return data after changes, }else return null;
    }//end of method pop
    public int Size(){//method used to check the size of stack and return it to user
        return Size; }//end of method size
    public int Peek(){//method that returns data at the top of stack without making changes to it
        /*rule of method: if stack not empty
        if(top !=null){ */
        int cs=Top.Data;//takes data from top and assign it to cs
        return cs;// }else return null;
    }//end of method peek
    public void Push(int cs){//Push(item) method, inserts cs of type L
        /* rule of method: stack not full
        if (stack.Full)
            return null; else{ */
        node Temporary=new node(cs);//assign cs as temporary(new element)
        Temporary.Next=Top;//make new element(temporary point to the top of stack)
        Top=Temporary;
        Size ++;//size increments by 1 after adding an element
        // }
    }// end of method push
```

```
    }// end of method push
    public void printinfo(){//method used to print info about the state of each tower/rod
        if(Size==0)//if case,if the tower/ rod is empty it will print a message declaring that it is empty
            System.out.println("Tower/ rod is empty");
        else{node Temporary = Top;//make node named temporary the pointer(top)
            while(Temporary!=null){//while loop with condition of node(temporary) isnt null/ empty
                System.out.println(Temporary.Data);//print out the data of the node temporary
                Temporary = Temporary.Next;}//go to the next node using the current node's next and assign the new node as (temporary)
        }//end of else case
    }//end of printinfo method
}//end of class stacks
```

For **class Stack,** we used the same code/ class used in Part 1 (class Stacks, Dequeue, Queue). Changes up the data type received/ sent out to integer since our use for the stacks in this instance is to name the towers (in numbers) and indicate where each disk has moved from which tower.

Added a **printinfo method** that helps in printing out the information for a certain tower by calling out the method (example: Tower1.printinfo();). This method aids in making the printing of each tower/ rod easier to identify where the disks are located at a specific instance.

```
package cs210b;
import java.util.Scanner;//import used to read/ obtain inputs from user using primitive data types
public class TowerOfHanoi{
    public static int N;//using static for the memory management
     public static Stack tower1=new Stack(); //creating three arrays(for 3 towers)of class stack
     public static Stack tower2=new Stack();
     public static Stack tower3=new Stack();
     public static int number;//declaring a variable to keep count of number of disks a user wants
}//end of TowerOfHanoi class
```

Created a second class, named TowerOfHanoi. We Created three different arrays all of the class stack (that was created above). Named each array as Tower 1, Tower 2, and Tower 3 (to represent the towers/ rods). Declared a static variable of type integer to keep count of the number of disks the user inputs when asked in the main class.

```
package cs210b;
import static cs210b.TowerOfHanoi.number;//import to avoid declaring variable number again
import static cs210b.TowerOfHanoi.tower1;//importing all arrays of class stack
import static cs210b.TowerOfHanoi.tower2;
import static cs210b.TowerOfHanoi.tower3;
import java.util.Scanner;
public class CS210B{
    public static void main(String[]args){
        Scanner scan=new Scanner(System.in);//receive user input and parse them into primitive data types
        System.out.println("Enter the number of disks:");//letting the user pick the number of disks since it stated"3 disks MINIMUM"in the instructions
        number=scan.nextInt();////saves user input to variable number
        //can declare variable number in this class again instead of import
        int moves=(int)Math.pow(2,number)-1;//can split into declaring and initializing separately
        //declaring variable moves to save the number of moves made till the end
        //a rule(using math.pow(number,to the power of),to find the num of moves depending on the num of disks,used to calculate and save it at variable moves
        for(int J=number;J>= 1;J--)//declare and initalize variable J for the for loop
            tower1.Push(J);//stop case for the loop is J=1,till then insert i(disks) into the first tower
        Stack ONE = tower1;//stack one has all(3)disks and this is a constant(not changing)
        Stack TWO;
        Stack THREE;//initializing previously declared arrays to named stacks
        System.out.println("Before moving disks:");//printing the info of rods prior to moving the disks
        if(moves%2==0){//if number of moves is even
            TWO = tower3;//intializing third tower to second stack
            THREE = tower2;//intializing second tower to third stack
            System.out.println("Source Tower (FIRST) data is:");//prints out the order of disks at source tower
            tower1.printinfo();//print info method used to get info about first tower
            System.out.println("Destination of Tower (SECOND) data is:");//print out info about disks at destination tower
            tower2.printinfo();}//print info method used to get info about third tower
        else{//if number of moves is odd
            TWO = tower2;//intializing second tower to second stack
            THREE = tower3;//intializing third tower to third stack
            System.out.println("Source Tower (FIRST) data is:");//prints out the order of disks at source tower
            tower1.printinfo();//print info method used to get info about first tower
            System.out.println("Destination of Tower (THIRD) data is:");//print out info about disks at destination tower
            tower3.printinfo();}//print info method used to get info about third tower
        for(int J=1;J<=moves;J++){//for loop with stop case that J doesnt exceed number of moves calculated above
            if(J%3==1)//if J=3
                movingOfDisks(ONE,THREE,"Tower 1","Tower 3");//send these parameters to method movingOfDisks
            else if(J%3==2)//if J=6
                movingOfDisks(ONE,TWO,"Tower 1","Tower 2");//send these parameters to method movingOfDisks
            else if(J%3==0)
                movingOfDisks(TWO,THREE,"Tower 2","Tower 3");}//send these parameters to method movingOfDisks, end of for loop
        System.out.println("After moving disks:");//print sentces that declares data after moving of disks
        if(tower3.Size()>0){//if the size of the third tower is greater than 0
            System.out.println("Source Tower data is ");
            tower1.printinfo();//print info for first tower(source)
            System.out.println("destination Tower data is ");
            tower3.printinfo();}//print info for third tower(destination)
        else if(tower2.Size()>0){//if the size of the second tower is greater than 0
```

Imported the three arrays from class TowerOfHanoi as well as, imported the variable number from the class TowerOfHanoi to avoid having to declare and initialize it again in class cs210B. Its **big O notation is O(n).**

Asked the user to input the number of disks to be used to solve the puzzle/ game using a scanner (import for the scanner is included), the input from the user is saved into the previously initialized variable, number. Math pow(number, to the power of) is used to deduce a rule for the game that calculated the number of moves to solve the game/ puzzle, depending on the number of disks the user inputs as there are different ways to solve the game if the number of moves is even or odd.

Used a For loop to help push all disks into the first rod/ tower and save the first tower/ rod as stack ONE, then declared two different stacks named TWO and THREE. Printed out a sentence to declare that the following data to be printed are the data of disks and towers PRIOR to moving/ changes. Separated the printout into even number of moves and odd, for the even number of moves: initialize the third tower to the second stack and the second tower to the third stack then print out the order of disks at the source tower (first) using the printinfo method, and print out info about disks at destination tower (second) using the printinfo method. For the odd number of moves: initialize the second tower to the second stack and the third tower to the third stack, print out the order of disks at the source tower (first), and print out info about disks at the destination tower (third) bot using the printinfo method.

Used a For loop with a stop case that J doesn't exceed the number of moves calculated above and split the loop into three different cases, where each case sends out a different parameter to the method **movingOfDisks** (will be discussed below) that is being called. A print sentence declaring that the following data to be printed are after moving/ changes is written followed by print sentences of the destination and source towers depending on the size/ number of disks in the towers.

movingOfDisks method moves disks between two towers, using parameters that receives two stacks, the source of the disk moving, and its destination as a string. A variable K of type integer is used to save disk numbers that are pushed/popped out of/in towers. If case is used to separate the method into four different cases each depending on the fullness of said tower, or if the peek of the said tower is > that peek of the other tower. Depending on the case that fits the situation, a disk is removed from the said tower and saved into variable K to be later on pushed into the other tower, then declare the moving/ change from and to said towers using print sentences. Its **big O notation is O(1).**

```java
public class Stacks<L>{
    class node<L>{
        public L Data;
        public node<L>Next;
        public node(){//default constructor
            Next=null;//pointer set to null
            Data=null;/*data in nodes set to null */ }//end of default constructor
        public node(L GI){//parameterized constructor with GI as value
            Data=GI;//value is assigned to data
            Next=null;/*pointer set to null */ }//end of method node(L GI)
    }//end of parameterized constructor
    private node<L>Top;//top pointer(that points to the top of stack)
    private int Size;//AKA counter, checks size of stack
    public Stacks(){//stacks constructor
        Top=null;//top pointer set to null, initializes data
        Size=0;/*size set to zero as stack is empty is always empty at the beginning */ }//end of stacks constructor
    public boolean isEmpty(){//method that checks if stack is empty by checking if top is null
        return Top==null; }//end of method isEmpty
    /*another way to check if stack is empty is by using size:
    public boolean isEmpty(){
        return size==0; } */
    public L Pop(){//method used to remove a node
        /*rule of method: if stack not empty
        if(top !=null){ */
        L cs=Top.Data;//take data from top of data as assign it to cs
        Top=Top.Next;//assign top to the data pointed to from the top's next
        Size--;//decrement size since we are removing an element
        return cs;//return data after changes, }else return null;
    }//end of method pop
    public int Size(){//method used to check the size of stack and return it to user
        return Size; }//end of method size
    public L Peek(){//method that returns data at the top of stack without making changes to it
        /*rule of method: if stack not empty
        if(top !=null){ */
        L cs=Top.Data;//takes data from top and assign it to cs
        return cs;// }else return null;
    }//end of method peek
    public void Push(L cs){//Push(item) method, inserts cs of type L
        /* rule of method: stack not full
        if (stack.Full)
            return null;
        else{ */
        node<L>Temporary=new node<L>(cs);//assign cs as temporary(new element)
        Temporary.Next=Top;//make new element(temporary point to the top of stack)
        Top=Temporary;
        Size ++;//size increments by 1 after adding an element
        // }
    }// end of method push
}//end of class stacks
```

For class Stack, we used the same code/ class used in Part 1 (class Stacks, Dequeue, Queue) with no changes.

```java
package palindrome;
public class Queue<L>{//as linkedlist not array
    class node<L>{
        public L Data;
        public node<L>Next;
        public node(){ //default constructor
            Data=null;//data in nodes set to null
            Next=null;//pointer set to null
        }//end of default constructor
        public node(L GI){//parameterized constructor with GI as value
            Next=null;//pointer set to null
            Data=GI;//value is assigned to data
        }//end of method node(L GI)
    }//end of parameterized constructor
    private node<L>tail,head;//head responsible for deleting elements, tail responsible for adding elements
    private int Size;//AKA counter, checks size of stack
    public Queue(){//Queue constructor
        tail=head=null;//initializes data by setting head and tail to null
        //can be rephrased as: head=null; tail=null;
        Size=0;//size set to zero as queue is empty is always empty at the beginning
    }//end of Queue constructor
    public void Enqueue(L cs){//method used to add a node
        if(tail==null){//if queue is empty
            head=tail=new node<L>(cs);//since queue is empty, added element will be pointed as head and tail
        }//end of if case
        else{//if queue is not empty
            tail.Next=new node<L>(cs);//new node made
            tail=tail.Next;//new node is assigned as tail
        }//end of else case
        Size++;//size increments everytime a node is added
    }//end of method enqueue
    public boolean isEmpty(){/*method that checks if queue is empty by checking if head and tail are null:
        head=tail=null;
        or by checking if size is zero: */
        return Size==0; }//end of method isEmpty
    public L Dequeue(){//method used to remove a node
        if(Size==0)//check if queue is empty
            return null;
        L SE=head.Data;//if not empty, take data from head and assign to SE
        head=head.Next;//change head pointer to data in previous head's next
        Size--;//decrement size since we are removing an element
        if(Size==0){//if size becomes 0, it means we removed the last element in queue
            tail=null; }//so tail will=null since queue is empty now
        return SE;//return data after changes
    }//end of method dequeue
    public int Size(){//method used to check the size of queue and return it to user
        return Size; }//end of method size
}//end of class queue
```

For class Queue, we used the same code/ class used in Part 1 (class Stacks, Dequeue, Queue) with no changes.

```java
package palindrome;
import java.util.Scanner;
public class Palindrome{
    public static void main(String[]args){
        Stacks<Character>S=new Stacks<>();
        Queue<Character>Q=new Queue<>();
        Scanner J=new Scanner(System.in);//imported a scanner package to use scanner and take input from user
        System.out.println("Enter string to check if it is a palindrome:");
        String K=J.nextLine();//input from user is saved in J as k reads off next line
        boolean P=true;//assume that p is palindrome
        for(int M=0;M<K.length();M++){//for loop with condition that M doesnt exceed length of K
            S.Push(K.charAt(M));//stack S inserts input from user to reverse the word/string
            Q.Enqueue(K.charAt(M));}//queue Q inserts input from user to keep it without reversing the word
            while(!Q.isEmpty()){//while loop as long as Q queue isnt empty
                Character C1=Q.Dequeue();//dequeue/ remove word/string from the queue Q and save it as C1
                Character C2=S.Pop();//remove te word/ string from the stack S  and save it as C2
                if(C1!=C2){//compare output from the stack and queue
                    //if case if they arent palindromes
                    System.out.println("String "+K+" is NOT a palindrome");//print out not palindrome message
                    P=false;//since they arent palindromes, p(boolean)is false
                    break;//no else case
                }//end of if case
            }//end of while loop
        if(P)//if p is true, enter
            System.out.println("String "+K+" IS a palindrome");//print out is palindrome message
    }//end of main method
}//end of class
```

For **class palindrome**, we created a stack and queue named S and Q. Used scanner import to get input from the user for a string that is saved as J as K reads off the next line. We assumed at P (boolean) is true, otherwise would be false if not palindrome, and used a for loop with condition that M doesn't exceed K's length.

In the for loop, stack S inserts input from the user to reverse the word/string as queue Q inserts input from the user to keep it without reversing the word. In the while loop (with the condition that Q is not empty), Q dequeues/ removes the word/string from the queue Q and saves it as C1 and S removes the word/ string from the stack S and saves it as C2. C1 and C2 are compared in the if statement/ case, if they are equal/ palindrome, a " is a palindrome " message is printed, if not a " NOT a palindrome " message is printed.

The palindrome class has a **big O notation of O(n)**.

Results

<u>Outputs:</u>

Output for part 1 (class stacks, dequeue, queue)

Output - CS210a (run)

```
run:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output for Tower of Hanoi

Number of disks: 3

```
run:
Enter the number of disks:
3
Before moving disks:
Source Tower (FIRST) data is:
1
2
3
Destination of Tower (THIRD) data is:
Tower/ rod is empty
Moved disk 1 from Tower 3 to Tower 1
Moved disk 2 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 3 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 2 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
After moving disks:
Source Tower data is
Tower/ rod is empty
destination Tower data is
1
2
3
BUILD SUCCESSFUL (total time: 2 seconds)
```

Number of disks: 4

```
run:
Enter the number of disks:
4
Before moving disks:
Source Tower (FIRST) data is:
1
2
3
4
Destination of Tower (THIRD) data is:
Tower/ rod is empty
Moved disk 1 from Tower 3 to Tower 1
Moved disk 2 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 3 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 2 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
Moved disk 4 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 2 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 3 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
Moved disk 2 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
After moving disks:
Source Tower data is
Tower/ rod is empty
destination Tower data is
1
2
3
4
BUILD SUCCESSFUL (total time: 2 seconds)
```

Number of disks: 5

```
Enter the number of disks:
5
Before moving disks:
Source Tower (FIRST) data is:
1
2
3
4
5
Destination of Tower (THIRD) data is:
Tower/ rod is empty
Moved disk 1 from Tower 3 to Tower 1
Moved disk 2 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 3 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 2 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
Moved disk 4 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 2 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 3 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
Moved disk 2 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 5 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 2 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
Moved disk 3 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 2 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 4 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
Moved disk 2 from Tower 2 to Tower 1
Moved disk 1 from Tower 3 to Tower 2
Moved disk 3 from Tower 3 to Tower 1
Moved disk 1 from Tower 2 to Tower 1
Moved disk 2 from Tower 3 to Tower 2
Moved disk 1 from Tower 3 to Tower 1
After moving disks:
Source Tower data is
Tower/ rod is empty
destination Tower data is
1
2
3
4
5
BUILD SUCCESSFUL (total time: 2 seconds)
```

Output of palindrome:

```
run:
Enter string to check if it is a palindrome:
madam
String madam IS a palindrome
BUILD SUCCESSFUL (total time: 3 seconds)
```

```
run:
Enter string to check if it is a palindrome:
moammad
String moammad is NOT a palindrome
BUILD SUCCESSFUL (total time: 5 seconds)
```

```
run:
Enter string to check if it is a palindrome:
mom
String mom IS a palindrome
BUILD SUCCESSFUL (total time: 3 seconds)
```

Conclusion

In this project, we learned how to manage to create a code using multiple method's implementations of classes enqueue, dequeue, and stacks which have the methods pop, push, enqueue, dequeue, peek, top… etc. The Tower of Hanoi was solved using stacks, unlike common solving that resorts to recursive methods or iterative methods. As we once again, made good use of the class stack we solved and attached for part 1.

Big O notations for each class and method were mentioned depending on the loops used, the loop's conditions, and any nested loops that have been used in our codes.

For the palindrome class, we used both stacks and queues to help us reverse (using stack) the string inserted/ input from the user while keeping a copy of the string unreversed to compare them.

We expanded our code to be error-free and used a set of rules and code standards for example. Also, we used proper commenting as our code will be read by various people, this helps whoever gets a hold of the code to understand better why e certain line of code was written and if there are any alternative lines of code to be used in that specific line.

References

1. https://stackoverflow.com/questions/43378044/iterating-through-nodes-in-a-push-method-for-a-linked-list-stack
2. https://www.javatpoint.com/java-linkedlist
3. https://www.youtube.com/watch?v=311qJJHiQjU
4. Class slides
5. https://www.kau.edu.sa/GetFile.aspx?id=152091&fn=Stacks_Queues_imp_smr.pptx
6. https://www.geeksforgeeks.org/java-program-to-write-into-a-file/
7. https://xperti.io/blogs/java-coding-best-practices/
8. https://www.sanfoundry.com/java-program-implement-solve-tower-of-hanoi-using-stacks/#:~:text=The%20Tower%20of%20Hanoi%20is,thus%20making%20a%20conical%20shape.
9. https://www.chegg.com/homework-help/questions-and-answers/write-java-program-solve-towers-hanoi-problem-tower-size-n-using-iteration-following-stack-q58360695
10. https://stackoverflow.com/questions/40455042/determining-a-palindrome-using-stack-and-queue-in-java
11. https://beginnersbook.com/2014/01/java-program-to-check-palindrome-string/
12. https://www.geeksforgeeks.org/check-whether-the-given-string-is-palindrome-using-stack/