PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

CSE 2001 - Data Structures and Algorithms

MODULE-2 LAB SHEET (Stack & Queue using Linked List and Recursion)

Implementation of Stack using Linked List

```java
import java.util.*;
public class StackLinkedList {
    Node top;
    public StackLinkedList() {
        this.top = null;
    }
    public boolean isEmpty() {
        return top == null;
    }
    public void pushItem(int item) {
        Node newNode = new Node();
        if (newNode == null)
            System.out.println("Stack: Overflow");
        else{
            newNode.data = item;
            newNode.next = top;
            top = newNode;
        }
    }
    public void popItem() {
        if (isEmpty()){
            System.out.println("Stack Underflow");
            return;
        }
        System.out.println("Data poped from top: "+top.data);
        top = top.next;
    }
    public void stackPeek() {
        if (isEmpty())
            System.out.println("Nothing in stack, stack is empty");
        else
            System.out.println("Element at peek is: "+top.data);
    }
    public void displayStack() {
        if (isEmpty())
```

```
                System.out.println("Stack is Empty");
            else{
                Node trav = top;
                System.out.println("Stack is :");
                while(trav!=null){
                    System.out.println("| "+trav.data+" |");
                    trav = trav.next;
                }
            }
        }
        public static void main(String[] args) {
            StackLinkedList stack = new StackLinkedList();
            Scanner sc = new Scanner(System.in);
            int op, item;
            System.out.println("Stack using link list");
            while (true) {
                System.out.print("1.Push operation\n2.Pop operation\n3.Get the
                stack peek\n4.Display Stack\n5.Exit\n");
                op = sc.nextInt();
                switch (op) {
                    case 1:
                        System.out.print("Enter value: ");
                        item = sc.nextInt();
                        stack.pushItem(item);
                        stack.displayStack();
                        break;
                    case 2:
                        stack.popItem();
                        stack.displayStack();
                        break;
                    case 3:
                        stack.stackPeek();
                        break;
                    case 4:
                        stack.displayStack();
                        break;
                    case 5:
                        System.exit(1);
                }
            }
        }
    }
```

Implementation of Queue using Linked List

```
import java.util.Scanner;
class Node{
        int data;
        Node next;
}
public class QueueLinkedList {
        Node front, rear;
        private int size; // Number of elements
        public QueueLinkedList() {
                front = null;
                rear = null;
        }
        public boolean isEmpty() {
                return (front == null);
        }
        //Dequeue from the front.
        public void dequeue() {
                if (isEmpty()){
                        System.out.println("Queue is empty deletion not possible");
                        rear = null;
                        return;
                }
                int data = front.data;
                front = front.next;
                System.out.println(data+" deleted from the queue");
        }
        //Enqueue operation at the rear end.
        public void enqueue(int data) {
                Node oldRear = rear;
                rear = new Node();
                rear.data = data;
                rear.next = null;
                if (isEmpty())
                        front = rear;
                else
                        oldRear.next = rear;
                System.out.println(data+" added to the queue");
        }
        public void queueSize() {
                int size = 0;
                if (isEmpty()){
                        System.out.println("Queue is empty");
                        return;
                }
                Node trav = front;
                while(trav!=rear){
                        size++;
                        trav = trav.next;
        }
        size++;
        System.out.println("Current size of queue is: "+size);
}
public void display() {
        if (isEmpty()){
                System.out.println("Queue is empty");
                return;
        }
        Node trav = front;
        System.out.print("Queue is: ");
        while(trav!=null){
                System.out.print(trav.data+" ");
                trav = trav.next;
                //for (Node trav = front; trav.next!=null;trav = trav.next)
        }
        System.out.println();
        }
public static void main(String a[]) {
        QueueLinkedList queue = new QueueLinkedList();
        Scanner sc = new Scanner(System.in);
        int op, item;
        System.out.println("Queue using link list");
        while(true) {
                System.out.print("1.Enqueue operation\n2.Dequeue
                operation\n3.Current size of queue\n4.Display queue\n5.Exit\n");
                op = sc.nextInt();
                switch(op) {
                        case 1:
                                System.out.print("Enter value: ");
                                item = sc.nextInt();
                                queue.enqueue(item);
                                queue.display();
                                break;
                        case 2:
                                queue.dequeue();
                                queue.display();
                                break;
                        case 3:
                                queue.queueSize();
                                break;
                        case 4:
                                queue.display();
                                break;
                        case 5:
                                System.exit(1);
                }
        }
}}
```

RECURSION

In **any programming language** it is possible for **functions** to call themselves.

A function is called recursive if a statement within the body of the function **calls** the same function. This is also known as **circular definition** or **self-reference**.

There is a distinct difference between the **normal** and **recursive** functions.

- A **normal function** will be invoked by another function.
- A **recursive function** is invoked by itself **directly** or **indirectly** as long as the given condition is satisfied.

Recursive functions are useful while constructing the **data structures** like stacks, queues, linked lists and trees.

Every **recursive function** must satisfy the following two criteria or properties.

1. Base criteria: There must be at least one base criteria or condition in the recursion process, such that, when this condition is met the function stops calling itself recursively.
2. Recursive step: The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

A recursive function that satisfies the above two properties is said to be a **well-defined recursive function**.

In a recursive program, the solution can have one or more base cases (base criteria) . A base case is a terminating scenario that does not use recursion to produce the result/solution to the problem.

To find factorial of a given number

```
import java.util.Scanner;
public class Factorial {
	int fact(int n) {
		if (n==0 || n==1)
			return 1;
		else
			return n*fact(n-1);
	}
	public static void main(String[] args) {
		int n;
		Factorial t = new Factorial();
		System.out.print("Enter the number to compute factorial: ");
		Scanner sc = new Scanner(System.in);
		n = sc.nextInt();
		System.out.print("Factorial of " + n + " is: " + t.fact(n));
	}
}
```

Towers of Hanoi Puzzle

It is a classic mathematical puzzle (inspired from the puzzle **Tower of Brahma**) used in computer programming. In this there are three rods, one of the rods (called source) contains a stack of disks of different sizes.
The objective of the puzzle is to move the entire stack present in the source rod to another rod which is called the destination rod.

While moving the disks, one has to follow the following rules:

- Only one disk can be moved at a time.
- While moving a disk, only the topmost disk can be moved at a time.
- Larger disks can't be placed on top of the smaller disks.

The solution of moving **n** disks from source to destination can be divided into the three following steps:
Step-1:

- Moving top **n - 1** disks from source rod to temporary rod using destination rod as temporary.
- After Step-1, we have only **1** disk i.e the largest disk in the source node and the destination node will be empty.

Step-2:

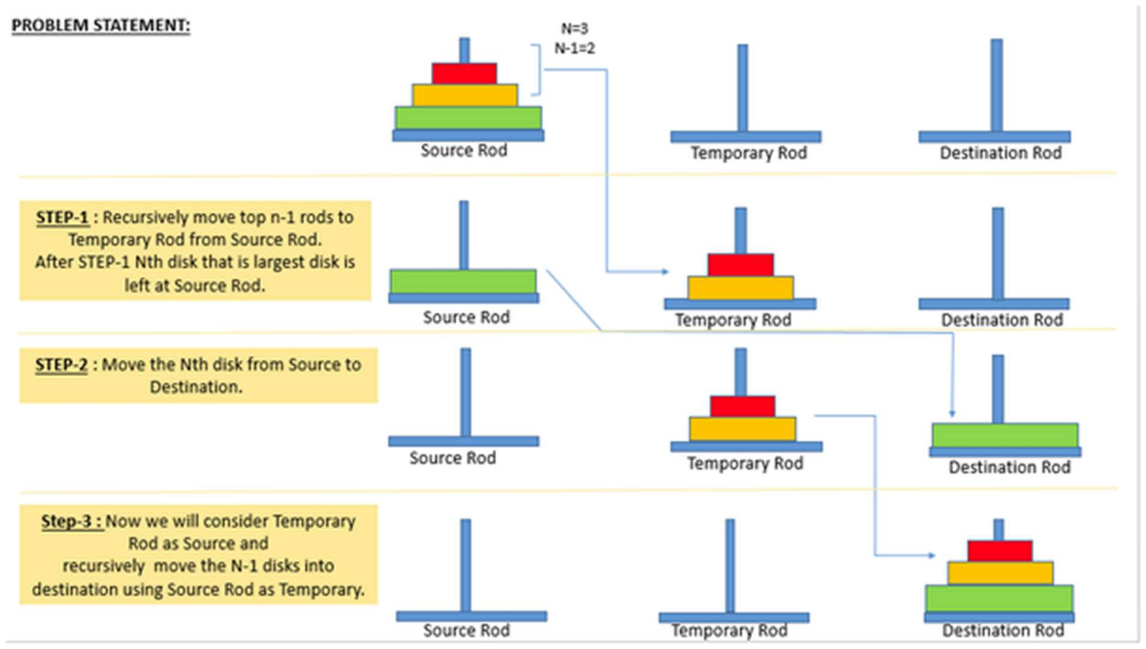
- Move the nth disk from source to destination.
- After Step-2 , the source rod is empty and the biggest disk is moved to destination.

Step-3:

- Move the **n - 1** disks from the temporary rod to the destination rod using source rod (which is empty After Step-2) as the intermediate node.

The base condition here is if the number of disks is **1** then we need to just move that disk from source to destination.

The three steps can be easily understood using the following diagram:


PROBLEM STATEMENT:
N=3
N-1=2
Source Rod
Temporary Rod
Destination Rod
STEP-1 : Recursively move top n-1 rods to Temporary Rod from Source Rod. After STEP-1 Nth disk that is largest disk is left at Source Rod.
Source Rod
Temporary Rod
Destination Rod
STEP-2 : Move the Nth disk from Source to Destination.
Source Rod
Temporary Rod
Destination Rod
Step-3 : Now we will consider Temporary Rod as Source and recursively move the N-1 disks into destination using Source Rod as Temporary.
Source Rod
Temporary Rod
Destination Rod

```java
import java.util.Scanner;
public class TowerOfHanoi {
	public void towerOfHanoiMethod(int n, char a, char b, char c){
		if (n==1){
			System.out.println("Move disk - 1 from pole " + a + " to " + c);
			return;
		}
		towerOfHanoiMethod(n-1, a, c, b);
		System.out.println("Move disk - "+n+ " from pole " + a + " to " + c);
		towerOfHanoiMethod(n-1, b, a, c);
	}
	public static void main(String[] args) {
		int n;
		TowerOfHanoi t = new TowerOfHanoi();
		System.out.print("Enter number of disks : ");
		Scanner sc = new Scanner(System.in);
		n = sc.nextInt();
		t.towerOfHanoiMethod(n, 'A', 'B', 'C');
	}
}
```