//-------shell sort--------//

package mypack;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ShellSortExample {

public static void shellSort(int[] arr, int n) {

int pass = 1;

for (int gap = n / 2; gap > 0; gap /= 2) {

for (int i = gap; i < n; i++) {

int temp = arr[i]; // storing gap element in temp

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {

arr[j] = arr[j - gap]; // moving element if it is greater

}

arr[j] = temp; // placing temp element at appropriate position

}

System.out.println("\nGap = " + gap + "\nPass " + pass + " : ");

for (int i = 0; i < n; i++) {

System.out.print(" " + arr[i] + " ");

}

pass++;

}

}

public static void main(String[] args) {

int size = 15;

int[] myArray = new int[size];

// Opening the file in read mode

try {

Scanner infile = new Scanner(new File("D:\\MCA\\Eclipse Java\\ADS\\src\\mypack\\num.txt"));

// Reading the values from the file and storing in array

for (int i = 0; i < size; ++i) {

myArray[i] = infile.nextInt();

}

// Closing the file

infile.close();

} catch (FileNotFoundException e) {

System.err.println("Failed to open file for reading.");

return;

}

System.out.println("\nArray before sorting: ");

for (int i = 0; i < size; i++) {

System.out.print(myArray[i] + " ");

}

shellSort(myArray, size);

System.out.println("\nArray after sorting: ");

for (int i = 0; i < size; i++) {

System.out.print(myArray[i] + " ");

}

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-------insertion sort-------//

import java.util.Scanner;

public class InsertionSortExample {

public static void insertionSort(int[] list, int n) {

int key, hold, walker, i;

int flag=0;

for (key = 1; key < n; key++) {

walker = key - 1;

hold = list[key];

while (walker >= 0 && hold < list[walker]) {

list[walker + 1] = list[walker];

walker--;

flag = 1;

}

list[walker + 1] = hold;

if (flag == 0) {

break;

}

System.out.print("\nPass : " + key + " - ");

for (i = 0; i < n; i++) {

System.out.print(list[i] + " ");

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the array size: ");

int size = scanner.nextInt();

int[] list = new int[size];

System.out.println("Enter the array elements: ");

for (int i = 0; i < size; i++) {

list[i] = scanner.nextInt();

}

System.out.print("\n\nUnsorted Array: ");

for (int i = 0; i < size; i++) {

System.out.print(list[i] + " ");

}

System.out.println("\n\nSorted array:\n");

insertionSort(list, size);

System.out.print("\n\nSorted Array: ");

for (int i = 0; i < size; i++) {

System.out.print(list[i] + " ");

}

scanner.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//------bubble Sort----------//

package practical1;

import java.util.Scanner;

public class BubbleSort {

public static void bsort(int arr[], int size)

{

for (int i=0; i<size-1; i++) {

for (int j=0; j<size-i-1; j++) {

if (arr[j]>arr[j+1]) {

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

public static void printArray(int arr[], int size) {

System.out.println("the sorted array is: ");

for (int i=0; i<size; i++) {

System.out.print(arr[i]+" ");

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

Scanner sc = new Scanner(System.in);

System.out.print("Enter the size of an array: ");

int size = sc.nextInt();

int arr[] = new int[size];

System.out.println("Enter the elements of an array: ");

for (int i=0; i<size; i++)

{

arr[i]=sc.nextInt();

}

System.out.println("the unsorted array is: " );

for (int i=0; i<size; i++) {

System.out.print(arr[i]+" ");

}

System.out.println();

bsort(arr, size);

printArray(arr, size);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-------Selection Sort----------//

package practical1;

import java.util.Scanner;

public class SelectionSort {

public static void selectionSort(long[] list) {

int size = list.length;

for (int hold = 0; hold < size - 1; hold++) {

int pos = hold;

// Find the smallest element in the unsorted part of the array

for (int walker = hold + 1; walker < size; walker++) {

if (list[walker] < list[pos]) {

pos = walker;

}

}

// Swap the found minimum element with the first element if necessary

if (hold != pos) {

long temp = list[pos];

list[pos] = list[hold];

list[hold] = temp;

}

// Print the array after each pass

System.out.print("\nPass " + (hold + 1) + ": ");

for (int i = 0; i < size; i++) {

System.out.print(list[i] + " ");

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the array size: ");

int size = scanner.nextInt();

long[] list = new long[size];

System.out.println("Enter the array elements:");

for (int i = 0; i < size; i++) {

list[i] = scanner.nextLong();

}

System.out.print("\nUnsorted Array: ");

for (long element : list) {

System.out.print(element + " ");

}

System.out.println("\n\nSorting array using selection sort:");

selectionSort(list);

System.out.print("\n\nSorted Array: ");

for (long element : list) {

System.out.print(element + " ");

}

scanner.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-------Linear Search--------//

package practical1;

import java.util.Scanner;

public class LinearSearch {

public static int LinSearch(int arr[], int elem)

{

for (int i=0; i<arr.length; i++) {

if (arr[i]==elem) {

return i;

}

}

return -1;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

int arr[]= {2,4,6,8,10,12,14,16};

Scanner sc = new Scanner(System.in);

System.out.print("enter the element to search: ");

int elem = sc.nextInt();

int result = LinSearch(arr, elem);

if (result!=-1) {

System.out.println("Element "+ elem + " found at index " + result);

}

else {

System.out.println("Element not found");

}

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-----Binary Search------//

package practical1;

import java.util.Arrays;

import java.util.Scanner;

public class BinarySearch {

public static int binSearch(int arr[], int elem) {

int low = 0;

int high = arr.length - 1;

while (low<=high) {

int mid = (low+high)/2;

if (arr[mid]==elem) {

return mid;

}

else if (arr[mid]<elem) {

low = mid + 1;

}

else {

high = mid -1;

}

}

return -1;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

int arr[] = {2,4,6,8,10,12, 1,21,3};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));

Scanner sc = new Scanner (System.in);

System.out.print("Enter the element to search: ");

int elem = sc.nextInt();

int result = binSearch(arr, elem);

if (result!=-1) {

System.out.println("Element " + elem + " found at index "+ result);

}

else {

System.out.println("Element not found");

}

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//--------HashTable--------//

package practical1;

import java.util.Arrays;

public class HashTable {

private static final int SIZE = 10;

private int[] hashTable;

// Constructor to initialize hash table

public HashTable() {

hashTable = new int[SIZE];

Arrays.fill(hashTable, -1); // -1 indicates an empty slot for probing

}

// Insert key using Linear Probing

public void insertLinearProbing(int key) {

int index = key % SIZE;

while (hashTable[index] != -1) {

System.out.println("\nCollision found at address " + index + " for " + key);

System.out.println("Searching next empty slot using linear probing!");

index = (index + 1) % SIZE; // Find the next available slot

}

System.out.println("\nNo collision at address " + index + " for " + key);

hashTable[index] = key;

}

// Display the hash table for probing methods

public void displayProbing() {

System.out.println();

for (int i = 0; i < SIZE; i++) {

if (hashTable[i] != -1) {

System.out.println(" Index " + i + " : " + hashTable[i]);

} else {

System.out.println(" Index " + i + " : NULL");

}

}

}

public static void main(String[] args) {

int[] keys = {25, 36, 47, 55, 63, 75, 88, 92}; // keys to insert

HashTable ht = new HashTable();

System.out.println("\nHashing using Linear Probing:");

for (int key : keys) {

ht.insertLinearProbing(key);

}

ht.displayProbing();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//------Digit extraction(hashing method)-------//

import java.util.Arrays;

public class HashTableDigitExt {

private static final int SIZE = 10; // size of the hash table

private int[] hashTable; // Array to store the hash table

// Constructor to initialize hash table

public HashTableDigitExt() {

hashTable = new int[SIZE];

Arrays.fill(hashTable, -1); // -1 indicates an empty slot for probing

}

// Insert key into hash table

public void insert(int key, int digitPosition) {

int numDigits = (int) Math.log10(key) + 1; // Count total number of digits in the key

if (digitPosition > numDigits || digitPosition <= 0) {

System.out.println(" Invalid digit position!");

return;

}

// Extract the digit at the desired position and assign to index (key/10^(numDigits-digitPosition))%10

int index = (key / (int) Math.pow(10, numDigits - digitPosition)) % 10;

while (hashTable[index] != -1) {

System.out.println("\nCollision found at address " + index + " for " + key);

System.out.println("Searching next empty slot using linear probing!");

index = (index + 1) % SIZE; // Find the next available slot

}

System.out.println("\nNo collision at address " + index + " for " + key);

hashTable[index] = key;

}

// Display the hash table

public void display() {

for (int i = 0; i < SIZE; i++) {

if (hashTable[i] != -1) {

System.out.println(" Index " + i + " : " + hashTable[i]);

} else {

System.out.println(" Index " + i + " : NULL");

}

}

}

public static void main(String[] args) {

int[] keys = {12345, 67890, 13579, 24680, 98765, 43210, 56789, 10234}; // keys to insert

int n = keys.length;

int digitPosition = 3;

System.out.print("Keys: ");

for (int key : keys) {

System.out.print(key+" " );

}

System.out.println();

HashTableDigitExt ht = new HashTableDigitExt();

for (int key : keys) {

ht.insert(key, digitPosition);

}

System.out.println("\n\nHash Table using Digit Extraction Hashing (place of digit is " + digitPosition + " )\n");

ht.display();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//------Stack using Array--------//

import java.util.Scanner;

class MyStack{

int maxSize;

int top;

int[] stackArray;

// constructor to initialize the stack

public MyStack(int size) {

maxSize = size;

stackArray = new int[maxSize];

top = -1;

}

//push operation

public void push(int value) {

if (isFull()) {

System.out.println("Stack is full! can not push more element!");

}

else {

stackArray[++top] = value;

System.out.println(value+" is push in stack");

}

}

//pop operation

public int pop() {

if(isEmpty()) {

System.out.println("Stack is empty! Can not pop element!");

return -1;

}

else {

return stackArray[top--];

}

}

// Peek operation

public int peek() {

if (isEmpty()) {

System.out.println("Stack is empty!");

return -1;

}

else {

return stackArray[top];

}

}

// check if the stack is full

public boolean isFull() {

return (top == maxSize-1);

}

// Check if the stack is empty

public boolean isEmpty() {

return top==-1;

}

void displayAll() {

if(isEmpty()) {

System.out.println("\nStack is empty! No elements to display! \n");

}

else {

System.out.println("Elements in the stack are: ");

for(int i=top; i>=0;i--) {

System.out.println(stackArray[i]);

}

System.out.println("");

}

}

}

public class StackDemo {

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

MyStack st = new MyStack(5);

int val;

int choice;

do {

System.out.println("\n1.Push");

System.out.println("2.Pop");

System.out.println("3.Peek");

System.out.println("4.Display");

System.out.println("5.Exit");

System.out.println("Enter your Choise:");

choice = sc.nextInt();

switch (choice) {

case 1:

System.out.println("Enter a value to be pushed :");

val = sc.nextInt();

st.push(val);

break;

case 2:

val = st.pop();

System.out.println("Popped element is : "+val);

break;

case 3:

System.out.println("Peeked element is "+st.peek());

break;

case 4:

st.displayAll();

break;

case 5:

System.out.println("Exiting the program.");

break;

default:

System.out.println("Wrong Choice!\n");

}

}while(choice!=5);

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//--------Queue using Array--------//

import java.util.Scanner;

class MyQueue{

int queueSize;

int arr[];

int q\_front;

int q\_rear;

public MyQueue(int size) {

queueSize = size;

arr = new int[queueSize];

q\_front = 0;

q\_rear = -1;

}

public void enqueue(int value) {

if(isFull()) {

System.out.println("Queue is full! Cannot enqueue.");

}

else {

q\_rear = q\_rear + 1;

arr[q\_rear] = value;

}

}

public int dequeue() {

if(isEmpty()) {

System.out.println("Queue is empty! Cannot dequeue.");

return -1;

}

else {

int q\_element = arr[q\_front];

q\_front = q\_front + 1;

return q\_element;

}

}

public boolean isEmpty() {

return (q\_rear == -1 || q\_front>q\_rear);

}

public boolean isFull() {

return (q\_rear == queueSize - 1);

}

public int size() {

return q\_rear - q\_front + 1;

}

public void displayAllElement() {

if(isEmpty()) {

System.out.println("\nQueue is empty! No element in the queue.");

}

else {

System.out.println("Elements in the queue are: ");

for(int i = q\_front; i<=q\_rear;i++) {

System.out.println(arr[i]+" ");

}

System.out.println(" ");

}

}

}

public class QueueDemo {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the size of the queue: ");

int size = sc.nextInt();

MyQueue queue = new MyQueue(size);

int choice, val;

do {

System.out.println("\n1. Enqueue");

System.out.println("2. Dequeue");

System.out.println("3. Check if Queue is Full");

System.out.println("4. Check if queue is Empty");

System.out.println("5. Display All Elements");

System.out.println("6. Display Queue Size");

System.out.println("7. Exit");

System.out.println("\nEnter your choice: ");

choice = sc.nextInt();

switch(choice) {

case 1:

if(!queue.isFull()) {

System.out.println("Enter value to be enqueued: ");

val = sc.nextInt();

queue.enqueue(val);

}

else {

System.out.println("Queue is full! cannot enqueue.");

}

break;

case 2:

val = queue.dequeue();

if(val!=-1) {

System.out.println("Dequeued element is: "+val);

}

break;

case 3:

if(queue.isFull()) {

System.out.println("Queue is full.");

}

else {

System.out.println("Queue is not full.");

}

break;

case 4:

if(queue.isEmpty()) {

System.out.println("Queue is empty.");

}

else {

System.out.println("Queue is not empty.");

}

break;

case 5:

queue.displayAllElement();

break;

case 6:

System.out.println("Current size of the queue: "+queue.size());

break;

case 7:

System.out.println("Exiting the program.");

break;

default:

System.out.println("Wrong choice! Please try again.");

}

}while(choice!=7);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//------Circular Queue using array-------//

import java.util.Scanner;

class CircularQueue{

int QUEUESIZE;

int arr[];

int q\_front;

int q\_rear;

public CircularQueue(int size) {

QUEUESIZE = size;

arr = new int[QUEUESIZE];

q\_front = -1;

q\_rear = -1;

}

public void enqueue(int val) {

// check for empty queue for adding first element

if(isEmpty()) {

q\_rear=0;

q\_front=0;

}

else {

q\_rear = (q\_rear+1)%QUEUESIZE;

}

arr[q\_rear] = val;

}

public int dequeue() {

int q\_element;

if(q\_front==q\_rear) {

// the only element in the queue

q\_element = arr[q\_front];

q\_rear = -1; // resetting rear for empty queue

q\_front = -1; // reseting front for empty queue

}

else {

// more than one element existing

q\_element=arr[q\_front];

q\_front = (q\_front+1)%QUEUESIZE;

}

return q\_element;

}

public boolean isEmpty() {

if(q\_rear==-1) {

return true;

}

else {

return false;

}

}

public boolean isFull() {

if((q\_rear+1)%QUEUESIZE==q\_front) {

return true;

}

else {

return false;

}

}

public int size() {

if(q\_rear>q\_front) {

return q\_rear-q\_front+1;

}

else {

return QUEUESIZE-q\_rear-q\_front+1;

}

}

public void displayAllElements() {

if(q\_rear == -1) {

System.out.println("No elements to display!");

return;

}

System.out.println("Elements in the queue are: ");

for(int i=q\_front; i!=q\_rear;i=(i+1)%QUEUESIZE) {

System.out.println(arr[i]+ " ");

}

System.out.println(arr[q\_rear]+" ");

System.out.println("");

}

}

public class CircularQueueDemo {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the size of the queue: ");

int size = sc.nextInt();

CircularQueue queue = new CircularQueue(size);

int choice, val;

do {

System.out.println("\n1. Enqueue");

System.out.println("2. Dequeue");

System.out.println("3. Check if Queue is Full");

System.out.println("4. Check if queue is Empty");

System.out.println("5. Display All Elements");

System.out.println("6. Display Queue Size");

System.out.println("7. Exit");

System.out.println("Enter your choice: ");

choice = sc.nextInt();

switch(choice) {

case 1:

if(!queue.isFull()) {

System.out.println("Enter value to be enqueued: ");

val = sc.nextInt();

queue.enqueue(val);

}

else {

System.out.println("Queue is full! cannot enqueue.");

}

break;

case 2:

val = queue.dequeue();

if(val!=-1) {

System.out.println("Dequeued element is: "+val);

}

break;

case 3:

if(queue.isFull()) {

System.out.println("Queue is full.");

}

else {

System.out.println("Queue is not full.");

}

break;

case 4:

if(queue.isEmpty()) {

System.out.println("Queue is empty.");

}

else {

System.out.println("Queue is not empty.");

}

break;

case 5:

queue.displayAllElements();

break;

case 6:

System.out.println("Current size of the queue: "+queue.size());

break;

case 7:

System.out.println("Exiting the program.");

break;

default:

System.out.println("Wrong choice! Please try again.");

}

}while(choice!=7);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//------ Singly Linked List using Stack---------//

//java program to implement stack using singly linked list

package stack;

class Node{

int data;

Node next;

Node(int d){

this.data=d;

this.next=null;

}

}

class Stack{

Node head;

Stack(){

this.head=null;

}

Boolean isEmpty() {

return head==null;

}

void push(int d) {

Node newNode = new Node(d);

if(newNode==null) {

System.out.println("\nStack Overflow!");

return;

}

newNode.next = head;

head = newNode;

System.out.println(d + " pushed to stack.");

}

void pop() {

if (isEmpty()) {

System.out.println("\nStack Underflow");

return;

}

else {

Node temp =head;

head= head.next;

temp= null;

}

}

int peek()

{

if(isEmpty()) {

return Integer.MIN\_VALUE;

}

else {

System.out.println("\nStack is empty");

return head.data;

}

}

public void displayStack() {

Node current = head;

if(head==null)

{

System.out.println("]nStack is Empty");

return;

}

System.out.println("\nThe data in Stack");

while(current != null)

{

System.out.print(current.data + " -> ");

current = current.next;

}

System.out.println("null");

System.out.println();

}

}

public class StackSLL {

public static void main(String[] args) {

Stack stack = new Stack();

stack.push(10);

stack.push(20);

stack.push(30);

stack.push(40);

stack.displayStack();

System.out.println("Top element is " + stack.peek());

stack.pop();

stack.pop();

stack.displayStack();

System.out.println("Top element is " + stack.peek());

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-------Circular Linked List-------//

package circularLinkedList;

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

public class CircularLinkedList {

Node head;

Node tail;

public CircularLinkedList() {

head = null;

tail = null;

}

public boolean isEmpty() {

return (head == null);

}

public int count() {

int count = 0;

Node current = head;

do {

current = current.next;

count++;

} while (current != head);

return count;

}

public void display() {

Node current = head;

do {

System.out.print(current.data + " --> ");

current = current.next;

} while (current != head);

System.out.println("Null\n");

}

public void insert(int data) {

Node newNode = new Node(data);

if (isEmpty()) {

head = newNode;

tail = newNode;

tail.next = head;

} else {

tail.next = newNode;

tail = newNode;

tail.next = head;

}

}

public void insertAtHead(int data) {

Node newNode = new Node(data);

if (isEmpty()) {

head = newNode;

tail = newNode;

tail.next = head;

} else {

newNode.next = head;

head = newNode;

tail.next = head;

}

}

public void insertAtPosition(int pos, int data) {

Node newNode = new Node(data);

Node current = head;

Node prev = null;

int totalCount = count();

if (pos == 1) {

newNode.next = head;

head = newNode;

tail.next = head;

} else if (pos > 1 && pos < totalCount + 1) {

if (pos == totalCount + 1) {

tail.next = newNode;

tail = newNode;

tail.next = head;

} else {

for (int i = 1; i < pos; i++) {

prev = current;

current = current.next;

}

prev.next = newNode;

newNode.next = current;

}

}

}

public void deleteAtHead() {

Node temp;

if (isEmpty()) {

System.out.println("the list is empty");

} else {

temp = head;

head = head.next;

temp = null;

tail.next = head;

}

}

public void deleteAtTail() {

if (isEmpty()) {

System.out.println("the list is empty");

} else if (head == tail) {

head = tail = null;

} else {

Node current = head;

while (current.next != tail) {

current = current.next;

}

current.next = null;

tail = current;

tail.next = head;

}

}

public void deleteAtPosition(int pos) {

if (isEmpty()) {

System.out.println("The list is empty!");

} else if (pos == 1) {

if (head == tail) {

head = tail = null;

} else {

Node temp = head;

head = head.next;

tail.next = head;

temp = null;

}

} else if (pos > 1 && pos <= count()) {

Node prev = null;

Node current = head;

for (int i = 1; i < pos; i++) {

prev = current;

current = current.next;

}

if (pos == count()) {

prev.next = null;

prev.next = head;

} else {

prev.next = current.next;

}

} else {

System.out.println("Invalid node position!");

}

}

public void search(int key) {

Node current = head;

int pos = 1;

int flag = 0;

if (isEmpty()) {

System.out.println("the " + key + " is not found! the list is empty!");

}

do {

if (current.data == key) {

System.out.println("The " + key + " is found at " + pos + " position!");

flag = 1;

}

current = current.next;

pos++;

} while (current != head);

if(flag==0) {

System.out.println("The " + key + " is not found!");

}

}

public static void main(String[] args) {

CircularLinkedList cll = new CircularLinkedList();

cll.insert(10);

cll.display();

cll.insert(20);

cll.display();

cll.insertAtHead(5);

cll.display();

cll.insertAtPosition(3, 32);

cll.display();

cll.insertAtPosition(2, 52);

cll.display();

// cll.deleteAtHead();

// cll.display();

// cll.deleteAtHead();

// cll.display();

// System.out.println("deleting at tail");

// cll.deleteAtTail();

// cll.display();

// cll.deleteAtTail();

// cll.display();

// cll.deleteAtPosition(1);

// cll.display();

// cll.deleteAtPosition(4);

// cll.display();

// cll.deleteAtPosition(2);

// cll.display();

cll.search(52);

cll.search(51);

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-------- Doubly linked list---------//

// methods - displayList, isEmpty, insert, insertAtHead, insertAtTail

// insertAtPosition, deleteAtHead, deleteAtTail, deleteAtPosition

// countNodes, search

package doublyLinkedList;

import java.util.Scanner;

class Node {

int data;

Node prev; // add

Node next;

public Node(int data) {

this.data = data;

this.prev = null; // add

this.next = null;

}

}

public class DoublyLinkedList {

// defining the head and tail of a singly linked list

Node head;

Node tail;

public DoublyLinkedList() {

head = null;

tail = null;

}

public boolean isEmpty() {

return (head == null);

}

// function to add a node to the list

public void insert(int data) {

Node newNode = new Node(data);

// if(isEmpty()){}

if (head == null) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail; // add

tail = newNode;

}

}

// function to add a node at head

public void insertAtHead(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

tail = newNode;

} else {

newNode.next = head;

head.prev = newNode; // add

head = newNode;

}

}

// function to add a node at tail

public void insertAtTail(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

// insert node between head and tail

public void insertAtPosition(int pos, int data) {

Node newNode = new Node(data);

// if list is empty

if (head == null) {

head = newNode;

tail = newNode;

}

// insert at head

else if (pos == 1) {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

// Insert at position

else {

// if (pos > 1 && pos <= totalNodes + 1) remove this condition from else

Node current = head;

int currPos = 1;

while (current != null && currPos < pos) {

current = current.next;

currPos++;

}

if (pos < 1 || pos > currPos) {

System.out.println("Invalid node position!");

return;

}

// insert node at tail

if (current == null) {

if (tail == null) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

} // insert at middle

else {

newNode.next = current;

newNode.prev = current.prev;

current.prev.next = newNode;

current.prev = newNode;

}

}

}

// delete node at head

public void deleteAtHead() {

if (head == null) {

System.out.println("doubly linked list is empty!");

} else if (head == tail) {

head = tail = null;

}

// If there's only one node in the list, head and tail both become null

else {

// Otherwise, move the head pointer to the next node

Node temp = head;

head = head.next; // move head to the next node

head.prev = null; // maintain the circular link

temp = null;

}

}

// delete node at tail

public void deleteAtTail() {

if (head == null) {

System.out.println("doubly linked list is empty!");

} else if (head == tail) {

// If there's only one node in the list, head and tail both become null

head = tail = null;

} else {

Node temp = tail;

tail = tail.prev;

tail.next = null;

temp.prev = null;

}

}

// deleting node at head or tail or between head and tail

public void deleteAtPosition(int pos) {

Node current = head;

int totalNodes = countNodes();

// if list is empty

if (head == null) {

System.out.println("Doubly linked list is empty");

}

// if head is to be deleted

else if (pos == 1) {

// single node case

if (head == tail) {

// if there's only one node left

head = null;

tail = null;

} else {

// more than one node

Node temp = head;

head = head.next;

head.prev = null; // add

temp.next = null; // add

}

} else if (pos > 1 && pos <= totalNodes + 1) { // delete node between head and tail

if (pos == totalNodes) {

// if only one node

if (head.next == null) {

head = null;

tail = null;

}

// if one or more nodes

else {

Node temp = tail;

tail = tail.prev;

tail.next = null;

temp.prev = null;

}

} else {

for (int i = 1; i < pos; i++) {

current = current.next;

}

current.prev.next = current.next; // add

current.next.prev = current.prev; // add

current.prev = null; // add

current.next = null; // add

}

} else {

System.out.println("Invalid node position!");

}

}

// function to display the data in the list

public void displayList() {

// Pointing the head to the node called current

Node current = head;

if (head == null) {

System.out.println("The given list is empty!");

return;

}

System.out.println("The data in the given list are: ");

while (current != null) { // change

// Printing each data in the list and next pointer pointing to the next node

System.out.print(current.data + " --> ");

current = current.next;

}

System.out.print("Null");

System.out.println();

}

// Function to count total nodes

public int countNodes() {

int count = 0;

Node current = head;

if (head == null)

return count; // add

while (current != null) { // change

// Increment the count by 1 for each node

count++;

current = current.next;

} // change

return count;

}

// search key in linked list

public void search(int key) {

Node current = head;

int flag = 0;

int pos = 1;

while (current != null) {

if (current.data == key) {

System.out.println("The " + key + " is found at " + pos + " position!");

flag = 1;

}

current = current.next;

pos++;

}

if (flag == 0) {

System.out.println(key + " not found!");

}

}

public static void main(String args[]) {

DoublyLinkedList dll = new DoublyLinkedList();

Scanner sc = new Scanner(System.in);

int data, pos;

int choice;

do {

System.out.println("1. Insert");

System.out.println("2. Insert At Head");

System.out.println("3. Insert At Tail");

System.out.println("4. Insert At Position");

System.out.println("5. Delete At Head");

System.out.println("6. Delete at Tail");

System.out.println("7. Delete At Position");

System.out.println("8. Search for key");

System.out.println("9. Total nodes");

System.out.println("10. Display list");

System.out.println("11. Exit");

System.out.println("\nPlease Enter your choise: ");

choice = sc.nextInt();

switch (choice) {

case 1:

System.out.println("Enter data: ");

data = sc.nextInt();

dll.insert(data);

dll.displayList();

break;

case 2:

System.out.println("Insert Node at head - Enter data: ");

data = sc.nextInt();

dll.insertAtHead(data);

dll.displayList();

break;

case 3:

System.out.println("Insert Node at Tail - Enter data: ");

data = sc.nextInt();

dll.insertAtTail(data);

dll.displayList();

break;

case 4:

System.out.println("Insert Node at Position - Enter position: ");

pos = sc.nextInt();

System.out.println("Enter the data: ");

data = sc.nextInt();

dll.insertAtPosition(pos, data);

System.out.println();

dll.displayList();

break;

case 5:

System.out.println("Delete node at head: ");

dll.deleteAtHead();

dll.displayList();

break;

case 6:

System.out.println("Delete node at tail: ");

dll.deleteAtTail();

dll.displayList();

break;

case 7:

System.out.println("Delete node at position - Enter position: ");

pos = sc.nextInt();

dll.deleteAtPosition(pos);

dll.displayList();

break;

case 8:

System.out.println("Enter a key to search:");

int key = sc.nextInt();

dll.search(key);

break;

case 9:

System.out.println("Total nodes in DLL: " + dll.countNodes());

break;

case 10:

System.out.println("Print all nodes in DLL: ");

dll.displayList();

break;

case 11:

System.out.println("Exiting the program.");

break;

default:

System.out.println("You entered wrong choice!");

}

} while (choice != 11);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//----------Queue Singly Lined List------------//

//Java program to implement queue data structure using linked list

class Node{

int data;

Node next;

Node(int data){

this.data = data;

this.next = null;

}

}

class Queue{

Node front, rear;

Queue(){

front = rear = null;

}

boolean isEmpty() {

return front==null && rear == null;

}

// Function to add an element in the queue

void enqueue(int d) {

// create a new linked list node

Node new\_node = new Node(d);

if(rear==null) {

front = rear = new\_node;

return;

}

rear.next = new\_node;

rear = new\_node;

}

// function to remove element from the queue

void dequeue() {

if(isEmpty()) {

System.out.println("Queue Underflow!");

return;

}

Node temp = front;

front = front.next;

if(front==null) {

rear=null;

}

}

int getFront() {

if(isEmpty()) {

System.out.println("Queue Underflow!");

return Integer.MIN\_VALUE;

}

else {

return front.data;

}

}

int getRear() {

if(isEmpty()) {

System.out.println("Queue Underflow!");

return Integer.MIN\_VALUE;

}

else {

return rear.data;

}

}

public void displayQueue() {

Node current = front;

if(front==null) {

System.out.println("Queue is empty!");

return;

}

System.out.println("The data in Queue: ");

while(current!=null) {

System.out.print(current.data+ " -> ");

current = current.next;

}

System.out.print("null\n");

}

}

public class QueueSLL {

public static void main(String[] args) {

Queue q = new Queue();

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.enqueue(40);

q.enqueue(50);

q.displayQueue();

System.out.println("\nFront element is "+q.getFront());

System.out.println("Rear element is "+q.getRear());

q.dequeue();

q.dequeue();

System.out.println();

q.displayQueue();

System.out.println("\nFront element is "+q.getFront());

System.out.println("Rear element is "+q.getRear());

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//-------Double Ended Queue-------//

package DoubleEndedQueeu;

import java.util.Scanner;

class Node {

int data;

Node next;

Node prev;

public Node(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

class DEQueue {

Node front, rear;

int size;

public DEQueue() {

front = null;

rear = null;

size = 0;

}

// Check if the deque is empty

public boolean isEmpty() {

return front == null;

}

// Get the size of the deque

public int size() {

return size;

}

// Insert at the front

public void insertFront(int data) {

Node newNode = new Node(data);

if (front == null) {

rear = front = newNode;

} else {

newNode.next = front;

front.prev = newNode;

front = newNode;

}

size++;

}

// Insert at the rear

public void insertRear(int data) {

Node newNode = new Node(data);

if (rear == null) {

rear = front = newNode;

} else {

newNode.prev = rear;

rear.next = newNode;

rear = newNode;

}

size++;

}

// Delete from the front

public void deleteFront() {

if (isEmpty()) {

System.out.println("List is empty");

} else {

front = front.next;

if (front == null) {

rear = null;

} else {

front.prev = null;

}

size--;

}

}

// Delete from the rear

public void deleteRear() {

if (isEmpty()) {

System.out.println("List is empty");

} else {

rear = rear.prev;

if (rear == null) {

front = null;

} else {

rear.next = null;

}

size--;

}

}

// Get the front element

public int getFront() {

if (isEmpty()) {

return -1;

}

return front.data;

}

// Get the rear element

public int getRear() {

if (isEmpty()) {

return -1;

}

return rear.data;

}

// Display the deque

public void display() {

if (isEmpty()) {

System.out.println("List is empty");

return;

}

System.out.print("NULL <- ");

Node current = front;

while (current != null) {

System.out.print(current.data + " <-> ");

current = current.next;

}

System.out.println("NULL");

}

// Main method for user interaction

public static void main(String[] args) {

DEQueue dq = new DEQueue();

Scanner sc = new Scanner(System.in);

int data, choice;

do {

System.out.println("\n1. Insert At Front");

System.out.println("2. Insert At Rear");

System.out.println("3. Delete At Front");

System.out.println("4. Delete At Rear");

System.out.println("5. Display List");

System.out.println("6. Exit");

System.out.print("Enter your choice: ");

choice = sc.nextInt();

switch (choice) {

case 1:

System.out.print("Insert at Front - Enter data: ");

data = sc.nextInt();

dq.insertFront(data);

dq.display();

break;

case 2:

System.out.print("Insert at Rear - Enter data: ");

data = sc.nextInt();

dq.insertRear(data);

dq.display();

break;

case 3:

System.out.println("Delete at Front");

dq.deleteFront();

dq.display();

break;

case 4:

System.out.println("Delete at Rear");

dq.deleteRear();

dq.display();

break;

case 5:

System.out.println("Display List:");

dq.display();

break;

case 6:

System.out.println("Exiting the program.");

break;

default:

System.out.println("Wrong choice!");

}

} while (choice != 6);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//--------Priority Queue using Linked List----------//

package PriorityQueue;

//Java program to implement priority queue using linked list

class Node{

int data;

int priority; // lower values indicate higher priority

Node next;

Node(int d, int p){

data = d;

priority = p;

next = null;

}

}

public class PriorityQueue {

Node head;

public PriorityQueue() {

head = null;

}

public void add(int d, int p) {

Node start = head;

Node newNode = new Node(d, p);

if(head==null) {

head = newNode;

return;

}

if(head.priority>p) {

newNode.next = head;

head = newNode;

}

else {

while(start.next != null && start.next.priority<p) {

start = start.next;

}

newNode.next = start.next;

start.next = newNode;

}

}

public Node remove() {

Node temp = head;

head = head.next;

temp = null;

return head;

}

int getHeadData() {

return head.data;

}

public boolean isEmpty() {

return head==null;

}

public void display() {

Node temp = head;

System.out.println();

while(temp!=null) {

System.out.print("Data: "+temp.data+", priority: "+temp.priority+" --> ");

temp=temp.next;

}

System.out.println("Null");

}

public static void main(String[] args) {

PriorityQueue pq = new PriorityQueue();

pq.add(4,1);

pq.add(5,2);

pq.add(6,3);

pq.add(7,0);

System.out.println("Head node data = "+pq.getHeadData());

pq.display();

//remove elements:

pq.remove();

pq.remove();

pq.display();

System.out.println("\nHead node data after removing elements= "+pq.getHeadData());

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//---------Binary Search tree------//

import java.util.Scanner;

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int val) {

this.data = val;

this.left = null;

this.right = null;

}

}

public class BinarySearchTree

{

private Node root;

static int count = 0;

public BinarySearchTree()

{

this.root = null;

}

public static int nodeCounts()

{

return count;

}

// Insert a node into the binary tree

public void insertNode(int val)

{

Node newNode = new Node(val);

if (root == null) {

root = newNode;

count++;

}

else

{

Node trav = root;

Node hold = null;

while (trav != null)

{

hold = trav;

if (val > trav.data)

trav = trav.right;

else if (val < trav.data)

trav = trav.left;

else

{

System.out.println("Duplicate data");

return;

}

}

if (val > hold.data)

hold.right = newNode;

else

hold.left = newNode;

count++;

}

}

// In-order traversal

public void inorder(Node root)

{

if (root != null) {

inorder(root.left);

System.out.print(root.data + " ");

inorder(root.right);

}

}

public void inorder() {

inorder(root);

}

// Pre-order traversal

public void preorder(Node root)

{

if (root != null)

{

System.out.print(root.data + " ");

preorder(root.left);

preorder(root.right);

}

}

public void preorder() {

preorder(root);

}

// Post-order traversal

public void postorder(Node root)

{

if (root != null)

{

postorder(root.left);

postorder(root.right);

System.out.print(root.data + " ");

}

}

public void postorder() {

postorder(root);

}

// Find the smallest element

public void smallest()

{

Node trav = root;

if (trav == null)

{

System.out.println("Tree is empty!");

return;

}

while (trav.left != null)

trav = trav.left;

System.out.println("Smallest Node is: " + trav.data);

}

// Find the largest element

public void largest()

{

Node trav = root;

if (trav == null)

{

System.out.println("Tree is empty!");

return;

}

while (trav.right != null)

trav = trav.right;

System.out.println("Largest Node is: " + trav.data);

}

// Search for a value in the tree

public void search(int val)

{

Node trav = root;

while (trav != null)

{

if (val > trav.data)

trav = trav.right;

else if (val < trav.data)

trav = trav.left;

else

{

System.out.println("Node with data " + val + " is found.");

return;

}

}

System.out.println("Node with data " + val + " is not found!");

}

// Remove a node from the tree

public void removeNode(int val) {

root = removeNodeRecursive(root, val);

if(root!=null)

count--;

}

public Node removeNodeRecursive(Node root, int val) {

if (root == null)

{

System.out.println("Node not found!");

return root;

}

if (val < root.data)

{

root.left = removeNodeRecursive(root.left, val);

}

else if (val > root.data)

{

root.right = removeNodeRecursive(root.right, val);

}

else

{

// Node with one child or no child

if (root.left == null)

return root.right;

else if (root.right == null)

return root.left;

// Node with two children: Get the inorder predecessor

Node trav = root.left;

while (trav.right != null)

trav = trav.right;

root.data = trav.data;

root.left = removeNodeRecursive(root.left, root.data);

}

return root;

}

/\*

private int findLargest(Node root) {

while (root.right != null)

root = root.right;

return root.data;

} \*/

public static void main(String[] args) {

BinarySearchTree bt = new BinarySearchTree();

//45,39,56,12,34,78,32,10,89,54,67,81

Scanner sc = new Scanner(System.in);

int data;

int choice;

System.out.print("\nBinary Search Tree\n\n");

do

{

System.out.print("\n1.Insert Node\n");

System.out.print("2.InOrder Traversal\n");

System.out.print("3.PreOrder Traversal\n");

System.out.print("4.PostOrder Traversal\n");

System.out.print("5.Smallest Node\n");

System.out.print("6.Largest Node\n");

System.out.print("7.Count Nodes\n");

System.out.print("8.Remove Node\n");

System.out.print("9.Search Node\n");

System.out.print("10.Exit\n");

System.out.print("Enter your choice : ");

choice = sc.nextInt();

switch (choice)

{

case 1: System.out.print("\nInsert Node - Enter data : ");

data= sc.nextInt();

bt.insertNode(data);

break;

case 2: System.out.print("\nInOrder Traversal : ");

bt.inorder();

break;

case 3: System.out.print("\nPreOrder Traversal : ");

bt.preorder();

break;

case 4: System.out.print("\nPostOrder Traversal : ");

bt.postorder();

break;

case 5: System.out.print("\nSmallest node is : ");

bt.smallest();

break;

case 6: System.out.print("\nLargest node is : ");

bt.largest();

break;

case 7: System.out.print("\nTotal node count : " + nodeCounts());

break;

case 8: System.out.print("\nRemove node - Enter key : ");

data=sc.nextInt();

bt.removeNode(data);

break;

case 9: System.out.print("\nSearch node - Enter Data : ");

data=sc.nextInt();

bt.search(data);

break;

case 10: System.out.println("Exiting the program.");

break;

default:

System.out.print("\nWrong choice! \n");

} /\*End of switch\*/

}while(choice!=10); /\*End of while\*/

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//------Max Heap--------//

package heap;

import java.util.Scanner;

class Node {

int data;

Node prev, next;

Node(int val) {

this.data = val;

this.prev = null;

this.next = null;

}

}

public class MaxHeap {

Node head, tail;

public MaxHeap() {

this.head = null;

this.tail = null;

}

// Re-heap Up for MaxHeap

private void reheapUp(Node node) {

Node parent = getParent(node);

while (parent != null && node.data > parent.data) {

// Swap the node with its parent

int temp = node.data;

node.data = parent.data;

parent.data = temp;

node = parent;

parent = getParent(node);

}

}

// Re-heap Down for Maxheap

private void reheapDown(Node node) {

while (node != null) {

Node leftChild = node.next;

Node rightChild = (leftChild != null) ? leftChild.next : null;

// If there are no children, break out of the loop

if (leftChild == null) {

break;

}

Node maxChild = leftChild;

if (rightChild != null && rightChild.data > leftChild.data) {

maxChild = rightChild;

}

// If the node is greater than or equal to its largest child, stop

if (node.data >= maxChild.data) {

break;

}

// Swap node with the largest child

int temp = node.data;

node.data = maxChild.data;

maxChild.data = temp;

node = maxChild;

}

}

// Insert a node in the MaxHeap

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

// Reheap up from the last inserted node

reheapUp(newNode);

}

}

// Delete the root node in the MaxHeap

public void delete() {

if (head == null) {

System.out.println("Heap is Empty!!");

return;

}

Node lastNode = tail;

// Move the last node to the root position

head.data = lastNode.data;

// Remove the last Node

if (tail.prev != null) {

tail = tail.prev;

tail.next = null;

} else {

head = null;

}

// Re-heap down from the root

reheapDown(head);

}

// Get the parent of a node in the doubly linked list

private Node getParent(Node node) {

return node.prev;

}

// Print the heap

public void printHeap() {

Node temp = head;

if (temp == null) {

System.out.println("Heap is empty.");

return;

}

while (temp != null) {

System.out.print(temp.data + " ");

temp = temp.next;

}

System.out.println();

}

// Main function with choices

public static void main(String[] args) {

MaxHeap maxHeap = new MaxHeap();

Scanner scanner = new Scanner(System.in);

while (true) {

System.out.println("\nMax Heap Operations Menu:");

System.out.println("1. Insert a value into MaxHeap");

System.out.println("2. Delete the root node from MaxHeap");

System.out.println("3. Print the MaxHeap");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter a value to insert into MaxHeap: ");

int value = scanner.nextInt();

maxHeap.insert(value);

break;

case 2:

System.out.println("Deleting root node from MaxHeap...");

maxHeap.delete();

break;

case 3:

System.out.println("MaxHeap:");

maxHeap.printHeap();

break;

case 4:

System.out.println("Exiting the program...");

scanner.close();

return; // Exit the program

default:

System.out.println("Invalid choice, please try again.");

}

}

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//--------Min Heap----------//

package MinHeap;

import java.util.Scanner;

class Node {

int data;

Node prev, next;

Node(int val) {

this.data = val;

this.prev = null;

this.next = null;

}

}

public class MinHeap {

Node head, tail;

public MinHeap() {

this.head = null;

this.tail = null;

}

// Re-heap Up for MinHeap

private void reheapUp(Node node) {

Node parent = getParent(node);

while (parent != null && node.data < parent.data) { // Reverse the comparison for MinHeap

// Swap the node with its parent

int temp = node.data;

node.data = parent.data;

parent.data = temp;

node = parent;

parent = getParent(node);

}

}

// Re-heap Down for MinHeap

private void reheapDown(Node node) {

while (node != null) {

Node leftChild = node.next;

Node rightChild = (leftChild != null) ? leftChild.next : null;

// If there are no children, break out of the loop

if (leftChild == null) {

break;

}

Node minChild = leftChild;

if (rightChild != null && rightChild.data < leftChild.data) { // Reverse the comparison for MinHeap

minChild = rightChild;

}

// If the node is less than or equal to its smallest child, stop

if (node.data <= minChild.data) { // Reverse the comparison for MinHeap

break;

}

// Swap node with the smallest child

int temp = node.data;

node.data = minChild.data;

minChild.data = temp;

node = minChild;

}

}

// Insert a node in the MinHeap

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

// Reheap up from the last inserted node

reheapUp(newNode);

}

}

// Delete the root node in the MinHeap

public void delete() {

if (head == null) {

System.out.println("Heap is Empty!!");

return;

}

Node lastNode = tail;

// Move the last node to the root position

head.data = lastNode.data;

// Remove the last Node

if (tail.prev != null) {

tail = tail.prev;

tail.next = null;

} else {

head = null;

}

// Re-heap down from the root

reheapDown(head);

}

// Get the parent of a node in the doubly linked list

private Node getParent(Node node) {

return node.prev;

}

// Print the heap

public void printHeap() {

if (head == null) {

System.out.println("Heap is empty.");

return;

}

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " ");

temp = temp.next;

}

System.out.println();

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap();

Scanner sc = new Scanner(System.in);

int choice, data;

do {

System.out.println("\nMinHeap Operations Menu:");

System.out.println("1. Insert Node");

System.out.println("2. Delete Root Node");

System.out.println("3. Print Heap");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

choice = sc.nextInt();

switch (choice) {

case 1:

System.out.print("\nEnter data to insert: ");

data = sc.nextInt();

minHeap.insert(data);

break;

case 2:

System.out.println("\nDeleting root node from MinHeap...");

minHeap.delete();

break;

case 3:

System.out.println("\nCurrent MinHeap: ");

minHeap.printHeap();

break;

case 4:

System.out.println("Exiting the program.");

break;

default:

System.out.println("Invalid choice! Please try again.");

break;

}

} while (choice != 4);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//--------Graph---------//

package Graph;

//java program to represent graph using adjacency Matrix, BFS and DFS

import java.util.Scanner;

public class Graph {

private int[][] adjacencyMatrix; // maintains edges

private int numVertices;

public Graph(int numVertices) {

this.numVertices = numVertices;

adjacencyMatrix = new int[numVertices][numVertices];

}

// method to add an edge to the graph

public void addEdge(int start, int end) {

adjacencyMatrix[start][end] = 1;

adjacencyMatrix[end][start] = 1; // For an undirected graph

}

// Breadth First Search (BFS)

public void bfs(int startVertex) {

boolean[] visited = new boolean[numVertices];

int[] queue = new int[numVertices];

int front = 0, rear = 0;

// Mark the start vertex as visited and enqueue it

visited[startVertex] = true;

queue[rear++] = startVertex;

System.out.println("BFS traversal starting from vertex : "+startVertex+ " : ");

while(front<rear) {

int currentVertex = queue[front++]; // Dequeue

System.out.print(currentVertex+" ");

// endqueue all adjacent unvisited vertices

for(int i=0; i<numVertices; i++) {

if(adjacencyMatrix[currentVertex][i]==1 && !visited[i]) {

visited[i] = true;

queue[rear++] = i;

}

}

}

System.out.println();

}

// Depth First Search (DFS)

public void dfs(int startVertex) {

boolean[] visited = new boolean[numVertices];

int[] stack = new int[numVertices];

int top = -1;

// Push the start vertex onto the stack and mark it as visited

stack[++top]= startVertex;

visited[startVertex] = true;

System.out.println("DFS travesal starting from vertex "+startVertex+" : ");

while(top>=0) {

int currentVertex = stack[top--]; // Pop from stack

System.out.print(currentVertex+ " ");

// Push all adjacent unvisited vertices onto the stack

for(int i=numVertices-1;i>=0;i--) {

// Reverse order to mimic recursion

if(adjacencyMatrix[currentVertex][i]==1 && !visited[i]) {

visited[i] = true;

stack[++top] = i;

}

}

}

System.out.println();

}

// Method to display the adjacency matrix

public void displayAdjacencyMatrix() {

System.out.println("Adjacency Matrix:");

for(int i=0;i<numVertices;i++) { // Vertices

for(int j=0;j<numVertices;j++){ // edges

System.out.print(adjacencyMatrix[i][j] + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of verices:");

int numVertices = sc.nextInt();

Graph graph = new Graph(numVertices);

System.out.println("Enter the number of edges: ");

int numEdges = sc.nextInt();

System.out.println("Enter the edges (source and destination): ");

for(int i=0; i<numEdges;i++) {

int src = sc.nextInt();

int dest = sc.nextInt();

graph.addEdge(src, dest);

}

graph.displayAdjacencyMatrix();

System.out.println("Enter the start vertex for BFS: ");

int bfsStartVertex = sc.nextInt();

graph.bfs(bfsStartVertex);

System.out.println("Enter the start vertex for DFS: ");

int dfsStartVertex = sc.nextInt();

graph.dfs(dfsStartVertex);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------------------------------

//--------Double Ended Queue-----------//

package DoubleEndedQueue;

import java.util.Scanner;

class Node{

int data;

Node prev;

Node next;

// Function to get a new node

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

public class DoubleEndedQueue {

Node front;

Node rear;

int Size;

public DoubleEndedQueue() {

front = rear = null;

Size = 0;

}

public boolean isEmpty() {

return (front==null);

}

// return the number of elements in the deque

public int size() {

return Size;

}

// insert an element at the front end

public void insertFront(int data) {

Node newNode = new Node(data);

if(newNode==null) {

System.out.println("OverFlow!\n");

}

else {

// if deque is empty

if(front == null) {

//insert node at the front end

rear = front = newNode;

}

else {

newNode.next = front;

front.prev = newNode;

front = newNode;

}

Size++; // to count element

}

}

// insert an element at the rear end

public void insertRear(int data) {

Node newNode = new Node(data);

if(newNode==null) {

System.out.println("Overflow!\n");

}

else {

// if deque is empty

if(rear==null) {

// insert node at the rear end

front = rear = newNode;

}

else {

newNode.prev = rear;

rear.next = newNode;

rear = newNode;

}

Size++;

}

}

// delete the element from the front end

public void deleteFront() {

// if deque is empty then underflow condition

if(isEmpty()) {

System.out.println("Underflow - deque is empty");

}

//deletes the node form the front end

else {

Node temp = front;

front = front.next;

// if only one element was present

if(front ==null) {

rear=null;

}

else {

front.prev = null;

}

// decrements the count of elements by 1

Size--;

}

}

// delete the element from the rear end

void deleteRear() {

// if deque is empty then 'underflow' condition

if(isEmpty()) {

System.out.println("Underflow - deque is empty!\n");

}

// deletes the node from the rear end

else {

Node temp = rear;

rear = rear.prev;

// if only one elements was present

if(rear==null) {

front=null;

}

else {

rear.next = null;

}

//Decrements count of elements by 1

Size--;

}

}

// return the elements at the front end

public int getFront() {

// if deque is empty, then return -1 value

if(isEmpty()) {

return -1;

}

return front.data;

}

// return the elements at the rear end

public int getRear() {

// if deque is empty, then return -1 value

if(isEmpty()) {

return -1;

}

return rear.data;

}

public void display() {

Node current = front;

if(front==null) {

System.out.println("The double ended queue is empty!");

return;

}

System.out.println("The data in double ended queue are: ");

System.out.print("Null <- ");

while(current!=null) {

System.out.print(current.data+" <-> ");

current = current.next;

}

System.out.print("Null");

System.out.println();

}

public static void main(String[] args) {

DoubleEndedQueue deq = new DoubleEndedQueue();

Scanner sc = new Scanner(System.in);

int data, pos;

int choice;

do {

System.out.println("1. Insert Front");

System.out.println("2. Insert Rear");

System.out.println("3. Delete Front");

System.out.println("4. Delete Rear");

System.out.println("5. Display");

System.out.println("6. Get Front");

System.out.println("7. Get Rear");

System.out.println("8. Exit");

System.out.println("\nPlease Enter your choise: ");

choice = sc.nextInt();

switch (choice) {

case 1:

System.out.println("\n Insert Node at front - Enter data: ");

data = sc.nextInt();

deq.insertFront(data);

deq.display();

break;

case 2:

System.out.println("\n Insert Node at rear - Enter data: ");

data = sc.nextInt();

deq.insertRear(data);

deq.display();

break;

case 3:

System.out.println("\n Delete node at front: ");

deq.deleteFront();

deq.display();

break;

case 4:

System.out.println("\n Delete node at Rear: ");

deq.deleteRear();

deq.display();

break;

case 5:

System.out.println("\n Display Nodes: ");

deq.display();

break;

case 6:

System.out.println("\n The front element is : "+deq.getFront());

break;

case 7:

System.out.println("\n The front element is : "+deq.getRear());

break;

case 8:

System.out.println("Exiting the program.");

break;

default:

System.out.println("You entered wrong choice!");

}

} while (choice != 8);

sc.close();

}

}

-------------------------------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------------------------------

//--------Kruskal Alogrithm----------//

package Graph;

import java.util.Scanner;

class Edge {

int src, dest, weight;

// Constructor

Edge(int src, int dest, int weight) {

this.src = src;

this.dest = dest;

this.weight = weight;

}

}

public class KruskalAlgorithm {

private int V; // Number of vertices

private int E; // Number of edges

private Edge[] edges; // Array to store all edges

private int edgeCount = 0; // Counter for added edges

// Constructor

public KruskalAlgorithm(int vertices, int edgesCount) {

this.V = vertices;

this.E = edgesCount;

edges = new Edge[edgesCount];

}

// Add an edge to the graph

public void addEdge(int src, int dest, int weight) {

edges[edgeCount++] = new Edge(src, dest, weight);

}

// Find the parent of a vertex (with path compression)

private int findParent(int[] parent, int vertex) {

if (parent[vertex] != vertex) {

parent[vertex] = findParent(parent, parent[vertex]);

}

return parent[vertex];

}

// Perform union of two sets

private void union(int[] parent, int[] rank, int x, int y) {

int rootX = findParent(parent, x);

int rootY = findParent(parent, y);

if (rootX != rootY) {

if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

// Function to sort edges by weight using a simple bubble sort

private void sortEdges() {

for (int i = 0; i < E - 1; i++) {

for (int j = 0; j < E - i - 1; j++) { // Corrected loop condition

if (edges[j].weight > edges[j + 1].weight) {

Edge temp = edges[j];

edges[j] = edges[j + 1];

edges[j + 1] = temp;

}

}

}

}

// Kruskal's algorithm to find MST

public void kruskalMST() {

// Sort edges by weight

sortEdges();

// Arrays for Union-Find

int[] parent = new int[V];

int[] rank = new int[V];

// Initialize Union-Find structure

for (int i = 0; i < V; i++) {

parent[i] = i;

rank[i] = 0;

}

Edge[] mst = new Edge[V - 1]; // Array to store the MST edges

int mstIndex = 0; // Index for MST edges

int mstWeight = 0; // Total weight of MST

// Iterate through the sorted edges

for (int i = 0; i < E; i++) {

if (mstIndex == V - 1) break; // Stop if MST is complete

Edge edge = edges[i];

int srcParent = findParent(parent, edge.src);

int destParent = findParent(parent, edge.dest);

// If adding this edge doesn't form a cycle

if (srcParent != destParent) {

mst[mstIndex++] = edge; // Include the edge in MST

mstWeight += edge.weight;

union(parent, rank, srcParent, destParent); // Merge sets

}

}

// Print the MST

System.out.println("Edges in the MST:");

System.out.println("Src -- Dest == Weight");

for (int i = 0; i < mstIndex; i++) {

System.out.println(mst[i].src + " -- " + mst[i].dest + " == " + mst[i].weight);

}

System.out.println("Total Weight of MST: " + mstWeight);

}

// Main method to test the program

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of vertices: ");

int V = sc.nextInt();

System.out.println("Enter the number of edges: ");

int E = sc.nextInt();

KruskalAlgorithm graph = new KruskalAlgorithm(V, E);

System.out.println("Enter the edges in the format: src dest weight");

for (int i = 0; i < E; i++) { // Removed incorrect semicolon

int src = sc.nextInt();

int dest = sc.nextInt();

int weight = sc.nextInt();

graph.addEdge(src, dest, weight);

}

graph.kruskalMST();

sc.close();

}

}