**C-DAC Mumbai Date 29/09/2024**

**Subject: Algorithm and Data Structure**

**Assignment 3**

**Solve the assignment with following thing to be added in each question.**

-Program

-Flow chart

-Explanation

-Output

-Time and Space complexity

Submission Date: 01/10/2024

**1. Implement a Stack using an array.**

* **Test Case 1**:  
  Input: Push 5, 3, 7, Pop  
  Output: Stack = [5, 3], Popped element = 7
* **Test Case 2**:  
  Input: Push 10, Push 20, Pop, Push 15  
  Output: Stack = [10, 15], Popped element = 20
* import java.util.\*;
* class Stack {
* private int arr[];
* private int top;
* public Stack(int size) {
* arr = new int[size];
* top = -1;
* }
* public void push(int val) {
* if (isFull())
* throw new RuntimeException("Stack is full!!");
* top++;
* arr[top] = val;
* }
* public void pop(int val) {
* if (isEmpty())
* throw new RuntimeException("Stack is empty!!");
* top--;
* arr[top] = val;
* }
* public int peek() {
* if (isEmpty())
* throw new RuntimeException("Stack is empty!!");
* return arr[top];
* }
* public boolean isFull() {
* return top == arr.length - 1;
* }
* public boolean isEmpty() {
* return top == -1;
* }
* }
* public class StackArr {
* public static void main(String[] args) {
* int val;
* Scanner sc = new Scanner(System.***in***);
* Stack s = new Stack(5);
* int ch;
* do {
* System.***out***.println("0.Exit \n1.Push \n2.Pop \n3.peek \n\nEnter the choice");
* ch = sc.nextInt();
* switch (ch) {
* case 1: // push
* try {
* System.***out***.println("Enter the value to push ");
* val = sc.nextInt();
* s.push(val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* case 2:// pop
* try {
* val = s.peek();
* s.pop(val);
* System.***out***.println("Popped :" + val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* case 3:// peek
* val = s.peek();
* try {
* System.***out***.println("Topmost :" + val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* }
* } while (ch != 0);
* sc.close();
* }
* }

**2. Check for balanced parentheses using a stack.**

* **Test Case 1**:  
  Input: "({[()]})"  
  Output: Balanced
* **Test Case 2**:  
  Input: "([)]"  
  Output: Not Balanced
* public class BracketValidator{
* private static boolean isBalanced(String expr) {
* int n = expr.length();
* char [] stack = new char[n];
* int top = -1;
* for(int i=0; i<n; i++){
* char c = expr.charAt(i);
* if(c== '(' || c == '' || c == '{'){
* stack[++top] = c;
* }
* else if(c== ')' || c == ']' || c == ''){
* if(top == -1) return false;
* char open = stack[top--];
* if((c == ')' && open != '(1
* c == ']' && open != '[' ||
* c == } && open != '{')){
* return false;
* }
* }
* return top == -1;
* }
* public static void main(String[] args){
* Scanner SC = new Scanner(System.***in***);
* String n= sc.nextLine();
* if(*isBalanced*(n) ){
* System.***out***.println("Balanced");
* } else{
* System.***out***.println("Not Balanced");
* }
* }
* }

**3. Reverse a string using a stack.**

* **Test Case 1**:  
  Input: "hello"  
  Output: "olleh"
* **Test Case 2**:  
  Input: "world"  
  Output: "dlrow"

**4. Evaluate a postfix expression using a stack.**

* **Test Case 1**:  
  Input: "5 3 + 2 \*"  
  Output: 16
* **Test Case 2**:  
  Input: "4 5 \* 6 /"  
  Output: 3
* import java.util.Stack;
* public class ExpressionMain {
* public static int calc(int a, int b, char operator) {
* switch (operator) {
* case '$': return (int)Math.pow(a, b);
* case '\*': return a \* b;
* case '/': return a / b;
* case '%': return a % b;
* case '+': return a + b;
* case '-': return a - b;
* }
* return 0;
* }
* public static int solvePostfix(String post) {
* // stack of operands
* Stack<Integer> s = new Stack<Integer>();
* // traverse postfix from left to right
* for (int i = 0; i < post.length(); i++) {
* // get each sym from expression
* char sym = post.charAt(i);
* // if sym is operand
* if(Character.isDigit(sym)) {
* // convert it to int & push on stack
* // e.g. '5' -toString()-> "5" -parseInt()-> 5
* String operand = Character.toString(sym);
* s.push(Integer.parseInt(operand));
* } else {
* // pop two operands from stack
* int b = s.pop();
* int a = s.pop();
* // calculate the result
* int c = calc(a, b, sym);
* // push result on stack
* s.push(c);
* }
* }// repeat for all syms in expression
* // pop final result from stack and return.
* return s.pop();
* }
* public static void main(String[] args) {
* String postfix = "59+4862/-\*-173-$+";
* int result = solvePostfix(postfix);
* System.out.println("Result: " + result);
* }
* }

**5. Convert an infix expression to postfix using a stack.**

* **Test Case 1**:  
  Input: "A + B \* C"  
  Output: "A B C \* +"
* **Test Case 2**:  
  Input: "A \* B + C / D"  
  Output: "A B \* C D / +"
* class Stack {
* private char[] arr;
* private int top;
* private int capacity;
* public Stack(int size) {
* arr = new char[size];
* capacity = size;
* top = -1;
* }
* public void push(char item) {
* if (top == capacity - 1) {
* throw new StackOverflowError("Stack is full");
* }
* arr[++top] = item;
* }
* public char pop() {
* if (isEmpty()) {
* throw new RuntimeException("Stack is empty");
* }
* return arr[top--];
* }
* public char peek() {
* if (isEmpty()) {
* throw new RuntimeException("Stack is empty");
* }
* return arr[top];
* }
* public boolean isEmpty() {
* return top == -1;
* }
* public int size() {
* return top + 1;
* }
* }
* public class InfixToPostfix {
* private static int precedence(char op) {
* switch (op) {
* case '+':
* case '-':
* return 1;
* case '\*':
* case '/':
* return 2;
* default:
* return 0;
* }
* }
* public static String convert(String infix) {
* StringBuilder postfix = new StringBuilder();
* Stack stack = new Stack(infix.length());
* for (char token : infix.toCharArray()) {
* if (Character.*isLetter*(token)) {
* postfix.append(token).append(' ');
* } else if (token == '(') {
* stack.push(token);
* } else if (token == ')') {
* while (!stack.isEmpty() && stack.peek() != '(') {
* postfix.append(stack.pop()).append(' ');
* }
* if (!stack.isEmpty() && stack.peek() == '(') {
* stack.pop();
* }
* } else if (*isOperator*(token)) {
* while (!stack.isEmpty() && *precedence*(stack.peek()) >= *precedence*(token)) {
* postfix.append(stack.pop()).append(' ');
* }
* stack.push(token);
* }
* }
* while (!stack.isEmpty()) {
* postfix.append(stack.pop()).append(' ');
* }
* return postfix.toString().trim();
* }
* private static boolean isOperator(char c) {
* return c == '+' || c == '-' || c == '\*' || c == '/';
* }
* public static void main(String[] args) {
* String expression1 = "A + B \* C";
* String expression2 = "A \* B + C / D";
* System.***out***.println("Postfix of \"" + expression1 + "\": " + *convert*(expression1));
* System.***out***.println("Postfix of \"" + expression2 + "\": " + *convert*(expression2));
* }
* }

**6. Implement a Queue using an array.**

* **Test Case 1**:  
  Input: Enqueue 5, Enqueue 10, Dequeue  
  Output: Queue = [10], Dequeued element = 5
* **Test Case 2**:  
  Input: Enqueue 1, 2, 3, Dequeue, Dequeue  
  Output: Queue = [3], Dequeued elements = 1, 2
* class Que {
* int size = 5;
* int Q[] = new int[size];
* int front, rear;
* Que() {
* front = -1;
* rear = -1;
* }
* boolean isEmpty() {
* return (front == -1);
* }
* boolean isFull() {
* return (rear == -1);
* }
* // enqueue in Queue
* void enqueue(int x){
* if(isFull()){
* System.***out***.println("Queue is full");
* }
* else {
* if (front == -1) {
* front =0;//set queue at 0 if it is empty queue
* }
* rear++;
* Q[rear]=x;
* System.***out***.println(x+ " inserted");
* }
* }
* int dequeue() {
* if (isEmpty()) {
* System.***out***.println("empty queue!!");
* return -1;
* }
* else {
* int x = Q[front];
* System.***out***.println(x+ " deleted");
* if (front > rear) {
* front = -1;
* rear = -1;
* }
* else {
* front++;
* }
* return x;
* }
* }
* void display() {
* if(isEmpty()) {
* System.***out***.println("Empty");
* }
* else {
* System.***out***.println("Queue elements are!! ");
* for(int i = front;i<=rear;i++)
* {
* System.***out***.print("[ " +Q[i]+" ]");
* }
* System.***out***.println();
* }
* }
* }
* public class SimpleQ {
* public static void main(String[] args) {
* Que q1= new Que();
* q1.enqueue(5);
* q1.enqueue(4);
* q1.display();


* }
* }

**7. Implement a Circular Queue using an array.**

* **Test Case 1**:  
  Input: Enqueue 4, 5, 6, 7, Dequeue, Enqueue 8  
  Output: Queue = [8, 5, 6, 7]
* **Test Case 2**:  
  Input: Enqueue 1, 2, 3, 4, Dequeue, Dequeue, Enqueue 5  
  Output: Queue = [5, 3, 4]
* import java.util.Scanner;
* //Queue operation using array;
* class Queue{
* private int[] arr;
* private int rear,front;
* public Queue(int size) {
* arr=new int [size];
* rear=-1;
* front=-1;
* }
* public boolean isEmpty() {
* return (front == rear && front == -1);
* }
* public boolean isFull() {
* return (front==-1 && rear==arr.length-1)||
* (front==rear && front!=-1);
* }
* // enqueue in Queue
* void enqueue(int val){
* if(isFull())
* throw new RuntimeException("Queue is full");
* rear=(rear+1)%arr.length;
* arr[rear]=val;
* }
* void dequeue() {
* if(isEmpty())
* throw new RuntimeException("Queue is empty");
* front=(front+1)%arr.length;
* if(front==rear){
* rear=-1;
* front=-1;
* }
* }
* public int peek() {
* if(isEmpty())
* throw new RuntimeException("Queue is empty");
* int index = (front+1)%arr.length;
* return arr[index];
* }
* }
* public class Cir\_queue {
* public static void main(String[] args) {
* int val;
* Scanner sc = new Scanner(System.***in***);
* Que s = new Que(5);
* int ch;
* do {
* System.***out***.println("0.Exit \n1.enquque \n2.dequeue \n3.peek \n\nEnter the choice");
* ch = sc.nextInt();
* switch (ch) {
* case 1: // push
* try {
* System.***out***.println("Enter the value to push ");
* val = sc.nextInt();
* s.enqueue(val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* case 2:// pop
* try {
* val = s.peek();
* s.dequeue();
* System.***out***.println("Popped :" + val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* case 3:// peek
* val = s.peek();
* try {
* System.***out***.println("Topmost :" + val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* }
* } while (ch != 0);
* sc.close();
* }
* }

**8. Implement a Queue using two Stacks.**

* **Test Case 1**:  
  Input: Enqueue 3, Enqueue 7, Dequeue  
  Output: Queue = [7], Dequeued element = 3
* **Test Case 2**:  
  Input: Enqueue 10, 20, Dequeue, Dequeue  
  Output: Queue = [], Dequeued elements = 10, 20
* public class QueueUsingStacks {
* private int[] inputStack;
* private int[] outputStack;
* private int inputTop;
* private int outputTop;
* // Constructor
* public QueueUsingStacks(int size) {
* inputStack = new int[size];
* outputStack = new int[size];
* inputTop = -1; // Initialize the top index of inputStack
* outputTop = -1; // Initialize the top index of outputStack
* }
* // Method to push an element onto a stack
* private void pushToInputStack(int value) {
* if (inputTop < inputStack.length - 1) {
* inputStack[++inputTop] = value;
* } else {
* System.***out***.println("Input stack is full!");
* }
* }
* // Method to pop an element from a stack
* private int popFromInputStack() {
* if (inputTop >= 0) {
* return inputStack[inputTop--];
* }
* System.***out***.println("Input stack is empty!");
* return -1; // Return -1 if the stack is empty
* }
* // Method to push an element onto output stack
* private void pushToOutputStack(int value) {
* if (outputTop < outputStack.length - 1) {
* outputStack[++outputTop] = value;
* } else {
* System.***out***.println("Output stack is full!");
* }
* }
* // Method to pop an element from output stack
* private int popFromOutputStack() {
* if (outputTop >= 0) {
* return outputStack[outputTop--];
* }
* System.***out***.println("Output stack is empty!");
* return -1; // Return -1 if the stack is empty
* }
* // Enqueue operation
* public void enqueue(int value) {
* pushToInputStack(value);
* }
* // Dequeue operation
* public int dequeue() {
* // If both stacks are empty, return -1
* if (inputTop == -1 && outputTop == -1) {
* System.***out***.println("Queue is empty!");
* return -1;
* }
* // If output stack is empty, transfer elements from input stack
* if (outputTop == -1) {
* while (inputTop >= 0) {
* pushToOutputStack(popFromInputStack());
* }
* }
* // Return the top element of the output stack
* return popFromOutputStack();
* }
* // Method to display the queue
* public void displayQueue() {
* System.***out***.print("Queue = [");
* // Temporary array to show the queue
* for (int i = outputTop; i >= 0; i--) {
* System.***out***.print(outputStack[i] + (i == 0 ? "" : ", "));
* }
* // Print elements from the input stack
* for (int i = 0; i <= inputTop; i++) {
* System.***out***.print(inputStack[i] + (i == inputTop ? "" : ", "));
* }
* System.***out***.println("]");
* }
* public static void main(String[] args) {
* QueueUsingStacks queue = new QueueUsingStacks(10); // Create a queue with a size of 10
* // Test Case 1
* queue.enqueue(3);
* queue.enqueue(7);
* int dequeuedElement1 = queue.dequeue();
* System.***out***.print("Test Case 1: ");
* queue.displayQueue(); // Output: [7]
* System.***out***.println("Dequeued element = " + dequeuedElement1); // Output: 3
* // Test Case 2
* queue.enqueue(10);
* queue.enqueue(20);
* int dequeuedElement2 = queue.dequeue();
* int dequeuedElement3 = queue.dequeue();
* System.***out***.print("Test Case 2: ");
* queue.displayQueue(); // Output: []
* System.***out***.println("Dequeued elements = " + dequeuedElement2 + ", " + dequeuedElement3); // Output: 10, 20
* }
* }

**9. Implement a Min-Heap.**

* **Test Case 1**:  
  Input: Insert 10, 15, 20, 17, Extract Min  
  Output: Min-Heap = [15, 17, 20], Extracted Min = 10
* **Test Case 2**:  
  Input: Insert 30, 40, 20, 50, Extract Min  
  Output: Min-Heap = [30, 40, 50], Extracted Min = 20
* class MinHeap {
* private int[] heap;
* private int size;
* private int capacity;
* public MinHeap(int capacity) {
* this.capacity = capacity;
* heap = new int[capacity];
* size = 0;
* }
* public void insert(int value) {
* if (size == capacity) return;
* heap[size] = value;
* size++;
* heapifyUp(size - 1);
* }
* public int extractMin() {
* if (size == 0) return Integer.***MIN\_VALUE***;
* int min = heap[0];
* heap[0] = heap[size - 1];
* size--;
* heapifyDown(0);
* return min;
* }
* private void heapifyUp(int index) {
* while (index > 0) {
* int parentIndex = (index - 1) / 2;
* if (heap[parentIndex] <= heap[index]) break;
* swap(parentIndex, index);
* index = parentIndex;
* }
* }
* private void heapifyDown(int index) {
* while (index < size) {
* int left = 2 \* index + 1;
* int right = 2 \* index + 2;
* int smallest = index;
* if (left < size && heap[left] < heap[smallest]) smallest = left;
* if (right < size && heap[right] < heap[smallest]) smallest = right;
* if (smallest == index) break;
* swap(index, smallest);
* index = smallest;
* }
* }
* private void swap(int i, int j) {
* int temp = heap[i];
* heap[i] = heap[j];
* heap[j] = temp;
* }
* public int[] toArray() {
* int[] result = new int[size];
* System.*arraycopy*(heap, 0, result, 0, size);
* return result;
* }
* }

**10. Implement a Max-Heap.**

* **Test Case 1**:  
  Input: Insert 12, 7, 15, 5, Extract Max  
  Output: Max-Heap = [12, 7, 5], Extracted Max = 15
* **Test Case 2**:  
  Input: Insert 8, 20, 10, 3, Extract Max  
  Output: Max-Heap = [10, 8, 3], Extracted Max = 20
* package Heap.ja.orh;
* public class maxhip {
* private int[] heap;
* private int size;
* private int capacity;
* public MaxHeap(int capacity) {
* this.capacity = capacity;
* heap = new int[capacity];
* size = 0;
* }
* public void insert(int value) {
* if (size == capacity) return;
* heap[size] = value;
* size++;
* heapifyUp(size - 1);
* }
* public int extractMax() {
* if (size == 0) return Integer.MAX\_VALUE;
* int max = heap[0];
* heap[0] = heap[size - 1];
* size--;
* heapifyDown(0);
* return max;
* }
* private void heapifyUp(int index) {
* while (index > 0) {
* int parentIndex = (index - 1) / 2;
* if (heap[parentIndex] >= heap[index]) break;
* swap(parentIndex, index);
* index = parentIndex;
* }
* }
* private void heapifyDown(int index) {
* while (index < size) {
* int left = 2 \* index + 1;
* int right = 2 \* index + 2;
* int largest = index;
* if (left < size && heap[left] > heap[largest]) largest = left;
* if (right < size && heap[right] > heap[largest]) largest = right;
* if (largest == index) break;
* swap(index, largest);
* index = largest;
* }
* }
* private void swap(int i, int j) {
* int temp = heap[i];
* heap[i] = heap[j];
* heap[j] = temp;
* }
* public int[] toArray() {
* int[] result = new int[size];
* System.arraycopy(heap, 0, result, 0, size);
* return result;
* }
* }
* }

**11. Sort an array using a heap (Heap Sort).**

* **Test Case 1**:  
  Input: [5, 1, 12, 3, 9]  
  Output: [1, 3, 5, 9, 12]

**Test Case 2**:  
Input: [20, 15, 8, 10]  
Output: [8, 10, 15, 20] class HeapSort {

public static void heapSort(int[] array) {

*buildMinHeap*(array);

for (int i = array.length - 1; i > 0; i--) {

*swap*(array, 0, i);

*heapify*(array, 0, i);

}

}

private static void buildMinHeap(int[] array) {

for (int i = array.length / 2 - 1; i >= 0; i--) {

*heapify*(array, i, array.length);

}

}

private static void heapify(int[] array, int index, int size) {

int smallest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if (left < size && array[left] < array[smallest]) smallest = left;

if (right < size && array[right] < array[smallest]) smallest = right;

if (smallest != index) {

*swap*(array, index, smallest);

*heapify*(array, smallest, size);

}

}

private static void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}



**12. Find the kth largest element in a stream of numbers using a heap.**

* **Test Case 1**:  
  Input: Stream = [3, 10, 5, 20, 15], k = 3  
  Output: 10
* **Test Case 2**:  
  Input: Stream = [7, 4, 8, 2, 9], k = 2  
  Output: 8
* import java.util.PriorityQueue;
* class KthLargest {
* private final PriorityQueue<Integer> minHeap;
* private final int k;
* public KthLargest(int k) {
* this.k = k;
* this.minHeap = new PriorityQueue<>();
* }
* public void add(int value) {
* if (minHeap.size() < k) {
* minHeap.offer(value);
* } else if (value > minHeap.peek()) {
* minHeap.poll();
* minHeap.offer(value);
* }
* }
* public int getKthLargest() {
* return minHeap.peek();
* }
* }

**13. Implement a Priority Queue using a heap.**

* **Test Case 1**:  
  Input: Enqueue with priorities: 3 (priority 1), 10 (priority 3), 5 (priority 2), Dequeue  
  Output: Dequeued element = 10 (highest priority), Priority Queue = [5, 3]
* **Test Case 2**:  
  Input: Enqueue with priorities: 7 (priority 4), 8 (priority 2), 6 (priority 3), Dequeue  
  Output: Dequeued element = 7, Priority Queue = [6, 8]
* class PriorityQueueElement {
* int value;
* int priority;
* PriorityQueueElement(int value, int priority) {
* this.value = value;
* this.priority = priority;
* }
* }
* class PriorityQueue {
* private final MaxHeap maxHeap;
* public PriorityQueue(int capacity) {
* maxHeap = new MaxHeap(capacity);
* }
* public void enqueue(int value, int priority) {
* maxHeap.insert(new PriorityQueueElement(value, priority).value);
* }
* public int dequeue() {
* return maxHeap.extractMax();
* }
* public int[] toArray() {
* return maxHeap.toArray();
* }
* }

**14. Design an algorithm to implement a stack with a getMin() function to return the minimum element in constant time.**

* **Test Case 1**:  
  Input: Push 5, Push 3, Push 7, Get Min  
  Output: Min = 3
* **Test Case 2**:  
  Input: Push 10, Push 8, Push 6, Push 12, Get Min  
  Output: Min = 6
* class MinStack {
* private final Stack<Integer> stack;
* private final Stack<Integer> minStack;
* public MinStack() {
* stack = new Stack<>();
* minStack = new Stack<>();
* }
* public void push(int value) {
* stack.push(value);
* if (minStack.isEmpty() || value <= minStack.peek()) {
* minStack.push(value);
* }
* }
* public int pop() {
* int value = stack.pop();
* if (value == minStack.peek()) {
* minStack.pop();
* }
* return value;
* }
* public int getMin() {
* return minStack.peek();
* }
* }

**15. Design a Circular Queue with a fixed size, supporting enqueue, dequeue, and isFull/isEmpty operations.**

* **Test Case 1**:  
  Input: Size = 4, Enqueue 1, 2, 3, 4, isFull()  
  Output: True
* **Test Case 2**:  
  Input: Size = 3, Enqueue 5, 6, Dequeue, Enqueue 7, isEmpty()  
  Output: False
* import java.util.Scanner;
* //Queue operation using array;
* class Queue{
* private int[] arr;
* private int rear,front;
* public Queue(int size) {
* arr=new int [size];
* rear=-1;
* front=-1;
* }
* public boolean isEmpty() {
* return (front == rear && front == -1);
* }
* public boolean isFull() {
* return (front==-1 && rear==arr.length-1)||
* (front==rear && front!=-1);
* }
* // enqueue in Queue
* void enqueue(int val){
* if(isFull())
* throw new RuntimeException("Queue is full");
* rear=(rear+1)%arr.length;
* arr[rear]=val;
* }
* void dequeue() {
* if(isEmpty())
* throw new RuntimeException("Queue is empty");
* front=(front+1)%arr.length;
* if(front==rear){
* rear=-1;
* front=-1;
* }
* }
* public int peek() {
* if(isEmpty())
* throw new RuntimeException("Queue is empty");
* int index = (front+1)%arr.length;
* return arr[index];
* }
* }
* public class Cir\_queue {
* public static void main(String[] args) {
* int val;
* Scanner sc = new Scanner(System.***in***);
* Que s = new Que(5);
* int ch;
* do {
* System.***out***.println("0.Exit \n1.enquque \n2.dequeue \n3.peek \n\nEnter the choice");
* ch = sc.nextInt();
* switch (ch) {
* case 1: // push
* try {
* System.***out***.println("Enter the value to push ");
* val = sc.nextInt();
* s.enqueue(val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* case 2:// pop
* try {
* val = s.peek();
* s.dequeue();
* System.***out***.println("Popped :" + val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* case 3:// peek
* val = s.peek();
* try {
* System.***out***.println("Topmost :" + val);
* } catch (Exception e) {
* System.***err***.println(e.getMessage());
* }
* break;
* }
* } while (ch != 0);
* sc.close();
* }
* }