C-DAC Mumbai Date 29/09/2024

Subject: Algorithm and Data Structure

Assignment 3

1. Implement a Stack using an array

Program

class Stack {

int[] stack;

int top;

int capacity;

public Stack(int size) {

stack = new int[size];

capacity = size;

top = -1;

}

public void push(int x) {

if (isFull()) {

System.out.println("Stack Overflow");

return;

}

stack[++top] = x;

}

public int pop() {

if (isEmpty()) {

System.out.println("Stack Underflow");

return -1;

}

return stack[top--];

}

public boolean isEmpty() {

return top == -1;

}

public boolean isFull() {

return top == capacity - 1;

}

public void display() {

if (isEmpty()) {

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

return;

}

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

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

}

System.out.println();

}

}

Stack s = new Stack(5);

s.push(5);

s.push(3);

s.push(7);

System.out.println("Popped element = " + s.pop());

s.display();

Output:

css

Popped element = 7

Stack = [5, 3]

Explanation

* We use an array to implement the stack. push() adds elements to the stack, and pop() removes and returns the top element.
* In Test Case 1, 7 is pushed last, so it is popped first (LIFO).

Flow Chart

* Start -> Push Elements -> Pop Element -> Display Stack -> End

Time Complexity:

* Push: O(1)
* Pop: O(1)

Space Complexity:

* O(n) (where n is the size of the stack)

2. Check for balanced parentheses using a stack

Program

import java.util.Stack;

public class BalancedParentheses {

public static boolean isBalanced(String expr) {

Stack<Character> stack = new Stack<>();

for (char ch : expr.toCharArray()) {

if (ch == '(' || ch == '{' || ch == '[') {

stack.push(ch);

} else {

if (stack.isEmpty()) return false;

char top = stack.pop();

if ((ch == ')' && top != '(') || (ch == '}' && top != '{') || (ch == ']' && top != '[')) {

return false;

}

}

}

return stack.isEmpty();

}

}

System.out.println(isBalanced("({[()]})")); // Output: Balanced

System.out.println(isBalanced("([)]")); // Output: Not Balanced

Output:

mathematica

Balanced

Not Balanced

Explanation

* We use a stack to track opening brackets and ensure matching closing brackets. If all brackets are matched, the string is balanced.

Flow Chart

* Start -> Push Opening Brackets -> Pop on Closing Bracket -> Check Balance -> End

Time Complexity:

* O(n) (where n is the length of the string)

Space Complexity:

* O(n) (due to the stack)

3. Reverse a string using a stack

Program

import java.util.Stack;

public class ReverseString {

public static String reverse(String str) {

Stack<Character> stack = new Stack<>();

for (char ch : str.toCharArray()) {

stack.push(ch);

}

StringBuilder reversed = new StringBuilder();

while (!stack.isEmpty()) {

reversed.append(stack.pop());

}

return reversed.toString();

}

}

Test Case 1

System.out.println(reverse("hello")); // Output: "olleh"

System.out.println(reverse("world")); // Output: "dlrow"

Output:

olleh

dlrow

Explanation

* Each character of the string is pushed to the stack, then popped in reverse order to create the reversed string.

Flow Chart

* Start -> Push Characters to Stack -> Pop to Reverse -> Display Reversed String -> End

Time Complexity:

* O(n)

Space Complexity:

* O(n)

4. Evaluate a postfix expression using a stack

Program

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String expr) {

Stack<Integer> stack = new Stack<>();

for (String token : expr.split(" ")) {

if (token.matches("-?\\d+")) {

stack.push(Integer.parseInt(token));

} else {

int val2 = stack.pop();

int val1 = stack.pop();

switch (token) {

case "+" -> stack.push(val1 + val2);

case "-" -> stack.push(val1 - val2);

case "\*" -> stack.push(val1 \* val2);

case "/" -> stack.push(val1 / val2);

}

}

}

return stack.pop();

}

}

System.out.println(evaluatePostfix("5 3 + 2 \*")); // Output: 16

System.out.println(evaluatePostfix("4 5 \* 6 /")); // Output: 3

Explanation

* The postfix expression is evaluated by scanning from left to right. Operands are pushed to the stack, and operators pop two operands, perform the operation, and push the result back.

Flow Chart

* Start -> Scan Expression -> Push Operand to Stack -> Pop for Operation -> Push Result to Stack -> End

Time Complexity:

* O(n) (where n is the number of tokens in the expression)

Space Complexity:

* O(n)

5. Convert an infix expression to postfix using a stack

Program

import java.util.Stack;

public class InfixToPostfix {

public static int precedence(char ch) {

return switch (ch) {

case '+', '-' -> 1;

case '\*', '/' -> 2;

default -> -1;

};

}

public static String infixToPostfix(String expr) {

Stack<Character> stack = new Stack<>();

StringBuilder result = new StringBuilder();

for (char c : expr.toCharArray()) {

if (Character.isLetterOrDigit(c)) {

result.append(c).append(' ');

} else if (c == '(') {

stack.push(c);

} else if (c == ')') {

while (!stack.isEmpty() && stack.peek() != '(') {

result.append(stack.pop()).append(' ');

}

stack.pop();

} else {

while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek())) {

result.append(stack.pop()).append(' ');

}

stack.push(c);

}

}

while (!stack.isEmpty()) {

result.append(stack.pop()).append(' ');

}

return result.toString().trim();

}

}

System.out.println(infixToPostfix("A + B \* C")); // Output: "A B C \* +"

System.out.println(infixToPostfix("A \* B + C / D")); // Output: "A B \* C D / +"

Explanation

* Operators are pushed to a stack. Operands are directly added to the result. When encountering a closing parenthesis, the operators are popped until the matching opening parenthesis is found.

Flow Chart

* Start -> Scan Expression -> Push Operators -> Add Operands -> Pop Operators at Parenthesis -> End

Time Complexity:

* O(n)

Space Complexity:

* O(n)

6. Implement a Queue using an array

Program

class Queue {

int[] queue;

int front, rear, capacity;

public Queue(int size) {

queue = new int[size];

capacity = size;

front = rear = -1;

}

public void enqueue(int x) {

if (rear == capacity - 1) {

System.out.println("Queue Overflow");

return;

}

if (front == -1) front = 0;

queue[++rear] = x;

}

public int dequeue() {

if (front == -1 || front > rear) {

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

return -1;

}

return queue[front++];

}

public void display() {

if (front == -1 || front > rear) {

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

return;

}

for (int i = front; i <= rear; i++) {

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

}

System.out.println();

}

}

Test Case 1

Queue q = new Queue(5);

q.enqueue(5);

q.enqueue(10);

System.out.println("Dequeued element = " + q.dequeue());

q.display(); // Output: Queue = [10]

Test Case 2

q.enqueue(1);

q.enqueue(2);

q.enqueue(3);

q.dequeue();

q.dequeue();

q.display(); // Output: Queue = [3]

Explanation

* Queue uses an array. Enqueue adds elements at the rear; Dequeue removes elements from the front.

Flow Chart

* Start -> Enqueue -> Dequeue -> Display Queue -> End

Time Complexity:

* O(1) (for both enqueue and dequeue)

Space Complexity:

* O(n)

7. Implement a Circular Queue using an array

Program

class CircularQueue {

private int[] queue;

private int front, rear, size, capacity;

public CircularQueue(int capacity) {

this.capacity = capacity;

queue = new int[capacity];

front = rear = -1;

size = 0;

}

public boolean isFull() {

return size == capacity;

}

public boolean isEmpty() {

return size == 0;

}

public void enqueue(int value) {

if (isFull()) {

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

return;

}

if (isEmpty()) {

front = rear = 0;

} else {

rear = (rear + 1) % capacity;

}

queue[rear] = value;

size++;

}

public int dequeue() {

if (isEmpty()) {

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

return -1;

}

int dequeuedValue = queue[front];

front = (front + 1) % capacity;

size--;

return dequeuedValue;

}

public void display() {

if (isEmpty()) {

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

return;

}

System.out.print("Queue = ");

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

System.out.print(queue[(front + i) % capacity] + " ");

}

System.out.println();

}

}

Test Case 1

CircularQueue cq = new CircularQueue(4);

cq.enqueue(4);

cq.enqueue(5);

cq.enqueue(6);

cq.enqueue(7);

cq.dequeue();

cq.enqueue(8);

cq.display(); // Output: Queue = [8, 5, 6, 7]

Test Case 2

Copy code

cq = new CircularQueue(4);

cq.enqueue(1);

cq.enqueue(2);

cq.enqueue(3);

cq.enqueue(4);

cq.dequeue();

cq.dequeue();

cq.enqueue(5);

cq.display(); // Output: Queue = [5, 3, 4]

Explanation

* A circular queue connects the rear and front, allowing efficient use of space. When the queue is full and rear is at the end, it wraps around to the beginning.

Flow Chart

* Start -> Enqueue -> Dequeue -> Wrap Around if Needed -> Display Queue -> End

Time Complexity:

* O(1) (for enqueue and dequeue operations)

Space Complexity:

* O(n)

8. Implement a Queue using two Stacks

Program

import java.util.Stack;

class QueueWithTwoStacks {

private Stack<Integer> stack1;

private Stack<Integer> stack2;

public QueueWithTwoStacks() {

stack1 = new Stack<>();

stack2 = new Stack<>();

}

public void enqueue(int value) {

stack1.push(value);

}

public int dequeue() {

if (stack2.isEmpty()) {

if (stack1.isEmpty()) {

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

return -1;

}

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

public void display() {

if (stack2.isEmpty()) {

System.out.println("Queue = " + stack1);

} else {

System.out.println("Queue = " + stack2);

}

}

}

QueueWithTwoStacks queue = new QueueWithTwoStacks();

queue.enqueue(3);

queue.enqueue(7);

System.out.println("Dequeued element = " + queue.dequeue()); // Output: Dequeued element = 3

queue.display(); // Output: Queue = [7]

Test Case 2

queue.enqueue(10);

queue.enqueue(20);

System.out.println("Dequeued element = " + queue.dequeue()); // Output: Dequeued element = 10

System.out.println("Dequeued element = " + queue.dequeue()); // Output: Dequeued element = 20

queue.display(); // Output: Queue = []

Explanation

* Two stacks are used: one for enqueue operations and another for dequeue operations. Elements are transferred from one stack to the other when a dequeue is requested.

Flow Chart

* Start -> Push to Stack1 for Enqueue -> Pop from Stack2 for Dequeue -> Transfer from Stack1 to Stack2 when Needed -> End

Time Complexity:

* O(1) for enqueue; O(n) for dequeue (when elements need to be transferred)

Space Complexity:

* O(n)

9. Implement a Min-Heap

Program

import java.util.PriorityQueue;

class MinHeap {

private PriorityQueue<Integer> minHeap;

public MinHeap() {

minHeap = new PriorityQueue<>();

}

public void insert(int value) {

minHeap.add(value);

}

public int extractMin() {

return minHeap.poll();

}

public void display() {

System.out.println("Min-Heap = " + minHeap);

}

}

MinHeap heap = new MinHeap();

heap.insert(10);

heap.insert(15);

heap.insert(20);

heap.insert(17);

System.out.println("Extracted Min = " + heap.extractMin()); // Output: Extracted Min = 10

heap.display(); // Output: Min-Heap = [15, 17, 20]

Test Case 2

heap.insert(30);

heap.insert(40);

heap.insert(20);

heap.insert(50);

System.out.println("Extracted Min = " + heap.extractMin()); // Output: Extracted Min = 20

heap.display(); // Output: Min-Heap = [30, 40, 50]

Explanation

* A Min-Heap is a binary tree where the root is the minimum element. The heap is maintained using a priority queue in Java.

Flow Chart

* Start -> Insert Elements -> Heapify After Insertions -> Extract Minimum -> Heapify After Extraction -> End

Time Complexity:

* O(log n) (for both insertion and extraction)

Space Complexity:

* O(n)

10. Implement a Max-Heap

Program

import java.util.Collections;

import java.util.PriorityQueue;

class MaxHeap {

private PriorityQueue<Integer> maxHeap;

public MaxHeap() {

maxHeap = new PriorityQueue<>(Collections.reverseOrder());

}

public void insert(int value) {

maxHeap.add(value);

}

public int extractMax() {

return maxHeap.poll();

}

public void display() {

System.out.println("Max-Heap = " + maxHeap);

}

}

MaxHeap heap = new MaxHeap();

heap.insert(12);

heap.insert(7);

heap.insert(15);

heap.insert(5);

System.out.println("Extracted Max = " + heap.extractMax()); // Output: Extracted Max = 15

heap.display(); // Output: Max-Heap = [12, 7, 5]

Test Case 2

heap.insert(8);

heap.insert(20);

heap.insert(10);

heap.insert(3);

System.out.println("Extracted Max = " + heap.extractMax()); // Output: Extracted Max = 20

heap.display(); // Output: Max-Heap = [10, 8, 3]

Explanation

* A Max-Heap is a binary tree where the root is the maximum element. Using a reversed priority queue, we achieve the Max-Heap.

Flow Chart

* Start -> Insert Elements -> Heapify After Insertions -> Extract Maximum -> Heapify After Extraction -> End

Time Complexity:

* O(log n) (for both insertion and extraction)

Space Complexity:

* O(n)

Problem 11: Heap Sort

Solution:

java

Copy code

import java.util.Arrays;

public class HeapSort {

public static void heapSort(int[] arr) {

int n = arr.length;

// Build max heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// Extract elements from heap

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

// Move current root to end

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

private static void heapify(int[] arr, int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// If left child is larger

if (left < n && arr[left] > arr[largest])

largest = left;

// If right child is larger

if (right < n && arr[right] > arr[largest])

largest = right;

// If largest is not root

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

public static void main(String[] args) {

int[] arr1 = {5, 1, 12, 3, 9};

heapSort(arr1);

System.out.println("Sorted array: " + Arrays.toString(arr1));

int[] arr2 = {20, 15, 8, 10};

heapSort(arr2);

System.out.println("Sorted array: " + Arrays.toString(arr2));

}

}

Time Complexity:

* Heapify: O(log n)
* Heap sort: O(n log n)

Space Complexity:

* O(1) (in-place sorting)

Problem 12: kth Largest Element in a Stream using a Min-Heap

Solution:

java

Copy code

import java.util.PriorityQueue;

public class KthLargest {

private PriorityQueue<Integer> minHeap;

private int k;

public KthLargest(int k, int[] stream) {

this.k = k;

minHeap = new PriorityQueue<>();

for (int num : stream) {

add(num);

}

}

public void add(int num) {

minHeap.offer(num);

if (minHeap.size() > k) {

minHeap.poll();

}

}

public int getKthLargest() {

return minHeap.peek();

}

public static void main(String[] args) {

int[] stream1 = {3, 10, 5, 20, 15};

KthLargest kthLargest1 = new KthLargest(3, stream1);

System.out.println("3rd largest element: " + kthLargest1.getKthLargest());

int[] stream2 = {7, 4, 8, 2, 9};

KthLargest kthLargest2 = new KthLargest(2, stream2);

System.out.println("2nd largest element: " + kthLargest2.getKthLargest());

}

}

Time Complexity:

* Add operation: O(log k)
* Get kth largest element: O(1)

Space Complexity:

* O(k) for storing the heap

Problem 13: Priority Queue using a Max-Heap

Solution:

java

Copy code

import java.util.PriorityQueue;

import java.util.Comparator;

public class PriorityQueueHeap {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());

// Test Case 1

pq.offer(10); // priority 3

pq.offer(5); // priority 2

pq.offer(3); // priority 1

System.out.println("Dequeued element: " + pq.poll()); // 10 (highest priority)

System.out.println("Priority Queue: " + pq);

// Test Case 2

pq.offer(7); // priority 4

pq.offer(6); // priority 3

pq.offer(8); // priority 2

System.out.println("Dequeued element: " + pq.poll()); // 7 (highest priority)

System.out.println("Priority Queue: " + pq);

}

}

Time Complexity:

* Insert (enqueue): O(log n)
* Remove (dequeue): O(log n)

Space Complexity:

* O(n)

Problem 14: Stack with getMin() in Constant Time

Solution:

java

Copy code

import java.util.Stack;

class MinStack {

private Stack<Integer> stack;

private Stack<Integer> minStack;

public MinStack() {

stack = new Stack<>();

minStack = new Stack<>();

}

public void push(int val) {

stack.push(val);

if (minStack.isEmpty() || val <= minStack.peek()) {

minStack.push(val);

}

}

public void pop() {

if (!stack.isEmpty()) {

int popped = stack.pop();

if (popped == minStack.peek()) {

minStack.pop();

}

}

}

public int getMin() {

return minStack.isEmpty() ? Integer.MAX\_VALUE : minStack.peek();

}

public static void main(String[] args) {

MinStack stack = new MinStack();

stack.push(5);

stack.push(3);

stack.push(7);

System.out.println("Min = " + stack.getMin()); // Min = 3

stack.push(10);

stack.push(8);

stack.push(6);

System.out.println("Min = " + stack.getMin()); // Min = 6

}

}

Time Complexity:

* Push/Pop/GetMin: O(1) (constant time)

Space Complexity:

* O(n) (for storing elements in both the stack and minStack)

Problem 15: Circular Queue with Fixed Size

Solution:

java

Copy code

class CircularQueue {

private int[] queue;

private int front, rear, size, capacity;

public CircularQueue(int capacity) {

this.capacity = capacity;

queue = new int[capacity];

front = 0;

size = 0;

rear = capacity - 1;

}

public boolean isFull() {

return size == capacity;

}

public boolean isEmpty() {

return size == 0;

}

public void enqueue(int value) {

if (isFull()) {

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

return;

}

rear = (rear + 1) % capacity;

queue[rear] = value;

size++;

}

public void dequeue() {

if (isEmpty()) {

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

return;

}

front = (front + 1) % capacity;

size--;

}

public static void main(String[] args) {

CircularQueue cq = new CircularQueue(4);

// Test Case 1

cq.enqueue(1);

cq.enqueue(2);

cq.enqueue(3);

cq.enqueue(4);

System.out.println("isFull: " + cq.isFull()); // True

// Test Case 2

CircularQueue cq2 = new CircularQueue(3);

cq2.enqueue(5);

cq2.enqueue(6);

cq2.dequeue();

cq2.enqueue(7);

System.out.println("isEmpty: " + cq2.isEmpty()); // False

}

}

Time Complexity:

* Enqueue/Dequeue: O(1)
* isFull/isEmpty: O(1)

Space Complexity:

* O(n), where n is the size of the queue