**Department of computer applications**

**National institute of technology,**

**Kurukshetra , Haryana**

**India**

****

**DSA lab-manual**

**[2024-2025]**

**SUBMITTED BY: submitted to:**

NAME:-Priyanshu Sharma Miss. Pooja

ROLL-NO:- 524410018

GROUP:- G2

SEMESTER:-2nd

1.Write a program to perform insert, delete and traverse operations on the singly linked list in the beginning, end and on any specific location.

class SinglyLinkedList {

// Node class

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

// Head node of the list

private Node head;

// Insert at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

// Insert at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

// Insert at specific position (0-based index)

public void insertAtPosition(int data, int position) {

if (position == 0) {

insertAtBeginning(data);

return;

}

Node newNode = new Node(data);

Node temp = head;

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

if (temp == null) {

System.out.println("Position out of bounds");

return;

}

temp = temp.next;

}

newNode.next = temp.next;

temp.next = newNode;

}

// Delete from beginning

public void deleteAtBeginning() {

if (head == null) {

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

return;

}

head = head.next;

}

// Delete from end

public void deleteAtEnd() {

if (head == null) {

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

return;

}

if (head.next == null) {

head = null;

return;

}

Node temp = head;

while (temp.next.next != null) {

temp = temp.next;

}

temp.next = null;

}

// Delete from specific position (0-based index)

public void deleteAtPosition(int position) {

if (head == null) {

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

return;

}

if (position == 0) {

deleteAtBeginning();

return;

}

Node temp = head;

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

if (temp == null || temp.next == null) {

System.out.println("Position out of bounds");

return;

}

temp = temp.next;

}

if (temp.next == null) {

System.out.println("Position out of bounds");

return;

}

temp.next = temp.next.next;

}

// Traverse (display) the linked list

public void traverse() {

if (head == null) {

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

return;

}

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

// Main function to test

public static void main(String[] args) {

SinglyLinkedList list = new SinglyLinkedList();

list.insertAtEnd(10);

list.insertAtBeginning(5);

list.insertAtEnd(15);

list.insertAtPosition(12, 2);

System.out.println("Linked List after insertions:");

list.traverse();

list.deleteAtBeginning();

list.deleteAtEnd();

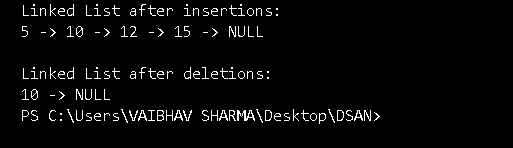
list.deleteAtPosition(1);

System.out.println("\nLinked List after deletions:");

list.traverse();

}

}



**2.** Write a program to rearrange the elements of a singly linked list in ascending or descending order.

import java.util.\*;

class SinglyLinkedListSort {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head;

// Insert at end

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newNode;

}

// Sort the linked list

public void sort(boolean ascending) {

if (head == null || head.next == null)

return;

for (Node i = head; i != null; i = i.next) {

for (Node j = i.next; j != null; j = j.next) {

if ((ascending && i.data > j.data) || (!ascending && i.data < j.data)) {

// Swap data

int temp = i.data;

i.data = j.data;

j.data = temp;

}

}

}

}

// Traverse the list

public void traverse() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

public static void main(String[] args) {

SinglyLinkedListSort list = new SinglyLinkedListSort();

list.insert(5);

list.insert(2);

list.insert(8);

list.insert(1);

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

list.traverse();

list.sort(true); // true = ascending

System.out.println("\nList after Ascending Sort:");

list.traverse();

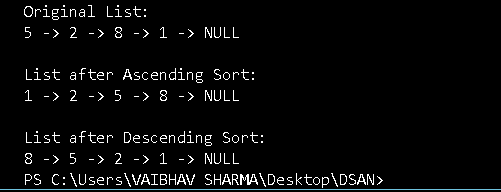
list.sort(false); // false = descending

System.out.println("\nList after Descending Sort:");

list.traverse();

}

}



**3.** Write a program to move the last node to the front of singly linked list.

class SinglyLinkedListMoveLastToFront {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head;

// Insert at end

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newNode;

}

// Move last node to front

public void moveLastToFront() {

if (head == null || head.next == null)

return;

Node secondLast = null;

Node last = head;

while (last.next != null) {

secondLast = last;

last = last.next;

}

secondLast.next = null;

last.next = head;

head = last;

}

// Traverse the list

public void traverse() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

public static void main(String[] args) {

SinglyLinkedListMoveLastToFront list = new SinglyLinkedListMoveLastToFront();

list.insert(10);

list.insert(20);

list.insert(30);

list.insert(40);

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

list.traverse();

list.moveLastToFront();

System.out.println("\nList after moving last node to front:");

list.traverse();

}

}

**4.** Write a program to print the elements of singly link list using recursion.

class SinglyLinkedListRecursivePrint {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head;

// Insert at end

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newNode;

}

// Recursive function to print list

public void printRecursively(Node node) {

if (node == null) {

System.out.println("NULL");

return;

}

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

printRecursively(node.next);

}

public void startPrint() {

printRecursively(head);

}

public static void main(String[] args) {

SinglyLinkedListRecursivePrint list = new SinglyLinkedListRecursivePrint();

list.insert(1);

list.insert(2);

list.insert(3);

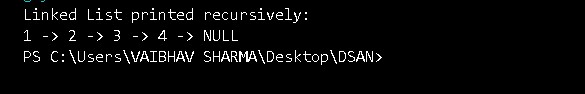
list.insert(4);

System.out.println("Linked List printed recursively:");

list.startPrint();

}

}



5. write a java code to reverse a linked list using iteration technique. this is question

class ReverseLinkedListIterative {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

Node head;

// Function to insert a new node at the end

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newNode;

}

// Function to reverse the linked list iteratively

public void reverse() {

Node prev = null;

Node current = head;

Node next = null;

while (current != null) {

next = current.next; // Save next node

current.next = prev; // Reverse current node's pointer

prev = current; // Move prev and current one step forward

current = next;

}

head = prev;

}

// Function to print the linked list

public void traverse() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

public static void main(String[] args) {

ReverseLinkedListIterative list = new ReverseLinkedListIterative();

list.insert(10);

list.insert(20);

list.insert(30);

list.insert(40);

System.out.println("Original Linked List:");

list.traverse();

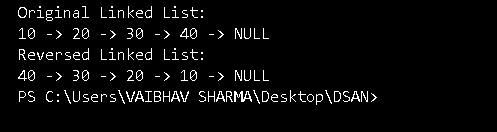
list.reverse();

System.out.println("Reversed Linked List:");

list.traverse();

}

}



6. write a program to reverse a linked list using recursion.

class ReverseLinkedListRecursive {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

Node head;

// Function to insert a new node at the end

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newNode;

}

// Recursive function to reverse the linked list

Node reverseRecursive(Node current) {

// Base case: if head is null or only one node

if (current == null || current.next == null) {

return current;

}

Node newHead = reverseRecursive(current.next);

current.next.next = current;

current.next = null;

return newHead;

}

// Function to start recursion and update head

public void reverse() {

head = reverseRecursive(head);

}

// Function to print the linked list

public void traverse() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

public static void main(String[] args) {

ReverseLinkedListRecursive list = new ReverseLinkedListRecursive();

list.insert(1);

list.insert(2);

list.insert(3);

list.insert(4);

list.insert(5);

System.out.println("Original Linked List:");

list.traverse();

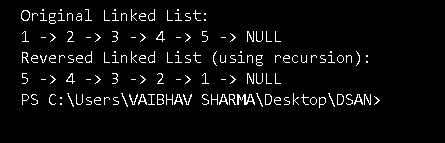
list.reverse();

System.out.println("Reversed Linked List (using recursion):");

list.traverse();

}

}



7. write a java program to implement a circular linked list.

class CircularLinkedList {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

Node last = null;

// Function to add a node to the empty list

public void addToEmpty(int data) {

if (last != null) {

return;

}

Node newNode = new Node(data);

last = newNode;

last.next = last; // Point to itself

}

// Function to add node at the end

public void addEnd(int data) {

if (last == null) {

addToEmpty(data);

return;

}

Node newNode = new Node(data);

newNode.next = last.next;

last.next = newNode;

last = newNode;

}

// Function to add node at the beginning

public void addBegin(int data) {

if (last == null) {

addToEmpty(data);

return;

}

Node newNode = new Node(data);

newNode.next = last.next;

last.next = newNode;

}

// Function to traverse the Circular Linked List

public void traverse() {

if (last == null) {

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

return;

}

Node temp = last.next;

do {

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

temp = temp.next;

} while (temp != last.next);

System.out.println("(back to head)");

}

public static void main(String[] args) {

CircularLinkedList cll = new CircularLinkedList();

cll.addToEmpty(10);

cll.addEnd(20);

cll.addEnd(30);

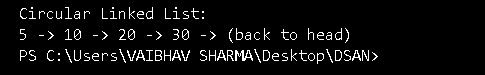
cll.addBegin(5);

System.out.println("Circular Linked List:");

cll.traverse();

}

}



8. write a program to check give linked list is is non- descending order or not.

class CheckNonDescendingOrder {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

Node head;

// Function to insert a new node at the end

public void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newNode;

}

// Function to check if the linked list is in non-descending order

public boolean isNonDescending() {

if (head == null || head.next == null) {

return true; // Empty or single-node list is trivially sorted

}

Node temp = head;

while (temp.next != null) {

if (temp.data > temp.next.data) {

return false; // Found a descending pair

}

temp = temp.next;

}

return true;

}

// Function to traverse and display the linked list

public void traverse() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

public static void main(String[] args) {

CheckNonDescendingOrder list = new CheckNonDescendingOrder();

list.insert(10);

list.insert(20);

list.insert(20);

list.insert(30);

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

list.traverse();

if (list.isNonDescending()) {

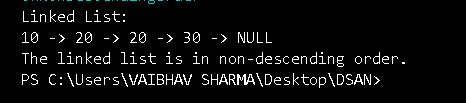
System.out.println("The linked list is in non-descending order.");

} else {

System.out.println("The linked list is NOT in non-descending order.");

}

}

} 

9.write a program to insert , delete , traverse on doubly linked list in the begin at the end and on any specific location.

class DoublyLinkedList {

// Node class to represent each element

class Node {

int data;

Node prev;

Node next;

public Node(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

// Head and tail pointers

Node head, tail;

// Constructor to initialize the doubly linked list

public DoublyLinkedList() {

head = tail = null;

}

// Insert at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

if (head == null) { // If the list is empty

head = tail = newNode;

} else {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

}

// Insert at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (tail == null) { // If the list is empty

head = tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

// Insert at a specific position

public void insertAtPosition(int data, int position) {

if (position < 1) {

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

return;

}

Node newNode = new Node(data);

Node current = head;

int count = 1;

// If inserting at the beginning

if (position == 1) {

insertAtBeginning(data);

return;

}

while (current != null && count < position - 1) {

current = current.next;

count++;

}

// If position is greater than the length of the list

if (current == null) {

System.out.println("Position exceeds list length");

return;

}

newNode.next = current.next;

if (current.next != null) {

current.next.prev = newNode;

}

current.next = newNode;

newNode.prev = current;

}

// Delete from the beginning

public void deleteFromBeginning() {

if (head == null) {

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

return;

}

if (head == tail) { // Only one element in the list

head = tail = null;

} else {

head = head.next;

head.prev = null;

}

}

// Delete from the end

public void deleteFromEnd() {

if (tail == null) {

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

return;

}

if (head == tail) { // Only one element in the list

head = tail = null;

} else {

tail = tail.prev;

tail.next = null;

}

}

// Delete from a specific position

public void deleteAtPosition(int position) {

if (head == null || position < 1) {

System.out.println("Invalid position or empty list");

return;

}

Node current = head;

int count = 1;

// If deleting the first node

if (position == 1) {

deleteFromBeginning();

return;

}

while (current != null && count < position) {

current = current.next;

count++;

}

// If position is greater than the length of the list

if (current == null) {

System.out.println("Position exceeds list length");

return;

}

// If deleting the last node

if (current == tail) {

deleteFromEnd();

return;

}

// Re-link the previous and next nodes

current.prev.next = current.next;

current.next.prev = current.prev;

}

// Traverse and print the list

public void traverse() {

if (head == null) {

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

return;

}

Node current = head;

while (current != null) {

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

current = current.next;

}

System.out.println("null");

}

public static void main(String[] args) {

DoublyLinkedList list = new DoublyLinkedList();

// Insert elements at different positions

list.insertAtBeginning(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtBeginning(5);

list.insertAtPosition(15, 3); // Insert 15 at position 3

// Traverse the list

System.out.println("List after insertions:");

list.traverse();

// Delete elements from different positions

list.deleteFromBeginning();

list.deleteFromEnd();

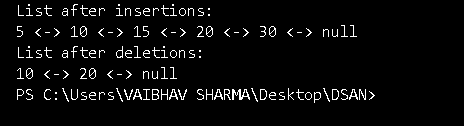
list.deleteAtPosition(2); // Delete element at position 2

// Traverse the list again

System.out.println("List after deletions:");

list.traverse();

}

} 

10. write a program to implement stack push and pop operation using array.

class Stack {

private int[] stack;

private int top;

private int capacity;

// Constructor to initialize the stack

public Stack(int size) {

capacity = size;

stack = new int[capacity];

top = -1; // Stack is initially empty

}

// Push operation to insert an element into the stack

public void push(int data) {

if (top == capacity - 1) {

System.out.println("Stack Overflow! Unable to push " + data);

} else {

stack[++top] = data; // Increment top and insert element

System.out.println("Pushed " + data);

}

}

// Pop operation to remove the top element from the stack

public void pop() {

if (top == -1) {

System.out.println("Stack Underflow! Unable to pop");

} else {

int poppedElement = stack[top--]; // Remove element and decrement top

System.out.println("Popped " + poppedElement);

}

}

// Display the elements in the stack

public void display() {

if (top == -1) {

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

} else {

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

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

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

}

System.out.println();

}

}

}

public class StackUsingArray {

public static void main(String[] args) {

// Create a stack with a maximum size of 5

Stack stack = new Stack(5);

// Perform stack operations

stack.push(10);

stack.push(20);

stack.push(30);

stack.push(40);

stack.push(50);

// Display the stack

stack.display();

// Attempt to push beyond the capacity

stack.push(60);

// Perform pop operations

stack.pop();

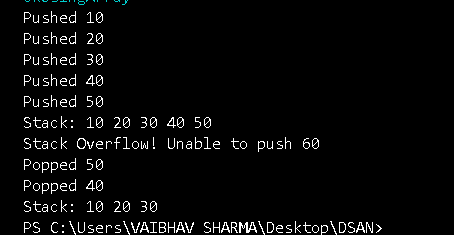
stack.pop();

// Display the stack after popping elements

stack.display();

}

}



11. write a program to implement stack using linked list.

class Stack {

// Node class to represent each element in the stack

class Node {

int data;

Node next;

// Constructor to initialize a new node

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node top; // Top node of the stack

// Constructor to initialize the stack

public Stack() {

top = null;

}

// Push operation to insert an element into the stack

public void push(int data) {

Node newNode = new Node(data);

if (top == null) {

top = newNode; // If stack is empty, new node becomes the top

} else {

newNode.next = top; // Point new node to the current top

top = newNode; // Make new node the top

}

System.out.println("Pushed " + data);

}

// Pop operation to remove the top element from the stack

public void pop() {

if (top == null) {

System.out.println("Stack Underflow! Unable to pop");

} else {

int poppedData = top.data; // Get the top element

top = top.next; // Move the top to the next node

System.out.println("Popped " + poppedData);

}

}

// Display the elements in the stack

public void display() {

if (top == null) {

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

} else {

Node current = top;

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

while (current != null) {

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

current = current.next;

}

System.out.println();

}

}

}

public class StackUsingLinkedList {

public static void main(String[] args) {

// Create a stack object

Stack stack = new Stack();

// Perform stack operations

stack.push(10);

stack.push(20);

stack.push(30);

stack.push(40);

stack.push(50);

// Display the stack

stack.display();

// Perform pop operations

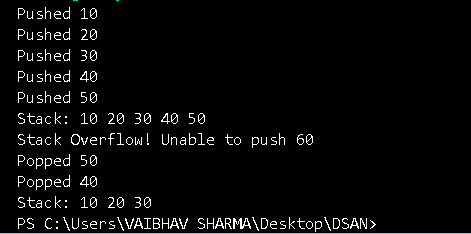
stack.pop();

stack.pop();

// Display the stack after popping elements

stack.display();

}

} 

12. write a program to implement a queue using circular array.

class Queue {

private int[] queue;

private int front, rear, size, capacity;

// Constructor to initialize the queue with a given capacity

public Queue(int capacity) {

this.capacity = capacity;

queue = new int[capacity];

front = rear = -1; // Initially both front and rear are -1

size = 0; // No elements initially

}

// Check if the queue is full

public boolean isFull() {

return size == capacity;

}

// Check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Enqueue operation to add an element to the queue

public void enqueue(int data) {

if (isFull()) {

System.out.println("Queue Overflow! Unable to enqueue " + data);

} else {

if (front == -1) {

front = 0; // First element is inserted

}

rear = (rear + 1) % capacity; // Circular increment

queue[rear] = data;

size++;

System.out.println("Enqueued " + data);

}

}

// Dequeue operation to remove an element from the queue

public void dequeue() {

if (isEmpty()) {

System.out.println("Queue Underflow! Unable to dequeue");

} else {

int dequeuedData = queue[front];

front = (front + 1) % capacity; // Circular increment

size--;

System.out.println("Dequeued " + dequeuedData);

}

}

// Display the elements in the queue

public void display() {

if (isEmpty()) {

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

} else {

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

int count = size;

int i = front;

while (count > 0) {

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

i = (i + 1) % capacity; // Circular increment

count--;

}

System.out.println();

}

}

}

public class CircularQueue {

public static void main(String[] args) {

// Create a queue with a capacity of 5

Queue queue = new Queue(5);

// Perform queue operations

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

queue.enqueue(50);

// Display the queue

queue.display();

// Attempt to enqueue when the queue is full

queue.enqueue(60);

// Perform dequeue operations

queue.dequeue();

queue.dequeue();

// Display the queue after dequeuing

queue.display();

// Enqueue again after dequeuing some elements

queue.enqueue(60);

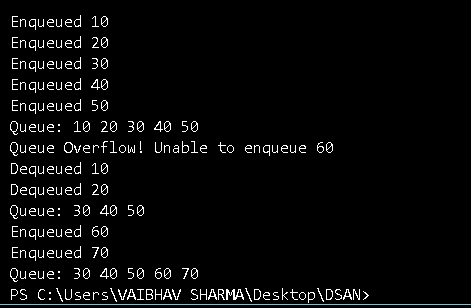
queue.enqueue(70);

// Final display of the queue

queue.display();

}

}



13. write a program to implement a queue using circular linked list.

class Queue {

// Node class to represent each element in the queue

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node front, rear;

// Constructor to initialize the queue

public Queue() {

front = rear = null; // Initially both front and rear are null

}

// Check if the queue is empty

public boolean isEmpty() {

return front == null;

}

// Enqueue operation to add an element to the queue

public void enqueue(int data) {

Node newNode = new Node(data);

if (isEmpty()) {

front = rear = newNode;

rear.next = front; // Point the last node to the front (circular)

} else {

rear.next = newNode; // Link the new node to the last node

rear = newNode; // Update the rear to the new node

rear.next = front; // Circular link: last node points to front

}

System.out.println("Enqueued " + data);

}

// Dequeue operation to remove an element from the queue

public void dequeue() {

if (isEmpty()) {

System.out.println("Queue Underflow! Unable to dequeue");

} else {

int dequeuedData = front.data; // Get the data from the front node

if (front == rear) {

front = rear = null; // If only one element was in the queue

} else {

front = front.next; // Move front to the next node

rear.next = front; // Circular link: rear points to new front

}

System.out.println("Dequeued " + dequeuedData);

}

}

// Display the elements in the queue

public void display() {

if (isEmpty()) {

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

} else {

Node current = front;

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

do {

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

current = current.next;

} while (current != front); // Circular condition: stop when we reach the front again

System.out.println();

}

}

}

public class CircularQueue {

public static void main(String[] args) {

// Create a queue object

Queue queue = new Queue();

// Perform queue operations

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

queue.enqueue(50);

// Display the queue

queue.display();

// Perform dequeue operations

queue.dequeue();

queue.dequeue();

// Display the queue after dequeuing

queue.display();

// Enqueue again after dequeuing some elements

queue.enqueue(60);

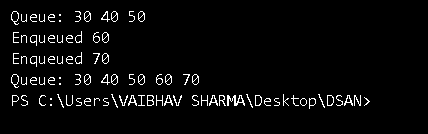
queue.enqueue(70);

// Final display of the queue

queue.display();

}

}



14. write a program to implement stack using queue.

import java.util.LinkedList;

import java.util.Queue;

class StackUsingQueues {

private Queue<Integer> queue1;

private Queue<Integer> queue2;

// Constructor to initialize two queues

public StackUsingQueues() {

queue1 = new LinkedList<>();

queue2 = new LinkedList<>();

}

// Push operation to add an element to the stack

public void push(int data) {

// Push data into queue2

queue2.add(data);

// Push all elements of queue1 into queue2 to maintain the order

while (!queue1.isEmpty()) {

queue2.add(queue1.poll());

}

// Swap the names of the two queues so that queue1 always holds the elements of the stack

Queue<Integer> temp = queue1;

queue1 = queue2;

queue2 = temp;

System.out.println("Pushed " + data);

}

// Pop operation to remove the top element from the stack

public void pop() {

if (queue1.isEmpty()) {

System.out.println("Stack Underflow! Unable to pop");

} else {

int poppedData = queue1.poll();

System.out.println("Popped " + poppedData);

}

}

// Peek operation to view the top element of the stack

public int peek() {

if (queue1.isEmpty()) {

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

return -1;

} else {

return queue1.peek();

}

}

// Display the elements in the stack

public void display() {

if (queue1.isEmpty()) {

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

} else {

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

for (int element : queue1) {

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

}

System.out.println();

}

}

}

public class StackUsingTwoQueues {

public static void main(String[] args) {

// Create a stack object

StackUsingQueues stack = new StackUsingQueues();

// Perform stack operations

stack.push(10);

stack.push(20);

stack.push(30);

stack.push(40);

stack.push(50);

// Display the stack

stack.display();

// Perform pop operations

stack.pop();

stack.pop();

// Display the stack after popping elements

stack.display();

// Peek operation

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

// Enqueue again after popping some elements

stack.push(60);

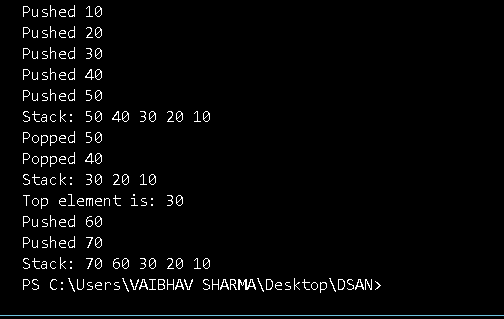
stack.push(70);

// Final display of the stack

stack.display();

}

}



15. write a program to implement a queue using two stacks.

import java.util.Stack;

class QueueUsingTwoStacks {

private Stack<Integer> stack1; // Stack for enqueue operations

private Stack<Integer> stack2; // Stack for dequeue operations

// Constructor to initialize the two stacks

public QueueUsingTwoStacks() {

stack1 = new Stack<>();

stack2 = new Stack<>();

}

// Enqueue operation to add an element to the queue

public void enqueue(int data) {

stack1.push(data);

System.out.println("Enqueued " + data);

}

// Dequeue operation to remove an element from the queue

public void dequeue() {

if (stack1.isEmpty() && stack2.isEmpty()) {

System.out.println("Queue Underflow! Unable to dequeue");

return;

}

// If stack2 is empty, transfer elements from stack1 to stack2

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

// If stack2 is not empty, pop from stack2

if (!stack2.isEmpty()) {

int dequeuedData = stack2.pop();

System.out.println("Dequeued " + dequeuedData);

}

}

// Peek operation to view the front element of the queue

public int peek() {

if (stack1.isEmpty() && stack2.isEmpty()) {

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

return -1;

}

// If stack2 is empty, transfer elements from stack1 to stack2

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

// The top element of stack2 is the front of the queue

return stack2.peek();

}

// Display the elements in the queue

public void display() {

if (stack1.isEmpty() && stack2.isEmpty()) {

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

} else {

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

// Display elements in stack2 first

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

while (!stack2.isEmpty()) {

tempStack.push(stack2.pop());

}

while (!tempStack.isEmpty()) {

System.out.print(tempStack.pop() + " ");

}

// Display elements in stack1

for (int element : stack1) {

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

}

System.out.println();

}

}

}

public class QueueUsingTwoStacks {

public static void main(String[] args) {

// Create a queue object

QueueUsingTwoStacks queue = new QueueUsingTwoStacks();

// Perform queue operations

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

queue.enqueue(50);

// Display the queue

queue.display();

// Perform dequeue operations

queue.dequeue();

queue.dequeue();

// Display the queue after dequeuing

queue.display();

// Peek operation

System.out.println("Front element is: " + queue.peek());

// Enqueue again after dequeuing some elements

queue.enqueue(60);

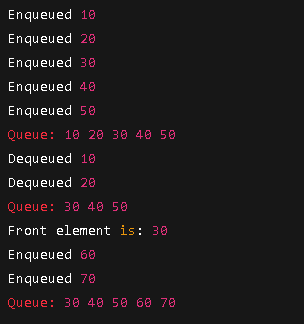
queue.enqueue(70);

// Final display of the queue

queue.display();

}

}



16. write a program to convert infix notation to postfix.

import java.util.Stack;

class InfixToPostfix {

// Function to get the precedence of operators

private static int precedence(char c) {

if (c == '+' || c == '-') {

return 1;

} else if (c == '\*' || c == '/') {

return 2;

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

return 3;

}

return -1; // For non-operators

}

// Function to check if a character is an operand (a letter or a digit)

private static boolean isOperand(char c) {

return Character.isLetterOrDigit(c);

}

// Function to convert the infix expression to postfix

public static String infixToPostfix(String infix) {

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

StringBuilder postfix = new StringBuilder();

// Loop through the infix expression

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

char c = infix.charAt(i);

if (isOperand(c)) {

// If it's an operand, append it to the result

postfix.append(c);

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

// If it's an opening parenthesis, push it to the stack

stack.push(c);

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

// If it's a closing parenthesis, pop from the stack until '(' is found

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

postfix.append(stack.pop());

}

stack.pop(); // Pop the '(' from the stack

} else {

// If it's an operator

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

postfix.append(stack.pop()); // Pop operators from stack if they have higher or equal precedence

}

stack.push(c); // Push the current operator to the stack

}

}

// Pop all the operators from the stack

while (!stack.isEmpty()) {

postfix.append(stack.pop());

}

return postfix.toString();

}

public static void main(String[] args) {

String infix = "A+B\*(C^D-E)^(F+G\*H)-I";

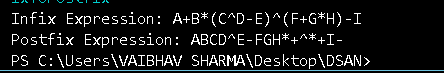
System.out.println("Infix Expression: " + infix);

String postfix = infixToPostfix(infix);

System.out.println("Postfix Expression: " + postfix);

}

}



17. write a program to evaluate a postfix expression.

import java.util.Stack;

class PostfixEvaluation {

// Function to check if a character is an operator

private static boolean isOperator(char c) {

return c == '+' || c == '-' || c == '\*' || c == '/';

}

// Function to perform arithmetic operations

private static int applyOperator(char operator, int operand1, int operand2) {

switch (operator) {

case '+': return operand1 + operand2;

case '-': return operand1 - operand2;

case '\*': return operand1 \* operand2;

case '/': return operand1 / operand2;

default: throw new IllegalArgumentException("Invalid operator");

}

}

// Function to evaluate a postfix expression

public static int evaluatePostfix(String expression) {

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

// Loop through each character of the postfix expression

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

char c = expression.charAt(i);

if (Character.isDigit(c)) {

// If it's a digit, push it to the stack

stack.push(c - '0'); // Convert char to int

} else if (isOperator(c)) {

// If it's an operator, pop two operands, apply the operator, and push the result

int operand2 = stack.pop();

int operand1 = stack.pop();

int result = applyOperator(c, operand1, operand2);

stack.push(result);

}

}

// The result should be the only element left in the stack

return stack.pop();

}

public static void main(String[] args) {

String postfixExpression = "23\*+5-"; // Example: "2 3 4 \* + 5 -"

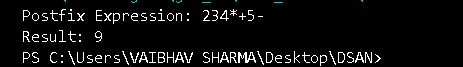
System.out.println("Postfix Expression: " + postfixExpression);

int result = evaluatePostfix(postfixExpression);

System.out.println("Result: " + result);

}

}



18. write a program to find out the preorder, inorder and postorder traversal of the tree.

// Program to perform Preorder, Inorder, and Postorder Traversal of a Binary Tree

class Node {

int data;

Node left, right;

// Constructor

public Node(int item) {

data = item;

left = right = null;

}

}

class BinaryTree {

// Root of Binary Tree

Node root;

// Constructor

BinaryTree() {

root = null;

}

// Function for Inorder Traversal (Left, Root, Right)

void inorder(Node node) {

if (node == null)

return;

inorder(node.left);

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

inorder(node.right);

}

// Function for Preorder Traversal (Root, Left, Right)

void preorder(Node node) {

if (node == null)

return;

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

preorder(node.left);

preorder(node.right);

}

// Function for Postorder Traversal (Left, Right, Root)

void postorder(Node node) {

if (node == null)

return;

postorder(node.left);

postorder(node.right);

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

}

public static void main(String[] args) {

BinaryTree tree = new BinaryTree();

/\* Let's create the following tree

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

System.out.println("Inorder traversal:");

tree.inorder(tree.root);

System.out.println("\nPreorder traversal:");

tree.preorder(tree.root);

System.out.println("\nPostorder traversal:");

tree.postorder(tree.root);

}

}

19. write a program to perform double-order traversal and triple-order traversal on the tree.

// Program to perform Double-order and Triple-order Traversal of a Binary Tree

class Node {

int data;

Node left, right;

// Constructor

public Node(int item) {

data = item;

left = right = null;

}

}

class BinaryTreeTraversal {

Node root;

BinaryTreeTraversal() {

root = null;

}

// Double-order Traversal (Visit Node twice)

void doubleOrderTraversal(Node node) {

if (node == null)

return;

// Visit before left subtree

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

doubleOrderTraversal(node.left);

// Visit again after left subtree

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

doubleOrderTraversal(node.right);

}

// Triple-order Traversal (Visit Node three times)

void tripleOrderTraversal(Node node) {

if (node == null)

return;

// Visit before left subtree

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

tripleOrderTraversal(node.left);

// Visit between left and right subtree

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

tripleOrderTraversal(node.right);

// Visit after right subtree

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

}

public static void main(String[] args) {

BinaryTreeTraversal tree = new BinaryTreeTraversal();

/\* Tree Structure

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

System.out.println("Double-order Traversal:");

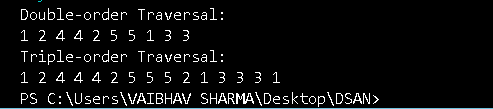
tree.doubleOrderTraversal(tree.root);

System.out.println("\nTriple-order Traversal:");

tree.tripleOrderTraversal(tree.root);

}

}



20. Write a program to find the number of binary trees possible with given number of nodes.

// Program to find number of binary trees possible with given number of nodes

import java.util.Scanner;

class CatalanNumber {

// Function to find factorial

static long factorial(int n) {

long result = 1;

for (int i = 2; i <= n; i++) {

result \*= i;

}

return result;

}

// Function to find Catalan Number using factorial

static long catalanNumber(int n) {

long fact2n = factorial(2 \* n);

long factn1 = factorial(n + 1);

long factn = factorial(n);

return fact2n / (factn1 \* factn);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of nodes: ");

int n = sc.nextInt();

long numberOfBinaryTrees = catalanNumber(n);

System.out.println("Number of Binary Trees possible with " + n + " nodes = " + numberOfBinaryTrees);

sc.close();

}

}



21. Write a program to perform indirect recursion on the tree.

// Program to perform Indirect Recursion on Binary Tree

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class IndirectRecursionTree {

Node root;

IndirectRecursionTree() {

root = null;

}

// First function for indirect recursion

void indirectA(Node node) {

if (node == null)

return;

System.out.print(node.data + " "); // Process node

indirectB(node.left); // Call B on left child

}

// Second function for indirect recursion

void indirectB(Node node) {

if (node == null)

return;

System.out.print(node.data + " "); // Process node

indirectA(node.right); // Call A on right child

}

public static void main(String[] args) {

IndirectRecursionTree tree = new IndirectRecursionTree();

/\* Tree Structure

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

System.out.println("Indirect Recursive Traversal:");

tree.indirectA(tree.root);

}

}



22. Write a program to find out possible labelled and unlabeled binary trees with the given number of nodes.

// Program to find possible Labelled and Unlabelled Binary Trees with given number of nodes

import java.util.Scanner;

class LabelledUnlabelledBinaryTrees {

// Function to find factorial

static long factorial(int n) {

long result = 1;

for (int i = 2; i <= n; i++) {

result \*= i;

}

return result;

}

// Function to find Catalan number

static long catalanNumber(int n) {

long fact2n = factorial(2 \* n);

long factn1 = factorial(n + 1);

long factn = factorial(n);

return fact2n / (factn1 \* factn);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of nodes: ");

int n = sc.nextInt();

long catalan = catalanNumber(n);

long labelledTrees = catalan \* factorial(n);

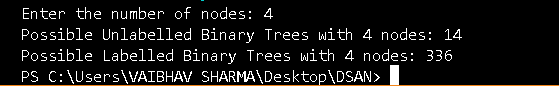
System.out.println("Possible Unlabelled Binary Trees with " + n + " nodes: " + catalan);

System.out.println("Possible Labelled Binary Trees with " + n + " nodes: " + labelledTrees);

sc.close();

}

}



23. Write a program to construct the unique binary tree using inorder and preorder traversal and hence find postorder.

// Program to construct unique binary tree using Inorder and Preorder Traversal and find Postorder

import java.util.HashMap;

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class ConstructTree {

Node root;

int preIndex = 0;

HashMap<Integer, Integer> inorderMap = new HashMap<>();

// Function to build tree

Node buildTree(int[] preorder, int[] inorder, int start, int end) {

if (start > end)

return null;

int current = preorder[preIndex++];

Node node = new Node(current);

if (start == end)

return node;

int inIndex = inorderMap.get(current);

node.left = buildTree(preorder, inorder, start, inIndex - 1);

node.right = buildTree(preorder, inorder, inIndex + 1, end);

return node;

}

// Function to print Postorder traversal

void printPostorder(Node node) {

if (node == null)

return;

printPostorder(node.left);

printPostorder(node.right);

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

}

public static void main(String[] args) {

ConstructTree tree = new ConstructTree();

// Example Data

int[] preorder = {1, 2, 4, 5, 3};

int[] inorder = {4, 2, 5, 1, 3};

// Build a map for fast lookup

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

tree.inorderMap.put(inorder[i], i);

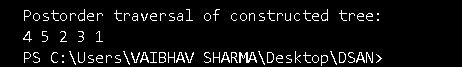
}

tree.root = tree.buildTree(preorder, inorder, 0, inorder.length - 1);

System.out.println("Postorder traversal of constructed tree:");

tree.printPostorder(tree.root);

}

}

24. Write a recursive program to count the total number of nodes in the tree.

// Program to count total number of nodes in a Binary Tree recursively

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class CountNodes {

Node root;

// Recursive function to count nodes

int countNodes(Node node) {

if (node == null)

return 0;

return 1 + countNodes(node.left) + countNodes(node.right);

}

public static void main(String[] args) {

CountNodes tree = new CountNodes();

/\* Example Tree

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

int totalNodes = tree.countNodes(tree.root);

System.out.println("Total number of nodes in the tree: " + totalNodes);

}

}



25. write a recursive program to count the number of the leaf or non-leaf nodes of the tree.

// Program to count leaf and non-leaf nodes in a Binary Tree recursively

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class CountLeafNonLeaf {

Node root;

int leafCount = 0;

int nonLeafCount = 0;

// Recursive function to count leaf and non-leaf nodes

void countNodes(Node node) {

if (node == null)

return;

if (node.left == null && node.right == null) {

leafCount++;

} else {

nonLeafCount++;

}

countNodes(node.left);

countNodes(node.right);

}

public static void main(String[] args) {

CountLeafNonLeaf tree = new CountLeafNonLeaf();

/\* Example Tree

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

tree.countNodes(tree.root);

System.out.println("Number of Leaf nodes: " + tree.leafCount);

System.out.println("Number of Non-Leaf nodes: " + tree.nonLeafCount);

}

}



26. write a recursive program to count the number of full nodes of the tree (Full Nodes are nodes which has both left and right children as non-empty).

// Program to count full nodes in a Binary Tree recursively

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class CountFullNodes {

Node root;

int fullNodeCount = 0;

// Recursive function to count full nodes

void countFullNodes(Node node) {

if (node == null)

return;

if (node.left != null && node.right != null) {

fullNodeCount++;

}

countFullNodes(node.left);

countFullNodes(node.right);

}

public static void main(String[] args) {

CountFullNodes tree = new CountFullNodes();

/\* Example Tree

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

tree.countFullNodes(tree.root);

System.out.println("Number of Full nodes: " + tree.fullNodeCount);

}

}

27. write a recursive program to find the height of the tree.

// Program to find height of a Binary Tree recursively

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class TreeHeight {

Node root;

// Recursive function to find height of tree

int findHeight(Node node) {

if (node == null)

return -1; // Return -1 if we consider height of empty tree as -1

int leftHeight = findHeight(node.left);

int rightHeight = findHeight(node.right);

// Return max of left height and right height plus 1 for the current node

return 1 + Math.max(leftHeight, rightHeight);

}

public static void main(String[] args) {

TreeHeight tree = new TreeHeight();

/\* Example Tree

1

/ \

2 3

/ \

4 5

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

int height = tree.findHeight(tree.root);

System.out.println("Height of the tree: " + height);

}

}



28. Write a program to construct BST using inorder and postorder traversal and hence find preorder.

import java.util.HashMap;

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class BSTFromInorderPostorder {

Node root;

int postIndex;

// Function to construct BST from inorder and postorder

Node buildTree(int[] inorder, int[] postorder, int inorderStart, int inorderEnd, HashMap<Integer, Integer> inorderMap) {

if (inorderStart > inorderEnd)

return null;

// The current root is the last element in postorder

int rootValue = postorder[postIndex--];

Node root = new Node(rootValue);

// Find root in inorder

int inorderIndex = inorderMap.get(rootValue);

// Recursively construct the right and left subtrees

root.right = buildTree(inorder, postorder, inorderIndex + 1, inorderEnd, inorderMap);

root.left = buildTree(inorder, postorder, inorderStart, inorderIndex - 1, inorderMap);

return root;

}

// Function to print Preorder traversal of the tree

void printPreorder(Node node) {

if (node == null)

return;

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

printPreorder(node.left);

printPreorder(node.right);

}

public static void main(String[] args) {

BSTFromInorderPostorder tree = new BSTFromInorderPostorder();

// Example Data

int[] inorder = {4, 2, 5, 1, 3};

int[] postorder = {4, 5, 2, 3, 1};

// Create map for fast lookup of inorder indices

HashMap<Integer, Integer> inorderMap = new HashMap<>();

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

inorderMap.put(inorder[i], i);

}

// Set the postIndex (index of last element in postorder)

tree.postIndex = postorder.length - 1;

// Build the tree

tree.root = tree.buildTree(inorder, postorder, 0, inorder.length - 1, inorderMap);

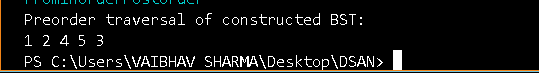
// Print Preorder traversal

System.out.println("Preorder traversal of constructed BST:");

tree.printPreorder(tree.root);

}

}



29. Write a program to find how many BSTs are possible with given distinct keys.

class CountBST {

// Function to calculate number of BSTs using Catalan number

int countBST(int n) {

int[] catalan = new int[n + 1];

// Base case

catalan[0] = catalan[1] = 1;

// Calculate Catalan numbers using DP

for (int i = 2; i <= n; i++) {

catalan[i] = 0;

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

catalan[i] += catalan[j] \* catalan[i - j - 1];

}

}

// The nth Catalan number is the number of unique BSTs

return catalan[n];

}

public static void main(String[] args) {

CountBST tree = new CountBST();

int n = 5; // Given number of distinct keys

int result = tree.countBST(n);

System.out.println("Number of unique BSTs that can be formed with " + n + " distinct keys: " + result);

}

}



30. Write a program to find out the postorder, preorder and inorder traversal on constructed bst and then perform delete operation on the tree and again perform inorder traversal.

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class BSTOperations {

Node root;

// Preorder traversal

void preorder(Node node) {

if (node == null)

return;

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

preorder(node.left);

preorder(node.right);

}

// Inorder traversal

void inorder(Node node) {

if (node == null)

return;

inorder(node.left);

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

inorder(node.right);

}

// Postorder traversal

void postorder(Node node) {

if (node == null)

return;

postorder(node.left);

postorder(node.right);

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

}

// Function to find the node with minimum value (used in deletion)

Node minValueNode(Node node) {

Node current = node;

while (current.left != null) {

current = current.left;

}

return current;

}

// Function to delete a node from the BST

Node deleteNode(Node root, int key) {

// Base case: If the tree is empty

if (root == null)

return root;

// Otherwise, recur down the tree

if (key < root.data)

root.left = deleteNode(root.left, key);

else if (key > root.data)

root.right = deleteNode(root.right, key);

else {

// Node to be deleted found

// Case 1: Node has only one child or no child

if (root.left == null)

return root.right;

else if (root.right == null)

return root.left;

// Case 2: Node has two children

// Get the inorder successor (smallest in the right subtree)

Node temp = minValueNode(root.right);

// Copy the inorder successor's content to this node

root.data = temp.data;

// Delete the inorder successor

root.right = deleteNode(root.right, temp.data);

}

return root;

}

public static void main(String[] args) {

BSTOperations tree = new BSTOperations();

/\* Example BST

50

/ \

30 70

/ \ / \

20 40 60 80

\*/

tree.root = new Node(50);

tree.root.left = new Node(30);

tree.root.right = new Node(70);

tree.root.left.left = new Node(20);

tree.root.left.right = new Node(40);

tree.root.right.left = new Node(60);

tree.root.right.right = new Node(80);

// Perform pre-order, in-order, and post-order traversals

System.out.println("Preorder traversal:");

tree.preorder(tree.root);

System.out.println("\nInorder traversal:");

tree.inorder(tree.root);

System.out.println("\nPostorder traversal:");

tree.postorder(tree.root);

// Perform delete operation

int keyToDelete = 20;

System.out.println("\n\nDeleting node with key " + keyToDelete);

tree.root = tree.deleteNode(tree.root, keyToDelete);

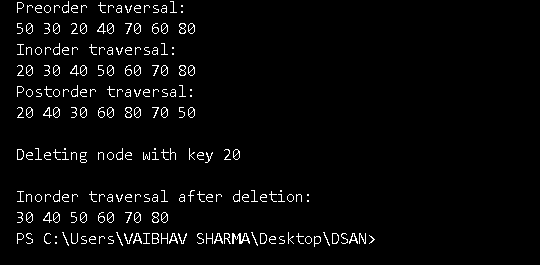
// Perform inorder traversal after deletion

System.out.println("\nInorder traversal after deletion:");

tree.inorder(tree.root);

}

}



31. Write a program to find the minimum and maximum key values from BSTss.

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class MinMaxBST {

Node root;

// Function to find the minimum value node

Node findMin(Node node) {

Node current = node;

while (current.left != null) {

current = current.left;

}

return current;

}

// Function to find the maximum value node

Node findMax(Node node) {

Node current = node;

while (current.right != null) {

current = current.right;

}

return current;

}

public static void main(String[] args) {

MinMaxBST tree = new MinMaxBST();

/\* Example BST

50

/ \

30 70

/ \ / \

20 40 60 80

\*/

tree.root = new Node(50);

tree.root.left = new Node(30);

tree.root.right = new Node(70);

tree.root.left.left = new Node(20);

tree.root.left.right = new Node(40);

tree.root.right.left = new Node(60);

tree.root.right.right = new Node(80);

// Find minimum and maximum key values

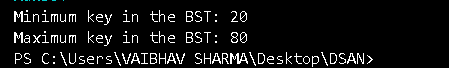
Node minNode = tree.findMin(tree.root);

Node maxNode = tree.findMax(tree.root);

System.out.println("Minimum key in the BST: " + minNode.data);

System.out.println("Maximum key in the BST: " + maxNode.data);

}

} 

32. Write a recursive program to check whether given tree is complete tree or not.

class Node {

int data;

Node left, right;

public Node(int item) {

data = item;

left = right = null;

}

}

class CompleteBinaryTree {

Node root;

// Function to count total number of nodes

int countNodes(Node node) {

if (node == null)

return 0;

return (1 + countNodes(node.left) + countNodes(node.right));

}

// Function to check if the tree is complete

boolean isComplete(Node node, int index, int numberNodes) {

if (node == null)

return true;

if (index >= numberNodes)

return false;

return (isComplete(node.left, 2 \* index + 1, numberNodes) &&

isComplete(node.right, 2 \* index + 2, numberNodes));

}

public static void main(String[] args) {

CompleteBinaryTree tree = new CompleteBinaryTree();

/\* Example tree:

1

/ \

2 3

/ \ /

4 5 6

\*/

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

tree.root.left.right = new Node(5);

tree.root.right.left = new Node(6);

int nodeCount = tree.countNodes(tree.root);

int index = 0;

if (tree.isComplete(tree.root, index, nodeCount))

System.out.println("The tree is a complete binary tree.");

else

System.out.println("The tree is NOT a complete binary tree.");

}

} 

33. Write a program to construct an AVL tree and perform postorder traversal.

class Node {

int key, height;

Node left, right;

Node(int d) {

key = d;

height = 1;

}

}

class AVLTree {

Node root;

// Get height of the node

int height(Node N) {

if (N == null)

return 0;

return N.height;

}

// Get balance factor of node

int getBalance(Node N) {

if (N == null)

return 0;

return height(N.left) - height(N.right);

}

// Right rotate subtree rooted with y

Node rightRotate(Node y) {

Node x = y.left;

Node T2 = x.right;

// Perform rotation

x.right = y;

y.left = T2;

// Update heights

y.height = Math.max(height(y.left), height(y.right)) + 1;

x.height = Math.max(height(x.left), height(x.right)) + 1;

// Return new root

return x;

}

// Left rotate subtree rooted with x

Node leftRotate(Node x) {

Node y = x.right;

Node T2 = y.left;

// Perform rotation

y.left = x;

x.right = T2;

// Update heights

x.height = Math.max(height(x.left), height(x.right)) + 1;

y.height = Math.max(height(y.left), height(y.right)) + 1;

// Return new root

return y;

}

// Insert a key into subtree rooted with node

Node insert(Node node, int key) {

// Normal BST insertion

if (node == null)

return (new Node(key));

if (key < node.key)

node.left = insert(node.left, key);

else if (key > node.key)

node.right = insert(node.right, key);

else // Duplicate keys not allowed

return node;

// Update height

node.height = 1 + Math.max(height(node.left), height(node.right));

// Get balance factor

int balance = getBalance(node);

// If unbalanced, there are 4 cases

// Left Left Case

if (balance > 1 && key < node.left.key)

return rightRotate(node);

// Right Right Case

if (balance < -1 && key > node.right.key)

return leftRotate(node);

// Left Right Case

if (balance > 1 && key > node.left.key) {

node.left = leftRotate(node.left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node.right.key) {

node.right = rightRotate(node.right);

return leftRotate(node);

}

// return unchanged node

return node;

}

// Postorder traversal

void postOrder(Node node) {

if (node != null) {

postOrder(node.left);

postOrder(node.right);

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

}

}

public static void main(String[] args) {

AVLTree tree = new AVLTree();

/\* Constructing AVL Tree

Example Insertion Order: 10, 20, 30, 40, 50, 25

\*/

tree.root = tree.insert(tree.root, 10);

tree.root = tree.insert(tree.root, 20);

tree.root = tree.insert(tree.root, 30);

tree.root = tree.insert(tree.root, 40);

tree.root = tree.insert(tree.root, 50);

tree.root = tree.insert(tree.root, 25);

System.out.println("Postorder traversal of AVL tree:");

tree.postOrder(tree.root);

}

} 

34. Write a program to find the minimum and maximum nodes in an AVL tree of given height.

import java.util.Scanner;

public class AVLTreeMinMaxNodes {

// Function to calculate minimum number of nodes

static int minNodes(int height) {

if (height == 0)

return 0;

if (height == 1)

return 1;

return 1 + minNodes(height - 1) + minNodes(height - 2);

}

// Function to calculate maximum number of nodes

static int maxNodes(int height) {

return (int) Math.pow(2, height + 1) - 1;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter height of AVL Tree: ");

int height = sc.nextInt();

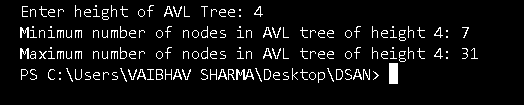
System.out.println("Minimum number of nodes in AVL tree of height " + height + ": " + minNodes(height));

System.out.println("Maximum number of nodes in AVL tree of height " + height + ": " + maxNodes(height));

sc.close();

}

}



35. Write a program to find the minimum number of nodes on a size-balanced tree.

import java.util.Scanner;

public class SizeBalancedTreeMinNodes {

// Function to find minimum number of nodes

static int minNodes(int height) {

if (height == 0)

return 0;

if (height == 1)

return 1;

return 1 + minNodes(height - 1) + minNodes(height - 2);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

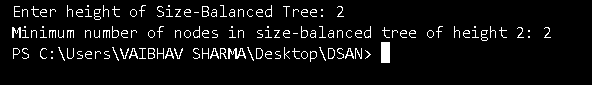
System.out.print("Enter height of Size-Balanced Tree: ");

int height = sc.nextInt();

System.out.println("Minimum number of nodes in size-balanced tree of height " + height + ": " + minNodes(height));

sc.close();

}

} 

36. write a program to implement a tree for a given infix expression.

import java.util.Stack;

// Node class

class Node {

String value;

Node left, right;

Node(String item) {

value = item;

left = right = null;

}

}

public class ExpressionTree {

// Function to check if a character is operator

boolean isOperator(String c) {

return c.equals("+") || c.equals("-") || c.equals("\*") || c.equals("/");

}

// Function to return precedence

int precedence(String op) {

switch (op) {

case "+": case "-":

return 1;

case "\*": case "/":

return 2;

}

return -1;

}

// Function to convert infix to postfix

String[] infixToPostfix(String[] infix) {

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

StringBuilder result = new StringBuilder();

for (String token : infix) {

if (!isOperator(token) && !token.equals("(") && !token.equals(")")) {

result.append(token).append(" ");

} else if (token.equals("(")) {

stack.push(token);

} else if (token.equals(")")) {

while (!stack.isEmpty() && !stack.peek().equals("(")) {

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

}

stack.pop();

} else {

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

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

}

stack.push(token);

}

}

while (!stack.isEmpty()) {

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

}

return result.toString().trim().split("\\s+");

}

// Function to build Expression Tree from postfix

Node buildTree(String[] postfix) {

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

for (String token : postfix) {

Node t = new Node(token);

if (isOperator(token)) {

t.right = stack.pop();

t.left = stack.pop();

}

stack.push(t);

}

return stack.peek();

}

// Inorder traversal (for testing)

void inorder(Node node) {

if (node != null) {

if (isOperator(node.value)) System.out.print("(");

inorder(node.left);

System.out.print(node.value);

inorder(node.right);

if (isOperator(node.value)) System.out.print(")");

}

}

public static void main(String[] args) {

ExpressionTree et = new ExpressionTree();

// Example: (a+b)\*(c-d)

String infixExpr = "( a + b ) \* ( c - d )";

String[] infix = infixExpr.split("\\s+");

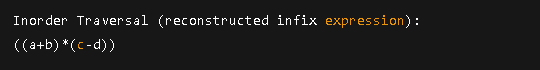
String[] postfix = et.infixToPostfix(infix);

Node root = et.buildTree(postfix);

System.out.println("Inorder Traversal (reconstructed infix expression):");

et.inorder(root);

}

} 

37. Write a program to draw a tree for a given nested tree representation expression.

import java.util.Stack;

// Node class

class TreeNode {

char data;

TreeNode left, right;

public TreeNode(char data) {

this.data = data;

this.left = this.right = null;

}

}

public class NestedTree {

// Function to construct tree from nested expression

public static TreeNode constructTree(String expr) {

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

TreeNode root = null, temp = null;

int k = 0; // 1 for left child, 2 for right child

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

char ch = expr.charAt(i);

if (ch == '(') {

stack.push(temp);

k = 1;

} else if (ch == ',') {

k = 2;

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

stack.pop();

} else if (Character.isLetterOrDigit(ch)) {

temp = new TreeNode(ch);

if (root == null) {

root = temp;

} else {

if (k == 1) {

stack.peek().left = temp;

} else if (k == 2) {

stack.peek().right = temp;

}

}

}

}

return root;

}

// Preorder traversal to print tree

public static void preorder(TreeNode node) {

if (node != null) {

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

preorder(node.left);

preorder(node.right);

}

}

public static void main(String[] args) {

String expr = "A(B(D,E),C(F))"; // Example nested tree expression

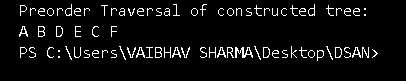
TreeNode root = constructTree(expr);

System.out.println("Preorder Traversal of constructed tree:");

preorder(root);

}

}



38. Write a program to find the left child of kth element from the given array representation tree.

public class ArrayTreeLeftChild {

public static void main(String[] args) {

int[] tree = {0, 12, 56, 3, 7, 9, 35, 11, 19, 25, 75}; // Index 0 is unused

int k = 3; // Find left child of kth element

if (2 \* k < tree.length) {

System.out.println("Left child of element at index " + k + " (" + tree[k] + "): " + tree[2 \* k]);

} else {

System.out.println("No left child for element at index " + k);

}

}

} 

39. Write a program to find the left child of kth element from the given leftmost child right sibling representation tree.

import java.util.\*;

class LCRSNode {

int data;

LCRSNode leftChild;

LCRSNode rightSibling;

public LCRSNode(int data) {

this.data = data;

this.leftChild = this.rightSibling = null;

}

}

public class LCRSTree {

// Store all nodes in list by level order to access kth easily

static List<LCRSNode> nodeList = new ArrayList<>();

// Utility to create sample LCRS tree from given array

public static LCRSNode createSampleLCRSTree(int[] values) {

// Let's manually create the tree structure

// Tree: 12(56(3,7,9),35(11,19),25,75)

LCRSNode root = new LCRSNode(values[0]);

nodeList.add(root); // 0 - 12

// Level 1

LCRSNode n56 = new LCRSNode(56);

nodeList.add(n56); // 1

LCRSNode n25 = new LCRSNode(25);

nodeList.add(n25); // 2

LCRSNode n75 = new LCRSNode(75);

nodeList.add(n75); // 3

root.leftChild = n56;

n56.rightSibling = n25;

n25.rightSibling = n75;

// Level 2

LCRSNode n3 = new LCRSNode(3);

nodeList.add(n3); // 4

LCRSNode n7 = new LCRSNode(7);

nodeList.add(n7); // 5

LCRSNode n9 = new LCRSNode(9);

nodeList.add(n9); // 6

n56.leftChild = n3;

n3.rightSibling = n7;

n7.rightSibling = n9;

LCRSNode n35 = new LCRSNode(35);

nodeList.add(n35); // 7

n56.rightSibling = n35;

LCRSNode n11 = new LCRSNode(11);

nodeList.add(n11); // 8

LCRSNode n19 = new LCRSNode(19);

nodeList.add(n19); // 9

n35.leftChild = n11;

n11.rightSibling = n19;

return root;

}

public static void main(String[] args) {

int[] values = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75}; // Input data

LCRSNode root = createSampleLCRSTree(values);

int k = 1; // Index in nodeList (0-based)

if (k >= 0 && k < nodeList.size()) {

LCRSNode kth = nodeList.get(k);

if (kth.leftChild != null) {

System.out.println("Left child of node " + kth.data + " is: " + kth.leftChild.data);

} else {

System.out.println("Node " + kth.data + " has no left child.");

}

} else {

System.out.println("Invalid k value.");

}

}

} 

40. Write a program for breadth first traversal on graph.

import java.util.\*;

public class GraphBFS {

private int vertices; // Number of vertices

private LinkedList<Integer>[] adjList; // Adjacency List

// Constructor

public GraphBFS(int v) {

vertices = v;

adjList = new LinkedList[v];

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

adjList[i] = new LinkedList<>();

}

}

// Add edge

public void addEdge(int u, int v) {

adjList[u].add(v);

adjList[v].add(u); // For undirected graph

}

// BFS traversal

public void BFS(int start) {

boolean[] visited = new boolean[vertices];

Queue<Integer> queue = new LinkedList<>();

visited[start] = true;

queue.add(start);

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

while (!queue.isEmpty()) {

int node = queue.poll();

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

for (int neighbor : adjList[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

queue.add(neighbor);

}

}

}

}

public static void main(String[] args) {

GraphBFS g = new GraphBFS(6); // Example with 6 vertices (0 to 5)

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.addEdge(4, 5);

g.BFS(0); // Start BFS from vertex 0

}

} 

41. Write a program for depth first traversal on graph.

import java.util.\*;

public class GraphDFS {

private int vertices; // Number of vertices

private LinkedList<Integer>[] adjList;

// Constructor

public GraphDFS(int v) {

vertices = v;

adjList = new LinkedList[v];

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

adjList[i] = new LinkedList<>();

}

}

// Add edge

public void addEdge(int u, int v) {

adjList[u].add(v);

adjList[v].add(u); // For undirected graph

}

// DFS Utility (Recursive)

private void DFSUtil(int node, boolean[] visited) {

visited[node] = true;

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

for (int neighbor : adjList[node]) {

if (!visited[neighbor]) {

DFSUtil(neighbor, visited);

}

}

}

// DFS Traversal

public void DFS(int start) {

boolean[] visited = new boolean[vertices];

System.out.println("DFS traversal starting from vertex " + start + ":");

DFSUtil(start, visited);

}

public static void main(String[] args) {

GraphDFS g = new GraphDFS(6); // Example with 6 vertices (0 to 5)

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

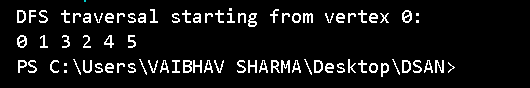
g.addEdge(2, 4);

g.addEdge(4, 5);

g.DFS(0); // Start DFS from vertex 0

}

}



42. Write a program to check whether there is a cycle in a given directed graph or not.

Use this data for all following programs:

12, 56, 3, 7, 9, 35, 11, 19, 25, 75

import java.util.\*;

public class DirectedGraphCycle {

private int vertices;

private LinkedList<Integer>[] adjList;

public DirectedGraphCycle(int v) {

vertices = v;

adjList = new LinkedList[v];

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

adjList[i] = new LinkedList<>();

}

public void addEdge(int u, int v) {

adjList[u].add(v); // directed edge

}

// Utility method to check for cycle

private boolean isCyclicUtil(int node, boolean[] visited, boolean[] recStack) {

if (recStack[node]) return true; // Found a back edge -> cycle

if (visited[node]) return false;

visited[node] = true;

recStack[node] = true;

for (int neighbor : adjList[node]) {

if (isCyclicUtil(neighbor, visited, recStack))

return true;

}

recStack[node] = false;

return false;

}

public boolean isCyclic() {

boolean[] visited = new boolean[vertices];

boolean[] recStack = new boolean[vertices];

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

if (!visited[i]) {

if (isCyclicUtil(i, visited, recStack))

return true;

}

}

return false;

}

public static void main(String[] args) {

DirectedGraphCycle graph = new DirectedGraphCycle(4);

graph.addEdge(0, 1);

graph.addEdge(1, 2);

graph.addEdge(2, 3);

graph.addEdge(3, 1); // Cycle from 3 → 1 → 2 → 3

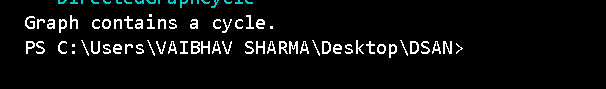
if (graph.isCyclic())

System.out.println("Graph contains a cycle.");

else

System.out.println("Graph does not contain a cycle.");

}

} 

43. Write a program to sort given elements using insertion sort method.

public class InsertionSort {

public static void insertionSort(int[] arr) {

int n = arr.length;

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1], that are greater than key

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

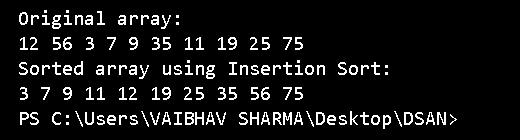
printArray(arr);

insertionSort(arr);

System.out.println("Sorted array using Insertion Sort:");

printArray(arr);

}

} 

44. Write a program to sort given elements using bubble sort method.

public class BubbleSort {

public static void bubbleSort(int[] arr) {

int n = arr.length;

boolean swapped;

// Outer loop for all passes

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

swapped = false;

// Last i elements are already in place

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

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

// Swap arr[j] and arr[j+1]

int temp = arr[j];

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

arr[j + 1] = temp;

swapped = true;

}

}

// If no two elements were swapped in inner loop, array is sorted

if (!swapped)

break;

}

}

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

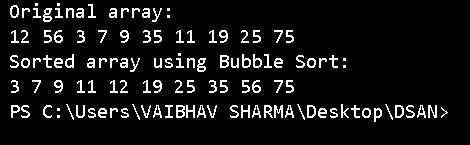
printArray(arr);

bubbleSort(arr);

System.out.println("Sorted array using Bubble Sort:");

printArray(arr);

}

} 

45. Write a program to sort given elements using bucket sort method.

import java.util.\*;

public class BucketSort {

public static void bucketSort(int[] arr, int numberOfBuckets) {

if (arr.length == 0)

return;

// 1. Find maximum and minimum values

int maxValue = arr[0];

int minValue = arr[0];

for (int num : arr) {

if (num > maxValue) maxValue = num;

if (num < minValue) minValue = num;

}

// 2. Initialize buckets

List<Integer>[] buckets = new ArrayList[numberOfBuckets];

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

buckets[i] = new ArrayList<>();

}

// 3. Distribute input array values into buckets

for (int num : arr) {

int bucketIndex = (num - minValue) \* (numberOfBuckets - 1) / (maxValue - minValue);

buckets[bucketIndex].add(num);

}

// 4. Sort individual buckets

for (List<Integer> bucket : buckets) {

Collections.sort(bucket);

}

// 5. Merge all buckets into original array

int index = 0;

for (List<Integer> bucket : buckets) {

for (int num : bucket) {

arr[index++] = num;

}

}

}

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

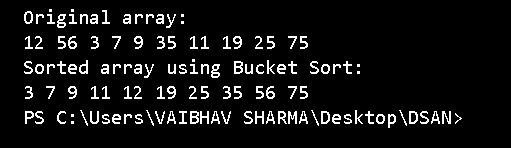
printArray(arr);

bucketSort(arr, 5); // You can change the number of buckets

System.out.println("Sorted array using Bucket Sort:");

printArray(arr);

}

} 

46. Write a program to sort given elements using merge sort method.

public class MergeSort {

// Method to merge two subarrays

public static void merge(int[] arr, int left, int middle, int right) {

// Sizes of two subarrays

int n1 = middle - left + 1;

int n2 = right - middle;

// Temporary arrays

int[] L = new int[n1];

int[] R = new int[n2];

// Copy data to temp arrays

for (int i = 0; i < n1; ++i)

L[i] = arr[left + i];

for (int j = 0; j < n2; ++j)

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

// Merge the temp arrays back into arr

int i = 0, j = 0;

int k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy remaining elements of L[]

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy remaining elements of R[]

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Method to sort array using Merge Sort

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

// Find the middle point

int middle = (left + right) / 2;

// Sort first and second halves

mergeSort(arr, left, middle);

mergeSort(arr, middle + 1, right);

// Merge the sorted halves

merge(arr, left, middle, right);

}

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

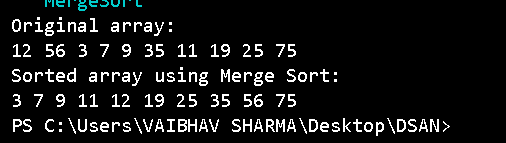
printArray(arr);

mergeSort(arr, 0, arr.length - 1);

System.out.println("Sorted array using Merge Sort:");

printArray(arr);

}

} 

47. Write a program to sort given elements using quick sort method.

public class QuickSort {

// Method to partition the array

public static int partition(int[] arr, int low, int high) {

int pivot = arr[high]; // Last element as pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j < high; j++) {

// If current element is smaller than or equal to the pivot

if (arr[j] <= pivot) {

i++;

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap arr[i+1] and arr[high] (pivot)

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

// Method to sort the array using Quick Sort

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

// Get the partition index

int pi = partition(arr, low, high);

// Recursively sort the two halves

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

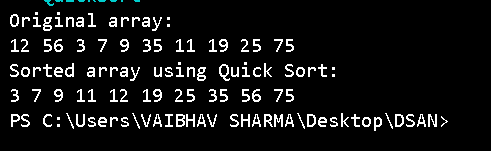
printArray(arr);

quickSort(arr, 0, arr.length - 1);

System.out.println("Sorted array using Quick Sort:");

printArray(arr);

}

} 

48. Write a program to sort given elements using heap sort method.

public class HeapSort {

// Method to heapify a subtree rooted at index i

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

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If left child is larger than root

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

largest = left;

}

// If right child is larger than root

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

largest = right;

}

// If largest is not root

if (largest != i) {

// Swap root with largest

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Method to perform heap sort

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);

}

// One by one 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 heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

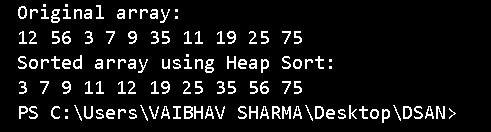
printArray(arr);

heapSort(arr);

System.out.println("Sorted array using Heap Sort:");

printArray(arr);

}

} 

49. Write a program to sort given elements using insertion sort method.

public class InsertionSort {

// Method to perform insertion sort

public static void insertionSort(int[] arr) {

int n = arr.length;

// Traverse through 1 to n

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key

// to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] arr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

System.out.println("Original array:");

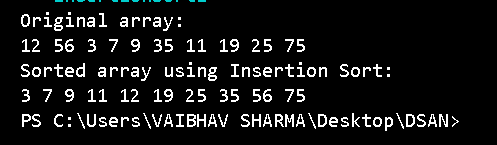
printArray(arr);

insertionSort(arr);

System.out.println("Sorted array using Insertion Sort:");

printArray(arr);

}

} 

50. Write a program to construct min heap and max heap.

public class HeapConstruction {

// Method to heapify a subtree rooted at index i for Min Heap

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

int smallest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Check if left child is smaller than root

if (left < n && arr[left] < arr[smallest]) {

smallest = left;

}

// Check if right child is smaller than root

if (right < n && arr[right] < arr[smallest]) {

smallest = right;

}

// If smallest is not root, swap and heapify recursively

if (smallest != i) {

int temp = arr[i];

arr[i] = arr[smallest];

arr[smallest] = temp;

heapifyMin(arr, n, smallest);

}

}

// Method to build Min Heap

public static void buildMinHeap(int[] arr) {

int n = arr.length;

// Start heapifying from the last non-leaf node

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

heapifyMin(arr, n, i);

}

}

// Method to heapify a subtree rooted at index i for Max Heap

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Check if left child is larger than root

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

largest = left;

}

// Check if right child is larger than root

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

largest = right;

}

// If largest is not root, swap and heapify recursively

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapifyMax(arr, n, largest);

}

}

// Method to build Max Heap

public static void buildMaxHeap(int[] arr) {

int n = arr.length;

// Start heapifying from the last non-leaf node

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

heapifyMax(arr, n, i);

}

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] minHeapArr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

int[] maxHeapArr = {12, 56, 3, 7, 9, 35, 11, 19, 25, 75};

// Building and displaying Min Heap

System.out.println("Min Heap:");

buildMinHeap(minHeapArr);

printArray(minHeapArr);

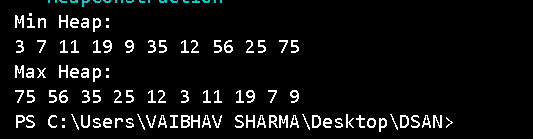
// Building and displaying Max Heap

System.out.println("Max Heap:");

buildMaxHeap(maxHeapArr);

printArray(maxHeapArr);

}

} 

51. Write a program to find the number of leaf and non-leaf nodes of a max heap.

public class MaxHeapNodeCount {

// Method to count leaf nodes in a Max Heap

public static int countLeafNodes(int[] arr, int n) {

int count = 0;

// Leaf nodes are at indices from n/2 to n-1

for (int i = n / 2; i < n; i++) {

count++;

}

return count;

}

// Method to count non-leaf nodes in a Max Heap

public static int countNonLeafNodes(int[] arr, int n) {

int count = 0;

// Non-leaf nodes are from index 0 to n/2 - 1

for (int i = 0; i < n / 2; i++) {

count++;

}

return count;

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] maxHeapArr = {75, 56, 35, 19, 25, 12, 11, 7, 3, 9};

int n = maxHeapArr.length;

System.out.println("Max Heap:");

printArray(maxHeapArr);

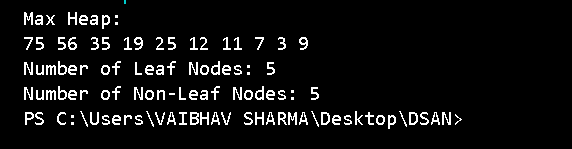
int leafNodes = countLeafNodes(maxHeapArr, n);

int nonLeafNodes = countNonLeafNodes(maxHeapArr, n);

System.out.println("Number of Leaf Nodes: " + leafNodes);

System.out.println("Number of Non-Leaf Nodes: " + nonLeafNodes);

}

} 

52. Write a program to delete maximum value from a max heap and then reheapify.

public class MaxHeapDeletion {

// Method to heapify the Max Heap

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Check if left child is larger than root

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

largest = left;

}

// Check if right child is larger than root

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

largest = right;

}

// If largest is not root, swap and heapify recursively

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapifyMax(arr, n, largest);

}

}

// Method to delete the root (maximum) from Max Heap

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

// Replace the root with the last element

arr[0] = arr[n - 1];

// Decrease the heap size by 1

n = n - 1;

// Call heapify to restore the heap property

heapifyMax(arr, n, 0);

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] maxHeapArr = {75, 56, 35, 19, 25, 12, 11, 7, 3, 9};

int n = maxHeapArr.length;

System.out.println("Original Max Heap:");

printArray(maxHeapArr);

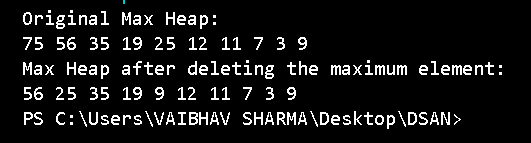
// Delete the maximum element (root) and reheapify

deleteMax(maxHeapArr, n);

System.out.println("Max Heap after deleting the maximum element:");

printArray(maxHeapArr);

}

} 

53. Write a program to insert 20 in max heap.

public class MaxHeapInsertion {

// Method to heapify the Max Heap

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Check if left child is larger than root

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

largest = left;

}

// Check if right child is larger than root

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

largest = right;

}

// If largest is not root, swap and heapify recursively

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapifyMax(arr, n, largest);

}

}

// Method to insert a new element into Max Heap

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

// First insert the value at the end

arr[n] = value;

int i = n;

// Move the inserted element up to restore the heap property

while (i > 0 && arr[(i - 1) / 2] < arr[i]) {

int temp = arr[i];

arr[i] = arr[(i - 1) / 2];

arr[(i - 1) / 2] = temp;

i = (i - 1) / 2;

}

}

// Method to print array

public static void printArray(int[] arr) {

for (int num : arr)

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

System.out.println();

}

public static void main(String[] args) {

int[] maxHeapArr = {75, 56, 35, 19, 25, 12, 11, 7, 3, 9};

int n = maxHeapArr.length;

System.out.println("Original Max Heap:");

printArray(maxHeapArr);

// Insert 20 into Max Heap

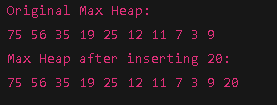
insertMaxHeap(maxHeapArr, n, 20);

n++;

System.out.println("Max Heap after inserting 20:");

printArray(maxHeapArr);

}

} 

54. Insert following keys 5, 28, 19, 15, 20, 33, 12, 17 and 10 in hash table using chaining hashing method and find minimum, maximum and average chain length in hash table.

import java.util.LinkedList;

public class HashTableWithChaining {

// Hash table size

static final int SIZE = 10;

// Create an array of LinkedLists to store elements at each index

LinkedList<Integer>[] table = new LinkedList[SIZE];

// Constructor to initialize the table

public HashTableWithChaining() {

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

table[i] = new LinkedList<>();

}

}

// Hash function to map values to a specific index

public int hashFunction(int key) {

return key % SIZE;

}

// Method to insert key into hash table

public void insert(int key) {

int index = hashFunction(key);

table[index].add(key); // Add key to the list at the hashed index

}

// Method to calculate the minimum, maximum, and average chain length

public void calculateChainLengths() {

int minLength = Integer.MAX\_VALUE, maxLength = Integer.MIN\_VALUE, totalChains = 0, totalElements = 0;

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

int chainLength = table[i].size();

if (chainLength > 0) {

totalChains++;

totalElements += chainLength;

minLength = Math.min(minLength, chainLength);

maxLength = Math.max(maxLength, chainLength);

}

}

// Calculate average chain length

double avgLength = totalChains > 0 ? (double) totalElements / totalChains : 0;

System.out.println("Minimum Chain Length: " + minLength);

System.out.println("Maximum Chain Length: " + maxLength);

System.out.println("Average Chain Length: " + avgLength);

}

// Method to display the hash table

public void displayHashTable() {

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

System.out.print("Index " + i + ": ");

for (Integer key : table[i]) {

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

}

System.out.println();

}

}

public static void main(String[] args) {

HashTableWithChaining hashTable = new HashTableWithChaining();

// Insert the keys into the hash table

int[] keys = {5, 28, 19, 15, 20, 33, 12, 17, 10};

for (int key : keys) {

hashTable.insert(key);

}

// Display the hash table

System.out.println("Hash Table:");

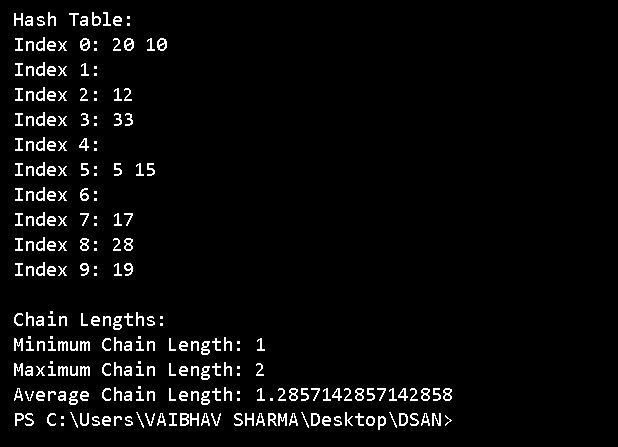
hashTable.displayHashTable();

// Calculate and display the chain lengths

System.out.println("\nChain Lengths:");

hashTable.calculateChainLengths();

}

} 

55. Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using Double hashing, where h(x)= x mod 10, h2(x) = x mod 6 + 1.

public class HashTableWithDoubleHashing {

static final int SIZE = 10;

int[] table = new int[SIZE];

// Constructor to initialize the hash table

public HashTableWithDoubleHashing() {

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

table[i] = -1; // -1 denotes an empty slot

}

}

// First hash function: h1(x) = x % SIZE

public int hashFunction1(int key) {

return key % SIZE;

}

// Second hash function: h2(x) = 1 + (x % (SIZE - 1))

public int hashFunction2(int key) {

return 1 + (key % (SIZE - 1));

}

// Method to insert key into hash table using double hashing

public void insert(int key) {

int index = hashFunction1(key); // Primary hash function

int stepSize = hashFunction2(key); // Step size from secondary hash function

// Resolve collision using double hashing

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

index = (index + stepSize) % SIZE;

}

table[index] = key;

}

// Method to display the hash table

public void displayHashTable() {

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

System.out.print("Index " + i + ": ");

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

System.out.print(table[i]);

} else {

System.out.print("Empty");

}

System.out.println();

}

}

public static void main(String[] args) {

HashTableWithDoubleHashing hashTable = new HashTableWithDoubleHashing();

// Elements to insert into the hash table

int[] keys = {17, 16, 22, 36, 33, 46, 26, 144};

// Insert the elements into the hash table

for (int key : keys) {

hashTable.insert(key);

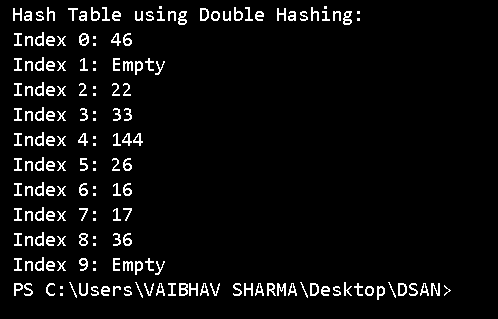
}

// Display the hash table

System.out.println("Hash Table using Double Hashing:");

hashTable.displayHashTable();

}

} 

56. Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using linear probing, where h(x)= x mod 10.

public class HashTableWithLinearProbing {

static final int SIZE = 10;

int[] table = new int[SIZE];

// Constructor to initialize the hash table

public HashTableWithLinearProbing() {

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

table[i] = -1; // -1 denotes an empty slot

}

}

// Hash function: h(x) = x % SIZE

public int hashFunction(int key) {

return key % SIZE;

}

// Method to insert key into hash table using linear probing

public void insert(int key) {

int index = hashFunction(key); // Primary hash function

// Resolve collision using linear probing

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

index = (index + 1) % SIZE; // Move to the next index

}

table[index] = key;

}

// Method to display the hash table

public void displayHashTable() {

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

System.out.print("Index " + i + ": ");

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

System.out.print(table[i]);

} else {

System.out.print("Empty");

}

System.out.println();

}

}

public static void main(String[] args) {

HashTableWithLinearProbing hashTable = new HashTableWithLinearProbing();

// Elements to insert into the hash table

int[] keys = {17, 16, 22, 36, 33, 46, 26, 144};

// Insert the elements into the hash table

for (int key : keys) {

hashTable.insert(key);

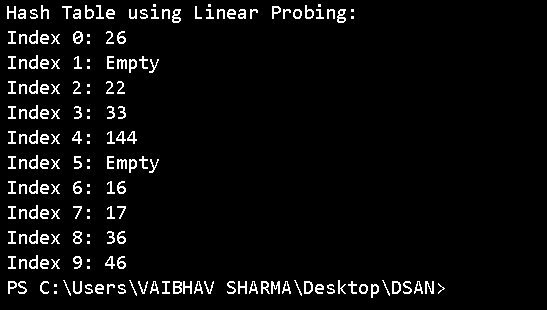
}

// Display the hash table

System.out.println("Hash Table using Linear Probing:");

hashTable.displayHashTable();

}

} 

57. Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using quadratic probing, where h(x)= x mod 10.

public class HashTableWithQuadraticProbing {

static final int SIZE = 10;

int[] table = new int[SIZE];

// Constructor to initialize the hash table

public HashTableWithQuadraticProbing() {

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

table[i] = -1; // -1 denotes an empty slot

}

}

// Hash function: h(x) = x % SIZE

public int hashFunction(int key) {

return key % SIZE;

}

// Method to insert key into hash table using quadratic probing

public void insert(int key) {

int index = hashFunction(key); // Primary hash function

int i = 1;

// Resolve collision using quadratic probing: index + i^2

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

index = (index + i \* i) % SIZE; // Quadratic probing formula

i++;

}

table[index] = key;

}

// Method to display the hash table

public void displayHashTable() {

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

System.out.print("Index " + i + ": ");

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

System.out.print(table[i]);

} else {

System.out.print("Empty");

}

System.out.println();

}

}

public static void main(String[] args) {

HashTableWithQuadraticProbing hashTable = new HashTableWithQuadraticProbing();

// Elements to insert into the hash table

int[] keys = {17, 16, 22, 36, 33, 46, 26, 144};

// Insert the elements into the hash table

for (int key : keys) {

hashTable.insert(key);

}

// Display the hash table

System.out.println("Hash Table using Quadratic Probing:");

hashTable.displayHashTable();

}

}

