**Advance Data Structure & Algorithm Course Code: R1UC503B**

Lab File

For

BACHELOR OF

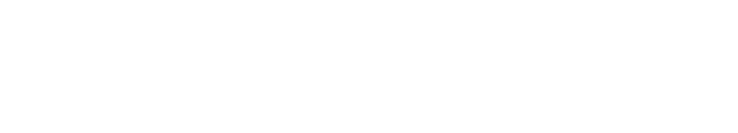
ENGINEERING & TECHNOLOGY



**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**GALGOTIAS UNIVERSITY, GREATER NOIDA**

**UTTAR PRADESH**



**Student**

**Name**

**:**

**Shiva Dixit**

**Admission**

**No:**

**23SCSE1011083**



**Semester**



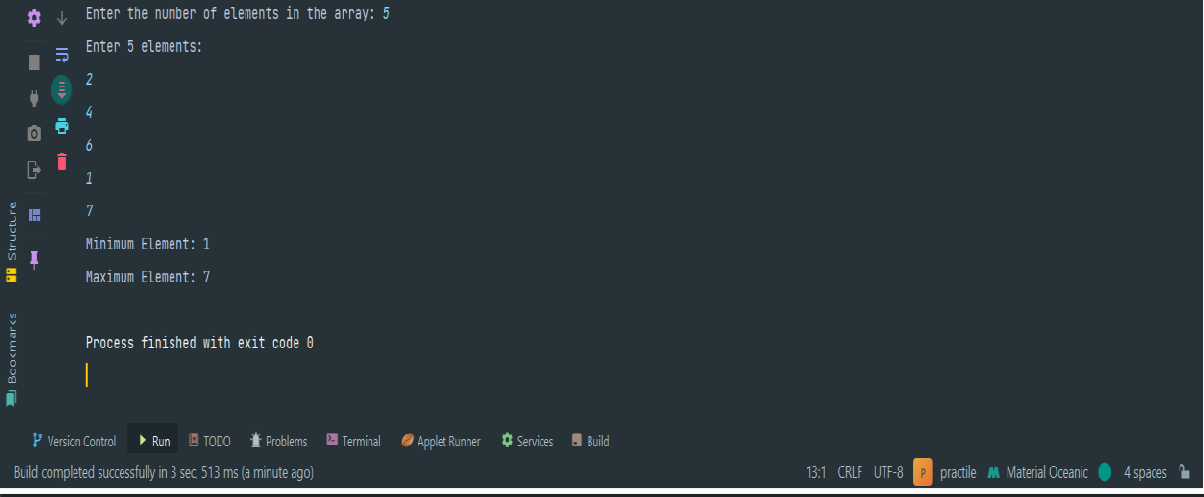
**:**

**V**

**STUDENT NAME: Shiva Dixit ADMISSION NO. : 23SCSE1011083**

**DATE: 10/01/2025 EXPERIMENT NO. : 1 to 49**

|  |
| --- |
| **Lab Experiment 1:**  **Find the Maximum and Minimum Elements in an Array: Write a function to find the maximum and minimum elements in an array.**  **Code:**  *import* java.util.Scanner; *public class* Question\_1 { *public static void* main(String[] args) {  Scanner = *new* Scanner(System.*in*);  System.*out*.print("Enter the number of elements in the array: "); *int* n = scanner.nextInt(); *int*[] array = *new int*[n];  System.*out*.println("Enter " + n + " elements:"); *for* (*int* i = 0; i < n; i++) { array[i] = scanner.nextInt();  }  *int* min = array[0]; *int* max = array[0]; *for* (*int* num : array) { *if* (num < min) { min = num;  }  *if* (num > max) { max = num;  }  }  System.*out*.println("Minimum Element: " + min); System.*out*.println("Maximum Element: " + max); scanner.close();  }  }  **Output:** |



**Lab**

**Experiment**

**2:**

**Reverse**

**an**

**Array:**

**Write**

**a**

**function**

**to**

**reverse**

**an**

**array**

**in**

**place.**

**Code:**

*import*

java.util.Arrays;

*public*

*class*

Question\_2

{

*public*

*static*

*void*

main(String[]

args)

{

*int*

[]

array

=

{1

,

2

,

3

,

4

,

5

,

;

6}

System.

*out*

.println("Original

Array:

"

Arrays.

+

*toString*

array));

(

*reverseArray*

array);

(

System.

*out*

.println("Reversed

Array:

"

+

Arrays.

*toString*

(

array));

}

*public*

*static void*

reverseArray(

*int*

[]

array)

{

*int*

start

=

0

;

*int*

end

=

array.length

-

1

;

*while*

(

start

<

end)

{

*int*

temp = array[start];

array[start]

=

array[end];

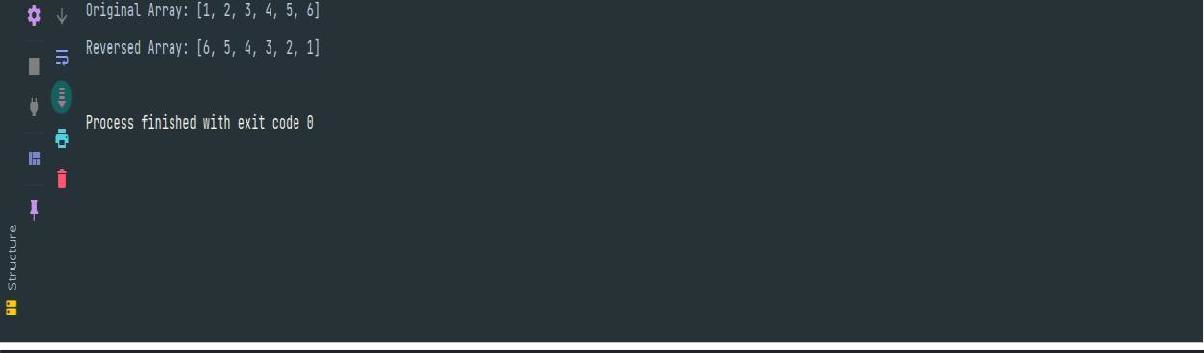
array[end] = temp;

start++;

end

--

;



}

}

}

**Output:**

**Lab**

**Experiment**

**3:**

**Find**

**the**

**Kth**

**Smallest/Largest**

**Element**

**in**

**an**

**Array:**

**Write**

**a**

**function**

**to find the Kth smallest or largest element in an array.**

**Code:**

*import*

java.util.Arrays;

*import*

java.util.Scanner;

*public*

*class*

KthElement

{

*public static void*

main(String[] args) {

Scanner

scanner

=

*new*

Scanner(System.

*in*

;

)

System.

*out*

.print("Enter

the

number

of

elements

in

the

array:

");

*int*

n

=

scanner.nextInt();

*int*

array =

[]

*new int*

n];

[

System.

*out*

.println("Enter

"

+

n

+

"

elements:");

*for*

(

*int*

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

array[i]

=

scanner.nextInt();

}

System.

*out*

.print("Enter

the

value

of

K:

");



*int*

k

=

scanner.nextInt();

*if*

(

k

<

1

||

k

>

n)

{

System.

*out*

.println("Invalid

value

of

K.

It

should

be

between

1

and " + n);

}

*else*

{

*int*

kthSmallest

=

*findKthSmallest*

(

array,

k);

*int*

kthLargest =

*findKthLargest*

(

array, k);

System.

*out*

.println("Kth

Smallest

Element:

"

+

kthSmallest);

System.

*out*

.println("Kth Largest Element: " + kthLargest);

}

scanner.close();

}

*public*

*static*

*int*

findKthSmallest(

*int*

[]

array,

*int*

k)

{

Arrays.

*sort*

(

array);

*return*

array[k

-

1]

;

}

*public*

*static*

*int*

findKthLargest(

*int*

[]

array,

*int*

k)

{

Arrays.

*sort*

(

array);

*return*

array[array.length

-

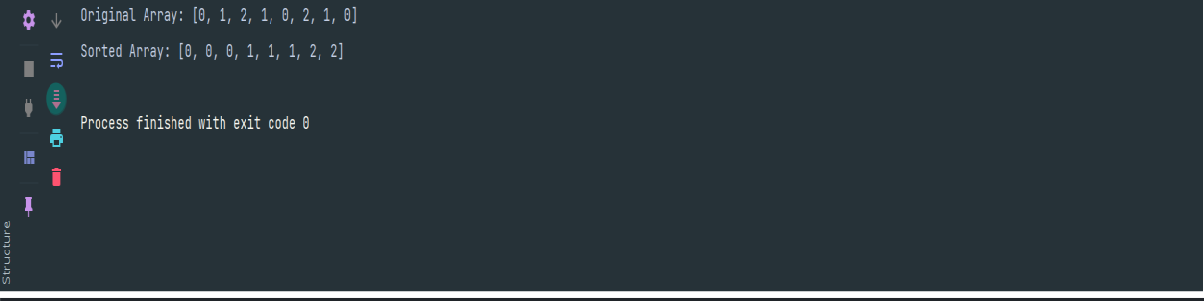
k];

}

}

**output:**

|  |
| --- |
| **Lab Experiment 4:**  **Sort an Array of 0s, 1s, and 2s: Given an array containing only 0s, 1s, and 2s, sort the array in linear time.**  **Code:**  *import* java.util.Arrays; *public class* Sort012Array { *public static void* main(String[] args) { *int*[] array = {0, 1, 2, 1, 0, 2, 1, 0};  System.*out*.println("Original Array: " + Arrays.*toString*(array)); *sortArray*(array);  System.*out*.println("Sorted Array: " + Arrays.*toString*(array));  }  *public static void* sortArray(*int*[] array) { *int* low = 0, mid = 0, high = array.length - 1; *while* (mid <= high) { *if* (array[mid] == 0) { *swap*(array, low, mid); low++;  mid++;  } *else if* (array[mid] == 1) { mid++;  } *else if* (array[mid] == 2) { *swap*(array, mid, high);  high--;  }  }  }  *private static void* swap(*int*[] array, *int* i, *int* j) { *int* temp = array[i]; array[i] = array[j]; |



array[j]

=

temp;

}

}

**Output:**

**Lab**

**Experiment**

**5:**

**Move**

**All**

**Zeroes**

**to**

**End**

**of**

**Array:**

**Write**

**a**

**function**

**to**

**move**

**all**

**zeroes**

**in an array to the end while maintaining the relative order of other**

**elements.**

**Code:**

*import*

java.util.Arrays;

*public*

*class*

MoveZeroesToEnd

{

*public*

*static*

*void*

main(String[]

args)

{

*int*

[]

array

=

,

{0

1

,

0

,

3

,

12}

;

System.

*out*

.println("Original

Array:

"

+

Arrays.

*toString*

(

array));

*moveZeroes*

(

array);

System.

*out*

.println("Array

after

moving

zeroes:

"

+

Arrays.

*toString*

(

array));

}

*public static*

*void*

moveZeroes(

*int*

[]

array)

{

*int*

index

=

0

;

*for*

(

*int*

num

:

array)

{

*if*

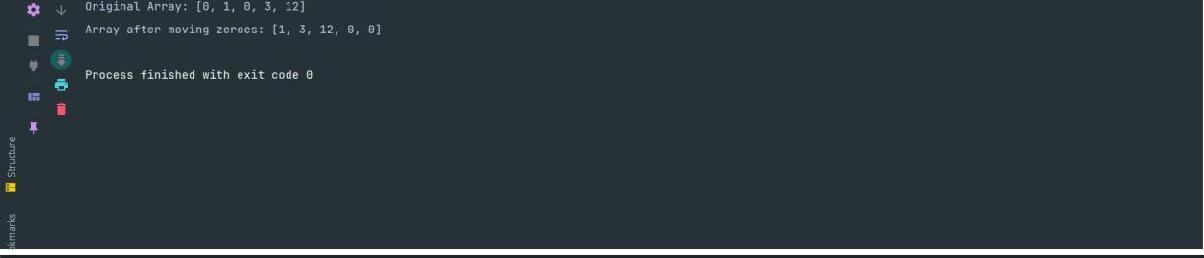
(

num != 0) {

array[index++]

=

num;



}

}

*while*

(

index

<

array.length)

{

array[index++] = 0;

}

}

}

**Output:**

**Lab**

**Experiment**

**6:**

**Reverse**

**a**

**Linked**

**List:**

**Write**

**a**

**function**

**to**

**reverse**

**a**

**singly**

**linked**

**list.**

**Code:**

*import*

java.util.LinkedList;

*import*

java.util.

*ListIterator*

;

*public*

*class*

ReverseLinkedList

{

*public static void*

main(String[] args) {

LinkedList<Integer>

list

=

*new*

LinkedList<>();

list.add(1);

list.add(2);

list.add(3);

list.add(4);

list.add(5);

System.

*out*

.println("Original Linked List: " + list);

*reverseLinkedList*

list);

(

System.

*out*

.println("Reversed

Linked

List:

"

+

list);

|  |
| --- |
| }  *public static void* reverseLinkedList(LinkedList<Integer> list) {  *ListIterator*<Integer> iterator = list.listIterator(); LinkedList<Integer> reversedList = *new* LinkedList<>(); *while* (iterator.hasNext()) {  Integer element = iterator.next();  reversedList.addFirst(element);  }  list.clear();  list.addAll(reversedList);  }  }  **Output:**    **Lab Experiment 7:**  **Detect a Cycle in a Linked List: Write a function to detect if a cycle exists in a linked list.**  **Code:**  *class* Node { *int* data;  Node next;  Node(*int* data) { *this*.data = data;  *this*.next = *null*;  }  } |

|  |
| --- |
| *public class* DetectCycleInLinkedList { *public static void* main(String[] args) { Node head = *new* Node(1); head.next = *new* Node(2); head.next.next = *new* Node(3); head.next.next.next = *new* Node(4); head.next.next.next.next = *new* Node(5); head.next.next.next.next.next = head.next; *if* (*hasCycle*(head)) {  System.*out*.println("The linked list has a cycle.");  } *else* {  System.*out*.println("The linked list has no cycle.");  }  }  *public static boolean* hasCycle(Node head) { *if* (head == *null* || head.next == *null*) { *return false*;  }  Node slow = head; Node fast = head;  *while* (fast != *null* && fast.next != *null*) { slow = slow.next; fast = fast.next.next;  *if* (slow == fast) { *return true*;  }  }  *return false*;  }  } |

|  |
| --- |
| **Output:**    **Lab Experiment 8:**  **Find the Middle of a Linked List: Write a function to find the middle element of a linked list.**  **Code:**  *class* Node { *int* data;  Node next;  Node(*int* data) { *this*.data = data;  *this*.next = *null*;  }  }  *public class* FindMiddleOfLinkedList { *public static void* main(String[] args) { Node head = *new* Node(1); head.next = *new* Node(2); head.next.next = *new* Node(3); head.next.next.next = *new* Node(4); head.next.next.next.next = *new* Node(5);  Node middle = *findMiddle*(head);  *if* (middle != *null*) {  System.*out*.println("The middle element is: " + middle.data);  } *else* {  System.*out*.println("The list is empty."); |

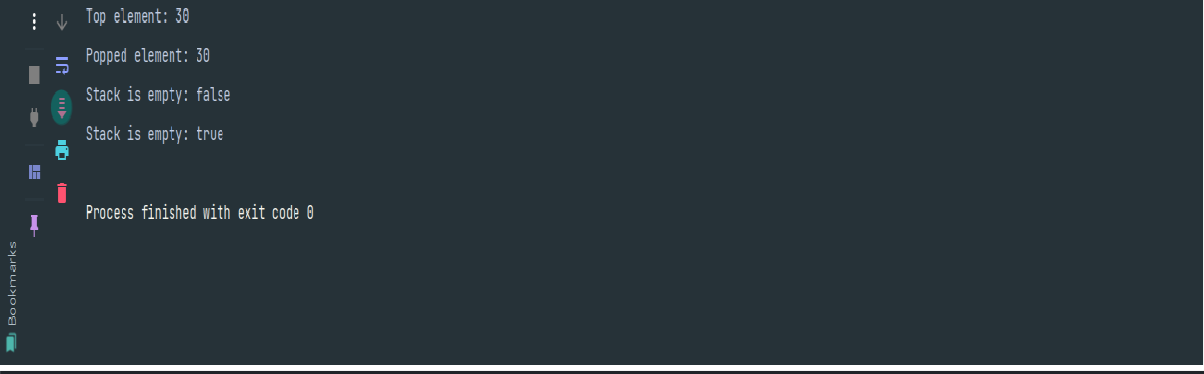
|  |
| --- |
| }  }  *public static* Node findMiddle(Node head) { *if* (head == *null*) { *return null*;  }  Node slow = head; Node fast = head;  *while* (fast != *null* && fast.next != *null*) { slow = slow.next;  fast = fast.next.next;  }  *return* slow;  }  }  **Output:**    **Lab Experiment 9:**  **Merge Two Sorted Linked Lists: Write a function to merge two sorted linked lists into one sorted linked list.**  **Code:**  *class Node { int data;*  *Node next;*  *Node(int data) { this.data = data; this.next = null;* |

|  |
| --- |
| *}*  *}*  *public class MergeSortedLinkedLists { public static void main(String[] args) { Node head1 = new Node(1); head1.next = new Node(3); head1.next.next = new Node(5); Node head2 = new Node(2); head2.next = new Node(4); head2.next.next = new Node(6);*  *Node mergedHead = mergeSortedLists(head1, head2); System.****out****.print("Merged Sorted Linked List: "); printList(mergedHead);*  *}*  *public static Node mergeSortedLists(Node head1, Node head2) { Node dummy = new Node(0); Node current = dummy;*  *while (head1 != null && head2 != null) { if (head1.data <= head2.data) { current.next = head1;*  *head1 = head1.next;*  *} else {*  *current.next = head2;*  *head2 = head2.next;*  *}*  *current = current.next;*  *}*  *if (head1 != null) { current.next = head1;*  *} else {*  *current.next = head2;*  *}* |

|  |
| --- |
| *return dummy.next;*  *}*  *public static void printList(Node head) { Node current = head; while (current != null) {*  *System.****out****.print(current.data + " "); current = current.next;*  *}*  *System.****out****.println();*  *}*  *}*  **Output:**    **Lab Experiment 10:**  **Remove Nth Node from End of List: Write a function to remove the Nth node from the start/end of a linked list.**  **Code:**  *class* Node { *int* data;  Node next;  Node(*int* data) { *this*.data = data;  *this*.next = *null*;  }  }  *public class* RemoveNthNodeFromEnd { |

|  |
| --- |
| *public static void* main(String[] args) { Node head = *new* Node(1); head.next = *new* Node(2); head.next.next = *new* Node(3); head.next.next.next = *new* Node(4); head.next.next.next.next = *new* Node(5); *int* N = 2;  head = *removeNthFromEnd*(head, N);  System.*out*.print("Modified Linked List: "); *printList*(head);  }  *public static* Node removeNthFromEnd(Node head, *int* N) { Node dummy = *new* Node(0);  dummy.next = head; Node first = dummy;  Node second = dummy;  *for* (*int* i = 1; i <= N + 1; i++) { first = first.next;  }  *while* (first != *null*) { first = first.next;  second = second.next;  }  second.next = second.next.next; *return* dummy.next;  }  *public static void* printList(Node head) { Node current = head; *while* (current != *null*) {  System.*out*.print(current.data + " "); current = current.next;  }  System.*out*.println(); |

|  |
| --- |
| }  }  **Output:**    **Lab Experiment 11:**  **Implement a Stack Using Arrays/Lists: Write a function to implement a stack using an array or list with basic operations: push, pop, peek, and isEmpty. Code:**  *import* java.util.ArrayList; *public class* Stack { *private* ArrayList<Integer> stack;  *public* Stack() { stack = *new* ArrayList<>();  }  *public void* push(*int* item) { stack.add(item);  }  *public int* pop() { *if* (isEmpty()) { *throw new* IllegalStateException("Pop from empty stack");  }  *return* stack.remove(stack.size() - 1);  }  *// Peek method to return the top item of the stack without removing it* |



*public*

*int*

peek()

{

*if*

(

isEmpty

())

{

*throw*

*new*

IllegalStateException("Peek

from

empty

stack");

}

*return*

stack.get(stack.size()

-

1)

;

}

*public*

*boolean*

isEmpty()

{

*return*

stack.isEmpty();

}

*public*

*static*

*void*

main(String[]

args)

{

Stack stack =

*new*

Stack();

stack.push(10);

stack.push(20);

stack.push(30);

System.

*out*

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

System.

*out*

.println("Popped element: " + stack.pop());

System.

*out*

.println("Stack

is

empty:

"

+

stack.isEmpty());

stack.pop();

stack.pop();

System.

*out*

.println("Stack

is

empty:

"

+

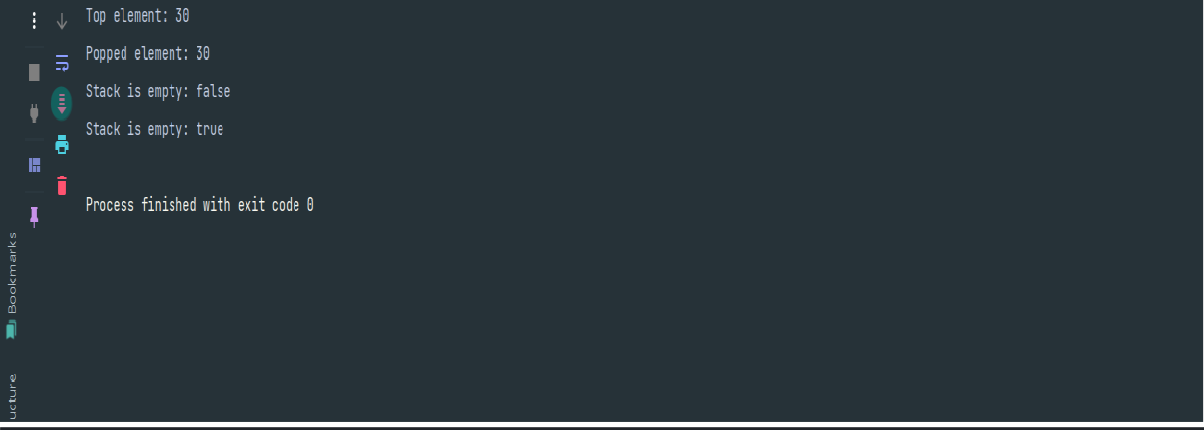
stack.isEmpty());

}

}

**Output:**

|  |
| --- |
| **Lab Experiment 12:**  **Implement a Stack Using Linked List: Write a function to implement a stack using alinked list with basic operations: push, pop, peek, and isEmpty.**  **Code:**  *public class* Stack{ *private static class* Node { *int* data;  Node next;  Node(*int* data) { *this*.data = data;  *this*.next = *null*;  }  }  *private* Node top; *public* Stack() { top = *null*;  }  *public void* push(*int* data) {  Node newNode = *new* Node(data); newNode.next = top;  top = newNode;  }  *public int* pop() { *if* (isEmpty()) { *throw new* IllegalStateException("Pop from empty stack");  }  *int* data = top.data; top = top.next;  *return* data;  } |



*public*

*int*

peek()

{

*if*

(

isEmpty

())

{

*throw*

*new*

IllegalStateException("Peek

from

empty

stack");

}

*return*

top.data;

}

*public*

*boolean*

isEmpty()

{

*return*

top

==

*null*

;

}

*public*

*static*

*void*

main(String[]

args)

{

Stack stack =

*new*

Stack();

stack.push(10);

stack.push(20);

stack.push(30);

System.

*out*

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

System.

*out*

.println("Popped element: " + stack.pop());

System.

*out*

.println("Stack

is

empty:

"

+

stack.isEmpty());

stack.pop();

stack.pop();

System.

*out*

.println("Stack

is

empty:

"

+

stack.isEmpty());

}

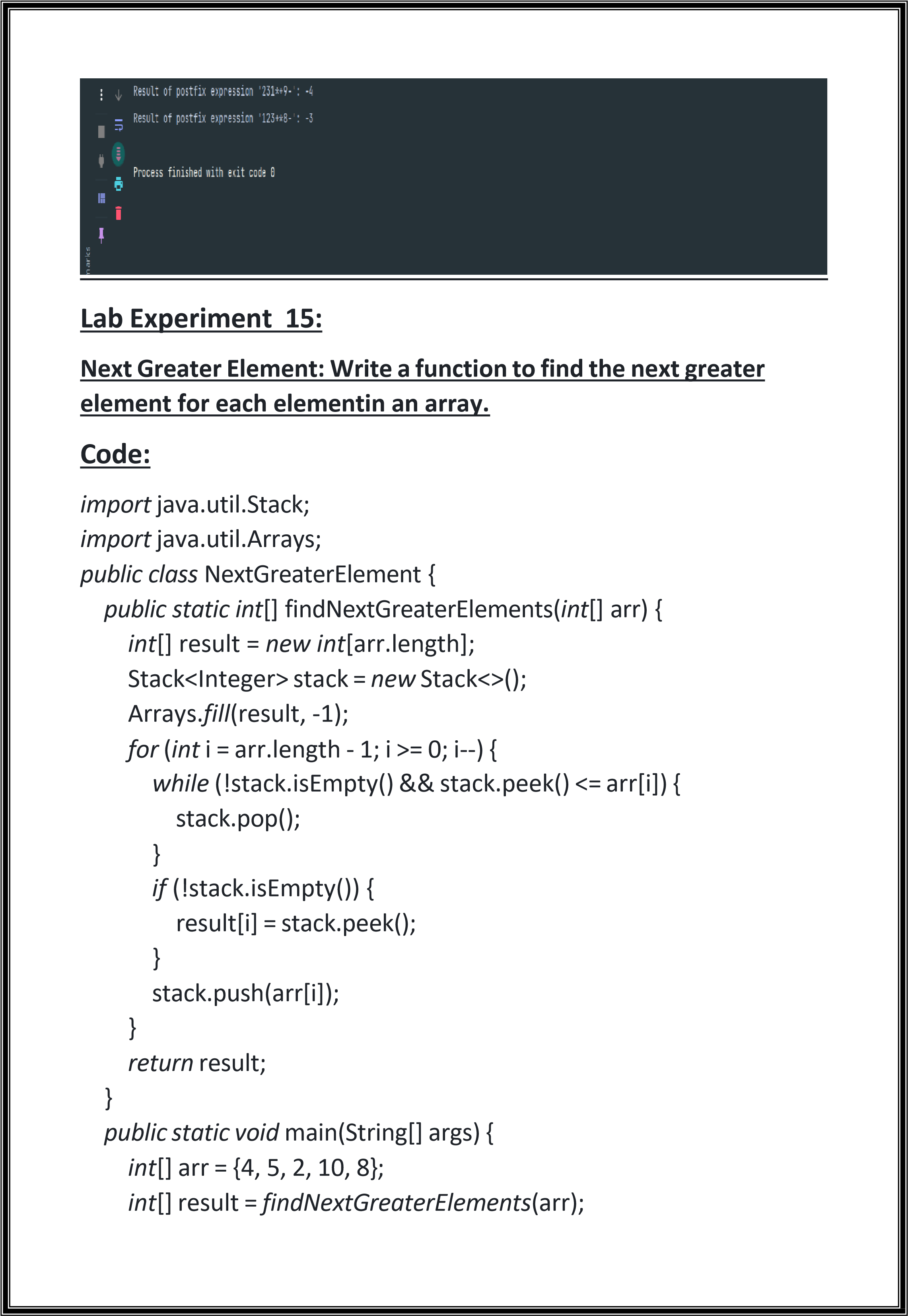
}

**Output:**

|  |
| --- |
| **Lab Experiment 13:**  **Check for Balanced Parentheses: Write a function to check if a string containing parentheses is balanced.**  **Code:**  *import* java.util.Stack;  *public class* BalancedParentheses { *public static boolean* isBalanced(String str) { Stack<Character> stack = *new* Stack<>();  *for* (*char* ch : str.toCharArray()) { *if* (ch == '(' || ch == '{' || ch == '[') { stack.push(ch);  } *else if* (ch == ')' || ch == '}' || ch == ']') { *if* (stack.isEmpty()) { *return false*;  }  *char* top = stack.pop(); *if* ((ch == ')' && top != '(') ||  (ch == '}' && top != '{') ||  (ch == ']' && top != '[')) { *return false*;  }  }  }  *return* stack.isEmpty();  }  *public static void* main(String[] args) {  String testString1 = "{[()]}";  System.*out*.println("Is the string '" + testString1 + "' balanced? "  + *isBalanced*(testString1));  String testString2 = "{[(])}";  System.*out*.println("Is the string '" + testString2 + "' balanced? " |

|  |
| --- |
| + *isBalanced*(testString2));  String testString3 = "((()))";  System.*out*.println("Is the string '" + testString3 + "' balanced? " + *isBalanced*(testString3));  }  }  **Output:**    **Lab Experiment 14:**  **Evaluate Postfix Expression: Write a function to evaluate a given postfix expression.**  **Code:**  *import* java.util.Stack; *public class* PostfixEvaluator { *public static int* evaluatePostfix(String expression) { Stack<Integer> stack = *new* Stack<>(); *for* (*char* ch : expression.toCharArray()) { *if* (Character.*isDigit*(ch)) { stack.push(ch - '0');  } *else* {  *int* operand2 = stack.pop(); *int* operand1 = stack.pop();  *switch* (ch) { *case* '+':  stack.push(operand1 + operand2); *break*; |

|  |
| --- |
| *case* '-':  stack.push(operand1 - operand2); *break*;  *case* '\*':  stack.push(operand1 \* operand2); *break*;  *case* '/':  stack.push(operand1 / operand2); *break*;  *default*:  *throw new* IllegalArgumentException("Invalid operator: " + ch);  }  }  }  *return* stack.pop();  }  *public static void* main(String[] args) {  String postfixExpression1 = "231\*+9-";  System.*out*.println("Result of postfix expression '" + postfixExpression1 + "': " + *evaluatePostfix*(postfixExpression1));  String postfixExpression2 = "123+\*8-";  System.*out*.println("Result of postfix expression '" + postfixExpression2 + "': " + *evaluatePostfix*(postfixExpression2));  }  }  **Output:** |

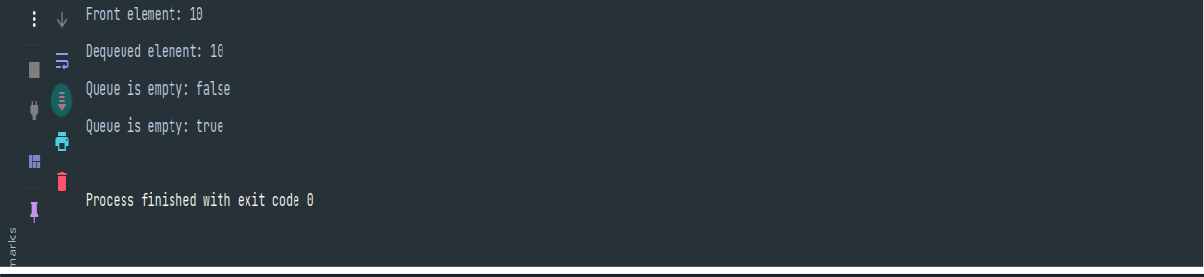


|  |
| --- |
| System.*out*.println("Array: " + Arrays.*toString*(arr));  System.*out*.println("Next Greater Elements: " + Arrays.*toString*(result));  }  }  **Output:**    **Lab Experiment 16:**  **Implement a Queue Using Arrays/Lists: Write a function to implement a queue using an array or list with basic operations: enqueue, dequeue, front, and isEmpty.**  **Code:**  *import* java.util.ArrayList; *public class* Queue { *private* ArrayList<Integer> queue; *public* Queue() { queue = *new* ArrayList<>();  }  *public void* enqueue(*int* item) { queue.add(item);  }  *public int* dequeue() { *if* (isEmpty()) { *throw new* IllegalStateException("Dequeue from empty |

|  |
| --- |
| queue");  }  *return* queue.remove(0);  }  *public int* front() { *if* (isEmpty()) { *throw new* IllegalStateException("Front from empty queue");  }  *return* queue.get(0);  }  *public boolean* isEmpty() { *return* queue.isEmpty();  }  *public static void* main(String[] args) { Queue = *new* Queue(); queue.enqueue(10); queue.enqueue(20); queue.enqueue(30);  System.*out*.println("Front element: " + queue.front());  System.*out*.println("Dequeued element: " + queue.dequeue()); System.*out*.println("Queue is empty: " + queue.isEmpty()); queue.dequeue(); queue.dequeue();  System.*out*.println("Queue is empty: " + queue.isEmpty());  }  }  **Output:** |

|  |
| --- |
| **Lab Experiment 17:**  **Implement a Queue Using Linked List: Write a function to implement a queue using a linked list with basic operations: enqueue, dequeue, front, and isEmpty.**  **Code:**  *public class* Queue { *private static class* Node { *int* data;  Node next;  Node(*int* data) { *this*.data = data;  *this*.next = *null*;  }  }  *private* Node front; *private* Node rear; *public* Queue() { front = *null*;  rear = *null*;  }  *public void* enqueue(*int* item) {  Node newNode = *new* Node(item);  *if* (rear == *null*) {  front = newNode;  rear = newNode;  } *else* {  rear.next = newNode;  rear = newNode;  }  }  *public int* dequeue() { |

|  |
| --- |
| *if* (isEmpty()) {  *throw new* IllegalStateException("Dequeue from empty  queue");  }  *int* data = front.data; front = front.next; *if* (front == *null*) { rear = *null*;  }  *return* data;  }  *public int* front() { *if* (isEmpty()) { *throw new* IllegalStateException("Front from empty queue");  }  *return* front.data;  }  *public boolean* isEmpty() { *return* front == *null*;  }  *public static void* main(String[] args) { Queue = *new* Queue(); queue.enqueue(10); queue.enqueue(20); queue.enqueue(30);  System.*out*.println("Front element: " + queue.front());  System.*out*.println("Dequeued element: " + queue.dequeue()); System.*out*.println("Queue is empty: " + queue.isEmpty()); queue.dequeue(); queue.dequeue();  System.*out*.println("Queue is empty: " + queue.isEmpty());  }  } |



**Output:**

**Lab**

**Experiment**

**18:**

**Implement a Circular Queue: Write a function to implement a**

**circular**

**queue**

**with**

**basic**

**operations:**

**enqueue,**

**dequeue,**

**front,**

**rear, and isEmpty.**

**Code:**

*public*

*class*

CircularQueue

{

*private int*

[]

queue;

*private int*

front;

*private int*

rear;

*private int*

size;

*private*

*int*

capacity;

*public*

CircularQueue(

*int*

capacity)

{

*this*

.capacity = capacity;

queue

=

*new*

*int*

capacity];

[

front =

-

;

1

rear

=

-

1

;

size

=

;

0

}

*public*

*void*

enqueue(

*int*

item)

{

*if*

isFull

())

(

{

*throw*

*new*

IllegalStateException("Enqueue

on

full

queue");

}

*if*

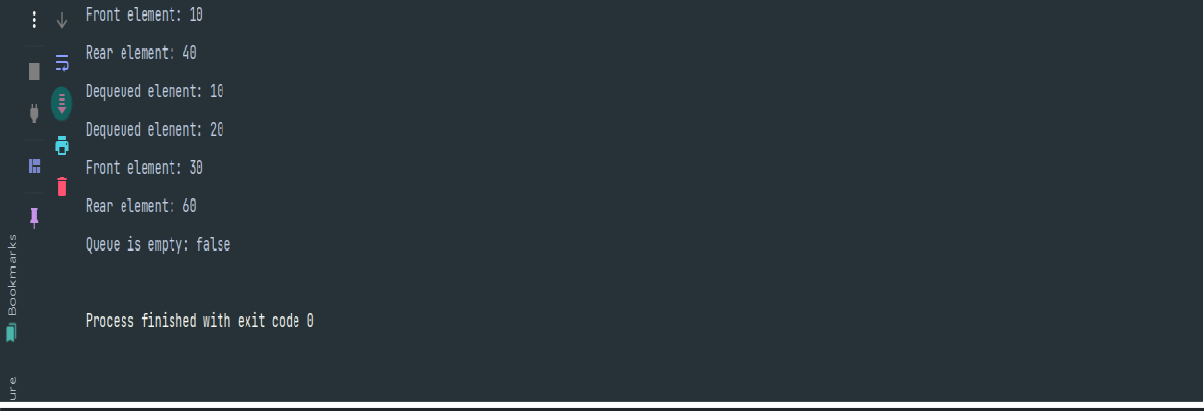
(

isEmpty

())

{

|  |
| --- |
| front = 0;  }  rear = (rear + 1) % capacity; queue[rear] = item;  size++;  }  *public int* dequeue() { *if* (isEmpty()) {  *throw new* IllegalStateException("Dequeue from empty  queue");  }  *int* item = queue[front]; front = (front + 1) % capacity; size--; *if* (size == 0) { front = -1;  rear = -1;  }  *return* item;  }  *public int* front() { *if* (isEmpty()) { *throw new* IllegalStateException("Front from empty queue");  }  *return* queue[front];  }  *public int* rear() { *if* (isEmpty()) {  *throw new* IllegalStateException("Rear from empty queue");  }  *return* queue[rear];  }  *public boolean* isEmpty() { |



*return*

size

==

0

;

}

*public*

*boolean*

isFull()

{

*return*

size

==

capacity;

}

*public static void*

main(String[] args) {

CircularQueue

queue

=

*new*

CircularQueue(5);

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

System.

*out*

.println("Front element: " + queue.front());

System.

*out*

.println("Rear element: " + queue.rear());

System.

*out*

.println("Dequeued

element:

"

+

queue.dequeue());

System.

*out*

.println("Dequeued

element:

"

+

queue.dequeue());

queue.enqueue(50);

queue.enqueue(60);

System.

*out*

.println("Front element: " + queue.front());

System.

*out*

.println("Rear element: " + queue.rear());

System.

*out*

.println("Queue

is

empty:

"

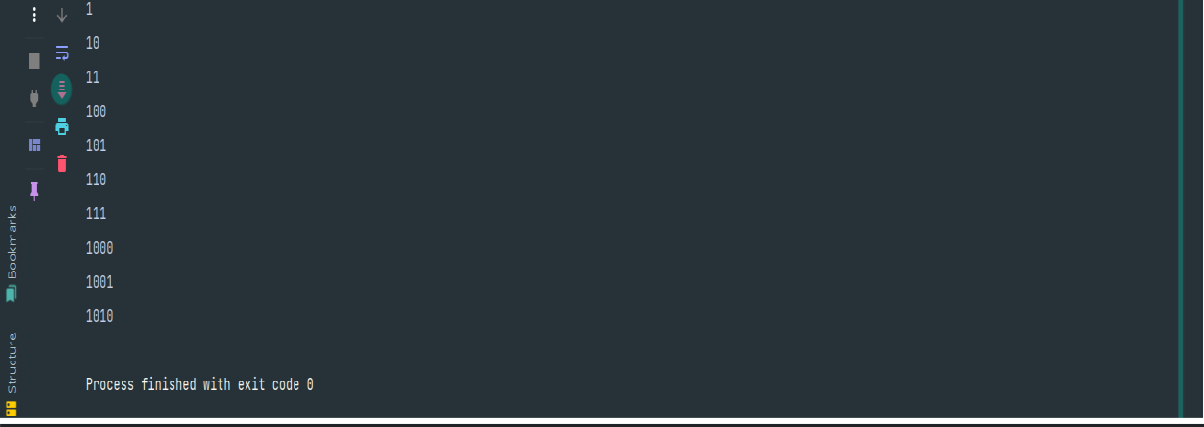
+

queue.isEmpty());

}

}

**Output:**



**Lab**

**Experiment**

**19:**

**Generate**

**Binary**

**Numbers**

**from**

**1**

**to**

**N:**

**Write**

**a**

**function**

**to**

**generate**

**binary numbers from1 to N using a queue.**

**Code:**

*import*

java.util.LinkedList;

*import*

java.util.

*Queue*

;

*public*

*class*

BinaryNumbers

{

*public*

*static*

*void*

generateBinaryNumbers(

*int*

N)

{

*Queue*

<

String> queue =

*new*

LinkedList<>();

queue.add("1");

*for*

(

*int*

i = 1; i <= N; i++) {

String

current

=

queue.poll();

System.

*out*

.println(current);

queue.add(current + "0");

queue.add(current

+

"1");

}

}

*public*

*static*

*void*

main(String[]

args)

{

*int*

N = 10;

*generateBinaryNumbers*

(

N);

}

}

**Output:**

|  |
| --- |
| **Lab Experiment 20:**  **Implement a Queue Using Stacks: Write a function to implement a queue using two stacks. (vice-versa).**  **Code:**  *import* java.util.Stack;  *public class* QueueUsingTwoStacks { *private* Stack<Integer> stack1; *private* Stack<Integer> stack2; *public* QueueUsingTwoStacks() { stack1 = *new* Stack<>(); stack2 = *new* Stack<>();  }  *public void* enqueue(*int* x) { stack1.push(x);  }  *public int* dequeue() { *if* (stack2.isEmpty()) { *if* (stack1.isEmpty()) {  System.*out*.println("Queue is empty!"); *return* -1;  }  *while* (!stack1.isEmpty()) { stack2.push(stack1.pop());  }  }  *return* stack2.pop();  }  *public int* peek() { *if* (stack2.isEmpty()) { *if* (stack1.isEmpty()) {  System.*out*.println("Queue is empty!"); |

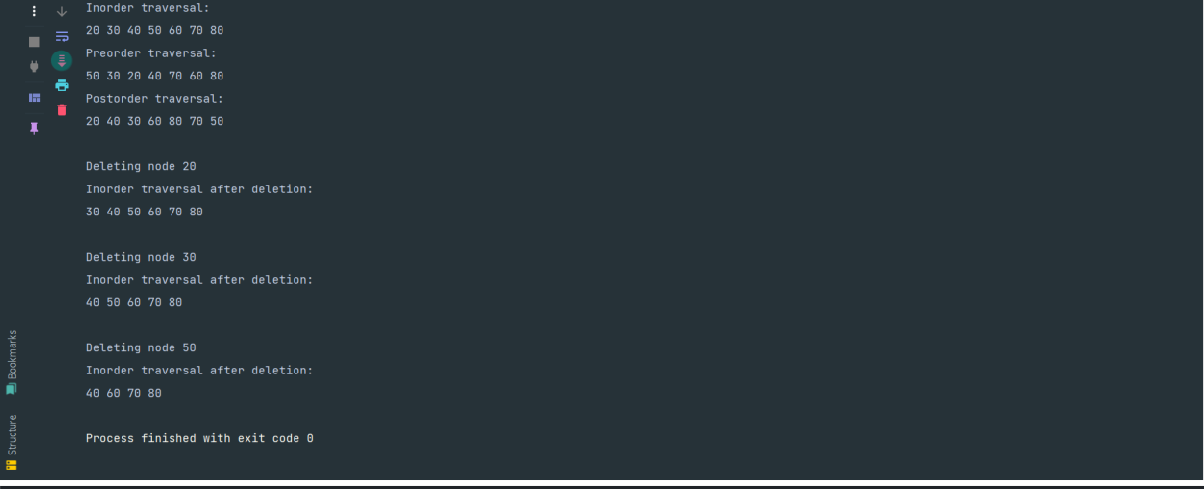
|  |
| --- |
| *return* -1;  }  *while* (!stack1.isEmpty()) { stack2.push(stack1.pop());  }  }  *return* stack2.peek();  }  *public boolean* isEmpty() { *return* stack1.isEmpty() && stack2.isEmpty();  }  *public static void* main(String[] args) {  QueueUsingTwoStacks queue = *new* QueueUsingTwoStacks(); queue.enqueue(10); queue.enqueue(20); queue.enqueue(30);  System.*out*.println("Dequeued: " + queue.dequeue()); System.*out*.println("Dequeued: " + queue.dequeue()); queue.enqueue(40); queue.enqueue(50);  System.*out*.println("Front of the queue: " + queue.peek());  System.*out*.println("Dequeued: " + queue.dequeue());  System.*out*.println("Dequeued: " + queue.dequeue());  System.*out*.println("Dequeued: " + queue.dequeue());  System.*out*.println("Dequeued: " + queue.dequeue());  }  }  **Output:** |

|  |
| --- |
| **Lab Experiment 21:**  **Implement a Binary Tree: Write a class to implement a basic binary tree with insert,delete, and traversal operations.**  **Code:**  *public class* BinaryTree { *static class* Node { *int* data;  Node left, right; *public* Node(*int* data) { *this*.data = data; *this*.left = *null*;  *this*.right = *null*;  }  }  *private* Node root; *public* BinaryTree() { root = *null*;  }  *public void* insert(*int* data) { root = insertRec(root, data);  }  *private* Node insertRec(Node root, *int* data) { *if* (root == *null*) { root = *new* Node(data);  *return* root;  }  *if* (data < root.data) { root.left = insertRec(root.left, data);  } *else if* (data > root.data) { root.right = insertRec(root.right, data);  } |

|  |
| --- |
| *return* root;  }  *public void* delete(*int* data) { root = deleteRec(root, data);  }  *private* Node deleteRec(Node root, *int* data) { *if* (root == *null*) { *return* root;  }  *if* (data < root.data) { root.left = deleteRec(root.left, data);  } *else if* (data > root.data) { root.right = deleteRec(root.right, data);  } *else* { *if* (root.left == *null*) { *return* root.right;  } *else if* (root.right == *null*) { *return* root.left;  }  root.data = minValue(root.right);  root.right = deleteRec(root.right, root.data);  }  *return* root;  }  *private int* minValue(Node root) { *int* minVal = root.data; *while* (root.left != *null*) { minVal = root.left.data; root = root.left;  }  *return* minVal;  } |

|  |
| --- |
| *public void* inorder() { inorderRec(root);  System.*out*.println();  }  *private void* inorderRec(Node root) { *if* (root != *null*) { inorderRec(root.left);  System.*out*.print(root.data + " "); inorderRec(root.right);  }  }  *public void* preorder() { preorderRec(root);  System.*out*.println();  }  *private void* preorderRec(Node root) { *if* (root != *null*) {  System.*out*.print(root.data + " "); preorderRec(root.left);  preorderRec(root.right);  }  }  *public void* postorder() { postorderRec(root); System.*out*.println();  }  *private void* postorderRec(Node root) { *if* (root != *null*) { postorderRec(root.left); postorderRec(root.right);  System.*out*.print(root.data + " ");  }  } |

|  |
| --- |
| *public static void* main(String[] args) { BinaryTree tree = *new* BinaryTree();  tree.insert(50); tree.insert(30); tree.insert(20); tree.insert(40); tree.insert(70); tree.insert(60); tree.insert(80);  System.*out*.println("Inorder traversal:"); tree.inorder();  System.*out*.println("Preorder traversal:"); tree.preorder();  System.*out*.println("Postorder traversal:"); tree.postorder();  System.*out*.println("\nDeleting node 20"); tree.delete(20);  System.*out*.println("Inorder traversal after deletion:"); tree.inorder();  System.*out*.println("\nDeleting node 30"); tree.delete(30);  System.*out*.println("Inorder traversal after deletion:"); tree.inorder();  System.*out*.println("\nDeleting node 50"); tree.delete(50);  System.*out*.println("Inorder traversal after deletion:"); tree.inorder();  }  }  **Output:** |



**Lab**

**Experiment**

**22:**

**Inorder**

**Traversal:**

**Write**

**a**

**function**

**to**

**perform**

**inorder**

**traversal**

**of**

**a**

**binary tree.**

**Code:**

*public*

*class*

BinaryTree

{

*static*

*class*

Node

{

*int*

data;

Node

left,

right;

*public*

Node(

*int*

data)

{

*this*

.data = data;

*this*

.left =

*null*

;

*this*

.right =

*null*

;

}

}

*public*

*static void*

inorderTraversal(Node

root)

{

*if*

(

root !=

*null*

)

{

*inorderTraversal*

root.left);

(

System.

*out*

.print(root.data

+

"

");

*inorderTraversal*

(

root.right);

}

}

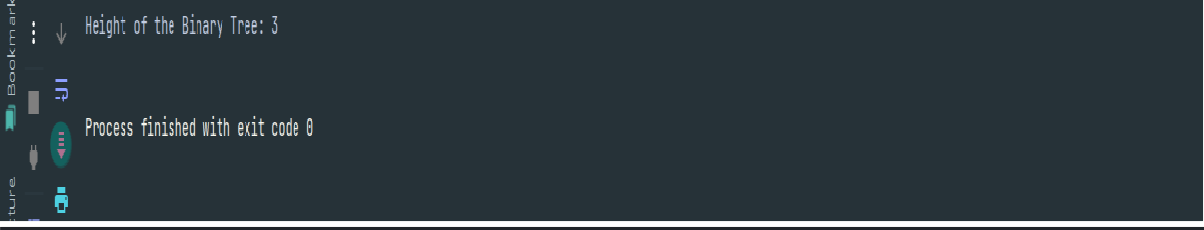
|  |
| --- |
| *public static void* main(String[] args) { Node root = *new* Node(50); root.left = *new* Node(30); root.right = *new* Node(70); root.left.left = *new* Node(20); root.left.right = *new* Node(40); root.right.left = *new* Node(60); root.right.right = *new* Node(80);  System.*out*.println("Inorder Traversal:"); *inorderTraversal*(root);  }  }  **Output:**    **Lab Experiment 23:**  **Preorder Traversal: Write a function to perform preorder traversal of a binary tree.**  **Code:**  *public class* BinaryTree { *static class* Node { *int* data;  Node left, right; *public* Node(*int* data) { *this*.data = data; *this*.left = *null*;  *this*.right = *null*;  }  } |

|  |
| --- |
| *public static void* preorderTraversal(Node root) { *if* (root != *null*) {  System.*out*.print(root.data + " "); *preorderTraversal*(root.left);  *preorderTraversal*(root.right);  }  }  *public static void* main(String[] args) { Node root = *new* Node(50); root.left = *new* Node(30); root.right = *new* Node(70); root.left.left = *new* Node(20); root.left.right = *new* Node(40); root.right.left = *new* Node(60); root.right.right = *new* Node(80);  System.*out*.println("Preorder Traversal:"); *preorderTraversal*(root);  }  }  **Output:**    **Lab Experiment 24:**  **Postorder Traversal: Write a function to perform postorder traversal of a binary tree.**  **Code:**  *public class* BinaryTree { *static class* Node { *int* data; |

|  |
| --- |
| Node left, right; *public* Node(*int* data) { *this*.data = data; *this*.left = *null*;  *this*.right = *null*;  }  }  *public static void* postorderTraversal(Node root) { *if* (root != *null*) { *postorderTraversal*(root.left); *postorderTraversal*(root.right);  System.*out*.print(root.data + " ");  }  }  *public static void* main(String[] args) { Node root = *new* Node(50); root.left = *new* Node(30); root.right = *new* Node(70); root.left.left = *new* Node(20); root.left.right = *new* Node(40); root.right.left = *new* Node(60); root.right.right = *new* Node(80);  System.*out*.println("Postorder Traversal:"); *postorderTraversal*(root);  }  }  **Output:** |

|  |
| --- |
| **Lab Experiment 25:**  **Level Order Traversal: Write a function to perform level order traversal of a binary tree. Code:**  *import* java.util.LinkedList; *import* java.util.*Queue*;  *public class* BinaryTree { *static class* Node { *int* data;  Node left, right; *public* Node(*int* data) { *this*.data = data; *this*.left = *null*;  *this*.right = *null*;  }  }  *public static void* levelOrderTraversal(Node root) { *if* (root == *null*) { *return*;  }  *Queue*<Node> queue = *new* LinkedList<>(); queue.add(root);  *while* (!queue.isEmpty()) {  Node current = queue.poll();  System.*out*.print(current.data + " ");  *if* (current.left != *null*) { queue.add(current.left);  }  *if* (current.right != *null*) { |

|  |
| --- |
| queue.add(current.right);  }  }  }  *public static void* main(String[] args) { Node root = *new* Node(50); root.left = *new* Node(30); root.right = *new* Node(70); root.left.left = *new* Node(20); root.left.right = *new* Node(40); root.right.left = *new* Node(60); root.right.right = *new* Node(80);  System.*out*.println("Level Order Traversal:"); *levelOrderTraversal*(root);  }  }  **Output:**    **Lab Experiment 26:**  **Height of a Binary Tree: Write a function to find the height of a binary tree.**  **Code:**  *public class* BinaryTree { *static class* Node { *int* data;  Node left, right; *public* Node(*int* data) { *this*.data = data; |



*this*

.left =

*null*

;

*this*

.right

=

*null*

;

}

}

*public*

*static*

*int*

findHeight(Node

root)

{

*if*

(

root

==

*null*

)

{

*return*

0

;

}

*int*

leftHeight =

*findHeight*

(

root.left);

*int*

rightHeight

=

*findHeight*

(

root.right);

*return*

Math.

*max*

(

leftHeight,

rightHeight)

+

1

;

}

*public*

*static*

*void*

main(String[]

args)

{

Node root =

*new*

Node(50);

root.left =

*new*

Node(30);

root.right =

*new*

Node(70);

root.left.left =

*new*

Node(20);

root.left.right =

*new*

Node(40);

root.right.left =

*new*

Node(60);

root.right.right =

*new*

Node(80);

System.

*out*

.println("Height

of

the

Binary

Tree:

"

+

*findHeight*

(

root));

}

}

**Output:**

|  |
| --- |
| **Lab Experiment 27:**  **Diameter of a Binary Tree: Write a function to find the diameter of a binary tree.**  **Code:**  *public class* BinaryTree { *static class* Node { *int* data;  Node left, right; *public* Node(*int* data) { *this*.data = data; *this*.left = *null*;  *this*.right = *null*;  }  }  *static class* Height { *int* value = 0;  }  *public static int* findDiameter(Node root, Height height) { *if* (root == *null*) { height.value = 0; *return* 0;  }  Height leftHeight = *new* Height(); Height rightHeight = *new* Height();  *int* leftDiameter = *findDiameter*(root.left, leftHeight); *int* rightDiameter = *findDiameter*(root.right, rightHeight); height.value = Math.*max*(leftHeight.value, rightHeight.value) + 1; *int* currentDiameter = leftHeight.value + rightHeight.value + 1; *return* Math.*max*(currentDiameter, Math.*max*(leftDiameter,  rightDiameter));  } |

|  |
| --- |
| *public static int* getDiameter(Node root) { Height = *new* Height();  *return findDiameter*(root, height);  }  *public static void* main(String[] args) { Node root = *new* Node(50); root.left = *new* Node(30); root.right = *new* Node(70); root.left.left = *new* Node(20); root.left.right = *new* Node(40); root.right.left = *new* Node(60); root.right.right = *new* Node(80);  System.*out*.println("Diameter of the Binary Tree: " + *getDiameter*(root));  }  }  **Output:**    **Lab Experiment 28:**  **Check if a Binary Tree is Balanced: Write a function to check if a binary tree is height balanced.**  **Code:**  *class* TreeNode { *int* val;  TreeNode left;  TreeNode right;  TreeNode(*int* val) { *this*.val = val; |

|  |
| --- |
| left = *null*;  right = *null*;  }  }  *public class* BinaryTree { *public static boolean* isBalanced(TreeNode root) { *return checkHeight*(root) != -1;  }  *private static int* checkHeight(TreeNode root) { *if* (root == *null*) { *return* 0;  }  *int* leftHeight = *checkHeight*(root.left);  *if* (leftHeight == -1) { *return* -1;  }  *int* rightHeight = *checkHeight*(root.right);  *if* (rightHeight == -1) { *return* -1;  }  *if* (Math.*abs*(leftHeight - rightHeight) > 1) { *return* -1;  }  *return* Math.*max*(leftHeight, rightHeight) + 1;  }  *public static void* main(String[] args) { TreeNode root = *new* TreeNode(1); root.left = *new* TreeNode(2); root.right = *new* TreeNode(3); root.left.left = *new* TreeNode(4); root.left.right = *new* TreeNode(5); root.left.left.left = *new* TreeNode(6); *if* (*isBalanced*(root)) { |

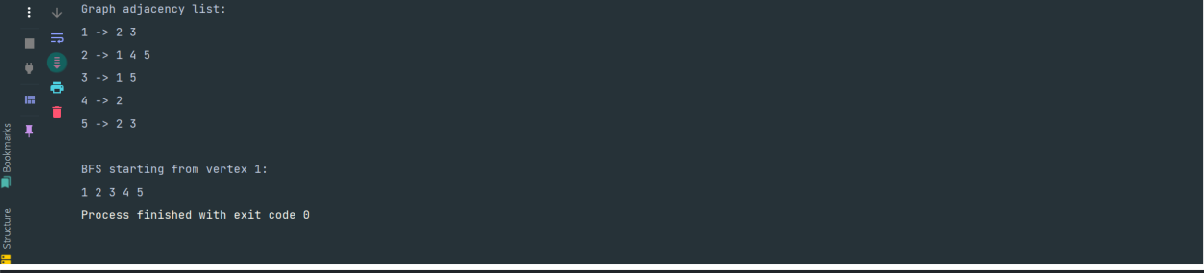
|  |
| --- |
| System.*out*.println("The binary tree is balanced.");  } *else* {  System.*out*.println("The binary tree is not balanced.");  }  }  }  **Output:**    **Lab Experiment 29:**  **Lowest Common Ancestor: Write a function to find the lowest common ancestor of two nodes in a binary tree.**  **Code:**  *class* TreeNode { *int* val;  TreeNode left;  TreeNode right;  TreeNode(*int* val) { *this*.val = val; left = *null*;  right = *null*;  }  }  *public class* LowestCommonAncestor { *public static* TreeNode lowestCommonAncestor(TreeNode root,  TreeNode p, TreeNode q) { *if* (root == *null* || root == p || root == q) { *return* root; |

|  |
| --- |
| }  TreeNode leftLCA = *lowestCommonAncestor*(root.left, p, q); TreeNode rightLCA = *lowestCommonAncestor*(root.right, p, q); *if* (leftLCA != *null* && rightLCA != *null*) { *return* root;  }  *return* (leftLCA != *null*) ? leftLCA : rightLCA;  }  *public static void* main(String[] args) { TreeNode root = *new* TreeNode(3); root.left = *new* TreeNode(5); root.right = *new* TreeNode(1); root.left.left = *new* TreeNode(6); root.left.right = *new* TreeNode(2); root.right.left = *new* TreeNode(0); root.right.right = *new* TreeNode(8); root.left.right.left = *new* TreeNode(7); root.left.right.right = *new* TreeNode(4);  TreeNode p = root.left;  TreeNode q = root.left.right.right;  TreeNode lca = *lowestCommonAncestor*(root, p, q);  System.*out*.println("The Lowest Common Ancestor is: " + (lca !=  *null* ? lca.val : "null"));  }  }  **Output:** |

|  |
| --- |
| **Lab Experiment 30:**  **Implement Graph Using Adjacency List: Write a class to implement a basic graph usingan adjacency list with methods to add vertices and edges.**  **Code:**  *import* java.util.\*; *public class* Graph { *private Map*<Integer, *List*<Integer>> adjList; *public* Graph() { adjList = *new* HashMap<>();  }  *public void* addVertex(*int* vertex) { adjList.putIfAbsent(vertex, *new* ArrayList<>());  }  *public void* addEdge(*int* vertex1, *int* vertex2) { addVertex(vertex1); addVertex(vertex2);  adjList.get(vertex1).add(vertex2); adjList.get(vertex2).add(vertex1);  }  *public void* displayGraph() { *for* (*Map*.*Entry*<Integer, *List*<Integer>> entry : adjList.entrySet()) { System.*out*.print(entry.getKey() + " -> "); *for* (Integer neighbor : entry.getValue()) { System.*out*.print(neighbor + " ");  }  System.*out*.println();  }  }  *public static void* main(String[] args) {  Graph = *new* Graph(); |

|  |
| --- |
| graph.addVertex(1); graph.addVertex(2); graph.addVertex(3); graph.addVertex(4); graph.addEdge(1, 2); graph.addEdge(1, 3); graph.addEdge(2, 4); graph.addEdge(3, 4);  graph.displayGraph();  }  }  **Output:**    **Lab Experiment 31:**  **Breadth-First Search (BFS): Write a function to perform BFS on a graph from a givenstart vertex.**  **Code:**  *import* java.util.\*; *public class* Graph { *private Map*<Integer, *List*<Integer>> adjList; *public* Graph() { adjList = *new* HashMap<>();  }  *public void* addVertex(*int* vertex) { adjList.putIfAbsent(vertex, *new* ArrayList<>());  } |

|  |
| --- |
| *public void* addEdge(*int* vertex1, *int* vertex2) { addVertex(vertex1); addVertex(vertex2);  adjList.get(vertex1).add(vertex2); adjList.get(vertex2).add(vertex1);  }  *public void* bfs(*int* startVertex) {  *Set*<Integer> visited = *new* HashSet<>(); *Queue*<Integer> queue = *new* LinkedList<>();  visited.add(startVertex); queue.offer(startVertex); *while* (!queue.isEmpty()) { *int* vertex = queue.poll(); System.*out*.print(vertex + " "); *for* (*int* neighbor : adjList.get(vertex)) { *if* (!visited.contains(neighbor)) { visited.add(neighbor);  queue.offer(neighbor);  }  }  }  }  *public void* displayGraph() { *for* (*Map*.*Entry*<Integer, *List*<Integer>> entry : adjList.entrySet()) { System.*out*.print(entry.getKey() + " -> "); *for* (Integer neighbor : entry.getValue()) { System.*out*.print(neighbor + " ");  }  System.*out*.println();  }  }  *public static void* main(String[] args) {  Graph = *new* Graph(); |



graph.addVertex(1);

graph.addVertex(2);

graph.addVertex(3);

graph.addVertex(4);

graph.addVertex(5);

graph.addEdge(1,

2)

;

graph.addEdge(1,

3)

;

graph.addEdge(2,

4)

;

graph.addEdge(2,

5)

;

graph.addEdge(3, 5);

System.

*out*

.println("Graph

adjacency

list:");

graph.displayGraph();

System.

*out*

.println("

\

nBFS

starting

from

vertex

1:

");

graph.bfs(1);

}

}

**Output:**

**Lab**

**Experiment**

**32:**

**Depth**

**-**

**First**

**Search**

**(**

**DFS**

**):**

**Write**

**a**

**function**

**to**

**perform**

**DFS**

**on**

**a**

**graph from a given start vertex.**

**Code:**

*import*

java.util.\*;

*public*

*class*

Graph

{

*private*

*Map*

Integer,

<

*List*

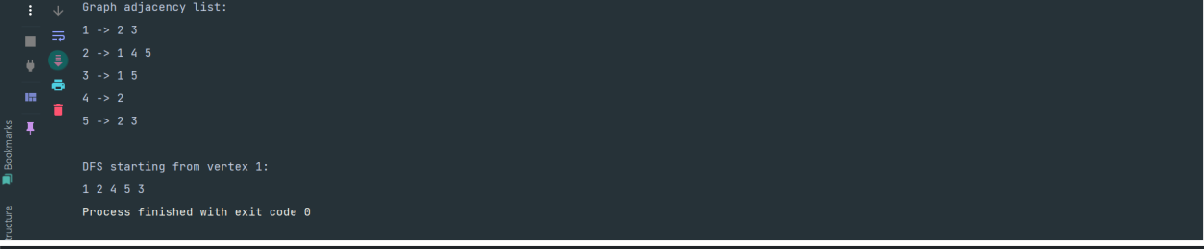
<

Integer

>>

adjList;

|  |
| --- |
| *public* Graph() { adjList = *new* HashMap<>();  }  *public void* addVertex(*int* vertex) { adjList.putIfAbsent(vertex, *new* ArrayList<>());  }  *public void* addEdge(*int* vertex1, *int* vertex2) { addVertex(vertex1); addVertex(vertex2);  adjList.get(vertex1).add(vertex2); adjList.get(vertex2).add(vertex1);  }  *public void* dfs(*int* startVertex) {  *Set*<Integer> visited = *new* HashSet<>(); dfsRecursive(startVertex, visited);  }  *private void* dfsRecursive(*int* vertex, *Set*<Integer> visited) { visited.add(vertex);  System.*out*.print(vertex + " "); *for* (*int* neighbor : adjList.get(vertex)) { *if* (!visited.contains(neighbor)) { dfsRecursive(neighbor, visited);  }  }  }  *public void* displayGraph() { *for* (*Map*.*Entry*<Integer, *List*<Integer>> entry : adjList.entrySet()) { System.*out*.print(entry.getKey() + " -> "); *for* (Integer neighbor : entry.getValue()) { System.*out*.print(neighbor + " ");  }  System.*out*.println();  } |



}

*public*

*static*

*void*

main(String[]

args)

{

Graph graph =

*new*

Graph();

graph.addVertex(1);

graph.addVertex(2);

graph.addVertex(3);

graph.addVertex(4);

graph.addVertex(5);

graph.addEdge(1, 2);

graph.addEdge(1,

3)

;

graph.addEdge(2,

4)

;

graph.addEdge(2,

5)

;

graph.addEdge(3, 5);

System.

*out*

.println("Graph

adjacency

list:");

graph.displayGraph();

System.

*out*

.println("

\

nDFS

starting

from

vertex

1:

");

graph.dfs(1);

}

}

**Output:**

**Lab**

**Experiment**

**33:**

**Detect**

**Cycle**

**in**

**an**

**Undirected**

**Graph:**

**Write**

**a**

**function**

**to**

**detect**

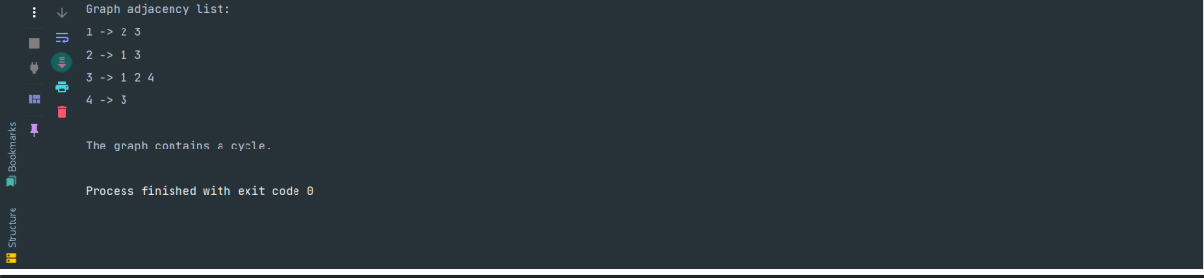
**if**

**there is a cycle in an undirected graph.**

**Code:**

|  |
| --- |
| *import* java.util.\*; *public class* Graph { *private Map*<Integer, *List*<Integer>> adjList; *public* Graph() { adjList = *new* HashMap<>();  }  *public void* addVertex(*int* vertex) { adjList.putIfAbsent(vertex, *new* ArrayList<>());  }  *public void* addEdge(*int* vertex1, *int* vertex2) { addVertex(vertex1); addVertex(vertex2);  adjList.get(vertex1).add(vertex2); adjList.get(vertex2).add(vertex1);  }  *public boolean* hasCycle() {  *Set*<Integer> visited = *new* HashSet<>(); *for* (Integer vertex : adjList.keySet()) { *if* (!visited.contains(vertex)) { *if* (hasCycleDFS(vertex, visited, -1)) { *return true*;  }  }  }  *return false*;  }  *private boolean* hasCycleDFS(*int* currentVertex, *Set*<Integer>  visited, *int* parent) { visited.add(currentVertex);  *for* (*int* neighbor : adjList.get(currentVertex)) { *if* (!visited.contains(neighbor)) { *if* (hasCycleDFS(neighbor, visited, currentVertex)) { *return true*; |

|  |
| --- |
| }  }  *else if* (neighbor != parent) { *return true*;  }  }  *return false*;  }  *public void* displayGraph() { *for* (*Map*.*Entry*<Integer, *List*<Integer>> entry : adjList.entrySet()) { System.*out*.print(entry.getKey() + " -> "); *for* (Integer neighbor : entry.getValue()) { System.*out*.print(neighbor + " ");  }  System.*out*.println();  }  }  *public static void* main(String[] args) { Graph = *new* Graph(); graph.addVertex(1); graph.addVertex(2); graph.addVertex(3); graph.addVertex(4); graph.addEdge(1, 2); graph.addEdge(1, 3); graph.addEdge(2, 3); graph.addEdge(3, 4);  System.*out*.println("Graph adjacency list:"); graph.displayGraph(); *if* (graph.hasCycle()) {  System.*out*.println("\nThe graph contains a cycle.");  } *else* {  System.*out*.println("\nThe graph does not contain a cycle."); |



}

}

}

**Output:**

**Lab**

**Experiment**

**34:**

**Connected**

**Components**

**in**

**an**

**Undirected**

**Graph:**

**Write**

**a**

**function**

**to**

**find the number of connected components in an undirected graph.**

**Code:**

*import*

java.util.\*;

*public*

*class*

Graph

{

*private*

*Map*

<

Integer,

*List*

Integer

>>

<

adjList;

*public*

Graph()

{

adjList

=

*new*

HashMap<>();

}

*public void*

addVertex(

*int*

vertex) {

adjList.putIfAbsent(vertex,

*new*

ArrayList<>());

}

*public*

*void*

addEdge(

*int*

vertex1,

*int*

vertex2)

{

addVertex(vertex1);

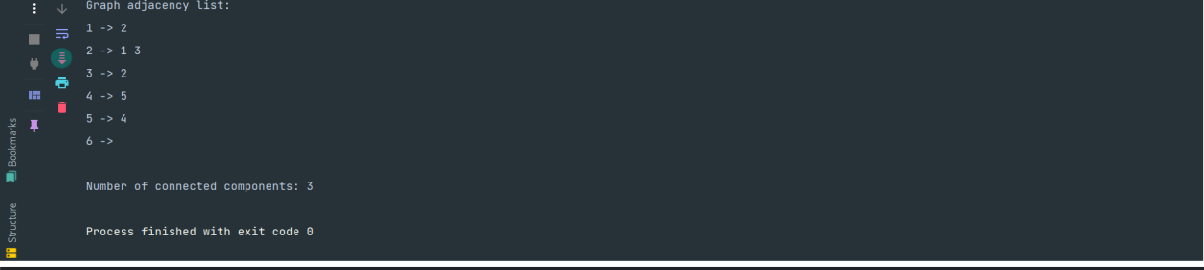
addVertex(vertex2);

adjList.get(vertex1).add(vertex2);

adjList.get(vertex2).add(vertex1);

}

|  |
| --- |
| *public int* findConnectedComponents() { *Set*<Integer> visited = *new* HashSet<>();  *int* componentCount = 0;  *for* (Integer vertex : adjList.keySet()) { *if* (!visited.contains(vertex)) { dfs(vertex, visited);  componentCount++;  }  }  *return* componentCount;  }  *private void* dfs(*int* currentVertex, *Set*<Integer> visited) { visited.add(currentVertex);  *for* (*int* neighbor : adjList.get(currentVertex)) { *if* (!visited.contains(neighbor)) { dfs(neighbor, visited);  }  }  }  *public void* displayGraph() { *for* (*Map*.*Entry*<Integer, *List*<Integer>> entry : adjList.entrySet()) { System.*out*.print(entry.getKey() + " -> "); *for* (Integer neighbor : entry.getValue()) { System.*out*.print(neighbor + " ");  }  System.*out*.println();  }  }  *public static void* main(String[] args) { Graph = *new* Graph(); graph.addVertex(1); graph.addVertex(2); |



graph.addVertex(3);

graph.addVertex(4);

graph.addVertex(5);

graph.addVertex(6);

graph.addEdge(1,

2)

;

graph.addEdge(2,

3)

;

graph.addEdge(4, 5);

System.

*out*

.println("Graph

adjacency

list:");

graph.displayGraph();

*int*

components = graph.findConnectedComponents();

System.

*out*

.println("

\

nNumber

of

connected

components:

"

+

components);

}

}

**Output:**

**Lab**

**Experiment**

**35:**

**Find**

**MST**

**Using**

**Kruskal’s**

**Algorithm:**

**Write**

**a**

**function**

**to**

**find**

**the**

**Minimum Spanning Tree of a graph using Kruskal’s algorithm.**

**Code:**

*import*

java.util.\*;

*import*

java.util.stream.Collectors;

*public*

*class*

Graph

{

*private*

*List*

>

<

Edge

edges;

*private*

*Map*

<

Integer,

Integer>

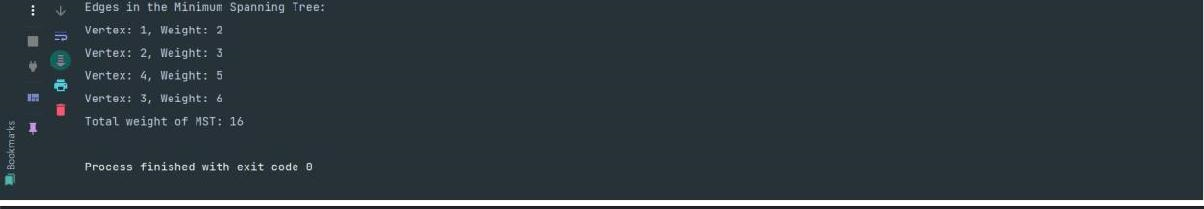
parent;

|  |
| --- |
| *private Map*<Integer, Integer> rank;  *public* Graph() { edges = *new* ArrayList<>(); parent = *new* HashMap<>();  rank = *new* HashMap<>();  }  *public void* addEdge(*int* u, *int* v, *int* weight) { edges.add(*new* Edge(u, v, weight)); parent.putIfAbsent(u, u); parent.putIfAbsent(v, v); rank.putIfAbsent(u, 0);  rank.putIfAbsent(v, 0);  }  *public List*<Edge> kruskalMST() { edges.sort(*Comparator*.*comparingInt*(edge -> edge.weight)); *List*<Edge> mst = *new* ArrayList<>(); *for* (Edge : edges) { *if* (find(edge.u) != find(edge.v)) { mst.add(edge); union(edge.u, edge.v);  }  }  *return* mst;  }  *private int* find(*int* u) { *if* (parent.get(u) != u) { parent.put(u, find(parent.get(u)));  }  *return* parent.get(u);  }  *private void* union(*int* u, *int* v) { |

|  |
| --- |
| *int* rootU = find(u); *int* rootV = find(v); *if* (rootU != rootV) { *if* (rank.get(rootU) > rank.get(rootV)) { parent.put(rootV, rootU);  } *else if* (rank.get(rootU) < rank.get(rootV)) { parent.put(rootU, rootV);  } *else* {  parent.put(rootV, rootU);  rank.put(rootU, rank.get(rootU) + 1);  }  }  }  *private static class* Edge { *int* u, v, weight;  Edge(*int* u, *int* v, *int* weight) { *this*.u = u;  *this*.v = v;  *this*.weight = weight;  }  }  *public static void* main(String[] args) { Graph = *new* Graph(); graph.addEdge(1, 2, 10); graph.addEdge(1, 3, 20); graph.addEdge(2, 3, 30); graph.addEdge(3, 4, 40); graph.addEdge(2, 4, 50);  *List*<Edge> mst = graph.kruskalMST();  mst.forEach(edge -> System.*out*.println(edge.u + " - " + edge.v + ": " + edge.weight)); |

|  |
| --- |
| }  }  **Output:**    **Lab Experiment 36:**  **Find MST Using Prim’s Algorithm: Write a function to find the Minimum Spanning Treeof a graph using Prim’s algorithm.**  **Code:**  *import* java.util.\*; *public class* Graph { *static class* Edge *implements Comparable*<Edge> { *int* vertex; *int* weight;  Edge(*int* vertex, *int* weight) { *this*.vertex = vertex;  *this*.weight = weight;  }  @Override  *public int* compareTo(Edge other) { *return this*.weight - other.weight;  }  }  *public static void* primMST(*int*[][] graph) { *int* vertices = graph.length;  *boolean*[] visited = *new boolean*[vertices]; PriorityQueue<Edge> pq = *new* PriorityQueue<>(); |

|  |
| --- |
| *List*<Edge> mst = *new* ArrayList<>(); pq.add(*new* Edge(0, 0));  *while* (!pq.isEmpty() && mst.size() < vertices - 1) { Edge current = pq.poll(); *if* (visited[current.vertex]) { *continue*;  }  visited[current.vertex] = *true*; *if* (current.weight != 0) { mst.add(current);  }  *for* (*int* neighbor = 0; neighbor < vertices; neighbor++) { *if* (!visited[neighbor] && graph[current.vertex][neighbor] !=  0) {  pq.add(*new* Edge(neighbor,  graph[current.vertex][neighbor]));  }  }  }  *int* totalWeight = 0;  System.*out*.println("Edges in the Minimum Spanning Tree:"); *for* (Edge : mst) {  System.*out*.println("Vertex: " + edge.vertex + ", Weight: " + edge.weight);  totalWeight += edge.weight;  }  System.*out*.println("Total weight of MST: " + totalWeight);  }  *public static void* main(String[] args) { *int*[][] graph = {  {0, 2, 0, 6, 0},  {2, 0, 3, 8, 5},  {0, 3, 0, 0, 7}, |



{6

,

8

,

0

,

0

,

9}

,

{0

,

5

,

7

,

9

,

0}

}

;

*primMST*

(

graph);

}

}

**Output:**

**Lab**

**Experiment**

**37:**

**Fibonacci**

**Sequence:**

**Write**

**a**

**function**

**to**

**compute**

**the**

**nth**

**Fibonacci**

**number using dynamic programming.**

**Code:**

*public*

*class*

FibonacciDP

{

*public*

*static*

*int*

fibonacci(

*int*

n)

{

*if*

(

n

<=

1)

*return*

n;

*int*

[]

dp

=

*new*

*int*

[

n

+

1]

;

dp[0] = 0;

dp[1]

=

1

;

*for*

(

*int*

i = 2; i <= n; i++) {

dp[i]

=

dp[i

-

1]

+

dp[i

-

2]

;

}

*return*

dp[n];

}

*public*

*static*

*void*

main(String[]

args)

{

*int*

n = 10;

System.

*out*

.println(

*fibonacci*

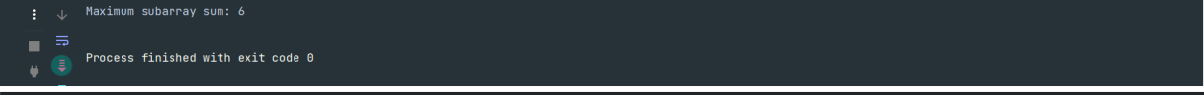
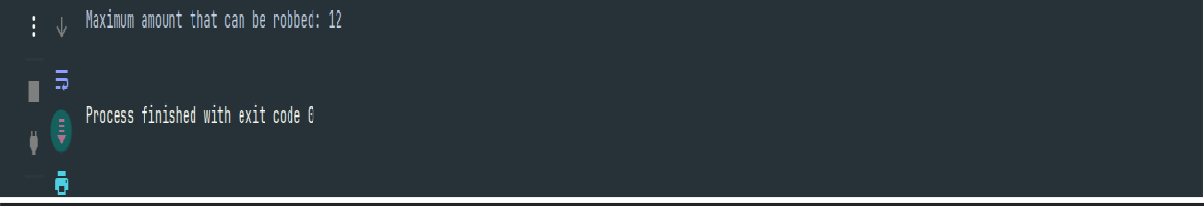
(

n));

|  |
| --- |
| }  }  **Output:**    **Lab Experiment 38:**  **Climbing Stairs: Write a function to determine how many distinct ways there are to climb a staircase with n steps if you can climb either 1 or 2 steps at a time.**  **Code:**  *public class* ClimbingStairs { *public int* climbStairs(*int* n) { *if* (n <= 1) *return* 1; *int* first = 1, second = 1; *for* (*int* i = 2; i <= n; i++) { *int* temp = first + second; first = second;  second = temp;  }  *return* second;  }  *public static void* main(String[] args) { ClimbingStairs cs = *new* ClimbingStairs(); *int* n = 5;  System.*out*.println("Number of ways to climb " + n + " steps: " + cs.climbStairs(n)); |

|  |
| --- |
| }  }  **Output:**    **Lab Experiment 39:**  **Min Cost Climbing Stairs: Write a function to determine the minimum cost to reach thetop of a staircase given a list of costs associated with each step. Code:**  *public class* ClimbingStairs { *public int* minCostClimbingStairs(*int*[] cost) { *int* n = cost.length;  *int* first = cost[0], second = cost[1];  *for* (*int* i = 2; i < n; i++) { *int* current = cost[i] + Math.*min*(first, second); first = second;  second = current;  }  *return* Math.*min*(first, second);  }  *public static void* main(String[] args) {  ClimbingStairs solution = *new* ClimbingStairs(); *int*[] cost = {10, 15, 20};  System.*out*.println("Minimum cost to climb the stairs: " + solution.minCostClimbingStairs(cost)); |

|  |
| --- |
| }  }  **Output:**    **Lab Experiment 40:**  **House Robber: Write a function to determine the maximum amount of money you can rob from a row of houses without robbing two adjacent houses.**  **Code:**  *public class* HouseRobber { *public int* rob(*int*[] nums) { *if* (nums.length == 0) *return* 0; *if* (nums.length == 1) *return* nums[0];  *int* first = 0, second = 0; *for* (*int* num : nums) { *int* temp = Math.*max*(second, first + num); first = second;  second = temp;  }  *return* second;  }  *public static void* main(String[] args) {  HouseRobber solution = *new* HouseRobber(); *int*[] nums = {2, 7, 9, 3, 1};  System.*out*.println("Maximum amount that can be robbed: " + solution.rob(nums));  }  } |



**Output:**

**Lab**

**Experiment**

**41:**

**Maximum**

**Subarray**

**Sum**

**(**

**Kadane’s**

**Algorithm):**

**Write**

**a**

**function**

**to**

**find the contiguous subarray with the maximum sum.**

**Code:**

*public*

*class*

MaximumSubarray

{

*public*

*int*

maxSubArray(

*int*

[]

nums)

{

*int*

maxSum

=

nums[0],

currentSum

=

nums[0];

*for*

(

*int*

i

=

1

;

i

<

nums.length;

i++)

{

currentSum

=

Math.

*max*

nums[i],

(

currentSum

+

nums[i]);

maxSum = Math.

*max*

(

maxSum, currentSum);

}

*return*

maxSum;

}

*public*

*static*

*void*

main(String[]

args)

{

MaximumSubarray

solution

=

*new*

MaximumSubarray();

*int*

[]

nums =

{

-

2

, 1,

-

3

,

4

,

-

1

,

2

,

1

,

-

,

5

;

4}

*// Example input*

System.

*out*

.println("Maximum subarray sum: " +

solution.maxSubArray(nums));

}

}

**Output:**

|  |
| --- |
| **Lab Experiment 42:**  **Activity Selection: Given a set of activities with start and end times, select the maximum number of activities that do not overlap.**  **Code:**  *import* java.util.Arrays; *import* java.util.*Comparator*; *public class* ActivitySelection { *public int* selectActivities(*int*[] start, *int*[] end) { *int* n = start.length; *int*[] result = *new int*[n]; *int*[][] activities = *new int*[n][2];  *for* (*int* i = 0; i < n; i++) { activities[i][0] = start[i];  activities[i][1] = end[i];  }  Arrays.*sort*(activities, *Comparator*.*comparingInt*(a -> a[1])); *int* count = 1;  *int* lastSelectedActivity = 0; *for* (*int* i = 1; i < n; i++) { *if* (activities[i][0] >= activities[lastSelectedActivity][1]) { count++;  lastSelectedActivity = i;  }  }  *return* count;  }  *public static void* main(String[] args) {  ActivitySelection solution = *new* ActivitySelection(); *int*[] start = {1, 3, 0, 5, 8, 5};  *int*[] end = {2, 4, 6, 7, 9, 9};  System.*out*.println("Maximum number of activities: " + |

|  |
| --- |
| solution.selectActivities(start, end));  }  }  **Output:**    **Lab Experiment 43:**  **Fractional Knapsack Problem: Given weights and values of items and the maximum capacity of a knapsack, determine the maximum value that can be obtained by including fractions of items.**  **Code:**  *import* java.util.Arrays; *class* Item { *int* value, weight;  Item(*int* value, *int* weight) { *this*.value = value;  *this*.weight = weight;  }  }  *public class* FractionalKnapsack { *public double* getMaxValue(*int* W, Item[] items) {  Arrays.*sort*(items, (a, b) -> (b.value \* a.weight) - (a.value \* b.weight));  *double* totalValue = 0.0; *for* (Item : items) { *if* (W == 0) *break*; *if* (item.weight <= W) {  W -= item.weight; |

|  |
| --- |
| totalValue += item.value;  } *else* {  totalValue += item.value \* ((*double*) W / item.weight); *break*;  }  }  *return* totalValue;  }  *public static void* main(String[] args) {  FractionalKnapsack solution = *new* FractionalKnapsack();  Item[] items = { *new* Item(60, 10), *new* Item(100, 20),  *new* Item(120, 30)  };  *int* W = 50;  System.*out*.println("Maximum value in Knapsack = " + solution.getMaxValue(W, items));  }  }  **Output:**    **Lab Experiment 44:**  **Huffman Coding: Given a set of characters and their frequencies, construct the HuffmanTree to encode the characters.**  **Code:**  *import* java.util.PriorityQueue; *class* Node { |

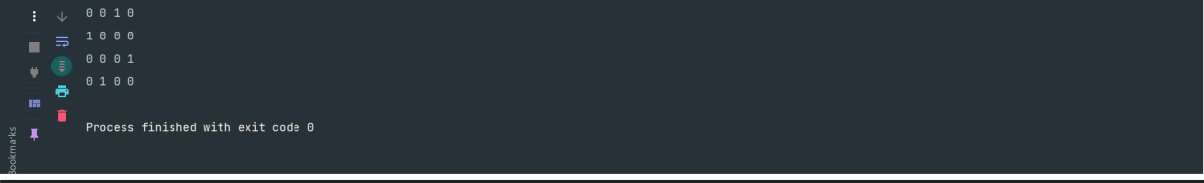
|  |
| --- |
| *char* ch; *int* freq;  Node left, right;  Node(*char* ch, *int* freq) { *this*.ch = ch; *this*.freq = freq;  *this*.left = *this*.right = *null*;  }  }  *public class* HuffmanCoding { *public void* buildHuffmanTree(*char*[] chars, *int*[] freqs) {  PriorityQueue<Node> pq = *new* PriorityQueue<>((a, b) -> a.freq - b.freq);  *for* (*int* i = 0; i < chars.length; i++) { pq.add(*new* Node(chars[i], freqs[i]));  }  *while* (pq.size() > 1) {  Node left = pq.poll();  Node right = pq.poll();  Node = *new* Node('\0', left.freq + right.freq); node.left = left; node.right = right;  pq.add(node);  }  printCodes(pq.peek(), "");  }  *private void* printCodes(Node root, String code) { *if* (root == *null*) *return*; *if* (root.ch != '\0') {  System.*out*.println(root.ch + ": " + code);  }  printCodes(root.left, code + "0"); |

|  |
| --- |
| printCodes(root.right, code + "1");  }  *public static void* main(String[] args) {  HuffmanCoding huffman = *new* HuffmanCoding();  *char*[] chars = {'a', 'b', 'c', 'd', 'e', 'f'};  *int*[] freqs = {5, 9, 12, 13, 16, 45};  huffman.buildHuffmanTree(chars, freqs);  }  }  **Output:**    **Lab Experiment 45:**  **Job Sequencing Problem: Given a set of jobs, each with a deadline and profit, maximize the total profit by scheduling the jobs to be done before their deadlines.**  **Code:**  *import* java.util.\*; *class* Job { *int* id, deadline, profit;  Job(*int* id, *int* deadline, *int* profit) { *this*.id = id;  *this*.deadline = deadline; *this*.profit = profit;  }  }  *public class* JobSequencing { |

|  |
| --- |
| *static void* jobScheduling(Job arr[], *int* n) { Arrays.*sort*(arr, (a, b) -> b.profit - a.profit);  *boolean*[] slots = *new boolean*[n];  *int*[] result = *new int*[n]; *for* (*int* i = 0; i < n; i++) { *for* (*int* j = Math.*min*(n, arr[i].deadline) - 1; j >= 0; j--) { *if* (!slots[j]) { result[j] = arr[i].id; slots[j] = *true*;  *break*;  }  }  }  *for* (*int* i = 0; i < n; i++) { *if* (slots[i]) {  System.*out*.print(result[i] + " ");  }  }  }  *public static void* main(String[] args) {  Job arr[] = { *new* Job(1, 4, 20), *new* Job(2, 1, 10), *new* Job(3, 1, 40),  *new* Job(4, 1, 30)  };  *int* n = arr.length;  *jobScheduling*(arr, n);  }  }  **Output:** |

|  |
| --- |
| **Lab Experiment 46:**  **Minimum Number of Coins: Given different denominations of coins and an amount, findthe minimum number of coins needed to make up that amount.**  **Code:**  *import* java.util.\*; *public class* CoinChange { *static int* minCoins(*int*[] coins, *int* amount) { *int* n = coins.length;  *int*[] dp = *new int*[amount + 1]; Arrays.*fill*(dp, amount + 1); dp[0] = 0;  *for* (*int* i = 1; i <= amount; i++) { *for* (*int* coin : coins) { *if* (coin <= i) { dp[i] = Math.*min*(dp[i], dp[i - coin] + 1);  }  }  }  *return* dp[amount] > amount ? -1 : dp[amount];  }  *public static void* main(String[] args) { *int*[] coins = {1, 2, 5}; *int* amount = 11;  System.*out*.println(*minCoins*(coins, amount));  }  }  **Output:** |

|  |
| --- |
| **Lab Experiment 47:**  **N-Queens Problem: Place N queens on an N×N chessboard so that no two queens threaten each other.**  **Code:**  *public class* NQueens { *static int N*;  *static boolean* isSafe(*int* board[][], *int* row, *int* col) { *for* (*int* i = 0; i < col; i++) { *if* (board[row][i] == 1) {  *return false*;  }  }  *for* (*int* i = row, j = col; i >= 0 && j >= 0; i--, j--) { *if* (board[i][j] == 1) { *return false*;  }  }  *for* (*int* i = row, j = col; j >= 0 && i < *N*; i++, j--) { *if* (board[i][j] == 1) { *return false*;  }  }  *return true*;  }  *static boolean* solveNQueens(*int* board[][], *int* col) { *if* (col >= *N*) { *return true*;  }  *for* (*int* i = 0; i < *N*; i++) { *if* (*isSafe*(board, i, col)) { board[i][col] = 1; |



*if*

(

*solveNQueens*

(

board,

col

+

1))

{

*return*

*true*

;

}

board[i][col]

=

0

;

}

}

*return*

*false*

;

}

*static*

*void*

printSolution(

*int*

board[][])

{

*for*

(

*int*

i

=

0

;

i

<

*N*

;

i++)

{

*for*

(

*int*

j = 0; j <

*N*

; j++) {

System.

*out*

.print(board[i][j]

+

"

");

}

System.

*out*

.println();

}

}

*public*

*static*

*void*

main(String[]

args)

{

*N*

=

4;

*// Change N to any number*

*int*

board[][] =

*new int*

[

*N*

][

*N*

]

;

*if*

(!

*solveNQueens*

(

board, 0)) {

System.

*out*

.println("Solution

does

not

exist");

}

*else*

{

*printSolution*

(

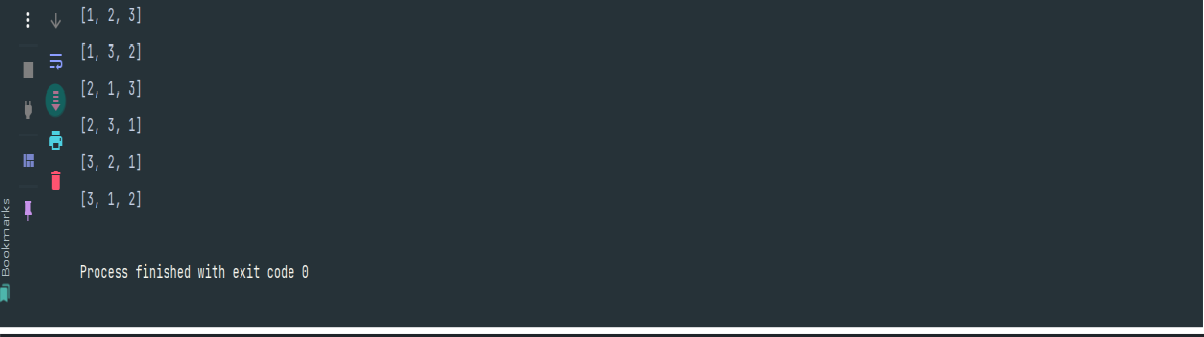
board);

}

}

}

**Output:**



**Lab**

**Experiment**

**48:**

**Permutations:**

**Generate**

**all**

**possible**

**permutations**

**of**

**a**

**given**

**list**

**of**

**numbers or characters.**

**Code:**

*import*

java.util.\*;

*public*

*class*

Permutations

{

*static*

*void*

generatePermutations(

*List*

<

Integer> nums,

*int*

index)

{

*if*

(

index == nums.size()) {

System.

*out*

.println(nums);

*return*

;

}

*for*

(

*int*

i = index; i < nums.size(); i++) {

Collections.

*swap*

(

nums, i, index);

*generatePermutations*

(

nums,

index

+

1)

;

Collections.

*swap*

(

nums, i, index);

}

}

*public*

*static*

*void*

main(String[]

args)

{

*List*

<

Integer

>

nums

=

Arrays.

*asList*

(1

,

2

,

3)

;

*generatePermutations*

(

nums,

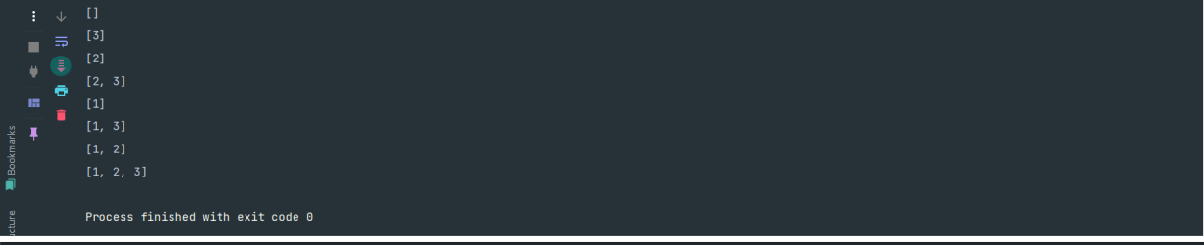
0)

;

}

}

**Output:**



**Lab**

**Experiment**

**49:**

**Subsets:**

**Generate**

**all**

**possible**

**subsets**

**of**

**a**

**given**

**set**

**of**

**numbers.**

**Code:**

*import*

java.util.\*;

*public*

*class*

Subsets

{

*static*

*void*

generateSubsets(

*List*

<

Integer

>

nums,

*int*

index,

*List*

<

Integer

>

currentSubset)

{

*if*

(

index == nums.size()) {

System.

*out*

.println(currentSubset);

*return*

;

}

*generateSubsets*

(

nums,

index

+

1

,

currentSubset);

currentSubset.add(nums.get(index));

*generateSubsets*

(

nums,

index

+

1

,

currentSubset);

currentSubset.remove(currentSubset.size()

-

1)

;

}

*public static void*

main(String[] args) {

*List*

<

Integer> nums = Arrays.

*asList*

(1

, 2, 3);

*generateSubsets*

(

nums,

0

,

*new*

ArrayList<>());

}

}

**Output:**

**80**

**|**

P

a

g

e