**Data Structures**

**(CSL 209)**

**Lab Workbook**



Faculty name: Student name: MohitMudgal

Roll No.:22csu471

Semester:3

Group:DS4

**Department of Computer Science and Engineering**

**The NorthCap University, Gurugram- 122017, India**

**Session 2023-24**

**INDEX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.No** | **Experiment** | **Date of Experiment** | **Date of Submission** | **Student Sign** | **CO Covered** | **Faculty**  **Sign** |
| **1** | Create an array of integer with size n. Return the difference between the largest and the smallest value inside that array. |  |  |  |  |  |
| 2 | 1. Write a program that initializes an array with ten random integers and then prints four lines of output, containing:  * Every element at an even index * Every odd element * All elements in reverse order   Only the first and last element |  |  |  |  |  |
| 3 | 1. Write a program to read numbers in an integer array of size 5 and display the following:  * Sum of all the elements * Sum of alternate elements in the array * Second highest element in the array |  |  |  |  |  |
| 4 | Write a program to create a singly linked list of n nodes and perform:  • Insertion   * At the beginning * At the end * At a specific location   • Deletion   * At the beginning * At the end   At a specific location |  |  |  |  |  |
| 5 | Write a program to create a doubly linked list of n nodes and perform:  • Insertion   * At the beginning * At the end * At a specific location   • Deletion   * At the beginning * At the end   At a specific location |  |  |  |  |  |
| 6 | Write a program to create a circular linked list of n nodes and perform:  • Insertion   * At the beginning * At the end * At a specific location   • Deletion   * At the beginning * At the end   At a specific location |  |  |  |  |  |
| 7 | Write a program to implement stack using arrays and linked lists. |  |  |  |  |  |
| 8 | Write a program to reverse a sentence using stack. |  |  |  |  |  |
| 9 | Write a program to check for balanced parenthesis in a given expression. |  |  |  |  |  |
| 10 | Write a program to convert infix expression to prefix and postfix expression. |  |  |  |  |  |
| 11 | Write a program to implement Tower of Hanoi using stacks |  |  |  |  |  |
| 12 | Write a program to implement Linear Queue using Array and Linked Lists. |  |  |  |  |  |
| 13 | Write a program to implement Circular Queue using Array and Linked Lists. |  |  |  |  |  |
| 14 | Write a program to implement Doubly Ended Queue using Array and Linked Lists. |  |  |  |  |  |
| 15 | Write a Program to implement Binary Search Tree operations. |  |  |  |  |  |
| 16 | Write a program to implement Bubble Sort, Selection Sort, Quick Sort, Merge Sort and Insertion Sort algorithm. |  |  |  |  |  |

**EXPERIMENT NO. 1**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure array and its basic operations |
| **Outcome:**  The students will be able to implement and use arrays for solving various problems |
| **Problem Statement:**  Create an array of integer with size n. Return the difference between the largest and the smallest value inside that array. |
| **Background Study:**  An Array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array. |
| **Algorithm (Student Work Area):**   1. **Define a function findDifference that takes an array of integers as input.** 2. **Initialize min and max variables with the first element of the array.** 3. **Iterate through the array starting from the second element.**    * **If the current element is less than min, update min.**    * **If the current element is greater than max, update max.** 4. **After the loop, return max - min, which gives the difference between the largest and smallest values.** |
| **Code (Student Work Area):**  public class ArrayDifference {  public static int findDifference(int[] arr) {  int min = arr[0];  int max = arr[0];    for (int i = 1; i < arr.length; i++) {  if (arr[i] < min) {  min = arr[i];  }  if (arr[i] > max) {  max = arr[i];  }  }    return max - min;  }  public static void main(String[] args) {  int[] arr = {5, 8, 3, 2, 9, 1, 7, 4, 6}; // Fixed array    int difference = findDifference(arr);  System.out.println("Array: " + java.util.Arrays.toString(arr));  System.out.println("Difference between max and min: " + difference);  }  } |
| **Question Bank:**  1. \*\*What is Data Structure?\*\*  - A data structure is a way of organizing and storing data so that it can be used efficiently. It defines the relationship between data and operations that can be performed on the data. Common data structures include arrays, linked lists, stacks, queues, trees, graphs, etc.  2. \*\*Why Array is called as Linear Data Structure?\*\*  - An array is called a linear data structure because it stores elements in a sequential manner, one after the other. Each element can be accessed directly by its index, which is a linear value (0, 1, 2, ... n-1).  3. \*\*What type of Indexing is used in Java?\*\*  - In Java, indexing is 0-based, which means the first element of an array is accessed using index 0, the second element with index 1, and so on.  4. \*\*How to find the missing number in an integer array of 1 to 100?\*\*  - One way to find the missing number in an integer array from 1 to 100 is to calculate the sum of all numbers from 1 to 100 using the arithmetic sum formula (n \* (n+1)) / 2. Then, find the sum of the elements in the array. The missing number is the difference between the expected sum and the actual sum.  5. \*\*How to find the second-highest value in a numeric array?\*\*  - One way to find the second-highest value in a numeric array is to iterate through the array, keeping track of the highest and second-highest values. Initialize two variables, `max` and `secondMax`, with very small values. Then, traverse the array, updating `max` and `secondMax` accordingly.  6. \*\*How to swap the first and last elements of an array?\*\*  - To swap the first and last elements of an array, you can use a temporary variable to hold one of the values, then perform the swap. Here is an example in Java:  ```java  int[] arr = {1, 2, 3, 4, 5};  int temp = arr[0];  arr[0] = arr[arr.length - 1];  arr[arr.length - 1] = temp;  ```  7. \*\*Write a Java Program to check if an Array contains a specific value (Linear Search)?\*\*  - Here is a Java program to perform a linear search to check if an array contains a specific value:  ```java  public class LinearSearch {  public static boolean containsValue(int[] arr, int target) {  for (int element : arr) {  if (element == target) {  return true;  }  }  return false;  }  public static void main(String[] args) {  int[] arr = {1, 3, 5, 7, 9};  int target = 5;  boolean contains = containsValue(arr, target);  if (contains) {  System.out.println("The array contains " + target);  } else {  System.out.println("The array does not contain " + target);  }  }  }  ```  - This program defines a method `containsValue` which performs a linear search to check if the array contains the specified target value. The `main` method demonstrates its usage. |

**EXPERIMENT NO. 2**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure array and its basic operations |
| **Outcome:**  The students will be able to implement and use arrays for solving various problems |
| **Problem Statement:**   1. Write a program that initializes an array with ten random integers and then prints four lines of output, containing:  * Every element at an even index * Every odd element * All elements in reverse order * Only the first and last element |
| **Background Study:**  An Array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array. |
| **Algorithm (Student Work area):**  1. \*\*Initialize an array with ten fixed integers.\*\*    2. \*\*Print every element at an even index:\*\*  - Iterate through the array with a loop variable `i` starting from 0.  - Inside the loop, check if `i` is even (i.e., `i % 2 == 0`).  - If `i` is even, print the element at index `i`.  3. \*\*Print every odd element:\*\*  - Iterate through the array with a loop variable `i` starting from 0.  - Inside the loop, check if the element at index `i` is odd (i.e., `arr[i] % 2 != 0`).  - If the element is odd, print it.  4. \*\*Print all elements in reverse order:\*\*  - Iterate through the array with a loop variable `i` starting from the last index (`arr.length - 1`) to 0.  - Print the element at index `i`.  5. \*\*Print only the first and last elements:\*\*  - Print the first element at index 0.  - Print the last element at index `arr.length - 1`.  The Java program provided in the previous message implements this algorithm. It initializes an array, then uses loops to perform the specified operations. |
| **Code (Student Work Area):**  public class ArrayOperations {  public static void main(String[] args) {  int[] arr = {12, 7, 32, 14, 15, 7, 5, 9, 3, 2};  // Every element at an even index  System.out.println("Elements at even indices:");  for (int i = 0; i < arr.length; i += 2) {  System.out.print(arr[i] + " ");  }  System.out.println();  // Every odd element  System.out.println("Odd elements:");  for (int i = 0; i < arr.length; i++) {  if (arr[i] % 2 != 0) {  System.out.print(arr[i] + " ");  }  }  System.out.println();  // All elements in reverse order  System.out.println("Elements in reverse order:");  for (int i = arr.length - 1; i >= 0; i--) {  System.out.print(arr[i] + " ");  }  System.out.println();  // Only the first and last element  System.out.println("First and last elements:");  System.out.println(arr[0] + " " + arr[arr.length - 1]);  }  } |
| **Question Bank:**  1. \*\*Segregate 0s and 1s in an array:\*\*  Algorithm:  - Initialize two pointers `left` and `right` at the start and end of the array respectively.  - Loop until `left` is less than `right`.  - Move `left` towards the right until you find a 1.  - Move `right` towards the left until you find a 0.  - Swap the elements at `left` and `right`.  - Continue these steps until `left` is less than `right`.  2. \*\*Reverse the array elements:\*\*  Algorithm:  - Initialize two pointers `start` and `end` at the beginning and end of the array respectively.  - Swap the elements at `start` and `end`.  - Increment `start` and decrement `end`.  - Continue these steps until `start` is less than `end`.  3. \*\*Find the index of an array element:\*\*  Algorithm:  - Iterate through the array using a loop.  - For each element, compare it with the target value.  - If they match, return the current index.  - If the loop completes without finding a match, return a value (e.g., -1) to indicate the element was not found.  4. \*\*Remove a specific element from an array:\*\*  Algorithm:  - Identify the index of the element to be removed.  - Shift all elements after that index one position to the left to fill the gap.  - Update the length of the array (if it's a dynamic array) to reflect the removal.  5. \*\*Insert an element at a specific position in an array:\*\*  Algorithm:  - Identify the index where you want to insert the element.  - If the array is dynamic, ensure it has enough space to accommodate the new element.  - Shift all elements from the target index to the right to make space for the new element.  - Insert the new element at the target index. |

**EXPERIMENT NO. 3**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure array and its basic operations |
| **Outcome:**  The students will be able to implement and use arrays for solving various problems |
| **Problem Statement:**   1. Write a program to read numbers in an integer array of size 5 and display the following:  * Sum of all the elements * Sum of alternate elements in the array * Second highest element in the array |
| **Background Study:**  An Array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array. |
| **Algorithm (Student Work Area):**  1. \*\*Initialize the Array\*\*: Create an integer array of size 5.  2. \*\*Read Input\*\*:  - Prompt the user to enter 5 numbers.  - Store each number in the array.  3. \*\*Calculate Sum of All Elements\*\*:  - Initialize a variable `sum` to zero.  - Iterate through the array and add each element to `sum`.  - At the end of the loop, `sum` will hold the sum of all elements.  4. \*\*Calculate Sum of Alternate Elements\*\*:  - Initialize a variable `alternateSum` to zero.  - Use a loop to iterate through the array with a step size of 2 (0, 2, 4).  - Add each element at these positions to `alternateSum`.  5. \*\*Find Second Highest Element\*\*:  - Initialize two variables `max1` and `max2` to the smallest possible integer value (to handle negative numbers).  - Iterate through the array.  - For each element:  - If it's greater than `max1`, set `max2 = max1` and `max1 = element`.  - Else if it's greater than `max2` but less than `max1`, set `max2 = element`.  6. \*\*Display Results\*\*:  - Print out the calculated values:  - Sum of all elements (`sum`).  - Sum of alternate elements (`alternateSum`).  - Second highest element (`max2`).  Note: Make sure to handle input validation if needed (e.g., check if the user enters valid integers).  This algorithm outlines the steps needed to perform the operations you specified in the problem. You can use this as a guide to implement the program in any programming language. |
| **Code (Student Work Area):**  import java.util.Scanner;  import java.util.Arrays;  public class Main {  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  int[] numbers = new int[5];  System.out.println("Enter 5 numbers:");  for (int i = 0; i < 5; i++) {  numbers[i] = scanner.nextInt();  }  // Calculate sum of all elements  int sum = Arrays.stream(numbers).sum();  // Calculate sum of alternate elements  int alternateSum = 0;  for (int i = 0; i < 5; i += 2) {  alternateSum += numbers[i];  }  // Find the second highest element  int max1 = Integer.MIN\_VALUE;  int max2 = Integer.MIN\_VALUE;  for (int num : numbers) {  if (num > max1) {  max2 = max1;  max1 = num;  } else if (num > max2 && num < max1) {  max2 = num;  }  }  System.out.println("Sum of all elements: " + sum);  System.out.println("Sum of alternate elements: " + alternateSum);  System.out.println("Second highest element: " + max2);  }  } |
| **Question Bank:**   1. How we can count occurrence of a given number in the array and its frequency.   public class Main {  public static void main(String[] args) {  int[] arr = {1, 2, 3, 4, 1, 2, 1, 2, 5};  int target = 1;  int frequency = countOccurrence(arr, target);  System.out.println("Frequency of " + target + ": " + frequency);  }  public static int countOccurrence(int[] arr, int target) {  int count = 0;  for (int num : arr) {  if (num == target) {  count++;  }  }  return count;  }  }  2. How we can print the following in 2-D integer array with each element of maximum 2 digits  a) Elements of the entered array.  b) Elements of the array after each element is multiplied by 2 if it is an odd number.  public class Main {  public static void print2DArray(int[][] arr) {  for (int[] row : arr) {  for (int element : row) {  System.out.printf("%2d ", element);  }  System.out.println();  }  }  public static void main(String[] args) {  int[][] arr = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  print2DArray(arr);  }  }  3. Given an array of integers, return the number of times that two 6's are next to each other in the array. Also count instances where the second element is 7  public class Main {  public static int countOccurrences(int[] arr) {  int count = 0;  for (int i = 0; i < arr.length - 1; i++) {  if (arr[i] == 6 && arr[i + 1] == 6) {  count++;  } else if (i < arr.length - 1 && arr[i + 1] == 7) {  count++;  }  }  return count;  }  public static void main(String[] args) {  int[] arr = {6, 6, 7, 6, 8, 7, 7};  int occurrences = countOccurrences(arr);  System.out.println("Occurrences: " + occurrences);  }  }  4. Write a method called swapPairs() that accepts an array of integers and swaps the elements at adjacent indexes. That is, elements 0 and 1 are swapped, elements 2 and 3 are swapped, and so on. If the array has an odd length, the final element should be left unmodified. For example, the list {10, 20, 30, 40, 50} should become {20, 10, 40, 30, 50} after a call to your method.  public class Main {  public static void swapPairs(int[] arr) {  for (int i = 0; i < arr.length - 1; i += 2) {  int temp = arr[i];  arr[i] = arr[i + 1];  arr[i + 1] = temp;  }  }  public static void main(String[] args) {  int[] arr = {10, 20, 30, 40, 50};  swapPairs(arr);  for (int num : arr) {  System.out.print(num + " ");  }  }  }   1. Write a method called *median*() that accepts an array of integers as its argument and returns the median of the numbers in the array. The median is the number that will appear in the middle if you arrange the elements in order.   **import java.util.Arrays;**  **public class Main {**  **public static double median(int[] arr) {**  **Arrays.sort(arr);**  **int n = arr.length;**  **if (n % 2 != 0) {**  **return arr[n / 2];**  **} else {**  **return (arr[n / 2 - 1] + arr[n / 2]) / 2.0;**  **}**  **}**  **public static void main(String[] args) {**  **int[] arr = {5, 2, 1, 8, 3, 4};**  **double result = median(arr);**  **System.out.println("Median: " + result);**  **}**  **}** |

**EXPERIMENT NO. 4**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Linked List and its basic operations |
| **Outcome:**  The students will be able to implement and use singly linked list for solving various problems |
| **Problem Statement:**  Write a program to create a singly linked list of n nodes and perform:  • Insertion   * At the beginning * At the end * At a specific location   • Deletion   * At the beginning * At the end * At a specific location |
| **Background Study:** **Insertion Operation** Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.  Linked List Insertion  Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −  NewNode.next −> RightNode;  It should look like this −  Linked List Insertion  Now, the next node at the left should point to the new node.  LeftNode.next −> NewNode;  Linked List Insertion  This will put the new node in the middle of the two. The new list should look like this −  Linked List Insertion  Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL. **Deletion Operation** Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.  Linked List Deletion  The left (previous) node of the target node now should point to the next node of the target node −  LeftNode.next −> TargetNode.next;  Linked List Deletion  This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.  TargetNode.next −> NULL;  Linked List Deletion  We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.  Linked List Deletion |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):**  class Node {  int data;  Node next;  Node(int data) {  this.data = data;  this.next = null;  }  }  class LinkedList {  Node head;  // Insertion at the beginning  public void insertAtBeginning(int data) {  Node newNode = new Node(data);  newNode.next = head;  head = newNode;  }  // Insertion 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;  }  // Insertion at a specific location  public void insertAtLocation(int data, int position) {  Node newNode = new Node(data);  if (position == 0) {  newNode.next = head;  head = newNode;  return;  }  Node temp = head;  for (int i = 0; i < position - 1 && temp != null; i++) {  temp = temp.next;  }  if (temp == null) {  System.out.println("Invalid position");  return;  }  newNode.next = temp.next;  temp.next = newNode;  }  // Deletion at the beginning  public void deleteAtBeginning() {  if (head == null) {  System.out.println("List is empty");  return;  }  head = head.next;  }  // Deletion at the 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;  }  // Deletion at a specific location  public void deleteAtLocation(int position) {  if (head == null) {  System.out.println("List is empty");  return;  }  if (position == 0) {  head = head.next;  return;  }  Node temp = head;  for (int i = 0; i < position - 1 && temp != null; i++) {  temp = temp.next;  }  if (temp == null || temp.next == null) {  System.out.println("Invalid position");  return;  }  temp.next = temp.next.next;  }  public void display() {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " -> ");  temp = temp.next;  }  System.out.println("null");  }  }  public class Main {  public static void main(String[] args) {  LinkedList list = new LinkedList();  list.insertAtEnd(10);  list.insertAtBeginning(5);  list.insertAtEnd(15);  list.insertAtLocation(8, 2);  System.out.println("Original list:");  list.display();  list.deleteAtBeginning();  System.out.println("List after deleting at the beginning:");  list.display();  list.deleteAtEnd();  System.out.println("List after deleting at the end:");  list.display();  list.deleteAtLocation(1);  System.out.println("List after deleting at location 1:");  list.display();  }  } |
| **Output – Screenshots (Student Work Area):**  **Original list:**  **5 -> 10 -> 8 -> 15 -> null**  **List after deleting at the beginning:**  **10 -> 8 -> 15 -> null**  **List after deleting at the end:**  **10 -> 8 -> null**  **List after deleting at location 1:**  **10 -> null** |
| **Question Bank:**  A linked list and an array are two fundamental data structures used to store collections of elements. Here are some key differences between them:  1. \*\*Memory Allocation\*\*:  - \*\*Linked List\*\*: In a linked list, each element (node) contains a value and a reference (link) to the next element in the sequence. Nodes can be scattered in memory.  - \*\*Array\*\*: Elements in an array are stored in contiguous memory locations. Each element can be accessed directly by its index.  2. \*\*Size Flexibility\*\*:  - \*\*Linked List\*\*: The size of a linked list can change dynamically by allocating or deallocating memory.  - \*\*Array\*\*: The size of an array is typically fixed when it is created. In some languages, dynamic arrays or resizable arrays are available.  3. \*\*Insertion and Deletion\*\*:  - \*\*Linked List\*\*: Insertion and deletion operations can be performed efficiently, especially at the beginning or end of the list. This is because no shifting of elements is required.  - \*\*Array\*\*: Insertion and deletion operations, especially in the middle of an array, can be less efficient, as elements need to be shifted.  4. \*\*Random Access\*\*:  - \*\*Linked List\*\*: Accessing an element in a linked list requires traversing the list from the head to the desired position, which can be slower for random access.  - \*\*Array\*\*: Accessing an element in an array is fast and can be done in constant time using the index.  Regarding the set of operations on a singly linked list:  1. \*\*Swapping the First and Last Node\*\*:  - To swap the first and last nodes, you need to update the `next` pointers of the nodes involved.  2. \*\*Pairwise Swap Elements\*\*:  - This operation involves swapping elements in pairs. You will need to adjust the `next` pointers of the nodes accordingly.  3. \*\*Get Location of First and Last Occurrence\*\*:  - To get the location of the first and last occurrence of an element, you'll need to traverse the list while keeping track of the positions.  4. \*\*Remove Duplicates\*\*:  - To remove duplicates from an unsorted linked list, you can use a hash set or nested loops to compare elements and remove duplicates.  5. \*\*Delete Alternate Nodes\*\*:  - This operation involves skipping every alternate node while traversing the list and adjusting the `next` pointers.   * If you'd like, I can provide sample Java code snippets for each of these operations. Would you like me to do that? |

**EXPERIMENT NO. 5**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Linked List and its basic operations |
| **Outcome:**  The students will be able to implement and use doubly linked list for solving various problems |
| **Problem Statement:**  Write a program to create a doubly linked list of n nodes and perform:  • Insertion   * At the beginning * At the end * At a specific location   • Deletion   * At the beginning * At the end * At a specific location |
| **Background Study:**  A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list. **Insertion Operation**A node can be added in three ways  **1)**At the front of the DLL  **2)** After a given node.  **3)** At the end of the DLL  **1) Add a node at the front:**dll_add_front **2) Add a node after a given node.:**  dll_add_middle **3) Add a node at the end:** dll_add_end **Deletion Operation** The deletion of a node in a doubly-linked list can be divided into three main categories:  Suppose we have a double-linked list with elements **1**, **2**, and **3**.  Original doubly linked list 1. Delete the First Node of Doubly Linked List **Reset value node after the del\_node (i.e. node two)**  Reorganize the pointers  ***Reorganize the pointers***  Finally, free the memory of del\_node. And, the linked will look like this  Final list  ***Final list*** 2. Deletion of the Inner Node If del\_node is an inner node (second node), we must have to reset the value of next and prev of the nodes before and after the del\_node.  **For the node before the del\_node (i.e. first node)**  Assign the value of next of del\_node to the next of the first node.  **For the node after the del\_node (i.e. third node)**  Assign the value of prev of del\_node to the prev of the third node.  Reorganize the pointers  ***Reorganize the pointers***  Finally, we will free the memory of del\_node. And, the final doubly linked list looks like this.  Final list  ***Final list*** 3. Delete the Last Node of Doubly Linked List In this case, we are deleting the last node with value **3** of the doubly linked list.  Here, we can simply delete the del\_node and make the next of node before del\_node point to NULL.  Reorganize the pointers  ***Reorganize the pointers***  The final doubly linked list looks like this.  Final list  ***Final list*** |
| **Algorithm (Student Work Area):**  Node Class:   1. Create a class **Node** with integer data, and references **next** and **prev** to the next and previous nodes respectively. 2. Initialize **next** and **prev** as **null** in the constructor.   DoublyLinkedList Class:   1. Create a class **DoublyLinkedList** with **head** and **tail** as pointers to the first and last nodes respectively.   Insertion at the Beginning (**insertAtBeginning**):   1. Create a new node **newNode** with the given data. 2. If the list is empty (**head** is **null**), set both **head** and **tail** to **newNode**. 3. If the list is not empty, set **newNode.next** to the current **head**. 4. If the list is not empty, set **head.prev** to **newNode**. 5. Set **head** to **newNode**.   Insertion at the End (**insertAtEnd**):   1. Create a new node **newNode** with the given data. 2. If the list is empty (**head** is **null**), set both **head** and **tail** to **newNode**. 3. If the list is not empty, set **newNode.prev** to the current **tail**. 4. If the list is not empty, set **tail.next** to **newNode**. 5. Set **tail** to **newNode**.   Insertion at a Specific Location (**insertAtLocation**):   1. Create a new node **newNode** with the given data. 2. If **position** is **0**, call **insertAtBeginning(data)**. 3. Otherwise, traverse the list to the node at position **position - 1**. 4. Insert **newNode** between the node at **position - 1** and its **next**. 5. Update the **prev** and **next** references accordingly.   Deletion at the Beginning (**deleteAtBeginning**):   1. If the list is empty (**head** is **null**), print a message and return. 2. If the list is not empty, set **head** to **head.next**. 3. If **head** is not **null**, set **head.prev** to **null**. 4. If **head** is **null**, also set **tail** to **null**.   Deletion at the End (**deleteAtEnd**):   1. If the list is empty (**tail** is **null**), print a message and return. 2. If the list is not empty, set **tail** to **tail.prev**. 3. If **tail** is not **null**, set **tail.next** to **null**. 4. If **tail** is **null**, also set **head** to **null**.   Deletion at a Specific Location (**deleteAtLocation**):   1. If **position** is **0**, call **deleteAtBeginning()**. 2. Otherwise, traverse the list to the node at position **position**. 3. Update the **prev.next** of the node before **position**. 4. Update the **next.prev** of the node after **position**. 5. If **position** is the last node, update **tail**. |
| **Code (Student Work Area):**  class Node {  int data;  Node next;  Node prev;  Node(int data) {  this.data = data;  this.next = null;  this.prev = null;  }  }  class DoublyLinkedList {  Node head;  Node tail;  // Insertion at the beginning  public void insertAtBeginning(int data) {  Node newNode = new Node(data);  if (head == null) {  head = tail = newNode;  } else {  newNode.next = head;  head.prev = newNode;  head = newNode;  }  }  // Insertion at the end  public void insertAtEnd(int data) {  Node newNode = new Node(data);  if (head == null) {  head = tail = newNode;  } else {  newNode.prev = tail;  tail.next = newNode;  tail = newNode;  }  }  // Insertion at a specific location  public void insertAtLocation(int data, int position) {  Node newNode = new Node(data);  if (position == 0) {  insertAtBeginning(data);  return;  }  Node temp = head;  for (int i = 0; i < position - 1 && temp != null; i++) {  temp = temp.next;  }  if (temp == null) {  System.out.println("Invalid position");  return;  }  newNode.next = temp.next;  newNode.prev = temp;  if (temp.next != null) {  temp.next.prev = newNode;  }  temp.next = newNode;  if (temp == tail) {  tail = newNode;  }  }  // Deletion at the beginning  public void deleteAtBeginning() {  if (head == null) {  System.out.println("List is empty");  return;  }  head = head.next;  if (head != null) {  head.prev = null;  } else {  tail = null;  }  }  // Deletion at the end  public void deleteAtEnd() {  if (tail == null) {  System.out.println("List is empty");  return;  }  tail = tail.prev;  if (tail != null) {  tail.next = null;  } else {  head = null;  }  }  // Deletion at a specific location  public void deleteAtLocation(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 && temp != null; i++) {  temp = temp.next;  }  if (temp == null) {  System.out.println("Invalid position");  return;  }  if (temp == tail) {  tail = temp.prev;  } else {  temp.next.prev = temp.prev;  }  if (temp == head) {  head = temp.next;  } else {  temp.prev.next = temp.next;  }  }  public void display() {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " <-> ");  temp = temp.next;  }  System.out.println("null");  }  }  public class Main {  public static void main(String[] args) {  DoublyLinkedList list = new DoublyLinkedList();  list.insertAtEnd(10);  list.insertAtBeginning(5);  list.insertAtEnd(15);  list.insertAtLocation(8, 2);  System.out.println("Original list:");  list.display();  list.deleteAtBeginning();  System.out.println("List after deleting at the beginning:");  list.display();  list.deleteAtEnd();  System.out.println("List after deleting at the end:");  list.display();  list.deleteAtLocation(1);  System.out.println("List after deleting at location 1:");  list.display();  }  } |
| **Output – Screenshots (Student Work Area):**  **Original list:**  **5 <-> 10 <-> 8 <-> 15 <-> null**  **List after deleting at the beginning:**  **10 <-> 8 <-> 15 <-> null**  **List after deleting at the end:**  **10 <-> 8 <-> null**  **List after deleting at location 1:**  **10 <-> null** |
| **Question Bank:**  1. \*\*Doubly Linked List\*\*:  - A doubly linked list is a type of linked list in which each node contains a data part and two pointers, `next` and `prev`.  - The `next` pointer points to the next node in the sequence, and the `prev` pointer points to the previous node.  - This allows for traversal in both forward and backward directions.  2. \*\*Complexity of Operations in Doubly Linked List\*\*:  - \*\*Traversal\*\*:  - Time Complexity: O(n)  - Explanation: In a doubly linked list, to traverse from the beginning to the end, you need to go through each node one by one. This requires visiting 'n' nodes for a list of size 'n', resulting in a linear time complexity.  - \*\*Insertion\*\*:  - \*\*At the Beginning\*\*:  - Time Complexity: O(1)  - Explanation: Inserting a node at the beginning of a doubly linked list involves updating a few pointers, which can be done in constant time.    - \*\*At the End\*\*:  - Time Complexity: O(1)  - Explanation: Inserting a node at the end of a doubly linked list also involves updating a few pointers, which can be done in constant time.    - \*\*At a Specific Location\*\*:  - Time Complexity: O(n)  - Explanation: Inserting a node at a specific location involves traversing the list to find the correct position, which takes O(n) time. The actual insertion operation, however, can be done in constant time once the position is found.  - \*\*Deletion\*\*:  - \*\*At the Beginning\*\*:  - Time Complexity: O(1)  - Explanation: Deleting a node from the beginning of a doubly linked list involves updating a few pointers, which can be done in constant time.  - \*\*At the End\*\*:  - Time Complexity: O(1)  - Explanation: Deleting a node from the end of a doubly linked list also involves updating a few pointers, which can be done in constant time.  - \*\*At a Specific Location\*\*:  - Time Complexity: O(n)  - Explanation: Deleting a node from a specific location involves traversing the list to find the correct position, which takes O(n) time. The actual deletion operation, however, can be done in constant time once the position is found.  - \*\*Overall\*\*:  - The doubly linked list provides efficient insertion and deletion at both ends, but when it comes to operations that involve finding a specific position (insertion or deletion), it takes linear time because you may have to traverse through a significant portion of the list. |

**EXPERIMENT NO. 6**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Linked List and its basic operations |
| **Outcome:**  The students will be able to implement and use Circular linked list for solving various problems |
| **Problem Statement:**  Write a program to create a Circular linked list of n nodes and perform:  • Insertion   * At the beginning * At the end * At a specific location   • Deletion   * At the beginning * At the end * At a specific location |
| **Background Study:**  **Circular linked list** is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.   **Insertion** We can insert a node in a circular linked list either as a first node (empty list), in the beginning, in the end, or in between the other nodes. Let us see each of these insertion operations using a pictorial representation below.  **#1) Insert in an empty list**  [Insert in an empty list](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/Insert-in-an-empty-list.png)  When there are no nodes in circular list and the list is empty, the last pointer is null, then we insert a new node N by pointing the last pointer to the node N as shown above. The next pointer of N will point to the node N itself as there is only one node. Thus N becomes the first as well as last node in the list.  **#2) Insert at the beginning of the list**  [Insert at the beginning of the list](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/Insert-at-the-beginning-of-the-list.png)  As shown in the above representation, when we add a node at the beginning of the list, the next pointer of the last node points to the new node N thereby making it a first node.  **N->next = last->next**  **Last->next = N**  **#3) Insert at the end of the list**  [last node points to the new node](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/last-node-points-to-the-new-node.png)  **To insert a new node at the end of the list, we follow these steps:**  **N-> next = last ->next; last ->next = N last = N**  **#4) Insert in between the list**  [Insert in between the list](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/To-insert-a-new-node-at-the-end-of-the-list.png)  Suppose we need to insert a new node N between N3 and N4, we first need to traverse the list and locate the node after which the new node is to be inserted, in this case, its N3.  **After the node is located, we perform the following steps.**  **N -> next = N3 -> next; N3 -> next = N**  This inserts a new node N after N3. **Deletion** The deletion operation of the circular linked list involves locating the node that is to be deleted and then freeing its memory.  For this we maintain two additional pointers curr and prev and then traverse the list to locate the node. The given node to be deleted can be the first node, the last node or the node in between. Depending on the location we set the curr and prev pointers and then delete the curr node.  **A pictorial representation of the deletion operation is shown below.**  [deletion operation](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/deletion.png) |
| **Algorithm (Student Work Area):**  Node Class:   1. Create a class **Node** with integer data, and a reference **next** to the next node. 2. Initialize **next** as **null** in the constructor.   CircularLinkedList Class:   1. Create a class **CircularLinkedList** with **head** as a pointer to the first node.   Insertion at the Beginning (**insertAtBeginning**):   1. Create a new node **newNode** with the given data. 2. If the list is empty (**head** is **null**), set **newNode.next** to **newNode** and set **head** to **newNode**. 3. If the list is not empty, find the last node (**temp**) using a loop. 4. Set **temp.next** to **newNode** and set **newNode.next** to **head**. 5. Set **head** to **newNode**.   Insertion at the End (**insertAtEnd**):   1. Create a new node **newNode** with the given data. 2. If the list is empty (**head** is **null**), set **newNode.next** to **newNode** and set **head** to **newNode**. 3. If the list is not empty, find the last node (**temp**) using a loop. 4. Set **temp.next** to **newNode** and set **newNode.next** to **head**.   Insertion at a Specific Location (**insertAtLocation**):   1. Create a new node **newNode** with the given data. 2. If **position** is **0**, call **insertAtBeginning(data)**. 3. Otherwise, traverse the list to the node at position **position - 1**. 4. Insert **newNode** between the node at **position - 1** and its **next**. 5. Update the **next** references accordingly.   Deletion at the Beginning (**deleteAtBeginning**):   1. If the list is empty (**head** is **null**), print a message and return. 2. If the list has only one node (**head.next** is **head**), set **head** to **null**. 3. Otherwise, find the last node (**temp**) using a loop. 4. Set **temp.next** to **head.next** and set **head** to **head.next**.   Deletion at the End (**deleteAtEnd**):   1. If the list is empty (**head** is **null**), print a message and return. 2. If the list has only one node (**head.next** is **head**), set **head** to **null**. 3. Otherwise, find the last node (**temp**) using a loop. 4. Traverse again to the second last node (**secondLast**) using a loop. 5. Set **secondLast.next** to **head**.   Deletion at a Specific Location (**deleteAtLocation**):   1. If **position** is **0**, call **deleteAtBeginning()**. 2. Otherwise, traverse the list to the node at position **position - 1**. 3. Set **temp.next** to **temp.next.next**.   Display (**display**):   1. If the list is empty (**head** is **null**), print a message and return. 2. Start from **head** and print the data of each node until you reach **head** again. |
| **Code (Student Work Area):**  class Node {  int data;  Node next;  Node(int data) {  this.data = data;  this.next = null;  }  }  class CircularLinkedList {  Node head;  // Insertion at the beginning  public void insertAtBeginning(int data) {  Node newNode = new Node(data);  if (head == null) {  newNode.next = newNode;  head = newNode;  } else {  Node temp = head;  while (temp.next != head) {  temp = temp.next;  }  temp.next = newNode;  newNode.next = head;  head = newNode;  }  }  // Insertion at the end  public void insertAtEnd(int data) {  Node newNode = new Node(data);  if (head == null) {  newNode.next = newNode;  head = newNode;  } else {  Node temp = head;  while (temp.next != head) {  temp = temp.next;  }  temp.next = newNode;  newNode.next = head;  }  }  // Insertion at a specific location  public void insertAtLocation(int data, int position) {  Node newNode = new Node(data);  if (position == 0) {  insertAtBeginning(data);  return;  }  Node temp = head;  for (int i = 0; i < position - 1 && temp.next != head; i++) {  temp = temp.next;  }  newNode.next = temp.next;  temp.next = newNode;  }  // Deletion at the beginning  public void deleteAtBeginning() {  if (head == null) {  System.out.println("List is empty");  return;  }  if (head.next == head) {  head = null;  } else {  Node temp = head;  while (temp.next != head) {  temp = temp.next;  }  temp.next = head.next;  head = head.next;  }  }  // Deletion at the end  public void deleteAtEnd() {  if (head == null) {  System.out.println("List is empty");  return;  }  if (head.next == head) {  head = null;  } else {  Node temp = head;  while (temp.next.next != head) {  temp = temp.next;  }  temp.next = head;  }  }  // Deletion at a specific location  public void deleteAtLocation(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 && temp.next != head; i++) {  temp = temp.next;  }  if (temp.next == head) {  System.out.println("Invalid position");  return;  }  temp.next = temp.next.next;  }  public void display() {  if (head == null) {  System.out.println("List is empty");  return;  }  Node temp = head;  do {  System.out.print(temp.data + " -> ");  temp = temp.next;  } while (temp != head);  System.out.println("...");  }  }  public class Main {  public static void main(String[] args) {  CircularLinkedList list = new CircularLinkedList();  list.insertAtEnd(10);  list.insertAtBeginning(5);  list.insertAtEnd(15);  list.insertAtLocation(8, 2);  System.out.println("Original list:");  list.display();  list.deleteAtBeginning();  System.out.println("List after deleting at the beginning:");  list.display();  list.deleteAtEnd();  System.out.println("List after deleting at the end:");  list.display();  list.deleteAtLocation(1);  System.out.println("List after deleting at location 1:");  list.display();  }  } |
| **Output – Screenshots (Student Work Area):**  **Original list:**  **5 -> 10 -> 8 -> 15 -> ...**  **List after deleting at the beginning:**  **10 -> 8 -> 15 -> ...**  **List after deleting at the end:**  **10 -> 8 -> ...**  **List after deleting at location 1:**  **10 -> ...** |
| **Question Bank:**  **\*\*Circular Linked List vs. Singly Linked List\*\*:**  **1. \*\*Definition\*\*:**  **- \*\*Singly Linked List\*\*: In a singly linked list, each node contains a data part and a reference (link) to the next node in the sequence.**  **- \*\*Circular Linked List\*\*: A circular linked list is a variation of a singly linked list where the last node points back to the first node, forming a circle.**  **2. \*\*Termination Condition\*\*:**  **- \*\*Singly Linked List\*\*: The last node points to null, indicating the end of the list.**  **- \*\*Circular Linked List\*\*: The last node points back to the first node, creating a loop.**  **3. \*\*Traversal\*\*:**  **- \*\*Singly Linked List\*\*: In a singly linked list, to traverse from the beginning to the end, you need to go through each node one by one, stopping at the last node (which points to null).**  **- \*\*Circular Linked List\*\*: In a circular linked list, since the last node points back to the first node, you can start at any node and traverse the entire list without encountering a null reference.**  **4. \*\*Operations\*\*:**  **- \*\*Insertion and Deletion\*\*:**  **- The operations for insertion and deletion in a circular linked list are similar to those in a singly linked list, with the added consideration that in a circular linked list, you may need to adjust the `next` pointer of the last node to maintain the circular structure.**  **- \*\*Search Operation\*\*:**  **- For both types of linked lists, searching for an element typically requires traversing the list, which takes O(n) time in the worst case.**  **\*\*Complexity Analysis\*\*:**  **1. \*\*Traversal\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(n)**  **- Explanation: Traversal in a singly linked list requires visiting each node, which takes linear time.**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(n)**  **- Explanation: Even though it's a circular list, the traversal still visits each node, resulting in linear time complexity.**  **2. \*\*Insertion\*\*:**  **- \*\*At the Beginning\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(1)**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(1)**  **- \*\*At the End\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(n)**  **- (Since you need to traverse the list to find the last node)**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(1)**  **- \*\*At a Specific Location\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(n)**  **- (Since you may need to traverse to the specified position)**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(n)**  **- (In the worst case, you may need to traverse the entire list to find the position)**  **3. \*\*Deletion\*\*:**  **- \*\*At the Beginning\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(1)**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(1)**  **- \*\*At the End\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(n)**  **- (Since you need to traverse the list to find the second last node)**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(n)**  **- (In the worst case, you may need to traverse the entire list to find the position)**  **- \*\*At a Specific Location\*\*:**  **- \*\*Singly Linked List\*\*:**  **- Time Complexity: O(n)**  **- (Since you may need to traverse to the specified position)**  **- \*\*Circular Linked List\*\*:**  **- Time Complexity: O(n)**  **- (In the worst case, you may need to traverse the entire list to find the position)**  **In summary, both singly linked lists and circular linked lists have similar time complexities for traversal, insertion, and deletion operations. The choice between them depends on the specific requirements of the application.** |

**EXPERIMENT NO. 7**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Stacks and its basic operations |
| **Outcome:**  The students will be able to implement and use Stacks for solving various problems |
| **Problem Statement:**  Write a program to create a stack and perform:   * POP * PUSH * PEEK * ISEMPTY * ISFULL  1. Use Arrays for Implementation 2. Use Linked List for Implementation |
| **Background:**  Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.  For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.  **Inserting and deleting elements**  Stacks have restrictions on the insertion and deletion of elements. Elements can be inserted or deleted only from one end of the stack i.e. from the top. The element at the top is called the top element. The operations of inserting and deleting elements are called push() and pop() respectively.  When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.  For example, in the stack of trays, if you take the tray on the top and do not replace it, then the second tray automatically becomes the top element (tray) of that stack.  **Features of stacks**   * Dynamic data structures * Do not have a fixed size * Do not consume a fixed amount of memory * Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.   A stack can be visualized as follows:  enter image description here |
| **Algorithm (Student Work Area):**  **ArrayStack Class**:   1. Initialize an array **stack** with a specified **maxSize**, **top** to -1, and **maxSize** to the given value.   **push(data)**:   1. Check if the stack is full by comparing **top** with **maxSize - 1**. 2. If the stack is not full, increment **top** and add **data** to the **top** index of the **stack**.   **pop()**:   1. Check if the stack is empty by comparing **top** with -1. 2. If the stack is not empty, retrieve the element at **stack[top]**, decrement **top**, and return the element.   **peek()**:   1. Check if the stack is empty by comparing **top** with -1. 2. If the stack is not empty, return the element at **stack[top]**.   **isEmpty()**:   1. Check if **top** is equal to -1. If true, return **true**; otherwise, return **false**.   **isFull()**:   1. Check if **top** is equal to **maxSize - 1**. If true, return **true**; otherwise, return **false**.   **2) Implementation using Linked List:**  Algorithm:  **Node Class**:   1. Create a class **Node** with integer **data** and a reference **next** to the next node.   **LinkedListStack Class**:   1. Initialize **top** as a reference to the top node.   **push(data)**:   1. Create a new node **newNode** with the given data. 2. Set **newNode.next** to the current **top**. 3. Update **top** to **newNode**.   **pop()**:   1. Check if the stack is empty by verifying if **top** is null. 2. If the stack is not empty, retrieve **top.data**, set **top** to **top.next**, and return the data.   **peek()**:   1. Check if the stack is empty by verifying if **top** is null. 2. If the stack is not empty, return **top.data**.   **isEmpty()**:   1. Check if **top** is null. If true, return **true**; otherwise, return **false**.   Note:   * In both implementations, the **push**, **pop**, **peek**, and **isEmpty** operations have a time complexity of O(1). * In the array-based implementation, there is an additional **isFull** method that checks if the array is full. This operation also has a time complexity of O(1). |
| **Code (Student Work Area):**  class ArrayStack {  private int[] stack;  private int top;  private int maxSize;  public ArrayStack(int maxSize) {  this.maxSize = maxSize;  this.stack = new int[maxSize];  this.top = -1;  }  public void push(int data) {  if (isFull()) {  System.out.println("Stack overflow");  return;  }  top++;  stack[top] = data;  }  public int pop() {  if (isEmpty()) {  System.out.println("Stack underflow");  return -1;  }  int data = stack[top];  top--;  return data;  }  public int peek() {  if (isEmpty()) {  System.out.println("Stack is empty");  return -1;  }  return stack[top];  }  public boolean isEmpty() {  return top == -1;  }  public boolean isFull() {  return top == maxSize - 1;  }  }  public class Main {  public static void main(String[] args) {  ArrayStack stack = new ArrayStack(5);  stack.push(10);  stack.push(20);  stack.push(30);  System.out.println("Top element: " + stack.peek());  System.out.println("Popping elements:");  while (!stack.isEmpty()) {  System.out.println(stack.pop());  }  }  }  class Node {  int data;  Node next;  public Node(int data) {  this.data = data;  this.next = null;  }  }  class LinkedListStack {  private Node top;  public void push(int data) {  Node newNode = new Node(data);  newNode.next = top;  top = newNode;  }  public int pop() {  if (isEmpty()) {  System.out.println("Stack underflow");  return -1;  }  int data = top.data;  top = top.next;  return data;  }  public int peek() {  if (isEmpty()) {  System.out.println("Stack is empty");  return -1;  }  return top.data;  }  public boolean isEmpty() {  return top == null;  }  }  public class Main {  public static void main(String[] args) {  LinkedListStack stack = new LinkedListStack();  stack.push(10);  stack.push(20);  stack.push(30);  System.out.println("Top element: " + stack.peek());  System.out.println("Popping elements:");  while (!stack.isEmpty()) {  System.out.println(stack.pop());  }  }  } |
| **Output – Screenshots (Student Work Area):**  **Top element: 30**  **Popping elements:**  **30**  **20**  **10** |
| **Question Bank:**  \*\*Stacks\*\*:  A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. In a stack, elements are added and removed from the same end, which is known as the top.  Operations on a stack include:  1. \*\*Push\*\*: Adds an element to the top of the stack.  2. \*\*Pop\*\*: Removes and returns the element from the top of the stack.  3. \*\*Peek (or Top)\*\*: Returns the element at the top of the stack without removing it.  4. \*\*isEmpty\*\*: Checks if the stack is empty.  5. \*\*isFull\*\* (in some implementations): Checks if the stack is full (applicable in fixed-size array implementations).  \*\*Applications of Stacks\*\*:  1. \*\*Function Calls and Recursion\*\*:  - Stacks are used to manage function calls in programming languages. When a function is called, its state (including local variables and the return address) is stored on the call stack.  2. \*\*Expression Evaluation\*\*:  - Stacks are used to evaluate arithmetic expressions, including infix, postfix, and prefix notations.  3. \*\*Backtracking Algorithms\*\*:  - In algorithms like Depth-First Search (DFS), stacks are used to keep track of visited nodes and the current path.  4. \*\*Undo Operations\*\*:  - In applications like text editors or graphic design software, stacks are used to implement undo functionality.  5. \*\*Browser History\*\*:  - The back and forward buttons in web browsers are implemented using two stacks to keep track of visited pages.  6. \*\*Parenthesis Matching\*\*:  - Stacks are used to verify the correctness of parentheses, braces, and brackets in mathematical expressions.  7. \*\*Compiler Implementations\*\*:  - Stacks are used in syntax analysis (parsing) to evaluate expressions and generate abstract syntax trees.  8. \*\*Postfix Evaluation\*\*:  - Stacks are used to evaluate expressions in postfix notation (also known as Reverse Polish Notation).  9. \*\*Depth-First Search (DFS)\*\*:  - In graph traversal algorithms like DFS, stacks are used to keep track of visited nodes and the current path.  10. \*\*Memory Management\*\*:  - Stacks are used in computer memory management for function calls, local variables, and managing memory allocation/deallocation.  11. \*\*Balanced Parentheses Checking\*\*:  - Stacks are used to check if a given expression has balanced parentheses.  12. \*\*Operating System Implementations\*\*:  - Stacks are used to manage the execution of processes and their respective states.  These are just a few examples. Stacks have a wide range of applications in computer science and everyday programming. They provide a simple and efficient way to manage data and control flow in various scenarios. |

**EXPERIMENT NO. 8**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Stacks and its applications. |
| **Outcome:**  The students will be able to implement and use Stacks for solving various problems |
| **Problem Statement:**  Write a program to create a stack and perform:  Reversal of a sentence using stack.  **Given a string str consisting of a sentence, the task is to reverse the entire sentence word by word.**  **Examples:**  **Input: str = “data structures and algorithms” Output:  algorithms and structures data** |
| **Background:**  Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.  For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.  **Inserting and deleting elements**  Stacks have restrictions on the insertion and deletion of elements. Elements can be inserted or deleted only from one end of the stack i.e. from the top. The element at the top is called the top element. The operations of inserting and deleting elements are called push() and pop() respectively.  When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.  For example, in the stack of trays, if you take the tray on the top and do not replace it, then the second tray automatically becomes the top element (tray) of that stack.  **Features of stacks**   * Dynamic data structures * Do not have a fixed size * Do not consume a fixed amount of memory * Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.   A stack can be visualized as follows:  enter image description here |
| **Algorithm (Student Work Area):**   1. Create a class **ReverseSentence**. 2. Define a static method **reverseSentence** that takes a string **str** as input. 3. Inside the method, create a stack **wordStack** to store words. 4. Split the input sentence **str** into an array of words using **str.split(" ")**. 5. Push each word onto the stack. 6. Construct the reversed sentence by popping words from the stack and appending them to a **StringBuilder** with spaces in between. 7. Return the reversed sentence as a string. |
| **Code (Student Work Area):**  import java.util.Stack;  public class ReverseSentence {  public static String reverseSentence(String str) {  // Create a stack to store words  Stack<String> wordStack = new Stack<>();  // Split the sentence into words  String[] words = str.split(" ");  // Push words onto the stack  for (String word : words) {  wordStack.push(word);  }  // Pop words from the stack to construct the reversed sentence  StringBuilder reversedSentence = new StringBuilder();  while (!wordStack.isEmpty()) {  reversedSentence.append(wordStack.pop()).append(" ");  }  return reversedSentence.toString().trim();  }  public static void main(String[] args) {  String inputSentence = "data structures and algorithms";  String reversedSentence = reverseSentence(inputSentence);  System.out.println("Reversed Sentence:");  System.out.println(reversedSentence);  }  } |
| **Output – Screenshots (Student Work Area):** |
| **Question Bank:**  \*\*Stacks\*\*:  A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. In a stack, elements are added and removed from the same end, which is known as the top.  \*\*Operations on a Stack\*\*:  1. \*\*Push\*\*: Adds an element to the top of the stack.  2. \*\*Pop\*\*: Removes and returns the element from the top of the stack.  3. \*\*Peek (or Top)\*\*: Returns the element at the top of the stack without removing it.  4. \*\*isEmpty\*\*: Checks if the stack is empty.  5. \*\*isFull\*\* (in some implementations): Checks if the stack is full (applicable in fixed-size array implementations).  \*\*Splitting a Sentence and Pushing it into a Stack\*\*:  To split a sentence and push it into a stack, you'll need to follow these steps:  1. \*\*Split the Sentence\*\*:  - Use a string manipulation function (like `split` in Java or similar functions in other programming languages) to split the sentence into individual words. The delimiter would typically be a space (' ').  2. \*\*Pushing into the Stack\*\*:  - Initialize a stack data structure.  - Iterate through the words obtained from step 1.  - For each word, push it onto the stack.  Here's an example in Java:  ```java  import java.util.Stack;  public class Main {  public static void main(String[] args) {  String sentence = "This is a sample sentence";  // Step 1: Split the Sentence  String[] words = sentence.split(" ");  // Step 2: Pushing into the Stack  Stack<String> wordStack = new Stack<>();  for (String word : words) {  wordStack.push(word);  }  // Printing the Stack  System.out.println("Words pushed into the stack:");  while (!wordStack.isEmpty()) {  System.out.println(wordStack.pop());  }  }  }  ```  In this example, the sentence "This is a sample sentence" is split into an array of words. Each word is then pushed onto a stack. Finally, the words are popped from the stack, resulting in the reversed order of words. |

**EXPERIMENT NO. 9**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Stacks and its applications. |
| **Outcome:**  The students will be able to implement and use Stacks for solving various problems |
| **Problem Statement:**  Write a program to check whether the parenthesis in the expression are balanced or not.  **Given a string str consisting of an expression**  **Examples:**  **Input: str = (a+b)\*c**  **Output: Parenthesis Balanced** |
| **Background:**  Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.  For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.  **Inserting and deleting elements**  Stacks have restrictions on the insertion and deletion of elements. Elements can be inserted or deleted only from one end of the stack i.e. from the top. The element at the top is called the top element. The operations of inserting and deleting elements are called push() and pop() respectively.  When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.  For example, in the stack of trays, if you take the tray on the top and do not replace it, then the second tray automatically becomes the top element (tray) of that stack.  **Features of stacks**   * Dynamic data structures * Do not have a fixed size * Do not consume a fixed amount of memory * Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.   A stack can be visualized as follows:  enter image description here |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **Q: How a stack helps in syntax analysis or compilation of a program?** |

**EXPERIMENT NO. 10**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Stacks and its applications. |
| **Outcome:**  The students will be able to implement and use Stacks for solving various problems |
| **Problem Statement:**  Write a program to convert Infix expression into Postfix.  **Given a string str consisting of an infix expression, convert it into Postfix**  **Examples:**  **Input: str = (a+b)\*c**  **Output: ab+\*** |
| **Background:**  Any expression can be represented using three types of expressions (Infix, Postfix, and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.  **Infix to postfix conversion** Scan through an expression, getting one token at a time.  1 Fix a priority level for each operator. For example, from high to low:      3.    - (unary negation)     2.    \* /     1.    + - (subtraction)  Thus, high priority corresponds to high number in the table.  2 If the token is an operand, do not stack it. Pass it to the output.  3 If token is an operator or parenthesis, do the following:     -- Pop the stack until you find a symbol of lower priority number than the current one. An incoming left parenthesis will be considered to have higher priority than any other symbol. A left parenthesis on the stack will not be removed unless an incoming right parenthesis is found. The popped stack elements will be written to output.     --Stack the current symbol.     -- If a right parenthesis is the current symbol, pop the stack down to (and including) the first left parenthesis. Write all the symbols except the left parenthesis to the output (i.e. write the operators to the output).     -- After the last token is read, pop the remainder of the stack and write any symbol (except left parenthesis) to output.  **Example:**  Convert A \* (B + C) \* D to postfix notation.   |  |  |  |  | | --- | --- | --- | --- | | **Move** | **Curren Ttoken** | **Stack** | **Output** | | 1 | A | empty | A | | 2 | \* | \* | A | | 3 | ( | (\* | A | | 4 | B | (\* | A B | | 5 | + | +(\* | A B | | 6 | C | +(\* | A B C | | 7 | ) | \* | A B C + | | 8 | \* | \* | A B C + \* | | 9 | D | \* | A B C + \* D | | 10 |  | empty |  | |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**   1. **Why conversion is required?** 2. **How we can convert infix to prefix and prefix to postfix?** |

**EXPERIMENT NO. 11**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Stacks and its application Recursion. |
| **Outcome:**  The students will be able to implement and use Stacks for solving Recursion problems |
| **Problem Statement:**  Write a program to implement Tower of Hanoi. |
| **Background:**  Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −  Tower Of Hanoi  These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same. **Rules** The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −   * Only one disk can be moved among the towers at any given time. * Only the "top" disk can be removed. * No large disk can sit over a small disk.  |  |  |  |  | | --- | --- | --- | --- | | Tower of Hanoi puzzle with n disks can be solved in minimum **2n−1** steps  . |  |  |  | |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**   1. **What is Recursion?** 2. **What is Base condition?** 3. **What are the number of steps required to solve n-Disc problem?** |

**EXPERIMENT NO. 12**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Queue and its applications. |
| **Outcome:**  The students will be able to implement and use Queues for solving various problems |
| **Problem Statement:**  Write a program to implement Following operations using Queue:   1. Enqueue() 2. Dequeue() 3. Isfull() 4. Isempty() 5. Peek() 6. Using array implementation 7. Using Linked List Implementation |
| **Background:**  **Queue** is also an abstract data type or a linear data structure, just like [stack data structure](https://www.studytonight.com/data-structures/stack-data-structure), in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).  This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.  Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.  The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.  Introduction to Queue **Basic features of Queue**  1. Like stack, queue is also an ordered list of elements of similar data types. 2. Queue is a FIFO( First in First Out ) structure. 3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element. 4. peek( ) function is oftenly used to return the value of first element without dequeuing it. |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**   1. **What are the applications of queues?** 2. **Queues can be implemented with the help of stack. How?** |

**EXPERIMENT NO. 13**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Circular Queue and its applications. |
| **Outcome:**  The students will be able to implement and use Circular Queues for solving various problems |
| **Problem Statement:**  Write a program to implement Following operations using Circular Queue:   1. Enqueue() 2. Dequeue()   Using array implementation |
| **Background:**  **Queue** is also an abstract data type or a linear data structure, just like [stack data structure](https://www.studytonight.com/data-structures/stack-data-structure), in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).  This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.  Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.  The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.  Circular Queue in C++ **Basic features of Queue**  1. Like stack, queue is also an ordered list of elements of similar data types. 2. Queue is a FIFO( First in First Out ) structure. 3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element. 4. peek( ) function is oftenly used to return the value of first element without dequeuing it. |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**   1. **What are the applications of Circular queues?** 2. **What is the complexity of all operations in Circular Queue?** |

**EXPERIMENT NO. 14**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with linear data structure Doubly Ended Queue and its applications. |
| **Outcome:**  The students will be able to implement and use Doubly ended Queues for solving various problems |
| **Problem Statement:**  Write a program to implement Following operations using Doubly ended Queue:   1. Enqueue() 2. Dequeue() 3. Isfull() 4. Isempty() 5. Peek()   Using array implementation |
| **Background:**  **Queue** is also an abstract data type or a linear data structure, just like [stack data structure](https://www.studytonight.com/data-structures/stack-data-structure), in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).  This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.  Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.  The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.   **Basic features of Queue**  1. Like stack, queue is also an ordered list of elements of similar data types. 2. Queue is a FIFO( First in First Out ) structure. 3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element. 4. peek( ) function is oftenly used to return the value of first element without dequeuing it. |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**   1. **What are the applications of Doubly ended queues?** 2. **What is the complexity of all operations?** |

**EXPERIMENT NO. 15**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with Non-linear data structure Binary Search Tree and its operations. |
| **Outcome:**  The students will be able to implement and use Binary Search Tree for solving various problems |
| **Problem Statement:**  Write a program to implement Following operations using Binary Search Tree:   1. Insertion 2. Deletion 3. Traversal [PREORDER, POSTORDER, INORDER] |
| **Background:**  **Binary Search Tree** is a node-based binary tree data structure which has the following properties:   * The left subtree of a node contains only nodes with keys lesser than the node’s key. * The right subtree of a node contains only nodes with keys greater than the node’s key. * The left and right subtree each must also be a binary search tree.     **Insertion In Binary Search Tree:**  1. Start from the root.  2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.  3. After reaching the end, just insert that node at left(if less than current) else right.  **Deletion from Binary Search Tree:**  **1)*Node to be deleted is the*** ***leaf:*** Simply remove from the tree.  50 50  / \ delete(20) / \  30 70 ---------> 30 70  / \ / \ \ / \  20 40 60 80 40 60 80  **2) *Node to be deleted has only one child:*** Copy the child to the node and delete the child  50 50  / \ delete(30) / \  30 70 ---------> 40 70  \ / \ / \  40 60 80 60 80  **3) *Node to be deleted has two children:***Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.  50 60  / \ delete(50) / \  40 70 ---------> 40 70  / \ \  60 80 80 |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**   1. **What is the difference between Binary Tree and Binary Search Tree?** 2. **What is the complexity of all search operations in BST?** |

**EXPERIMENT NO. 16**

|  |
| --- |
| **Student Name and Roll Number:** |
| **Semester /Section:** |
| **Link to Code:** |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To familiarize the students with different sorting operations. |
| **Outcome:**  The students will be able to implement and use various sorting techniques. |
| **Problem Statement:**  Write a program to implement:   1. Bubble Sort 2. Insertions Sort 3. Selection Sort 4. Quick Sort 5. Merge Sort |
| **Background:**  Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array A = {A1, A2, A3, A4, …. An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > .. > An .  **Consider an array;**  int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9 )  After Sorting array would be:  A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }  There are many techniques by using which, sorting can be performed.   |  |  |  | | --- | --- | --- | | **SN** | **Sorting Algorithms** | **Description** | | 1 | [Bubble Sort](https://www.javatpoint.com/bubble-sort) | It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly. | | 2 | [Insertion Sort](https://www.javatpoint.com/insertion-sort) | As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge. | | 3 | [Merge Sort](https://www.javatpoint.com/merge-sort) | Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array. | | 4 | [Quick Sort](https://www.javatpoint.com/quick-sort) | Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach. | | 5 | [Selection Sort](https://www.javatpoint.com/selection-sort) | Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worst than insertion sort. | |
| **Algorithm (Student Work Area):** |
| **Code (Student Work Area):** |
| **Output – Screenshots (Student Work Area):** |
| **QUESTION BANK:**  **Compare and contrast all Sorting techniques?** |