## **EXPERIMENT 1**

### **Aim:**

1. Write a program in C to perform linear and binary search.
2. Write a program in C to perform bubble sort, insertion sort and selection sort. Take the array size and array elements from user.
3. Write a program in C that obtains the minimum and maximum element from the array. Modify this program to give the second largest and second smallest element of the array.

### **Theory (Linear Search):**

Linear search is a straightforward algorithm for finding a target value within an array. It involves sequentially checking each element from the beginning until the target value is found or the end of the array is reached. This method is simple but can be inefficient for large arrays, as it may require examining every element.

### **Program:**

#include <stdio.h>

int linearSearch(int item, int size, int arr[]);

int main()

{

    //VedeshP

    // Linear search

    int item;

    int arr[] = {3,4,1,2,9,7,0,5,8,6};

    printf("Enter the number you want to search: ");

    scanf("%d", &item);

    int size = sizeof(arr) / sizeof(arr[0]);

    int index = linearSearch(item, size, arr);

    // printf("%d", size);

    if (index == -1)

    {

        printf("Number not found");

        return -1;

    }

    else

    {

        printf("Index is %d", index);

        return 0;

    }

}

**int linearSearch(int item, int size, int arr[])**

**{**

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

**{**

**if (arr[i] == item)**

**{**

**return i;**

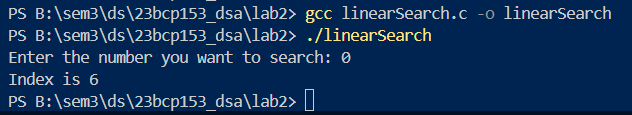
**}**

**}**

**return -1;**

**}**

### **Output:**



### **Time Complexity:**

* **Worst Case:** – The target element is at the last position or not present, so every element in the array must be checked.
* **Average Case:** – On average, the search will check half of the elements, which still results in linear time complexity.
* **Best Case:** – The target element is at the first position, so only one comparison is needed.

### **Theory (Binary Search):**

Binary search is an efficient algorithm for finding a target value in a sorted array. It works by repeatedly dividing the search interval in half. Starting with the entire array, the algorithm compares the target value to the middle element. If the target is equal to the middle element, the search is complete. If the target is less than the middle element, the search continues in the lower half; if greater, in the upper half. This process continues until the target is found or the interval is empty.

### **Program:**

#include <stdio.h>

// Get time here too for complexity

int binarySearch(int arr[], int start, int end, int target);

void insertionSort(int arr[], int n);

int main(void)

{

    int n = 0;

    printf("How many elements do you want to enter ?: ");

    scanf("%d", &n);

    int arr[n];

    for (int i = 0; i < n; i++)

    {

        printf("Enter %d number: ", i + 1);

        scanf("%d", &arr[i]);

    }

    insertionSort(arr, n);

    printf("Sorted Array:\n");

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

    int target;

    printf("What element do you want to find? : ");

    scanf("%d", &target);

    int location = binarySearch(arr, 0, n - 1, target);

    if (location == -1)

    {

        printf("Element not found!");

        return -1;

    }

    printf("Location of your target is: %d", location);

    return 0;

}

**int binarySearch(int arr[], int start, int end, int target)**

**{**

**while (start <= end)**

**{**

**int mid = (start + end) / 2;**

**if (arr[mid] == target)**

**{**

**return mid;**

**}**

**else if (arr[mid] < target)**

**{**

**start = mid + 1;**

**}**

**else**

**{**

**end = mid - 1;**

**}**

**}**

**return -1;**

**}**

void insertionSort(int arr[], int n)

{

    for (int i = 1; i < n; i++)

    {

        int key = arr[i];

        int j = i - 1;

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

        {

            arr[j + 1] = arr[j];

            j--;

        }

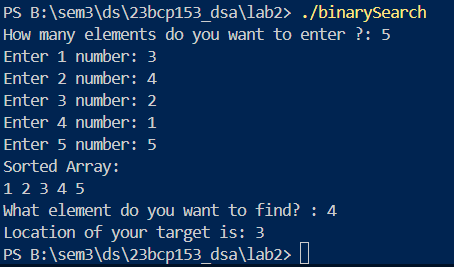
        arr[j + 1] = key;

    }

    return;

}

### **Output:**



### **Time Complexity:**

**Worst Case:**

* **Explanation:** In binary search, the array is divided in half with each iteration. Mathematically, this means the number of elements to check is reduced exponentially. After k iterations, the number of elements left to check is . We stop when becomes 1, which means . Taking the logarithm (base 2) of both sides, we get . Thus, the time complexity is .
* Because in this we do the iterations in this manner:
* When we are left with only one element we get: , therefore

**Average Case:**

* **Explanation:** On average, binary search will also perform log(n) comparisons. This is because, regardless of the position of the target value, the algorithm always divides the search space in half, leading to the same logarithmic growth rate.

**Best Case:**

* **Explanation:** The best case occurs when the target value is the middle element of the array in the first comparison. In this case, only one comparison is needed, resulting in constant time complexity, .

### **Theory (Bubble Sort):**

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated for each element in the list until the entire list is sorted. In each pass through the list, the largest unsorted element "bubbles up" to its correct position. The algorithm continues to reduce the number of elements considered in each subsequent pass, effectively sorting the list from the end towards the beginning. The number of passes needed is one less than the number of elements in the list.

### **Program:**

#include <stdio.h>

void bubbleSort(int arr[], int n);

int main(void)

{

    int n = 0;

    printf("How many elements do you want to enter ?: ");

    scanf("%d", &n);

    int arr[n];

    for (int i = 0; i < n; i++)

    {

        printf("Enter %d number: ", i + 1);

        scanf("%d", &arr[i]);

    }

    bubbleSort(arr, n);

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    return 0;

}

**void bubbleSort(int arr[], int n)**

**{**

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

**{**

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

**{**

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

**{**

**int temp = arr[j];**

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

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

**}**

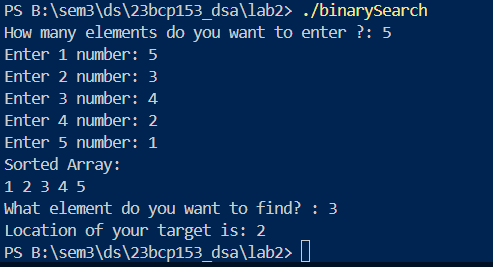
**}**

**}**

**return;**

**}**

### **Output:**



### **Time Complexity:**

**Worst Case:**

* **Explanation:** Occurs when the array is sorted in reverse order. The algorithm needs to perform a maximum number of comparisons and swaps, resulting in a quadratic time complexity.

**Average Case:**

* **Explanation:** On average, bubble sort performs a quadratic number of comparisons and swaps, as it does not have an efficient way to handle partially sorted arrays.

**Best Case:**

* **Explanation:** Occurs when the array is already sorted. With an optimized version of bubble sort that checks if any swaps were made in a pass, the algorithm can complete in linear time. If no swaps are needed during a pass, the algorithm terminates early.

### **Theory (Insertion Sort):**

Insertion sort maintains a sorted section of the array and iterates through the unsorted section. For each element in the unsorted section, it compares the element with those in the sorted section and inserts it into the correct position within the sorted part. This process involves shifting elements in the sorted part to make space for the new element. The algorithm continues until all elements are sorted, resulting in a fully ordered array.

### **Program:**

#include <stdio.h>

void insertionSort(int arr[], int n);

int main(void)

{

    int n = 0;

    printf("How many elements do you want to enter ?: ");

    scanf("%d", &n);

    int arr[n];

    for (int i = 0; i < n; i++)

    {

        printf("Enter %d number: ", i + 1);

        scanf("%d", &arr[i]);

    }

    insertionSort(arr, n);

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    return 0;

}

**void insertionSort(int arr[], int n)**

**{**

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

**{**

**int key = arr[i];**

**int j = i - 1;**

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

**{**

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

**j--;**

**}**

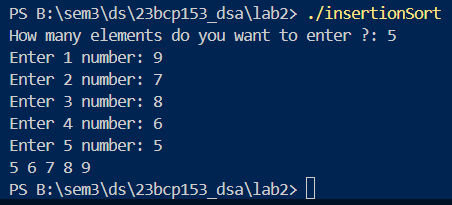
**arr[j + 1] = key;**

**}**

**return;**

**}**

### **Output:**



### **Time Complexity:**

**Worst Case:**

* **Explanation:** Occurs when the array is sorted in reverse order. For each element, the algorithm might need to compare it with every element in the sorted portion, leading to quadratic time complexity.

**Average Case:**

* **Explanation:** On average, the algorithm performs a quadratic number of comparisons and shifts, as elements are not uniformly distributed, requiring significant comparisons and shifts.

**Best Case:**

* **Explanation:** Occurs when the array is already sorted. The algorithm only needs to make a single pass through the array, performing a constant amount of work for each element as no elements need to be shifted.

### **Theory (Selection Sort):**

Selection sort is a simple sorting algorithm that sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and moving it to the end of the sorted portion. Here’s a step-by-step explanation:

1. **Find the Minimum:** Start by finding the minimum element in the unsorted part of the array.
2. **Swap:** Swap this minimum element with the first element of the unsorted part. This places the minimum element in its correct position in the sorted part of the array.
3. **Move the Boundary:** The boundary between the sorted and unsorted parts of the array moves one element forward.
4. **Repeat:** Repeat the process for the remaining unsorted elements until the entire array is sorted.

### **Program:**

#include <stdio.h>

void selectionSort(int arr[], int n);

int main(void)

{

    int n = 0;

    printf("How many elements do you want to enter ?: ");

    scanf("%d", &n);

    int arr[n];

    for (int i = 0; i < n; i++)

    {

        printf("Enter %d number: ", i + 1);

        scanf("%d", &arr[i]);

    }

    selectionSort(arr, n);

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    return 0;

}

**void selectionSort(int arr[], int n)**

**{**

**for (int i = 0; i < n; i++)**

**{**

**int min = i;**

**for (int j = i; j < n; j++)**

**{**

**if (arr[j] < arr[min])**

**{**

**min = j;**

**}**

**}**

**if (min != i)**

**{**

**int temp = arr[i];**

**arr[i] = arr[min];**

**arr[min] = temp;**

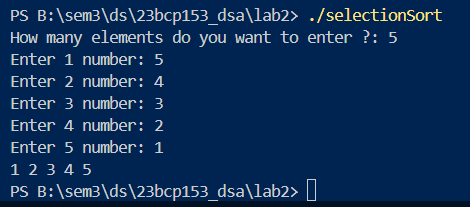
**}**

**}**

**return;**

**}**

### **Output:**



### **Time Complexity:**

**Worst Case:**

* **Explanation:** In every iteration, selection sort scans the unsorted portion of the array to find the minimum element. This scanning process involves O(n) comparisons for each of the n elements, resulting in a quadratic time complexity.

**Average Case:**

* **Explanation:** Regardless of the initial order of elements, selection sort always performs a fixed number of comparisons and swaps. The overall number of comparisons remains quadratic.

**Best Case:**

* **Explanation:** Even if the array is already sorted, selection sort still performs the same number of comparisons to find the minimum element for each iteration. Hence, the best case also has quadratic time complexity.

### **Theory (Minimum and Maximum elements in array):**

Using insertion sort to sort the array and then providing the first element, and the last element for the minimum element and the maximum element as the output respectively. Similarly providing 2nd element and 2nd last element for the 2nd minimum and 2nd maximum elements respectively.

### **Program:**

#include <stdio.h>

void insertionSort(int arr[], int n);

int main(void)

{

    int n = 0;

    printf("How many elements do you want to enter ?: ");

    scanf("%d", &n);

    int arr[n];

    for (int i = 0; i < n; i++)

    {

        printf("Enter %d number: ", i + 1);

        scanf("%d", &arr[i]);

    }

    insertionSort(arr, n);

    printf("Sorted Array:\n");

    for (int i = 0; i < n; i++)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

    printf("Minumum element is %d\n", arr[0]);

    printf("Maximum element is %d\n", arr[n - 1]);

    printf("Second Minimum element is %d\n", arr[1]);

    printf("Second Maximum element is %d\n", arr[n - 2]);

    return 0;

}

**void insertionSort(int arr[], int n)**

**{**

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

**{**

**int key = arr[i];**

**int j = i - 1;**

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

**{**

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

**j--;**

**}**

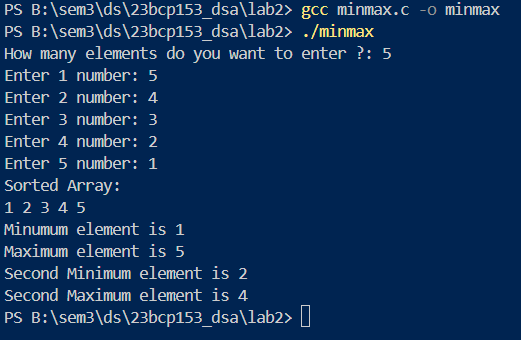
**arr[j + 1] = key;**

**}**

**return;**

**}**

### **Output:**



### **Time Complexity:**

**Worst Case:**

* **Explanation:** Occurs when the array is sorted in reverse order. For each element, the algorithm might need to compare it with every element in the sorted portion, leading to quadratic time complexity.

**Average Case:**

* **Explanation:** On average, the algorithm performs a quadratic number of comparisons and shifts, as elements are not uniformly distributed, requiring significant comparisons and shifts.

**Best Case:**

* **Explanation:** Occurs when the array is already sorted. The algorithm only needs to make a single pass through the array, performing a constant amount of work for each element as no elements need to be shifted.

Same as insertion sort as we are performing insertion sort here.

## **EXPERIMENT 2**

### **Aim:**

1. Create a structure Student in C with student name, student roll number and student address as its data members. Create the variable of type student and print the values.
2. Modify the above program to implement arrays of structure. Create an array of 5 students and print their values.
3. Create a structure Organization with organization name and organization ID as its data members. Next, create another structure Employee that is nested in structure Organization with employee ID, employee salary and employee name as its data members. Write a program in such a way that there are two organizations and each of these contains two employees.

### **Theory:**

* **Definition:** A struct is a user-defined data type in C that groups related variables of different types into a single unit.
* **Members:** Each variable within a struct is called a member and can be of any data type, including other structs.
* **Usage:** Structures help organize complex data, encapsulating it in a manageable format.
* **Advantages:** They enhance code readability and maintainability by logically grouping related data.
* **Modeling:** Structures are useful for representing real-world entities and aligning data in memory efficiently.

#### **Structure:**

**Code:**

#include <stdio.h>

#include <string.h>

**typedef struct Student {**

**char name[30];**

**int roll\_no;**

**char address[100];**

**}**

Student;

int main(void)

{

Student s1;

printf("Enter the name of the student: ");

scanf("%s", &s1.name);

printf("Enter the roll no of the student: ");

scanf("%d", &s1.roll\_no);

printf("Enter the address of the student: ");

scanf("%s", &s1.address);

printf("%s\n", s1.name);

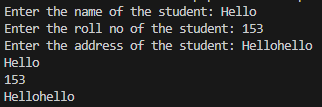
printf("%d\n", s1.roll\_no);

printf("%s\n", s1.address);

return 0;

}

**Output:**



#### **Array of Structure:**

**Code:**

#include <stdio.h>

#include <string.h>

**typedef struct Student {**

**char name[30];**

**int roll\_no;**

**char address[100];**

**}**

**Student;**

int main(void)

{

Student students[5];

int i;

for (i = 0; i < 5; i++) {

printf("Enter the name of student %d: ", i + 1);

scanf(" %s", students[i].name);

printf("Enter the roll no of student %d: ", i + 1);

scanf("%d", &students[i].roll\_no);

printf("Enter the address of student %d: ", i + 1);

scanf(" %s", students[i].address);

}

printf("\nStudent Details:\n");

for (i = 0; i < 5; i++) {

printf("Student %d:\n", i + 1);

printf("Name: %s\n", students[i].name);

printf("Roll No: %d\n", students[i].roll\_no);

printf("Address: %s\n\n", students[i].address);

}

return 0;

}

**Output:**

|  |  |
| --- | --- |
| **INPUT** | **OUTPUT** |
|  |  |

#### **Nested Array of Structure:**

**Code:**

#include <stdio.h>

#define ORGANIZATIONS 2

#define EMPLOYEES 2

**typedef struct Employee**

**{**

**char name[30];**

**float salary;**

**int id;**

**}**

**Employee;**

**typedef struct Organization {**

**int name[30];**

**int id;**

**Employee employees[2];**

**}**

**Organization;**

int main(void)

{

Organization organizations[2];

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

printf("Enter the name of organization %d: ", i + 1);

scanf("%s", organizations[i].name);

printf("Enter the ID of organization %d: ", i + 1);

scanf("%d", &organizations[i].id);

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

printf("\tEnter the name of employee %d in organization %d: ", j + 1, i + 1);

scanf("%s", organizations[i].employees[j].name);

printf("\tEnter the ID of employee %d in organization %d: ", j + 1, i + 1);

scanf("%d", &organizations[i].employees[j].id);

printf("\tEnter the salary of employee %d in organization %d: ", j + 1, i + 1);

scanf("%f", &organizations[i].employees[j].salary);

printf("\n");

}

}

printf("\nOrganization Details:\n");

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

printf("Organization %d:\n", i + 1);

printf("Name: %s\n", organizations[i].name);

printf("ID: %d\n", organizations[i].id);

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

printf("\tEmployee %d:\n", j + 1);

printf("\tName: %s\n", organizations[i].employees[j].name);

printf("\tID: %d\n", organizations[i].employees[j].id);

printf("\tSalary: %.2f\n", organizations[i].employees[j].salary);

printf("\n");

}

printf("\n");

}

return 0;

}

**Output:**

|  |  |
| --- | --- |
| **INPUT** | **OUTPUT** |
|  |  |

## **EXPERIMENT 3**

### **Aim:**

1. Write a program in C to implement arrays of pointers and pointers to arrays.
2. Write a program in C to implement pointers to structures.
3. Write a program in C to perform swapping of two numbers by passing addresses of the variables to the functions.

### **Theory (Arrays of Pointers and Pointers to Arrays):**

* An array of pointers is a collection where each element is a pointer to a variable rather than a direct value. In this scenario, instead of storing actual values, the array stores addresses of variables. For example, if we have several integer variables and want to create an array that holds the addresses of these variables, we can use an array of pointers. This allows us to access and modify the values of the original variables indirectly by dereferencing the pointers stored in the array. This approach is particularly useful when dealing with dynamic data structures or when we need to manage multiple variables efficiently.
* A pointer to an array is a pointer that points to the first element of an array. By using this pointer, we can traverse and access elements of the array through pointer arithmetic. Instead of directly accessing the array elements by their index, we can use the pointer to move across the array. This method is beneficial when passing arrays to functions, as it allows the function to work with the original array without needing to copy it. It also simplifies operations on arrays, especially when dealing with large data sets or when the array needs to be modified.

### **Program:**

#include <stdio.h>

// \* - "value at" operator

// & - "address of" operator

int main(void)

{

**int arr[10] = {11,23,32,45,54,63,72,81,90,10};**

**int \*ptrtoarr = &arr[0];**

    int arrlen = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < arrlen; i++)

    {

        printf("%d element: %d\n", i + 1, **\*(ptrtoarr + i)**);

    }

    printf("\n");

**int a = 112;**

**int b = 123;**

**int c = 34;**

**int d = 234;**

**int e = 123;**

**int \*arrofptr[5] = {&a, &b, &c, &d, &e};**

    int aoplen = sizeof(arrofptr) / sizeof(arrofptr[0]);

    for (int i = 0; i < aoplen; i++)

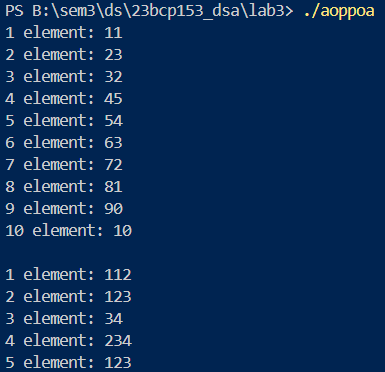
    {

        printf("%d element: %d\n", i + 1, **\*arrofptr[i]**);

    }

}

### **Output:**



### **Theory (Pointers to Structures):**

In C, pointers to structures allow us to efficiently manage and manipulate data stored within structures. A structure is a user-defined data type that groups different data types under a single name, making it easier to manage related data. When we use pointers with structures, we can directly access and modify the members of the structure through the pointer, which holds the address of the structure.

In the code below, a structure named Student is defined, which contains the student's name, roll number, and CGPA. A pointer to the structure ptr is declared and assigned the address of the structure variable s1. Using this pointer, the members of the structure can be accessed using the -> operator. This approach allows us to work with the structure efficiently, especially when passing it to functions or working with dynamic data.

### **Program:**

#include <stdio.h>

#include <string.h>

**typedef struct Student {**

**char name[30];**

**int roll\_no;**

**float cgpa;**

**}**

**Student;**

int main(void)

{

**Student s1;**

**Student \*ptr;**

**ptr = &s1;**

    printf("Enter Student's Name: ");

**fgets(ptr->name, sizeof(ptr->name), stdin);**

**ptr->name[strcspn(ptr->name, "\n")] = 0;**

    printf("Enter Student's Roll No.: ");

    scanf("%d", **&ptr->roll\_no**);

    printf("Enter CGPA of Student: ");

    scanf("%f", **&ptr->cgpa**);

    printf("\nStudent Details:\n");

    printf("Name: %s\n", **ptr->name**);

    printf("Age: %d\n", **ptr->roll\_no**);

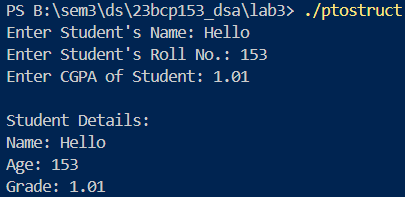
    printf("Grade: %.2f\n", **ptr->cgpa**);

    return 0;

}

// <https://stackoverflow.com/questions/2693776/removing-trailing-newline-character-from-fgets-input>

### **Output:**



### **Theory (Swapping by Addresses):**

In C, swapping two numbers by passing their addresses to a function is an example of call by reference. When we pass the addresses of variables to a function, the function can directly access and modify the values stored at those addresses. This is different from call by value, where a copy of the variable is passed, and changes made within the function do not affect the original variables.

In the code below, the swap function takes two pointers as parameters, which hold the addresses of the variables a and b. Inside the function, the values at these addresses are swapped using a temporary variable temp. This operation changes the actual values of a and b in the main function, demonstrating the effectiveness of call by reference in such scenarios.

### **Program:**

#include <stdio.h>

void swap(int \*x, int \*y);

int main(void)

{

    int a = 1;

    int b = 2;

    printf("Value of a is %d and value of b is %d\n", a, b);

**swap(&a, &b);**

    printf("Value of a is %d and value of b is %d\n", a, b);

    return 0;

}

**void swap(int \*x, int \*y)**

**{**

**int temp = \*x;**

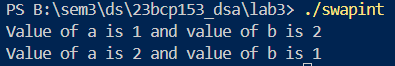
**\*x = \*y;**

**\*y = temp;**

**return;**

**}**

### **Output:**



## **EXPERIMENT 4**

### **Aim:**

* 1. Implement a stack using an array having following functionalities:
     1. isEmpty – to check if the stack if empty or not
     2. isFull – to check if the stack if full or not
     3. push – to insert the element into the stack
     4. pop – to delete an element from the stack
     5. print\_top – to print the top most element of the stack.

### **Theory (Stack using Array):**

* **Stack** is a linear data structure following the Last In, First Out (LIFO) principle.
* **Array-based stack** uses a fixed-size array to store elements, with an index variable top pointing to the most recently added element.
* **Operations**:

1. **Push (Insert)**: Adds an element to the top of the stack. If the stack is full (top == n-1), an overflow condition occurs.
2. **Pop (Delete)**: Removes and returns the top element from the stack. If the stack is empty (top == -1), an underflow condition occurs.
3. **isFull() and isEmpty()**: Check if the stack is full or empty before performing operations.
4. **printTop()**: Displays the top element without modifying the stack.

* **Time Complexity**: Each operation (push, pop, check) has O(1) time complexity, ensuring constant time for basic stack manipulations.

### **Program:**

#include <stdio.h>

#include <stdbool.h>

bool isEmpty();

bool isFull(int n);

void insert(int arr[], int ele);

void delete(int arr[]);

void printStack(int arr[]);

void printTop(int arr[]);

int top = -1;

int main(void)

{

    int n;

    printf("What is the Maximum number of elements you want in your stack: ");

    scanf("%d", &n);

    int arr[n];

    int ele;

    while(true)

    {

        int operation;

        printf("Press 1 to insert\nPress 2 to delete\nPress 3 to Print top element\nPress 0 to stop the program\n");

        scanf("%d", &operation);

        switch (operation)

        {

        case 1:

            printf("Enter the element to insert in the stack: ");

            scanf("%d", &ele);

            insert(arr, ele);

            break;

        case 2:

            delete(arr);

            break;

        case 3:

            printTop(arr);

            break;

        default:

            return 0;

            break;

        }

    }

    return 0;

}

bool isEmpty()

{

    if (top == -1)

    {

        return true;

    }

    return false;

}

bool isFull(int n)

{

    if (top == n - 1)

    {

        return true;

    }

    return false;

}

void insert(int arr[], int ele)

{

    if (isFull(top))

    {

        printf("Array is Full!\n");

        printStack(arr);

        return;

    }

    top++;

    arr[top] = ele;

    printStack(arr);

    return;

}

void delete(int arr[])

{

    if(isEmpty())

    {

        printf("Stack is Empty\n");

        return;

    }

    int popped = arr[top];

    top--;

    printf("Popped value: %d\n", popped);

    printStack(arr);

    return;

}

void printStack(int arr[])

{

    for (int i = top; i >= 0; i--)

    {

        printf("%d ", arr[i]);

    }

    printf("\n");

    return;

}

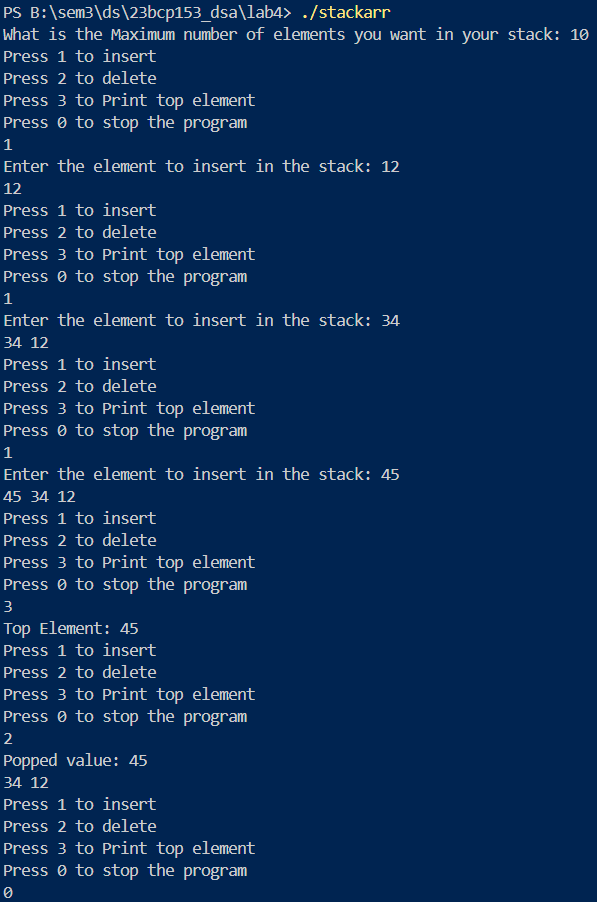
void printTop(int arr[])

{

    printf("Top Element: %d\n", arr[top]);

}

### **Output:**



### **Time Complexity:**

Each operation (push, pop, check) has time complexity, ensuring constant time for basic stack manipulations.

## **EXPERIMENT 5**

### **Aim:**

1. Write a program to evaluate the following given postfix expressions: a. 2 3 1 ∗ + 9 – Output: -4 b. 2 2 + 2 / 5 ∗ 7 + Output: 17
2. Convert the given infix expression into postfix expression using stack. Example- Input: 𝑎 + 𝑏 ∗(𝑐^𝑑 −𝑒)^(𝑓 +𝑔∗ℎ)−𝑖 Output: 𝑎𝑏𝑐𝑑^𝑒 − 𝑓𝑔ℎ ∗ +^∗+𝑖 –
3. Given an expression, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in the expression or not. Example: Input: exp = “[( )]{ }{[( )( )]( )}” Input: exp = “[( ])” Output: Balanced Output: Not Balanced

### **Theory (Evaluation of Postfix Expression):**

Postfix (Reverse Polish Notation) expressions place operators after operands, removing the need for parentheses. To evaluate:

1. **Scan left to right**.
2. **Push operands** onto a stack.
3. **When encountering an operator**, pop two operands, apply the operator, and push the result back.
4. **Final result** will be on the stack after processing the entire expression.

### **Program:**

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Stack

{

    int top;

    int capacity;

    int \*array;

} Stack;

Stack \*createStack(int n);

int isEmpty(Stack \*s);

int isFull(Stack \*s);

int pop(Stack \*s);

void push(Stack \*s, int value);

int evaluatePostfix(char \*postfix);

int performOperation(int a, int b, char op);

int main(void)

{

    char postfix[100];

    printf("Enter Postfix Expression: ");

    fgets(postfix, sizeof(postfix), stdin);

    size\_t len = strlen(postfix);

    if (len > 0 && postfix[len - 1] == '\n')

    {

        postfix[len - 1] = '\0';

    }

    int result = evaluatePostfix(postfix);

    printf("The result is: %d\n", result);

    return 0;

}

Stack \*createStack(int n)

{

    Stack \*s = (Stack \*)malloc(sizeof(Stack));

    s->capacity = n;

    s->top = -1;

    s->array = (int \*)malloc(s->capacity \* sizeof(int));

    return s;

}

int isEmpty(Stack \*s)

{

    return s->top == -1;

}

int isFull(Stack \*s)

{

    return s->top == s->capacity - 1;

}

int pop(Stack \*s)

{

    if (isEmpty(s))

    {

        printf("Stack is Empty\n");

        return -1;

    }

    return s->array[s->top--];

}

void push(Stack \*s, int value)

{

    if (isFull(s))

    {

        printf("Stack Overflow!\n");

        return;

    }

    s->array[++s->top] = value;

}

int evaluatePostfix(char \*postfix)

{

    int strlength = strlen(postfix);

    Stack \*mystack = createStack(strlength);

    for (int i = 0; postfix[i] != '\0'; i++)

    {

        char ch = postfix[i];

        if (isdigit(ch))

        {

            push(mystack, ch - '0');

            // value of 0 in ASCII is 48

        }

        else if (ch == ' ')

        {

            continue;

        }

        else

        {

            int val2 = pop(mystack);

            int val1 = pop(mystack);

            int result = performOperation(val1, val2, ch);

            push(mystack, result);

        }

    }

    int finalResult = pop(mystack);

    free(mystack->array);

    free(mystack);

    return finalResult;

}

int performOperation(int a, int b, char op)

{

    switch (op)

    {

    case '+':

        return a + b;

    case '-':

        return a - b;

    case '\*':

        return a \* b;

    case '/':

        return a / b;

    default:

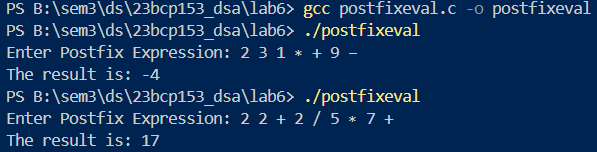
        printf("Invalid operator encountered: %c\n", op);

        return 0;

    }

}

### **Output:**



### **Theory (Infix to Postfix):**

Infix notation (e.g., A + B) requires parentheses for precedence, while postfix (e.g., AB+) doesn’t. To convert:

1. **Scan the infix expression** left to right.
2. **Push operators** to a stack and **add operands** to the output.
3. **Handle parentheses**: Push (, pop to the output until ) is found.
4. **Pop remaining operators** to the output at the end.

### **Program:**

// Also always follow alphabetical order

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Stack

{

    int top;

    int capacity;

    char \*array;

} Stack;

Stack \*createStack(int n);

int isEmpty(Stack \*s);

int isFull(Stack \*s);

char pop(Stack \*s);

void push(Stack \*s, int value);

char peek(Stack \*s);

void removeSpaces(const char \*input, char \*output);

void infixToPostfix(char \*infix, char \*postfix);

int precedence(char ch);

int isOperator(char ch);

int main(void)

{

    char inputstr[100];

    char infix[100];

    char postfix[100];

    printf("Enter Your Operation String: ");

    fgets(inputstr, sizeof(inputstr), stdin);

    removeSpaces(inputstr, infix);

    size\_t len = strlen(infix);

    if (len > 0 && infix[len - 1] == '\n') {

        infix[len - 1] = '\0';

    }

    infixToPostfix(infix, postfix);

    printf("Your Postfix Expression:-\n%s", postfix);

    return 0;

}

Stack \*createStack(int n)

{

    Stack \*s = (Stack \*)malloc(sizeof(Stack));

    s->capacity = n;

    s->top = -1;

    s->array = (char \*)malloc(s->capacity \* sizeof(int));

    return s;

}

char pop(Stack \*s)

{

    if (isEmpty(s))

    {

        // Underflow

        printf("Stack is Empty\n");

        return '\0';

    }

    char popped = s->array[s->top];

    s->top--;

    return popped;

}

void push(Stack \*s, int value)

{

    // Overflow (as we do in algo in class)

    if (isFull(s))

    {

        printf("Stack is Full!\n");

        return;

    }

    s->top++;

    s->array[s->top] = value;

    return;

}

int isFull(Stack \*s)

{

    return s->top == s->capacity - 1;

}

int isEmpty(Stack \*s)

{

    return s->top == -1;

}

void removeSpaces(const char \*input, char \*output)

{

    // Index for output string

    int j = 0;

    for (int i = 0; input[i] != '\0'; i++)

    {

        if (input[i] != ' ')

        {

            output[j++] = input[i];

        }

    }

    // Null terminating o/p string

    output[j] = '\0';

}

void infixToPostfix(char \*infix, char \*postfix)

{

    int strlength = strlen(infix);

    // printf("%d\n", strlength);

    // printf("%s", infix);

    Stack \*mystack = createStack(strlength);

    int i, j = 0;

    push(mystack, '(');

    // Double quotes here in strcat are necessary becasue function takes string type argument only

    strcat(infix, ")");

    for (i = 0; infix[i] != '\0'; i++)

    {

        char ch = infix[i];

        if (isalnum(ch))

        {

            postfix[j++] = ch;

        }

        else if (ch == '(')

        {

            push(mystack, ch);

        }

        else if (isOperator(ch))

        {

            while (isOperator(peek(mystack)) && precedence(peek(mystack)) >= precedence(ch))

            {

                postfix[j++] = pop(mystack);

            }

            push(mystack, ch);

        }

        else if (ch == ')')

        {

            while (peek(mystack) != '(')

            {

                postfix[j++] = pop(mystack);

            }

            // Then pop '('

            pop(mystack);

        }

    }

    postfix[j] = '\0';

    free(mystack->array);

    free(mystack);

}

char peek(Stack \*s)

{

    if (isEmpty(s))

    {

        return '\0';

    }

    return s->array[s->top];

}

int precedence(char ch)

{

    if (ch == '+' || ch == '-')

    {

        return 1;

    }

    if (ch == '\*' || ch == '/')

    {

        return 2;

    }

    if (ch == '^')

    {

        return 3;

    }

    return 0;

}

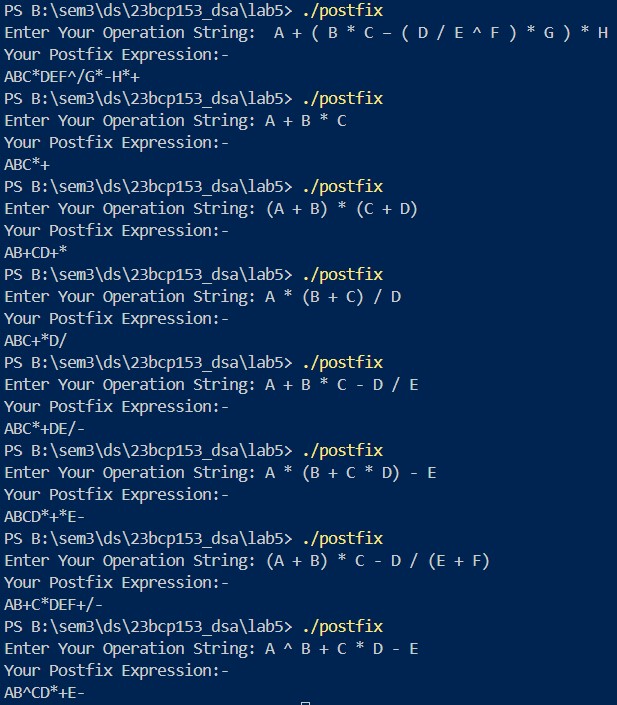
int isOperator(char ch)

{

    return (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^');

}

### **Output:**



### **Theory (Balanced Brackets):**

Check if all brackets ((), {}, []) are correctly paired and nested. Steps:

1. **Push opening brackets** onto a stack.
2. **Pop the stack** when encountering a closing bracket; ensure it matches the most recent opening bracket.
3. If the stack is empty at the end, the brackets are **balanced**.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Stack

{

    int top;

    int capacity;

    char \*array;

} Stack;

Stack \*createStack(int n);

int isEmpty(Stack \*s);

void push(Stack \*s, char value);

char pop(Stack \*s);

char peek(Stack \*s);

int isMatchingPair(char left, char right);

int isBalanced(char \*exp);

int main(void)

{

    char exp[100];

    printf("Enter expression: ");

    fgets(exp, sizeof(exp), stdin);

    size\_t len = strlen(exp);

    if (len > 0 && exp[len - 1] == '\n')

    {

        exp[len - 1] = '\0';

    }

    if (isBalanced(exp))

        printf("Balanced\n");

    else

        printf("Not Balanced\n");

    return 0;

}

Stack \*createStack(int n)

{

    Stack \*s = (Stack \*)malloc(sizeof(Stack));

    s->capacity = n;

    s->top = -1;

    s->array = (char \*)malloc(s->capacity \* sizeof(char));

    return s;

}

int isEmpty(Stack \*s)

{

    return s->top == -1;

}

void push(Stack \*s, char value)

{

    s->array[++s->top] = value;

}

char pop(Stack \*s)

{

    if (isEmpty(s))

    {

        printf("Stack Underflow\n");

        return '\0';

    }

    return s->array[s->top--];

}

char peek(Stack \*s)

{

    if (isEmpty(s))

    {

        return '\0';

    }

    return s->array[s->top];

}

int isMatchingPair(char left, char right)

{

    return (left == '(' && right == ')') ||

           (left == '{' && right == '}') ||

           (left == '[' && right == ']');

}

int isBalanced(char \*exp)

{

    int n = strlen(exp);

    Stack \*stack = createStack(n);

    for (int i = 0; exp[i] != '\0'; i++)

    {

        char ch = exp[i];

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

        {

            push(stack, ch);

        }

        else if (ch == ')' || ch == '}' || ch == ']')

        {

            if (isEmpty(stack) || !isMatchingPair(pop(stack), ch))

            {

                free(stack->array);

                free(stack);

                return 0; // Not Balanced

            }

        }

    }

    int balanced = isEmpty(stack);

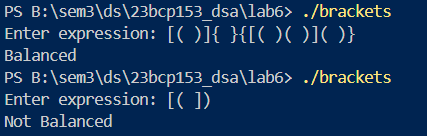
    free(stack->array);

    free(stack);

    return balanced;

}

### **Output:**



## **EXPERIMENT 6**

### **Aim:**

1. Implement the following functionalities of the Circular queue using Arrays:
   1. isFull – to check if the queue is full or not.
   2. isEmpty – to check if the queue is empty or not.
   3. enqueue – to insert the element in the queue.
   4. dequeue – to delete the element from the queue.
   5. front and rear – to print the front and rear element of the queue.

### **Theory (Circular Queue using Array):**

* **Queue Representation**: The queue is implemented as an array with fixed size, and the elements are added and removed in a circular manner.
* **Front and Rear Pointers**:
* front: Points to the element at the front of the queue.
* rear: Points to the most recently added element.
* Both are initialized to -1, indicating the queue is empty.
* **Circular Nature**:
* When rear reaches the end of the array, it wraps around to the beginning  
  .
* Similarly, front increments circularly after each removal   
  .
* **isEmpty**: The queue is empty if .
* **isFull**: The queue is full if.
* **enqueue Operation**:
* If the queue is not full, the element is inserted at rear, and rear is incremented circularly.
* If the queue was empty, both front and rear are set to 0 to handle the first insertion.
* **dequeue Operation**:
* Removes the element at front and circularly increments front.
* If front equals rear after removal, both pointers are reset to -1 to mark the queue as empty.

### **Program:**

#include <stdio.h>

#include <stdbool.h>

#define SIZE 10

int queue[SIZE];

int front = -1;

int rear = -1;

bool isEmpty();

bool isFull();

void enqueue(int ele);

void dequeue();

void printQueue();

int main(void)

{

    printQueue();

    enqueue(18);

    enqueue(27);

    enqueue(36);

    enqueue(45);

    printQueue();

    enqueue(54);

    enqueue(63);

    enqueue(72);

    enqueue(81);

    enqueue(90);

    enqueue(99);

    enqueue(99);

    printQueue();

    dequeue();

    dequeue();

    dequeue();

    printQueue();

    enqueue(999);

    enqueue(153);

    // Roll no. above

    printQueue();

    return 0;

}

bool isEmpty()

{

    return (front == -1);

}

bool isFull()

{

    return ((rear + 1) % SIZE == front);

}

void enqueue(int ele)

{

    if (isFull())

    {

        printf("Queue is FULL!\n");

        return;

    }

    if (isEmpty())

    {

        front = rear = 0;

    }

    else

    {

        rear = (rear + 1) % SIZE;

    }

    queue[rear] = ele;

    printf("Inserted: %d\n", ele);

}

void dequeue()

{

    if (isEmpty())

    {

        printf("Queue is EMPTY!\n");

        return;

    }

    printf("Deleted: %d\n", queue[front]);

    if (front == rear)

    {

        front = rear = -1;

    }

    else

    {

        front = (front + 1) % SIZE;

    }

}

void printQueue()

{

    if (isEmpty())

    {

        printf("Queue is EMPTY!\n");

        return;

    }

    printf("Queue: ");

    int i = front;

    while (i != rear)

    {

        printf("%d ", queue[i]);

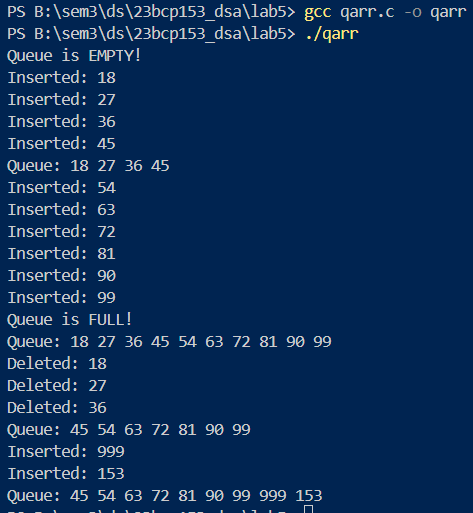
        i = (i + 1) % SIZE;

    }

    printf("%d\n", queue[rear]);

}

### **Output:**



### **Time Complexity:**

The time complexity for **enqueue** and **dequeue** operations in a circular queue is , as both involve constant-time operations to add or remove an element, regardless of the queue's size.

## **EXPERIMENT 7**

### **Aim:**

1. Write a program to insert a new node into the linked list. A node can be added into the linked list using three ways: [Write code for all the three ways.]
2. At the front of the list
3. After a given node
4. At the end of the list.
5. Write a program to delete a node from the linked list. A node can be deleted from the linked list using three ways: [Write code for all the three ways.]
6. Delete from the beginning
7. Delete from the end
8. Delete from the middle.
9. Write a program that takes two sorted lists as inputs and merge them into one sorted list. For example, if the first linked list A is 5 =>10 =>15, and the other linked list B is 2 => 3 => 20, then output should be 2 => 3 => 5 => 10 => 15 => 20.
10. Implement the circular linked list and perform the operation of traversal on it. In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again.
11. Implement the doubly linked list and perform the deletion and/ or insertion operation on it. Again, you can perform insertion deletion according to the three ways as given above. Implement all of them according to availability of time.

### **Theory (Linked List Insert):**

In a **singly linked list**, each node contains two parts: **data (info)** and a **link** to the next node. The insertion can be performed at three different positions:

1. **Insert at Beginning:**
   * Create a new node.
   * Set the new node's link to the current head (start).
   * Update the head to point to the new node.
2. **Insert After a Node:**
   * Traverse the list to find the node with previnfo.
   * Create a new node.
   * Set the new node's link to the next node.
   * Update the found node's link to the new node.
3. **Insert at End:**
   * Traverse to the last node.
   * Create a new node with its link set to NULL.
   * Update the last node's link to point to the new node.

### **Time Complexity:**

* **Insert at Beginning:** — The operation is performed directly without traversal.
* **Insert After a Node:** — The list needs to be traversed to find the node with the specified value.
* **Insert at End:** — The list is traversed to the last node before inserting.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

typedef struct List

{

    int info;

    struct List \*link;

} List;

List \*insertbeg(List \*start, int info);

void printlist(List \*start);

List \*insertafter(List \*start, int previnfo, int info);

List \*insertend(List \*start, int info);

void freelist(List \*start);

int main(void)

{

    List \*start = NULL;

    start = insertbeg(start, 10);

    printlist(start);

    start = insertbeg(start, 20);

    printlist(start);

    start = insertbeg(start, 30);

    printlist(start);

    start = insertafter(start, 20, 12);

    printlist(start);

    start = insertend(start, 45);

    printlist(start);

    start = insertend(start, 72);

    printlist(start);

    freelist(start);

    return 0;

}

List \*insertbeg(List \*start, int info)

{

    List \*node = (List \*)malloc(sizeof(List));

    if (node == NULL)

    {

        printf("Unable to allocate memory for new node\n");

        return start;

    }

    node->info = info;

    node->link = start;

    return node;

}

List \*insertafter(List \*start, int previnfo, int info)

{

    List \*node = (List \*)malloc(sizeof(List));

    if (node == NULL)

    {

        printf("Unable to allocate memory for new node\n");

        return start;

    }

    List \*current = start;

    while (current->info != previnfo && current != NULL)

    {

        current = current->link;

    }

    if (current->info == previnfo)

    {

        node->info = info;

        node->link = current->link;

        current->link = node;

    }

    else

    {

        printf("Node with value %d not found!\n", previnfo);

        return start;

    }

    return start;

}

List \*insertend(List \*start, int info)

{

    List \*current = start;

    List \*node = (List \*)malloc(sizeof(List));

    if (node == NULL)

    {

        printf("Unable to allocate memory for new node\n");

        return start;

    }

    node->info = info;

    node->link = NULL;

    if (start == NULL)

    {

        return node;

    }

    while (current->link != NULL)

    {

        current = current->link;

    }

    current->link = node;

    return start;

}

void printlist(List \*start)

{

    while (start != NULL)

    {

        printf("%d", start->info);

        start = start->link;

        if (start != NULL)

        {

            printf("->");

        }

    }

    printf("->NULL\n");

}

void freelist(List \*start)

{

    List \*current = start;

    if (start == NULL)

    {

        return;

    }

    while (current != NULL)

    {

        List \*node = current;

        current = current->link;

        // printf("removed %d from memory\t", current->info);

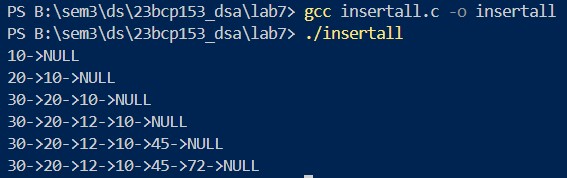
        free(node);

    }

    return;

}

### **Output:**



### **Theory (Linked List Delete):**

This code handles three types of deletions in a singly linked list:

1. **Delete from the Beginning (deletebegthis)**:  
   The first node is removed by updating the start pointer to the second node, and the first node’s memory is freed.
2. **Delete a Specific Node (deletenodethis)**:  
   The list is traversed to find the node with a given value (info). Once found, the previous node is linked to the next one, and the target node is freed.
3. **Delete from the End (deleteendthis)**:  
   The last node is removed by traversing to the second-to-last node, setting its link to NULL, and freeing the last node.

### **Time Complexity:**

* **Delete from Beginning**:
* **Delete a Specific Node**:
* **Delete from End**:

### **Program:**

#include <stdio.h>

#include <stdlib.h>

#include "mylistlib.h"

// compile command:

// gcc deletelist.c mylistlib.c -o deletelist

List \*deletebegthis(List \*start);

List \*deletenodethis(List \*start, int info);

List \*deleteendthis(List \*start);

int main(void)

{

    List \*start = NULL;

    start = insertbeg(start, 10);

    start = insertbeg(start, 20);

    start = insertbeg(start, 30);

    start = insertafter(start, 20, 12);

    start = insertend(start, 45);

    start = insertend(start, 72);

    printlist(start);

    start = deletebegthis(start);

    printlist(start);

    start = deletenodethis(start, 45);

    printlist(start);

    start = deleteendthis(start);

    printlist(start);

    start = deleteend(start);

    printlist(start);

    freelist(start);

    return 0;

}

List \*deletebegthis(List \*start)

{

    if (start == NULL)

    {

        printf("List is Empty\n");

        return start;

    }

    // To Avoid Memory Leaks my friend

    List \*temp = start;

    start = start->link;

    free(temp);

    return start;

}

List \*deletenodethis(List \*start, int info)

{

    if (start == NULL)

    {

        printf("List is Empty\n");

        return start;

    }

    List \*current = start;

    if (current->info == info)

    {

        start = current->link;

        free(current);

        return start;

    }

    while (current->link->info != info && current->link != NULL)

    {

        current = current->link;

    }

    if (current->link == NULL)

    {

        printf("Node with value %d was not found!\n", info);

        return start;

    }

    List \*temp = current->link;

    // Bypassing the node

    current->link = current->link->link;

    free(temp);

    return start;

}

List \*deleteendthis(List \*start)

{

    // If only one node

    if (start->link == NULL)

    {

        free(start);

        return NULL;

    }

    List \*current = start;

    while (current->link->link != NULL)

    {

        current = current->link;

    }

    // Freeing last node my friend - prevent memory leaks

    // always check for memory leaks

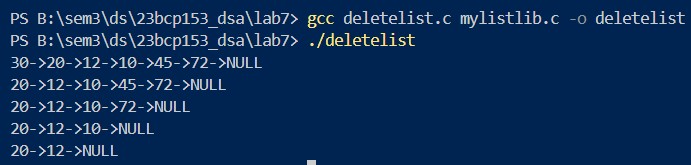
    free(current->link);

    current->link = NULL;

    return start;

}

### **Output:**



### **Theory (Merging Two Sorted Lists):**

The function mergeSortedlist takes two sorted linked lists (A and B) and merges them into a single sorted linked list. It compares the nodes of both lists, attaches the smaller node to the merged list, and continues until all nodes from both lists are merged. A dummy node is used to simplify the merging process.

Steps:

1. If either list is empty, the other list is returned.
2. A dummy node is used to build the merged list.
3. The nodes from A and B are compared and linked accordingly.
4. Any remaining nodes from A or B are appended after one list is exhausted.

### **Time Complexity:**

**Merging two sorted lists**: , where n is the number of nodes in list A and m is the number of nodes in list B.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

#include "mylistlib.h"

List \*mergeSortedlist(List \*A, List \*B);

int main(void)

{

    List \*A = NULL;

    List \*B = NULL;

    A = insertend(A, 5);

    A = insertend(A, 10);

    A = insertend(A, 15);

    B = insertend(B, 2);

    B = insertend(B, 3);

    B = insertend(B, 20);

    printlist(A);

    printlist(B);

    List \*merged = NULL;

    merged = mergeSortedlist(A, B);

    printlist(merged);

    freelist(merged);

    freelist(A);

    freelist(B);

    return 0;

}

List \*mergeSortedlist(List \*A, List \*B)

{

    if (A == NULL)

    {

        return B;

    }

    if (B == NULL)

    {

        return A;

    }

    List \*dummy = (List \*)malloc(sizeof(List));

    if (dummy == NULL)

    {

        printf("Memory allocation failed\n");

        return NULL;

    }

    dummy->link = NULL;

    List \*current = dummy;

    while (A != NULL && B != NULL)

    {

        if (A->info <= B->info)

        {

            current->link = A;

            A = A->link;

        }

        else

        {

            current->link = B;

            B = B->link;

        }

        current = current->link;

    }

    if (A != NULL)

    {

        current->link = A;

    }

    if (B != NULL)

    {

        current->link = B;

    }

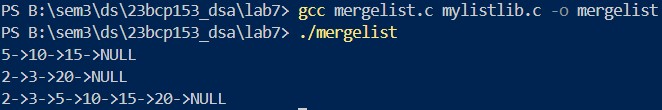
    List \*merged = dummy->link;

    free(dummy);

    return merged;

}

### **Output:**



### **Theory (Circular Linked List):**

In a **circular linked list**, the last node points back to the first node, forming a loop. Unlike a standard linked list, it doesn't end with NULL.

1. **Insertion at Beginning** (insertbegcircthis): A new node is inserted at the beginning, and the last node's link is updated to point to the new start.
2. **Insertion at End** (insertendcircthis): A new node is inserted at the end, and its link is set to point to the head of the list, maintaining the circular nature.
3. **Printing the List** (printlistcircthis): It traverses the list and prints until it circles back to the first node.
4. **Freeing the List** (freelistcircthis): It frees all nodes while maintaining the circular structure until all nodes are removed.

### **Time Complexity:**

* **Insertion at Beginning**: due to the traversal required to update the last node's link.
* **Insertion at End**: as it requires traversal to the last node.
* **Printing**: , where n is the number of nodes in the circular list.
* **Freeing**: , as all nodes must be visited and freed.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

#include "mylistlib.h"

List \*insertbegcircthis(List \*start, int info);

List \*insertendcircthis(List \*start, int info);

void printlistcircthis(List \*start);

void freelistcircthis(List \*start);

int main(void)

{

    List \*start = NULL;

    start = insertbegcircthis(start, 10);

    printlistcircthis(start);

    start = insertendcircthis(start, 30);

    printlistcircthis(start);

    start = insertendcircthis(start, 20);

    printlistcircthis(start);

    start = insertbegcircthis(start, 45);

    printlistcircthis(start);

    freelistcircthis(start);

    return 0;

}

List \*insertbegcircthis(List \*start, int info)

{

    List \*node = (List \*)malloc(sizeof(List));

    if (node == NULL)

    {

        printf("Unable to allocate memory for new node\n");

        return start;

    }

    node->info = info;

    if (start == NULL)

    {

        node->link = node;

        return node;

    }

    List \*current = start;

    while (current->link != start)

    {

        current = current->link;

    }

    current->link = node;

    node->link = start;

    return node;

}

List \*insertendcircthis(List \*start, int info)

{

    List \*node = (List \*)malloc(sizeof(List));

    if (node == NULL)

    {

        printf("Unable to allocate memory for new node\n");

        return start;

    }

    node->info = info;

    if (start == NULL)

    {

        node->link = node;

        return node;

    }

    List \*current = start;

    while (current->link != start)

    {

        current = current->link;

    }

    current->link = node;

    node->link = start;

    return start;

}

void printlistcircthis(List \*start)

{

    if (start == NULL)

    {

        printf("The list is empty.\n");

        return;

    }

    List \*current = start;

    do

    {

        printf("%d->", current->info);

        current = current->link;

    }

    while (current != start);

    printf("(back to %d)\n", start->info);

}

void freelistcircthis(List \*start)

{

    if (start == NULL)

    {

        return;

    }

    List \*current = start;

    List \*node;

    do

    {

        node = current->link;

        printf("Removing %d from memory\t", current->info);

        free(current);

        current = node;

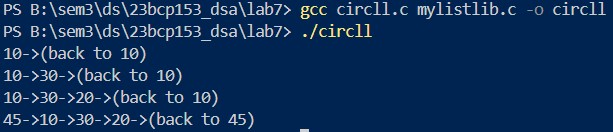
    }

    while (current != start);

    return;

}

### **Output:**



### **Theory (Doubly Linked List):**

A doubly linked list allows traversal in both directions using two pointers (prev and next). Insertion at the end involves traversing to the last node and linking the new node, while deletion from the beginning removes the first node and adjusts the pointers of the new start node.

### **Time Complexity:**

* Insertion at the end: (for traversing to the last node).
* Deletion from the beginning: (adjusting pointers).
* Traversal/printing: .

### **Program:**

#include <stdio.h>

#include <stdlib.h>

typedef struct DoubListthis {

    int info;

    struct DoubListthis \*prev;

    struct DoubListthis \*next;

} DoubListthis;

DoubListthis \*insertendDoubthis(DoubListthis \*start, int info);

DoubListthis \*deletebegDoubthis(DoubListthis \*start);

void printlistDoubthis(DoubListthis \*start);

int main(void)

{

    DoubListthis \*start = NULL;

    start = insertendDoubthis(start, 45);

    printlistDoubthis(start);

    start = insertendDoubthis(start, 63);

    printlistDoubthis(start);

    start = insertendDoubthis(start, 81);

    printlistDoubthis(start);

    start = insertendDoubthis(start, 90);

    printlistDoubthis(start);

    start = deletebegDoubthis(start);

    printlistDoubthis(start);

    start = deletebegDoubthis(start);

    printlistDoubthis(start);

    start = deletebegDoubthis(start);

    printlistDoubthis(start);

    return 0;

}

DoubListthis \*insertendDoubthis(DoubListthis \*start, int info)

{

    DoubListthis \*node = (DoubListthis \*)malloc(sizeof(DoubListthis));

    if (node == NULL)

    {

        printf("Unable to allocate memory for new node\n");

        return start;

    }

    node->info = info;

    node->prev = NULL;

    node->next = NULL;

    if (start == NULL)

    {

        return node;

    }

    DoubListthis \*current = start;

    while (current->next != NULL)

    {

        current = current->next;

    }

    current->next = node;

    node->prev = current;

    return start;

}

DoubListthis \*deletebegDoubthis(DoubListthis \*start)

{

    if (start == NULL)

    {

        printf("List is Empty\n");

        return start;

    }

    DoubListthis \*temp = start;

    if (start->next == NULL)

    {

        free(start);

        return NULL;

    }

    start = start->next;

    start->prev = NULL;

    free(temp);

    return start;

}

void printlistDoubthis(DoubListthis \*start)

{

    DoubListthis \*current = start;

    while (current != NULL)

    {

        printf("%d", current->info);

        current = current->next;

        if (current != NULL)

        {

            printf("<->");

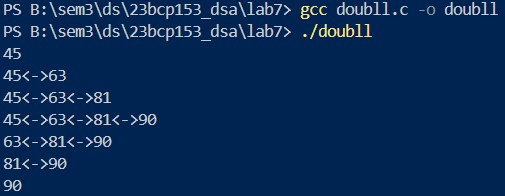
        }

    }

    printf("\n");

}

### **Output:**



## **EXPERIMENT 8**

### **Aim:**

1. Implement the Binary Tree and perform any one of the following three types of traversals: (Implement iterative method of traversal, not recursive) a. Pre-order Traversal b. In-order Traversal c. Post-order Traversal

### **Theory (Inorder Traversal):**

**Inorder Traversal (Left, Root, Right)**: In iterative inorder traversal, we use a stack to simulate the recursive process. We traverse the left subtree first, push nodes onto the stack, and process each node after visiting its left child, before moving to the right child.

### **Time Complexity:**

- We visit each node exactly once, where is the number of nodes in the tree.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

// left root right - inorder

typedef struct Tree {

    int info;

    struct Tree \*left;

    struct Tree \*right;

} Tree;

typedef struct Stacktree

{

    int top;

    int capacity;

    Tree \*\*array;

} Stacktree;

Tree \*makeTnode(int info);

void printInorder(Tree \*root);

void freetree(Tree \*root);

void printInorderiter(Tree \*root);

Stacktree \*createStackt(int n);

Tree \*popt(Stacktree \*s);

void pusht(Stacktree \*s, Tree \*node);

int isFullt(Stacktree \*s);

int isEmptyt(Stacktree \*s);

Tree \*peekt(Stacktree \*s);

int main(void)

{

    Tree \*A = makeTnode(1);

    Tree \*B = makeTnode(2);

    Tree \*C = makeTnode(3);

    Tree \*D = makeTnode(4);

    Tree \*E = makeTnode(5);

    Tree \*F = makeTnode(6);

    Tree \*G = makeTnode(7);

    Tree \*H = makeTnode(8);

    Tree \*I = makeTnode(9);

    A->left = B;

    A->right = C;

    B->left = D;

    B->right = E;

    C->left = F;

    D->left = G;

    D->right = H;

    G->left = I;

    printf("Inorder:\n");

    printInorder(A);

    printf("\n");

    printInorderiter(A);

    freetree(A);

    return 0;

}

void printInorder(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    printInorder(root->left);

    printf("%d ", root->info);

    printInorder(root->right);

}

// Make these below type of dynamic functions later

// Tree \*insert()

// {

// }

Tree \*makeTnode(int info)

{

    Tree \*node = (Tree \*)malloc(sizeof(Tree));

    if (node == NULL)

    {

        printf("Memory Allocation Failed!\n");

        return NULL;

    }

    node->info = info;

    node->left = NULL;

    node->right = NULL;

    return node;

}

void freetree(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    freetree(root->left);

    freetree(root->right);

    free(root);

}

void printInorderiter(Tree \*root)

{

    // use array based stack here for simplicity - remove this comment - don't use libraires - straight away ds and implementation

    if (root == NULL)

    {

        return;

    }

    Stacktree \*mystack = createStackt(9);

    Tree \*current = root;

    printf("Inorder Traversal iterative:\n");

    while (current != NULL || !isEmptyt(mystack))

    {

        while (current != NULL)

        {

            pusht(mystack, current);

            current = current->left;

        }

        current = popt(mystack);

        printf("%d ", current->info);

        current = current->right;

    }

    free(mystack->array);

    free(mystack);

}

Stacktree \*createStackt(int n)

{

    Stacktree \*s = (Stacktree \*)malloc(sizeof(Stacktree));

    s->capacity = n;

    s->top = -1;

    s->array = (Tree \*\*)malloc(s->capacity \* sizeof(Tree \*));

    return s;

}

Tree \*popt(Stacktree \*s)

{

    if (isEmptyt(s))

    {

        // Underflow

        printf("Stack is Empty\n");

        return NULL;

    }

    Tree \*popped = s->array[s->top];

    s->top--;

    return popped;

}

void pusht(Stacktree \*s, Tree \*node)

{

    // Overflow (as we do in algo in class)

    if (isFullt(s))

    {

        printf("Stack is Full!\n");

        return;

    }

    s->top++;

    s->array[s->top] = node;

    return;

}

int isFullt(Stacktree \*s)

{

    return s->top == s->capacity - 1;

}

int isEmptyt(Stacktree \*s)

{

    return s->top == -1;

}

Tree \*peekt(Stacktree \*s)

{

    if (isEmptyt(s))

    {

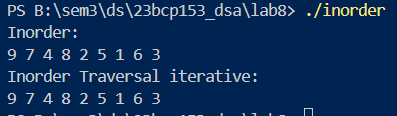
        return NULL;

    }

    return s->array[s->top];

}

### **Output:**



### **Theory (Preorder Traversal):**

**Preorder Traversal (Root, Left, Right)**: In iterative preorder traversal, a stack is used to maintain nodes, starting with the root. The root is processed first, then the right child is pushed, followed by the left child, ensuring left is processed before right.

### **Time Complexity:**

— Each node is visited once.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

// root left right - Preorder

typedef struct Tree {

    int info;

    struct Tree \*left;

    struct Tree \*right;

} Tree;

typedef struct Stacktree

{

    int top;

    int capacity;

    Tree \*\*array;

} Stacktree;

Tree \*makeTnode(int info);

void printPreorder(Tree \*root);

void freetree(Tree \*root);

void printPreorderiter(Tree \*root);

Stacktree \*createStackt(int n);

Tree \*popt(Stacktree \*s);

void pusht(Stacktree \*s, Tree \*node);

int isFullt(Stacktree \*s);

int isEmptyt(Stacktree \*s);

Tree \*peekt(Stacktree \*s);

int main(void)

{

    Tree \*A = makeTnode(1);

    Tree \*B = makeTnode(2);

    Tree \*C = makeTnode(3);

    Tree \*D = makeTnode(4);

    Tree \*E = makeTnode(5);

    Tree \*F = makeTnode(6);

    Tree \*G = makeTnode(7);

    Tree \*H = makeTnode(8);

    Tree \*I = makeTnode(9);

    A->left = B;

    A->right = C;

    B->left = D;

    B->right = E;

    C->left = F;

    D->left = G;

    D->right = H;

    G->left = I;

    printf("Preorder:\n");

    printPreorder(A);

    printf("\n");

    printPreorderiter(A);

    freetree(A);

    return 0;

}

void printPreorder(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    printf("%d ", root->info);

    printPreorder(root->left);

    printPreorder(root->right);

}

Tree \*makeTnode(int info)

{

    Tree \*node = (Tree \*)malloc(sizeof(Tree));

    if (node == NULL)

    {

        printf("Memory Allocation Failed!\n");

        return NULL;

    }

    node->info = info;

    node->left = NULL;

    node->right = NULL;

    return node;

}

void freetree(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    freetree(root->left);

    freetree(root->right);

    free(root);

}

void printPreorderiter(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    Stacktree \*mystack = createStackt(9);

    Tree \*current = root;

    printf("Preorder Traversal Iterative:\n");

    pusht(mystack, root);

    while (!isEmptyt(mystack))

    {

        Tree \*current = popt(mystack);

        printf("%d ", current->info);

        if (current->right != NULL)

        {

            pusht(mystack, current->right);

        }

        if (current->left != NULL)

        {

            pusht(mystack, current->left);

        }

    }

    printf("\n");

    free(mystack->array);

    free(mystack);

}

Stacktree \*createStackt(int n)

{

    Stacktree \*s = (Stacktree \*)malloc(sizeof(Stacktree));

    s->capacity = n;

    s->top = -1;

    s->array = (Tree \*\*)malloc(s->capacity \* sizeof(Tree \*));

    return s;

}

Tree \*popt(Stacktree \*s)

{

    if (isEmptyt(s))

    {

        // Underflow

        printf("Stack is Empty\n");

        return NULL;

    }

    Tree \*popped = s->array[s->top];

    s->top--;

    return popped;

}

void pusht(Stacktree \*s, Tree \*node)

{

    // Overflow (as we do in algo in class)

    if (isFullt(s))

    {

        printf("Stack is Full!\n");

        return;

    }

    s->top++;

    s->array[s->top] = node;

    return;

}

int isFullt(Stacktree \*s)

{

    return s->top == s->capacity - 1;

}

int isEmptyt(Stacktree \*s)

{

    return s->top == -1;

}

Tree \*peekt(Stacktree \*s)

{

    if (isEmptyt(s))

    {

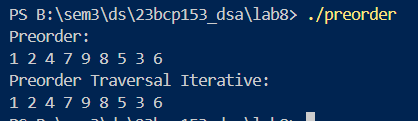
        return NULL;

    }

    return s->array[s->top];

}

### **Output:**



### **Theory (Postorder Traversal):**

**Postorder Traversal (Left, Right, Root)**: Iterative postorder traversal uses two stacks. The first stack helps to visit nodes in a modified postorder, and the second stack stores the nodes in reverse order to finally print them in correct postorder sequence.

### **Time Complexity:**

— Nodes are pushed and popped from the stack twice, but still results in linear time.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

// left right root - Postorder

typedef struct Tree {

    int info;

    struct Tree \*left;

    struct Tree \*right;

} Tree;

typedef struct Stacktree

{

    int top;

    int capacity;

    Tree \*\*array;

} Stacktree;

Tree \*makeTnode(int info);

void printPostorder(Tree \*root);

void freetree(Tree \*root);

void printPostorderiter(Tree \*root);

Stacktree \*createStackt(int n);

Tree \*popt(Stacktree \*s);

void pusht(Stacktree \*s, Tree \*node);

int isFullt(Stacktree \*s);

int isEmptyt(Stacktree \*s);

Tree \*peekt(Stacktree \*s);

int main(void)

{

    Tree \*A = makeTnode(1);

    Tree \*B = makeTnode(2);

    Tree \*C = makeTnode(3);

    Tree \*D = makeTnode(4);

    Tree \*E = makeTnode(5);

    Tree \*F = makeTnode(6);

    Tree \*G = makeTnode(7);

    Tree \*H = makeTnode(8);

    Tree \*I = makeTnode(9);

    A->left = B;

    A->right = C;

    B->left = D;

    B->right = E;

    C->left = F;

    D->left = G;

    D->right = H;

    G->left = I;

    printf("Postrder:\n");

    printPostorder(A);

    printf("\n");

    printf("Postorder Traversal Iterative:\n");

    printPostorderiter(A);

    printf("\n");

    freetree(A);

    return 0;

}

void printPostorder(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    printPostorder(root->left);

    printPostorder(root->right);

    printf("%d ", root->info);

}

// Make these below type of dynamic functions later

// Tree \*insert()

// {

// }

Tree \*makeTnode(int info)

{

    Tree \*node = (Tree \*)malloc(sizeof(Tree));

    if (node == NULL)

    {

        printf("Memory Allocation Failed!\n");

        return NULL;

    }

    node->info = info;

    node->left = NULL;

    node->right = NULL;

    return node;

}

void freetree(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    freetree(root->left);

    freetree(root->right);

    free(root);

}

void printPostorderiter(Tree \*root)

{

    if (root == NULL)

    {

        return;

    }

    Stacktree \*mainstack = createStackt(9);

    Stacktree \*outputstack = createStackt(9);

    pusht(mainstack, root);

    while (!isEmptyt(mainstack))

    {

        Tree \*current = popt(mainstack);

        pusht(outputstack, current);

        if (current->left != NULL)

        {

            pusht(mainstack, current->left);

        }

        if (current->right != NULL)

        {

            pusht(mainstack, current->right);

        }

    }

    while (!isEmptyt(outputstack))

    {

        Tree \*current = popt(outputstack);

        printf("%d ", current->info);

    }

    free(mainstack->array);

    free(mainstack);

    free(outputstack->array);

    free(outputstack);

}

Stacktree \*createStackt(int n)

{

    Stacktree \*s = (Stacktree \*)malloc(sizeof(Stacktree));

    s->capacity = n;

    s->top = -1;

    s->array = (Tree \*\*)malloc(s->capacity \* sizeof(Tree \*));

    return s;

}

Tree \*popt(Stacktree \*s)

{

    if (isEmptyt(s))

    {

        // Underflow

        printf("Stack is Empty\n");

        return NULL;

    }

    Tree \*popped = s->array[s->top];

    s->top--;

    return popped;

}

void pusht(Stacktree \*s, Tree \*node)

{

    // Overflow (as we do in algo in class)

    if (isFullt(s))

    {

        printf("Stack is Full!\n");

        return;

    }

    s->top++;

    s->array[s->top] = node;

    return;

}

int isFullt(Stacktree \*s)

{

    return s->top == s->capacity - 1;

}

int isEmptyt(Stacktree \*s)

{

    return s->top == -1;

}

Tree \*peekt(Stacktree \*s)

{

    if (isEmptyt(s))

    {

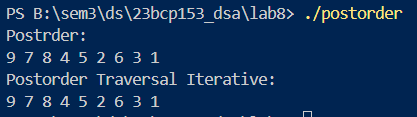
        return NULL;

    }

    return s->array[s->top];

}

### **Output:**



## **EXPERIMENT 9**

### **Aim:**

1. For a given graph , study and implement the Breadth First Search (or traversal) i.e., BFS. Also, perform complexity analysis of this algorithm in-terms of time and space.
2. For a given graph 𝐺=(𝑉,𝐸), study and implement the Depth First Search (or traversal) i.e., DFS. Also, perform complexity analysis of this algorithm in-terms of time and space.

### **Theory (Breadth First Search [BFS]):**

BFS explores a graph layer by layer, starting from a source node, and visiting all neighboring nodes before moving on to nodes at the next distance level. It uses a **queue** data structure to keep track of nodes to visit next. BFS marks each node as visited when enqueued to prevent re-processing. BFS is typically used to find the shortest path in an unweighted graph.

1. Enqueue the starting node and mark it as visited.
2. While the queue is not empty:
   1. Dequeue a node, process it (e.g., print or store it).
   2. Enqueue all unvisited adjacent nodes of the dequeued node and mark them as visited.
3. Repeat until the queue is empty, ensuring all nodes at each level are visited before advancing.

### **Time Complexity:**

1. **Adjacency List**: , where is the number of vertices and is the number of edges. Each node and each edge is processed once.
2. **Adjacency Matrix**: , as we check all pairs of nodes for edges.

### **Space Complexity:**

1. **Adjacency List**: , storing nodes and edges in the list.
2. **Queue**: , as the queue holds all nodes in the worst case.

### **Program:**

// status array

// status

// 1- ready

// 2- waiting

// 3 - processed

#include <stdio.h>

#include <stdbool.h>

#define MAX 10

int queue[MAX];

int front = -1;

int rear = -1;

// int adj[MAX][MAX];

// int status[MAX];

int status[9];

bool isEmpty();

bool isFull();

void enqueue(int ele);

int dequeue();

void printQueue();

void bfs(int start, int n, int adj[9][9]);

int main(void)

{

    int adj[9][9] = {

        {0,1,1,1,0,0,0,0,0},

        {1,0,1,0,1,0,0,0,0},

        {1,1,0,1,1,1,1,0,0},

        {1,0,1,0,0,0,1,0,0},

        {0,1,1,0,0,0,0,0,1},

        {0,0,1,0,0,0,1,1,1},

        {0,0,1,1,0,1,0,1,0},

        {0,0,0,0,0,1,1,0,1},

        {0,0,0,0,1,1,0,1,0}

    };

    for (int i = 0; i < 9; i++)

    {

        status[i] = 1;

    }

    bfs(0, 9, adj);

    return 0;

}

void bfs(int start, int n, int adj[9][9])

{

    printf("BFS Traversal: ");

    enqueue(start);

    status[start] = 2;

    while (!isEmpty())

    {

        int v = dequeue();

        printf("%d ", v);

        status[v] = 3;

        for (int i = 0; i < n; i++)

        {

            if (adj[v][i] == 1 && status[i] == 1)

            {

                enqueue(i);

                status[i] = 2;

            }

        }

    }

    printf("\n");

}

bool isEmpty()

{

    return (front == -1);

}

bool isFull()

{

    return ((rear + 1) % MAX == front);

}

void enqueue(int ele)

{

    if (isFull())

    {

        printf("Queue is FULL!\n");

        return;

    }

    if (isEmpty())

    {

        front = rear = 0;

    }

    else

    {

        rear = (rear + 1) % MAX;

    }

    queue[rear] = ele;

    // printf("Inserted: %d\n", ele);

}

int dequeue()

{

    if (isEmpty())

    {

        printf("Queue is EMPTY!\n");

        return -1;

    }

    int dequeued = queue[front];

    // printf("Deleted: %d\n", queue[front]);

    if (front == rear)

    {

        front = rear = -1;

    }

    else

    {

        front = (front + 1) % MAX;

    }

    return dequeued;

}

void printQueue()

{

    if (isEmpty())

    {

        printf("Queue is EMPTY!\n");

        return;

    }

    printf("Queue: ");

    int i = front;

    while (i != rear)

    {

        printf("%d ", queue[i]);

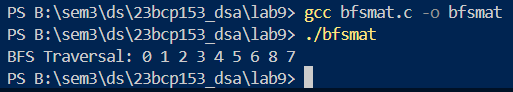
        i = (i + 1) % MAX;

    }

    printf("%d\n", queue[rear]);

}

### **Output:**



### **Theory (Depth First Search [DFS]):**

DFS explores as far as possible down one branch before backtracking, utilizing a **stack** (either explicitly or through recursive function calls). Starting from a source node, DFS goes to an unvisited adjacent node, then continues exploring deeper until reaching a dead end. After backtracking, DFS continues from the next unvisited node of the previously visited nodes. DFS is helpful in applications like detecting cycles, topological sorting, and finding connected components.

1. Push the starting node onto the stack and mark it as visited.
2. While the stack is not empty:
   1. Pop the top node, process it.
   2. Push all unvisited adjacent nodes of the popped node onto the stack and mark them as visited.
3. Repeat until the stack is empty, allowing deep exploration before moving to other branches.

### **Time Complexity:**

1. **Adjacency List**: , as we process each vertex and each edge once.
2. **Adjacency Matrix**: , checking all pairs for connectivity.

### **Space Complexity:**

1. **Adjacency List**: , to store vertices and edges.
2. **Stack**: for the stack space, as each vertex is pushed onto the stack at most once.

### **Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// stack functions are included in stackarrlib.c/.h

#define MAX 10

typedef struct Stack

{

    int top;

    int capacity;

    int \*array;

} Stack;

void dfs(int start, int n, int adj[9][9]);

// int adj[MAX][MAX];

// int status[MAX];

int status[9];

Stack \*createStack(int n);

char pop(Stack \*s);

void push(Stack \*s, int value);

int isFull(Stack \*s);

int isEmpty(Stack \*s);

int peek(Stack \*s);

int main(void)

{

    int adj[9][9] = {

        {0,1,1,1,0,0,0,0,0},

        {1,0,1,0,1,0,0,0,0},

        {1,1,0,1,1,1,1,0,0},

        {1,0,1,0,0,0,1,0,0},

        {0,1,1,0,0,0,0,0,1},

        {0,0,1,0,0,0,1,1,1},

        {0,0,1,1,0,1,0,1,0},

        {0,0,0,0,0,1,1,0,1},

        {0,0,0,0,1,1,0,1,0}

    };

    for (int i = 0; i < 9; i++)

    {

        status[i] = 1;

    }

    dfs(0, 9, adj);

    return 0;

}

void dfs(int start, int n, int adj[9][9])

{

    Stack \*mystack = createStack(9);

    printf("DFS Traversal: ");

    push(mystack, start);

    status[start] = 2;

    while(!isEmpty(mystack))

    {

        int v = pop(mystack);

        printf("%d ", v);

        status[v] = 3;

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

        {

            if (adj[v][i] == 1 && status[i] == 1)

            {

                push(mystack, i);

                status[i] = 2;

            }

        }

    }

    printf("\n");

}

Stack \*createStack(int n)

{

    Stack \*s = (Stack \*)malloc(sizeof(Stack));

    s->capacity = n;

    s->top = -1;

    s->array = (int \*)malloc(s->capacity \* sizeof(int));

    return s;

}

char pop(Stack \*s)

{

    if (isEmpty(s))

    {

        // Underflow

        printf("Stack is Empty\n");

        return -1;

    }

    char popped = s->array[s->top];

    s->top--;

    return popped;

}

void push(Stack \*s, int value)

{

    // Overflow (as we do in algo in class)

    if (isFull(s))

    {

        printf("Stack is Full!\n");

        return;

    }

    s->top++;

    s->array[s->top] = value;

    return;

}

int isFull(Stack \*s)

{

    return s->top == s->capacity - 1;

}

int isEmpty(Stack \*s)

{

    return s->top == -1;

}

int peek(Stack \*s)

{

    if (isEmpty(s))

    {

        return -1;

    }

    return s->array[s->top];

}

### **Output:**

