



ਸੰਤ ਲੌਂਗੋਵਾਲ ਇੰਜੀਨੀਅਰਿੰਗ ਐਂਡ ਟੈਕਨਾਲੋਜੀ
संत लौंगोवाल अभियांत्रिकी एवं प्रौद्योगिकी संस्थान
Sant Longowal Institute of Engineering and Technology
(Deemed-to-be-University, under Ministry of Education, Govt. of India)

Data Structure Lab Manual

Subject Code: CS215

Class: ICD III Semester



DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that this manual is a bonafide record of practical work in the **Data Structure's Lab** in **3rd Semester of II Year ICD (CSE) program** during the academic year **2024-24**. This book was prepared by **Dr. Amar Nath (Assistant Professor)**, Department of Computer Science and Engineering.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

Syllabus

Title of the course : **Data Structures Lab**

Subject Code : **CS-215**

Course Outcomes: At the end of the course, students will be able to

CO1	Develop ADT for stack, queue and linked list applications.
CO2	Implement different tree algorithms.
CO3	Implement and analyze different search and sorting algorithms.

CO/PO Mapping : (Strong(S)/Medium(M)/Weak(W) indicates strength of correlation)										
COs	Programme Outcomes (POs)									
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10
CO1		S	W	S						S
CO2		S	M	S						S
CO3		S	W	S						S

LIST OF PRACTICALS

1. WAP to generate Fibonacci Series using recursion.
2. Write a function that interchanges the first element with last element, second element with second last element and so on.
3. WAP to multiply two Matrices.
4. Write a Function that removes all duplicate elements from an Array.
5. WAP that insert an element in beginning of Linear Link List.
6. WAP that delete an element from the beginning of the Linear Link List.
7. WAP that delete an element from the end of the Linear Link List.
8. WAP that delete an element after a given element of the given Linear Link List.
9. WAP that reverse the element of the Linear Link List.
10. WAP that concatenate two Linear Linked List.
11. WAP to remove the Top element of Stack.
12. WAP to insert (or push) an element at the Top of Stack.
13. WAP to insert an element at the end of queue.
14. WAP to remove the first element of the queue.
15. WAP to illustrate the implementation of Binary Search Tree.
16. WAP to sort an array of integer in Ascending Order using Bubble Sort.
17. WAP to sort an array of integer in Ascending Order using Insertion Sort.
18. WAP to sort an array of integer in Ascending Order using Quick Sort.
19. WAP to search an element using Linear Search Method.
20. WAP to search an element using Binary Search Method.

INDEX

S.No	Content	Page No:
1.	Write a C program that generates the Fibonacci series using recursion	9-14
2.	Write a function that interchanges the first element with the last element, the second element with the second last element, and so on	15-17
3.	WAP to multiply two Matrices	17-20
4.	Write a Function that removes all duplicate elements from an Array.	20-22
5.	WAP that inserts an element at the beginning of the Linear Link List.	22-24
6.	WAP that deletes an element from the beginning of the Linear Link List.	24-26
7.	WAP that deletes an element from the end of the Linear Link List.	26-29
8.	WAP that deletes an element after a given element of the given Linear Link List.	30-32
9.	WAP that reverses the element of the Linear Link List.	32-35
10.	WAP that concatenates two Linear Linked lists.	35-38
11.	WAP to remove the Top element of the Stack.	38-41
12.	WAP to insert (or push) an element at the Top of the Stack.	41-43
13.	WAP to insert an element at the end of the queue.	43-45
14.	WAP to remove the first element of the queue.	48-51
15.	WAP to illustrate the implementation of Binary Search Tree.	51-54
16.	WAP to sort an array of integers in ascending order using Bubble Sort.	54-57
17.	WAP to sort an array of integers in ascending order using Insertion Sort.	57-58
18.	18. WAP to sort an array of integer in Ascending Order using Quick Sort.	59-61
19.	WAP to search an element using the Linear Search Method	61-63
20.	WAP to search an element using the Binary Search Method	63-65

PREFACE

This "**Data Structures**" lab manual is designed to teach the fundamentals of data structure design and analysis, emphasizing implementation in the C programming language. Readers of this manual are expected to be familiar with the syntax of C and similar procedural languages. Data structure concepts are increasingly critical to the IT industry, particularly for software development at the system level.

This practical manual has been carefully prepared to enhance the development of procedural programming skills. It includes a variety of exercises and their solutions so that students can understand them quickly and easily. This manual will prove valuable to Computer Science & Engineering students in grasping the applied aspects of data structures. There is always room for improvement, and we welcome suggestions from readers and users for future editions.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

ACKNOWLEDGEMENT

It was a wonderful experience working on the “Data Structures Lab” manual. First, I would like to express my sincere gratitude to **Prof. Birmohan**, Head of the Department of Computer Science and Engineering, for his continuous support and technical guidance in preparing this document. I am deeply indebted and would like to acknowledge the invaluable support and patronage of **Prof. Mani Kant Paswan**, Director of the institute, for providing me with this excellent opportunity and his constant encouragement throughout the process. Finally, I extend my heartfelt thanks to the entire faculty of the CSE Department, whose inspiration and assistance helped me achieve this goal.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

General Instructions

1. **Punctuality:** Students must arrive on time for the Data Structures lab. Latecomers will not be allowed to participate in the lab session.
2. **Attendance:** Experiments missed due to tardiness will be avoided. Students are expected to be on time.
3. **Preparation:** Students should prepare at home for the sessions' scheduled experiments.
4. **ID Cards:** Displaying an identity card is mandatory for entry into the lab.
5. **Mobile Phones:** Students need help to bring mobile phones into the lab.
6. **Responsibility for Equipment:** Any damage or loss of equipment, such as keyboards or mice, during the lab session will be the student's responsibility. A penalty or fine will be imposed if necessary.
7. **Lab Records:** Students must update their lab observation books and records after each session. Before leaving, they must get their lab observation book signed by the faculty member.
8. **Submission of Lab Records:** Lab records must be submitted to the faculty in the staffroom during the next lab session for correction and return.
9. **Movement in the Lab:** Students should remain at their assigned stations and avoid moving around during the lab session.
10. **Emergencies:** In an emergency, students must obtain written permission from the faculty member in charge.
11. **Disciplinary Action:** Faculty members can suspend students from the lab session for disciplinary reasons.
12. **Original Work:** Students should not copy outputs from others. They must write their own results.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

Vision & Mission and Programme Educational Objectives

Vision

- To achieve technical & research excellence in Computer Science and Engineering with industrial & social perspectives.

Mission

- To provide an environment for imparting high-quality technical education, skill development, research, and development.
- To disseminate sound knowledge of recent Computer Technologies by organizing seminars/workshops/short-term courses.
- To develop interaction and collaboration with the industry.
- To facilitate Hands-on training to the students to promote Self-Employment

Program Educational Objectives (PEOs) – ICD

1. To provide students with the basic knowledge of Computer fundamentals, Hardware, Operating Systems, Internet and Networking, Databases, and Computer Programming.
2. Enhance students' critical thinking skills with the help of practical training and in-house training to develop their analytical, problem-solving, and decision-making skills.
3. To provide students with a deep insight into various cutting-edge technologies & tools, creating diverse career opportunities.
4. To provide a solid academic, technical, and intellectual background to enable them to pursue degree courses.

Experiment 1. Write a C program that generates the Fibonacci series using recursion

```
#include <stdio.h>

// Function to calculate Fibonacci number using recursion
int fibonacci(int n) {
    if (n == 0)
        return 0; // Base case 1: Fibonacci(0) = 0
    else if (n == 1)
        return 1; // Base case 2: Fibonacci(1) = 1
    else
        return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}

int main() {
    int n, i;

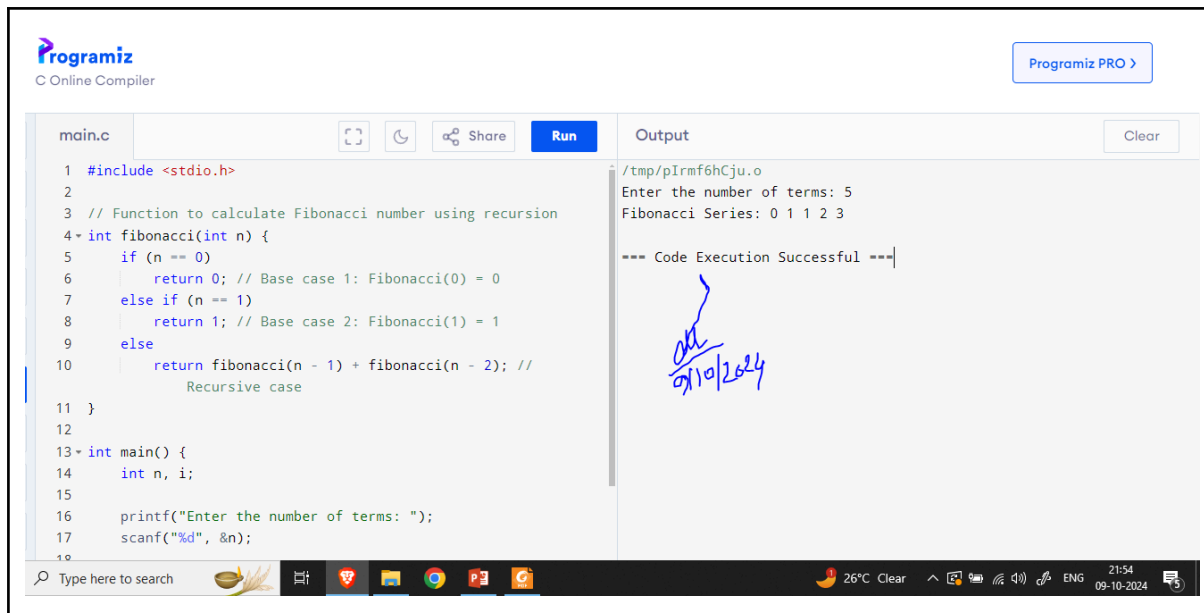
    printf("Enter the number of terms: ");
    scanf("%d", &n);

    printf("Fibonacci Series: ");
    for (i = 0; i < n; i++) {
        printf("%d ", fibonacci(i)); // Call the recursive function
    }

    return 0;
}
```

Explanation:

- The **Fibonacci** function calculates the Fibonacci number of a given **n** using recursion.
- The base cases are when **n == 0** (returns 0) and **n == 1** (returns 1).
- The function recursively calls itself with the previous two terms (**n-1**) and (**n-2**) to get the next Fibonacci number.
- The number of terms **n** is taken from the user in the **main** function, and the Fibonacci series is printed using a loop.



```

main.c
1 #include <stdio.h>
2
3 // Function to calculate Fibonacci number using recursion
4 int fibonacci(int n) {
5     if (n == 0)
6         return 0; // Base case 1: Fibonacci(0) = 0
7     else if (n == 1)
8         return 1; // Base case 2: Fibonacci(1) = 1
9     else
10        return fibonacci(n - 1) + fibonacci(n - 2); //
            Recursive case
11 }
12
13 int main() {
14     int n, i;
15
16     printf("Enter the number of terms: ");
17     scanf("%d", &n);
18 }

```

Output

```

/tmp/pIrmf6hCju.o
Enter the number of terms: 5
Fibonacci Series: 0 1 1 2 3

--- Code Execution Successful ---

```

26°C Clear 21:54 09-10-2024

Viva-questions

1. What is recursion in C? How does it work?
 - a. Follow-up: Can you explain the difference between recursion and iteration?
2. What is the base case in a recursive function? Why is it important in recursion?
 - a. Follow-up: What will happen if a base case is not provided in a recursive function?
3. Can you explain the time complexity of the Fibonacci program using recursion?
 - a. Follow-up: How can the time complexity be improved?
4. What is the difference between the iterative and recursive approach for generating a Fibonacci series?
5. How is the stack used during the execution of recursive function calls?
 - a. Follow-up: What is a stack overflow error, and how could it occur in this program?
6. What is the role of the function call stack in recursion?
 - a. Follow-up: How does the function call stack behave when the **Fibonacci** function is called recursively?
7. Can you explain the flow of execution when the Fibonacci series is generated using recursion?
 - a. Follow-up: What are the values of **Fibonacci (3)** and **Fibonacci (4)**?
8. How would you modify the program to handle large Fibonacci numbers without causing stack overflow or excessive function calls?
9. What are the advantages and disadvantages of using recursion over loops to generate a Fibonacci series?
10. What is memoization, and how can it be applied to optimize the recursive Fibonacci program?

Answers

What is recursion in C? How does it work?

Recursion in C is a technique where a function calls itself to solve a problem. It works by breaking down a complex problem into smaller instances of the same problem, solving these smaller instances, and then combining the results. The function keeps calling itself until it reaches a *base case*, which terminates the recursion. Without a base case, the recursion would continue infinitely.

2. Can you explain the difference between recursion and iteration?

- **Recursion:** A function repeatedly calls itself with simpler inputs, solving the problem by reducing it step by step until reaching a base case. Recursion requires more memory due to the function call stack and may be less efficient for some problems.
 - **Iteration:** Uses loops (e.g., `for`, `while`) to execute a code block repeatedly. It's generally more efficient than recursion because it doesn't require extra memory for function calls and does not involve stack overhead.
-

3. What is the base case in a recursive function? Why is it important in recursion?

The *base case* is the condition that stops the recursion. It's crucial because the function would keep calling itself indefinitely without it, leading to a stack overflow. In the Fibonacci example, the base cases are `Fibonacci (0)`, returning 0, and `Fibonacci (1)` returning 1.

4. What will happen if a base case is not provided in a recursive function?

Without a base case, the function will continue calling itself indefinitely, creating more and more frames in the function call stack. This eventually leads to a *stack overflow* error as the program exceeds the stack memory limit.

5. Can you explain the time complexity of the Fibonacci program using recursion?

The time complexity of the recursive Fibonacci program is $O(2^n)$. This is because, for each call of `Fibonacci (n)`, the function makes two additional recursive calls: `Fibonacci (n-1)` and `Fibonacci (n-2)`. As a result, the function performs redundant calculations, leading to exponential growth in the number of calls.

6. How can the time complexity be improved?

Time complexity can be improved using *memoization*, a technique that stores previously computed values and reuses them when needed. This reduces the number of redundant calculations, improving the time complexity to $O(n)$. Alternatively, an iterative approach can also reduce the time complexity to $O(n)$.

7. What is the difference between the iterative and recursive approach for generating a Fibonacci series?

- Recursive approach: The function calls itself repeatedly, with a high time complexity of $O(2^n)$ and additional overhead from function calls.
 - Iterative approach: Uses loops to calculate the Fibonacci sequence straightforwardly. It has a linear time complexity of $O(n)$ and is generally more efficient in terms of both time and space.
-

8. How is the stack used to execute recursive function calls?

In recursion, a new stack frame is created each time *a function* is called that stores the function's local variables, parameters, and return address. The stack grows with each recursive call and shrinks as the function returns. When the base case is reached, the function starts unwinding, returning results to the previous frames in the stack.

9. What is a stack overflow error, and how could it occur in this program?

A *stack overflow* error occurs when the call stack exceeds its memory limit due to too many function calls, which may happen if the recursion depth becomes too large. In the Fibonacci program, if n is very large, the recursive function calls will build up too many stack frames, leading to this error.

10. What is the role of the function call stack in recursion?

The function call stack maintains information about active function calls, storing their local variables, parameters, and return addresses. In recursion, each function call adds a new frame to the stack, and when a function returns, its frame is popped off the stack.

11. How does the function call stack behave when the Fibonacci function is called recursively?

When `Fibonacci (n)` is called, the function pushes a frame onto the stack for each recursive

call. As the function calls `Fibonacci (n-1)` and `Fibonacci (n-2)`, more frames are added to the stack. Once the base case is reached, the results are passed back through the stack, and each frame is popped off as the recursion unwinds.

12. Can you explain the flow of execution when the Fibonacci series is generated using recursion?

When `fibonacci(n)` is called:

1. It checks the base cases (`n == 0` or `n == 1`).
 2. If the base case is not reached, the function calls itself twice for `Fibonacci (n-1)` and `Fibonacci (n-2)`.
 3. This process continues recursively, building up a call stack until the base case is reached.
 4. Once the base case is reached, the results are returned back up the call stack.
 5. The intermediate Fibonacci numbers are combined to get the final result.
-

13. What are the values of `fibonacci(3)` and `fibonacci(4)`?

- `Fibonacci(3) = 2`
 - `Fibonacci(4) = 3`
-

14. How would you modify the program to handle large Fibonacci numbers without causing stack overflow or excessive function calls?

To handle large Fibonacci numbers:

- Use *memoization* to store previously calculated Fibonacci numbers, avoiding redundant recursive calls.
 - Switch to an *iterative approach* to avoid the recursive call stack altogether.
 - Alternatively, use *dynamic programming* or *tail recursion* if supported, to optimize the space and avoid stack overflow.
-

15. What are the advantages and disadvantages of using recursion over loops for generating a Fibonacci series?

- Advantages of recursion:
 - Easier to write and understand certain problems.
 - More elegant and concise for problems that naturally fit a recursive solution (e.g., tree traversal).
- Disadvantages of recursion:
 - This can lead to stack overflow for large inputs.

- Higher time and space complexity due to multiple function calls and call stack usage.
 - Advantages of iteration:
 - More efficient in terms of time and space complexity.
 - It avoids the overhead of multiple function calls and uses constant space.
 - Disadvantages of iteration:
 - It may be harder to conceptualize some problems compared to recursion.
-

16. What is memoization, and how can it be applied to optimize the recursive Fibonacci program?

Memoization is a technique used to store the results of expensive function calls and reuse them when the same inputs occur again, thus avoiding redundant calculations. In the Fibonacci program, memoization can be applied by storing the results of **Fibonacci (n)** in an array or hash table. Before making a recursive call, the function checks if the **Fibonacci (n)** result is already stored. If it is, the function returns the stored result, avoiding redundant calculations. This reduces the time complexity from $O(2^n)$ to $O(n)$.

Experiment 2

Write a function that interchanges the first element with the last element, the second element with the second last element, and so on.

```
#include <stdio.h>

void swapElements(int arr[], int size) {
    int temp;
    // Loop to swap elements
    for (int i = 0; i < size / 2; i++) {
        // Swap arr[i] with arr[size - i - 1]
        temp = arr[i];
        arr[i] = arr[size - i - 1];
        arr[size - i - 1] = temp;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    // Call the function to swap elements
    swapElements(arr, size);

    printf("\nArray after swapping: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

Original array: 1 2 3 4 5 6
Array after swapping: 6 5 4 3 2 1

```
main.c [ ] [ ] [ ] Share Run Output Clear
16 int size = sizeof(arr) / sizeof(arr[0]);
17
18 printf("Original array: ");
19 for (int i = 0; i < size; i++) {
20     printf("%d ", arr[i]);
21 }
22
23 // Call the function to swap elements
24 swapElements(arr, size);
25
26 printf("\nArray after swapping: ");
27 for (int i = 0; i < size; i++) {
28     printf("%d ", arr[i]);
29 }
30
31 return 0;
32 }
33

/tmp/SnNAj2sWhV.o
Original array: 1 2 3 4 5 6
Array after swapping: 6 5 4 3 2 1

=== Code Execution Successful ===
```

Viva questions

1. How does the function **swapElements** work to interchange the elements of the array?

- *Expected Answer:* The function loops through the first half of the array and swaps the elements at the beginning with those at the end using a temporary variable. It continues this process until it has reached the middle of the array.

2. Why only loop until **size / 2** in the function?

- *Expected Answer:* We only need to loop until the middle of the array because after swapping the first half with the second half, the entire array will be reversed. Swapping beyond the middle would undo the changes.

3. What is the role of the **temp** variable in this program?

- *Expected Answer:* The **temp** variable temporarily holds the value of one of the elements during the swap. Without it, the value of one element would be overwritten before the swap is complete.

4. What would happen if we do not include the statement **arr[size - i - 1] = temp** in the function?

- *Expected Answer:* Without this statement, the second element in the swap would

not get its correct value. The array would lose data, and the swap wouldn't be completed properly.

5. How would the program behave if the array size is odd, for example, `arr[] = {1, 2, 3, 4, 5}`?

- *Expected Answer:* If the array has an odd size, the middle element will remain in its place. The function only swaps elements at opposite ends and doesn't affect the middle one, so in this case, the output will be `5 4 3 2 1`, where 3 stays in the middle.

Experiment 3. WAP to multiply two Matrices.

```
#include <stdio.h>

#define MAX 10 // Define the maximum size of the matrix

void multiplyMatrices(int firstMatrix[MAX][MAX], int secondMatrix[MAX][MAX], int
result[MAX][MAX], int row1, int col1, int row2, int col2) {
    // Initializing elements of result matrix to 0
    for (int i = 0; i < row1; i++) {
        for (int j = 0; j < col2; j++) {
            result[i][j] = 0;
        }
    }

    // Multiplying firstMatrix and secondMatrix and storing in result
    for (int i = 0; i < row1; i++) {
        for (int j = 0; j < col2; j++) {
            for (int k = 0; k < col1; k++) {
                result[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}

int main() {
    int firstMatrix[MAX][MAX], secondMatrix[MAX][MAX], result[MAX][MAX];
    int row1, col1, row2, col2;

    // Taking input for the dimensions of the matrices
    printf("Enter rows and columns for the first matrix: ");
    scanf("%d %d", &row1, &col1);
```

```
printf("Enter rows and columns for the second matrix: ");
scanf("%d %d", &row2, &col2);

// Check if matrix multiplication is possible
if (col1 != row2) {
    printf("Error: Column of the first matrix must be equal to row of the second
matrix.\n");
    return 0;
}

// Taking input for the first matrix
printf("\nEnter elements of the first matrix:\n");
for (int i = 0; i < row1; i++) {
    for (int j = 0; j < col1; j++) {
        scanf("%d", &firstMatrix[i][j]);
    }
}

// Taking input for the second matrix
printf("\nEnter elements of the second matrix:\n");
for (int i = 0; i < row2; i++) {
    for (int j = 0; j < col2; j++) {
        scanf("%d", &secondMatrix[i][j]);
    }
}

// Function to multiply the matrices
multiplyMatrices(firstMatrix, secondMatrix, result, row1, col1, row2, col2);

// Display the result of the multiplication
printf("\nResultant matrix:\n");
for (int i = 0; i < row1; i++) {
    for (int j = 0; j < col2; j++) {
        printf("%d ", result[i][j]);
    }
    printf("\n");
}

return 0;
}
```

Input

Enter rows and columns for the first matrix: 2 3

Enter rows and columns for the second matrix: 3 2

Enter elements of the first matrix:

1 2 3

4 5 6

Enter elements of the second matrix:

7 8

9 10

11 12

Output

Resultant matrix:

58 64

139 154

Viva questions**1. What are the conditions for two matrices to be multiplied?**

- **Expected Answer:** To multiply two matrices, the number of columns in the first matrix must equal the number of rows in the second matrix. If the first matrix is of size $m \times n$ and the second matrix is of size $n \times p$, the resultant matrix will be of size $m \times p$.
-

2. How is the resultant matrix initialized before performing multiplication?

- **Expected Answer:** The resultant matrix is initialized to zero before performing multiplication. This is done using a nested loop that sets each element of the result matrix to zero, ensuring that any previous values do not interfere with the final result.
-

3. Explain the triple nested loop used in the multiplication function.

- **Expected Answer:** The outer two loops iterate over the rows of the first matrix and the columns of the second matrix, respectively. The innermost loop iterates over the columns of the first matrix (or rows of the second matrix) to calculate the dot

product of the corresponding row from the first matrix and the column from the second matrix. The products are summed up and stored in the corresponding position in the resultant matrix.

4. What would happen if you multiply two matrices with incompatible dimensions?

- **Expected Answer:** If you try to multiply two matrices with incompatible dimensions (where the number of columns in the first matrix does not equal the number of rows in the second matrix), the program will output an error message and terminate without performing any multiplication.
-

5. How can the program be modified to handle larger matrices or dynamic memory allocation?

- **Expected Answer:** The program can be modified to handle larger matrices using dynamic memory allocation with `malloc` to allocate memory for matrices at runtime based on user input. This allows for greater flexibility in terms of matrix sizes. Additionally, we would need to free the allocated memory at the end of the program to prevent memory leaks.

Experiments 4. Write a Function that removes all duplicate elements from an Array.

```
#include <stdio.h>

void removeDuplicates(int arr[], int n) {
    int temp[n]; // Temporary array to store unique elements
    int j = 0;   // Variable to store the index of unique elements

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

        // Check if the element is already in the temp array
        for (int k = 0; k < j; k++) {
            if (arr[i] == temp[k]) {
                isDuplicate = 1;
                break;
            }
        }

        // If not a duplicate, add it to the temp array
        if (!isDuplicate) {
```

```
        temp[j] = arr[i];
        j++;
    }
}

// Copy the unique elements back to the original array
for (int i = 0; i < j; i++) {
    arr[i] = temp[i];
}

// Print the array with duplicates removed
printf("Array after removing duplicates: ");
for (int i = 0; i < j; i++) {
    printf("%d ", arr[i]);
}
printf("\n");
}

int main() {
    int arr[] = {1, 2, 2, 3, 4, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    removeDuplicates(arr, n);

    return 0;
}
```

Output.**Input array int arr[] = {1, 2, 2, 3, 4, 4, 5};****Array after removing duplicates: 1 2 3 4 5****Viva Questions:**

1. What is the purpose of the temporary array in this function?
 - The temporary array is used to store only the unique elements from the original array.
2. Can you explain how the algorithm checks for duplicate elements?
 - For each element in the array, the algorithm compares it with the elements already stored in the temporary array. If it finds a match, it marks it as a duplicate and does not add it to the temporary array.
3. What is the time complexity of this algorithm?
 - The time complexity is $O(n^2)$, where n is the number of elements in the array, due to the nested loop structure used to check for duplicates.

4. Can this algorithm be optimized? If so, how?
 - Yes, by using a data structure like a hash set to store unique elements, the time complexity can be reduced to $O(n)$.
5. What happens if the original array contains all unique elements?
 - If all elements are unique, the function will simply copy all elements to the temporary array and return the original array without any changes.

Experiments 5

WAP that inserts an element at the beginning of the Linear Link List.

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the beginning
void insertAtBeginning(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // Assign data to the new node
    new_node->data = new_data;

    // Make the new node point to the current head
    new_node->next = *head_ref;

    // Move the head to point to the new node
    *head_ref = new_node;
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

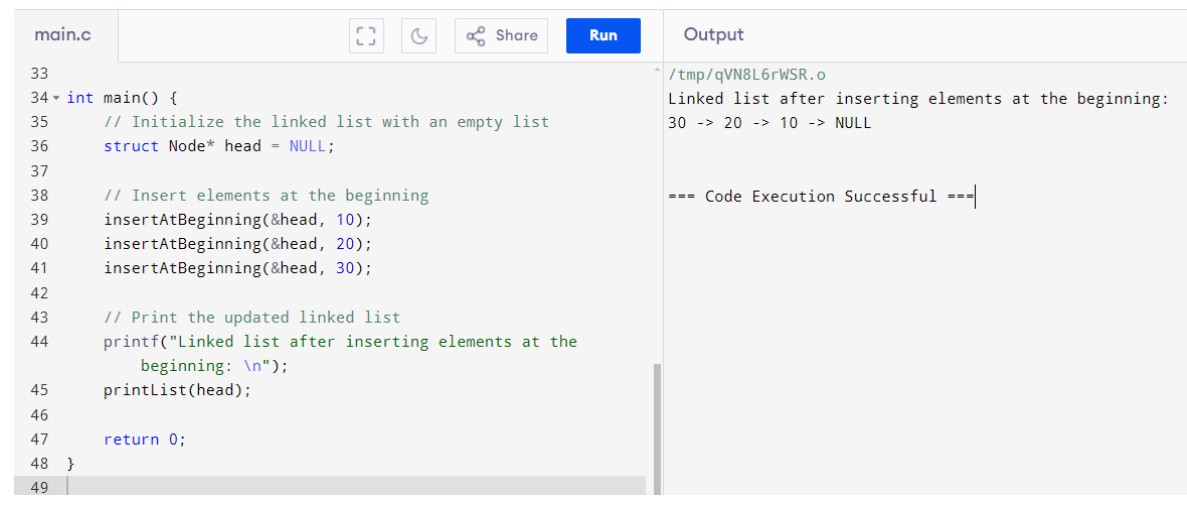
int main() {
```

```
// Initialize the linked list with an empty list
struct Node* head = NULL;

// Insert elements at the beginning
insertAtBeginning(&head, 10);
insertAtBeginning(&head, 20);
insertAtBeginning(&head, 30);

// Print the updated linked list
printf("Linked list after inserting elements at the beginning: \n");
printList(head);

return 0;
}
```



```
main.c [Run] [Share] [Refresh] [Fullscreen]
33
34+ int main() {
35 // Initialize the linked list with an empty list
36 struct Node* head = NULL;
37
38 // Insert elements at the beginning
39 insertAtBeginning(&head, 10);
40 insertAtBeginning(&head, 20);
41 insertAtBeginning(&head, 30);
42
43 // Print the updated linked list
44 printf("Linked list after inserting elements at the
beginning: \n");
45 printList(head);
46
47 return 0;
48 }
49
```

Output

```
~/tmp/qVN8L6rWSR.o
Linked list after inserting elements at the beginning:
30 -> 20 -> 10 -> NULL

=== Code Execution Successful ===
```

Explanation:

1. **Node Structure:** The program defines a structure **Node** with two fields: **data** to store the element and **next** to point to the next node in the list.
2. **insertAtBeginning Function:**
 - This function creates a new node, assigns it the value (**new_data**), and sets its **next** pointer to the current head of the list.
 - It then updates the head to point to this new node, effectively inserting the element at the beginning.
3. **printList Function:** This function traverses the linked list, printing the **data** of each node.
4. **Main Function:** Elements **30**, **20**, and **10** are inserted at the beginning, and the list is printed.

Viva questions.**1. What is the purpose of the `insertAtBeginning` function?**

- **Expected Answer:** The `insertAtBeginning` function inserts a new node at the beginning of the linked list by creating a new node, assigning data to it, and then making it point to the current head of the list. Finally, it updates the head to point to this new node.
-

2. How does the `malloc` function work in the context of linked lists?

- **Expected Answer:** The `malloc` function dynamically allocates memory for a new node in the linked list. It returns a pointer to the allocated memory, which is then assigned to the new node. This is necessary because we must allocate memory for each node at runtime.
-

3. What would happen if you don't update the head pointer after inserting a new node?

- **Expected Answer:** If the head pointer is not updated to point to the new node, the linked list will lose its reference to the newly inserted node, and the list will remain unchanged. The new node would not become the first element in the list.
-

4. Can you explain the time complexity of inserting an element at the beginning of a linked list?

- **Expected Answer:** The time complexity of inserting an element at the beginning of a linked list is $O(1)$ because we only need to create a new node and adjust a few pointers, regardless of the size of the list.
-

5. What is the difference between inserting at the beginning and inserting at the end of a linked list in terms of time complexity?

- **Expected Answer:** Inserting at the beginning of a linked list has a time complexity of $O(1)$ because it only requires adjusting the head pointer. In contrast, inserting at the end of a singly linked list has a time complexity of $O(n)$, where N is the number of nodes in the list because we need to traverse the list to find the last node before inserting the new node.

Experiments 6**WAP that deletes an element from the beginning of the Linear Link List.**

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to delete a node from the beginning of the linked list
struct Node* deleteFromBeginning(struct Node* head) {
    if (head == NULL) {
        printf("List is already empty.\n");
        return NULL;
    }

    // Store the head node temporarily
    struct Node* temp = head;

    // Move head to the next node
    head = head->next;

    // Free the old head node
    free(temp);

    return head;
}

// Function to push elements into the linked list (at the beginning)
void push(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    // Insert the data
    new_node->data = new_data;

    // Make the new node point to the old head
    new_node->next = (*head_ref);

    // Move the head to the new node
    (*head_ref) = new_node;
}

// Function to print the linked list
```

```
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Adding some elements to the list
    push(&head, 10);
    push(&head, 20);
    push(&head, 30);

    printf("Initial linked list: ");
    printList(head);

    // Deleting element from the beginning
    head = deleteFromBeginning(head);

    printf("Linked list after deletion: ");
    printList(head);

    return 0;
}
```

Output:

/tmp/4QUVNUoRH1.o

Initial linked list: 30 -> 20 -> 10 -> NULL

Linked list after deletion: 20 -> 10 -> NULL

Viva questions

1. What is a linked list, and how does it differ from an array?
 - a. *Follow-up:* Can you explain how memory allocation works in a linked list compared to an array?
2. Explain how the `deleteFromBeginning` function works. What happens if the linked list is empty?
 - a. *Follow-up:* Why is it important to free the memory of the deleted node?
3. Why do we need to update the head pointer after deleting the first node?
 - a. *Follow-up:* What would happen if we forgot to update the head pointer?
4. What is the time complexity of deleting a node from the beginning of a singly linked list?

- a. *Follow-up*: How would this differ for deletion from the end of the list?
- 5. What is the role of the **struct** in the linked list implementation, and why do we use dynamic memory allocation (**malloc**) to create nodes?
 - a. *Follow-up*: What are the potential risks of not using **free()** correctly after deleting nodes?

Experiments 7

WAP that deletes an element from the end of the Linear Link List.

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to delete the last node from the linked list
struct Node* deleteFromEnd(struct Node* head) {
    // If the list is empty
    if (head == NULL) {
        printf("List is already empty.\n");
        return NULL;
    }

    // If there is only one node in the list
    if (head->next == NULL) {
        free(head);
        return NULL;
    }

    // Traverse to the second last node
    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }

    // Free the last node and update the second last node's next pointer to NULL
    free(temp->next);
    temp->next = NULL;

    return head;
}
```

```
// Function to push elements into the linked list (at the beginning)
void push(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    // Insert the data
    new_node->data = new_data;

    // Make the new node point to the old head
    new_node->next = (*head_ref);

    // Move the head to the new node
    (*head_ref) = new_node;
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Adding some elements to the list
    push(&head, 10);
    push(&head, 20);
    push(&head, 30);

    printf("Initial linked list: ");
    printList(head);

    // Deleting the last element
    head = deleteFromEnd(head);

    printf("Linked list after deletion from end: ");
    printList(head);

    return 0;
}
```

Output

Initial linked list: 30 -> 20 -> 10 -> NULL

Linked list after deletion from end: 30 -> 20 -> NULL

**Explanation:**

- **deleteFromEnd**: This function deletes the last node of the linked list.
 - If the list is empty, it returns **NULL** and prints a message.
 - If the list has only one node, it deletes that node and returns **NULL** (indicating the list is now empty).
 - Otherwise, it traverses the list to find the second-to-last node, frees the last node, and sets the second-to-last node's **next** pointer to **NULL**.
- **push**: This function adds new elements to the front of the list.
- **printList**: This function prints the elements of the list.

Viva question

1. Can you explain how the **deleteFromEnd** function works in this program?
 - *Follow-up*: What is the role of the second-to-last node in the deletion process?
2. What happens if the linked list has only one node, and you try to delete the last node using the **deleteFromEnd** function?
 - *Follow-up*: How do we handle this special case in the program?
3. What is the time complexity of deleting the last node from a singly linked list, and why?
 - *Follow-up*: How would this change if we worked with a doubly linked list?
4. Why do we need to traverse to the second-to-last node when deleting the last node in a singly linked list?
 - *Follow-up*: What challenges would arise if we directly try to access the last node?
5. What is the role of dynamic memory management (**malloc** and **free**) in linked lists, and what could happen if we do not free memory correctly after deletion?
 - *Follow-up*: What is a memory leak, and how can it affect a program over time?

Experiment 8**WAP that deletes an element after a given element of the given Linear Link List.**

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to delete a node after a given element
void deleteAfter(struct Node* head, int key) {
    struct Node* temp = head;

    // Traverse the list to find the node with the given key
    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }

    // If the key is not present or the key is the last node, nothing to delete
    if (temp == NULL || temp->next == NULL) {
        printf("Element not found or no node exists after the given element.\n");
        return;
    }

    // Store the node to be deleted
    struct Node* nodeToDelete = temp->next;

    // Adjust the next pointer to skip the node to be deleted
    temp->next = nodeToDelete->next;

    // Free the memory of the node to be deleted
    free(nodeToDelete);
}

// Function to push elements into the linked list (at the beginning)
void push(struct Node** head_ref, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    // Insert the data
    new_node->data = new_data;

    // Make the new node point to the old head
    new_node->next = (*head_ref);
}
```

```
// Move the head to the new node
(*head_ref) = new_node;
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Adding some elements to the list
    push(&head, 50);
    push(&head, 40);
    push(&head, 30);
    push(&head, 20);
    push(&head, 10);

    printf("Initial linked list: ");
    printList(head);

    // Deleting the element after 30
    deleteAfter(head, 30);

    printf("Linked list after deletion after element 30: ");
    printList(head);

    return 0;
}
```

Output:**Initial linked list: 10 -> 20 -> 30 -> 40 -> 50 -> NULL****Linked list after deletion after element 30: 10 -> 20 -> 30 -> 50 -> NULL****Explanation:**

- **deleteAfter**: This function deletes the node after the node with the specified key (**key**). It checks for the existence of the key and whether there's a node after it. If found, it adjusts the links to skip the node and frees its memory.
- **push**: Adds elements to the front of the linked list.
- **printList**: Prints the elements in the linked list.

Viva questions

1. How does the **deleteAfter** function work, and what are the key steps involved in deleting a node after a given element?
 - *Follow-up*: What happens if the node with the given key is not found or if it is the last node?
2. Why do we need to traverse the list to find the node with the given key before deleting the node after it?
 - *Follow-up*: How would this process differ in a doubly linked list?
3. What is the time complexity of the **deleteAfter** function, and why?
 - *Follow-up*: Would the time complexity change if the key was located at the end of the list?
4. What happens to the deleted node's memory, and why is it important to free it after deletion?
 - *Follow-up*: What issues can arise if we forget to free the memory of the deleted node?
5. What is the role of the **temp** pointer in the **deleteAfter** function, and why is it necessary to adjust the **next** pointers during the deletion?
 - *Follow-up*: What would happen if the **next** pointer of the current node was not updated correctly?

Experiment 9

WAP that reverses the element of the Linear Link List.

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to push a new node at the beginning of the linked list
```



```
void push(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Function to reverse the linked list
void reverse(struct Node** head_ref) {
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL) {
        next = current->next; // Store the next node
        current->next = prev; // Reverse the current node's pointer
        prev = current;      // Move the prev and current pointers one step forward
        current = next;
    }
    *head_ref = prev; // Update the head to point to the new first node
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

// Main function to test the reverse function
int main() {
    struct Node* head = NULL;

    // Create a linked list 1 -> 2 -> 3 -> 4 -> 5 -> NULL
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("Original linked list:\n");
    printList(head);

    reverse(&head);

    printf("Reversed linked list:\n");
    printList(head);
}
```

```
return 0;  
}
```

Output:

Original linked list:

1 -> 2 -> 3 -> 4 -> 5 -> NULL

Reversed linked list:

5 -> 4 -> 3 -> 2 -> 1 -> NULL

Explanation:

1. Node Structure: The linked list comprises nodes, and each node contains integer data and a pointer to the next node.
2. push() Function: This function adds a node to the beginning of the linked list.
3. reverse() Function: This function reverses the linked list by changing the direction of the pointers between the nodes.
4. printList() Function: This function prints the elements of the linked list.
5. Main: In the `main()` function, we create a linked list, reverse it, and then print both the original and reversed **lists**.

Viva questions

1. What is a linked list, and how does it differ from an array?
 - Expected Answer: A linked list is a data structure consisting of nodes where each node contains data and a reference (or pointer) to the next node in the sequence. Unlike arrays, linked lists allow dynamic memory allocation, making them more efficient for operations like insertions and deletions. Still, they do not provide direct access to elements by index-like arrays.
2. Explain the process of reversing a singly linked list.
 - Expected Answer: To reverse a singly linked list, we need to iterate through the list and change the direction of the pointers for each node. We maintain three pointers: `prev`, `current`, and `next`. Initially, `prev` is set to `NULL`, and we iterate through the list while making each node's next pointer point to the previous node. At the end of the iteration, `prev` becomes the new head of the reversed list.
3. What is the time complexity of reversing a linked list, and why?
 - Expected Answer: The time complexity of reversing a linked list is $O(n)$, where n is the number of nodes. This is because we have to traverse each node once to reverse the direction of the pointers.
4. Can we reverse a linked list using recursion? If yes, how?
 - Expected Answer: Yes, we can reverse a linked list using recursion. In a recursive approach, we recursively move to the last node and adjust the next pointers backward. Each node's next pointer is set to its previous node, and

the base case is when the function reaches the last node (where the current node's next is **NULL**).

5. What is the difference between a singly linked list and a doubly linked list?
- Expected Answer: In a singly linked list, each node contains data and a pointer to the next node in the list. In a doubly linked list, each node contains data, a pointer to the next node, and a pointer to the previous node. A doubly linked list allows traversal in both directions (forward and backward), while a singly linked list only allows forward traversal.

Experiments 10.

WAP that concatenates two Linear Linked lists.

```
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of a linked list
void insertNode(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

```
// Function to display the linked list
void displayList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Function to concatenate two linked lists
void concatenateLists(struct Node** head1, struct Node* head2) {
    if (*head1 == NULL) {
        *head1 = head2;
        return;
    }
    struct Node* temp = *head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head2;
}

// Main function
int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    // Insert elements in the first linked list
    insertNode(&list1, 10);
    insertNode(&list1, 20);
    insertNode(&list1, 30);

    // Insert elements in the second linked list
    insertNode(&list2, 40);
    insertNode(&list2, 50);
    insertNode(&list2, 60);

    printf("List 1: ");
    displayList(list1);

    printf("List 2: ");
    displayList(list2);

    // Concatenate the lists
    concatenateLists(&list1, list2);

    printf("Concatenated List: ");
    displayList(list1);
}
```

```
return 0;  
}
```

Output:**List 1: 10 -> 20 -> 30 -> NULL****List 2: 40 -> 50 -> 60 -> NULL****Concatenated List: 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> NULL****Viva questions**

1. What is a Linked List? How is it different from an array?
 - A linked list is a linear data structure storing elements in nodes, and each node points to the next node. In an array, memory is allocated contiguously, whereas in a linked list, memory is dynamically allocated and may not be contiguous. Linked lists offer dynamic size and easier insertion/deletion at any position than arrays.
2. What are the types of Linked Lists?
 - The three common types of linked lists are:
 - **Singly Linked List:** Each node contains a data part and a pointer to the next node.
 - **Doubly Linked List:** Each node contains a data part, a pointer to the next node, and a pointer to the previous node.
 - **Circular Linked List:** In this list, the last node points back to the first node, forming a circular structure.
3. How does the concatenation of two linked lists work?
 - To concatenate two linked lists, the pointer of the last node of the first list is updated to point to the head node of the second list. The second list is appended to the end of the first one.
4. What are the advantages and disadvantages of using a linked list over an array?
 - **Advantages:**
 - Dynamic memory allocation (no predefined size).
 - Efficient insertion and deletion (no need to shift elements).
 - **Disadvantages:**
 - Random access is not possible; elements must be accessed sequentially.
 - Extra memory is required for storing pointers.
5. What is the time complexity of concatenating two singly linked lists?
 - The time complexity is $O(n)$, where n is the length of the first list. You need to traverse the first list to reach its last node, then link it to the head of the

second list. Traversing the second list is not required.

Experiments 11

WAP to remove the Top element of the Stack.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Stack structure definition
struct Stack {
    int data[MAX];
    int top;
};

// Function to initialize the stack
void initStack(struct Stack *s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

// Function to push an element onto the stack
void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack overflow!\n");
        return;
    }
    s->data[++(s->top)] = value;
    printf("%d pushed to stack\n", value);
}

// Function to remove the top element from the stack
void pop(struct Stack *s) {
    if (isEmpty(s)) {
```

```
        printf("Stack is empty, nothing to pop!\n");
        return;
    }
    printf("Popped element: %d\n", s->data[(s->top)--]);
}

// Function to display the current stack
void display(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = 0; i <= s->top; i++) {
        printf("%d ", s->data[i]);
    }
    printf("\n");
}

int main() {
    struct Stack s;
    initStack(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    display(&s);

    // Remove the top element
    pop(&s);
    display(&s);

    return 0;
}
```

Output:

The screenshot shows the Programiz C Online Compiler interface. On the left, the code for 'main.c' is displayed, and on the right, the 'Output' window shows the results of the program execution.

```

main.c
38- void pop(struct Stack *s) {
39-     if (isEmpty(s)) {
40-         printf("Stack is empty, nothing to pop!\n");
41-         return;
42-     }
43-     printf("Popped element: %d\n", s->data[(s->top)--]);
44- }
45
46 // Function to display the current stack
47- void display(struct Stack *s) {
48-     if (isEmpty(s)) {
49-         printf("Stack is empty!\n");
50-         return;
51-     }
52-     printf("Stack elements: ");
53-     for (int i = 0; i <= s->top; i++) {
54-         printf("%d ", s->data[i]);
55-     }
56-     printf("\n");
57- }
58
59- int main() {
60-     struct Stack s;
61-     initStack(&s);
62
63-     push(&s, 10);
64-     push(&s, 20);
65-     push(&s, 30);
66-     display(&s);
67
68-     // Remove the top element
69-     pop(&s);
70-     display(&s);
71
72-     return 0;

```

Output:

```

/tmp/mEd77R4L0K.o
10 pushed to stack
20 pushed to stack
30 pushed to stack
Stack elements: 10 20 30
Popped element: 30
Stack elements: 10 20

=== Code Execution Successful ===

```

Explanation:

1. **initStack**: Initializes the stack with a **top** pointer set to **-1**.
2. **isEmpty**: Checks if the stack is empty by checking if **top** is **-1**.
3. **isFull**: Checks if the stack is full by comparing **top** with **MAX-1**.
4. **push**: Adds a new element to the stack if it's not full.
5. **pop**: Removes the top element from the stack if it's not empty.
6. **display**: Displays all elements in the stack.

Viva Questions:

1. What is the significance of the **top** variable in a stack?
 - The **top** variable indicates the index of the current top element in the stack.
2. What happens if you try to pop an element from an empty stack?
 - It results in "Stack Underflow," meaning there are no elements to remove.
3. What is the time complexity of the **push** and **pop** operations in this stack implementation?
 - Both **push** and **pop** operations have a time complexity of $O(1)$ because they involve simple index manipulation.
4. How would you modify this program to implement a stack using a dynamic array instead of a fixed-size array?
 - By dynamically allocating memory using **malloc** and resizing the array

using `realloc` when the stack grows beyond the current size.

5. What is the difference between stack overflow and stack underflow?
 - Stack overflow occurs when attempting to `push` an element onto a full stack, while stack underflow occurs when trying to `pop` an element from an empty stack.

Experiment 12

WAP to insert (or push) an element at the Top of the Stack.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Stack structure definition
struct Stack {
    int data[MAX];
    int top;
};

// Function to initialize the stack
void initStack(struct Stack *s) {
    s->top = -1;
}

// Function to check if the stack is full
int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack overflow!\n");
        return;
    }
    s->data[++(s->top)] = value;
    printf("%d pushed to stack\n", value);
}
```

```

}

// Function to display the current stack
void display(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = 0; i <= s->top; i++) {
        printf("%d ", s->data[i]);
    }
    printf("\n");
}

int main() {
    struct Stack s;
    initStack(&s);

    // Pushing elements to stack
    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    // Display current stack
    display(&s);

    return 0;
}

```

Output:

 Programiz
C Online Compiler

main.c	Output
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 #define MAX 100 5 //Dr Amar Nath CSE SLIET Longowal 6 // Stack structure definition 7- struct Stack { 8 int data[MAX]; 9 int top; 10 }; 11 12 // Function to initialize the stack 13- void initStack(struct Stack *s) { 14 s->top = -1; 15 } 16 17 // Function to check if the stack is full 18- int isFull(struct Stack *s) { 19 return s->top == MAX - 1; 20 } 21 22 // Function to check if the stack is empty 23- int isEmpty(struct Stack *s) { 24 return s->top == -1; 25 } 26 27 // Function to push an element onto the stack 28- void push(struct Stack *s, int value) { 29- if (isFull(s)) { 30 printf("Stack overflow!\n"); 31 return; 32 } 33 s->data[++(s->top)] = value; 34 printf("%d pushed to stack\n", value); 35 } 36 37 // Function to display the current stack </pre>	<pre> /tmp/t140pIzX35.o 10 pushed to stack 20 pushed to stack 30 pushed to stack Stack elements: 10 20 30 === Code Execution Successful === </pre>

Viva questions

1. What is the difference between a Stack and a Queue?
 - A Stack follows the LIFO (Last-In-First-Out) principle, meaning the last element inserted is the first to be removed.
 - A Queue follows the FIFO (First-In-First-Out) principle, meaning the first element inserted is the first to be removed.
2. How can you prevent Stack Overflow?
 - A stack overflow occurs when you try to push an element onto a stack that is already full. Always check if the stack is full using an `isFull()` function before pushing an element to prevent this.
3. What is the time complexity of the `push` operation?
 - The time complexity of the `push` operation is $O(1)$ because it only involves updating the top pointer and inserting the element.
4. What are the limitations of using an array-based stack?
 - An array-based stack has a fixed size, meaning once the stack is full, no more elements can be added (stack overflow). This limitation can be overcome by using a dynamic structure like a linked list for the stack.
5. What is the primary application of a stack in recursion?
 - In recursion, the function calls are stored in a call stack. When a function is called, its data is pushed onto the stack. When the function returns, the data is popped from the stack. This allows recursive functions to return control back to the previous function.

Experiments 13

WAP to insert an element at the end of the queue.

```
#include <stdio.h>
#define MAX 100

// Queue structure definition
struct Queue {
    int data[MAX];
    int front;
    int rear;
};
```

```
// Function to initialize the queue
void initQueue(struct Queue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(struct Queue *q) {
    return q->rear == MAX - 1;
}

// Function to check if the queue is empty
int isEmpty(struct Queue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function to insert an element at the end of the queue
void enqueue(struct Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue overflow!\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->data[++(q->rear)] = value;
    printf("%d inserted into the queue\n", value);
}

// Function to display the current queue
void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->data[i]);
    }
    printf("\n");
}

int main() {
    struct Queue q;
    initQueue(&q);

    // Enqueue elements into the queue
    enqueue(&q, 10);
```

```
enqueue(&q, 20);
enqueue(&q, 30);

// Display current queue
display(&q);

return 0;
}
```

Output:

10 inserted into the queue
20 inserted into the queue
30 inserted into the queue
Queue elements: 10 20 30

Viva Questions:

1. What is the difference between a Queue and a Stack?
 - A Queue follows the FIFO (First-In-First-Out) principle, meaning the first element inserted is the first to be removed. A Stack follows the LIFO (Last-In-First-Out) principle, where the last element inserted is the first to be removed.
2. What is the time complexity of the **enqueue** operation?
 - The time complexity of the **enqueue** operation is $O(1)$ because it only involves updating the rear pointer and adding the element at the end.
3. What happens when you try to insert an element into a full queue?
 - If you try to insert an element into a full queue, the operation fails and results in a queue overflow. Before attempting to enqueue an element, you should check if the queue is full using the **isFull()** function.
4. What are the types of queues?
 - There are several types of queues: simple queue (linear queue), circular queue, priority queue, and double-ended queue (deque).
5. How is a queue different from a circular queue?
 - In a queue, once the rear reaches the end of the array, no more elements can be added, even if spaces are at the front. In a circular queue, the rear can wrap around to the front of the array (circular fashion), allowing better space utilization.

Experiment 14**WAP to remove the first element of the queue.**

```
#include <stdio.h>
#define MAX 100

// Queue structure definition
struct Queue {
    int data[MAX];
    int front;
    int rear;
};

// Function to initialize the queue
void initQueue(struct Queue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is empty
int isEmpty(struct Queue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function to remove the first element of the queue (dequeue)
void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue underflow! Cannot dequeue.\n");
        return;
    }
    printf("Removed element: %d\n", q->data[q->front++]);

    // Reset the queue if it's empty after dequeue
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
}

// Function to insert an element at the end of the queue (enqueue)
void enqueue(struct Queue *q, int value) {
    if (q->rear == MAX - 1) {
        printf("Queue overflow!\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;
    }
}
```

```
    q->data[++(q->rear)] = value;
}

// Function to display the current queue
void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->data[i]);
    }
    printf("\n");
}

int main() {
    struct Queue q;
    initQueue(&q);

    // Enqueue elements into the queue
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);

    // Display current queue
    display(&q);

    // Dequeue the first element
    dequeue(&q);

    // Display current queue after dequeue
    display(&q);

    return 0;
}
```

Output:**Queue elements: 10 20 30****Removed element: 10****Queue elements: 20 30****Explanation:**

1. `initQueue`: Initializes the queue with `front` and `rear` pointers set to `-1`.
2. `isEmpty`: Checks if the queue is empty by verifying if `front` is `-1` or `front > rear`.

3. dequeue: Removes the first element from the queue by incrementing the **front** pointer. If the queue becomes empty after dequeuing, it resets both **front** and **rear** to **-1**.
4. enqueue: Adds an element to the end of the queue if it's not full.
5. display: Displays all the elements currently in the queue.

Viva Questions:

1. What is the difference between **enqueue** and **dequeue** operations in a queue?
 - Enqueue inserts an element at the rear (end) of the queue, while dequeue removes the element from the front (start) of the queue.
2. What happens when you try to dequeue an element from an empty queue?
 - Trying to dequeue an element from an empty queue results in a queue underflow, meaning no elements are left to remove.
3. What is the time complexity of the **dequeue** operation?
 - The time complexity of the **dequeue** operation is $O(1)$, as it only involves incrementing the **front** pointer and checking conditions.
4. What are the real-world examples of a queue?
 - Some real-world examples of a queue are:
 - Waiting in line at a bank or a movie theater.
 - Job scheduling in operating systems.
 - Print queue in a printer.
5. How is memory managed in a simple queue implementation using arrays?
 - In a simple queue, memory is allocated as a fixed-size array. No more elements can be added once the **rear** reaches the maximum array size. Memory is not reused unless a circular queue is implemented.

Experiment 15**WAP to remove the first element of the queue.**

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100 // Maximum size of the queue

// Queue structure
typedef struct {
    int items[MAX];
    int front;
    int rear;
} Queue;
```



```
// Function to initialize the queue
void initialize(Queue* q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is empty
int isEmpty(Queue* q) {
    return q->front == -1;
}

// Function to check if the queue is full
int isFull(Queue* q) {
    return q->rear == MAX - 1;
}

// Function to insert an element into the queue
void enqueue(Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow! Unable to enqueue %d\n", value);
        return;
    }
    if (q->front == -1) {
        q->front = 0; // Set front to 0 when the first element is inserted
    }
    q->items[++(q->rear)] = value;
    printf("%d enqueued to queue\n", value);
}

// Function to remove the first element from the queue
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow! No element to dequeue\n");
        return -1;
    }
    int removedElement = q->items[q->front];

    // If there's only one element in the queue, reset front and rear
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }

    return removedElement;
}

// Function to display the queue
void display(Queue* q) {
```

```
if (isEmpty(q)) {
    printf("Queue is empty\n");
    return;
}
printf("Queue elements: ");
for (int i = q->front; i <= q->rear; i++) {
    printf("%d ", q->items[i]);
}
printf("\n");
}

// Main function
int main() {
    Queue q;
    initialize(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);

    display(&q); // Display queue before dequeuing

    int removedElement = dequeue(&q); // Remove the first element
    if (removedElement != -1) {
        printf("Removed first element: %d\n", removedElement);
    }

    display(&q); // Display queue after dequeuing

    return 0;
}
```

Output.

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
Queue elements: 10 20 30 40
Removed first element: 10
Queue elements: 20 30 40
```

Viva Questions:

1. What is the difference between **front** and **rear** in a queue?
 - **front** represents the index of the first element, while **rear** represents the index of the last element in the queue.
2. What happens if you try to dequeue an element from an empty queue?
 - It results in "Queue Underflow," meaning there are no elements to remove from the queue.
3. What is the time complexity of the **enqueue** and **dequeue** operations in this queue implementation?
 - Both **enqueue** and **dequeue** operations have a time complexity of $O(1)$ because they involve simple index manipulations.
4. How would you implement a circular queue using this array-based queue structure?
 - By modifying the **enqueue** and **dequeue** functions to wrap around the array using the modulo operator, i.e., $(\text{rear} + 1) \% \text{MAX}$.
5. What is the primary difference between a stack and a queue?
 - A stack follows the LIFO (Last In, First Out) principle, while a queue follows the FIFO (First In, First Out) principle.

Experiment 16**WAP to illustrate the implementation of Binary Search Tree.**

```
#include <stdio.h>
#include <stdlib.h>

// Structure for a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a new node in the BST
struct Node* insert(struct Node* root, int data) {
```

```
if (root == NULL) {
    return createNode(data);
}
if (data < root->data) {
    root->left = insert(root->left, data);
} else {
    root->right = insert(root->right, data);
}
return root;
}

// Function to perform inorder traversal of the BST
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Function to search for a value in the BST
struct Node* search(struct Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    } else {
        return search(root->right, data);
    }
}

int main() {
    struct Node* root = NULL;

    // Inserting nodes into the BST
    root = insert(root, 15);
    insert(root, 10);
    insert(root, 20);
    insert(root, 8);
    insert(root, 12);
    insert(root, 17);
    insert(root, 25);

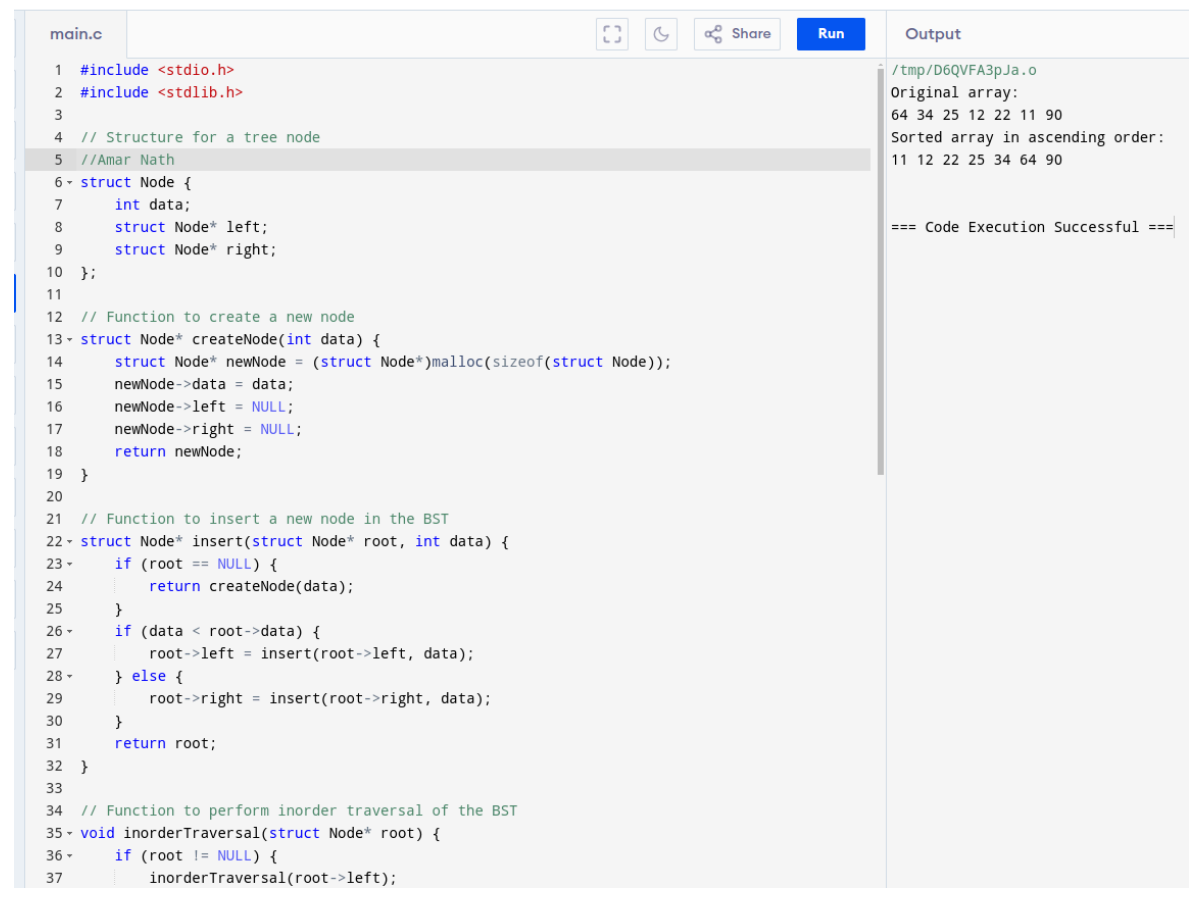
    // Inorder traversal of the BST
    printf("Inorder traversal of the BST:\n");
    inorderTraversal(root);
    printf("\n");
}
```

```
// Searching for a value in the BST
int searchValue = 10;
struct Node* result = search(root, searchValue);
if (result != NULL) {
    printf("Value %d found in the BST.\n", searchValue);
} else {
    printf("Value %d not found in the BST.\n", searchValue);
}

return 0;
}
```

Output:

 Programiz
C Online Compiler



```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Structure for a tree node
5 //Amar Nath
6 struct Node {
7     int data;
8     struct Node* left;
9     struct Node* right;
10 };
11
12 // Function to create a new node
13 struct Node* createNode(int data) {
14     struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
15     newNode->data = data;
16     newNode->left = NULL;
17     newNode->right = NULL;
18     return newNode;
19 }
20
21 // Function to insert a new node in the BST
22 struct Node* insert(struct Node* root, int data) {
23     if (root == NULL) {
24         return createNode(data);
25     }
26     if (data < root->data) {
27         root->left = insert(root->left, data);
28     } else {
29         root->right = insert(root->right, data);
30     }
31     return root;
32 }
33
34 // Function to perform inorder traversal of the BST
35 void inorderTraversal(struct Node* root) {
36     if (root != NULL) {
37         inorderTraversal(root->left);
```

```
/tmp/D6QVFA3pJa.o
Original array:
64 34 25 12 22 11 90
Sorted array in ascending order:
11 12 22 25 34 64 90

=== Code Execution Successful ===
```

Explanation:

1. **Node Structure:** Each node has data, a pointer to the left child, and a pointer to the right child.
2. **createNode:** Allocates memory for a new node and initializes its data and pointers.
3. **insert:** Recursively inserts a new value into the BST. If the tree is empty, it creates a new node; otherwise, it finds the correct position based on the value.
4. **inorderTraversal:** Prints the values of the BST in ascending order.

5. **search:** Searches for a specific value in the BST and returns the node if found.
6. **main:** Demonstrates insertion of nodes, performs inorder traversal, and searches for a specific value.

Experiment 17.

WAP to sort an array of integers in ascending order using Bubble Sort.

```
#include <stdio.h>

// Function to perform bubble sort
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            // Swap if the current element is greater than the next element
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, n);

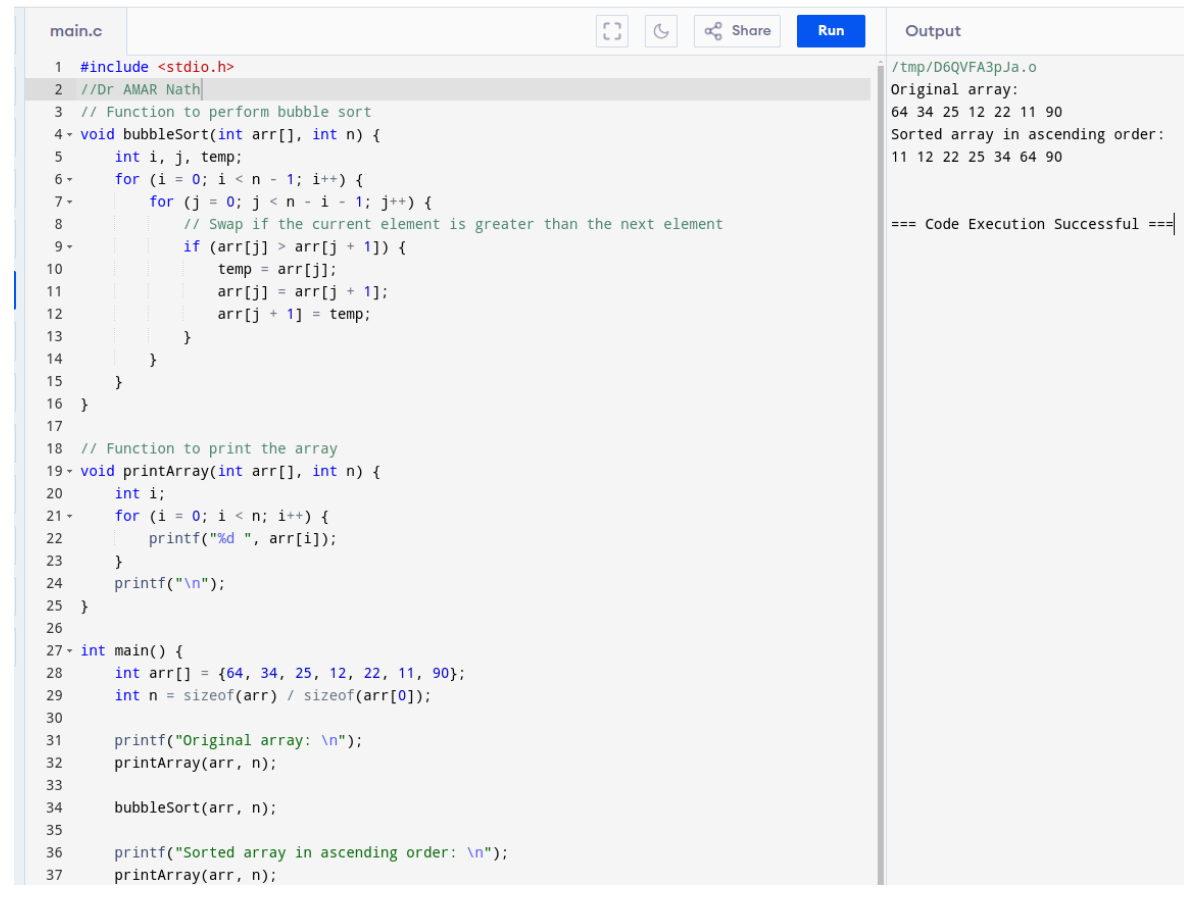
    bubbleSort(arr, n);

    printf("Sorted array in ascending order: \n");
    printArray(arr, n);
}
```

```
return 0;
}
```

Output:


C Online Compiler



```
main.c
1 #include <stdio.h>
2 //Dr AMAR Nath
3 // Function to perform bubble sort
4 void bubbleSort(int arr[], int n) {
5     int i, j, temp;
6     for (i = 0; i < n - 1; i++) {
7         for (j = 0; j < n - i - 1; j++) {
8             // Swap if the current element is greater than the next element
9             if (arr[j] > arr[j + 1]) {
10                temp = arr[j];
11                arr[j] = arr[j + 1];
12                arr[j + 1] = temp;
13            }
14        }
15    }
16 }
17
18 // Function to print the array
19 void printArray(int arr[], int n) {
20     int i;
21     for (i = 0; i < n; i++) {
22         printf("%d ", arr[i]);
23     }
24     printf("\n");
25 }
26
27 int main() {
28     int arr[] = {64, 34, 25, 12, 22, 11, 90};
29     int n = sizeof(arr) / sizeof(arr[0]);
30
31     printf("Original array: \n");
32     printArray(arr, n);
33
34     bubbleSort(arr, n);
35
36     printf("Sorted array in ascending order: \n");
37     printArray(arr, n);
}
```

Output

```
/tmp/D6QVFA3pJa.o
Original array:
64 34 25 12 22 11 90
Sorted array in ascending order:
11 12 22 25 34 64 90

=== Code Execution Successful ===
```

Viva Questions:

1. How does Bubble Sort work?
 - Bubble Sort works by repeatedly swapping adjacent elements if they are in the wrong order. The largest unsorted element "bubbles up" to its correct position in each pass.
2. What is the time complexity of Bubble Sort?
 - The time complexity of Bubble Sort is $O(n^2)$ in the worst and average cases, where n is the number of elements in the array.
3. Can Bubble Sort be optimized?
 - Yes, if no elements are swapped in a pass, it means the array is already sorted, and further passes are unnecessary. This can be checked using a flag, reducing unnecessary iterations.
4. What is the space complexity of Bubble Sort?

- The space complexity of Bubble Sort is $O(1)$, since it only requires a constant amount of extra space (i.e., the swap variable).
5. What are the limitations of Bubble Sort?
- Bubble Sort is inefficient for large datasets because of its quadratic time complexity. It's mainly used for educational purposes or when the dataset is small and nearly sorted.

Experiment 18

WAP to sort an array of integers in ascending order using Insertion Sort.

```
#include <stdio.h>

// Function to perform insertion sort
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key, to one position ahead
        // of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
```



```
printf("Original array: \n");
printArray(arr, n);

insertionSort(arr, n);

printf("Sorted array in ascending order: \n");
printArray(arr, n);

return 0;
}
```

Output:**Original array:****12 11 13 5 6****Sorted array in ascending order:****5 6 11 12 13****Viva Questions:**

1. How does Insertion Sort work?
 - Insertion Sort works by dividing the array into a sorted and an unsorted part. It picks elements from the unsorted part individually and places them correctly in the sorted part.
2. What is the time complexity of Insertion Sort?
 - The time complexity of Insertion Sort is $O(n^2)$ in the worst and average cases and $O(n)$ in the best case (when the array is already sorted).
3. What is the space complexity of Insertion Sort?
 - The space complexity is $O(1)$ because it requires constant extra space.
4. When is Insertion Sort preferred over other sorting algorithms?
 - Insertion Sort is preferred when the dataset is small or nearly sorted since its best-case time complexity is $O(n)$.
5. How is Insertion Sort different from Bubble Sort?
 - Insertion Sort places each element in its correct position right from the beginning, while Bubble Sort compares adjacent elements and swaps them. Insertion Sort tends to be more efficient than Bubble Sort, especially for small or partially sorted datasets.

Experiment 19.**WAP to sort an array of integer in Ascending Order using Quick Sort.**

```
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as pivot
    int i = (low - 1);    // Index of smaller element

    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]); // Swap the pivot element with the element at i + 1
    return (i + 1); // Return the partitioning index
}

// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
```

```
    }
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array in ascending order:\n");
    printArray(arr, n);

    return 0;
}
```

Output:**Original array:****10 7 8 9 1 5****Sorted array in ascending order:****1 5 7 8 9 10****Explanation:**

1. swap: A helper function that swaps two integers in the array.
2. partition: This function selects a pivot (the last element) and rearranges the array of elements, placing all smaller elements to the left of the pivot and all larger elements to the right. It returns the final position of the pivot.
3. quickSort: This recursive function applies the QuickSort algorithm. It partitions the array and recursively sorts the sub-arrays.
4. printArray: A utility function to print the elements of the array.
5. main: Initializes an array, prints it, calls the **quickSort** function, and then prints the sorted array.

Viva Questions:

1. What is Quick Sort?
 - Quick Sort is a divide-and-conquer sorting algorithm that selects a 'pivot' element and partitions the array into two sub-arrays: elements less than the

- pivot and elements greater than the pivot. It then recursively sorts the sub-arrays.
2. What is the average and worst-case time complexity of Quick Sort?
 - The average-case time complexity of Quick Sort is $O(n \log n)$, while the worst-case time complexity is $O(n^2)$, which occurs when the smallest or largest element is always chosen as the pivot.
 3. What is the role of the pivot in Quick Sort?
 - The pivot is an element used to partition the array into sub-arrays. Its position determines the division of elements into those less than or equal to the pivot and those greater than the pivot.
 4. How does Quick Sort compare to other sorting algorithms like Merge Sort and Bubble Sort?
 - Quick Sort is generally faster than Bubble Sort, which has a time complexity of $O(n^2)$, and it performs well in practice compared to Merge Sort, which has a consistent $O(n \log n)$ time complexity but requires additional space for the temporary arrays used during merging.
 5. Can Quick Sort be implemented iteratively? If so, how?
 - Yes, Quick Sort can be implemented iteratively using an explicit stack to hold the sub-arrays low and high indices instead of recursion. This avoids the potential risk of stack overflow with large arrays.

Experiment 20.

WAP to search an element using the Linear Search Method.

```
#include <stdio.h>

// Function to perform linear search
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        // Check if the current element is equal to the target
        if (arr[i] == target) {
            return i; // Return the index if found
        }
    }
    return -1; // Return -1 if not found
}

int main() {
    int arr[] = {34, 78, 12, 90, 23, 56};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target;

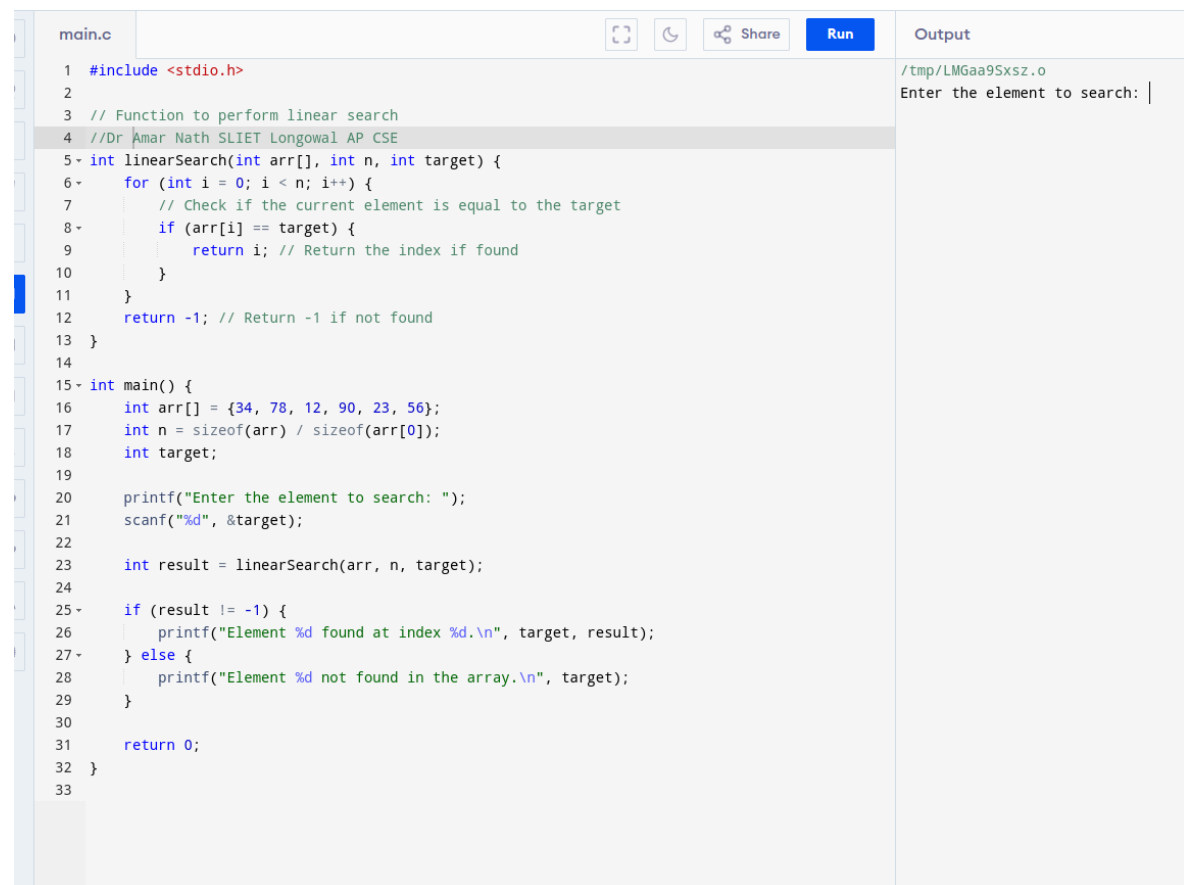
    printf("Enter the element to search: ");
    scanf("%d", &target);
```

```
int result = linearSearch(arr, n, target);

if (result != -1) {
    printf("Element %d found at index %d.\n", target, result);
} else {
    printf("Element %d not found in the array.\n", target);
}

return 0;
}
```

C Online Compiler



The screenshot shows an online C compiler interface. On the left, there is a code editor with a file named 'main.c'. The code defines a function 'linearSearch' that iterates through an array to find a target value. The 'main' function initializes an array with values {34, 78, 12, 90, 23, 56}, prompts the user to enter a target value, and calls the 'linearSearch' function. The output window on the right shows the prompt 'Enter the element to search: |'.

```
main.c
1 #include <stdio.h>
2
3 // Function to perform linear search
4 //Dr Amar Nath SLIET Longowal AP CSE
5 int linearSearch(int arr[], int n, int target) {
6     for (int i = 0; i < n; i++) {
7         // Check if the current element is equal to the target
8         if (arr[i] == target) {
9             return i; // Return the index if found
10        }
11    }
12    return -1; // Return -1 if not found
13 }
14
15 int main() {
16     int arr[] = {34, 78, 12, 90, 23, 56};
17     int n = sizeof(arr) / sizeof(arr[0]);
18     int target;
19
20     printf("Enter the element to search: ");
21     scanf("%d", &target);
22
23     int result = linearSearch(arr, n, target);
24
25     if (result != -1) {
26         printf("Element %d found at index %d.\n", target, result);
27     } else {
28         printf("Element %d not found in the array.\n", target);
29     }
30
31     return 0;
32 }
33
```

Output
/tmp/LMGaa9Sxsz.o
Enter the element to search: |

Viva Questions:

1. What is Linear Search?
 - Linear Search is a simple search algorithm that checks each list element sequentially until the desired element is found or the list ends.
2. What is the time complexity of Linear Search?
 - The time complexity of Linear Search is $O(n)$, where n is the number of

- elements in the array. In the worst case, every element must be checked.
3. What are the advantages of Linear Search?
 - Linear Search is easy to implement, works on sorted and unsorted arrays, and does not require additional memory for data structures.
 4. When would you prefer Linear Search over more complex algorithms like Binary Search?
 - Linear Search is preferred when dealing with small datasets or when the array is unsorted. Binary Search is more efficient for sorted arrays with a time complexity of $O(\log n)$.
 5. What are the limitations of Linear Search?
 - The main limitation is its inefficiency for large datasets, which may require checking every element in the worst case, leading to longer search times than more efficient algorithms like Binary Search.

Experiment 20

WAP to search an element using the Binary Search Method.

```
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int size, int target) {
    int left = 0;           // Starting index
    int right = size - 1;  // Ending index

    while (left <= right) {
        int mid = left + (right - left) / 2; // Calculate middle index

        // Check if the target is present at mid
        if (arr[mid] == target) {
            return mid; // Return index if found
        }
        // If target is greater, ignore left half
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        // If target is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }
    return -1; // Return -1 if not found
}

int main() {
```

```
int arr[] = {12, 23, 34, 56, 78, 90}; // Array must be sorted
int n = sizeof(arr) / sizeof(arr[0]);
int target;

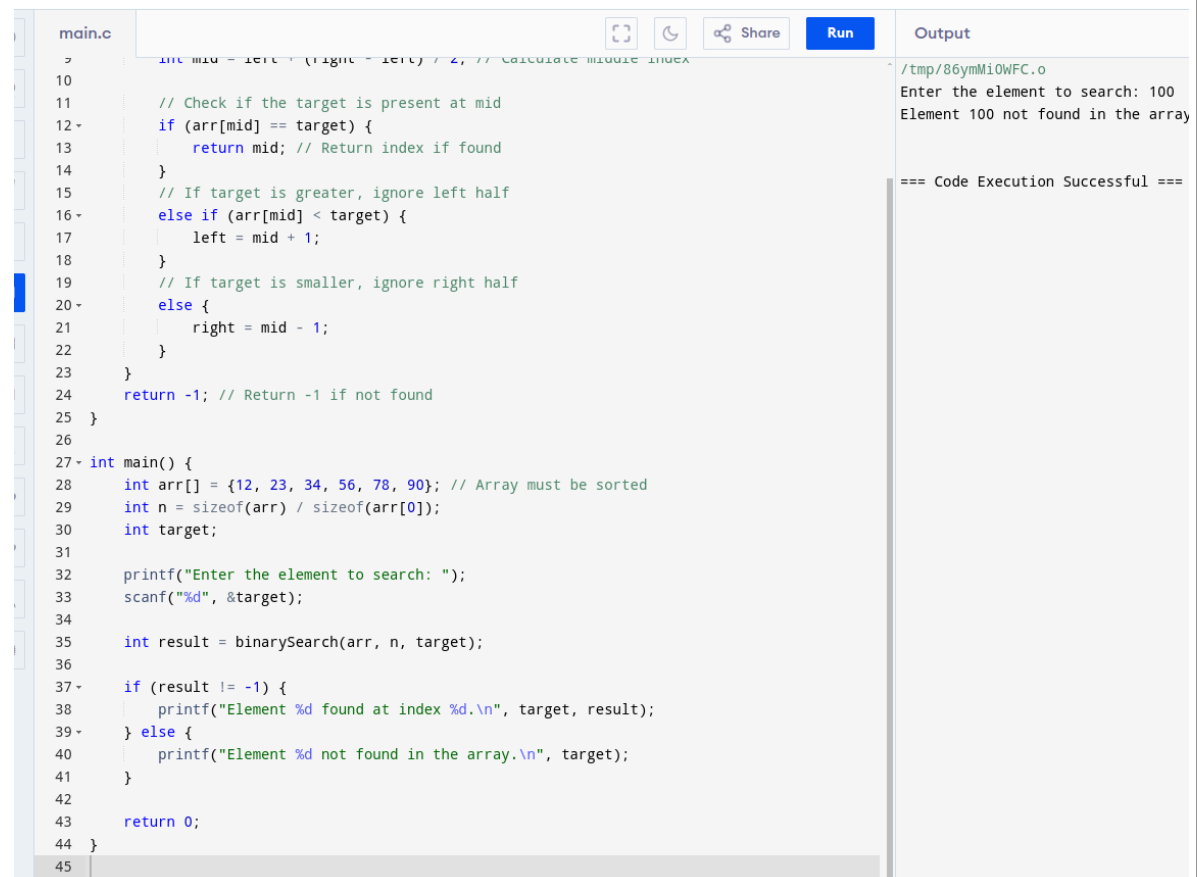
printf("Enter the element to search: ");
scanf("%d", &target);

int result = binarySearch(arr, n, target);

if (result != -1) {
    printf("Element %d found at index %d.\n", target, result);
} else {
    printf("Element %d not found in the array.\n", target);
}

return 0;
}
```

C Online Compiler



The screenshot displays an online C compiler interface. On the left, a code editor shows a C program for binary search. The code defines an array of integers, prompts the user for a target value, and calls a `binarySearch` function. The function uses a recursive-like approach to find the target in a sorted array. The output window on the right shows the program's execution with the input '100' and the message 'Element 100 not found in the array'. A success message '=== Code Execution Successful ===' is also visible.

```
main.c
10 int mid = left + (right - left) / 2; // Calculate middle index
11
12 // Check if the target is present at mid
13 if (arr[mid] == target) {
14     return mid; // Return index if found
15 }
16 // If target is greater, ignore left half
17 else if (arr[mid] < target) {
18     left = mid + 1;
19 }
20 // If target is smaller, ignore right half
21 else {
22     right = mid - 1;
23 }
24 return -1; // Return -1 if not found
25 }
26
27 int main() {
28     int arr[] = {12, 23, 34, 56, 78, 90}; // Array must be sorted
29     int n = sizeof(arr) / sizeof(arr[0]);
30     int target;
31
32     printf("Enter the element to search: ");
33     scanf("%d", &target);
34
35     int result = binarySearch(arr, n, target);
36
37     if (result != -1) {
38         printf("Element %d found at index %d.\n", target, result);
39     } else {
40         printf("Element %d not found in the array.\n", target);
41     }
42
43     return 0;
44 }
45
```

Output

```
/tmp/86ymMi0WFC.o
Enter the element to search: 100
Element 100 not found in the array

=== Code Execution Successful ===
```

Viva Questions:

1. What is Binary Search?
 - Binary Search is a search algorithm that finds the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half.
2. What is the time complexity of Binary Search?
 - The time complexity of Binary Search is $O(\log n)$, where n is the number of elements in the array. This makes it much more efficient than linear search for large datasets.
3. What are the prerequisites for using Binary Search?
 - The array must be sorted in ascending or descending order before performing Binary Search.
4. How does Binary Search differ from Linear Search?
 - Binary Search divides the array into halves and eliminates one half from consideration at each step, while Linear Search checks each element sequentially.
5. What are the limitations of Binary Search?
 - The main limitation is that it requires the array to be sorted. Additionally, for very small arrays, the overhead of calculating midpoints and maintaining indices may not justify using Binary Search over Linear Search.