

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore-560028



A T M E

College of Engineering

**DEPARTMENT OF COMPUTER SCIENCE &
ENGINEERING (ACADEMIC YEAR 2025-26)**

LABORATORY MANUAL

SUBJECT: DATA STRUCTURES LABORATORY

SUB CODE: BCSL305

SEMESTER: III-2025CBCS Scheme

Composed by

Sathisha H D

verified by

Mr.RAGHURAM A S, Mr.SANDESH R

Dr. Shilpa B L

Approved by

Dr.PUTTEGOWDA D

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and international levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and international levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancement in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

Department of Computer Science & Engineering

Vision of the Department

- To develop highly talented individual sin Computer Science and Engineering to deal with real world challenges in industry, education, research and society.

Mission of the Department

- To inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.
- Motivate studentstoputtheirthoughtsandideasadoptablebyindustryortopursue higher studies leading to research.

Program Outcomes(POs)

Engineering Graduates will be able to:

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multi-disciplinary settings.

PO10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multi-disciplinary environments.

PO12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes(PSOs)

PSO1: Ability to apply skills in the field of algorithms, database design, web design, cloud computing and data analytics.

PSO2: Apply knowledge in the field of computer networks for building network and internet based applications.

Program Educational Objectives(PEOs):

1. Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.
2. Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.
3. Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.
4. Ability to function ethically and responsibly in a rapidly changing environment by applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.

DATA STRUCTURES LABORATORY		Scheme	2025
Course Code	1BCSL306	Semester	3
Teaching Hours/Week (L:T:P)	0:0:2	CIE Marks	50
Total Hours of Pedagogy L:T:P:SL:TW	0:0:28:2:30	SEE Marks	50
Credits	1	Total Marks	100
Examination type (SEE)	Practical	Exam Hours	03

Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

1. Develop C programs to implement fundamental data structures and their operations for solving computational problems.
2. Make use of appropriate data structures to organize, store, and manipulate data for solving real-world problems using C.
3. Build a laboratory record consisting of problem statements, algorithm/design, implementation, program outputs, and result with appropriate inferences.

Note:

1. The laboratory syllabus consists of PART-A and PART-B. While PART-A has 6 conventional experiments, PART-B has 6 typical open-ended experiments. The maximum marks for the laboratory course are 100.
2. Both PART-A and PART-B are considered for CIE and SEE.
3. Students have answer 1(one) question from PART-A and 1(one) question from PART-B.
 - a. The questions set for SEE shall be from among the experiments under PART-A. It is evaluated for **70 marks** out of the maximum 100 marks.
 - b. The open-ended question set for SEE shall be any other open-ended question and not selected from the experiments under PART-A. It shall be evaluated for **30 marks**.
4. For Continuous Internal Evaluation (CIE) during the semester, classwork shall typically include open-ended questions from **Part B**, along with other similar questions designed to enhance the students' skills.

PART - A FIXED SET OF EXPERIMENTS

1. Develop a Program in C for the following:

- a. Define a structure Book with the following fields: Book_ID, Title, Author, Price, Availability Status (Available/Issued)
- b. Dynamically allocate memory to store the details of N books, where N is entered by the user.
- c. Implement the following functions:
 - i. create() – To allocate memory and read the details of N books and store them in the allocated memory.
 - ii. display() – To display the details of all books available in the library.
 - iii. search() – To search for a book using Book ID and display the book details if found.
 - iv. issueBook() – To issue a book by updating its availability status and display an appropriate message.
 - v. returnBook() – To return an issued book by updating its availability status and display an appropriate message.
- d. Provide a menu-driven interface with the following options:
 - i. Add Book Records
 - ii. Display All Book Records

- iii. Search Book by Book ID
- iv. Issue a Book
- v. Return a Book
- vi. Exit

2. Develop a menu driven Program in C for the following operations on STACK of Integers (Array Implementation of Stack with maximum size MAX)

- a. Push an Element on to Stack
- b. Pop an Element from Stack
- c. Demonstrate how Stack can be used to check Palindrome
- d. Demonstrate Overflow and Underflow situations on Stack
- e. Display the status of Stack
- f. Exit

Support the program with appropriate functions for each of the above operations

3. Develop a program in C for simulating a Printer Queue using Queue data structure with the following operations:

- a. Add a print job to the Printer Queue.
- b. Process (delete) a print job from the Printer Queue.
- c. Display all pending print jobs in the Queue.
- d. Display the number of print jobs currently waiting in the Queue.
- e. Demonstrate Queue overflow and underflow conditions.
- f. Exit.

4. Develop a menu-driven Program in C for the following operations on a Singly Linked List of Student Records:

- a. Create a linked list using front insertion.
- b. Display all student records and count the number of nodes.
- c. Insert and delete nodes at the beginning and end of the list.
- d. Search for a student record using USN.

5. Develop a Program in C for the following:

- a. Create a Binary Tree by inserting integer elements entered by the user in level order.
- b. Implement the following functions:
 - i. createTree() – Create a binary tree using dynamic memory allocation.
 - ii. preorder() – Display the elements using Preorder Traversal.
 - iii. inorder() – Display the elements using Inorder Traversal.
 - iv. postorder() – Display the elements using Postorder Traversal.
- c. Provide a menu-driven interface with the following options:

- i. Create Binary Tree
- ii. Display Preorder Traversal
- iii. Display Inorder Traversal
- iv. Display Postorder Traversal
- v. Exit

6. Develop a Program in C for the following operations on Graph(G) of Cities

- a. Create an Adjacency Matrix representing the graph of N cities
- b. Print all the nodes reachable from a given starting node in a digraph using DFS/BFS method.

PART - B OPEN ENDED EXPERIMENTS

Open-ended experiments are a type of laboratory activity where the outcome is not predetermined and students are given the freedom to explore, design, and conduct the experiment based on the problem statements as per the concepts defined by the course coordinator. It encourages creativity, critical thinking, and inquiry-based learning.

1. Develop a C program to:

- a. Read two sparse matrices of the same order.
- b. Convert both into 3-tuple representation.
- c. Perform matrix addition.
- d. Display the resultant matrix in both normal matrix form and 3-tuple form.

2. Develop a C-based Expression Conversion tool that uses stack data structure to convert arithmetic expressions from infix notation to postfix notation. The tool should support alphanumeric operands, parentheses, and the operators (+, -, *, /, %, ^). Analyze the correctness of the conversion by testing various valid expressions.

3. Develop a menu-driven Program in C for the following operations on Circular Queue (integer/floating point data):

- a. Insert an element into the Circular Queue.
- b. Delete an element from the Circular Queue.
- c. Display the contents of the Circular Queue.
- d. Demonstrate overflow and underflow conditions.
- e. Exit.

4. Develop a menu driven Program in C for the following operations on Doubly Linked List (DLL) of Employee Data with the fields: SSN, Name, Dept, Designation, Sal, PhNo etc.

- a. Create a DLL of N Employees Data by using end insertion.
- b. Display the status of DLL and count the number of nodes in it
- c. Perform Insertion and Deletion at End of DLL
- d. Perform Insertion and Deletion at Front of DLL

- e. Demonstrate how this DLL can be used as Double Ended Queue.
- f. Exit

5. Develop a menu driven Program in C for the following operations on Binary Search Tree (BST) of Integers.

- a. Create a BST of N Integers
- b. Traverse the BST in Inorder, Preorder and Post Order
- c. Search the BST for a given element (KEY) and report the appropriate message
- d. Exit

6. Develop a Program in C for an Employee Record Management System using Hashing with the following operations:

- a. Create a hash table to store employee records using Employee ID as the key.
- b. Insert employee records into the hash table using the Hash function $H: K \rightarrow L$ as $H(K) = K \bmod m$
- c. Resolve collisions using Linear Probing.
- d. Search for an employee record using a given Employee ID and display the details.
- e. Display the contents of the hash table along with the corresponding memory locations.

CONTENTS

Sl. No.	EXPERIMENTNAME	Page No
1.	Introduction	
2.	Program1: Implementation of function create(), display(), search(), Issue Book(), return Book()	
3.	Program2: Implementation of Stack Operations	
4.	Program3: Simulating a Printer Queue Using data structure Operations	
5.	Program4: Implementation of Singly Linked List	
6.	Program5: Implementation of Binary Search tree	
7.	Program6: Implementation of Graphs (BFS & DFS Methods)	
8.	Program7: Implementation of Both normal matrix form and 3-tuple form.	
9.	Program8: Design, Develop and Implement Programing C for the following Stack Applications 1. Evaluation of Suffix expression with single digit operands and operators: +,-, *,/,%,^. 2. Solving Tower of Hanoi problem with n disks	
10.	Program9: Circular Queue Operations	
11.	Program10: Implementation of Doubly Linked List	
12.	Program11: Implementation of Binary Search Tree (BST) of Integers operations. like preorder. postorder.	
13.	Program12: Implementation of Hashing &Linear Probing	

Introduction to Data Structure

Basic Concepts:-

The logical or mathematical model of a particular organization of data is called data structures. Data structures is the study of logical relationship existing between individual data elements, the way the data is organized in the memory and the efficient way of storing, accessing and manipulating the data elements.

Data Structures can be classified as:

- Primitive data structures
- Non-Primitive data structures.

Primitive data structures are the basic data structures that can be directly manipulated/operated by machine instructions. Some of these are character, integer, real, pointers etc.

Non-primitive data structures are derived from primitive data structures, they cannot be directly manipulated/operated by machine instructions, and they are a group of homogeneous or heterogeneous data items. Some of these are Arrays, stacks, queues, trees, graphs etc.

Data structures are also classified as

- Linear data structures
- Non-Linear data structures.

In the Linear data structures processing of data is possible in linear fashion, i.e., data can be processed one by one sequentially.

Example of such data structures are:

- Array
- Linked list
- Stacks
- Queues

A data structure in which insertion and deletion is not possible in a linear fashion is called as non-linear data structure. i.e., which does not show the relationship of logical adjacency between the elements is called as non-linear data structure. Such as trees, graphs and files.

Data structure operations:

The particular data structures that one chooses for a given situation depends largely on the frequency with which specific operations are performed.

The following operations play major role in the processing of data.

- i) Traversing.
- ii) Searching.
- iii) Inserting.
- iv) Deleting.
- v) Sorting.
- vi) Merging

STACKS: -

A stack is an ordered collection of items in to which new items may be inserted and from which items may be deleted at the same end, called the TOP of the stack. A stack is a non-primitive linear data structure. 1 2 3 4 5

As all the insertion and deletion are done from the same end, the first element inserted into the stack is the last element deleted from the stack and the last element inserted into the stack is the first element to be deleted. Therefore, the stack is called Last-In First-Out (LIFO) data structure.

QUEUES: -

A queue is a non-primitive linear data structure. Where the operation on the queue is based on First-In-First-Out FIFO process — the first element in the queue will be the first one out. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be removed.

For inserting elements into the queue are done from the rear end and deletion is done from the front end, we use external pointers called as rear and front to keep track of the status of the queue. During insertion, Queue Overflow condition has to be checked. Like wise during deletion, Queue Underflow condition is checked.

APPLICATION OF QUEUE: -

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

- Serving request on, a single shared resource, like a printer, CPU task scheduling etc.
- In real life, Call Center phone systems will use Queues, to hold people calling the min an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

LINKED LIST

Disadvantages of static/sequential allocation technique:

- If an item has to be deleted then all the following items will have to be moved by one allocation. Wastage of time.
- Inefficient memory utilization.
- If no consecutive memory (free) is available, execution is not possible.

Linear Linked Lists

Types of Linked lists:

- Single Linked lists
- Circular Single Linked Lists
- Double Linked Lists
- Circular Double Linked Lists.

NODE:

Each node consists of two fields. Information (info) field and next address (next) field. The info field consists of actual information/data/item that has to be stored in a list. The second field next/link contains the address of the next node. Since next field contains the address,

It is of type pointer. Here the nodes in the list are logically adjacent to each other. Nodes that are physically adjacent need not be logically adjacent in the list.

The entire linked list is accessed from an external pointer FIRST that points to (contains the address of) the first node in the list. (By an external pointer, we mean, one that is not included within a node. Rather its value can be accessed directly by referencing a variable).

NODE

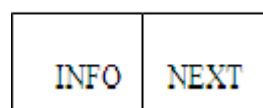


Fig.1 Linked List

The list containing 4 items/data 10, 20, 30 and 40 is shown below.

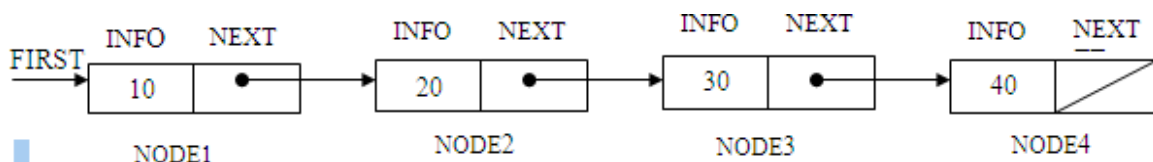


Fig.2 Linked List

The nodes in the list can be accessed using a pointer variable. In the above fig. FIRST is the pointer having the address of the first node of the list, initially before creating the list, as list is empty. The FIRST will always be initialized to NULL in the beginning. Once the list is created, FIRST contains the address of the first node of the list.

As each node is having only one link/next, the list is called single linked list and all the nodes are linked in one direction. Each node can be accessed by the pointer pointing (holding the address) to that node, Say P is pointer to a particular node, then the information field of that node can be accessed using info(P) and the next field can be accessed using next(P).

The arrows coming out of the next field in the fig. indicates that the address of the succeeding node is stored in that field. The link field of last node contains a special value known as NULL which is shown using a diagonal line pictorially. This NULL pointer is used to signal the end of a list.

The basic operations of linked lists are Insertion, Deletion and Display. A list is a dynamic data structure. The number of nodes on a list may vary dramatically as elements are inserted and deleted(removed).

The dynamic nature of list may be contrasted with the static nature of an array, whose size remains constant. When an item has to inserted, we will have to create a node, which has to be got from the available free memory of the computer system, So we shall use a mechanism to find an unused node which makes it available to us. For this purpose we shall use the get node operation (get node() function).

The C language provides the built-in functions like malloc(), calloc(), realloc() and free(), which are stored in alloc.h or stdlib.h header files. To dynamically allocate and release the memory locations from/to the computer system.

TREES:-

Definition:-

A data structure which is accessed beginning at the root node. Each node is either a leaf for an internal node. An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom. A tree can also be defined as a connected, acyclic di-graph.

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

Tree data structure is a collection of data(Node) which is organized in hierarchical structure and this is a recursive definition.

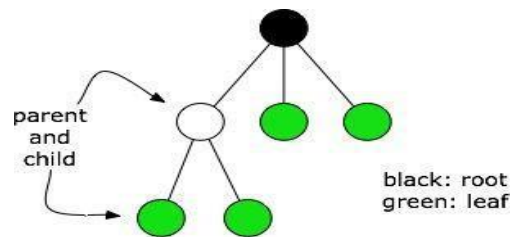


Figure: tree data structure

Fig-3 Tree data structure

Binary tree: A tree with most two children for each node.

Complete Binary Tree: A binary tree in which every level, except possibly the deepest, is completely filled. At depth n , the height of the tree, all nodes must be as far left as possible. Binary search tree: A binary tree where every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.

Tree traversal is a technique for processing the nodes of a tree in some order. The different tree traversal techniques are Pre-order, In-order and Post-order traversal. In Pre-order traversal, the tree node is visited first and the left subtree is traversed recursively and later right sub-tree is traversed recursively.

1. Develop a Program in C for the following:
 - a. Define a structure Book with the following fields: Book_ID, Title, Author, Price, Availability Status (Available/Issued)
 - b. Dynamically allocate memory to store the details of N books, where N is entered by the user.
 - c. Implement the following functions:
 - i. create() – To allocate memory and read the details of N books and store them in the allocated memory.
 - ii. display() – To display the details of all books available in the library.
 - iii. search() – To search for a book using Book ID and display the book details if found.
 - iv. issue Book() – To issue a book by updating its availability status and display an appropriate message.
 - v. return Book() – To return an issued book by updating its availability status and display an appropriate message.
 - d. Provide a menu-driven interface with the following options:
 - i. Add Book Records
 - ii. Display All Book Records
 - iii. Search Book by Book ID
 - iv. Issue a Book
 - v. Return a Book
 - vi. Exit.

To develop a **Library Management System in C** using **structures, dynamic memory allocation, functions, strings, pointers, and a menu-driven interface**. The program allows the user to add, display, search, issue, and return books.

In this program, a Book structure is used to store the details of each book:

```
struct Book
{
int Book_ID;
char Title[100];
char Author[100];
float Price;
char Availability_Status[20];
};
```

Each book contains:

- **Book_ID** – uniquely identifies the book.
- **Title** – stores the book title.
- **Author** – stores the author's name.
- **Price** – stores the price of the book.
- **Availability Status** – stores either "Available" or "Issued".

Advantages:-

- Memory is allocated according to the number of books entered by the user.
- Structures allow all book-related information to be stored together.
- Functions make the program modular and easier to understand.
- Book status can be updated without creating a new record.
- The menu-driven approach makes the program user-friendly.
- Dynamically allocated memory is released using free(), preventing unnecessary memory usage.

Program 1:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Book
{
int Book_ID;
char Title[100];
char Author[100];
float Price;
char Status[15]; // Available / Issued
};
struct Book *b = NULL;
int n = 0;
// Function to create book records
void create()
{
int i;
printf("Enter the number of books: ");
scanf("%d", &n);
b = (struct Book *)malloc(n * sizeof(struct Book));
if (b == NULL)
{
printf("Memory allocation failed!\n");
return;
}
for (i = 0; i < n; i++)
{
```

```

printf("\nEnter details of Book %d\n", i + 1);
printf("Book ID: ");
scanf("%d", &b[i].Book_ID);
getchar(); // Clear input buffer
printf("Title: ");
fgets(b[i].Title, sizeof(b[i].Title), stdin);
b[i].Title[strcspn(b[i].Title, "\n")] = '\0';
printf("Author: ");
fgets(b[i].Author, sizeof(b[i].Author), stdin); b[i].Author[strcspn(b[i].Author, "\n")] = '\0';
printf("Price: ");
scanf("%f", &b[i].Price);
strcpy(b[i].Status, "Available");
}
printf("\nBook records added successfully.\n");
}
// Function to display all books
void display(){
int i;
if (b == NULL)
{
printf("\nNo book records found.\n");
return;
}
printf("\n----- Book Records ----- \n");
for (i = 0; i < n; i++)
{
printf("\nBook %d\n", i + 1);
printf("Book ID : %d\n", b[i].Book_ID);
printf("Title   : %s\n", b[i].Title);
printf("Author  : %s\n", b[i].Author);
printf("Price   : %.2f\n", b[i].Price);
printf("Status  : %s\n", b[i].Status);
}
}
// Function to search a book
void search()
{

```

```
int id, i;
if (b == NULL)
{
printf("\nNo records available.\n");
return;
}
printf("Enter Book ID to search: ");
scanf("%d", &id);
for (i = 0; i < n; i++)
{
if (b[i].Book_ID == id)
{
printf("\nBook Found!\n");
printf("Book ID : %d\n", b[i].Book_ID);
printf("Title   : %s\n", b[i].Title);
printf("Author  : %s\n", b[i].Author);
printf("Price   : %.2f\n", b[i].Price);
printf("Status  : %s\n", b[i].Status);
return;
}
}
printf("Book not found.\n");
}

// Function to issue a book
void issueBook()
{
int id, i;
if (b == NULL)
{
printf("\nNo records available.\n");
return;
}
printf("Enter Book ID to issue: ");
scanf("%d", &id);
for (i = 0; i < n; i++)
{
if (b[i].Book_ID == id)
```

```

{
if (strcmp(b[i].Status, "Available") == 0)
{
strcpy(b[i].Status, "Issued");
printf("Book issued successfully.\n");
}
else
{
printf("Book is already issued.\n");
}
return;
}
}
printf("Book not found.\n");
}
// Function to return a book
void returnBook()
{
int id, i;
if (b == NULL)
{
printf("\nNo records available.\n");
return;
}
printf("Enter Book ID to return: ");
scanf("%d", &id);
for (i = 0; i < n; i++)
{
if (b[i].Book_ID == id)
{
if (strcmp(b[i].Status, "Issued") == 0)
{
strcpy(b[i].Status, "Available");
printf("Book returned successfully.\n");
}
Else
{

```

```
printf("Book is already available.\n");
}
return;
}
}
printf("Book not found.\n");
}
// Main Function
int main()
{
    int choice;
    do
    {
        printf("\n===== Library Management =====\n");
        printf("1. Add Book Records\n");
        printf("2. Display All Book Records\n");
        printf("3. Search Book by Book ID\n");
        printf("4. Issue a Book\n");
        printf("5. Return a Book\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                create();
                break;
            case 2:
                display();
                break;
            case 3:
                search();
                break;
            case 4:
                issueBook();
                break;
            case 5:
```

```
returnBook();
break;
case 6:
printf("Exiting program...\n");
break;
default:
printf("Invalid choice! Please try again.\n");
}
} while (choice != 6);
free(b);
return 0;
}
```

Output:-

Library Management

1. Add Book Records
2. Display All Book Records
3. Search Book by Book ID
4. Issue a Book
5. Return a Book
6. Exit

Enter your choice :1

Enter the number of books: 1

Enter details of Book 1

Book ID : 12

Title : 123

Author: 456

Price: 20

Book records added successfully.

Library Management

1. Add Book Records
2. Display All Book Records
3. Search Book by Book ID
4. Issue a Book
5. Return a Book
6. Exit

Enter your choice : 2

Book Records

Book 1

Book ID : 12

Title : 123

Author: 456

Price: 20.00

Status : Available

Library Management

1. Add Book Records
2. Display All Book Records
3. Search Book by Book ID
4. Issue a Book
5. Return a Book
6. Exit

Enter your choice : 3

Enter Book ID to Search: 12

Book Found!

Book ID: 12

Title : 123

Author: 456

Price: 20.00

Status : Available

Library Management

1. Add Book Records
2. Display All Book Records
3. Search Book by Book ID
4. Issue a Book
5. Return a Book
6. Exit

Enter your Choice: 4

Enter Book ID to issue: 12
Book issued successfully.

Library Management

1. Add Book Records
2. Display All Book Records
3. Search Book by Book ID
4. Issue a Book
5. Return a Book
6. Exit

Enter your Choice: 5

Enter Book ID to return: 12
Book returned successfully.

Library Management

1. Add Book Records
2. Display All Book Records
3. Search Book by Book ID
4. Issue a Book
5. Return a Book
6. Exit

Enter your Choice: 6
Exiting program...

2. Develop a menu driven Program in C for the following operations on STACK of Integers (Array Implementation of Stack with maximum size MAX)

- a. Push an Element on to Stack
- b. Pop an Element from Stack
- c. Demonstrate how Stack can be used to check Palindrome
- d. Demonstrate Overflow and Underflow situations on Stack
- e. Display the status of Stack
- f. Exit

Support the program with appropriate functions for each of the above operations.

Program objective:

- Understand the concept of palindrome.
- Understand the stack data structures.
- Understand the different functions on stacks i.e., push, pop and implement the same.
- Underst and stack overflow and underflow.

Algorithm:-

PUSH(item)

Step1: Read an element to be pushed on to stack item

Step2: check overflow condition of stack before inserting element into
stack Top=max-1

Step3: update the top pointer and insert an element into stack
Top=top+1

S[top]<-item

POP(item)

Step1: check underflow condition of stack before deleting element from stack top=-1

Step2: Display deleted element pointed by top
Deleted element<- s[top]

Step3:Decrementtoppointerby1
top<-top-1

Palindrome

Step1: Two pointers are required; one is pointed to top of stack
another is bottom of stack

Step2: compare top and bottom elements of stack if it is equal updated opened
bottom pointer by 1

Step3: if all elements are equal, then stack content is palindrome

THEORY

It is called as last in, first out. The element inserted first is the last one to be deleted. It is used for various applications like infix to postfix expression, postfix evaluation and for maintaining stack frames for function calling

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from top of the stack only. Likewise, Stack ADT allows all data operations at one end only.

At any given time, we can only access the top element of a stack. This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Below given diagram tries to depict a stack and its operations –

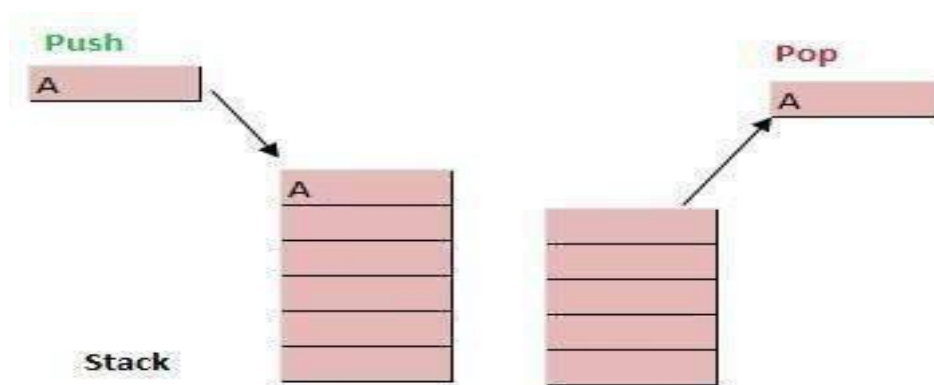


Fig4-ExampleofStack

A stack can be implemented by means of Array, Structure, Pointer and Linked-List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

Here, we are going to implement stack using arrays which makes it a fixed size stack implementation.

Basic Operations per formed on stack:

- **push()**-pushing(storing)an element on the stack.
- **pop()**-removing(accessing) an element from the stack.

To use a stack efficiently we need to check status of stack as well. For the same purpose, the following functionality is added to stacks;

- **peek()**—get the top data element of the stack, without removing it.
- **Is Full()**—check if stack is full.
- **Is Empty()**—check if stack is empty.

Program 2 :-

```
#include <stdio.h>
#include <string.h>
#define MAX 5
int stack[MAX];
int top = -1;
// Function to Push an element
void push(int item)
{
    if (top == MAX - 1)
    {
        printf("\nStack Overflow! Cannot insert %d\n", item);
    }
    else
    {
        stack[++top] = item;
        printf("\n%d pushed into stack.\n", item);
    }
}
// Function to Pop an element
void pop()
{
    if (top == -1)
    {
        printf("\nStack Underflow! Stack is empty.\n");
    }
}
```

```

}
else
{
printf("\nPopped element: %d\n", stack[top--]);
}
}
// Function to Display Stack
void display()
{
int i;
if (top == -1)
{
printf("\nStack is Empty.\n");
}
else
{
printf("\nStack Elements are:\n");
for (i = top; i >= 0; i--)
printf("%d\n", stack[i]);
}
}
// Function to Check Palindrome using Stack
void palindrome()
{
char str[100];
char rev[100];
int i, len;
char s[100];
int t = -1;
printf("Enter a string: ");
scanf("%s", str);
len = strlen(str);
//Push characters into stack
for (i = 0; i < len; i++)
s[++t] = str[i];
//Pop characters to reverse string
for (i = 0; i < len; i++)
rev[i] = s[t--];
rev[len] = '\0';
if (strcmp(str, rev) == 0)
printf("\n%s is a Palindrome.\n", str);
else
printf("\n%s is NOT a Palindrome.\n", str);
}

```

```

}
// Function to Demonstrate Overflow
void overflowDemo()
{
int i;
top = -1;
printf("\nDemonstrating Stack Overflow:\n");
for (i = 1; i <= MAX + 1; i++)
push(i * 10);
}
// Function to Demonstrate Underflow
void underflowDemo()
{
printf("\nDemonstrating Stack Underflow:\n");
top = -1;
pop();
}
// Main Function
int main()
{
int choice, item;
while (1)
{
printf("\n===== STACK MENU =====\n");
printf("1. Push\n");
printf("2. Pop\n");
printf("3. Check Palindrome\n");
printf("4. Demonstrate Overflow\n");
printf("5. Demonstrate Underflow\n");
printf("6. Display Stack\n");
printf("7. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice)
{
case 1:
printf("Enter element to push: ");
scanf("%d", &item);
push(item);
break;
case 2:
pop();
break;

```

```
case 3:
palindrome();
break;
case 4:
overflowDemo();
break;
case 5:
underflowDemo();
break;
case 6:
display();
break;
case 7:
printf("\nExiting Program...\n");
return 0;
default:
printf("\nInvalid Choice! Try Again.\n");
}
}
return 0;
}
```

Output:-

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice: 1

Enter element to push:1

1 pushed into stack.

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice: 2

Popped element: 1

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice: 3

Enter a string: 1

1 is a palindrome.

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice: 4

Demonstrating stack Overflow:

10 pushed into stack.
20 pushed into stack.
30 pushed into stack.
40 pushed into stack.
50 pushed into stack.

Stack Overflow! Cannot insert 60

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice: 5

Demonstrating Stack Underflow:

Stack Underflow! Stack is empty.

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice: 6

Stack is Empty.

STACK MENU

1. Push
2. Pop
3. Check palindrome
4. Demonstrate Overflow
5. Demonstrate underflow
6. Display Stack
7. Exit

Enter your choice:7

Exiting Program...

3. Develop a program in C for simulating a Printer Queue using Queue data structure with the following

operations:

- a. Add a print job to the Printer Queue.
- b. Process (delete) a print job from the Printer Queue.
- c. Display all pending print jobs in the Queue.
- d. Display the number of print jobs currently waiting in the Queue.
- e. Demonstrate Queue overflow and underflow conditions.
- f. Exit

Theory: Printer Queue Using Queue Data Structure

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. This means the element that is inserted first will be removed first, just like a printer processes print jobs in the order they are received.

In this program, the queue is used to simulate a **Printer Queue**, where each element represents a print job waiting to be processed.

Queue Operations

The program uses two important pointers:

- **Front** – points to the first print job in the queue. The job at the front is processed first.
- **Rear** – points to the last print job in the queue. New print jobs are added at the rear.

The main operations are:

1. Enqueue (Add Print Job)

A new print job is added at the rear of the queue. If the queue is full, a **Queue Overflow** condition occurs.

2. Dequeue (Process Print Job)

The print job at the front is removed and processed. If there are no jobs in the queue, a **Queue Underflow** condition occurs.

3. Display

Displays all the print jobs currently waiting in the queue from front to rear.

4. Count

Calculates the number of pending print jobs using:

$$\text{Number of jobs} = \text{rear} - \text{front} + 1$$

Overflow Condition

Since the program uses an array of size $MAX = 5$, it can store a maximum of 5 print jobs.

If the user tries to add another job when the queue is full:

Queue Overflow! Printer Queue is full.

Underflow Condition

If the user tries to process a print job when the queue is empty:

Queue Underflow! No print jobs to process.

Working Example

Suppose the following jobs are added:

Job1 → Job2 → Job3

Here:

Front		Rear
↓		↓
Job1	→	Job2 → Job3

When a job is processed, **Job1** is removed first:

Front		Rear
↓		↓
Job2	→	Job3

Thus, the queue follows **FIFO — First In, First Out**.

Explanation:-

The Printer Queue program demonstrates how a **queue data structure** can be used in a real-world application. It manages print jobs in the order they are received, using **enqueue** to add jobs and **dequeue** to process them. The program also demonstrates **overflow and underflow conditions**, making it useful for understanding the basic implementation and operation of queues in C.

Program 3 :-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 5
char queue[MAX][50];
int front = -1, rear = -1;
// Function to add a print job
void enqueue() {
    if (rear == MAX - 1) {
        printf("\nQueue Overflow! Printer Queue is full.\n");
        return;
    }
    if (front == -1)
        front = 0;
    rear++;
    printf("Enter Print Job Name: ");
    scanf(" %[^\\n]", queue[rear]);
    printf("Print Job \"%s\" added successfully.\n", queue[rear]);
}
// Function to process a print job
void dequeue() {
    if (front == -1 || front > rear) {
        printf("\nQueue Underflow! No print jobs in the queue.\n");
        front = rear = -1;
        return;
    }
    printf("\nProcessing Print Job: %s\n", queue[front]);
    front++;
    if (front > rear) {
        front = rear = -1;
    }
}
// Function to display all print jobs
void display() {
    int i;
```

```
if (front == -1) {
printf("\nPrinter Queue is Empty.\n");
return;
}
printf("\nPending Print Jobs:\n");
for (i = front; i <= rear; i++) {
printf("%d. %s\n", i - front + 1, queue[i]);
}
}
// Function to display number of print jobs
void countJobs() {
if (front == -1)
printf("\nNumber of waiting print jobs: 0\n");
else
printf("\nNumber of waiting print jobs: %d\n", rear - front + 1);
}
// Main Function
int main() {
int choice;
while (1) {
printf("\n===== Printer Queue Menu =====\n");
printf("1. Add Print Job\n");
printf("2. Process Print Job\n");
printf("3. Display Pending Print Jobs\n");
printf("4. Display Number of Waiting Jobs\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
enqueue();
break;
case 2:
dequeue();
break;
case 3:
display();
```

```
break;
case 4:
countJobs();
break;
case 5:
printf("\nExiting Program...\n");
exit(0);
default:
printf("\nInvalid Choice! Please try again.\n");
}
}
return 0;
}
```

Output:-

Printer Queue Menu

1. Add Print Job
2. Process Print Job
3. Display Pending Print jobs
4. Display Number of waiting Jobs
5. Exit

Enter your Choice: 1

Enter Print job Name: SS

Print job "SS" added successfully.

Printer Queue Menu

1. Add Print Job
2. Process Print Job
3. Display Pending Print jobs
4. Display Number of waiting Jobs
5. Exit

Enter your Choice: 2

Processing Print job: SS

Printer Queue Menu

1. Add Print Job

2. Process Print Job
3. Display Pending Print jobs
4. Display Number of waiting Jobs
5. Exit

Enter your Choice: 3

Printer Queue is Empty.

Printer Queue Menu

1. Add Print Job
2. Process Print Job
3. Display Pending Print jobs
4. Display Number of waiting Jobs
5. Exit

Enter your Choice: 4

Number of waiting print jobs: 0

Printer Queue Menu

1. Add Print Job
2. Process Print Job
3. Display Pending Print jobs
4. Display Number of waiting Jobs
5. Exit

Enter your Choice: 5

Exiting Program...

4. Develop a menu-driven Program in C for the following operations on a Singly Linked List of Student.

Records:

- a. Create a linked list using front insertion.
- b. Display all student records and count the number of nodes.
- c. Insert and delete nodes at the beginning and end of the list.
- d. Search for a student record using USN.

Program objective:

- Understand the Singly Linked List (SLL) data structures.
- Understand the methodology to insert and delete the element at the front of SLL.
- Understand the methodology to insert and delete the element at the end of SLL.
- Get the knowledge of how SLL can be used as both stack and queue.

Algorithm

Step 1: declare structure of node create empty list

head->null

Step 2: Insert at front

head<-null

return temp

if list is empty

temp->link=head

return head

Step 3: Insert at rear

head=null return temp

if list is empty

cur->head

while(cur!=null)

cur=cur->link

cur->link=temp;

return head

Step 4: Delete at front

head-

>link=null; return null

ull

if list has only one node

```
cur=head
```

```
head=head->link
```

```
free(cur)
```

Step 5: Delete at Rear end

```
head-
```

```
>link=nullreturnnu
```

```
ll
```

```
if only one node
```

```
cur<-head
```

```
while(cur!=null)
```

```
prev<-cur,cur=cur->link;
```

```
free(cur);
```

THEORY

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of an node in a single linked list is as follows...



Fig-7 Graphical Representation of Linked List

In a single linked list, the address of the first node is always stored in a reference node known as "front" (Sometimes it is also known as "head"). Always next part (reference part) of the last node must be NULL.

They are a dynamic in nature which allocates the memory when required.

- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Drawbacks:

Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.

Extra memory space for a pointer is required with each element of the list.

Program 4:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Student
{
char usn[20];
char name[30];
int sem;
struct Student *next;
};
struct Student *head = NULL;
// Create node
struct Student *createNode()
{
struct Student *newnode;
newnode = (struct Student *)malloc(sizeof(struct Student));
printf("Enter USN: ");
scanf("%s", newnode->usn);
printf("Enter Name: ");
scanf("%s", newnode->name);
printf("Enter Semester: ");
scanf("%d", &newnode->sem);
newnode->next = NULL;
return newnode;
}
// Create list using front insertion
void createFront()
{
int n, i;
struct Student *newnode;
printf("Enter number of students: ");
scanf("%d", &n);
for(i = 0; i < n; i++)
{
newnode = createNode();
newnode->next = head;
```

```
head = newnode;
}
}
// Display list
void display()
{
    struct Student *temp = head;
    int count = 0;
    if(head == NULL)
    {
        printf("List is empty.\n");
        return;
    }
    printf("\nStudent Records:\n");
    while(temp != NULL)
    {
        printf("USN : %s\n", temp->usn);
        printf("Name: %s\n", temp->name);
        printf("Sem : %d\n\n", temp->sem);
        count++;
        temp = temp->next;
    }
    printf("Total number of nodes = %d\n", count);
}
// Insert at beginning
void insertBegin()
{
    struct Student *newnode = createNode();
    newnode->next = head;
    head = newnode;
    printf("Node inserted at beginning.\n");
}
// Insert at end
void insertEnd()
{
    struct Student *newnode = createNode();
    struct Student *temp;
```

```
if(head == NULL)
{
head = newnode;
return;
}
temp = head;
while(temp->next != NULL)
temp = temp->next;
temp->next = newnode;
printf("Node inserted at end.\n");
}
// Delete at beginning
void deleteBegin()
{
struct Student *temp;
if(head == NULL)
{
printf("List is empty.\n");
return;
}
temp = head;
head = head->next;
printf("Deleted Student: %s\n", temp->usn);
free(temp);
}
// Delete at end
void deleteEnd()
{
struct Student *temp, *prev;
if(head == NULL)
{
printf("List is empty.\n");
return;
}
if(head->next == NULL)
{
printf("Deleted Student: %s\n", head->usn);
```

```
free(head);
head = NULL;
return;
}
prev = NULL;
temp = head;
while(temp->next != NULL)
{
prev = temp;
temp = temp->next;
}
prev->next = NULL;
printf("Deleted Student: %s\n", temp->usn);
free(temp);
}
// Search using USN
void searchUSN()
{
char key[20];
struct Student *temp = head;
printf("Enter USN to search: ");
scanf("%s", key);
while(temp != NULL)
{
if(strcmp(temp->usn, key) == 0)
{
printf("\nRecord Found\n");
printf("USN : %s\n", temp->usn);
printf("Name: %s\n", temp->name);
printf("Sem : %d\n", temp->sem);
return;
}
temp = temp->next;
}
printf("Student record not found.\n");
}
// Main
```

```
int main()
{
int choice;
while(1)
{
printf("\n===== MENU =====\n");
printf("1. Create List (Front Insertion)\n");
printf("2. Display Records & Count Nodes\n");
printf("3. Insert at Beginning\n");
printf("4. Insert at End\n");
printf("5. Delete at Beginning\n");
printf("6. Delete at End\n");
printf("7. Search by USN\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch(choice)
{
case 1: createFront();
break;
case 2: display ();
break;
case 3: insertBegin();
break;
case 4: insertEnd();
break;
case 5: deleteBegin();
break;
case 6: deleteEnd();
break;
case 7: searchUSN();
break;
case 8: exit(0);
default: printf("Invalid choice.\n");
}
}
return 0;
```

}

Output:-

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End
7. Search by USN
8. EXIT

Enter your choice: 1

Enter number of students: 2

Enter USN: 4AD23CS001

Enter name: AA

Enter Semester: 3

Enter USN: 4AD23CS002

Enter name: BB

Enter Semester: 3

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End

7. Search by USN

8. EXIT

Enter your choice: 2

Student records:

USN: 4AD23CS002

Name: BB

Sem: 3

USN: 4AD23CS001

Name: AA

Sem: 3

Total number of nodes = 2

MENU

1. Create List (Front Insertion)

2. Display Records & Count Nodes

3. Insert at Beginning

4. Insert at End

5. Delete at Beginning

6. Delete at End

7. Search by USN

8. EXIT

Enter your choice: 3

Enter USN: 4AD23CS001

Enter Name: AA

Enter Semester: 3

Node inserted Beginning

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End
7. Search by USN
8. EXIT

Enter your choice: 4

Enter USN: 4AD23CS002

Enter Name: BB

Enter Semester: 3

Node inserted at End.

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End
7. Search by USN
8. EXIT

Enter your choice: 5

Deleted Student; 4AD23CS001

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End
7. Search by USN
8. EXIT

Enter your choice: 6

Deleted Student: 4AD23CS002

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End
7. Search by USN
8. EXIT

Enter your choice: 7

Enter USN to Search: 4AD23CS001

Record Found

USN: 4AD23CS001

Name: AA

Sem: 3

MENU

1. Create List (Front Insertion)
2. Display Records & Count Nodes
3. Insert at Beginning
4. Insert at End
5. Delete at Beginning
6. Delete at End
7. Search by USN
8. EXIT

Enter your choice: 8

Exit...

5. Develop a Program in C for the following:

- a. Create a Binary Tree by inserting integer elements entered by the user in level order.
- b. Implement the following functions:
 - i. createTree() – Create a binary tree using dynamic memory allocation.
 - ii. preorder() – Display the elements using Preorder Traversal.
 - iii. inorder() – Display the elements using Inorder Traversal.
 - iv. postorder() – Display the elements using Postorder Traversal.
- c. Provide a menu-driven interface with the following options:
 - i. Create Binary Tree
 - ii. Display Preorder Traversal
 - iii. Display Inorder Traversal
 - iv. Display Postorder Traversal
 - v. Exit

Program objective:

- Understand the concept of Binary Search Tree(BST).
- Understand the different traversal method on BST.
- Get to know the methodology of searching a key element in BST.
- Understand the methodology of deleting an element from BST.

Algorithm:

Preorder Traversal

Step1:Display root information

Step2: Traverse left sub tree in preorder

Step3:Traverse right sub tree in preorder

Inorder Traversal

Step1: Traversal the left sub tree inorder

Step 2: Display root information

Step3: Traversal right sub tree inorder

Postorder Traversal

Step 1: traverse the left sub tree in post order

Step2: traverse the right sub tree in post order

Step 3: Display root information

THEORY:-

A binary search tree (BST) is a tree in which all nodes follow the below mentioned properties

- The left sub-tree of a node has key less than or equal to its parent node's key.
- The right sub-tree of a node has key greater than or equal to its parent node's key.

Thus, a binary search tree (BST) divides all its sub-trees into two segments; left sub-tree and right sub-tree and can be defined as

$\text{left_subtree}(\text{keys}) \leq \text{node}(\text{key}) \leq \text{right_subtree}(\text{keys})$

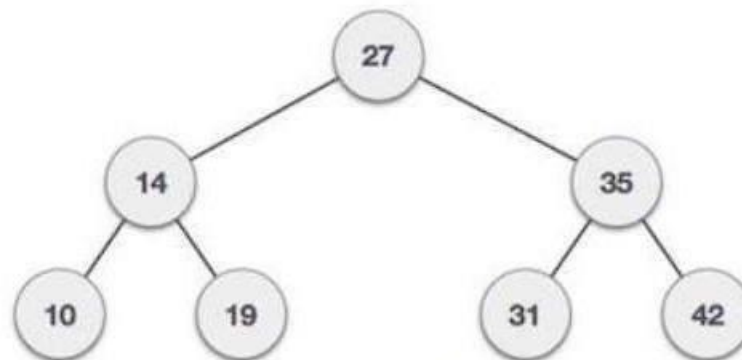


Fig: An example of BST

Fig9-Example of BST

Following are basic primitive operations of a tree which are following.

- **Search**—search an element in a tree.
- **Insert**—insert an element in a tree.
- **Preorder Traversal**—traversal a tree in a preorder manner.
- **Inorder Traversal**—traversal a tree in an inorder manner.
- **Postorder Traversal**—traversal tree in a post order manner.

Program 5.

```
#include <stdio.h>
#include <stdlib.h>
/* Structure of a Binary 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;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
/* Function to create a binary tree in level order */
struct Node* createTree() {
    struct Node *root = NULL;
    struct Node **queue;
    struct Node *current;
    int n, i, data;
    int front = 0, rear = 0;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Number of elements must be positive.\n");
        return NULL;
    }
    /* Dynamically allocate memory for queue */
```

```

queue = (struct Node**)malloc(n * sizeof(struct Node*));
if (queue == NULL) {
printf("Memory allocation failed!\n");
exit(1);
}
/* Create root node */
printf("Enter element 1: ");
scanf("%d", &data);
root = createNode(data);
queue[rear++] = root;
/* Insert remaining nodes in level order */
for (i = 2; i <= n; i++) {
printf("Enter element %d: ", i);
scanf("%d", &data);
current = createNode(data);
/* Get parent node from queue */
if (queue[front]->left == NULL) {
queue[front]->left = current;
} else {
queue[front]->right = current;
front++;
}
queue[rear++] = current;
}
free(queue);
printf("Binary tree created successfully!\n");
return root;
}
/* Preorder Traversal: Root -> Left -> Right */
void preorder(struct Node *root) {
if (root != NULL) {
printf("%d ", root->data);
preorder(root->left);
preorder(root->right);
}
}
/* Inorder Traversal: Left -> Root -> Right */

```

```

void inorder(struct Node *root) {
if (root != NULL) {
inorder(root->left);
printf("%d ", root->data);
inorder(root->right);
}
}

/* Postorder Traversal: Left -> Right -> Root */
void postorder(struct Node *root) {
if (root != NULL) {
postorder(root->left);
postorder(root->right);
printf("%d ", root->data);
}
}

/* Function to free the entire tree */
void freeTree(struct Node *root) {
if (root != NULL) {
freeTree(root->left);
freeTree(root->right);
free(root);
}
}

/* Main function */
int main() {
struct Node *root = NULL;
int choice;
while (1) {
printf("\n===== BINARY TREE MENU =====\n");
printf("1. Create Binary Tree\n");
printf("2. Display Preorder Traversal\n");
printf("3. Display Inorder Traversal\n");
printf("4. Display Postorder Traversal\n");
printf("5. Exit\n");
printf("=====\n");
printf("Enter your choice: ");
scanf("%d", &choice);

```

```
switch (choice) {
case 1:
/* Free previously created tree, if any */
if (root != NULL) {
freeTree(root);
root = NULL;
}
root = createTree();
break;
case 2:
if (root == NULL) {
printf("Please create the binary tree first.\n");
} else {
printf("Preorder Traversal: ");
preorder(root);
printf("\n");
}
break;
case 3:
if (root == NULL) {
printf("Please create the binary tree first.\n");
} else {
printf("Inorder Traversal: ");
inorder(root);
printf("\n");
}
break;
case 4:
if (root == NULL) {
printf("Please create the binary tree first.\n");
} else {
printf("Postorder Traversal: ");
postorder(root);
printf("\n");
}
break;
case 5:
```

```
freeTree(root);
printf("Program terminated.\n");
return 0;
default:
printf("Invalid choice! Please try again.\n");
}
}
return 0;
}
```

Output: -

BINARY TREE MENU

1. Create Binary Tree
2. Display Preorder Traversal
3. Display Inorder Traversal
4. Display Postorder Traversal
5. Exit

Enter your choice: 1

Enter the number of elements: 2

Enter element 1: 5

Enter element 2: 9

Binary tree created successfully!

BINARY TREE MENU

1. Create Binary Tree
2. Display Preorder Traversal
3. Display Inorder Traversal
4. Display Postorder Traversal
5. Exit

Enter your choice: 2

Preorder Traversal: 5 9

BINARY TREE MENU

1. Create Binary Tree
2. Display Preorder Traversal
3. Display Inorder Traversal
4. Display Postorder Traversal
5. Exit

Enter your choice: 3

Inorder Traversal: 9 5

BINARY TREE MENU

1. Create Binary Tree
2. Display Preorder Traversal
3. Display Inorder Traversal
4. Display Postorder Traversal
5. Exit

Enter your choice: 4

Postorder Traversal: 9 5

BINARY TREE MENU

1. Create Binary Tree
2. Display Preorder Traversal
3. Display Inorder Traversal
4. Display Postorder Traversal
5. Exit

Enter your choice: 5

Program terminated...

6. Develop a Program in C for the following operations on Graph(G) of Cities
- Create an Adjacency Matrix representing the graph of N cities
 - Print all the nodes reachable from a given starting node in a digraph using DFS/BFS method.

Program objective:

- Understand the concept of trees and adjacency matrix.
- Understand the concept of connected graph.
- Understand the Breath First Search(BFS) and Depth First Search (DFS)

Algorithm:

Step1: Initialize front, rear, visit and number of nodes

Step 2: Read adjacency matrix for graph

Step3: select source vertex from graph i. e v

Step4: source node is added into queue and cover all the nodes (adjacent) to v.

Once it is covered adjacent/traversed mark as visited.

Step5: Read next vertex from queue and cover all the nodes. If it is not visited, visit the nodes.

Step6: Repeat the process 3-5 until all nodes are covered in queue

THEORY:

BFS first visits all the vertices that are adjacent to a starting vertex. Every time it adds the adjacent vertex to a queue array q. On each successive iteration of the algorithm, the next vertex on the queue is examined to see if there are any unvisited vertices adjacent to it which can be added to the queue. Whenever a new vertex is taken from the queue, it is marked as a visited node in the visited array.

Applications of BFS :

- To check connectivity of a graph (number of times queue becomes empty tells the number of components in the graph)
- To check if a graph is a cyclic. (no cross edges indicates no cycle)

- To find minimum edge path in a graph

Depth first search is a graph algorithm required for processing vertices or edges of a graph in a systematic fashion. Depth first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to one it is currently in.

The algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to starting vertex, with the latter being dead end. By then, all vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at anyone of them.

Here we use a **STACK** to trace the depth first search. We push a vertex onto the stack when the vertex is reached for the first time, and we pop a vertex off the **stack** when it becomes a dead end.

Applications of DFS:

- The two orderings are advantage ous for various applications like to pological sorting, etc.
- To check connectivity of a graph (number of times stack becomes empty tells the number of components in the graph)
- To check if a graph is a cyclic. (no back edges indicates no cycle)
- To find articulation point in a graph

Program 6

```
#include <stdio.h>
#define MAX 20
int graph[MAX][MAX];
int visited[MAX];
int n;
/* DFS function */
void DFS(int v)
{
    int i;
    visited[v] = 1;
    printf("%d ", v);
    for (i = 0; i < n; i++)
    {
        if (graph[v][i] == 1 && visited[i] == 0)
        {
            DFS(i);
        }
    }
}
/* BFS function */
void BFS(int start)
{
    int queue[MAX];
    int front = 0, rear = 0;
    int i, v;
    for (i = 0; i < n; i++)
        visited[i] = 0;
    visited[start] = 1;
    queue[rear++] = start;
    while (front < rear)
    {
        v = queue[front++];
        printf("%d ", v);
        for (i = 0; i < n; i++)
        {
```

```
if (graph[v][i] == 1 && visited[i] == 0)
{
visited[i] = 1;
queue[rear++] = i;
}
}
}
}
int main()
{
int i, j, start, choice;
printf("Enter the number of cities: ");
scanf("%d", &n);
printf("\nEnter the adjacency matrix:\n");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
{
scanf("%d", &graph[i][j]);
}
}
printf("\nAdjacency Matrix:\n");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
{
printf("%d ", graph[i][j]);
}
printf("\n");
}
printf("\nEnter the starting city (0 to %d): ", n - 1);
scanf("%d", &start);
printf("\n1. DFS\n");
printf("2. BFS\n");
printf("Enter your choice: ");
scanf("%d", &choice);
for (i = 0; i < n; i++)
```

```
visited[i] = 0;
printf("\nCities reachable from city %d: ", start);
if (choice == 1)
{
DFS(start);
}
else if (choice == 2)
{
BFS(start);
}
else
{
printf("Invalid choice!");
}
printf("\n");
return 0;
}
```

Output:-

Enter the number of cities: 2

Enter the adjacency matrix:

1 5

2 7

adjacency matrix:

1 5

2 7

Enter the starting city (0 to 1): 5

1. DFS

2. BFS

Enter your choice: 1

Cities reachable from city 5: 5

Program outcomes:

- a. Create graph using adjacency matrix.
- b. Implement Breadth First Search(BFS) and Depth First Search (DFS)
- c. Familiarized with connected graph.
- d. Identify the applications of graphs.

Program outcomes:

- a. Create graph using adjacency matrix.
 - b. Implement Breadth First Search (BFS) and Depth First Search (DFS).
 - c. Familiarized with connected graph.
 - d. Identify the applications of graphs.
-

7. Develop a C program to:

- a. Read two sparse matrices of the same order.
- b. Convert both into 3-tuple representation.
- c. Perform matrix addition.
- d. Display the resultant matrix in both normal matrix form and 3-tuple form.

Theory:

Sparse Matrix Addition Using 3-Tuple Representation

A **sparse matrix** is a matrix in which most of the elements are **zero**. Storing all the zero elements requires unnecessary memory. Therefore, a sparse matrix can be stored efficiently using a **3-tuple representation**.

1. 3-Tuple Representation

In 3-tuple representation, only the **non-zero elements** of the matrix are stored.

The first row contains:

(Number of rows, Number of columns, Number of non-zero elements)

The remaining rows contain:

(Row number, Column number, Value)

For example, consider:

1 0 0

0 2 0

0 0 3

Its 3-tuple representation is:

3 3 3

0 0 1

1 1 2

2 2 3

Here, 3 3 3 means the original matrix has **3 rows, 3 columns, and 3 non-zero elements**.

2. Conversion to 3-Tuple

The program reads the matrix and checks every element.

- If the element is 0, it is ignored.
- If the element is non-zero, its **row index, column index, and value** are stored in the tuple.
- The count of non-zero elements is stored in the first row.

3. Addition of Sparse Matrices

The two matrices must have the **same order** (same number of rows and columns).

The program compares the positions of non-zero elements in the two tuples:

- If an element exists only in the first matrix, it is copied to the result.
- If an element exists only in the second matrix, it is copied to the result.
- If both matrices have an element at the same position, their values are added.
- If the sum is zero, that element is not stored in the result because the result is a sparse matrix.

4. Conversion Back to Normal Matrix

After addition, the resultant 3-tuple is converted back into normal matrix form.

First, all positions are initialized to 0. Then the non-zero values from the tuple are placed at their corresponding row and column positions.

5. Functions Used

The program uses the following functions:

- Convert To Tuple() – Converts a normal matrix into 3-tuple form.
- Display Tuple() – Displays the 3-tuple representation.
- Add Sparse() – Adds two sparse matrices using their tuple representations.
- Tuple To Matrix() – Converts the resultant tuple back to normal matrix form.
- Display Matrix() – Displays the resultant matrix.

Advantages of 3-Tuple Representation

- Saves **memory** by storing only non-zero elements.
- Makes operations on sparse matrices more efficient.
- Avoids storing large numbers of unnecessary zero values.
- Useful when matrices contain a very large number of zero elements.

Program 7.

```
#include <stdio.h>
#define MAX 100
// Function to convert a matrix into 3-tuple representation
void convertToTriplet(int mat[MAX][MAX], int rows, int cols, int triplet[MAX][3])
{
    int i, j, k = 1;
    triplet[0][0] = rows;
    triplet[0][1] = cols;
    triplet[0][2] = 0;
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < cols; j++)
        {
            if(mat[i][j] != 0)
            {
                triplet[k][0] = i;
                triplet[k][1] = j;
                triplet[k][2] = mat[i][j];
                k++;
            }
            triplet[0][2]++;
        }
    }
}

// Function to display triplet representation
void displayTriplet(int triplet[MAX][3])
{
    int i, n = triplet[0][2];
    printf("Row\tCol\tValue\n");
    for(i = 0; i <= n; i++)
    {
        printf("%d\t%d\t%d\n", triplet[i][0], triplet[i][1], triplet[i][2]);
    }
}

int main()
{
```

```
int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
int T1[MAX][3], T2[MAX][3], T3[MAX][3];
int rows, cols;
int i, j;
printf("Enter number of rows and columns: ");
scanf("%d%d", &rows, &cols);
printf("\nEnter elements of Matrix A:\n");
for(i = 0; i < rows; i++)
{
    for(j = 0; j < cols; j++)
    {
        scanf("%d", &A[i][j]);
    }
}
printf("\nEnter elements of Matrix B:\n");
for(i = 0; i < rows; i++)
{
    for(j = 0; j < cols; j++)
    {
        scanf("%d", &B[i][j]);
    }
}
// Matrix Addition
for(i = 0; i < rows; i++)
{
    for(j = 0; j < cols; j++)
    {
        C[i][j] = A[i][j] + B[i][j];
    }
}
// Convert to triplet
convertToTriplet(A, rows, cols, T1);
convertToTriplet(B, rows, cols, T2);
convertToTriplet(C, rows, cols, T3);
// Display Matrix A Triplet
printf("\n3-Tuple Representation of Matrix A:\n");
displayTriplet(T1);
```

```

// Display Matrix B Triplet
printf("\n3-Tuple Representation of Matrix B:\n");
displayTriplet(T2);
// Display Result Matrix
printf("\nResultant Matrix (A + B):\n");
for(i = 0; i < rows; i++)
{
for(j = 0; j < cols; j++)
{
printf("%4d", C[i][j]);
}
printf("\n");
}
// Display Result Triplet
printf("\n3-Tuple Representation of Result Matrix:\n");
displayTriplet(T3);
return 0;
}

```

Output:-

Enter number of rows and columns: 2

1 3

Enter elements of matrix A:

1 4

Enter elements of matrix B:

1 5

3-Tuple Representation of Matrix A:

Row	Col	Value
2	1	2
0	0	3
1	0	1

3-Tuple Representation of Matrix B:

Row	Col	Value
2	1	2
0	0	4
1	0	1

Resultant Matrix (A+B):

7

2

3-Tuple Representation of Result Matrix:

Row	Col	Value
2	1	2
0	0	7
1	0	2

Conclusion:-

Thus, the program demonstrates how two sparse matrices of the same order can be **represented using 3-tuples, added efficiently, and displayed in both normal and 3-tuple forms.**

8. Develop a C-based Expression Conversion tool that uses stack data structure to convert arithmetic expressions from infix notation to postfix notation. The tool should support alphanumeric operands, parentheses, and the operators (+, -, *, /, %, ^). Analyze the correctness of the conversion by testing various valid expressions.

Program objective:

1. Understand different polish notation.
2. Understand the methodology of evaluating suffix expression.
3. Get the knowledge of operator precedence and associativity.

Algorithm

Step1: Read the suffix/post fix expression

Step2: Scan the postfix expression from left to right character by character

Step 3: if scanned symbol is operand push data into stack.

If scanned symbol is operator pop two elements from stack Evaluate result and result is pushed onto stack

Step4: Repeat step 2-3 until all symbols are scanned completely

Program 8.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
/* Stack structure */
typedef struct {
    char items[MAX];
    int top;
} Stack;
/* Initialize stack */
void initStack(Stack *s) {
    s->top = -1;
}
/* Check if stack is empty */
int isEmpty(Stack *s) {
    return s->top == -1;
}
/* Check if stack is full */
int isFull(Stack *s) {
    return s->top == MAX - 1;
}
/* Push an element onto the stack */
void push(Stack *s, char ch) {
    if (isFull(s)) {
        printf("Stack Overflow\n");
        exit(EXIT_FAILURE);
    }
    s->items[++s->top] = ch;
}
/* Pop an element from the stack */
char pop(Stack *s) {
    if (isEmpty(s)) {
        return '\0';
    }
    return s->items[s->top--];
}
```

```

}
/* Return the top element */
char peek(Stack *s) {
if (isEmpty(s)) {
return '\0';
}
return s->items[s->top];
}
/* Return precedence of an operator */
int precedence(char op) {
switch (op) {
case '^':
return 3;
case '*':
case '/':
case '%':
return 2;
case '+':
case '-':
return 1;
default:
return 0;
}
}
/* Check whether a character is an operator */
int isOperator(char ch) {
return ch == '+' || ch == '-' || ch == '*' ||
ch == '/' || ch == '%' || ch == '^';
}
/*
* Convert infix expression to postfix expression.
* Returns 1 for success and 0 for invalid expression.
*/
int infixToPostfix(const char *infix, char *postfix) {
Stack s;
int i = 0, j = 0;
int expectOperand = 1;
initStack(&s);
while (infix[i] != '\0') {
char ch = infix[i];

```

```

/* Ignore whitespace */
if (isspace((unsigned char)ch)) {
    i++;
    continue;
}
/* Operand */
if (isalnum((unsigned char)ch)) {
    postfix[j++] = ch;
    expectOperand = 0;
}
/* Opening parenthesis */
else if (ch == '(') {
    if (!expectOperand) {
        return 0;
    }
    push(&s, ch);
    expectOperand = 1;
}
/* Closing parenthesis */
else if (ch == ')') {
    if (expectOperand) {
        return 0;
    }
    while (!isEmpty(&s) && peek(&s) != '(') {
        postfix[j++] = pop(&s);
    }
    /* No matching '(' */
    if (isEmpty(&s)) {
        return 0;
    }
    pop(&s); /* Remove '(' */
    expectOperand = 0;
}
/* Operator */
else if (isOperator(ch)) {
    if (expectOperand) {
        return 0;
    }
}
/*
 * ^ is right-associative.

```

```

* Other operators are left-associative.
*/
while (!isEmpty(&s) && peek(&s) != '(' &&
precedence(peek(&s)) > precedence(ch) ||
(precedence(peek(&s)) == precedence(ch) && ch != '^')) {
postfix[j++] = pop(&s);
}
push(&s, ch);
expectOperand = 1;
}
/* Invalid character */
else {
return 0;
}
i++;
}
/* Expression cannot end with an operator */
if (expectOperand) {
return 0;
}
/* Pop remaining operators */
while (!isEmpty(&s)) {
if (peek(&s) == '(') {
return 0; /* Unmatched '(' */
}
postfix[j++] = pop(&s);
}
postfix[j] = '\0';
return 1;
}

int main(void) {
char infix[MAX];
char postfix[MAX];
printf("=====\n");
printf("    INFIX TO POSTFIX CONVERTER\n");
printf("=====\n");
printf("Supported operators: + - * / %% ^\n");
printf("Enter an infix expression: ");
if (fgets(infix, sizeof(infix), stdin) == NULL) {
return 1;
}
if (infixToPostfix(infix, postfix)) {
printf("Postfix expression: %s\n", postfix);
} else {
printf("Invalid expression.\n");
}
}

```

```
}  
return 0;  
}
```

Output:-

INFIX TO POSTFIX CONVERTER

Enter an infix expression: 1 5 7

Supported operators: + - * / % ^

Infix Expression: 1 5 7

Postfix Expression: 157

9. Develop a menu-driven Program in C for the following operations on Circular Queue (integer/floating point data):
- Insert an element into the Circular Queue.
 - Delete an element from the Circular Queue.
 - Display the contents of the Circular Queue.
 - Demonstrate overflow and underflow conditions.
 - Exit

Program objective:

- 1 Understand the working of circular queue
- 2 Know the advantages of circular queue over liner queue.
- 3 Understand the insertion and deletion operation on circular queue.
- 4 Understand overflow and underflow conditions in circular queue.

ALGORITHM:

Step1: Initialize front and rear pointer and also count

front->0,count<-0,rear<--1

Step2: Insert an element into queue before check overflow condition Count=max

Insert an element rear<-(rear+1)%max

q[rear]<-item and count=count+1

Step3: Delete an element from queue. Check underflow condition

Count=0 underflow condition. Count<-count-1

Item<-q[front]Deleted element

Step4: Display contents of queue. Number of elements represents count. Check empty queue condition before displaying an element

THEORY

Circular queue is a linear data structure. It follows FIFO principle. In **circular queue** the last node is connected back to the first node to make a **circle**.

It is also called FIFO structure. Elements are added at the rear end and the elements are deleted at front end of the **queue**. The queue is considered as a circular queue when the positions 0 and MAX-1 are adjacent.

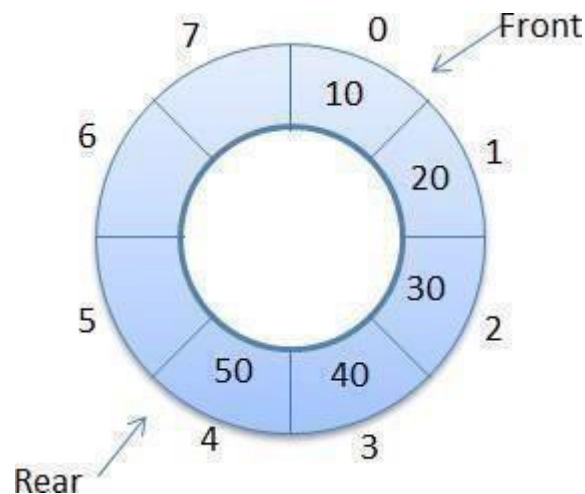


Fig6-circularqueue

The **limitation** of **simple queue** is that even if there is a free memory space available in the simple queue we cannot use that free memory space to insert element. **Circular Queue is designed to overcome the limitation of Simple Queue.**

Program 9:-

```
#include <stdio.h>
#define MAX 5
float queue[MAX];
int front = -1, rear = -1;
/* Insert an element into the circular queue */
void insert(float value)
{
/* Queue is full */
if ((rear + 1) % MAX == front)
{
printf("\nOverflow! Circular Queue is full.\n");
return;
}
/* First element */
if (front == -1)
{
front = 0;
rear = 0;
}
else
{
rear = (rear + 1) % MAX;
}
queue[rear] = value;
printf("\n%.2f inserted into the queue.\n", value);
}
/* Delete an element from the circular queue */
void deleteElement()
{
float value;
/* Queue is empty */
if (front == -1)
{
printf("\nUnderflow! Circular Queue is empty.\n");
return;
}
```

```

value = queue[front];
/* Only one element was present */
if (front == rear)
{
front = -1;
rear = -1;
}
else
{
front = (front + 1) % MAX;
}
printf("\n%.2f deleted from the queue.\n", value);
}
/* Display the circular queue */
void display()
{
int i;
if (front == -1)
{
printf("\nCircular Queue is empty.\n");
return;
}
printf("\nContents of Circular Queue:\n");
i = front;
while (1)
{
printf("%.2f ", queue[i]);
if (i == rear)
break;
i = (i + 1) % MAX;
}
printf("\n");
}
/* Main menu */
int main()
{
int choice;

```

```

float value;
while (1)
{
printf("\n===== CIRCULAR QUEUE =====\n");
printf("1. Insert an element\n");
printf("2. Delete an element\n");
printf("3. Display the queue\n");
printf("4. Demonstrate Overflow\n");
printf("5. Demonstrate Underflow\n");
printf("6. Exit\n");
printf("=====\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice)
{
case 1:
printf("Enter the element: ");
scanf("%f", &value);
insert(value);
break;
case 2:
deleteElement();
break;
case 3:
display();
break;
case 4:
printf("\nDemonstrating Overflow:\n");
/*
* Try inserting MAX elements.
* If the queue is already partially filled,
* insert until it becomes full.
*/
while ((rear + 1) % MAX != front)
{
printf("Enter element: ");
scanf("%f", &value);

```

```
insert(value);
}
printf("\nQueue is now full.\n");
printf("Trying one more insertion...\n");
printf("Enter element: ");
scanf("%f", &value);
insert(value);
break;
case 5:
printf("\nDemonstrating Underflow:\n");
if (front != -1)
{
printf("Deleting all elements...\n");
while (front != -1)
deleteElement();
}
printf("Queue is now empty.\n");
printf("Trying one more deletion...\n");
deleteElement();
break;
case 6:
printf("\nExiting program...\n");
return 0;
default:
printf("\nInvalid choice! Please try again.\n");
}
}
return 0;
}
```

Output:-

CIRCULAR QUEUE

1. Insert an element
2. Delete an element
3. Display the queue
4. Demonstrate Overflow
5. Demonstrate Underflow
6. Exit

Enter your choice: 1

Enter your element: 2

2.00 inserted into the queue.

CIRCULAR QUEUE

1. Insert an element
2. Delete an element
3. Display the queue
4. Demonstrate Overflow
5. Demonstrate Underflow
6. Exit

Enter your choice: 2

2.00 deleted from the queue.

CIRCULAR QUEUE

1. Insert an element
2. Delete an element
3. Display the queue
4. Demonstrate Overflow
5. Demonstrate Underflow
6. Exit

Enter your choice: 3

Circular Queue is empty.

CIRCULAR QUEUE

1. Insert an element
2. Delete an element
3. Display the queue
4. Demonstrate Overflow
5. Demonstrate Underflow
6. Exit

Enter your choice: 4

Demonstrating Overflow:

Enter element: 1

1.00 inserted into the queue.

Enter element: 2

2.00 inserted into the queue.

Enter element: 1

1.0 inserted into the queue.

Enter element: 2

2.00 inserted into the queue.

Enter element: 3

3.00 inserted into the queue.

Queue is now full.

Trying one more insertion...

Enter element: 1

Overflow! Circular Queue is full.\

CIRCULAR QUEUE

1. Insert an element
2. Delete an element
3. Display the queue
4. Demonstrate Overflow
5. Demonstrate Underflow
6. Exit

Enter your choice: 5

Demonstrating Underflow:

Deleting all elements...

1.00deleted from the queue.

2.00eleted from the queue.

1.00 deleted from the queue.

2.00 deleted from queue.

3.00 deleted from the queue.

Underflow! Circular Queue is empty.

CIRCULAR QUEUE

1. Insert an element
2. Delete an element
3. Display the queue
4. Demonstrate Overflow
5. Demonstrate Underflow
6. Exit

Enter your choice: 6

Exiting Program...

10. Develop a menu driven Program in C for the following operations on Doubly Linked List (DLL) of Employee

Data with the fields: SSN, Name, Dept, Designation, Sal, PhNo etc.

- a. Create a DLL of N Employees Data by using end insertion.
- b. Display the status of DLL and count the number of nodes in it
- c. Perform Insertion and Deletion at End of DLL
- d. Perform Insertion and Deletion at Front of DLL
- e. Demonstrate how this DLL can be used as Double Ended Queue.
- f. Exit.

Program objective:

1. Understand the Doubly Linked List (DLL) data structures.
2. Understand the methodology to insert and delete the element at the front of DLL.
3. Understand the methodology to insert and delete the element at the end of DLL.
4. Get the knowledge of how DLL can be used as double ended queue.

Algorithm

Insertion at front end of list.

Step1: Allocate memory for emp node and assign values to node

Step2: if list is empty, temp is attached to list directly

head=null return temp

if list is not empty

temp->rlink=head

head->llink=temp

return head

Insertion at rear end of list.

Step1: Read node information and allocate memory for temp node

Step2: traverse the list

Reach node up to end of list then attach node cur to temp cur->rlink=temp;

temp->llink=cur

Step3: return starting address of list

Return head;

Delete from front end of list.

Step1: check if list has only one node

```
head=NULL;
```

```
returnnull;
```

```
iflistisempty head-
```

```
>rlink=NULL
```

```
returnNULL;
```

```
iflisthasonlyonenode
```

Step2: other wise first node address is shifted to next node

```
cur=head
```

```
head=head->rlink
```

```
free(cur)
```

Step3: return starting address of list

```
return head
```

Delete node from rear end

Step1: two pointer s r equires one is cur and prev

Cur is one which points, node to be deleted.

Step2: Traverse the cur node upto end of list before updating current pointer save the

Address to prev pointer.

```
While(cur->rlink!=null)
```

```
{
```

```
prev=cur;
```

```
cur=cur->rlink;
```

```
}
```

```
prev->rlink=null; cur-
```

```
>llink=null;
```

```
free(cur);
```

Step3: return starting address of the list

THEORY

- In computer science, a doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes.
- Each node contains two fields, called links, that are references to the previous and

to the next node in the sequence of nodes. The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

- A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.
- The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal and no need to traverse the list to find the previous node, so that its link can be modified.

Program 10:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Employee
{
char SSN[20];
char Name[50];
char Dept[30];
char Designation[30];
float Sal;
char PhNo[20];
struct Employee *prev;
struct Employee *next;
};
typedef struct Employee NODE;
NODE *head = NULL;
NODE *tail = NULL;
/* Function to create a new employee node */
NODE *createNode()
{
NODE *newNode = (NODE *)malloc(sizeof(NODE));
if (newNode == NULL)
{
printf("\nMemory allocation failed!\n");
exit(1);
}
printf("\nEnter Employee Details\n");
printf("SSN      : ");
scanf(" %19[^\n]", newNode->SSN);
printf("Name      : ");
scanf(" %49[^\n]", newNode->Name);
printf("Department : ");
scanf(" %29[^\n]", newNode->Dept);
printf("Designation : ");
scanf(" %29[^\n]", newNode->Designation);
printf("Salary    : ");
scanf("%f", &newNode->Sal);
printf("Phone Number : ");
```

```
scanf(" %19[^\n]", newNode->PhNo);
newNode->prev = NULL;
newNode->next = NULL;
return newNode;
}
/* Insert node at the end */
void insertEnd()
{
    NODE *newNode = createNode();
    if (head == NULL)
    {
        head = tail = newNode;
    }
    else
    {
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }
    printf("\nEmployee inserted at the end successfully.\n");
}
/* Insert node at the front */
void insertFront()
{
    NODE *newNode = createNode();
    if (head == NULL)
    {
        head = tail = newNode;
    }
    else
    {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("\nEmployee inserted at the front successfully.\n");
}
/* Delete node from the end */
void deleteEnd()
{

```

```

NODE *temp;
if (head == NULL)
{
printf("\nDLL is empty. Deletion not possible.\n");
return;
}
temp = tail;
if (head == tail)
{
head = tail = NULL;
}
else
{
tail = tail->prev;
tail->next = NULL;
}
printf("\nDeleted Employee: %s\n", temp->Name);
free(temp);
}
/* Delete node from the front */
void deleteFront()
{
NODE *temp;
if (head == NULL)
{
printf("\nDLL is empty. Deletion not possible.\n");
return;
}
temp = head;
if (head == tail)
{
head = tail = NULL;
}
else
{
head = head->next;
head->prev = NULL;
}
printf("\nDeleted Employee: %s\n", temp->Name);
free(temp);
}

```

```

}
/* Display DLL and count nodes */
void display()
{
    NODE *temp = head;
    int count = 0;
    if (head == NULL)
    {
        printf("\nDLL is empty.\n");
        return;
    }
    printf("\n===== EMPLOYEE DLL =====\n");
    while (temp != NULL)
    {
        printf("\nEmployee %d\n", count + 1);
        printf("-----\n");
        printf("SSN      : %s\n", temp->SSN);
        printf("Name       : %s\n", temp->Name);
        printf("Department  : %s\n", temp->Dept);
        printf("Designation : %s\n", temp->Designation);
        printf("Salary      : %.2f\n", temp->Sal);
        printf("Phone Number : %s\n", temp->PhNo);
        count++;
        temp = temp->next;
    }
    printf("\n===== \n");
    printf("Total number of employees = %d\n", count);
}
/* Create DLL with N employees using end insertion */
void createDLL()
{
    int n, i;
    printf("\nEnter number of employees: ");
    scanf("%d", &n);
    if (n <= 0)
    {
        printf("\nInvalid number of employees.\n");
        return;
    }
    for (i = 0; i < n; i++)

```

```

{
printf("\n===== Employee %d =====\n", i + 1);
insertEnd();
}
printf("\nDLL created successfully with %d employees.\n", n);
}
/* Demonstrate DLL as a double-ended queue */
void dequeMenu()
{
int choice;
while (1)
{
printf("\n===== DOUBLE ENDED QUEUE =====\n");
printf("1. Insert at Front\n");
printf("2. Insert at Rear\n");
printf("3. Delete from Front\n");
printf("4. Delete from Rear\n");
printf("5. Display Queue\n");
printf("6. Exit Deque Menu\n");
printf("=====\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice)
{
case 1:
insertFront();
break;
case 2:
insertEnd();
break;
case 3:
deleteFront();
break;
case 4:
deleteEnd();
break;
case 5:
display();
break;
case 6:

```

```

return;
default:
printf("\nInvalid choice!\n");
}
}
}
/* Main menu */
int main()
{
int choice;
while (1)
{
printf("\n\n===== DOUBLY LINKED LIST =====\n");
printf("1. Create DLL of N Employees\n");
printf("2. Display DLL and Count Nodes\n");
printf("3. Insert at End\n");
printf("4. Delete from End\n");
printf("5. Insert at Front\n");
printf("6. Delete from Front\n");
printf("7. Demonstrate DLL as Double Ended Queue\n");
printf("8. Exit\n");
printf("=====\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice)
{
case 1:
createDLL();
break;
case 2:
display();
break;
case 3:
insertEnd();
break;
case 4:
deleteEnd();
break;
case 5:
insertFront();

```

```
break;
case 6:
deleteFront();
break;
case 7:
dequeMenu();
break;
case 8:
printf("\nExiting program...\n");
exit(0);
default:
printf("\nInvalid choice! Please try again.\n");
}
}
return 0;
}
```

Output:-

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 1

Enter number of employees: 1

Employee 1

Enter Employee Details

SSN: 4AD24CS001

Name: AA

Department: CSE

Destination: Programmer

Salary: 10.000

Phone Number: 1234567890

Employee inserted at the end successfully.

DEL created successfully with 1 employees.

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 2

EMPLOYEE DELL

Employee 1

SSN: 4AD24CS001

Name: AA

Department: CSE

Destination: Programmer

Salary: 10.000

Phone Number: 1234567890

Total number of employees = 1

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 3

Enter Employee Details

SSN: 4AD24CS002

Name: BB

Department: CSE

Destination: Programmer

Salary: 10.000

Phone Number: 1234567890

Employee inserted at the end successfully.

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 4

Deleted Employee: BB

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 5

Enter Employee Details

SSN: 4AD24CS003

Name: CC

Department: CSE

Destination: Programmer

Salary: 10.000

Phone Number: 1234567890

Employee inserted at the front successfully.

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 6

Deleted Employee: CC

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 7

DOUBLE ENDED QUEUE

1. Insert at front
2. Insert at rear
3. Delete from front
4. Delete from rear
5. Display Queue
6. Exit Deque Menu

Enter your choice: 1

Enter Employee Details

SSN: 4AD24CS004

Name: DD
Department: CSE
Designation: Programmer
Salary: 10000
Phone number: 1234567890

Employee inserted at the front successfully.

DOUBLE ENDED QUEUE

1. Insert at front
2. Insert at rear
3. Delete from front
4. Delete from rear
5. Display Queue
6. Exit Deque Menu

Enter your choice: 2

Enter Employee Details

SSN: 4AD24CS005
Name: EE
Department: CSE
Designation: Programmer
Salary: 10000
Phone number: 1234567890

Employee inserted at the end successfully.

DOUBLE ENDED QUEUE

1. Insert at front
2. Insert at rear
3. Delete from front
4. Delete from rear
5. Display Queue
6. Exit Deque Menu

Enter your choice: 3

Deleted Employee: DD

DOUBLE ENDED QUEUE

1. Insert at front
2. Insert at rear
3. Delete from front
4. Delete from rear
5. Display Queue
6. Exit Deque Menu

Enter your choice: 4

Deleted Employee: EE

DOUBLE ENDED QUEUE

1. Insert at front
2. Insert at rear
3. Delete from front
4. Delete from rear
5. Display Queue
6. Exit Deque Menu

Enter your choice: 5

EMPLOYEE DELL

Employee 1

SSN: 4AD24CS001

Name: AA

Department: CSE

Destination: Programmer

Salary: 10.000

Phone Number: 1234567890

Total number of employees = 1

DOUBLE ENDED QUEUE

1. Insert at front
2. Insert at rear
3. Delete from front
4. Delete from rear
5. Display Queue
6. Exit Deque Menu

Enter your choice: 6

DOUBLY LINKED LIST

1. Create DELL of N Employees
2. Display DLL and Count nodes
3. Insert at End
4. Delete from End
5. Insert at Front
6. Delete from Front
7. Demonstrate DLL as Double Ended Queue
8. Exit.

Enter your choice: 8

Exiting Program...

Program outcome:

- Implement Doubly Linked List.
- Implement insertion at the front and end of DLL.
- Implement deletion at the front and end of DLL.
- Identify the applications of DLL.
- Familiarized how DLL can be used as double ended queue.

Viva Questions:

- What are doubly linked lists?
- What is the difference between singly and doubly linked lists?
- What are the advantages of double linked list over single linked list?

11. Develop a menu driven Program in C for the following operations on Binary Search Tree (BST) of Integers.

- a. Create a BST of N Integers
- b. Traverse the BST in Inorder, Preorder and Post Order
- c. Search the BST for a given element (KEY) and report the appropriate message
- d. Exit

Theory:-

Binary Search Tree (BST)

A **Binary Search Tree (BST)** is a binary tree in which each node contains an integer and follows a specific ordering rule:

- All elements in the **left subtree** are smaller than the root node.
- All elements in the **right subtree** are greater than the root node.
- The left and right subtrees are themselves Binary Search Trees.
- Duplicate elements are generally not inserted.

Structure of a BST Node

Each node contains three parts:

```
struct node {  
    int data;  
    struct node *left;  
    struct node *right;  
};
```

- data stores the integer value.
- left points to the left child.
- right points to the right child.

Operations Performed :-

1. Creating a BST

The program first reads **N integers** from the user. Each integer is inserted into the BST using the insert() function.

For every new element:

- If the tree is empty, it becomes the root.
- If the value is smaller than the current node, it is inserted into the left subtree.
- If the value is greater, it is inserted into the right subtree.
- If it is equal, it is treated as a duplicate and is not inserted.

2. Tree Traversals

The program performs three standard tree traversals:

Inorder:

Left → Root → Right

In a BST, inorder traversal displays elements in **ascending order**.

Preorder:

Root → Left → Right

It visits the root before its subtrees.

Postorder:

Left → Right → Root

It visits the root after its subtrees.

For example,

```
    50
   /  \
  30   70
 / \  / \
20 40 60 80
```

The traversals are:

Inorder: 20 30 40 50 60 70 80

Preorder: 50 30 20 40 70 60 80

Postorder: 20 40 30 60 80 70 50

3. Searching for a KEY

The search() function searches for a given element.

- If the KEY equals the current node's value, the element is found.
- If the KEY is smaller, the search continues in the left subtree.
- If the KEY is greater, the search continues in the right subtree.
- If the pointer becomes NULL, the KEY is not present in the BST.

This is more efficient than searching every node because the BST property allows half of the tree to be eliminated at each step in a balanced tree.

4. Exit The program uses a **menu-driven approach** with a switch statement. The user can repeatedly select an operation until option **4 (Exit)** is selected.

Time Complexity:-

Operation	Average Case	Worst Case
Insertion	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$

The worst case occurs when the BST becomes **skewed**, similar to a linked list.

Conclusion

Thus, the program implements a **menu-driven Binary Search Tree** using dynamic memory allocation and recursion. It allows the user to create a BST, perform inorder, preorder and postorder traversals, search for a specified KEY, and exit the program.

```

#include <stdio.h>
#include <stdlib.h>
// Structure for a BST 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 an element into 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 if (data > root->data)
root->right = insert(root->right, data);
else
printf("Duplicate element %d not inserted.\n", data);
return root;
}
// Inorder traversal: Left -> Root -> Right
void inorder(struct Node *root) {
if (root != NULL) {
inorder(root->left);
printf("%d ", root->data);
inorder(root->right);
}
}
// Preorder traversal: Root -> Left -> Right

```

```

void preorder(struct Node *root) {
if (root != NULL) {
printf("%d ", root->data);
preorder(root->left);
preorder(root->right);
}
}

// Postorder traversal: Left -> Right -> Root
void postorder(struct Node *root) {
if (root != NULL) {
postorder(root->left);
postorder(root->right);
printf("%d ", root->data);
}
}

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

// Main function
int main() {
struct Node *root = NULL;
struct Node *result;
int choice, n, i, value, key;
while (1) {
printf("\n\n===== BINARY SEARCH TREE MENU =====\n");
printf("1. Create a BST of N Integers\n");
printf("2. Inorder Traversal\n");
printf("3. Preorder Traversal\n");
printf("4. Postorder Traversal\n");
printf("5. Search for an Element\n");
printf("6. Exit\n");
printf("Enter your choice: ");

```

```
scanf("%d", &choice);
switch (choice) {
case 1:
// Reset the tree before creating a new BST
root = NULL;
printf("Enter the number of elements: ");
scanf("%d", &n);
printf("Enter %d integers:\n", n);
for (i = 0; i < n; i++) {
scanf("%d", &value);
root = insert(root, value);
}
printf("BST created successfully.\n");
break;
case 2:
if (root == NULL) {
printf("BST is empty. Please create the BST first.\n");
} else {
printf("Inorder Traversal: ");
inorder(root);
printf("\n");
}
break;
case 3:
if (root == NULL) {
printf("BST is empty. Please create the BST first.\n");
} else {
printf("Preorder Traversal: ");
preorder(root);
printf("\n");
}
break;
case 4:
if (root == NULL) {
printf("BST is empty. Please create the BST first.\n");
} else {
printf("Postorder Traversal: ");
```

```
postorder(root);
printf("\n");
}
break;
case 5:
if (root == NULL) {
printf("BST is empty. Please create the BST first.\n");
} else {
printf("Enter the element to search: ");
scanf("%d", &key);
result = search(root, key);
if (result != NULL)
printf("KEY %d is found in the BST.\n", key);
else
printf("KEY %d is not found in the BST.\n", key);
}
break;
case 6:
printf("Exiting the program...\n");
exit(0);
default:
printf("Invalid choice! Please enter a valid choice.\n");
}
}
return 0;
}
```

Output: -

BINARY SEARCH TREE MENU

1. Create a BST of N Integers
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Search for an Element
6. Exit

Enter your choice: 1

Enter the number of element: 2

Enter 2 integers:

1 3

BST created successfully.

BINARY SEARCH TREE MENU

1. Create a BST of N Integers
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Search for an Element
6. Exit

Enter your choice: 2

Inorder Traversal : 1 3

BINARY SEARCH TREE MENU

1. Create a BST of N Integers
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Search for an Element
6. Exit

Enter your choice: 3

Preorder Traversal: 1 3

BINARY SEARCH TREE MENU

1. Create a BST of N Integers
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Search for an Element
6. Exit

Enter your choice: 4

Postorder Traversal: 3 1

BINARY SEARCH TREE MENU

1. Create a BST of N Integers
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Search for an Element
6. Exit

Enter your choice: 5

Enter element to search: 1

KEY 1 is found in the BST.

BINARY SEARCH TREE MENU

1. Create a BST of N Integers
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Search for an Element
6. Exit

Enter your choice: 6

Exiting Program...

12. Develop a Program in C for an Employee Record Management System using Hashing with the following operations:

- a. Create a hash table to store employee records using Employee ID as the key.
- b. Insert employee records into the hash table using the Hash Function $H: K \rightarrow L$ as $H(K)=K \bmod m$
- c. Resolve collisions using Linear Probing.
- d. Search for an employee record using a given Employee ID and display the details.
- e. Display the contents of the hash table along with the corresponding memory locations.

Theory:

Employee Record Management System Using Hashing

An **Employee Record Management System** can be implemented efficiently using a **hash table**. A hash table stores employee records in an array, where the **Employee ID** is used as the key to determine the location of the record.

1. Hashing :-

Hashing is a technique used to store and retrieve data quickly. A **hash function** converts a key into an index of the hash table.

In this program, the Employee ID is the key and the hash function is:

$$H(K)=K \bmod m$$

where:

- K = Employee ID
- m = size of the hash table
- H(K) = index at which the employee record is stored

For example, if the table size is 10 and the Employee ID is 1234:

$$H(1234)=1234 \bmod 10=4$$

Therefore, the record is initially stored at index 4.

2. Hash Table :-

A **hash table** is an array used to store employee records. In the program, the hash table is declared as:

```
struct Employee hashTable[MAX];
```

Each employee record contains:

- Employee ID
- Employee name
- Salary
- Occupied status

The occupied field indicates whether a particular location contains an employee record.

3. **Collision :-**

A **collision** occurs when two different Employee IDs produce the same hash index.
For example:

Employee ID 1234 $\rightarrow 1234 \% 10 = 4$

Employee ID 5674 $\rightarrow 5674 \% 10 = 4$

Both records have the same hash index 4, resulting in a collision.

4. **Collision Resolution Using Linear Probing**

The program resolves collisions using **Linear Probing**.

When the calculated position is already occupied, the program checks the next consecutive location:

$4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow \dots$

The statement:

Index = (index + 1) % MAX;

moves to the next position. The modulo operation allows the search to wrap around to index 0 when the end of the table is reached.

For example:

1234 $\rightarrow$ index 4

5674 $\rightarrow$ index 4 (collision)

$\downarrow$

index 5 (empty)

Thus, employee 5674 is stored at index 5.

5. Insertion Operation

The **insertion operation** adds a new employee record to the hash table.

The steps are:

1. Read Employee ID, name, and salary.
2. Calculate the hash index using $ID \% MAX$.
3. Check whether the calculated position is empty.
4. If empty, insert the record.
5. If occupied, use linear probing to find the next available position.
6. If the Employee ID already exists, insertion is rejected.
7. If the entire table is full, insertion is not possible.

6. Searching an Employee :-

To search for an employee, the Employee ID is given as the key.

The program:

1. Calculates the hash index.
2. Checks the record at that index.
3. If the ID matches, the employee details are displayed.
4. If it does not match, linear probing checks subsequent positions.
5. The search stops when the employee is found or the appropriate search sequence has been exhausted.

This makes searching much faster than checking every employee sequentially in typical cases.

7. Displaying the Hash Table

The display operation shows every location of the hash table.

For each index, it displays:

- Hash table index
- Employee ID
- Employee name
- Salary
- Memory address

The memory address is obtained using:

`(void *)&hashTable[i]`

Program 12 :-

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
// Structure for employee record
struct Employee {
    int id;
    char name[50];
    float salary;
    int occupied;
};
// Hash function:  $H(K) = K \bmod m$ 
int hashFunction(int key) {
    return key % MAX;
}
// Initialize hash table
void initialize(struct Employee table[]) {
    for (int i = 0; i < MAX; i++) {
        table[i].occupied = 0;
    }
}
// Insert employee record using linear probing
void insert(struct Employee table[]) {
    struct Employee emp;
    int index, originalIndex;
    printf("\nEnter Employee ID: ");
    scanf("%d", &emp.id);
    printf("Enter Employee Name: ");
    scanf("%[^\n]", emp.name);
    printf("Enter Employee Salary: ");
    scanf("%f", &emp.salary);
    index = hashFunction(emp.id);
    originalIndex = index;
    // Linear probing for collision resolution
    while (table[index].occupied) {
        index = (index + 1) % MAX;
    }
    // Hash table is full
    if (index == originalIndex) {
        printf("\nHash table is full. Cannot insert record.\n");
    }
}
```

```

return;
}
}
table[index] = emp;
table[index].occupied = 1;
printf("\nEmployee record inserted successfully at index %d.\n", index);
}

// Search employee record
void search(struct Employee table[]) {
int id, index, originalIndex;
printf("\nEnter Employee ID to search: ");
scanf("%d", &id);
index = hashFunction(id);
originalIndex = index;
while (table[index].occupied) {
if (table[index].id == id) {
printf("\nEmployee Found!\n");
printf("Hash Table Index : %d\n", index);
printf("Memory Location : %p\n", (void *)&table[index]);
printf("Employee ID    : %d\n", table[index].id);
printf("Employee Name   : %s\n", table[index].name);
printf("Employee Salary  : %.2f\n", table[index].salary);
return;
}
index = (index + 1) % MAX;
if (index == originalIndex)
break;
}
printf("\nEmployee with ID %d not found.\n", id);
}

// Display hash table
void display(struct Employee table[]) {
printf("\n===== HASH TABLE =====\n");
printf("%-8s %-15s %-15s %-15s %-20s\n",
"Index", "Employee ID", "Name", "Salary", "Memory Location");
printf("-----\n");
for (int i = 0; i < MAX; i++) {
if (table[i].occupied) {
printf("%-8d %-15d %-15s %-15.2f %p\n",
i,

```

```

table[i].id,
table[i].name,
table[i].salary,
(void *)&table[i]);
} else {
printf("%-8d %-15s %-15s %-15s %p\n",
i, "EMPTY", "-", "-", (void *)&table[i]);
}
}
printf("=====\n");
}
int main() {
struct Employee hashTable[MAX];
int choice;
initialize(hashTable);
while (1) {
printf("\n\n==== EMPLOYEE RECORD MANAGEMENT SYSTEM =====\n");
printf("1. Insert Employee Record\n");
printf("2. Search Employee Record\n");
printf("3. Display Hash Table\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
insert(hashTable);
break;
case 2:
search(hashTable);
break;
case 3:
display(hashTable);
break;
case 4:
printf("\nExiting program...\n");
exit(0);
default:
printf("\nInvalid choice! Please try again.\n");
}
}
}

```

```
return 0;  
}
```

Output :-

EMPLOYEE RECORD MANAGEMENT SYSTEM

1. Insert Employee Record
2. Search Employee Record
3. Display Hash Table
4. Exit.

Enter your choice: 1

Enter Employee ID: 123

Enter Employee Name: SS

Enter Employee salary: 5000

Employee record inserted successfully at index 3.

EMPLOYEE RECORD MANAGEMENT SYSTEM

1. Insert Employee Record
2. Search Employee Record
3. Display Hash Table
4. Exit.

Enter your choice: 2

Enter Employee ID to Search: 123

Employee Found!

Hash Table Index: 3

Memory Location: 0x7ffea27f2350

Employee ID: 123

Employee Name: SS

Employee salary: 5000.00

EMPLOYEE RECORD MANAGEMENT SYSTEM

1. Insert Employee Record
2. Search Employee Record
3. Display Hash Table
4. Exit.

Enter your choice: 3

HASH TABLE

Index	Employee ID	Name	Salary	Memory Location
0	EMPTY	-	-	0x7ffea27f2290
1	EMPTY	-	-	0x7ffea27f22d0
2	EMPTY	-	-	0x7ffea27f2310
3	EMPTY	SS	5000.00	0x7ffea27f2350
4	EMPTY	-	-	0x7ffea27f2390
5	EMPTY	-	-	0x7ffea27f23d0
6	EMPTY	-	-	0x7ffea27f2410
7	EMPTY	-	-	0x7ffea27f2450
8	EMPTY	-	-	0x7ffea27f2490
9	EMPTY	-	-	0x7ffea27f24d0

EMPLOYEE RECORD MANAGEMENT SYSTEM

1. Insert Employee Record
2. Search Employee Record
3. Display Hash Table
4. Exit.

Enter your choice: 4

Exiting Program...

