BRAINWARE UNIVERSITY
School of Engineering
Department of Computer Science & Engineering—Cyber Security & Data Science
398, Ramkrishnapur Road, Barasat, North 24 Parganas, Kolkata - 700 125

Department of Computer Science & Engineering—Cyber Security & Data Science

Laboratory Manual

Bachelor of Technology in Computer Science & Engineering—Data Science
BTD39104, Data Structure and Algorithm Lab
&
Bachelor of Technology in Computer Science & Engineering— Cyber Security
BTY39106, Data Structure and Algorithm Lab

Compiled by
Mr. Dulal Adak
Assistant Professor
Dept. of CSE- CS & Ds
Brainware University

BRAINWARE UNIVERSITY
School of Engineering
Department of Computer Science & Engineering—Cyber Security & Data Science
398, Ramkrishnapur Road, Barasat, North 24 Parganas, Kolkata - 700 125

Content

Contents

Date	Compiled by	Description
January 2025	Mr. Dulal Adak	Updated with 15 points for each example
January 2023	Dept. of CSE	Initial release

Table: Revision History

Experiment No. 1: Basic Implementation of stack and queue Using Array

1. Aim / Purpose of the Experiment

To implement and understand the basic operations (insertion, deletion, and display) of **Stack** and **Queue** using **arrays**, demonstrating the principles of LIFO (Last In First Out) and FIFO (First In First Out) data structures respectively.

2. Learning Outcomes

After performing this experiment, students will be able to:

- Understand the working of **stack and queue** data structures.
- Implement **push, pop, enqueue, and dequeue** operations using arrays.
- Recognize overflow and underflow conditions in stack and queue.
- Write structured and modular C programs using arrays and control statements.

3. Prerequisites

- Basic knowledge of **C programming**.
- Understanding of **arrays**, **functions**, and **pointers**.
- Fundamental concepts of **data structures** (stack and queue).
- Use of control structures (if-else, loops).

4. Materials / Equipment / Apparatus / Software Required

- PC or laptop
- C compiler (Turbo C / GCC)
- Text editor or IDE (Code::Blocks, VS Code, Turbo C++)
- Lab notebook

5. Introduction and Theory

Stack

A stack is a linear data structure that follows the **LIFO (Last In First Out)** principle. The element inserted last is accessed first.

- **Operations:**
  - Push: Add an element to the top.
  - Pop: Remove the top element.
  - Peek: View the top element without removing it.
  - Display: View all elements from top to bottom.

Queue

A queue follows the **FIFO (First In First Out)** principle. The element inserted first is accessed first.

- **Operations:**
  - Enqueue: Insert element at rear end.
  - Dequeue: Remove element from front.
  - Display: View all elements from front to rear.

6. Operating Procedure

Stack Operations

1. Initialize top = -1.
2. **Push**: Check for overflow (top == MAX - 1). Increment top and add element.
3. **Pop**: Check for underflow (top == -1). Retrieve and decrement top.
4. **Display**: Loop from top to 0 and print elements.

Queue Operations

1. Initialize front = -1, rear = -1.
2. **Enqueue**: Check for overflow (rear == MAX - 1). Update rear, insert value.
3. **Dequeue**: Check for underflow (front == -1 || front > rear). Retrieve and increment front.
4. **Display**: Loop from front to rear and print values.

Stack Implementation

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

// Function to check if the stack is full
int isFull() {
   return top == MAX_SIZE - 1;
}

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

// Function to push an element onto the stack
void push(int data) {
   if (isFull()) {
      printf("Stack Overflow\n");
      return;
   }
   stack[++top] = data;
}

// Function to pop an element from the stack
int pop() {
   if (isEmpty()) {
      printf("Stack Underflow\n");
      return -1;
   }
   return stack[top--];
```

Queue Implementation

```
#include <stdio.h>
#include <stdlib>
#define MAX_SIZE 100

int queue[MAX_SIZE];
int front = -1;
int rear = -1;

// Function to check if the queue is full
int isFull() {
   return rear == MAX_SIZE - 1;
}

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

// Function to enqueue an element into the queue
void enqueue(int data) {
   if (isFull()) {
      printf("Queue Overflow\n");
      return;
   }
   if (front == -1) {
      front = 0;
   }
   queue[++rear] = data;
}

// Function to dequeue an element from the queue
int dequeue() {
   if (isEmpty()) {
```

```
}

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

int main() {
    push(10);
    push(20);
    push(30);
    display();
    printf("Popped element: %d\n", pop());
    display();
    return 0;
}
```

```
        printf("Queue Underflow\n");
        return -1;
    }
    return queue[front++];
}

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

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    printf("Dequeued element: %d\n", dequeue());
    display();
    return 0;
}
```

7. Precautions and/or Troubleshooting

- Check for **array bounds** to prevent overflow/underflow.
- Always initialize top, front, and rear correctly.
- Handle **empty structure** conditions gracefully.
- Ensure memory limits are not exceeded (especially in constrained environments).

8. Observations

Operation	Input Element	Stack State / Queue State	Notes
Push	10	[10] (Stack)	
Push	20	[10, 20]	
Pop	--	[10]	20 is removed
Enqueue	5	[5] (Queue)	
Enqueue	15	[5, 15]	
Dequeue	--	[15]	5 is removed

9. Calculations & Analysis

- Stack and Queue implemented using array have time complexity:
  - **Push/Pop/Enqueue/Dequeue**: **O(1)**
  - **Display**: **O(n)**
- Stack overflow occurs when top == MAX - 1.
- Queue overflow occurs when rear == MAX - 1.

10. Result & Interpretation

- Stack and queue operations were successfully implemented using arrays.
- Overflow and underflow conditions were properly handled.
- Students demonstrated understanding of LIFO and FIFO behavior through code.

11. Follow-up Questions

1. What happens when you try to pop from an empty stack?
2. How would you implement a circular queue using arrays?
3. Can a queue be implemented using two stacks?
4. What is the maximum size of a stack in this implementation?
5. What are real-world applications of stacks and queues?

12. Extension and Follow-up Activities (if applicable)

- Implement **stack and queue using linked list**.
- Implement **circular queue using array**.
- Implement **priority queue** or **double-ended queue (deque)**.
- Visualize operations using graphical or simulation tools.

13. Assessments

- Quiz on LIFO vs FIFO.
- Code correction (identify bugs in stack/queue code).
- Viva-voce questions on overflow/underflow scenarios.
- Write functions for peek (stack) and isFull/isEmpty checks.

14. Suggested Readings

- *"Data Structures Using C"* by Reema Thareja
- *"Fundamentals of Data Structures in C"* by Ellis Horowitz, Sartaj Sahni
- Online Tutorials: GeeksforGeeks, Programiz, TutorialsPoint

15. List of Related Experiments

1. Stack operations using arrays
2. Queue operations using arrays
3. Circular Queue implementation
4. Stack using linked list
5. Queue using linked list
6. Expression evaluation using stack
7. Priority queue implementation

Experiment No 2: Implementation of Basic Operations on a Circular Queue using Arrays

1. Aim / Purpose of the Experiment

To implement and demonstrate the basic operations (Insertion, Deletion, and Display) of a **Circular Queue** using arrays in a programming language (C/C++/Java/Python).

2. Learning Outcomes

Upon completion of this experiment, students will be able to:

- Understand the concept of circular queue and its advantages over linear queue.
- Implement the basic operations (enqueue, dequeue, and display) on circular queues.
- Handle overflow and underflow conditions in circular queues.
- Use array-based data structures for queue implementation.

3. Prerequisites

- Knowledge of Arrays
- Understanding of Queue data structure
- Basic programming skills (C/C++/Java/Python)
- Logical and analytical thinking

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer system with:
  - C/C++/Java/Python IDE (e.g., Turbo C++, Code::Blocks, Eclipse, VS Code)
  - Compiler/Interpreter installed
- Text editor for writing code
- Paper/Notebook for manual dry-run and logic design

5. Introduction and Theory

A **Circular Queue** is a linear data structure that follows the FIFO (First In First Out) principle but connects the end of the queue back to the front, forming a circle. It helps in efficient memory utilization by overcoming the limitation of a simple linear queue.

Key Operations:

- **Enqueue** – Insert an element into the circular queue.
- **Dequeue** – Remove an element from the circular queue.
- **Display** – Show all elements in the queue from front to rear.

Circular Queue Conditions:

- **Queue is Empty:** front == -1
- **Queue is Full:** (rear + 1) % size == front

6. Operating Procedure

Step 1: Initialize

- Set front = -1 and rear = -1
- Define an array queue[size]

Step 2: Enqueue Operation

- Check if the queue is full
- If not, update rear = (rear + 1) % size and insert element

- If the queue was initially empty, set front = 0

Step 3: Dequeue Operation

- Check if the queue is empty
- If not, remove queue[front]
- If front == rear, reset both to -1 (queue becomes empty)
- Otherwise, update front = (front + 1) % size

Step 4: Display Operation

- Traverse and print elements from front to rear circularly

Step 5: Execute the program

- Compile and run with various inputs to test all cases

Circular Queue Using Array in C

```
#include <stdio.h>
#define SIZE 5

int queue[SIZE];
int front = -1, rear = -1;

void enqueue(int value) {
   if ((rear + 1) % SIZE == front)
      printf("Queue Overflow\n");
   else {
      if (front == -1) front = 0;
      rear = (rear + 1) % SIZE;
      queue[rear] = value;
      printf("%d enqueued\n", value);
   }
}

void dequeue() {
   if (front == -1)
      printf("Queue Underflow\n");
   else {
      printf("%d dequeued\n", queue[front]);
      if (front == rear)
         front = rear = -1;
      else
         front = (front + 1) % SIZE;
   }
}

void display() {
   if (front == -1)
      printf("Queue is empty\n");
   else {
      printf("Queue: ");
```

```
        int i = front;
        while (1) {
            printf("%d ", queue[i]);
            if (i == rear) break;
            i = (i + 1) % SIZE;
        }
        printf("\n");
    }
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    display();
    dequeue();
    display();
    enqueue(50);
    enqueue(60);  // will cause overflow if queue is full
    display();
    return 0;
}
```

7. Precautions and/or Troubleshooting

- Always check for **queue full** condition before insertion.
- Always check for **queue empty** condition before deletion.
- Ensure correct usage of modulo operation for circular indexing.
- Validate array index boundaries to avoid segmentation faults.

8. Observations

Test Case	Operation Performed	Queue Content (Front to Rear)	Remarks
1	Enqueue(5), (10)	5, 10	Normal Insertion
2	Dequeue()	10	Normal Deletion
3	Fill Queue	-	Queue Full
4	Empty Queue	-	Queue Empty

9. Calculations & Analysis

- **Time Complexity:**
  - Enqueue/Dequeue: **O(1)**
  - Display: **O(n)** (worst case)
- **Space Complexity: O(n)**, where n is the size of the array
- Efficient utilization of space compared to linear queues

10. Result & Interpretation

The circular queue was successfully implemented using arrays. All basic operations performed correctly, and the issues of space wastage in linear queues were resolved.

11. Follow-up Questions

1. What is the difference between a linear queue and a circular queue?
2. How does modulo arithmetic help in circular queue implementation?
3. What happens when we try to insert an element in a full queue?
4. What real-world applications use circular queues?
5. Can circular queues be implemented using linked lists?

12. Extension and Follow-up Activities (if applicable)

- Implement **Circular Queue using Linked List**
- Extend to handle **priority queues** or **double-ended queues (deque)**
- Visualize queue operations using GUI or animations
- Study multi-queue systems and scheduling algorithms

13. Assessments

Short Answer / Viva:

1. Define Circular Queue.
2. What is FIFO?
3. Write conditions to check queue full and empty.

Coding Test:

- Implement a menu-driven circular queue program in a given language.

Assignment:

- Modify your circular queue code to accept character data type instead of integers.

14. Suggested Readings

- Data Structures Using C – Reema Thareja
- Fundamentals of Data Structures – Ellis Horowitz & Sartaj Sahni
- Data Structures and Algorithms Made Easy – Narasimha Karumanchi
- Online Tutorials: GeeksforGeeks, Programiz, TutorialsPoint

15. List of Experiments

1. Implementation of Stack using Array
2. Implementation of Queue using Array
3. Circular Queue using Array *(Current Experiment)*
4. Stack and Queue using Linked List
5. Infix to Postfix Conversion using Stack
6. Evaluation of Postfix Expression
7. Deque Implementation
8. Priority Queue Implementation

Experiment No 3: Evaluation of expressions operations on stacks & queues

1. Aim / Purpose of the Experiment

To implement the evaluation of arithmetic expressions (infix, postfix, prefix) using stacks and queues, and understand how these data structures help in parsing, converting, and evaluating expressions efficiently.

2. Learning Outcomes

After completing this lab, the student will be able to:

- Understand and differentiate between infix, postfix, and prefix notations.
- Use stacks to convert infix to postfix or prefix.
- Evaluate postfix and prefix expressions using stack-based algorithms.
- Understand how queues can be applied for expression management.

3. Prerequisites

- Basic understanding of expressions (arithmetic, logical).
- Knowledge of data structures like Stack and Queue.
- Familiarity with operator precedence and associativity.
- Basics of C programming and arrays/pointers.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer system with C compiler (GCC/MinGW/Turbo C)
- Code editor (VS Code, Code::Blocks, etc.)
- Lab manual/notebook for documentation

5. Introduction and Theory

Expressions are commonly used in programming and must often be evaluated. There are 3 common notations:

- **Infix**: Operators placed between operands (e.g., A + B)
- **Postfix (Reverse Polish Notation)**: Operators follow operands (e.g., A B +)
- **Prefix (Polish Notation)**: Operators precede operands (e.g., + A B)

Why use Stacks?

- Stacks follow the LIFO principle, which aligns with the nested and recursive nature of expression evaluation.
- Used to convert infix to postfix/prefix and to evaluate postfix/prefix expressions.

Why Queues (optional)?

- Queues can be used to store postfix tokens in order or manage intermediate expression data.

6. Operating Procedure

A. Infix to Postfix Conversion (using stack)

1. Scan the infix expression.
2. If operand, add to output.
3. If operator, pop from stack to output based on precedence, then push the current operator.
4. Push (to stack, pop all until) when encountered.
5. At the end, pop all remaining operators from the stack.

B. Postfix Evaluation (using stack)

1. Scan the postfix expression.

2. If operand, push to stack.
3. If operator, pop two operands, apply the operator, push result.

C. Prefix Evaluation (similar to postfix, but scanned right to left)

Postfix Expression Evaluation Using Stack

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

int stack[20], top = -1;

void push(int val) {
   stack[++top] = val;
}

int pop() {
   return stack[top--];
}

int main() {
   char exp[] = "53*62/+4-"; // (5*3)+(6/2)-4 = 15+3-4 = 14
   int i, op1, op2;
   for (i = 0; exp[i]; i++) {
      if (isdigit(exp[i]))
         push(exp[i] - '0');
      else {
         op2 = pop(); op1 = pop();
         switch (exp[i]) {
            case '+': push(op1 + op2); break;
            case '-': push(op1 - op2); break;
            case '*': push(op1 * op2); break;
            case '/': push(op1 / op2); break;
         }
      }
   }
   printf("Result = %d\n", pop());
   return 0;
}
```

7. Precautions and/or Troubleshooting

- Always balance parentheses in infix expressions.
- Ensure valid postfix/prefix expressions.
- Avoid invalid characters and operators.
- Handle divide-by-zero errors carefully.

8. Observations

Expression Type	Input Expression	Output (Converted/Evaluated)
Infix to Postfix	A + B * C	A B C * +
Postfix Eval	5 6 + 2 *	22
Prefix Eval	* + 5 6 2	22

9. Calculations & Analysis

Postfix: 6 2 3 + - 3 8 2 / + *

Step-by-step:

1. 2 3 + → 5
2. 6 - 5 → 1
3. 8 2 / → 4
4. 3 + 4 → 7
5. 1 * 7 → 7 → Final Result

10. Result & Interpretation

The implemented program successfully converts infix expressions to postfix/prefix and evaluates postfix/prefix using stack operations. The correctness of results confirms the usefulness of stacks in expression evaluation.

11. Follow-up Questions

1. What is the advantage of postfix over infix?
2. Can an infix expression be evaluated without conversion?
3. Why is stack more suitable for expression evaluation than queue?
4. What happens if you use a stack for queue operations?

12. Extension and Follow-up Activities (if applicable)

- Add support for multi-digit operands and variables.
- Implement error checking for invalid expressions.
- Build a mini calculator using these operations.
- Explore expression trees for evaluation.

13. Assessments

- Convert A + (B - C) * D / E to postfix and prefix.
- Evaluate: Postfix 3 4 * 5 6 * +
- Code snippets: Identify outputs of given stack-based evaluations.
- Viva: Explain each step of the postfix evaluation algorithm.

14. Suggested Readings

- "Data Structures Using C" by Reema Thareja
- "Fundamentals of Data Structures in C" by Horowitz & Sahni
- GeeksforGeeks – Expression Evaluation
- NPTEL Lectures on Data Structures

15. List of Related Experiments

1. Implementation of Stack using Array and Linked List
2. Implementation of Queue using Array and Linked List
3. Infix to Postfix Conversion
4. Postfix Expression Evaluation
5. Infix to Prefix Conversion
6. Prefix Expression Evaluation
7. Expression Evaluation using Multiple Stacks
8. Expression Handling using Queue (optional)

Experiment No 4: Implementation of Insertion, Deletion, and Inversion (Reversing) in a Singly Linked List.

1. Aim / Purpose of the Experiment

To implement insertion, deletion, and inversion (reversing) operations on a singly linked list and understand the dynamic management of data structures using pointers in C programming.

2. Learning Outcomes

- Understand the concept and structure of singly linked lists.
- Learn how to dynamically insert nodes at various positions.
- Implement deletion of nodes from the linked list.
- Perform inversion (reversal) of the singly linked list.
- Develop skills in pointer manipulation and dynamic memory allocation.

3. Prerequisites

- Basic knowledge of C programming language.
- Understanding of pointers and dynamic memory allocation.
- Familiarity with data structures, especially linked lists.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (e.g., GCC)
- IDE or text editor (e.g., Code::Blocks, Visual Studio Code)
- Operating system supporting C programming (Windows/Linux/MacOS)

5. Introduction and Theory

Singly Linked List:

A linked list is a dynamic data structure consisting of nodes. Each node contains data and a pointer to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory allocation.

Operations:

- **Insertion**: Adding a new node at the beginning, end, or any position in the list.
- **Deletion**: Removing a node from the beginning, end, or a specified position.
- **Inversion (Reversing)**: Changing the direction of the linked list so that the last node becomes the first.

These operations help in understanding memory management, pointer manipulation, and dynamic data handling.

6. Operating Procedure

1. **Create a singly linked list node structure:**

```
struct Node {
   int data;
   struct Node* next;
};
```

2. **Insertion operations:**
   - Insert at the beginning.
   - Insert at the end.
   - Insert at a specified position.
3. **Deletion operations:**
   - Delete from the beginning.
   - Delete from the end.
   - Delete from a specified position.
4. **Inversion operation:**
   - Traverse the list and reverse the next pointers.
5. **Display the list after each operation to verify correctness.**

Implementation of Insertion, Deletion, and Inversion (Reversing) in a Singly Linked List

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

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

struct Node* head = NULL;

void insert(int val) {
   struct Node* newNode = malloc(sizeof(struct Node));
   newNode->data = val;
   newNode->next = head;
   head = newNode;
}

void delete(int val) {
   struct Node *temp = head, *prev = NULL;
   while (temp && temp->data != val) {
      prev = temp;
      temp = temp->next;
   }
   if (!temp) {
      printf("%d not found\n", val);
      return;
   }
   if (!prev)
      head = temp->next;
```

```
    else
        prev->next = temp->next;
    free(temp);
}

void reverse() {
    struct Node *prev = NULL, *curr = head, *next;
    while (curr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    head = prev;
}

void display() {
    struct Node* temp = head;
    printf("List: ");
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    insert(10);
    insert(20);
    insert(30);
    display();      // 30 20 10
    delete(20);
    display();      // 30 10
    reverse();
    display();      // 10 30
    return 0;
}
```

7. Precautions and/or Troubleshooting

- Always check for NULL pointers before dereferencing.
- Ensure proper memory allocation and deallocation using malloc and free.
- Validate user input for positions in insertion and deletion.
- After deletion, update pointers carefully to avoid memory leaks.
- Test the program with edge cases like empty list, single element list, and invalid positions.

8. Observations

- Observe the list contents after each insertion and deletion.

- Verify the correctness of the reversed linked list.
- Note the behavior on boundary cases.

9. Calculations & Analysis

- Time complexity for insertion and deletion at the beginning: **O(1)**.
- Time complexity for insertion and deletion at a specific position or end: **O(n)**.
- Space complexity depends on the number of nodes (dynamic).

10. Result & Interpretation

- The linked list allows dynamic memory allocation.
- Insertions and deletions modify the list correctly without memory corruption.
- The inversion operation correctly reverses the linked list pointers.

11. Follow-up Questions

1. How does the linked list differ from an array?
2. What are the advantages and disadvantages of linked lists?
3. How does pointer manipulation help in dynamic data structures?
4. What modifications are required for doubly linked lists?
5. How to detect and prevent memory leaks in linked list operations?

12. Extension and Follow-up Activities (if applicable)

- Implement circular linked lists.
- Implement doubly linked lists with forward and backward traversal.
- Add functions for sorting the linked list.
- Implement search operation in the linked list.
- Write programs to detect and remove loops in linked lists.

13. Assessments

- Write a program for insertion at beginning, end, and a given position.
- Write a program for deletion from beginning, end, and a given position.
- Implement a function to reverse the linked list.
- Test all functions with valid and invalid inputs.
- Analyze memory usage and pointer safety.

14. Suggested Readings

- “Data Structures Using C” by Reema Thareja
- “Data Structures and Algorithms in C” by Adam Drozdek
- “Let Us C” by Yashavant Kanetkar (Linked list chapters)
- Online tutorials on GeeksforGeeks and Tutorialspoint for linked lists.

15. List of Experiments

1. Implementation of singly linked list basic operations (creation, traversal).
2. Insertion operations at beginning, end, and specific position.
3. Deletion operations from beginning, end, and specific position.

4. Inversion (Reversing) of a singly linked list.
5. Detecting and handling edge cases in linked list operations.

Experiment No 5: Implementation of Circular Linked List and Doubly Linked List

1. Aim / Purpose of the Experiment

To implement and perform basic operations (insertion, deletion, traversal) on **Circular Linked List** and **Doubly Linked List** using C programming.

2. Learning Outcomes

After completing this experiment, students will be able to:

- Understand the structure and working of circular and doubly linked lists.
- Implement insertion and deletion operations in both list types.
- Traverse circular and doubly linked lists.
- Compare linear, circular, and doubly linked list structures.

3. Prerequisites

- Knowledge of pointers and dynamic memory allocation in C.
- Understanding of single linked list.
- Concepts of data structures and memory management.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (GCC/MinGW/Turbo C)
- Code editor (Code::Blocks, VS Code, etc.)
- Lab notebook/manual for observations

5. Introduction and Theory

Doubly Linked List:

A list where each node contains:

- data: to store value
- prev: pointer to the previous node
- next: pointer to the next node
  Allows **bidirectional traversal**.

Circular Linked List:

A variation of a singly/doubly linked list where the **last node points to the first node**, forming a loop. Traversal must be controlled to avoid infinite looping.

6. Operating Procedure

A. Doubly Linked List:

1. Define a struct node with prev, data, and next.
2. Perform operations:
   - Insertion at beginning/end

- Deletion from beginning/end
- Forward and backward traversal

B. Circular Linked List:

1. Define a struct node with data and next.
2. Insertion:
   - At beginning (adjust last node's next)
   - At end (traverse and insert)
3. Deletion:
   - From beginning or end
4. Traverse until the starting node is reached again.

Doubly Linked List (Insertion at End & Display)

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

struct Node {
   int data;
   struct Node *prev, *next;
} *head = NULL;

void insert(int val) {
   struct Node *newNode = malloc(sizeof(struct Node));
   newNode->data = val;
   newNode->next = NULL;
   if (!head) {
      newNode->prev = NULL;
      head = newNode;
   } else {
      struct Node *temp = head;
      while (temp->next)
         temp = temp->next;
      temp->next = newNode;
      newNode->prev = temp;
   }
}

void display() {
   struct Node *temp = head;
   while (temp) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
}

int main() {
   insert(5); insert(15); insert(25);
   display();
```

Circular Linked List (Insertion & Display)

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

struct Node {
   int data;
   struct Node *next;
} *head = NULL;

void insert(int val) {
   struct Node *newNode = malloc(sizeof(struct Node));
   newNode->data = val;
   if (!head) {
      head = newNode;
      newNode->next = head;
   } else {
      struct Node *temp = head;
      while (temp->next != head)
         temp = temp->next;
      temp->next = newNode;
      newNode->next = head;
   }
}

void display() {
   if (!head) return;
   struct Node *temp = head;
   do {
      printf("%d ", temp->data);
      temp = temp->next;
   } while (temp != head);
}

int main() {
   insert(10); insert(20); insert(30);
   display();
```

```
    return 0;
}
```

```
    return 0;
}
```

7. Precautions and/or Troubleshooting

- Avoid memory leaks by freeing deleted nodes.
- Ensure loop termination in circular traversal.
- Update all necessary pointers (prev and next) carefully.
- Check for NULL pointers before dereferencing.

8. Observations

Operation Type	Linked List Type	Inputs	Outputs / Traversal Order
Insertion at End	Doubly	10 → 20 → 30	10 20 30 (Forward & Reverse)
Deletion at Start	Circular	10 → 20 → 30	20 30 (circular traversal)
Insertion at Middle	Doubly	Insert 15	10 15 20 30

9. Calculations & Analysis

- Memory used = number of nodes × size of struct node
- Pointer updates per insertion/deletion = 2–4 depending on position
- Time complexity:
    - Insertion/Deletion: O(1) at known position
    - Traversal: O(n)

10. Result & Interpretation

The experiment successfully demonstrated the implementation of both Circular and Doubly Linked Lists. All insertion, deletion, and traversal operations were executed and verified through outputs.

11. Follow-up Questions

1. How does a circular list differ from a singly linked list?
2. Why is backward traversal not possible in singly linked list?
3. What are real-world uses of circular linked lists?
4. What are the advantages of a doubly linked list over singly linked list?

12. Extension and Follow-up Activities (if applicable)

- Implement sorted insertion in doubly linked list.
- Create a circular doubly linked list.
- Develop a music playlist application using circular linked list logic.

13. Assessments

- Write a program to delete a node with a given key in a circular linked list.
- Viva Questions:
    - What happens if the list is empty?
    - How do you detect the end of a circular list?

- MCQs on pointer manipulations and list types.

14. Suggested Readings

- **"Data Structures Using C"** – Reema Thareja
- **"Fundamentals of Data Structures in C"** – Horowitz & Sahni
- GeeksforGeeks: Linked Lists
- TutorialsPoint: Circular and Doubly Linked Lists

15. List of Experiments

1. Implementation of Singly Linked List
2. Insertion and Deletion in Circular Linked List
3. Insertion and Deletion in Doubly Linked List
4. Creation of Circular Doubly Linked List
5. Reversing a Doubly Linked List
6. Searching a Node in a Linked List
7. Sorting Elements in a Linked List

Experiment No 6: Implementation of stacks & queues using linked lists

1. Aim / Purpose of the Experiment

To implement stack and queue data structures using linked lists in C language and understand their dynamic behavior through pointer manipulation and dynamic memory allocation.

2. Learning Outcomes

- Understand the concepts of stack and queue data structures.
- Learn how to implement stacks and queues using linked lists dynamically.
- Perform push and pop operations for stacks.
- Perform enqueue and dequeue operations for queues.
- Develop skills in handling pointers, dynamic memory, and linked list traversal.

3. Prerequisites

- Basic knowledge of C programming language.
- Understanding of pointers and dynamic memory allocation in C.
- Basic knowledge of linear data structures like stacks and queues.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (e.g., GCC)
- Text editor or IDE (e.g., Code::Blocks, Visual Studio Code)
- Operating system supporting C programming (Windows/Linux/MacOS)

5. Introduction and Theory

Stack:
A linear data structure that follows Last In First Out (LIFO) principle. Elements are added (pushed) and removed (popped) only from the top of the stack. Using linked lists, stacks grow dynamically without fixed size constraints.
Queue:
A linear data structure that follows First In First Out (FIFO) principle. Elements are added (enqueued) at the rear and removed (dequeued) from the front. Using linked lists, queues also grow dynamically.
Link List:
A collection of nodes where each node contains data and a pointer to the next node. It enables dynamic memory management for stack and queue operations without size limitations.

6. **Operating Procedure**
Stack Implementation using Linked List:

1. Define a node structure with data and next pointer.
2. Initialize the top pointer as NULL.
3. Push operation:
    - Create a new node dynamically.
    - Point its next to the current top.
    - Update top to this new node.
4. Pop operation:
    - Check if stack is empty (top == NULL).
    - Save the data from the top node.
    - Move top to next node and free the popped node.
    - Return popped data.

Queue Implementation using Linked List:

1. Define a node structure with data and next pointer.
2. Maintain two pointers: front and rear. Initialize both to NULL.
3. Enqueue operation:
    - Create a new node dynamically.
    - If queue is empty, set both front and rear to new node.
    - Else, link the new node after rear and update rear.
4. Dequeue operation:
    - Check if queue is empty (front == NULL).
    - Save the data from front node.
    - Move front to next node and free the dequeued node.
    - If front becomes NULL, set rear to NULL.
    - Return dequeued data.

Stack Using Linked List in C	Queue Using Linked List in C

#include <stdio.h>
#include <stdlib.h>
 	
struct Node {
  int data;
  struct Node* next;
};
	struct Node* front = NULL;
	struct Node* rear = NULL;
	
	void enqueue(int val) {
	  struct Node* temp = malloc(sizeof(struct Node));
	  temp->data = val;
	  temp->next = NULL;

```
struct Node* top = NULL;

void push(int val) {
   struct Node* temp = malloc(sizeof(struct Node));
   temp->data = val;
   temp->next = top;
   top = temp;
}

void pop() {
   if (!top) {
      printf("Stack Underflow\n");
      return;
   }
   struct Node* temp = top;
   printf("Popped: %d\n", top->data);
   top = top->next;
   free(temp);
}

void displayStack() {
   struct Node* temp = top;
   printf("Stack: ");
   while (temp) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}
```

```
   if (!rear)
      front = rear = temp;
   else {
      rear->next = temp;
      rear = temp;
   }
}

void dequeue() {
   if (!front) {
      printf("Queue Underflow\n");
      return;
   }
   struct Node* temp = front;
   printf("Dequeued: %d\n", front->data);
   front = front->next;
   if (!front) rear = NULL;
   free(temp);
}

void displayQueue() {
   struct Node* temp = front;
   printf("Queue: ");
   while (temp) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}
```

7. **Precautions and/or Troubleshooting**
   - Always check for NULL pointers before dereferencing.
   - Properly allocate and free memory to avoid leaks.
   - Validate operations to prevent underflow (pop or dequeue on empty structure).
   - Test edge cases such as empty stack/queue and single element scenarios.
   - Handle pointer updates carefully during insertion and deletion.

8. **Observations**
   - Observe the behavior of push, pop, enqueue, and dequeue operations on the linked list.
   - Verify dynamic size changes of stack and queue with successive operations.
   - Confirm that the order of insertion and removal respects LIFO for stack and FIFO for queue.

9. **Calculations & Analysis**
   - Time complexity for push and pop (stack): O(1).

- Time complexity for enqueue and dequeue (queue): O(1).
- Space complexity is dynamic and proportional to the number of elements stored.

10. **Result & Interpretation**
    - The stack and queue are successfully implemented using linked lists with dynamic memory allocation.
    - Push, pop, enqueue, and dequeue operations function correctly following their respective principles.
    - Memory is efficiently used without any fixed size constraints.

11**. Follow-up Questions**
    1. How does using a linked list for stack/queue differ from using arrays?
    2. What are the advantages and disadvantages of linked list implementations?
    3. How do stack overflow and underflow errors occur, and how to handle them?
    4. How can you implement a circular queue using linked lists?
    5. What are real-life applications of stacks and queues?

12. **Extension and Follow-up Activities (if applicable)**
    - Implement a circular queue using linked lists.
    - Implement a double-ended queue (deque) using doubly linked lists.
    - Write programs to convert infix expressions to postfix using stacks.
    - Implement priority queues using linked lists.
    - Analyze memory usage during various operations with dynamic input sizes.

13. **Assessments**
    - Write programs for push and pop operations in a stack using linked lists.
    - Write programs for enqueue and dequeue operations in a queue using linked lists.
    - Demonstrate stack and queue operations with sample data.
    - Analyze the time and space efficiency of your implementation.
    - Test for boundary conditions and error handling.

14. **Suggested Readings**
    - “Data Structures Using C” by Reema Thareja
    - “Data Structures and Algorithms in C” by Adam Drozdek
    - “Let Us C” by Yashavant Kanetkar
    - Online tutorials on GeeksforGeeks and TutorialsPoint (Stack and Queue using linked lists)

15. **List of Experiments**
    1. Implementation of basic stack operations using linked lists.
    2. Implementation of basic queue operations using linked lists.
    3. Handling underflow and overflow scenarios in stacks and queues.
    4. Conversion of infix to postfix expression using stack (linked list).
    5. Implementing circular queues using linked lists (advanced).
    6. Implementing double-ended queues (deque) using doubly linked lists.

Experiment No 7: Polynomial Representation and Operations using array.

1. Aim / Purpose of the Experiment

To represent polynomials using arrays and perform basic polynomial operations such as addition and multiplication using array manipulation in C language.

2. Learning Outcomes

- Understand how to represent polynomials using arrays.
- Learn how to perform polynomial addition and multiplication.
- Develop skills in array manipulation and algorithmic thinking for polynomial operations.
- Gain experience in implementing mathematical algorithms in C.

3. Prerequisites

- Basic knowledge of C programming language.
- Understanding of arrays and loops in C.
- Basic understanding of polynomials and their arithmetic.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (e.g., GCC)
- Text editor or IDE (e.g., Code::Blocks, Visual Studio Code)
- Operating system supporting C programming (Windows/Linux/MacOS)

5. Introduction and Theory

A **polynomial** is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients.

For example:

$P(x) = 4x^3 + 2x^2 + 5x + 7$

In array representation, each term can be stored using a structure with two fields:

- **Coefficient**
- **Exponent**

Or directly in an array, where each index represents the exponent and the value at that index is the coefficient.

Operations:

- **Addition**: Combine terms with the same exponents.
- **Multiplication**: Multiply each term of the first polynomial with every term of the second.

6. Operating Procedure

1. Define two arrays to store coefficients of the two polynomials.
2. Input the degree and coefficients of each polynomial from the user.
3. Initialize result arrays for addition and multiplication.
4. For addition, add coefficients of corresponding indices and store in result array.
5. For multiplication, use nested loops to multiply each term and add to the proper index in result array.
6. Display the resulting polynomial after addition and multiplication.

Polynomial Representation and Addition Using Arrays in C

```
#include <stdio.h>

#define MAX 10

struct Term {
   int coeff;
   int expo;
};

void readPoly(struct Term poly[], int *n) {
   printf("Enter number of terms: ");
   scanf("%d", n);
   for (int i = 0; i < *n; i++) {
      printf("Enter coeff and expo for term %d: ", i + 1);
      scanf("%d%d", &poly[i].coeff, &poly[i].expo);
   }
}

void displayPoly(struct Term poly[], int n) {
   for (int i = 0; i < n; i++) {
      printf("%dx^%d", poly[i].coeff, poly[i].expo);
      if (i != n - 1)
         printf(" + ");
   }
   printf("\n");
}

void addPoly(struct Term a[], int n1, struct Term b[], int n2, struct Term res[], int *n3) {
   int i = 0, j = 0, k = 0;
   while (i < n1 && j < n2) {
      if (a[i].expo > b[j].expo)
         res[k++] = a[i++];
      else if (a[i].expo < b[j].expo)
         res[k++] = b[j++];
      else {
         res[k].expo = a[i].expo;
         res[k].coeff = a[i].coeff + b[j].coeff;
         i++; j++; k++;
      }
   }
   while (i < n1) res[k++] = a[i++];
   while (j < n2) res[k++] = b[j++];
   *n3 = k;
}
```

```
int main() {
    struct Term p1[MAX], p2[MAX], sum[MAX];
    int n1, n2, n3;

    printf("First Polynomial\n");
    readPoly(p1, &n1);
    printf("Second Polynomial\n");
    readPoly(p2, &n2);

    addPoly(p1, n1, p2, n2, sum, &n3);

    printf("P1: "); displayPoly(p1, n1);
    printf("P2: "); displayPoly(p2, n2);
    printf("Sum: "); displayPoly(sum, n3);

    return 0;
}
```

7. **Precautions and/or Troubleshooting**
   - Ensure that input degrees and coefficients are correctly entered.
   - Handle cases where polynomials have different degrees by initializing arrays properly.
   - Avoid array out-of-bounds errors by careful indexing.
   - Verify the output by manually calculating small polynomials.

8. **Observations**
   - Observe the coefficients of the resultant polynomial after addition.
   - Observe the coefficients and degree of the resultant polynomial after multiplication.
   - Compare results with manual calculations to verify correctness.

9. **Calculations & Analysis**
   - Addition involves element-wise sum of coefficients.
   - Multiplication involves convolution-like operation where degree of result is sum of degrees of input polynomials.
   - Analyze the time complexity: addition is O(n), multiplication is O(n^2) where nnn is the degree of polynomial.

10. **Result & Interpretation**
   - Successful representation of polynomials using arrays.
   - Correct polynomial addition and multiplication results as output.
   - Dynamic handling of polynomials of different degrees.

11. **Follow-up Questions**

1. How does array representation simplify polynomial operations?
2. What changes are needed to represent sparse polynomials efficiently?
3. How can you implement polynomial subtraction using this method?
4. What is the time complexity of polynomial multiplication and how can it be optimized?
5. Can you use linked lists to represent polynomials instead? What are the pros and cons?

12. **Extension and Follow-up Activities (if applicable)**
- Implement polynomial subtraction and division.
- Represent and operate on sparse polynomials using linked lists.
- Implement Horner's method for polynomial evaluation.
- Optimize polynomial multiplication using algorithms like Karatsuba.
- Extend the program to handle polynomials with floating-point coefficients.

13. **Assessments**
- Write a C program to add two polynomials represented as arrays.
- Write a C program to multiply two polynomials represented as arrays.
- Test the program with various polynomial degrees and coefficients.
- Analyze and explain the output of your programs.
- Discuss the limitations of array-based polynomial representation.

14. **Suggested Readings**
- "Data Structures Using C" by Reema Thareja
- "Let Us C" by Yashavant Kanetkar
- "Introduction to Algorithms" by Cormen, Leiserson, Rivest (for algorithm analysis)
- Online tutorials on polynomial operations in C (GeeksforGeeks, TutorialsPoint)

15. **List of Experiments**
1. Polynomial representation using arrays.
2. Polynomial addition using arrays.
3. Polynomial multiplication using arrays.
4. Polynomial subtraction using arrays (optional).
5. Polynomial evaluation using Horner's method (optional).
6. Sparse polynomial representation and operations using linked lists (advanced).

Experiment No 8: Representation and Operations on Sparse Matrices Using Arrays

1. Aim / Purpose of the Experiment

To represent sparse matrices efficiently using arrays and perform basic operations such as addition and multiplication on sparse matrices.

2. Learning Outcomes
- Understand the concept of sparse matrices and their efficient representation.
- Learn how to store sparse matrices using compact array formats (like triplet representation).

- Implement addition and multiplication operations on sparse matrices using arrays.
- Improve skills in handling matrix operations with optimized memory usage.

3. Prerequisites

- Basic understanding of matrices and matrix operations.
- Familiarity with arrays and basic programming in C.
- Knowledge of two-dimensional arrays and loops.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (e.g., GCC)
- Text editor or IDE (e.g., Code::Blocks, Visual Studio Code)
- Operating system supporting C programming (Windows/Linux/MacOS)

5. Introduction and Theory

Sparse Matrices:

A sparse matrix is a matrix in which most of the elements are zero. Storing all elements, including zeros, wastes memory.

Representation:

Instead of storing the entire matrix, only store non-zero elements with their row and column indices, often using a 3-tuple (row, column, value). This is called the triplet representation.

Operations:

- Addition: Add corresponding non-zero elements. If an element is zero in one matrix, the non-zero element of the other matrix is retained.
- Multiplication: Multiply matrices based on row-column multiplication rules but only compute when elements are non-zero to save time and space.

6. Operating Procedure

1. Input the dimensions and non-zero elements of two sparse matrices.
2. Store the matrices in triplet form: arrays for rows, columns, and values.
3. Implement addition by merging triplets from both matrices with the same row and column.
4. Implement multiplication by multiplying matching row and column indices and accumulating sums.
5. Display the resulting sparse matrix in triplet form and optionally in full matrix form.

Sparse Matrix Representation and Addition in C

```
#include <stdio.h>

#define MAX 10

typedef struct {
   int row, col, val;
} Term;

void readSparse(Term mat[], int *n) {
   printf("Enter number of non-zero elements: ");
```

```
    scanf("%d", n);
    printf("Enter row col val for each:\n");
    for (int i = 0; i < *n; i++)
        scanf("%d%d%d", &mat[i].row, &mat[i].col, &mat[i].val);
}

void displaySparse(Term mat[], int n) {
    printf("Row Col Val\n");
    for (int i = 0; i < n; i++)
        printf("%3d %3d %3d\n", mat[i].row, mat[i].col, mat[i].val);
}

void addSparse(Term a[], int n1, Term b[], int n2, Term res[], int *n3) {
    int i = 0, j = 0, k = 0;
    while (i < n1 && j < n2) {
        if (a[i].row == b[j].row && a[i].col == b[j].col) {
            res[k] = a[i];
            res[k++].val += b[j++].val;
            i++;
        } else if (a[i].row < b[j].row ||
                (a[i].row == b[j].row && a[i].col < b[j].col)) {
            res[k++] = a[i++];
        } else {
            res[k++] = b[j++];
        }
    }
    while (i < n1) res[k++] = a[i++];
    while (j < n2) res[k++] = b[j++];
    *n3 = k;
}

int main() {
    Term A[MAX], B[MAX], Sum[MAX];
    int n1, n2, n3;

    printf("Matrix A:\n");
    readSparse(A, &n1);
    printf("Matrix B:\n");
    readSparse(B, &n2);

    addSparse(A, n1, B, n2, Sum, &n3);

    printf("\nMatrix A:\n"); displaySparse(A, n1);
    printf("Matrix B:\n"); displaySparse(B, n2);
```

```
    printf("Sum:\n");     displaySparse(Sum, n3);

    return 0;
}
```

7. Precautions and/or Troubleshooting

- Verify the number of non-zero elements does not exceed matrix size.
- Ensure indices entered are within matrix dimensions.
- Handle the case where addition leads to zero values which should be excluded.
- Be careful to avoid array overflow when storing triplets.
- Test the program with small matrices to validate correctness.

8. Observations

- Non-zero elements and their indices for input matrices.
- Number of non-zero elements in the resultant matrix after operations.
- Differences in memory usage compared to full matrix representation.
- Verify results by comparing with manual calculations.

9. Calculations & Analysis

- Analyze space complexity: triplet representation uses O(k) space where k is the number of non-zero elements.
- Time complexity of addition is approximately O(k1+k2) where k1, k2 are non-zero counts of input matrices.
- Multiplication time depends on the sparsity and requires nested loops on non-zero elements.

10. Result & Interpretation

- Efficient storage and representation of sparse matrices.
- Correct computation of addition and multiplication operations with reduced memory use.
- Results validated through comparison with conventional full matrix operations.

11. Follow-up Questions

1. What are the advantages of using sparse matrix representation over full matrix storage?
2. How would you modify the program to handle sparse matrix transpose?
3. What are alternative sparse matrix representations and their benefits?
4. How does sparsity affect the time complexity of matrix multiplication?
5. How can sparse matrix operations be parallelized?

12. Extension and Follow-up Activities (if applicable)

- Implement transpose operation on sparse matrices using triplet form.
- Implement other sparse matrix formats like Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC).
- Explore and implement sparse matrix-vector multiplication.
- Compare performance and memory usage with full matrix operations for large datasets.

13. Assessments

- Write a C program to represent a sparse matrix using triplet arrays.
- Implement addition and multiplication of two sparse matrices stored in triplet form.
- Test the program with different sparse matrices and analyze output.
- Explain the benefits and limitations of your sparse matrix implementation.

14. Suggested Readings

- "Data Structures and Algorithms in C" by Adam Drozdek
- "Introduction to Algorithms" by Cormen et al.
- Online tutorials on sparse matrix representation and operations (GeeksforGeeks, TutorialsPoint)
- Research papers and articles on sparse matrix storage formats.

15. List of Experiments

1. Representation of sparse matrix using triplet arrays.
2. Addition of two sparse matrices using array representation.
3. Multiplication of two sparse matrices using array representation.
4. Transpose of a sparse matrix (extension).
5. Sparse matrix-vector multiplication (extension).

Experiment No 9: Implementation of Binary Search Tree (BST) and Its Operations.

1. Aim / Purpose of the Experiment

To implement a Binary Search Tree (BST) and perform fundamental operations like insertion, deletion, and searching in the BST structure.

2. Learning Outcomes

After completing this experiment, students will be able to:

- Understand the concept and structure of a BST.
- Implement insertion, deletion, and search operations.
- Traverse a BST using inorder, preorder, and postorder methods.
- Analyze time and space complexities of BST operations.
- Apply BSTs to real-world problem-solving.

3. Prerequisites

- Knowledge of data structures (especially trees).
- Proficiency in programming using C/C++ or similar.
- Understanding of recursion and pointers.
- Familiarity with binary tree traversal techniques.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C/C++ compiler (e.g., GCC)
- Code editor (e.g., Code::Blocks, Turbo C++, VS Code)
- Lab manual and record notebook
- Graph paper (for drawing tree diagrams)

5. Introduction and Theory

A Binary Search Tree (BST) is a binary tree with the following properties:

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- Both left and right subtrees must also be binary search trees.

Operations on BST:

- Insertion: Place the new element at its appropriate position.
- Searching: Compare the key with the root and move left/right accordingly.
- Deletion: Three cases:
    - Node is a leaf.
    - Node has one child.
    - Node has two children (find inorder successor/predecessor).

Traversals:

- Inorder (LNR): Visits nodes in ascending order.
- Preorder (NLR)
- Postorder (LRN)

6. Operating Procedure

1. Start the program and define a BST node structure.
2. Initialize the root node to NULL.
3. Implement:
    - Insert function using recursion or iteration.
    - Search function that returns the node if found.
    - Delete function considering all three deletion cases.
4. Implement traversal functions.
5. Display tree traversals.
6. Test with sample input data.

Binary Search Tree (BST) Implementation in C

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

typedef struct Node {
   int data;
   struct Node *left, *right;
} Node;

Node* newNode(int val) {
   Node* n = malloc(sizeof(Node));
   n->data = val; n->left = n->right = NULL;
   return n;
}
```

```
Node* insert(Node* root, int val) {
   if (!root) return newNode(val);
   if (val < root->data) root->left = insert(root->left, val);
   else if (val > root->data) root->right = insert(root->right, val);
   return root;
}

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

Node* search(Node* root, int val) {
   if (!root || root->data == val) return root;
   if (val < root->data) return search(root->left, val);
   return search(root->right, val);
}

int main() {
   Node* root = NULL;
   int arr[] = {50, 30, 20, 40, 70, 60, 80};
   for (int i = 0; i < 7; i++)
      root = insert(root, arr[i]);

   printf("Inorder: ");
   inorder(root);
   printf("\n");

   int key = 40;
   printf("Search %d: %s\n", key, search(root, key) ? "Found" : "Not Found");

   return 0;
}
```

7. Precautions and/or Troubleshooting

- Always initialize pointers properly.
- Carefully handle deletion cases (especially two-child case).
- Avoid memory leaks by properly freeing deleted nodes.
- Check for null pointers before accessing child nodes.

8. Observations

Input Sequence	Inorder Traversal	Preorder Traversal	Postorder Traversal
50, 30, 70	30 50 70	50 30 70	30 70 50
40, 20, 60, 80	20 40 60 80	40 20 60 80	20 80 60 40

9. Calculations & Analysis

- Time Complexity:
  - Best/Average Case: O(log n)
  - Worst Case (unbalanced): O(n)
- Space Complexity:
  - Recursive Stack: O(h) where h = height of tree

10. Result & Interpretation

The Binary Search Tree was successfully implemented. The BST allows efficient insertion, deletion, and searching when balanced. Traversals verified tree structure correctness.

11. Follow-up Questions

1. What are the advantages of using a BST over a linked list?
2. How do you balance a BST?
3. What happens when elements are inserted in sorted order?
4. Compare BST with AVL tree.

12. Extension and Follow-up Activities (if applicable)

- Implement self-balancing trees (AVL, Red-Black).
- Visualize the tree structure graphically.
- Count total nodes, leaf nodes, height of the BST.
- Implement BST using file input.

13. Assessments

- Code walkthrough and explanation.
- Trace insert/delete/search operations manually.
- Analyze time complexity in different scenarios.
- Short quiz on BST properties and traversals.

14. Suggested Readings

- “Data Structures Using C” by Reema Thareja
- “Fundamentals of Data Structures in C” by Horowitz, Sahni
- NPTEL Lectures on Trees and BSTs
- GeeksforGeeks: BST Tutorials

15. List of Experiments

1. Create and insert nodes in BST.
2. Search a node in BST.
3. Delete a node from BST.

4. Traverse BST using inorder, preorder, and postorder.
5. Count nodes, height of BST.
6. Application of BST in searching and sorting.

Experiment No 10: Implementation of threaded binary tree traversal

1. Aim / Purpose of the Experiment

To implement and understand the traversal of a threaded binary tree using inorder traversal without recursion or stack.

2. Learning Outcomes

- Understand the concept of threaded binary trees.
- Learn how to create threaded binary trees.
- Perform inorder traversal of threaded binary trees efficiently.
- Avoid recursion and stack by using threaded pointers.

3. Prerequisites

- Basic knowledge of binary trees.
- Understanding of binary tree traversals (inorder, preorder, postorder).
- Familiarity with pointers and dynamic memory allocation in C.
- Concepts of recursion and iterative algorithms.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (e.g., GCC)
- IDE or text editor (Code::Blocks, Dev-C++, Visual Studio Code, etc.)
- Operating system (Windows/Linux/MacOS)

5. Introduction and Theory

Threaded Binary Tree:

In a conventional binary tree, many NULL pointers exist for nodes with missing children. Threaded binary trees use these NULL pointers to point to the inorder predecessor or successor, enabling traversal without recursion or stack.

Right Thread:

If the right child pointer is NULL, it can be used to point to the inorder successor, called a "thread". A flag or boolean is maintained to distinguish normal right child pointers from threads.

Advantages:

- Efficient inorder traversal without extra memory (stack or recursion).
- Faster traversal in some cases.

6. Operating Procedure

1. Create Node Structure: Include data, left and right pointers, and a rightThread flag.
2. Insert Nodes: Insert nodes maintaining threads (right pointers to inorder successor).
3. Find Leftmost Node: Start traversal at the leftmost node.
4. Traverse Inorder:

- Print current node's data.
- If rightThread flag is set, move to right pointer (inorder successor).
- Else, find the leftmost node in the right subtree.

5. Compile and run the program.

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

typedef struct Node {
   int data;
   struct Node *left, *right;
   int rthread; // 1 if right pointer is thread
} Node;

Node* insert(Node *root, int key) {
   if (!root) {
      Node* n = malloc(sizeof(Node));
      n->data = key; n->left = n->right = NULL; n->rthread = 1;
      return n;
   }
   if (key < root->data)
      root->left = insert(root->left, key);
   else if (key > root->data) {
      if (root->rthread) {
         Node* n = malloc(sizeof(Node));
         n->data = key; n->left = NULL; n->right = root->right; n->rthread = 1;
         root->right = n; root->rthread = 0;
      } else
         root->right = insert(root->right, key);
   }
   return root;
}

Node* leftmost(Node* root) {
   while (root && root->left) root = root->left;
   return root;
}

void inorder(Node* root) {
   for (Node* cur = leftmost(root); cur != NULL; ) {
      printf("%d ", cur->data);
      if (cur->rthread)
         cur = cur->right;
      else
```

```
        cur = leftmost(cur->right);
    }
}

int main() {
    Node* root = NULL;
    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 25);
    root = insert(root, 35);

    printf("Inorder traversal of threaded binary tree:\n");
    inorder(root);
    return 0;
}
```

7. **Precautions and/or Troubleshooting**

- Ensure proper initialization of thread flags.
- Avoid inserting duplicate elements.
- Verify correct updating of threaded pointers during insertion.
- Check for memory leaks by freeing allocated nodes (not shown in simple implementation).
- Use appropriate data types and handle user inputs carefully.

8. Observations

- On inserting nodes in increasing or mixed order, note how threads are set.
- Observe traversal output matches inorder traversal sequence.
- No recursion or stack is used during traversal.

9. Calculations & Analysis

- Time complexity of insertion: O(h) where h is tree height.
- Traversal time complexity: O(n), n = number of nodes.
- Space complexity improved by not using auxiliary stack or recursion.

10. Result & Interpretation

The threaded binary tree traversal prints the nodes in inorder sequence correctly using threads instead of recursion or stack. The program demonstrates efficient traversal with minimal overhead.

11. Follow-up Questions

- What are the benefits of threaded binary trees over regular binary trees?
- How would you modify the code to support preorder or postorder threaded trees?
- How is the threaded binary tree different from a threaded AVL tree?
- How can deletion be implemented in threaded binary trees?

12. Extension and Follow-up Activities (if applicable)

- Implement deletion operation in threaded binary trees.
- Implement preorder threaded binary tree traversal.
- Modify the program to create double-threaded binary trees (threads on both left and right NULL pointers).
- Compare performance with recursive inorder traversal on large datasets.

13. Assessments

- Write a program to insert nodes in a threaded binary tree.
- Perform inorder traversal using threads.
- Explain how threads are maintained during insertion.
- Identify and fix issues in a given threaded tree implementation.
- Compare traversal times with recursive and iterative methods.

14. Suggested Readings

- "Data Structures Using C" by Reema Thareja
- "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss
- "Fundamentals of Data Structures" by Horowitz, Sahni & Anderson-Freed
- Online tutorials on threaded binary trees (GeeksforGeeks, TutorialsPoint)
- Research papers on threaded binary trees and their applications

15. List of Experiments (related to this module)

- Implementation of Binary Tree Traversals (Recursive and Non-Recursive)
- Implementation of Threaded Binary Tree Traversal
- Construction of Binary Search Tree and its Traversals
- Insertion and Deletion in Threaded Binary Trees
- Application of Threaded Binary Trees in Expression Parsing

Experiment No 11: Implementation of AVL tree

1. Aim / Purpose of the Experiment

To implement an AVL tree and perform insertion operations while maintaining its balanced property using rotations.

2. Learning Outcomes

After completing this experiment, students will be able to:

- Understand the concept and properties of AVL trees.
- Implement insertion in AVL trees with appropriate rotations.
- Maintain balance factors of nodes.
- Apply tree rotations (LL, RR, LR, RL) correctly.
- Analyze time complexity improvements using self-balancing trees.

3. Prerequisites

- Knowledge of Binary Search Trees (BST).

- Understanding of recursion and pointer manipulation.
- Familiarity with height-balanced trees.
- Basic programming skills in C/C++.

4. Materials / Equipment / Apparatus / Devices / Software Required

- PC or Laptop
- Text editor or IDE (e.g., Code::Blocks, Dev-C++, VS Code)
- GCC or compatible C compiler
- Operating System: Windows/Linux

5. Introduction and Theory

AVL Tree (Adelson-Velsky and Landis Tree):

An AVL tree is a self-balancing binary search tree in which the difference between heights of left and right subtrees (known as the balance factor) is at most one for every node.

Key Properties:

- Balance Factor = Height(Left Subtree) - Height(Right Subtree)
- Must be in {-1, 0, +1}
- Imbalance corrected using Rotations:
    - LL Rotation
    - RR Rotation
    - LR Rotation
    - RL Rotation

6. Operating Procedure

1. Start program.
2. Define a node structure with key, height, left and right pointers.
3. Implement utility functions:
    - height(): returns node height.
    - getBalance(): calculates balance factor.
    - rotateLeft(), rotateRight(), rotateLR(), rotateRL(): to rebalance tree.
4. Implement insert() function to:
    - Insert node as in BST.
    - Update heights.
    - Check balance factor.
    - Perform appropriate rotations.
5. Traverse the tree (inorder/preorder) to display the structure.
6. Compile and run.
7. Verify balance factor after each insertion.

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

typedef struct Node {
   int key, height;
```

```
   struct Node *left, *right;
} Node;

int height(Node *N) {
   return N ? N->height : 0;
}

int max(int a, int b) { return (a > b) ? a : b; }

Node* newNode(int key) {
   Node* node = malloc(sizeof(Node));
   node->key = key; node->height = 1; node->left = node->right = NULL;
   return node;
}

Node* rightRotate(Node* y) {
   Node* x = y->left; Node* T2 = x->right;
   x->right = y; y->left = T2;
   y->height = max(height(y->left), height(y->right)) + 1;
   x->height = max(height(x->left), height(x->right)) + 1;
   return x;
}

Node* leftRotate(Node* x) {
   Node* y = x->right; Node* T2 = y->left;
   y->left = x; x->right = T2;
   x->height = max(height(x->left), height(x->right)) + 1;
   y->height = max(height(y->left), height(y->right)) + 1;
   return y;
}

int getBalance(Node* N) {
   return N ? height(N->left) - height(N->right) : 0;
}

Node* insert(Node* node, int key) {
   if (!node) return newNode(key);
   if (key < node->key) node->left = insert(node->left, key);
   else if (key > node->key) node->right = insert(node->right, key);
   else return node;

   node->height = 1 + max(height(node->left), height(node->right));
   int balance = getBalance(node);
```

```
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);
  if (balance < -1 && key > node->right->key)
    return leftRotate(node);
  if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }
  if (balance < -1 && key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }
  return node;
}

void inorder(Node* root) {
  if (root) {
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
  }
}

int main() {
  Node* root = NULL;
  int keys[] = {10, 20, 30, 40, 50, 25};
  for (int i = 0; i < 6; i++) root = insert(root, keys[i]);

  printf("Inorder traversal of AVL tree:\n");
  inorder(root);
  return 0;
}
```

7. Precautions and/or Troubleshooting

- Ensure memory allocation is successful (malloc check).
- Correctly update height after every insertion and rotation.
- Be careful with rotation logic to prevent segmentation faults.
- Check balance factor calculations to avoid incorrect rotations.
- Test edge cases (e.g., sorted insertion causing skew).

8. Observations

Insertion Order	Inorder Traversal	Balance Maintained	Rotations Used
10, 20, 30	10 20 30	Yes	RR

Insertion Order	Inorder Traversal	Balance Maintained	Rotations Used
30, 20, 10	10 20 30	Yes	LL
10, 30, 20	10 20 30	Yes	RL
30, 10, 20	10 20 30	Yes	LR

9. Calculations & Analysis

- Time Complexity: O(log n) for insertion and searching.
- Space Complexity: O(n) for storage.
- Rotations ensure the height is maintained at log(n), improving performance over unbalanced BSTs.

---

10. Result & Interpretation

The AVL tree maintains balance during insertions using appropriate rotations. All traversals confirm sorted structure with balanced height properties.

11. Follow-up Questions

1. What is the difference between AVL Tree and Binary Search Tree?
2. Explain each rotation with an example.
3. What happens if balance factor exceeds ±1?
4. How would deletion affect AVL trees?
5. How do AVL Trees compare with Red-Black Trees?

12. Extension and Follow-up Activities

- Implement deletion in AVL trees.
- Compare AVL tree with Red-Black tree in terms of insertion cost.
- Visualize AVL rotations using Graphviz or GUI tools.
- Extend AVL to store duplicate keys using count field.

13. Assessments

- Write code to insert elements into an AVL tree.
- Trace rotation steps for a given sequence.
- Identify and fix bugs in a faulty AVL rotation code.
- Predict output of inorder traversal after insertions.

14. Suggested Readings

- “Data Structures and Algorithm Analysis in C” by Mark Allen Weiss
- “Data Structures Using C” by Reema Thareja
- GeeksforGeeks - AVL Trees
- TutorialsPoint - AVL Tree Basics

15. List of Experiments (Module III - Tree)

- Construction and traversal of binary tree (recursive & non-recursive)
- Representation and operations on sparse matrices using arrays
- Threaded binary tree traversal

- Implementation of AVL tree
- Application of trees in expression evaluation and directory structures

Experiment No 12: Sorting Techniques - Quick Sort (Practice set – Bubble sort, Insertion and Selection Sort, Merge Sort)

1. Aim / Purpose of the Experiment

To implement and analyze various sorting techniques including **Quick Sort** and practice fundamental sorting algorithms like **Bubble Sort**, **Insertion Sort**, **Selection Sort**, and **Merge Sort** using C programming.

2. Learning Outcomes

After successful completion of this experiment, students will be able to:

- Understand different comparison-based sorting algorithms.
- Implement and test sorting algorithms using C.
- Analyze the time and space complexity of sorting techniques.
- Identify the appropriate sorting algorithm for a given problem based on input size and performance.

3. Prerequisites

- Basic understanding of arrays and functions in C.
- Concepts of loops, recursion, and time complexity.
- Familiarity with the concept of divide-and-conquer (especially for quick and merge sort).

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (Turbo C, GCC, Code::Blocks, or VS Code)
- Lab manual/notebook
- Input datasets for testing
- IDE or text editor

5. Introduction and Theory

Sorting is the process of arranging data in a particular order (ascending or descending). There are many sorting techniques; each differs in complexity and use cases.

Common Sorting Algorithms:

Algorithm	Type	Best Case	Worst Case	Space	Stable
Bubble Sort	Comparison-based	O(n)	$O(n^2)$	O(1)	Yes
Insertion Sort	Comparison-based	O(n)	$O(n^2)$	O(1)	Yes
Selection Sort	Comparison-based	$O(n^2)$	$O(n^2)$	O(1)	No
Merge Sort	Divide & Conquer	O(n log n)	O(n log n)	O(n)	Yes
Quick Sort	Divide & Conquer	O(n log n)	$O(n^2)$	O(log n)	No

6. Operating Procedure

Quick Sort:

1. Choose a pivot.

2. Partition array such that elements < pivot go left, > pivot go right.
3. Recursively apply Quick Sort to left and right partitions.

Practice Set Implementation:

- **Bubble Sort**: Repeatedly swap adjacent elements if they are in wrong order.
- **Insertion Sort**: Insert each element into the correct position of sorted part.
- **Selection Sort**: Select minimum from unsorted and place it in sorted portion.
- **Merge Sort**: Divide array in halves, recursively sort and merge them.

Quick Sort

```
#include <stdio.h>

void swap(int a[], int i, int j) {
   int t = a[i];
   a[i] = a[j];
   a[j] = t;
}

int partition(int a[], int low, int high) {
   int pivot = a[high], i = low - 1;
   for (int j = low; j < high; j++)
      if (a[j] < pivot) {
         i++;
         swap(a, i, j);
      }
   swap(a, i + 1, high);
   return i + 1;
}

void quickSort(int a[], int low, int high) {
   if (low < high) {
      int pi = partition(a, low, high);
      quickSort(a, low, pi - 1);
      quickSort(a, pi + 1, high);
   }
}

int main() {
   int a[6] = {9, 4, 7, 2, 6, 3}, n = 6;
   quickSort(a, 0, n - 1);
   for (int i = 0; i < n; i++) printf("%d ", a[i]);
   return 0;
}
```

7. Precautions and/or Troubleshooting

- Carefully handle recursive base conditions.
- Check array boundaries in partitioning (Quick & Merge).

- Ensure array is not accessed out of bounds.
- Debug swaps or pointer errors during partition and merge.

8. Observations

Test Case	Input Array	Algorithm	Sorted Output	Time Taken
TC1	[10, 5, 2, 8, 7]	Quick Sort	[2, 5, 7, 8, 10]	XX ms
TC2	[4, 3, 2, 1]	Bubble Sort	[1, 2, 3, 4]	XX ms
TC3	[6, 2, 8, 3]	Merge Sort	[2, 3, 6, 8]	XX ms

9. Calculations & Analysis

Time Complexity Comparison

Algorithm	Best	Average	Worst
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)

Note: Quick Sort is preferred for large datasets, despite its worst-case behavior.

10. Result & Interpretation

All sorting algorithms were implemented and tested successfully. Quick Sort outperformed others for larger arrays, while simpler algorithms like Insertion Sort performed well on nearly sorted or small datasets.

11. Follow-up Questions

1. Why is Quick Sort considered efficient despite its worst-case time complexity?
2. What makes Merge Sort stable but not in-place?
3. Can Quick Sort be made stable? How?
4. What is the role of recursion in Merge and Quick Sort?
5. Which sort is best when input is almost sorted?

12. Extension and Follow-up Activities

- Implement **Heap Sort**.
- Compare the runtime performance on large datasets.
- Visualize sorting steps using graphical tools.
- Implement sorting of structures (e.g., students by marks).

13. Assessments

- Write code to implement Quick Sort and analyze its performance on random vs sorted data.
- Identify output of intermediate steps in Bubble Sort.
- MCQs on complexity, stability, and in-place nature of algorithms.

14. Suggested Readings

- "Data Structures Using C" – Reema Thareja
- "Introduction to Algorithms" – Cormen et al.
- GeeksforGeeks – Sorting Algorithms
- TutorialsPoint – Data Structure Sorting

15. List of Experiments

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort *(Extension)*
7. Counting Sort *(Non-comparison based)*
8. Comparison of sorting algorithms using graphs

Experiment No 13: Binary Search Techniques

1. Aim / Purpose of the Experiment

To implement the **Binary Search algorithm** and understand its functionality for locating elements efficiently in a sorted array.

2. Learning Outcomes

After completing this experiment, students will be able to:

- Understand and implement binary search using iterative and recursive methods.
- Analyze the time and space complexity of binary search.
- Compare binary search with linear search.
- Identify scenarios where binary search is applicable.

3. Prerequisites

- Knowledge of arrays and sorting.
- Basics of algorithm design and C programming.
- Understanding of control structures (loops, conditions, functions).

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C compiler (GCC, Turbo C, etc.)
- Text/code editor (VS Code, Code::Blocks)
- Lab notebook/manual

5. Introduction and Theory

Binary Search is a fast search algorithm that works on a **sorted array** by repeatedly dividing the search interval in half.

Characteristics:

- Works only on **sorted** data.
- Divides the array and eliminates half each time.

- **Time complexity**: O(log n)

Algorithm Steps:

1. Set low = 0, high = n-1.
2. Repeat while low <= high:
    - mid = (low + high) / 2
    - If arr[mid] == key: Element found.
    - If arr[mid] < key: Search right half.
    - If arr[mid] > key: Search left half.

6. Operating Procedure

Step-by-step Instructions:

1. Input the sorted array and the element to be searched.
2. Choose either iterative or recursive binary search.
3. Implement the algorithm:
    - For iterative: use a loop.
    - For recursive: use function calls with updated low and high.
4. Display whether the element is found and its position.
5. Repeat for different cases: found/not found, first/last element, etc.

```
#include <stdio.h>

int binarySearch(int arr[], int n, int key) {
   int low = 0, high = n - 1;
   while (low <= high) {
      int mid = (low + high) / 2;
      if (arr[mid] == key)
         return mid;  // Key found at index mid
      else if (arr[mid] < key)
         low = mid + 1;  // Search in right half
      else
         high = mid - 1; // Search in left half
   }
   return -1; // Key not found
}

int main() {
   int arr[] = {2, 4, 6, 8, 10, 12, 14};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key = 10;
   int result = binarySearch(arr, n, key);
   if (result != -1)
      printf("Element found at index %d\n", result);
   else
      printf("Element not found\n");
```

```
    return 0;
}
```

7. Precautions and/or Troubleshooting

- Ensure the input array is sorted before performing binary search.
- Prevent integer overflow when calculating mid: mid = low + (high - low)/2.
- Handle edge cases like empty arrays or single-element arrays.
- Avoid infinite recursion in recursive implementation.

8. Observations

Test Case	Input Array	Search Element	Output
TC1	[10, 20, 30, 40, 50]	30	Found at index 2
TC2	[5, 10, 15, 20]	25	Not Found
TC3	[1]	1	Found at index 0
TC4	[]	10	Array is empty

9. Calculations & Analysis

- **Worst-case time complexity**: O(log n)
- **Best-case**: O(1) (when the element is at mid)
- **Space complexity**:
  - Iterative: O(1)
  - Recursive: O(log n) due to call stack

10. Result & Interpretation

The binary search algorithm was successfully implemented using both iterative and recursive approaches. The results confirm that binary search is efficient for sorted arrays and significantly reduces the number of comparisons compared to linear search.

11. Follow-up Questions

1. Why does binary search require a sorted array?
2. What is the difference between recursive and iterative binary search?
3. What will happen if the array is not sorted?
4. How does binary search compare with linear search in time complexity?

12. Extension and Follow-up Activities (if applicable)

- Implement binary search on **strings** or **floating-point numbers**.
- Modify the algorithm to return the **first or last occurrence** of duplicates.
- Use binary search on a **rotated sorted array**.
- Implement exponential search using binary search as a helper.

13. Assessments

- MCQs on time complexity and steps of binary search.
- Coding: Implement binary search recursively and find the number of comparisons.

- Trace a binary search step-by-step for a given input.

14. Suggested Readings

- "Data Structures Using C" – Reema Thareja
- "Introduction to Algorithms" – Cormen et al.
- GeeksforGeeks – Binary Search
- NPTEL Data Structures Lectures

15. List of Related Experiments

1. Linear Search Implementation
2. Binary Search (Iterative & Recursive)
3. Sorting Algorithms (Bubble, Insertion, Quick)
4. Search in Rotated Sorted Array
5. Search in Linked List
6. String Search Algorithms (e.g., KMP)

Experiment No 14: Implementation of Graph Traversals –BFS and DFS

1. Aim / Purpose of the Experiment

To understand and implement Graph Traversals using Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms on a given graph structure.

2. Learning Outcomes

After completing this experiment, students will be able to:

- Represent graphs using adjacency matrix and adjacency list.
- Implement BFS and DFS traversal algorithms.
- Differentiate between BFS and DFS based on their traversal mechanism.
- Apply graph traversal concepts in real-world scenarios (e.g., pathfinding, web crawling).
- Analyze the time and space complexities of BFS and DFS.

3. Prerequisites

- Basic understanding of graph theory and its terminologies.
- Proficiency in C programming or any equivalent language.
- Knowledge of stack, queue, and recursion.
- Understanding of arrays, pointers, and dynamic memory allocation.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer System
- C Compiler (e.g., GCC)
- Code Editor (e.g., Code::Blocks, VS Code, Turbo C++)
- Pen, Notebook for writing observations
- Lab Manual for experiment record

5. Introduction and Theory

A graph is a data structure consisting of a set of nodes (vertices) and a set of edges connecting pairs of nodes.

- Breadth-First Search (BFS): Traverses the graph level by level using a queue. Useful for finding the shortest path in unweighted graphs.
- Depth-First Search (DFS): Traverses the graph by exploring as far as possible along each branch before backtracking. Uses recursion or a stack.

Applications:

- BFS: Web crawlers, GPS navigation, social networks
- DFS: Solving puzzles, Topological sorting, Pathfinding in AI

6. Operating Procedure

1. Define the graph using adjacency matrix or list.
2. Input the number of vertices and edges.
3. For BFS:
    - Use a queue to store nodes.
    - Use a visited array to avoid re-visiting nodes.
4. For DFS:
    - Use a recursive function or stack.
    - Mark nodes as visited during traversal.
5. Display the order of traversal.
6. Test both traversals on different graph inputs.

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

#define MAX 100

int queue[MAX], front = 0, rear = 0, visited[MAX];
int graph[MAX][MAX], n;

void enqueue(int v) { queue[rear++] = v; }
int dequeue() { return queue[front++]; }
int empty() { return front == rear; }

void bfs(int start) {
   enqueue(start);
   visited[start] = 1;
   while (!empty()) {
      int u = dequeue();
      printf("%d ", u);
      for (int v = 0; v < n; v++) {
         if (graph[u][v] && !visited[v]) {
            enqueue(v);
            visited[v] = 1;
```

```
        }
      }
    }
}

int main() {
    n = 5;
    int edges[][2] = {{0,1},{0,2},{1,2},{2,3},{3,4}};
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            graph[i][j] = 0;
    for (int i=0; i<5; i++)
        graph[edges[i][0]][edges[i][1]] = graph[edges[i][1]][edges[i][0]] = 1;

    for (int i=0; i<n; i++) visited[i] = 0;

    printf("BFS starting from node 0:\n");
    bfs(0);

    return 0;
}
```

7. Precautions and/or Troubleshooting

- Ensure no node is visited more than once.
- Handle disconnected graphs separately.
- Validate input data (e.g., no invalid vertex indices).
- Initialize the visited array before each traversal.

8. Observations

Graph Input	BFS Output	DFS Output
0-1-2-3	0 1 2 3	0 1 2 3
0→1, 0→2	0 1 2	0 2 1
Disconnected	Separate traversal	Separate traversal

9. Calculations & Analysis

- Time Complexity:
  - BFS: O(V + E)
  - DFS: O(V + E)
  - V = number of vertices, E = number of edges
- Space Complexity:
  - O(V) for visited array, queue/stack

10. Result & Interpretation

Both BFS and DFS were successfully implemented. BFS explores the graph level-wise using a queue, while DFS explores as deep as possible using recursion/stack. Each traversal is suitable for different types of graph problems.

11. Follow-up Questions

1. How does BFS differ from DFS in terms of space usage?
2. Can DFS be implemented without recursion?
3. What is the impact of using BFS in finding the shortest path?
4. How do BFS and DFS behave in cyclic and disconnected graphs?

12. Extension and Follow-up Activities (if applicable)

- Modify code to handle weighted graphs using Dijkstra's algorithm.
- Extend traversal to detect cycles or connected components.
- Visualize traversal using graph-drawing tools.
- Apply traversal to real-world applications like maze solvers.

13. Assessments

- Code implementation of BFS and DFS using both adjacency matrix and list.
- Oral quiz on differences and use-cases of BFS and DFS.
- Written test on time/space complexity analysis.
- Real-world application questions.

14. Suggested Readings

- "Fundamentals of Data Structures in C" by Horowitz, Sahni
- "Data Structures Using C" by Reema Thareja
- GeeksforGeeks: BFS and DFS Tutorials
- NPTEL Video Lectures on Graph Theory

15. List of Experiments

1. Representing a graph using adjacency matrix and list.
2. Implementing BFS traversal on a graph.
3. Implementing DFS traversal using recursion.
4. Implementing DFS traversal using a stack.
5. Finding connected components using BFS/DFS.
6. Detecting cycle in a graph using DFS.
7. Application of BFS in shortest path algorithms.
8. Comparative analysis of BFS and DFS traversal orders.

Experiment No 15: Implementation of Hash tables for searching & sorting techniques.

1. Aim / Purpose of the Experiment

To implement a hash table data structure for efficient searching and sorting of data using appropriate collision resolution techniques and hashing strategies.

2. Learning Outcomes

After completion of this lab experiment, students will be able to:

- Understand the concept and utility of hash tables.
- Implement searching using hash tables.
- Integrate sorting techniques for maintaining and retrieving ordered data.
- Analyze performance and apply appropriate collision resolution methods.

3. Prerequisites

- Understanding of arrays, linked lists.
- Basic knowledge of searching and sorting algorithms (e.g., binary search, quicksort).
- Familiarity with hash functions and collision resolution techniques.

4. Materials / Equipment / Apparatus / Devices / Software Required

- Computer with C/C++ compiler (Turbo C++, GCC, Code::Blocks).
- Text editor or IDE (e.g., VS Code, Dev C++).
- Lab record book and pen.

5. Introduction and Theory

A Hash Table is a key-value based data structure used for efficient search and retrieval operations. Hashing transforms a key into an array index using a hash function. It enables average-case O(1) time complexity for search, insert, and delete operations. Searching in hash tables is performed by hashing the key and probing the corresponding index. Sorting, while not inherent to hashing, may be applied to hash table values or keys if needed (e.g., display sorted keys).

Collision Handling Methods:

- Chaining: Uses linked lists to handle collisions at the same index.
- Open Addressing: Searches for alternate empty slots using linear or quadratic probing.

6. Operating Procedure

1. Initialize a hash table (array of fixed size).
2. Choose a hash function (e.g., key % size).
3. Implement Insert: Compute index using hash function and insert value.
4. Implement Search: Use the hash index to find if the value exists.
5. Implement Delete (optional): Remove a key by marking it deleted.
6. Sorting:
    - Extract keys/values from the hash table.
    - Apply a sorting algorithm (like Bubble Sort, Quick Sort).
7. Test with sample data.

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

#define SIZE 10
```

```
struct Node {
   int data;
   struct Node* next;
};

struct Node* hashTable[SIZE];

int hash(int key) {
   return key % SIZE;
}

void insert(int key) {
   int i = hash(key);
   struct Node* newNode = malloc(sizeof(struct Node));
   newNode->data = key;
   newNode->next = hashTable[i];
   hashTable[i] = newNode;
}

int search(int key) {
   int i = hash(key);
   struct Node* temp = hashTable[i];
   while (temp) {
      if (temp->data == key) return 1;
      temp = temp->next;
   }
   return 0;
}

void collectElements(int arr[], int* n) {
   *n = 0;
   for (int i = 0; i < SIZE; i++) {
      struct Node* temp = hashTable[i];
      while (temp) {
         arr[(*n)++] = temp->data;
         temp = temp->next;
      }
   }
}

void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++)
      for (int j = 0; j < n-i-1; j++)
         if (arr[j] > arr[j+1]) {
            int t = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = t;
         }
```

```
}

int main() {
   int keys[] = {23, 43, 13, 27, 17, 33};
   int n = sizeof(keys) / sizeof(keys[0]);
   for (int i = 0; i < n; i++) insert(keys[i]);

   printf("Search 17: %s\n", search(17) ? "Found" : "Not Found");

   int all[100], count;
   collectElements(all, &count);
   bubbleSort(all, count);

   printf("Sorted Elements: ");
   for (int i = 0; i < count; i++)
      printf("%d ", all[i]);
   return 0;
}
```

7. Precautions and/or Troubleshooting

- Ensure proper handling of collisions.
- Avoid infinite loops in open addressing by tracking probing limits.
- Do not use complex hash functions for small datasets.
- Always validate array bounds while accessing hash indexes.

8. Observations

Operation	Key	Hash Index	Status	Remarks
Insert	10	0	Inserted	No collision
Insert	20	0	Chained	Collision handled
Search	10	0	Found	Successful
Sort Keys	-	-	Sorted: 10, 20	Bubble sort applied

9. Calculations & Analysis

- Hash Function Used: key % table_size
- Collision Technique: Chaining / Linear Probing
- Search Complexity:
  - Average Case: O(1)
  - Worst Case: O(n) (if clustering occurs)
- Sorting Complexity:
  - Depends on sorting algorithm used (e.g., $O(n^2)$ for Bubble Sort)

---

10. Result & Interpretation

The hash table was successfully implemented for search and sort operations. Collision handling via chaining/open addressing ensured correct insertion. Sorting applied on keys/values provided ordered output for verification.

11. Follow-up Questions

1. How does load factor affect performance in hash tables?
2. Can you sort a hash table directly? Why or why not?
3. How would you choose between chaining and open addressing?
4. What changes are needed for dynamic hash tables?

12. Extension and Follow-up Activities (if applicable)

- Implement rehashing to dynamically increase table size.
- Compare hashing with binary search trees for search efficiency.
- Implement double hashing as an advanced collision resolution method.
- Visualize hash table operations step-by-step using animations.

13. Assessments

- Implement and explain hash table with both chaining and probing.
- Given a data set, manually compute hash indices and simulate insertion.
- Apply sorting on hash values and explain the algorithm used.
- Write pseudo code for hash-based searching.

14. Suggested Readings

- *Data Structures Using C* by Reema Thareja
- *Fundamentals of Data Structures in C* by Horowitz and Sahni
- GeeksforGeeks – Hashing Techniques
- NPTEL Lectures on Hash Tables and Searching Algorithms

15. List of Experiments

1. Hash Table Insertion and Search (Chaining)
2. Hash Table with Open Addressing (Linear Probing)
3. Sorting Values Extracted from Hash Table
4. Comparison of Hashing vs Binary Search
5. Collision Resolution Techniques: Performance Study
6. Application: Symbol Table using Hashing