G H Raisoni College of Engineering & Management, Pune
NAAC Accredited A+ Grade
(An Empowered Autonomous Institute Affiliated to Savitribai Phule Pune University)
Department of CSE Artificial Intelligence

Course File (2024-25)

Class: B-Tech Second Year Sem: III

Data Structures & Algorithms

(23UAIPCL2301)

Lab Manual

Faculty Name:

Dr.Devidas S. Thosar

Data Structure & Algorithms Lab
Practical List
(Session 2024-25)

Name of Subject: DATA STRUCTURE & ALGORITHM (23UAIPCL2301)		Semester: III
COURSE OUTCOME		
CO1	Describe the fundamental principles of algorithms and their performance metrics.	
CO2	Demonstrate the use of linear and non-linear data structures in practical applications.	
CO3	Utilize efficient searching and sorting techniques in problem-solving scenarios.	
CO4	Understand hashing and file organization techniques for effective data management.	

List of Experiment

Sr.No.	Experiment List	Unit No.	Mapping With CO
1	Write a program to find the largest number in an array of integers. Analyze its time and space complexity.	1	CO1
2	Implement a stack data structure using arrays.	2	CO2
3	Implement a stack data structure using linked list.	4	CO2
4	Implement a linear queue data structure using arrays.	2	CO2
5	Implement a linear queue data structure using linked list.	4	CO2
6	Write a program to search for an element in an array using linear search and binary search. Compare their efficiencies.	3	CO3
7	Write a program to sort the elements using Bubble Sort.	3	CO3
8	Write a program to sort the elements using Insertion Sort.	3	CO3
9	Write a program to sort the elements using Selection Sort.	3	CO3
10	Implement Merge Sort and Quick sort to sort the given elements. Compare their time complexities.	3	CO3
11	Write a program to implement Circular Linked List.	4	CO2
12	Implement a binary search tree (BST) to perform insertion, deletion, and search operations.	4	CO2
13	Implement graph traversal algorithms BFS and DFS.	4	CO2
14	Implement a hash table using linear probing and quadratic probing techniques.	5	CO4
15	Develop a program to manage student records	5	CO4

	using different file organization methods, including sequential, indexed, and direct file organization.		
16	Write programs to demonstrate the construction of minimum spanning trees using Prim's and Kruskal's algorithms and find the shortest path using Dijkstra's algorithm.	5	CO2
17	Revision	5	CO3

(Dr.Devidas S. Thosar)
Lab In – charge

(Mrs. Rachna Sable)
HOD

Experiment no. 1

Title: Write a program to find the largest number in an array of integers. Analyze its time and space complexity.

Theory:
1.Include time.h Library: This library provides the clock() function and the clock_t type.
2.Variables for Timing: start and end are variables of type clock_t to store the time before and after the function execution. cpu_time_used will store the calculated execution time.
3.Measure Time:
start = clock(); captures the clock ticks before the function execution.
end = clock(); captures the clock ticks after the function execution.
4.Calculate Execution Time: The execution time is calculated by subtracting start from end and then dividing by CLOCKS_PER_SEC to convert clock ticks to seconds.
5.Print Execution Time: The execution time is printed in seconds.

Time Complexity
The time complexity of this program is ð‘,(ð‘›)O(n), where n is the number of elements in the array. This is because the program goes through each element in the array exactly once to find the largest number.

Space Complexity
The space complexity of this program is O(1). This is because the amount of extra space used by the program does not depend on the size of the input array. The only additional space used is for a few variables (largest and num), which do not scale with the input size.

Algorithm
1.Initialize: Start by assuming the first element of the array is the largest. Store this value in a variable called largest.
2.Iterate: Loop through each element in the array.
3.Compare: For each element, compare it with the current largest. If the element is greater than largest, update largest to this element.
Return: After completing the loop, the largest variable will hold the largest number in the array.

Pseudocode

```
function find_largest(arr):
    if arr is empty:
        raise ValueError("Array is empty")

    largest = arr[0]
    for each element in arr:
        if element > largest:
            largest = element

    return largest
```

Example

Consider an example array:

[3,5,7,2,8,â^’1,4,10,12].

Initialize largest with the first element, 3.
Iterate through the array:
Compare 3 with 5: update largest to 5.
Compare 5 with 7: update largest to 7.
Compare 7 with 2: largest remains 7.
Compare 7 with 8: update largest to 8.
Compare 8 with -1: largest remains 8.
Compare 8 with 4: largest remains 8.
Compare 8 with 10: update largest to 10.
Compare 10 with 12: update largest to 12.
Return 12 as the largest number in the array.

Applications

1. Data Analysis and Statistics
2. Computer Science and Software Development
3. Financial Services
4. Engineering and Scientific Research
5. Competitive Programming
6. Everyday Applications-Grades and Scores and E-commerce

Conclusion

Finding the largest number in an array is a basic yet crucial operation with applications ranging from data analysis and algorithm optimization to financial services and everyday decision-making. It is a building block for more complex algorithms and is widely used across various domains to extract meaningful insights from data.

Code

```
#include <stdio.h>
#include <limits.h>
#include <time.h>

int find_largest(int arr[], int n) {
   if (n <= 0) {
      printf("Array is empty\n");
      return INT_MIN;
   }

   int largest = arr[0];
   for (int i = 1; i < n; i++) {
      if (arr[i] > largest) {
         largest = arr[i];
      }
   }
   return largest;
}
```

```
int main() {
    int array[] = {3, 5, 7, 2, 8, -1, 4, 10, 12};
    int n = sizeof(array) / sizeof(array[0]);

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    int largest = find_largest(array, n);
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    if (largest != INT_MIN) {
        printf("The largest number is: %d\n", largest);
    }
    printf("Execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

Output

```
The largest number is: 12
Execution time: 0.000001 seconds
```

Experiment no. 2

Title: Implement a stack data structure using arrays.

Theory:
Array-An array is a data structure that stores a fixed-size sequential collection of elements of the same type. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
Key Characteristics:
1.Fixed Size: The size of an array is defined when it is created and cannot be changed.
2.Same Data Type: All elements in an array must be of the same data type.
3.Indexed: Each element in the array can be accessed using its index, with the first element at index 0.

Stack-A stack is a linear data structure that follows the Last In First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed.
Stack Operations
1.Push: Add an element to the top of the stack.
2.Pop: Remove the top element from the stack.
3.Peek/Top: Return the top element without removing it.
4.isEmpty: Check if the stack is empty.
5.isFull: Check if the stack is full.

a)Stack.h: This is the header file where we define the stack structure and declare the stack functions.

1.MAX: Defines the maximum size of the stack.
2.Stack: A struct to represent the stack. It contains an array items to store stack elements and an integer top to keep track of the top element.

b)Stack.c: This is the implementation file where we define the stack functions.

1.initStack(Stack* stack): Initializes the stack by setting top to -1.
2.isFull(Stack* stack): Checks if the stack is full.
3.isEmpty(Stack* stack): Checks if the stack is empty.
4.push(Stack* stack, int item): Adds an element to the top of the stack.
5.pop(Stack* stack): Removes and returns the top element of the stack.
6.peek(Stack* stack): Returns the top element without removing it.

c)Main.c: This is the main program where we test the stack implementation.

We initialize the stack, push elements onto the stack, pop an element, and peek at the top element.

Algorithm
Initialize Stack:
Set top to -1.

isFull:
Return true if top equals MAX - 1, else false.

isEmpty:
Return true if top equals -1, else false.

Push:
If isFull, print "Stack overflow".
Else, increment top and set stack[top] to the item.

Pop:
If isEmpty, print "Stack underflow" and return INT_MIN.
Else, return stack[top] and decrement top.

Peek:
If isEmpty, print "Stack is empty" and return INT_MIN.
Else, return stack[top].

Main Program:
Initialize the stack.
Perform push operations.
Perform pop operation and print the popped value.
Perform peek operation and print the top value.

Pseudocode

```
// Initialize Stack
Procedure initStack(stack)
    stack.top = -1

// Check if stack is full
Function isFull(stack)
    return stack.top == MAX - 1

// Check if stack is empty
Function isEmpty(stack)
    return stack.top == -1

// Push operation
Procedure push(stack, item)
    if isFull(stack)
        Print "Stack overflow"
    else
        stack.top = stack.top + 1
        stack.items[stack.top] = item

// Pop operation
Function pop(stack)
    if isEmpty(stack)
        Print "Stack underflow"
        return INT_MIN
    else
        item = stack.items[stack.top]
        stack.top = stack.top - 1
```

```
        return item

// Peek operation
Function peek(stack)
    if isEmpty(stack)
        Print "Stack is empty"
        return INT_MIN
    else
        return stack.items[stack.top]

// Main Program
Procedure main
    Declare stack of type Stack
    Call initStack(stack)

    Call push(stack, 10)
    Call push(stack, 20)
    Call push(stack, 30)

    item = Call pop(stack)
    Print item, " popped from stack"

    topItem = Call peek(stack)
    Print "Top element is ", topItem
```

Examples

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

#define MAX 3

typedef struct {
    int items[MAX];
    int top;
} Stack;

void initStack(Stack* stack) {
    stack->top = -1;
}

int isFull(Stack* stack) {
    return stack->top == MAX - 1;
}

int isEmpty(Stack* stack) {
    return stack->top == -1;
}

void push(Stack* stack, int item) {
    if (isFull(stack)) {
        printf("Stack overflow\n");
```

```
    } else {
        stack->top++;
        stack->items[stack->top] = item;
        printf("%d pushed to stack\n", item);
    }
}

int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow\n");
        return INT_MIN;
    } else {
        int item = stack->items[stack->top];
        stack->top--;
        return item;
    }
}

int peek(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return INT_MIN;
    } else {
        return stack->items[stack->top];
    }
}

int main() {
    Stack stack;
    initStack(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    push(&stack, 40); // This will cause stack overflow

    printf("Top element is %d\n", peek(&stack));

    pop(&stack);
    pop(&stack);
    pop(&stack);
    pop(&stack); // This will cause stack underflow

    return 0;
}
```

Output

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
Stack overflow
```

Top element is 30
30 popped from stack
20 popped from stack
10 popped from stack
Stack underflow

Applications
1.Expression Evaluation
2.Function Call Management
3.Backtracking
4.Memory Management
5.Undo Mechanisms
6.Parsing
7.History Management
8.Expression Matching

Conclusion
Array: A fixed-size sequential collection of elements of the same type, accessed by index.
Stack: A linear data structure following LIFO principle, with operations to push, pop, and peek elements, often implemented using arrays.

Experiment no. 3

Title: Implement a stack data structure using linked list.

Theory:
A linked list is a linear data structure where elements are stored in nodes. Unlike arrays, linked list elements are not stored in contiguous memory locations; instead, each node points to the next node in the sequence, forming a chain-like structure.

A stack implemented using a linked list operates on the Last In First Out (LIFO) principle. Elements are added and removed from only one end, which is the top of the stack. The operations typically include:
1.Push: Adds an element to the top of the stack.
2.Pop: Removes the element from the top of the stack.
3.Peek: Returns the element on the top of the stack without removing it.
4.isEmpty: Checks if the stack is empty.

Algorithm
1.Define Node Structure: Each node will contain two parts:

data: Holds the value of the element.
next: Pointer to the next node in the stack.
Initialize Stack: Create an empty stack by initializing a top pointer to NULL.

2.isEmpty Function:

Check if top is NULL.
Return true if stack is empty, false otherwise.
Push Operation:

3.Create a new node with the given data.
Set the next pointer of the new node to point to the current top.
Update top to point to the new node.
Pop Operation:

4.Check if stack is empty using isEmpty.
If not empty, save the data of the top node.
Move top to the next node.
Free the memory of the popped node.
Return the saved data.
Peek Operation:

5.Check if stack is empty using isEmpty.
Return the data of the top node without removing it.

Pseudocode-

```
// Define Node Structure
struct Node {
    int data;
    struct Node* next;
```

```
};

// Initialize Stack
Function initStack(stack)
    stack.top = NULL

// Check if stack is empty
Function isEmpty(stack)
    if stack.top == NULL
        return true
    else
        return false

// Push operation
Function push(stack, data)
    newNode = createNode(data)
    newNode.next = stack.top
    stack.top = newNode

// Pop operation
Function pop(stack)
    if isEmpty(stack)
        Print "Stack underflow"
        return INT_MIN
    else
        temp = stack.top
        poppedData = temp.data
        stack.top = stack.top.next
        free(temp)
        return poppedData

// Peek operation
Function peek(stack)
    if isEmpty(stack)
        Print "Stack is empty"
        return INT_MIN
    else
        return stack.top.data

// Main function to demonstrate stack operations
Function main()
    Declare stack of type Stack
    initStack(stack)

    push(stack, 10)
    push(stack, 20)
    push(stack, 30)

    Print "Top element is ", peek(stack)

    Print pop(stack), " popped from stack"
```

```
    Print "Top element is ", peek(stack)

    Print pop(stack), " popped from stack"
    Print pop(stack), " popped from stack"

    Print "Is the stack empty? ", isEmpty(stack) ? "Yes" : "No"
```

Example

```
#include <stdio.h>
#include <stdlib.h> // For malloc and free

// Define a structure for a stack node
typedef struct StackNode {
    int data;
    struct StackNode* next;
} StackNode;

// Define a stack structure
typedef struct {
    StackNode* top;
} Stack;

// Function to create a new node
StackNode* createNode(int data) {
    StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize stack
void initStack(Stack* stack) {
    stack->top = NULL;
}

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

// Function to push an element onto the stack
void push(Stack* stack, int data) {
    StackNode* newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
    printf("%d pushed to stack\n", data);
```

```
}

// Function to pop an element from the stack
int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow\n");
        return INT_MIN;
    }
    StackNode* temp = stack->top;
    int poppedData = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return poppedData;
}

// Function to peek the top element of the stack
int peek(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return INT_MIN;
    }
    return stack->top->data;
}

// Main function to demonstrate the stack operations
int main() {
    Stack stack;
    initStack(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Top element
```

Application

1.Expression Evaluation
2.Function Call Stack
3.Undo/Redo Functionality
4.Memory Management
5.Depth-First Search (DFS) in Graph Algorithms

Conclusion

Using a stack implemented with a linked list provides flexibility and efficient memory management for applications requiring last-in, first-out (LIFO) operations. Whether for managing function calls, evaluating expressions, or implementing undo functionality, linked list-based stacks are versatile and widely applicable in computer science and software development.

Experiment no. 4

Title: Implement a linear queue data structure using arrays.

Theory:
A linear queue, often simply referred to as a "queue," is a fundamental data structure in computer science that follows the First-In-First-Out (FIFO) principle. It operates much like a queue in everyday life, where the first person to join the queue is the first to be served or processed.

1.Struct Definition:
Queue: A structure containing an array items to store queue elements, and two integers front and rear to keep track of the front and rear indices of the queue, respectively.

2.Initialization (initQueue):
Initializes front and rear indices to -1 indicating an empty queue.

3.isEmpty (isEmpty):
Checks if both front and rear are -1, indicating the queue is empty.

4.isFull (isFull):
Checks if rear is equal to MAX_SIZE - 1, indicating the queue is full (in this fixed-size implementation).

5.enqueue (enqueue):
Adds an element to the queue by incrementing rear and storing the new element at rear.

6.dequeue (dequeue):
Removes and returns the element at front by incrementing front after retrieval. If the queue becomes empty after removal, it resets front and rear to -1.

7.front (front):
Retrieves the element at front without removing it.

8.display (display):
Prints all elements in the queue from front to rear.

9.Main Function:
Demonstrates usage of the queue with sample operations (enqueue, dequeue, front, and display).

Algorithm for Linear Queue using Arrays
1.Initialization
Initialize a queue structure (Queue) with an array (items) and two integers (front and rear).
Set front and rear to -1 to indicate an empty queue.

2.isEmpty
Check if both front and rear are -1.
If true, return true (queue is empty).
Otherwise, return false.

3.isFull
Check if rear is equal to MAX_SIZE - 1, where MAX_SIZE is the maximum capacity of the queue.
If true, return true (queue is full).
Otherwise, return false.

4.Enqueue (Add Element)
Check if the queue is full using isFull.
If true, display an error message and exit.
If the queue is empty (front and rear are -1), set front to 0.
Increment rear.
Add the new element at rear in the items array.

5.Dequeue (Remove Element)
Check if the queue is empty using isEmpty.
If true, display an error message and exit.
Retrieve the element at front.
If front equals rear (only one element in the queue), reset front and rear to -1.
Otherwise, increment front to remove the element.

6.Front (Retrieve Front Element)
Check if the queue is empty using isEmpty.
If true, display an error message and exit.
Return the element at front in the items array.

7.Display Queue
Check if the queue is empty using isEmpty.
If true, display a message indicating the queue is empty.
Otherwise, iterate through the items array from front to rear and print each element.

Pseudocode

```
// Structure definition for Queue
Queue:
    items[MAX_SIZE]  // Array to hold queue elements
    front          // Index of the front element
    rear           // Index of the rear element

// Function to initialize the queue
function initQueue(q: Queue):
    q.front <- -1
    q.rear <- -1

// Function to check if the queue is empty
function isEmpty(q: Queue) -> boolean:
    return (q.front == -1 and q.rear == -1)

// Function to check if the queue is full
function isFull(q: Queue) -> boolean:
    return (q.rear == MAX_SIZE - 1)
```

```
// Function to add an element to the queue (enqueue operation)
function enqueue(q: Queue, value: integer):
    if isFull(q):
        print("Queue is full. Cannot enqueue.")
        return
    if isEmpty(q):
        q.front <- 0  // Initialize front if it's the first element
    q.rear <- q.rear + 1  // Move rear to the next position
    q.items[q.rear] <- value  // Add the new element to the rear

// Function to remove an element from the queue (dequeue operation)
function dequeue(q: Queue) -> integer:
    if isEmpty(q):
        print("Queue is empty. Cannot dequeue.")
        return -1
    removedItem <- q.items[q.front]  // Get the element at the front
    if q.front == q.rear:
        // Reset queue to empty state if there was only one element
        q.front <- -1
        q.rear <- -1
    else:
        q.front <- q.front + 1  // Move front to the next position
    return removedItem

// Function to retrieve the element at the front of the queue without removing it
function front(q: Queue) -> integer:
    if isEmpty(q):
        print("Queue is empty. No front element.")
        return -1
    return q.items[q.front]

// Function to display the elements in the queue
function display(q: Queue):
    if isEmpty(q):
        print("Queue is empty. Nothing to display.")
        return
    print("Queue elements:")
    for i <- q.front to q.rear:
        print(q.items[i])

// Example usage:
function main():
    // Initialize a queue
    Queue q
    initQueue(q)

    // Enqueue elements
    enqueue(q, 10)
    enqueue(q, 20)
    enqueue(q, 30)
```

```
    // Display queue elements
    display(q)  // Output: Queue elements: 10 20 30

    // Dequeue an element
    dequeue(q)

    // Display updated queue
    display(q)  // Output: Queue elements: 20 30

// Call main function to run example
main()
```

Example

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

#define MAX_SIZE 100

// Structure for the Queue
typedef struct {
    int items[MAX_SIZE];
    int front;  // Index of the front element
    int rear;   // Index of the rear element
} Queue;

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

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

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

// Function to add an element to the queue (enqueue operation)
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;  // Initialize front if it's the first element
    }
```

```
    q->rear++;  // Move rear to the next position
    q->items[q->rear] = value;  // Add the new element to the rear
}

// Function to remove an element from the queue (dequeue operation)
int dequeue(Queue *q) {
    int removedItem;
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }
    removedItem = q->items[q->front];  // Get the element at the front
    if (q->front == q->rear) {
        // Reset queue to empty state if there was only one element
        q->front = -1;
        q->rear = -1;
    } else {
        q->front++;  // Move front to the next position
    }
    return removedItem;
}

// Function to retrieve the element at the front of the queue without removing it
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No front element.\n");
        return -1;
    }
    return q->items[q->front];
}

// Function to display the elements in the queue
void display(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Nothing to display.\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->items[i]);
    }
    printf("\n");
}

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

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

```
    display(&q);  // Output: Queue elements: 10 20 30

    printf("Front element: %d\n", front(&q));  // Output: Front element: 10

    dequeue(&q);  // Remove the front element

    display(&q);  // Output: Queue elements: 20 30

    return 0;
}
```

Application

1.Task Scheduling
2.Print Queue Management
3.Buffer Management in Networking
4.Browsing History
5.Message Queues

Conclusion

The linear queue data structure, implemented using arrays, serves as a fundamental tool in computer science and applications where managing data in a First-In-First-Out (FIFO) manner is essential. It offers efficient handling of tasks, processes, and data packets, ensuring that operations are performed in the order they were initiated. Whether in operating systems for task scheduling, networking for packet management, or web browsers for history tracking, the linear queue facilitates orderly and predictable data flow, making it indispensable in various computational and organizational contexts.

Experiment no. 5

Title: Implement a linear queue data structure using linked list.

Theory:
Implementing a linear queue data structure using a linked list in C involves using dynamically allocated memory to store elements and pointers to maintain the structure of the queue. Here's a detailed explanation including the theory, algorithm, and pseudocode for implementing a linear queue using a linked list.

In a linked list-based queue:
1.Each element (node) in the queue contains a data field and a pointer field (next) that points to the next node in the queue.
2.The queue maintains pointers (front and rear) to the front and rear nodes:
3.front points to the front (oldest) node where elements are dequeued.
4.rear points to the rear (newest) node where elements are enqueued.

Algorithm

1.Structure Definition: Define a structure Node to represent each element in the queue, containing a data field (data) and a pointer to the next node (next).
2.Initialization: Initialize front and rear pointers to NULL to indicate an empty queue.
3.isEmpty: Check if front is NULL to determine if the queue is empty.
4.Enqueue (Add Element):
  Allocate memory for a new node.
  Set the data field of the new node.
  Update next pointers to maintain the queue order:
  If rear is NULL, set both front and rear to the new node.
  Otherwise, update the next pointer of the current rear node to point to the new node, and update rear to point to the new node.
5.Dequeue (Remove Element):
  Check if the queue is empty.
  Retrieve data from the front node.
  Move front pointer to the next node.
  Free memory of the dequeued node.
6.Front (Retrieve Front Element): Return data from the front node without removing it.
7.Display: Traverse the queue from front to rear, printing each element's data field.

Pseudocode

```
// Structure definition for Node
Node:
    data   // Data field
    next   // Pointer to the next node

// Structure definition for Queue
Queue:
    front  // Pointer to the front (oldest) node
    rear   // Pointer to the rear (newest) node
```

```
// Function to check if the queue is empty
function isEmpty(q: Queue) -> boolean:
    return (q.front == NULL)

// Function to add an element to the queue (enqueue operation)
function enqueue(q: Queue, value: integer):
    // Create a new node
    newNode = allocate memory for Node
    newNode.data = value
    newNode.next = NULL

    // If queue is empty, set front and rear to the new node
    if isEmpty(q):
        q.front = newNode
        q.rear = newNode
    else:
        // Otherwise, add the new node at the end
        q.rear.next = newNode
        q.rear = newNode

// Function to remove an element from the queue (dequeue operation)
function dequeue(q: Queue) -> integer:
    // Check if queue is empty
    if isEmpty(q):
        print "Queue is empty. Cannot dequeue."
        return -1

    // Retrieve data from the front node
    removedItem = q.front.data

    // Move front to the next node
    temp = q.front
    q.front = q.front.next

    // Free memory of the dequeued node
    free(temp)

    return removedItem

// Function to retrieve the element at the front of the queue without removing it
function front(q: Queue) -> integer:
    // Check if queue is empty
    if isEmpty(q):
        print "Queue is empty. No front element."
        return -1

    // Return data from the front node
    return q.front.data

// Function to display the elements in the queue
function display(q: Queue):
```

```
    // Check if queue is empty
    if isEmpty(q):
        print "Queue is empty. Nothing to display."
        return

    // Traverse the queue and print each element's data
    currentNode = q.front
    while currentNode != NULL:
        print currentNode.data
        currentNode = currentNode.next

// Example usage:
function main():
    // Initialize an empty queue
    Queue q
    q.front = NULL
    q.rear = NULL

    // Enqueue elements
    enqueue(q, 10)
    enqueue(q, 20)
    enqueue(q, 30)

    // Display queue elements
    display(q)  // Output: 10, 20, 30

    // Dequeue an element
    dequeue(q)

    // Display updated queue
    display(q)  // Output: 20, 30

// Call main function to run example
main()
```

Example

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

// Define a structure for a Node
typedef struct Node {
    int data;          // Data stored in the node
    struct Node *next;  // Pointer to the next node
} Node;

// Define a structure for the Queue
typedef struct {
    Node *front;  // Pointer to the front (oldest) node
    Node *rear;   // Pointer to the rear (newest) node
} Queue;
```

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

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

// Function to add an element to the queue (enqueue operation)
void enqueue(Queue *q, int value) {
    // Create a new node
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    // If queue is empty, set both front and rear to the new node
    if (isEmpty(q)) {
        q->front = newNode;
        q->rear = newNode;
    } else {
        // Add the new node at the end of the queue
        q->rear->next = newNode;
        q->rear = newNode;
    }
}

// Function to remove an element from the queue (dequeue operation)
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    // Retrieve data from the front node
    Node *temp = q->front;
    int removedItem = temp->data;

    // Move front to the next node
    q->front = q->front->next;

    // Free memory of the dequeued node
    free(temp);
```

```
    // If front becomes NULL, set rear to NULL as well
    if (q->front == NULL) {
        q->rear = NULL;
    }

    return removedItem;
}

// Function to retrieve the element at the front of the queue without removing it
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No front element.\n");
        return -1;
    }
    return q->front->data;
}

// Function to display the elements in the queue
void display(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Nothing to display.\n");
        return;
    }

    printf("Queue elements: ");
    Node *current = q->front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Function to free memory allocated for the queue nodes
void freeQueue(Queue *q) {
    Node *current = q->front;
    Node *next;
    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
    q->front = NULL;
    q->rear = NULL;
}

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

    enqueue(&q, 10);
```

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

    display(&q);  // Output: Queue elements: 10 20 30

    printf("Dequeued item: %d\n", dequeue(&q));  // Output: Dequeued item: 10

    display(&q);  // Output: Queue elements: 20 30

    printf("Front item: %d\n", front(&q));  // Output: Front item: 20

    freeQueue(&q);  // Free memory allocated for the queue nodes

    return 0;
}
```

Applications

1.Breadth-First Search (BFS) in Graph Traversal
2.Job Scheduling in Operating Systems
3.Network Packet Management
4.Print Queue Management
5.Event Handling in Event-Driven Systems

Conclusion

The implementation of a linear queue using a linked list in C provides a flexible and efficient data structure for managing elements in a First-In-First-Out (FIFO) manner. This approach allows dynamic memory allocation, making it suitable for applications where the size of the queue may vary during runtime. Key operations such as enqueue, dequeue, checking if the queue is empty, retrieving the front element, and displaying the queue contents are efficiently handled. Overall, this implementation offers a foundational tool for organizing and processing data sequentially, essential in various computing domains such as operating systems, network packet management, and simulation systems.

Experiment no. 6

Title: Write a program to search for an element in an array using linear search and binary search. Compare their efficiencies.

Theory:

1.linearSearch Function: Performs a linear search through the array arr to find key. It increments comparisons for each comparison made. Returns the index of key if found, otherwise returns -1.

2.binarySearch Function: Performs a binary search through the sorted array arr to find key. It increments comparisons for each comparison made. Returns the index of key if found, otherwise returns -1.

3.Main Function:
Initializes an array arr and defines its size n.
Sets key to search for within the array.
Calls linearSearch and binarySearch functions to perform the searches.
Prints the results of both searches including whether the element was found, its index (if found), and the number of comparisons made.

Efficiency Comparison:
1.Linear Search:
Efficiency: O(n) in worst-case scenario.
Number of comparisons linearly increases with the size of the array.
2.Binary Search:
Efficiency: O(logn) in worst-case scenario (for sorted arrays).
Number of comparisons grows logarithmically with the size of the array.

Algorithm for Linear Search:
1.Input: Array arr, size n, element key to search.
2.Output: Index of key if found, otherwise -1.
3.Algorithm:
Initialize index to -1.
Iterate through each element of arr from index 0 to n-1.
Compare each element with key.
If a match is found, set index to current index and break out of loop.
After loop completion, return index.

Algorithm for Binary Search:
1.Input: Sorted array arr, indices left and right, element key to search.
2.Output: Index of key if found, otherwise -1.
3.Algorithm:
While left is less than or equal to right:
Calculate mid as (left + right) / 2.
If arr[mid] is equal to key, return mid.
If arr[mid] is less than key, set left to mid + 1.
Otherwise, set right to mid - 1.
If key is not found after loop completion, return -1.

Pseudocode for linear search

```
function linearSearch(arr, n, key) returns integer
   index = -1
   for i = 0 to n-1 do
      if arr[i] == key then
         index = i
         break
      end if
   end for
   return index
```

Pseudocode for binary search

```
function binarySearch(arr, left, right, key) returns integer
   while left <= right do
      mid = (left + right) / 2
      if arr[mid] == key then
         return mid
      else if arr[mid] < key then
         left = mid + 1
      else
         right = mid - 1
   end while
   return -1
```

Example

```
#include <stdio.h>

// Function for linear search
int linearSearch(int arr[], int n, int key, int *comparisons) {
   for (int i = 0; i < n; i++) {
      (*comparisons)++;
      if (arr[i] == key) {
         return i;  // Return index of key if found
      }
   }
   return -1;  // Return -1 if key not found
}

// Function for binary search (array must be sorted)
int binarySearch(int arr[], int left, int right, int key, int *comparisons) {
   while (left <= right) {
      (*comparisons)++;
      int mid = left + (right - left) / 2;

      // Check if key is present at mid
      if (arr[mid] == key) {
         return mid;  // Return index of key if found
```

```
        }

        // If key is greater, ignore left half
        if (arr[mid] < key) {
            left = mid + 1;
        }
        // If key is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }

    return -1;  // Return -1 if key not found
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 23;

    // Perform linear search
    int comparisons_linear = 0;
    int result_linear = linearSearch(arr, n, key, &comparisons_linear);

    // Perform binary search (array must be sorted)
    int comparisons_binary = 0;
    int result_binary = binarySearch(arr, 0, n - 1, key, &comparisons_binary);

    // Print results
    if (result_linear != -1) {
        printf("Linear Search:\n");
        printf("Element %d found at index %d\n", key, result_linear);
        printf("Number of comparisons: %d\n", comparisons_linear);
    } else {
        printf("Linear Search:\n");
        printf("Element %d not found\n", key);
        printf("Number of comparisons: %d\n", comparisons_linear);
    }
    printf("\n");

    if (result_binary != -1) {
        printf("Binary Search:\n");
        printf("Element %d found at index %d\n", key, result_binary);
        printf("Number of comparisons: %d\n", comparisons_binary);
    } else {
        printf("Binary Search:\n");
        printf("Element %d not found\n", key);
        printf("Number of comparisons: %d\n", comparisons_binary);
    }

    return 0;
```

}

Output

Linear Search:
Element 23 found at index 5
Number of comparisons: 6

Binary Search:
Element 23 found at index 5
Number of comparisons: 3

Applications
for Linear Search
1.Unsorted Lists or Arrays
2.Small Data Sets
3.Sequential Access in Files

For Binary search
1.Sorted Arrays
2.Binary Search Trees (BST)
3.Numerical Methods

Conclusion
linear search is straightforward and effective for searching in unsorted arrays or small datasets, while binary search excels in sorted arrays with its logarithmic time complexity. The choice between them depends on whether the data is sorted and the size of the dataset, optimizing for simplicity or efficiency accordingly.

Experiment no. 7

Title: Write a program to sort the elements using Bubble Sort.

Theory: Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. It is named because smaller elements "bubble" to the top of the list with each pass.

Algorithm
1.Start from the beginning of the list.
2.Compare the first two elements. If the first element is greater than the second, swap them.
3.Move to the next pair of elements, repeat the comparison and swap if necessary.
4.Continue this process for each pair of adjacent elements in the list, passing through the entire list multiple times if necessary.
5.Stop when no more swaps are needed during a pass, indicating the list is sorted.

Pseudocode

```
procedure bubbleSort(arr: list of elements)
    n = length of arr
    for i = 0 to n-1 do
        swapped = false
        for j = 0 to n-i-2 do
            if arr[j] > arr[j+1] then
                swap(arr[j], arr[j+1])
                swapped = true
        if swapped == false then
            break
```

Example:

```
#include <stdio.h>

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++) {
            // Swap if the element found is greater than the next element
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

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

```
        printf("%d ", arr[i]);
    }
    printf("\n");
}

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

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

    // Perform Bubble Sort
    bubbleSort(arr, n);

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

    return 0;
}
```

Application:
Bubble sort is primarily used for educational purposes due to its simplicity and ease of understanding. It is rarely used in practice for large datasets due to its ð‘‚(ð‘›2) time complexity, which makes it inefficient compared to more advanced sorting algorithms like merge sort or quicksort.

Conclusion
Bubble Sort is a simple and straightforward sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Experiment no. 8

Title: Write a program to sort the elements using Insertion Sort.

Theory:
Insertion Sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is similar to sorting playing cards in your hands, where you pick up one card at a time and insert it into its correct position among the already sorted cards.

Algorithm:
1.Start: Assume the first element is already sorted (considered as the sorted part).
2.Iterate: Move through the remaining elements of the array.
3.Insertion: For each element, compare it with elements in the sorted part of the array (from right to left).
4.Shift: Shift the sorted elements to the right until the correct position is found for the current element.
5.Place: Insert the current element into its correct position.
6.Repeat: Continue until all elements are sorted.

Pseudocode:

```
procedure insertionSort(arr: list of elements)
   n = length of arr
   for i = 1 to n-1 do
      key = arr[i]
      j = i - 1
      // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their
current position
      while j >= 0 and arr[j] > key do
         arr[j + 1] = arr[j]
         j = j - 1
      end while
      arr[j + 1] = key
```

Example

```
#include <stdio.h>

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

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

```
        }
        arr[j + 1] = key;
    }
}

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

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

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

    // Perform Insertion Sort
    insertionSort(arr, n);

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

    return 0;
}
```

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

Applications:
Insertion Sort is effective for small datasets or partially sorted arrays where simplicity and stability are more desirable than absolute performance.

It is often used as a building block for more advanced sorting algorithms like Shell Sort and as an example in algorithm design and analysis courses.

Conclusion

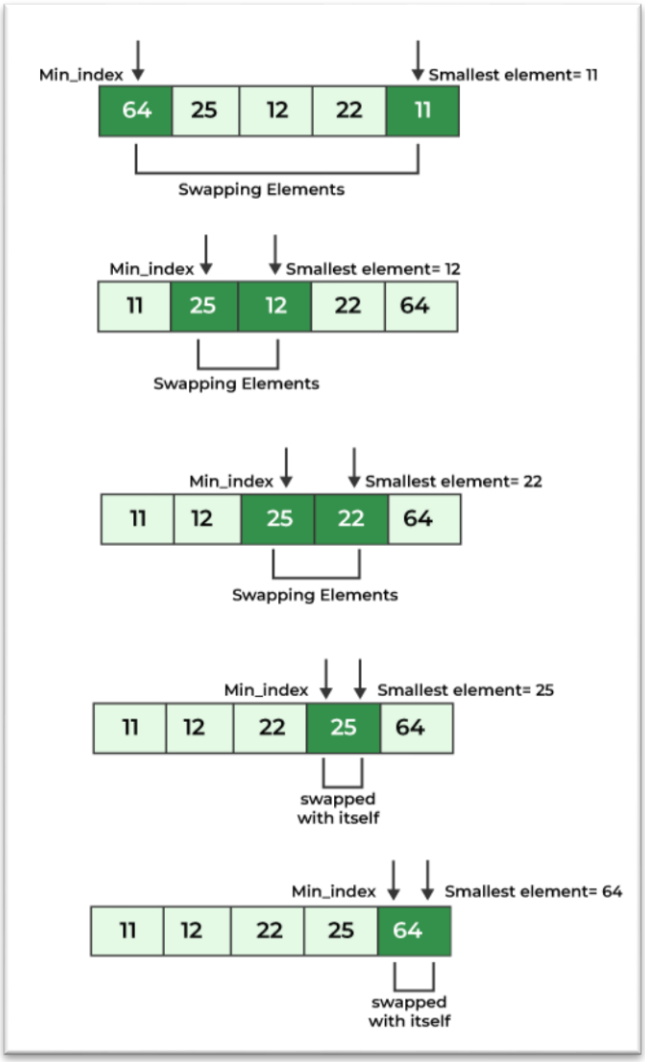
Insertion Sort provides a straightforward method for sorting small datasets efficiently and is particularly useful when elements are added to an already partially sorted array. While not as efficient as more advanced algorithms for large datasets, its simplicity and stability make it a valuable tool in various educational and practical contexts.

Experiment no. 9

Title: Write a program to sort the elements using Selection Sort.

Theory:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted subarray and putting it at the beginning of the sorted subarray.
In each pass, the minimum element is identified from the unsorted part of the array and placed at the beginning of the sorted part of the array. The sorted portion of the array gradually grows with each pass until the entire array becomes sorted.



Approach:

- Initialize minimum value min_idx to location 0.
- Traverse the array to find the minimum element in the array.
- While traversing if any element smaller than min_idx is found then swap both values.
- Then, increment min_idx to point to the next element.
- Repeat until the array is sorted.

Algorithm:

- Find the smallest element in the unsorted array.
- Swap it with the element at the beginning of the unsorted array.
- Repeat this process for the remaining unsorted portion until the entire array is sorted.

Pseudocode:

```
procedure selectionSort(A : list of sortable items)
   n = length(A)
   for i = 0 to n-2 do:
      min_index = i
      for j = i+1 to n-1 do:
         if A[j] < A[min_index] then:
            min_index = j
      swap A[i] and A[min_index]
   end for
end procedure
```

Example:

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
   int i, j, min_index;

   for (i = 0; i < n-1; i++) {
      min_index = i;
      for (j = i+1; j < n; j++) {
         if (arr[j] < arr[min_index]) {
            min_index = j;
         }
      }
      // Swap the found minimum element with the first element
      int temp = arr[min_index];
      arr[min_index] = arr[i];
      arr[i] = temp;
   }
}

int main() {
   int n, i;
   printf("Enter number of elements: ");
   scanf("%d", &n);

   int arr[n];
   printf("Enter elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
```

```
    selectionSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter number of elements: 4
Enter elements:
5 345 234 213
Sorted array:
5 213 234 345
```

Time Complexity:

The time complexity of the Selection Sort algorithm is determined by the nested loops:

1. The outer loop runs n−1n-1n−1 times.
2. The inner loop runs n−i−1n-i-1n−i−1 times in the worst case for each iteration of the outer loop.

Therefore, the total number of comparisons is: T(n)=(n−1)+(n−2)+...+1=n(n−1)2T(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2}T(n)=(n−1)+(n−2)+...+1=2n(n−1) This simplifies to O(n2)O(n^2)O(n2).

- **Best Case:** O(n2)O(n^2)O(n2) - even if the array is already sorted, Selection Sort will still go through the entire process.
- **Average Case:** O(n2)O(n^2)O(n2)
- **Worst Case:** O(n2)O(n^2)O(n2)

Space Complexity:

Selection Sort is an in-place sorting algorithm, meaning it does not require additional storage for a copy of the array, except for a few temporary variables.

- **Space Complexity:** O(1)O(1)O(1) - only a constant amount of extra space is needed regardless of the input size.

Applications:

Small datasets:

Selection sort performs reasonably well on small datasets, where its quadratic time complexity isn't a significant bottleneck.

Situations where simplicity is prioritized:

Due to its easy implementation and understanding, selection sort might be preferred in scenarios where code readability and maintainability are more important than speed, such as educational settings or prototyping.

Memory-constrained environments:

Selection sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory. This makes it suitable for environments where memory usage is a concern.

Conclusion:

In practice, Selection Sort is not efficient for large datasets compared to more advanced algorithms like Quicksort or Mergesort, which have better average and worst-case time complexities. However, its simplicity and ease of implementation make it a good introductory algorithm for learning purposes.

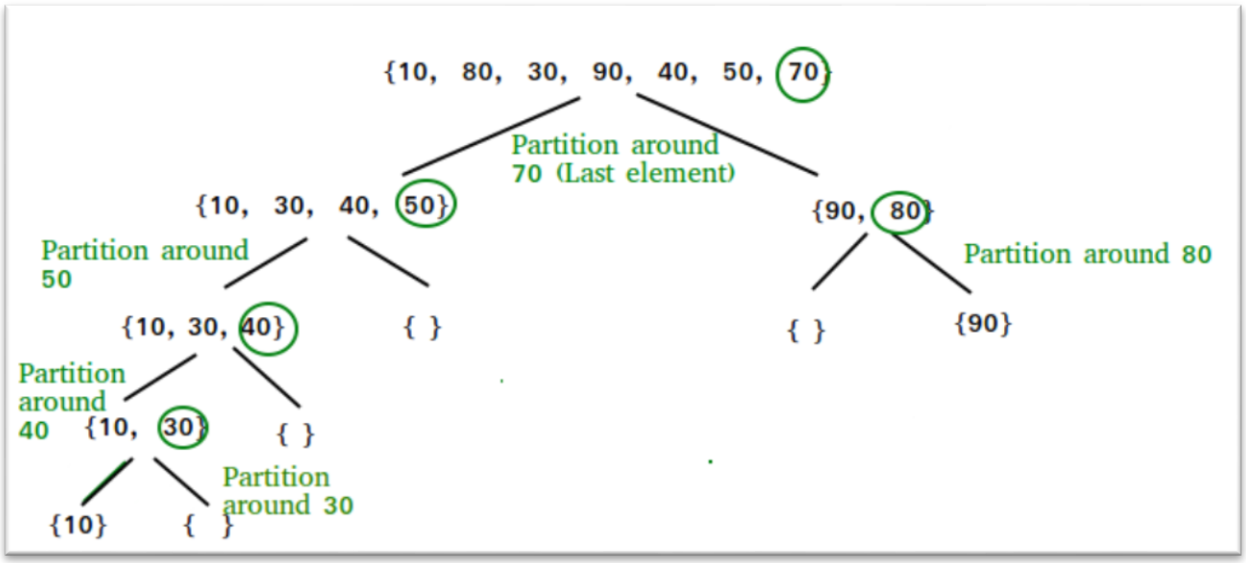
Experiment no. 10

Title: Implement Merge Sort and Quick sort to sort the given elements. Compare their time complexities.

Theory:

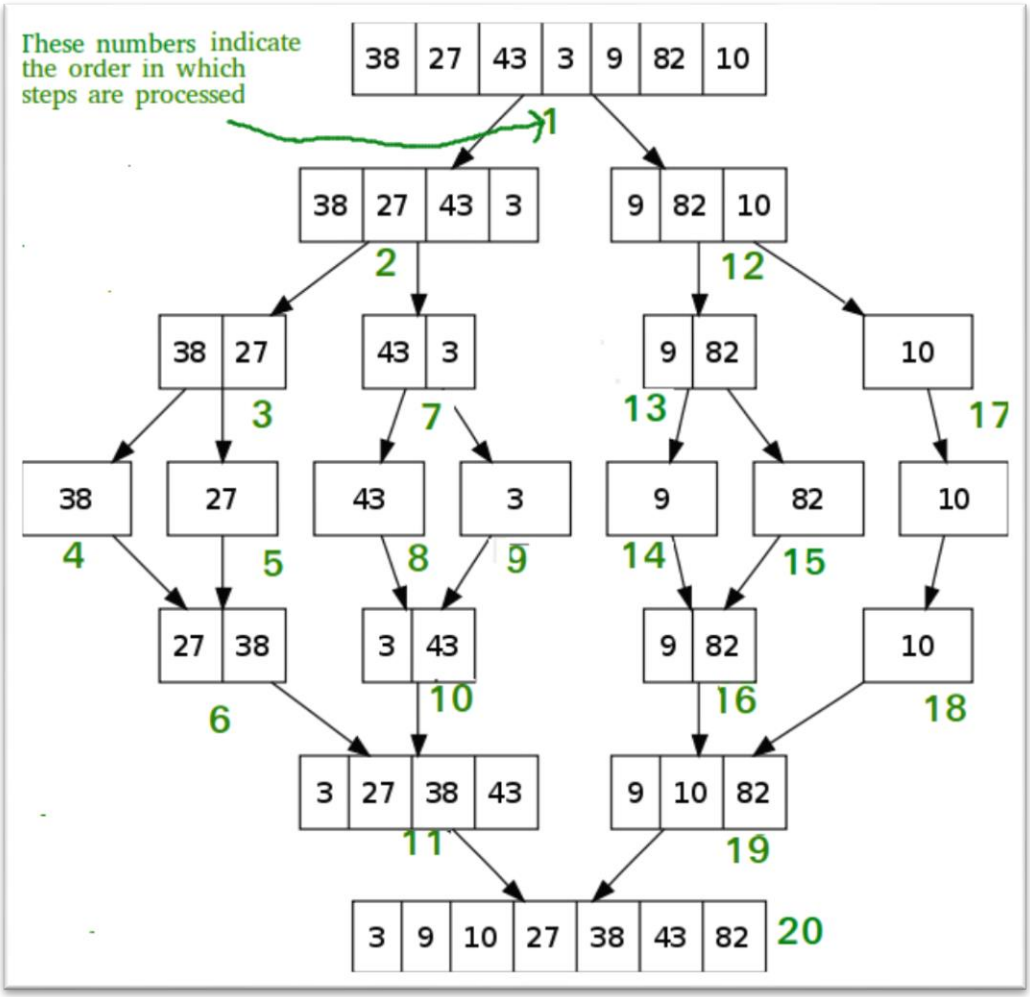
Quick sort is an internal algorithm which is based on divide and conquer strategy. In this:

The array of elements is divided into parts repeatedly until it is not possible to divide it further.
It is also known as "partition exchange sort".
It uses a key element (pivot) for partitioning the elements.
One left partition contains all those elements that are smaller than the pivot and one right partition contains all those elements which are greater than the key element.



Merge sort is an external algorithm and based on divide and conquer strategy. In this:

The elements are split into two sub-arrays (n/2) again and again until only one element is left.
Merge sort uses additional storage for sorting the auxiliary array.
Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.
At last, the all sub arrays are merged to make it 'n' element size of the array.



MERGE SORT:

Algorithm:

Divide the array into two halves.
Recursively sort each half.
Merge the two sorted halves to produce the sorted array.

Pseudocode:

```
procedure mergeSort(arr, left, right)
    if left < right
        mid = (left + right) / 2
        mergeSort(arr, left, mid)
        mergeSort(arr, mid + 1, right)
        merge(arr, left, mid, right)
    end if
end procedure

procedure merge(arr, left, mid, right)
    n1 = mid - left + 1
```

```
    n2 = right - mid
    create arrays L[0..n1-1] and R[0..n2-1]
    for i = 0 to n1-1
        L[i] = arr[left + i]
    for j = 0 to n2-1
        R[j] = arr[mid + 1 + j]
    i = 0, j = 0, k = left
    while i < n1 and j < n2
        if L[i] <= R[j]
            arr[k] = L[i]
            i = i + 1
        else
            arr[k] = R[j]
            j = j + 1
        k = k + 1
    while i < n1
        arr[k] = L[i]
        i = i + 1
        k = k + 1
    while j < n2
        arr[k] = R[j]
        j = j + 1
        k = k + 1
end procedure
```

C Program:

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

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
```

```
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

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

    return 0;
}
```

Output:

Enter number of elements: 4

Enter elements:
7 14 6 12
Sorted array:
6 7 12 14

QUICK SORT:

Algorithm:

Choose a pivot element from the array.
Partition the array into two halves such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
Recursively apply the above steps to the two halves.

Pseudocode:

```
procedure quickSort(arr, low, high)
   if low < high
      pivot_index = partition(arr, low, high)
      quickSort(arr, low, pivot_index - 1)
      quickSort(arr, pivot_index + 1, high)
   end if
end procedure

procedure partition(arr, low, high)
   pivot = arr[high]
   i = low - 1
   for j = low to high - 1
      if arr[j] <= pivot
         i = i + 1
         swap arr[i] with arr[j]
   swap arr[i + 1] with arr[high]
   return i + 1
end procedure
```

C Program:

```
#include <stdio.h>

void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}

int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
```

```
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Output:

```
Enter number of elements: 7
Enter elements:
4 364 243 67 568 132 74
```

Sorted array:
4 67 74 132 243 364 568

Time Complexity Comparison

- **Merge Sort:**
  - **Best Case:** O(nlog⁡n)O(n \log n)O(nlogn)
  - **Average Case:** O(nlog⁡n)O(n \log n)O(nlogn)
  - **Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn)
  - **Space Complexity:** O(n)O(n)O(n) due to the temporary arrays used during merging.
- **Quick Sort:**
  - **Best Case:** O(nlog⁡n)O(n \log n)O(nlogn)
  - **Average Case:** O(nlog⁡n)O(n \log n)O(nlogn)
  - **Worst Case:** O(n2)O(n^2)O(n2) when the pivot chosen is the smallest or largest element repeatedly.
  - **Space Complexity:** O(log⁡n)O(\log n)O(logn) due to the recursive call stack.

Conclusion:

Both Merge Sort and Quick Sort are efficient sorting algorithms with their own advantages and disadvantages:

- **Merge Sort** is stable and guarantees O(nlog⁡n)O(n \log n)O(nlogn) time complexity for all cases, but it requires additional space.
- **Quick Sort** is generally faster for large datasets and is in-place, but its worst-case time complexity is O(n2)O(n^2)O(n2). However, this can be mitigated with good pivot selection techniques (e.g., using median-of-three).

Experiment no. 11

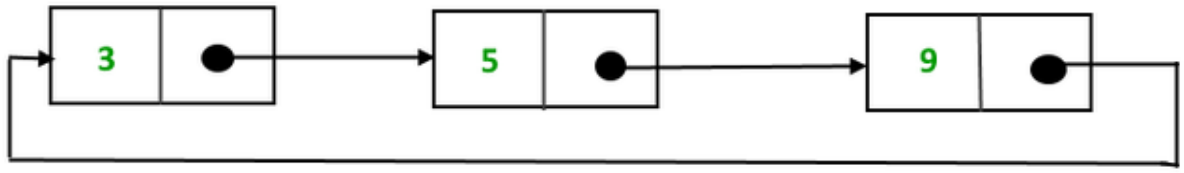
Title: Write a program to implement Circular Linked List.

Theory:

The circular linked list is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

There are generally two types of circular linked lists:

- **Circular singly linked list:** In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



Representation of Circular singly linked list

- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.

Representation of circular doubly linked list

Algorithm:

Insertion at the Beginning:

Create a new node.
Point the new node to the current head node.

Update the last node to point to the new node if the list is not empty.
Make the new node the head.
Insertion at the End:

Create a new node.
Traverse to the last node.
Update the last node to point to the new node.
Point the new node to the head node.
Deletion of a Node:

Traverse to find the node to delete and its previous node.
Update the previous node to point to the next node of the node to be deleted.
Traversal:

Start from the head node.
Follow the next pointers until reaching the head node again.

Pseudocode:

```
procedure insertAtBeginning(head, value)
    create newNode
    newNode.value = value
    if head is null
        newNode.next = newNode
        head = newNode
    else
        temp = head
        while temp.next != head
            temp = temp.next
        newNode.next = head
        temp.next = newNode
        head = newNode
    end if
end procedure

procedure insertAtEnd(head, value)
    create newNode
    newNode.value = value
    if head is null
        newNode.next = newNode
        head = newNode
    else
        temp = head
        while temp.next != head
            temp = temp.next
        temp.next = newNode
        newNode.next = head
    end if
end procedure

procedure deleteNode(head, key)
```

```
    if head is null
        return
    if head.value == key and head.next == head
        head = null
        return
    temp = head
    if head.value == key
        while temp.next != head
            temp = temp.next
        temp.next = head.next
        head = head.next
    else
        prev = null
        while temp.next != head and temp.value != key
            prev = temp
            temp = temp.next
        if temp.value == key
            prev.next = temp.next
    end if
end procedure

procedure traverse(head)
    if head is null
        return
    temp = head
    do
        print temp.value
        temp = temp.next
    while temp != head
end procedure
```

Example:

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

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

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

```
        }
        newNode->next = *head;
        temp->next = newNode;
        *head = newNode;
    }
}

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

void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        return;
    }

    struct Node *temp = *head, *prev;

    if (temp->data == key && temp->next == *head) {
        *head = NULL;
        free(temp);
        return;
    }

    if (temp->data == key) {
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = (*head)->next;
        free(*head);
        *head = temp->next;
    } else {
        while (temp->next != *head && temp->data != key) {
            prev = temp;
            temp = temp->next;
        }
        if (temp->data == key) {
            prev->next = temp->next;
            free(temp);
```

```
        }
    }
}

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

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

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 30);

    printf("Circular Linked List after inserting at beginning:\n");
    traverse(head);

    insertAtEnd(&head, 40);
    insertAtEnd(&head, 50);

    printf("Circular Linked List after inserting at end:\n");
    traverse(head);

    deleteNode(&head, 20);

    printf("Circular Linked List after deleting a node:\n");
    traverse(head);

    return 0;
}
```

Output:

```
Circular Linked List after inserting at beginning:
30 20 10
Circular Linked List after inserting at end:
30 20 10 40 50
Circular Linked List after deleting a node:
30 10 40 50
```

Time and Space Complexity

Time Complexity:

- **Insertion at the Beginning:** O(n)O(n)O(n) due to the traversal to the last node.
- **Insertion at the End:** O(n)O(n)O(n) due to the traversal to the last node.
- **Deletion of a Node:** O(n)O(n)O(n) due to the traversal to find the node and its previous node.
- **Traversal:** O(n)O(n)O(n) due to visiting each node once.

Space Complexity:

- **Space Complexity:** O(n)O(n)O(n) due to the space required to store the nodes of the circular linked list.

Applications:

Implementation of a Queue: A circular linked list can be used to implement a queue, where the front and rear of the queue are the first and last nodes of the list, respectively. In this implementation, when an element is enqueued, it is added to the rear of the list and when an element is dequeued, it is removed from the front of the list.
Music or Media Player: Circular linked lists can be used to create a playlist for a music or media player. The playlist can be represented as a circular linked list, where each node contains information about a song or media item, and the next node points to the next song in the playlist. This allows for continuous playback of the playlist.
Hash table implementation: Circular linked lists can be used to implement a hash table, where each index in the table is a circular linked list. This is a form of collision resolution, where when two keys hash to the same index, the nodes are added to the circular linked list at that index.
Memory allocation: In computer memory management, circular linked lists can be used to keep track of allocated and free blocks of memory. Each node in the list represents a block of memory and its status, with the next node pointing to the next block of memory. When a block of memory is freed, it can be added back to the circular linked list.
Navigation systems: Circular linked lists can be used to model the movements of vehicles on a circular route, such as buses, trains or planes that travel in a loop or circular route. Each node in the list represents the location of the vehicle on the route, with the next node pointing to the next location on the route.

Conclusion:

The Circular Linked List provides efficient memory utilization by using a single pointer for each node. It's particularly useful for applications requiring cyclic iterations over the data, such as round-robin scheduling, buffering, and memory management. However, insertion and deletion operations are linear in time complexity, making it less efficient than other data structures like arrays or doubly linked lists for these operations.

Experiment no. 12

Title: Implement a binary search tree (BST) to perform insertion, deletion, and search operations.

Theory:

Definition**:**

A binary *sea*rch tree is a binary tree data structure where each node has at most two children, referred to as the left child and the right child. The key in each node must be greater than all keys in its left subtree and less than all keys in its right subtree.

Operations

Insertion:

Start at the root.
Compare the new key with the key of the current node.
If smaller, move to the left child; if larger or equal, move to the right child.
Repeat until an appropriate empty location is found, then insert the new node.

Deletion:

Three cases to consider:
Node has no children: Simply remove it.
Node has one child: Replace node with its child.
Node has two children: Find the minimum value in the right subtree (or maximum in the left subtree), replace the node's value with this value, and recursively delete the minimum node from the right subtree (or maximum from the left subtree).

Search:

Start from the root.
Compare the key with the current node's key.
If found, return the node; if smaller, move to the left child; if larger, move to the right child.
Repeat until the node is found or a null pointer is encountered.

1. Insertion Algorithm

Algorithm:

1. If the tree is empty, create a new node as the root.
2. Otherwise, start from the root and compare the key to be inserted with the current node's key.

3. If the key is less than the current node's key, move to the left child.
4. If the key is greater than the current node's key, move to the right child.
5. Repeat steps 2-4 until the appropriate null position is found.
6. Insert the new node at the found position.

Pseudocode:

```
plaintext
Copy code
INSERT(node, key)
    IF node IS NULL
        RETURN newNode(key)
    ENDIF

    IF key < node.key
        node.left = INSERT(node.left, key)
    ELSE IF key > node.key
        node.right = INSERT(node.right, key)
    ENDIF

    RETURN node
```

2. Deletion Algorithm

Algorithm:

1. If the tree is empty, return null.
2. Otherwise, start from the root and find the node to be deleted.
3. If the node to be deleted has no children, simply remove it.
4. If the node has one child, replace the node with its child.
5. If the node has two children, find the inorder successor (smallest node in the right subtree).
6. Replace the node's key with the inorder successor's key.
7. Recursively delete the inorder successor.

Pseudocode:

```
plaintext
Copy code
DELETE(node, key)
    IF node IS NULL
        RETURN node
    ENDIF

    IF key < node.key
        node.left = DELETE(node.left, key)
    ELSE IF key > node.key
        node.right = DELETE(node.right, key)
    ELSE
        IF node.left IS NULL
            temp = node.right
            FREE(node)
            RETURN temp
        ELSE IF node.right IS NULL
            temp = node.left
            FREE(node)
            RETURN temp
```

```
        ENDIF

        temp = MIN_VALUE_NODE(node.right)
        node.key = temp.key
        node.right = DELETE(node.right, temp.key)
    ENDIF

    RETURN node
```

MIN_VALUE_NODE Pseudocode:

```
plaintext
Copy code
MIN_VALUE_NODE(node)
    current = node
    WHILE current.left IS NOT NULL
        current = current.left
    ENDWHILE
    RETURN current
```

3. Search Algorithm

Algorithm:

1. If the tree is empty or the key is found at the root, return the node.
2. Otherwise, compare the key with the root's key.
3. If the key is less than the root's key, search in the left subtree.
4. If the key is greater than the root's key, search in the right subtree.
5. Repeat steps 2-4 until the key is found or a null pointer is encountered.

Pseudocode:

```
plaintext
Copy code
SEARCH(node, key)
    IF node IS NULL OR node.key == key
        RETURN node
    ENDIF

    IF key < node.key
        RETURN SEARCH(node.left, key)
    ELSE
        RETURN SEARCH(node.right, key)
    ENDIF
```

4. Inorder Traversal Algorithm

Algorithm:

1. If the tree is not empty, traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

Pseudocode:

```
plaintext
Copy code
INORDER(node)
    IF node IS NOT NULL
        INORDER(node.left)
        PRINT node.key
        INORDER(node.right)
    ENDIF
```

Example:

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

// Structure of a BST node
struct TreeNode {
   int key;
   struct TreeNode *left, *right;
};

// Function to create a new BST node
struct TreeNode* newNode(int item) {
   struct TreeNode* temp = (struct TreeNode*)malloc(sizeof(struct TreeNode));
   temp->key = item;
   temp->left = temp->right = NULL;
   return temp;
}

// Function to insert a new key in BST
struct TreeNode* insert(struct TreeNode* node, int key) {
   // If the tree is empty, return a new node
   if (node == NULL) return newNode(key);

   // Otherwise, recur down the tree
   if (key < node->key)
      node->left = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);

   // return the (unchanged) node pointer
   return node;
}

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

```
// Function to search a given key in BST
struct TreeNode* search(struct TreeNode* root, int key) {
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

// Function to find the minimum value node in a given BST
struct TreeNode* minValueNode(struct TreeNode* node) {
    struct TreeNode* current = node;

    // Loop down to find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

// Function to delete a node with given key from BST
struct TreeNode* deleteNode(struct TreeNode* root, int key) {
    // Base case
    if (root == NULL) return root;

    // Recur down the tree
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // If key is same as root's key, then This is the node to be deleted
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct TreeNode* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct TreeNode* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: Get the inorder successor (smallest in the right subtree)
        struct TreeNode* temp = minValueNode(root->right);
```

```
        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Main function to test the above functions
int main() {
    struct TreeNode* root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    struct TreeNode* searchResult = search(root, 70);
    if (searchResult != NULL)
        printf("\nFound key 70 in the tree\n");
    else
        printf("\nKey 70 not found in the tree\n");

    return 0;
}
```

Output:

Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
Found key 70 in the tree

- **Time Complexity:**
  - Insertion: O(log⁡n)O(\log n)O(logn) on average, O(n)O(n)O(n) in the worst case.
  - Deletion: O(log⁡n)O(\log n)O(logn) on average, O(n)O(n)O(n) in the worst case.
  - Search: O(log⁡n)O(\log n)O(logn) on average, O(n)O(n)O(n) in the worst case.
  - Inorder Traversal: O(n)O(n)O(n).
- **Space Complexity:**
  - Insertion: O(1)O(1)O(1) iterative, O(h)O(h)O(h) recursive.
  - Deletion: O(1)O(1)O(1) iterative, O(h)O(h)O(h) recursive.
  - Search: O(1)O(1)O(1) iterative, O(h)O(h)O(h) recursive.
  - Inorder Traversal: O(1)O(1)O(1) iterative, O(h)O(h)O(h) recursive.

Applications of BST

- **Data Storage:** Efficient for storing and retrieving data in sorted order.
- **Symbol Tables:** Used in compilers and interpreters to store identifiers and their attributes.
- **Database Indexing:** Helps in quick retrieval of data based on keys.
- **Priority Queues:** Supports efficient insertion, deletion, and extraction of minimum or maximum.

Conclusion

Binary search trees provide a versatile data structure for managing sorted data efficiently. They support operations such as insertion, deletion, and search in logarithmic time in the average case, making them suitable for various applications where sorted data manipulation is required. Understanding their implementation and applications helps in designing efficient algorithms and data structures for real-world problems.

Experiment no. 13

Title: Implement graph traversal algorithms BFS and DFS.

Theory:

Breadth-First Search (BFS)

BFS is a graph traversal algorithm that starts at a selected node (called the source node) and explores all of the neighbor nodes at the present depth before moving on to nodes at the next depth level.

Applications:

- Finding the shortest path in an unweighted graph.
- Crawling websites.
- Broadcasting in networks.
- Finding all connected components in a graph.

Depth-First Search (DFS)

DFS is a graph traversal algorithm that starts at a selected node and explores as far as possible along each branch before backtracking.

Applications:

- Detecting cycles in a graph.
- Pathfinding.
- Topological sorting.
- Solving puzzles and games.

Algorithms and Pseudocode

BFS Algorithm

1. Initialize a queue and enqueue the starting node.
2. Mark the starting node as visited.
3. While the queue is not empty:
    - Dequeue a node from the queue.
    - Process the node.
    - Enqueue all unvisited neighbors of the dequeued node and mark them as visited.

Pseudocode:

```
plaintext
Copy code
BFS(graph, start_vertex):
    create a queue Q
    mark start_vertex as visited
    enqueue start_vertex into Q
```

```
    while Q is not empty:
        vertex = dequeue Q
        process vertex

        for each neighbor of vertex:
            if neighbor is not visited:
                mark neighbor as visited
                enqueue neighbor into Q
```

DFS Algorithm

1. Initialize a stack and push the starting node.
2. Mark the starting node as visited.
3. While the stack is not empty:
    - Pop a node from the stack.
    - Process the node.
    - Push all unvisited neighbors of the popped node and mark them as visited.

Pseudocode:

```
plaintext
Copy code
DFS(graph, start_vertex):
    create a stack S
    mark start_vertex as visited
    push start_vertex into S

    while S is not empty:
        vertex = pop S
        process vertex

        for each neighbor of vertex:
            if neighbor is not visited:
                mark neighbor as visited
                push neighbor into S
```

Time and Space Complexity

Time Complexity:

- BFS: O(V+E)O(V + E)O(V+E)
- DFS: O(V+E)O(V + E)O(V+E)

Space Complexity:

- BFS: O(V)O(V)O(V) for the queue and O(V)O(V)O(V) for the visited array.
- DFS: O(V)O(V)O(V) for the stack and O(V)O(V)O(V) for the visited array.

Implementations in C

BFS in C

```
c
Copy code
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
```

```c
int adj[MAX][MAX];
int visited[MAX];
int queue[MAX], front = -1, rear = -1;

void enqueue(int v) {
    if (rear == MAX - 1) return;
    if (front == -1) front = 0;
    queue[++rear] = v;
}

int dequeue() {
    if (front == -1 || front > rear) return -1;
    return queue[front++];
}

void bfs(int start, int n) {
    int i;
    enqueue(start);
    visited[start] = 1;

    while (front <= rear) {
        int v = dequeue();
        printf("%d ", v);
        for (i = 0; i < n; i++) {
            if (adj[v][i] && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}

int main() {
    int n, i, j, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

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

    printf("BFS Traversal starting from vertex %d: ", start);
    bfs(start, n);

    return 0;
}
```

DFS in C

c

Copy code

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

int adj[MAX][MAX];
int visited[MAX];
```

```
void dfs(int v, int n) {
    int i;
    printf("%d ", v);
    visited[v] = 1;
    for (i = 0; i < n; i++) {
        if (adj[v][i] && !visited[i]) {
            dfs(i, n);
        }
    }
}

int main() {
    int n, i, j, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

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

    printf("DFS Traversal starting from vertex %d: ", start);
    dfs(start, n);

    return 0;
}
```

Conclusion

BFS and DFS are fundamental graph traversal algorithms used in various applications such as finding shortest paths, detecting cycles, and performing topological sorting. BFS uses a queue to explore nodes level by level, while DFS uses a stack (or recursion) to explore as far as possible along each branch. Both algorithms have a time complexity of O(V+E)O(V + E)O(V+E) and require additional space for storing visited nodes and traversal data structures.

Experiment no. 14

Title: Implement a hash table using linear probing and quadratic probing techniques.

Theory

Hash Table

A hash table is a data structure that maps keys to values using a hash function. The hash function computes an index into an array in which an element will be inserted or searched. When a collision occurs (i.e., two keys hash to the same index), we need a strategy to resolve it. Two common techniques for collision resolution are linear probing and quadratic probing.

Linear Probing

In linear probing, when a collision occurs, we linearly search for the next available slot in the hash table.

Quadratic Probing

In quadratic probing, when a collision occurs, we use a quadratic function to search for the next available slot. This helps in spreading out the keys more uniformly compared to linear probing.

Algorithms

Hash Function

A simple hash function can be `key % table_size`, where `key` is the integer key and `table_size` is the size of the hash table.

Insertion with Linear Probing

1. Compute the initial index using the hash function.
2. If the slot is empty, insert the key.
3. If the slot is occupied, linearly probe to find the next available slot.

Pseudocode:

```
plaintext
Copy code
INSERT_LINEAR_PROBING(key):
    index = HASH_FUNCTION(key)
    i = 0
    while table[(index + i) % table_size] is not empty:
        i = i + 1
    table[(index + i) % table_size] = key
```

Insertion with Quadratic Probing

1. Compute the initial index using the hash function.
2. If the slot is empty, insert the key.

3. If the slot is occupied, use a quadratic function to find the next available slot.

Pseudocode:

```
plaintext
Copy code
INSERT_QUADRATIC_PROBING(key):
    index = HASH_FUNCTION(key)
    i = 0
    while table[(index + i * i) % table_size] is not empty:
        i = i + 1
    table[(index + i * i) % table_size] = key
```

Search with Probing

1. Compute the initial index using the hash function.
2. If the slot contains the key, return success.
3. If the slot is empty, return failure.
4. If the slot contains a different key, probe using the respective probing method until the key is found or an empty slot is reached.

Time and Space Complexity

Time Complexity:

- Average: O(1)O(1)O(1)
- Worst case: O(n)O(n)O(n) (in case of many collisions)

Space Complexity:

- O(n)O(n)O(n), where nnn is the size of the hash table.

Applications

- Database indexing
- Caches
- Sets
- Associative arrays

Conclusion

Hash tables are efficient data structures for storing and retrieving data using keys. Linear probing and quadratic probing are techniques to handle collisions, with linear probing being simpler and quadratic probing providing better distribution of keys.

C Implementation

Linear Probing

```
c
Copy code
#include <stdio.h>
#define TABLE_SIZE 10
#define EMPTY -1
```

```
int hashTable[TABLE_SIZE];

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void insertLinearProbing(int key) {
    int index = hashFunction(key);
    int i = 0;
    while (hashTable[(index + i) % TABLE_SIZE] != EMPTY) {
        i++;
    }
    hashTable[(index + i) % TABLE_SIZE] = key;
}

int searchLinearProbing(int key) {
    int index = hashFunction(key);
    int i = 0;
    while (hashTable[(index + i) % TABLE_SIZE] != EMPTY) {
        if (hashTable[(index + i) % TABLE_SIZE] == key) {
            return (index + i) % TABLE_SIZE;
        }
        i++;
    }
    return -1;
}

void displayHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != EMPTY) {
            printf("Index %d: %d\n", i, hashTable[i]);
        } else {
            printf("Index %d: EMPTY\n", i);
        }
    }
}

int main() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = EMPTY;
    }

    insertLinearProbing(12);
    insertLinearProbing(22);
    insertLinearProbing(32);

    displayHashTable();

    int index = searchLinearProbing(22);
    if (index != -1) {
        printf("Key 22 found at index %d\n", index);
    } else {
        printf("Key 22 not found\n");
    }

    return 0;
}
```

Quadratic Probing

c

Copy code

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

```c
#define TABLE_SIZE 10
#define EMPTY -1

int hashTable[TABLE_SIZE];

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void insertQuadraticProbing(int key) {
    int index = hashFunction(key);
    int i = 0;
    while (hashTable[(index + i * i) % TABLE_SIZE] != EMPTY) {
        i++;
    }
    hashTable[(index + i * i) % TABLE_SIZE] = key;
}

int searchQuadraticProbing(int key) {
    int index = hashFunction(key);
    int i = 0;
    while (hashTable[(index + i * i) % TABLE_SIZE] != EMPTY) {
        if (hashTable[(index + i * i) % TABLE_SIZE] == key) {
            return (index + i * i) % TABLE_SIZE;
        }
        i++;
    }
    return -1;
}

void displayHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != EMPTY) {
            printf("Index %d: %d\n", i, hashTable[i]);
        } else {
            printf("Index %d: EMPTY\n", i);
        }
    }
}

int main() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = EMPTY;
    }

    insertQuadraticProbing(12);
    insertQuadraticProbing(22);
    insertQuadraticProbing(32);

    displayHashTable();

    int index = searchQuadraticProbing(22);
    if (index != -1) {
        printf("Key 22 found at index %d\n", index);
    } else {
        printf("Key 22 not found\n");
    }

    return 0;
}
```

These implementations cover basic insertion and search operations using linear probing and quadratic probing techniques in C. You can expand them to handle deletions and other functionalities as needed.

Experiment no. 15

Title: Develop a program to manage student records using different file organization methods, including sequential, indexed, and direct file organization.

Theory

File Organization Methods

File organization refers to the way records are arranged and accessed in a file. There are three main methods of file organization:

1. **Sequential File Organization**: Records are stored in a sequence, one after the other. Accessing records involves reading them sequentially from the beginning until the desired record is found.
2. **Indexed File Organization**: An index is maintained alongside the file to speed up searches. The index contains pointers to the locations of the records.
3. **Direct (or Hash) File Organization**: Records are placed in a file using a hash function to determine their position. This allows direct access to records based on their key values.

Algorithms and Pseudocode

Sequential File Organization

Algorithm:

1. Open the file in append or read mode.
2. To add a record, append it to the end of the file.
3. To search for a record, read the file sequentially until the record is found.

Pseudocode:

```
plaintext
Copy code
ADD_RECORD_SEQUENTIAL(record):
    open file in append mode
    write record to file
    close file

SEARCH_RECORD_SEQUENTIAL(key):
    open file in read mode
    while not end of file:
        read record
        if record.key == key:
            return record
    close file
    return not found
```

Indexed File Organization

Algorithm:

1. Maintain an index file that contains the key and the position of each record.
2. To add a record, append it to the end of the data file and update the index file.
3. To search for a record, use the index to find the position and read the record directly.

Pseudocode:

```
plaintext
Copy code
ADD_RECORD_INDEXED(record):
    open data file in append mode
    write record to data file
    close data file
    update index file with key and position

SEARCH_RECORD_INDEXED(key):
    open index file in read mode
    if key found in index:
        get position from index
        open data file and seek to position
        read and return record
    close files
    return not found
```

Direct (Hash) File Organization

Algorithm:

1. Use a hash function to determine the position of the record.
2. To add a record, compute its position and write it to that position in the file.
3. To search for a record, compute its position and read the record directly.

Pseudocode:

```
plaintext
Copy code
HASH_FUNCTION(key):
    return key % table_size

ADD_RECORD_DIRECT(record):
    position = HASH_FUNCTION(record.key)
    open file in read/write mode
    seek to position
    write record to file
    close file

SEARCH_RECORD_DIRECT(key):
    position = HASH_FUNCTION(key)
    open file in read mode
    seek to position
    read record
    if record.key == key:
        return record
    close file
    return not found
```

Time and Space Complexity

- **Sequential File Organization:**

    - Time Complexity: O(n)O(n)O(n) for search, O(1)O(1)O(1) for add.
    - Space Complexity: O(n)O(n)O(n) for storing records.
- **Indexed File Organization:**
    - Time Complexity: O(logn)O(\log n)O(logn) for search with a balanced index, O(1)O(1)O(1) for add.
    - Space Complexity: O(n)O(n)O(n) for storing records and O(n)O(n)O(n) for storing the index.
- **Direct File Organization:**
    - Time Complexity: O(1)O(1)O(1) for search and add.
    - Space Complexity: O(n)O(n)O(n) for storing records (with possible wasted space due to hashing).

Applications

- **Sequential File Organization**: Suitable for applications where records are mostly read in sequence or added without frequent updates, such as log files.
- **Indexed File Organization**: Useful for applications with frequent searches and updates, such as databases.
- **Direct File Organization**: Ideal for applications requiring fast access to records, such as caching and real-time systems.

Conclusion

Each file organization method has its advantages and use cases. Sequential organization is simple but slow for searches, indexed organization offers a good balance for searches and updates, and direct organization provides the fastest access at the cost of potentially more complex implementation and wasted space.

C Program Implementation

Sequential File Organization

```
c
Copy code
#include <stdio.h>
#include <string.h>

typedef struct {
    int id;
    char name[50];
    int age;
} Student;

void addRecordSequential(const char *filename, Student student) {
    FILE *file = fopen(filename, "ab");
    fwrite(&student, sizeof(Student), 1, file);
    fclose(file);
}

int searchRecordSequential(const char *filename, int id, Student *student) {
    FILE *file = fopen(filename, "rb");
    while (fread(student, sizeof(Student), 1, file)) {
        if (student->id == id) {
            fclose(file);
            return 1;
        }
```

```
    }
    fclose(file);
    return 0;
}

int main() {
    const char *filename = "students.dat";
    Student student;

    // Add records
    student.id = 1; strcpy(student.name, "Alice"); student.age = 20;
    addRecordSequential(filename, student);
    student.id = 2; strcpy(student.name, "Bob"); student.age = 21;
    addRecordSequential(filename, student);

    // Search record
    int id = 2;
    if (searchRecordSequential(filename, id, &student)) {
        printf("Record found: ID=%d, Name=%s, Age=%d\n", student.id,
student.name, student.age);
    } else {
        printf("Record not found\n");
    }

    return 0;
}
```

Indexed File Organization

```
c
Copy code
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    int id;
    char name[50];
    int age;
    long pos;
} Index;

typedef struct {
    int id;
    char name[50];
    int age;
} Student;

void addRecordIndexed(const char *dataFilename, const char *indexFilename,
Student student) {
    FILE *dataFile = fopen(dataFilename, "ab");
    FILE *indexFile = fopen(indexFilename, "ab");
    Index index;

    fseek(dataFile, 0, SEEK_END);
    student.pos = ftell(dataFile);
    fwrite(&student, sizeof(Student), 1, dataFile);

    index.id = student.id;
    index.pos = student.pos;
    fwrite(&index, sizeof(Index), 1, indexFile);

    fclose(dataFile);
```

```
    fclose(indexFile);
}

int searchRecordIndexed(const char *dataFilename, const char *indexFilename,
int id, Student *student) {
    FILE *dataFile = fopen(dataFilename, "rb");
    FILE *indexFile = fopen(indexFilename, "rb");
    Index index;

    while (fread(&index, sizeof(Index), 1, indexFile)) {
        if (index.id == id) {
            fseek(dataFile, index.pos, SEEK_SET);
            fread(student, sizeof(Student), 1, dataFile);
            fclose(dataFile);
            fclose(indexFile);
            return 1;
        }
    }

    fclose(dataFile);
    fclose(indexFile);
    return 0;
}

int main() {
    const char *dataFilename = "students.dat";
    const char *indexFilename = "index.dat";
    Student student;

    // Add records
    student.id = 1; strcpy(student.name, "Alice"); student.age = 20;
    addRecordIndexed(dataFilename, indexFilename, student);
    student.id = 2; strcpy(student.name, "Bob"); student.age = 21;
    addRecordIndexed(dataFilename, indexFilename, student);

    // Search record
    int id = 2;
    if (searchRecordIndexed(dataFilename, indexFilename, id, &student)) {
        printf("Record found: ID=%d, Name=%s, Age=%d\n", student.id,
student.name, student.age);
    } else {
        printf("Record not found\n");
    }

    return 0;
}
```

Direct (Hash) File Organization

```
c
Copy code
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define TABLE_SIZE 10

typedef struct {
    int id;
    char name[50];
    int age;
} Student;
```

```
int hashFunction(int id) {
    return id % TABLE_SIZE;
}

void addRecordDirect(const char *filename, Student student) {
    FILE *file = fopen(filename, "rb+");
    int index = hashFunction(student.id);
    fseek(file, index * sizeof(Student), SEEK_SET);
    fwrite(&student, sizeof(Student), 1, file);
    fclose(file);
}

int searchRecordDirect(const char *filename, int id, Student *student) {
    FILE *file = fopen(filename, "rb");
    int index = hashFunction(id);
    fseek(file, index * sizeof(Student), SEEK_SET);
    fread(student, sizeof(Student), 1, file);
    fclose(file);
    return (student->id == id);
}

int main() {
    const char *filename = "students_hash.dat";
    Student student;

    FILE *file = fopen(filename, "wb");
    for (int i = 0; i < TABLE_SIZE; i++) {
        Student empty = {0, "", 0};
        fwrite(&empty, sizeof(Student), 1, file);
    }
    fclose(file);

    // Add records
    student.id = 1; strcpy(student.name, "Alice"); student.age = 20;
    addRecordDirect(filename, student);
    student.id = 2; strcpy(student.name, "Bob"); student.age = 21;
    addRecordDirect(filename, student);

    // Search record
    int id = 2;
    if (searchRecordDirect(filename, id, &student)) {
        printf("Record found: ID=%d, Name=%s, Age=%d\n", student.id,
student.name, student.age);
    } else {
        printf("Record not found\n");
    }

    return 0;
}
```

These implementations cover basic operations using different file organization methods. Each method offers different trade-offs in terms of time complexity, space complexity, and ease of implementation.

Experiment no. 16

Title: Write programs to demonstrate the construction of minimum spanning trees using Prim's and Kruskal's algorithms and find the shortest path using Dijkstra's algorithm.

Theory

Minimum Spanning Tree (MST)

A Minimum Spanning Tree (MST) of a weighted graph is a subset of the edges that connects all the vertices together, without any cycles, and with the minimum possible total edge weight. Two common algorithms for finding the MST are Prim's and Kruskal's algorithms.

Prim's Algorithm

Prim's algorithm starts with a single vertex and grows the MST one edge at a time, always adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.

Kruskal's Algorithm

Kruskal's algorithm builds the MST by sorting all the edges by weight and adding the smallest edge to the MST, ensuring no cycles are formed, until all vertices are connected.

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest paths from a single source vertex to all other vertices in a graph with non-negative weights.

Algorithms and Pseudocode

Prim's Algorithm

Algorithm:

1. Start with an arbitrary vertex.
2. At each step, add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
3. Repeat until all vertices are included.

Pseudocode:

```
plaintext
Copy code
PRIM_MST(graph):
    initialize mstSet[] to false for all vertices
    initialize key[] to infinity for all vertices
    key[0] = 0
    parent[0] = -1
    for count from 0 to V-1:
```

```
        u = vertex with minimum key value that is not in mstSet
        mstSet[u] = true
        for each vertex v adjacent to u:
            if weight[u][v] < key[v] and v is not in mstSet:
                key[v] = weight[u][v]
                parent[v] = u
    return parent[]
```

Kruskal's Algorithm

Algorithm:

1. Sort all edges in non-decreasing order of their weight.
2. Initialize an empty MST.
3. Add edges to the MST one by one, ensuring no cycles are formed, until all vertices are connected.

Pseudocode:

```
plaintext
Copy code
KRUSKAL_MST(graph):
    sort edges[] in non-decreasing order of weight
    initialize parent[] and rank[] for union-find
    mst = []
    for each edge (u, v) in sorted edges:
        if find(u) != find(v):
            mst.append((u, v))
            union(u, v)
    return mst
```

Dijkstra's Algorithm

Algorithm:

1. Initialize distances from the source to all vertices as infinity, except the source itself which is set to 0.
2. Use a priority queue to repeatedly extract the vertex with the smallest distance.
3. Update the distances of the adjacent vertices of the extracted vertex.

Pseudocode:

```
plaintext
Copy code
DIJKSTRA(graph, src):
    initialize distances[] to infinity
    distances[src] = 0
    initialize minHeap
    minHeap.insert((0, src))
    while minHeap is not empty:
        u = minHeap.extract_min()
        for each vertex v adjacent to u:
            if distances[u] + weight[u][v] < distances[v]:
                distances[v] = distances[u] + weight[u][v]
                minHeap.insert((distances[v], v))
    return distances[]
```

Time and Space Complexity

- **Prim's Algorithm:**
  - Time Complexity: O(V2)O(V^2)O(V2) for adjacency matrix, O(ElogV)O(E \log V)O(ElogV) for adjacency list with binary heap.
  - Space Complexity: O(V)O(V)O(V) for storing the MST.
- **Kruskal's Algorithm:**
  - Time Complexity: O(ElogE)O(E \log E)O(ElogE) or O(ElogV)O(E \log V)O(ElogV) with union-find.
  - Space Complexity: O(E)O(E)O(E) for storing edges and union-find data structures.
- **Dijkstra's Algorithm:**
  - Time Complexity: O(V2)O(V^2)O(V2) for adjacency matrix, O(E+VlogV)O(E + V \log V)O(E+VlogV) for adjacency list with binary heap.
  - Space Complexity: O(V)O(V)O(V) for storing distances and priority queue.

Applications

- **MST (Prim's and Kruskal's Algorithms)**:
  - Network design (e.g., designing communication, electrical, or transportation networks).
  - Approximation algorithms for NP-hard problems like the traveling salesman problem.
- **Dijkstra's Algorithm**:
  - GPS and mapping services.
  - Network routing protocols.

Conclusion

Prim's and Kruskal's algorithms are efficient methods for finding the MST of a graph, with different approaches to edge selection and addition. Dijkstra's algorithm is a fundamental algorithm for finding the shortest path in graphs with non-negative weights, widely used in various real-world applications.

C Program Implementation

Prim's Algorithm

```
c
Copy code
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define V 5

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v], min_index = v;
        }
    }
    return min_index;
}

void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
```

```
    }
}

void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    bool mstSet[V];

    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX, mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u, key[v] = graph[u][v];
            }
        }
    }
    printMST(parent, graph);
}

int main() {
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    primMST(graph);

    return 0;
}
```

Kruskal's Algorithm

```
c
Copy code
#include <stdio.h>
#include <stdlib.h>

#define V 5
#define E 7

typedef struct {
    int src, dest, weight;
} Edge;

typedef struct {
    int parent, rank;
} Subset;

int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
```

```
    }
    return subsets[i].parent;
}

void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int compareEdges(const void *a, const void *b) {
    Edge *edgeA = (Edge *)a;
    Edge *edgeB = (Edge *)b;
    return edgeA->weight - edgeB->weight;
}

void KruskalMST(Edge edges[], int numEdges) {
    Edge result[V];
    int e = 0;
    int i = 0;

    qsort(edges, numEdges, sizeof(Edge), compareEdges);

    Subset *subsets = (Subset *)malloc(V * sizeof(Subset));
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < numEdges) {
        Edge next_edge = edges[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
result[i].weight);
    }

    free(subsets);
}

int main() {
    Edge edges[E] = {
        {0, 1, 2},
        {0, 3, 6},
        {1, 2, 3},
```

```
        {1, 3, 8},
        {1, 4, 5},
        {2, 4, 7},
        {3, 4, 9}
    };

    KruskalMST(edges, E);

    return 0;
}
```

Dijkstra's Algorithm

```
c
Copy code
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define V 5

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v], min_index = v;
        }
    }
    return min_index;
}

void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX, sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printSolution(dist);
}
```

```
int main() {
    int graph[V][V] = {
        {0, 10, 0, 0, 0},
        {10, 0, 5, 0, 0},
        {0, 5, 0, 2, 1},
        {0, 0, 2, 0, 6},
        {0, 0, 1, 6, 0}
    };

    dijkstra(graph, 0);

    return 0;
}
```

These implementations demonstrate the construction of MSTs using Prim's and Kruskal's algorithms and finding the shortest path using Dijkstra’s algorithm in C. Each method offers different trade-offs and is suited to different types of problems and graph structures.