EXPERIMENT 2

Aim:- Implement sparse matrix using array

ALGORITHM

Input: Take a sparse matrix as input.

Initialization: Get the number of rows (rows) and columns (cols) in the matrix.

Print Header: Print a header indicating that the output is a sparse matrix, and include column headers for "Row," "Column," and "Value."

Loop Through Matrix: Use nested loops to iterate through each element in the matrix.
a. For each element at position (i, j), where i is the row index and j is the column index:
b. Check if the value at (i, j) is non-zero.
c. If non-zero, print a line with the row index, column index, and value.

Output: The printed lines represent the non-zero elements of the sparse matrix in bullet-point format.

SOURCE CODE

```cpp
#include<iostream>
using namespace std;

int main()
{

    int sparseMatrix[4][5] = { {0 , 0 , 3 , 0 , 4 },
                               {0 , 0 , 5 , 7 , 0 },
                               {0 , 0 , 0 , 0 , 0 },
                               {0 , 2 , 6 , 0 , 0 } };
    int size = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
                size++;
    int compactMatrix[3][size];
    int k = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
            {
                compactMatrix[0][k] = i;
                compactMatrix[1][k] = j;
                compactMatrix[2][k] = sparseMatrix[i][j];
                k++;
            }
    for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            printf("%d ", compactMatrix[i][j]);
        printf("\n");
    }
    return 0;
}
```

OUTPUT

```
0 0 1 1 3 3
2 4 2 3 1 2
3 4 5 7 2 6
```

EXPERIMENT 3

Aim:- Implementation of Stack and Queue using array.

STACK IMPLEMENTATION USING ARRAY

ALGORITHM

1. Initialize the Stack:
   A. Create an array to store the elements of the stack.
   B. Initialize a variable top to -1 to represent an empty stack.
2. Push Operation:
   A. Check if the stack is full (overflow).
   B. Increment the top variable.
   C. Add the new element to the array at the position indicated by top.
3. Pop Operation:
   A. Check if the stack is empty (underflow).
   B. Retrieve the element at the position indicated by top.
   C. Decrement the top variable.

SOURCE CODE

```
#include <stdio.h>

#define MAX_SIZE 5
int stack[MAX_SIZE];
int top = -1;
int is_empty() {
    return top == -1;
}
int is_full() {
    return top == MAX_SIZE - 1;
}
void push(int item) {
    if (!is_full()) {
        top++;
        stack[top] = item;
        printf("Pushed %d to the stack.\n", item);
    } else {
```

```c
        printf("Stack Overflow - Cannot push to a full stack.\n");
    }
}
int pop() {
    if (!is_empty()) {
        int item = stack[top];
        top--;
        printf("Popped %d from the stack.\n", item);
        return item;
    } else {
        printf("Stack Underflow - Cannot pop from an empty stack.\n");
        return -1;
    }
}

int main() {
    push(5);
    push(10);
    push(15);
    push(20);
    pop();
    pop();
    pop();
    pop();
    return 0;
}
```

OUTPUT:

```
Pushed 5 to the stack.
Pushed 10 to the stack.
Pushed 15 to the stack.
Stack Overflow - Cannot push to a full stack.
Popped 15 from the stack.
Popped 10 from the stack.
Popped 5 from the stack.
Stack Underflow - Cannot pop from an empty stack.
PS C:\Users\haari\ayaan\College\oop lab>
```

QUEUE IMPLEMENTATION USING ARRAY

ALGORITHM

1. Initialize the Queue:
   a) Create an array to store queue elements.
   b) Initialize front and rear pointers to -1.
2. Enqueue Operation:
   a) Check if the queue is full (rear == array size - 1).
   b) If full, display an overflow message.
   c) If not full:
   d) Increment the rear pointer.
   e) Add the new element at the rear position in the array.
3. Dequeue Operation:
   a) Check if the queue is empty (front > rear).
   b) If empty, display an underflow message.
   c) If not empty:
   d) Increment the front pointer.
   e) Retrieve and return the element at the front position.

SOURCE CODE

```
#include <stdio.h>
#define MAX_SIZE 3
int queue[MAX_SIZE];
int front = -1, rear = -1;
int is_empty() {
    return front == -1;
}
int is_full() {
    return (rear + 1) % MAX_SIZE == front;
}
void enqueue(int item) {
    if (!is_full()) {
        if (is_empty())
            front = rear = 0;
        else
            rear = (rear + 1) % MAX_SIZE;
        queue[rear] = item;
        printf("Enqueued %d to the queue.\n", item);
    } else {
        printf("Queue Overflow - Cannot enqueue to a full queue.\n");
    }
}
int dequeue() {
```

```c
    if (!is_empty()) {
        int item = queue[front];
        if (front == rear)
            front = rear = -1;
        else
            front = (front + 1) % MAX_SIZE;

        printf("Dequeued %d from the queue.\n", item);
        return item;
    } else {
        printf("Queue Underflow - Cannot dequeue from an empty queue.\n");
        return -1; // Assuming -1 is not a valid element in the queue
    }
}
int main() {
    enqueue(1);
    enqueue(2);
    enqueue(3);
    enqueue(4);
    dequeue();
    dequeue();
    dequeue();
    dequeue();
    return 0;
}
```

OUTPUT

```
Enqueued 1 to the queue.
Enqueued 2 to the queue.
Enqueued 3 to the queue.
Queue Overflow - Cannot enqueue to a full queue.
Dequeued 1 from the queue.
Dequeued 2 from the queue.
Dequeued 3 from the queue.
Queue Underflow - Cannot dequeue from an empty queue.
```

EXPERIMENT 4

AIM: Create a linked list and perform following operations:

I. Insert a new node at specified position.

II. Deletion of a node with specified position.

III. Reversal of that linked list.

ALGORITHM

I. Insert a new node at specified position:

A. Create a new node with the given data.
B. Traverse the linked list to the (position - 1)th node.
C. Set the next pointer of the new node to the next pointer of the (position - 1)th node.
D. Set the next pointer of the (position - 1)th node to the new node.

II. Deletion of a node with specified position:

A. Traverse the linked list to the (position - 1)th node.
B. Save the reference to the node to be deleted (position-th node).
C. Update the next pointer of the (position - 1)th node to skip the node to be deleted.
D. Free the memory allocated for the node to be deleted.

III. <u>Reversal of the linked list:</u>

A. Initialize three pointers - current, prev, and next.
B. Start with current pointing to the head of the linked list and
C. prev and next as null.
D. Traverse the linked list:
    a. Set next to the next node of the current node.
    b. Update the next pointer of the current node to point to the prev node.
    c. Move prev and current one step forward.
E. Update the head of the linked list to the prev node.

SOURCE CODE:

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

// Define a Node structure
struct Node {
    int data;
    struct Node *next;
};
// Function to print the linked list
void printList(struct Node *head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
// Function to insert a new node at a specified position
void insertNodeAtPosition(struct Node **head, int data, int position) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        printf("Node with data %d inserted at position %d.\n", data, position);
        return;
    }
    struct Node *temp = *head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Invalid position. Node not inserted.\n");
    } else {
```

```c
        newNode->next = temp->next;
        temp->next = newNode;
        printf("Node with data %d inserted at position %d.\n", data, position);
    }
}
// Function to delete a node at a specified position
void deleteNodeAtPosition(struct Node **head, int position) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete from an empty list.\n");
        return;
    }
    struct Node *temp = *head;
    if (position == 1) {
        *head = temp->next;
        free(temp);
        printf("Node deleted from position %d.\n", position);
        return;
    }
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position. Node not deleted.\n");
    } else {
        struct Node *deletedNode = temp->next;
        temp->next = deletedNode->next;
        free(deletedNode);
        printf("Node deleted from position %d.\n", position);
    }
}
// Function to reverse the linked list
void reverseList(struct Node **head) {
    struct Node *prev = NULL, *current = *head, *next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;
    printf("Linked list reversed.\n");
}
// Function to free the memory allocated for the linked list
void freeList(struct Node **head) {
    struct Node *temp;
    while (*head != NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
int main() {
    // Initialize an empty linked list
    struct Node *head = NULL;
    // Insert nodes
    insertNodeAtPosition(&head, 1, 1);
    insertNodeAtPosition(&head, 2, 2);
    insertNodeAtPosition(&head, 3, 2);
    insertNodeAtPosition(&head, 4, 1);
    // Print the initial linked list
    printf("Initial Linked List: ");
```

```c
    printList(head);
    // Delete a node
    deleteNodeAtPosition(&head, 2);
    printf("Linked List after deletion: ");
    printList(head);
    // Reverse the linked list
    reverseList(&head);
    printf("Linked List after reversal: ");
    printList(head);
    // Free the memory allocated for the linked list
    freeList(&head);
    return 0;
}
```

OUTPUT

```
Node with data 1 inserted at position 1.
Node with data 2 inserted at position 2.
Node with data 3 inserted at position 2.
Node with data 4 inserted at position 1.
Initial Linked List: 4 -> 1 -> 3 -> 2 -> NULL
Node deleted from position 2.
Linked List after deletion: 4 -> 3 -> 2 -> NULL
Linked list reversed.
Linked List after reversal: 2 -> 3 -> 4 -> NULL
PS C:\Users\haari\ayaan\College\oop lab>
```

EXPERIMENT 8

Aim: Implement a Linear Queue using Linked List and Perform following operations : Insert, Delete, and Display the queue elements

ALGORITHM

1. Insert (Enqueue):
   - Create a new node with the given data.
   - If the queue is empty, set both Front and Rear to the new node.
   - Otherwise, set the Next of the current Rear to the new node and update Rear to the new node.

2. Delete (Dequeue):
   - If the queue is empty (Front is null), display an underflow message.
   - Otherwise, remove the node pointed by Front.
   - If the queue becomes empty after deletion, set both Front and Rear to null.
   - Otherwise, update Front to the Next of the removed node.

3. Display:
   - If the queue is empty (Front is null), display an empty message.
   - Otherwise, start from Front and traverse the queue, displaying each element in bullet points.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
// Define a Node structure
struct Node {
    int data;
    struct Node *next;
};
// Define a Queue structure
struct Queue {
```

```c
    struct Node *front, *rear;
};
// Function to initialize an empty queue
void initQueue(struct Queue *q) {
    q->front = q->rear = NULL;
}
// Function to check if the queue is empty
int is_empty(struct Queue *q) {
    return q->front == NULL;
}
// Function to insert an element into the queue
void enqueue(struct Queue *q, int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (is_empty(q)) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
    printf("Enqueued %d to the queue.\n", data);
}
// Function to delete an element from the queue
int dequeue(struct Queue *q) {
    if (is_empty(q)) {
        printf("Queue Underflow - Cannot dequeue from an empty queue.\n");
        return -1; // Assuming -1 is not a valid element in the queue
    }
    struct Node *temp = q->front;
    int data = temp->data;
    if (q->front == q->rear) {
        q->front = q->rear = NULL;
    } else {
        q->front = temp->next;
    }
    free(temp);
    printf("Dequeued %d from the queue.\n", data);
    return data;
}
// Function to display the elements of the queue
void displayQueue(struct Queue *q) {
    if (is_empty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue elements: ");
    struct Node *temp = q->front;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Function to free the memory allocated for the queue
void freeQueue(struct Queue *q) {
    while (!is_empty(q)) {
        dequeue(q);
    }
}
int main() {
```

```c
    // Initialize an empty queue
    struct Queue queue;
    initQueue(&queue);
    // Enqueue elements
    enqueue(&queue, 1);
    enqueue(&queue, 2);
    enqueue(&queue, 3);
    // Display the initial queue
    displayQueue(&queue);
    // Dequeue an element
    dequeue(&queue);
    // Display the queue after dequeue
    displayQueue(&queue);
    // Free the memory allocated for the queue
    freeQueue(&queue);
    return 0;
}
```

OUTPUT

```
Enqueued 1 to the queue.
Enqueued 2 to the queue.
Enqueued 3 to the queue.
Queue elements: 1 2 3
Dequeued 1 from the queue.
Queue elements: 2 3
Dequeued 2 from the queue.
Dequeued 3 from the queue.
```

EXPERIMENT 9

AIM: Create a Binary Tree using linked list (Display using Graphics) and perform the following operations: Tree traversals (Preorder, Postorder, Inorder) using the concept of recursion

ALGORITHM

1. Define Node Structure:

   Create a structure Node with integer data, left child pointer, and right child pointer

   .
2. Create Node Function:

   Implement a function createNode that takes an integer data as a parameter, allocates memory for a new node, initializes its data, left, and right pointers, and returns the new node

   .
3. Tree Traversal Functions:

   a) Implement three functions for tree traversals: preorderTraversal, inorderTraversal, and postorderTraversal.

   b) Each function takes a pointer to a node as a parameter and performs the respective traversal recursively.

   c) For Preorder traversal: Print the data, then recursively traverse the left subtree, and finally recursively traverse the right subtree.

   d) For Inorder traversal: Recursively traverse the left subtree, print the data, and then recursively traverse the right subtree.

   e) For Postorder traversal: Recursively traverse the left subtree, recursively traverse the right subtree, and then print the data.

SOURCE CODE:

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

// Define a Node structure for the binary tree
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
// Function to create a new node
struct Node *createNode(int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
// Function to perform Preorder traversal
void preorderTraversal(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
// Function to perform Inorder traversal
void inorderTraversal(struct Node *root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
// Function to perform Postorder traversal
void postorderTraversal(struct Node *root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}
int main() {
    // Creating a sample binary tree
    struct Node *root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);
    // Display the tree traversals
    printf("Preorder Traversal: ");
```

```
    preorderTraversal(root);
    printf("\n");
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
    return 0;
}
```

OUTPUT:

```
Preorder Traversal: 1 2 4 5 3 6 7
Inorder Traversal: 4 2 5 1 6 3 7
Postorder Traversal: 4 5 2 6 7 3 1
```

EXPERIMENT 10

AIM: Implement Insertion, Deletion and Display (Inorder, Preorder and Postorder) on Binary Search Tree

ALGORITHM

Insertion in BST:

1. Start at the root of the tree.
2. If the tree is empty, create a new node with the given value and make it the root.
3. If the value to be inserted is less than the current node's value, move to the left subtree.
4. If the value to be inserted is greater than the current node's value, move to the right subtree.
5. Repeat steps 3-4 until an appropriate position is found.
6. Insert the new node at that position.

Deletion from BST:

1. Start at the root of the tree.
2. Search for the node to be deleted.
3. If the node has no children, simply remove it.
4. If the node has one child, replace it with its child.
5. If the node has two children, find the node's in-order successor (or predecessor).
6. Replace the node's value with the in-order successor (or predecessor) value.
7. Recursively delete the in-order successor (or predecessor) node.

Inorder Traversal:

1. Traverse the left subtree recursively.
2. Visit the current node.
3. Traverse the right subtree recursively.
4. print the value of each node as it is visited

Preorder Traversal:

1. Visit the current node.
2. Traverse the left subtree recursively.
3. Traverse the right subtree recursively.
4. print the value of each node as it is visited

Postorder Traversal:

1. Traverse the left subtree recursively.
2. Traverse the right subtree recursively.
3. Visit the current node.
4. print the value of each node as it is visited

SOURCE CODE

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

// Define a Node structure for the binary search tree
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
};
// Function to create a new node
struct Node *createNode(int key) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    return newNode;
}
// Function to insert a key into the BST
struct Node *insert(struct Node *root, int key) {
    if (root == NULL) {
        return createNode(key);
    }
```

```
    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;
}
// Function to find the minimum key in a BST
struct Node *findMin(struct Node *root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}
// Function to delete a key from the BST
struct Node *deleteNode(struct Node *root, int key) {
    if (root == NULL) {
        return root;
    }
    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == NULL) {
            struct Node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node *temp = root->left;
            free(root);
            return temp;
        }
        struct Node *temp = findMin(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
// Function to perform Inorder traversal
void inorderTraversal(struct Node *root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->key);
        inorderTraversal(root->right);
    }
}
// Function to perform Preorder traversal
void preorderTraversal(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
// Function to perform Postorder traversal
void postorderTraversal(struct Node *root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
```

```c
        printf("%d ", root->key);
    }
}
int main() {
    struct Node *root = NULL;
    // Insert keys into the BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    // Display the BST using different traversals
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
    // Delete a key from the BST
    int keyToDelete = 30;
    root = deleteNode(root, keyToDelete);
    printf("BST after deleting key %d:\n", keyToDelete);
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");
    return 0;
}
```

OUTPUT:

```
Inorder Traversal: 20 30 40 50 60 70 80
Preorder Traversal: 50 30 20 40 70 60 80
Postorder Traversal: 20 40 30 60 80 70 50
BST after deleting key 30:
Inorder Traversal: 20 40 50 60 70 80
```

EXPERIMENT 11

AIM: To implement Sorting techniques using array. (Insertion sort, Merge sort, Quick sort, Bubble sort, Bucket sort, Radix sort, Shell sort, Selection sort, Heap sort and Exchange sort)

INSERTION SORT

SOURCE CODE:

```
#include <stdio.h>

// Function to perform Insertion Sort on an array
void insertionSort(int arr[], int size) {
    int i, key, j;
    for (i = 1; i < size; 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 an 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 size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    // Perform Insertion Sort
    insertionSort(arr, size);
    printf("Sorted Array (using Insertion Sort): ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 12 11 13 5 6
Sorted Array (using Insertion Sort): 5 6 11 12 13
```

MERGE SORT

SOURCE CODE

```c
#include <stdio.h>

void merge(int arr[], int left, int middle, int right) {
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
    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) {
```

```c
        int middle = left + (right - left) / 2;
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}
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, 7};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    mergeSort(arr, 0, size - 1);
    printf("Sorted Array: ");
    printArray(arr, size);
    return 0;
}
```

<u>OUTPUT</u>

```
Original Array: 12 11 13 5 6 7
Sorted Array: 5 6 7 11 12 13
```

QUICK SORT

SOURCE CODE:

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; 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);
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    quickSort(arr, 0, size - 1);
    printf("Sorted Array: ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 64 25 12 22 11
Sorted Array: 11 12 22 25 64
```

BUBBLE SORT

SOURCE CODE:

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
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 size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    bubbleSort(arr, size);
    printf("Sorted Array (Bubble Sort): ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 64 34 25 12 22 11 90
Sorted Array (Bubble Sort): 11 12 22 25 34 64 90
```

BUCKET SORT

SOURCE CODE:

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

struct Node {
    int data;
    struct Node* next;
};
void insertionSort(struct Node** bucket) {
    struct Node* current;
    struct Node* prev;
    struct Node* temp;
    for (int i = 0; i < 10; i++) {
        if (bucket[i] != NULL) {
            current = bucket[i]->next;
            prev = bucket[i];
            while (current != NULL) {
                if (current->data < prev->data) {
                    prev->next = current->next;
                    current->next = bucket[i];
                    bucket[i] = current;
                    current = prev;
                }
                prev = current;
                current = current->next;
            }
        }
    }
}
void bucketSort(int arr[], int size) {
    struct Node* bucket[10];
    for (int i = 0; i < 10; i++) {
        bucket[i] = NULL;
    }
    for (int i = 0; i < size; i++) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
        newNode->data = arr[i];
        newNode->next = NULL;
        int index = arr[i] / 10;
        if (bucket[index] == NULL) {
            bucket[index] = newNode;
        } else {
            struct Node* current = bucket[index];
            while (current->next != NULL) {
                current = current->next;
            }
            current->next = newNode;
        }
    }
    insertionSort(bucket);
```

```c
    int index = 0;
    for (int i = 0; i < 10; i++) {
        struct Node* current = bucket[i];
        while (current != NULL) {
            arr[index++] = current->data;
            current = current->next;
        }
    }
    for (int i = 0; i < 10; i++) {
        struct Node* current = bucket[i];
        while (current != NULL) {
            struct Node* temp = current;
            current = current->next;
            free(temp);
        }
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    bucketSort(arr, size);
    printf("Array after Bucket Sort: ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 64 25 12 22 11
Array after Bucket Sort: 11 12 22 25 64
```

RADIX SORT

SOURCE CODE:

```
#include <stdio.h>

void countingSort(int arr[], int size, int exp) {
    int output[size];
    int count[10] = {0};
    for (int i = 0; i < size; i++)
        count[(arr[i] / exp) % 10]++;
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (int i = size - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (int i = 0; i < size; i++)
        arr[i] = output[i];
}
void radixSort(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++)
        if (arr[i] > max)
            max = arr[i];
    for (int exp = 1; max / exp > 0; exp *= 10)
        countingSort(arr, size, exp);
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    radixSort(arr, size);
    printf("Array after Radix Sort: ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 170 45 75 90 802 24 2 66
Array after Radix Sort: 2 24 45 66 75 90 170 802
```

SHELL SORT

SOURCE CODE:

```
#include <stdio.h>

void shellSort(int arr[], int size) {
    for (int gap = size / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < size; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
    }
}
int main() {
    int arr[] = {12, 34, 54, 2, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
    shellSort(arr, size);
    printf("Sorted Array: ");
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

OUTPUT

```
Original Array: 12 34 54 2 3
Sorted Array: 2 3 12 34 54
```

SELECTION SORT

SOURCE CODE:

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++)
            if (arr[j] < arr[minIndex])
                minIndex = j;
        swap(&arr[minIndex], &arr[i]);
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    selectionSort(arr, size);
    printf("Array after Selection Sort: ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 64 25 12 22 11
Array after Selection Sort: 11 12 22 25 64
```

HEAP SORT

SOURCE CODE:

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void heapify(int arr[], int size, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < size && arr[left] > arr[largest])
        largest = left;
    if (right < size && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, size, largest);
    }
}
void heapSort(int arr[], int size) {
    for (int i = size / 2 - 1; i >= 0; i--)
        heapify(arr, size, i);
    for (int i = size - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
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, 7};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    heapSort(arr, size);
    printf("Sorted Array using Heap Sort: ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 12 11 13 5 6 7
Sorted Array using Heap Sort: 5 6 7 11 12 13
```

EXCHANGE SORT

SOURCE CODE:

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void exchangeSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] > arr[j]) {
                swap(&arr[i], &arr[j]);
            }
        }
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: ");
    printArray(arr, size);
    exchangeSort(arr, size);
    printf("Array after Exchange Sort: ");
    printArray(arr, size);
    return 0;
}
```

OUTPUT

```
Original Array: 64 25 12 22 11
Array after Exchange Sort: 11 12 22 25 64
```