Experiment no. 1

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

CODE

```
#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;
}
```

<u>OUTPUT</u>

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
Stack overflow
Top element is 30
Stack underflow
```

Experiment no. 3

<u>CODE</u>

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

// 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);

    // Corrected line to print the top element
    printf("Top element is %d\n", peek(&stack));

    return 0;
}
```

OUTPUT

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
Top element is 30
```

Experiment no. 4

<u>CODE</u>

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

typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
} Queue;

void initQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(Queue *q) {
    return (q->front == -1 && q->rear == -1);
}

int isFull(Queue *q) {
    return (q->rear == MAX_SIZE - 1);
}

void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->rear++;
    q->items[q->rear] = value;
}

int dequeue(Queue *q) {
    int removedItem;
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }
    removedItem = q->items[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front++;
    }
    return removedItem;
}

int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No front element.\n");
        return -1;
    }
    return q->items[q->front];
}

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);
    printf("Front element: %d\n", front(&q));
    dequeue(&q);
    display(&q);
    return 0;
}
```

OUTPUT

```
Queue elements: 10 20 30
Front element: 10
Queue elements: 20 30
```

Experiment no. 5

CODE

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

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

typedef struct {
   Node *front;
   Node *rear;
} Queue;

void initQueue(Queue *q) {
   q->front = NULL;
   q->rear = NULL;
}

int isEmpty(Queue *q) {
   return (q->front == NULL);
}

void enqueue(Queue *q, int value) {
   Node *newNode = (Node *)malloc(sizeof(Node));
   if (newNode == NULL) {
      printf("Memory allocation failed.\n");
      return;
   }
   newNode->data = value;
   newNode->next = NULL;
   if (isEmpty(q)) {
      q->front = newNode;
      q->rear = newNode;
   } else {
      q->rear->next = newNode;
      q->rear = newNode;
   }
}

int dequeue(Queue *q) {
   if (isEmpty(q)) {
      printf("Queue is empty. Cannot dequeue.\n");
      return -1;
   }
   Node *temp = q->front;
   int removedItem = temp->data;
   q->front = q->front->next;
```

```
    free(temp);
    if (q->front == NULL) {
        q->rear = NULL;
    }
    return removedItem;
}

int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No front element.\n");
        return -1;
    }
    return q->front->data;
}

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");
}

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);
    printf("Dequeued item: %d\n", dequeue(&q));
    display(&q);
    printf("Front item: %d\n", front(&q));
    freeQueue(&q);
    return 0;
}
```

OUTPUT

```
Queue elements: 10 20 30
Dequeued item: 10
Queue elements: 20 30
Front item: 20
```

Experiment no. 6

CODE

```
#include <stdio.h>

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 -1;
}

int binarySearch(int arr[], int left, int right, int key, int *comparisons) {
    while (left <= right) {
        (*comparisons)++;
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) {
            return mid;
        }
        if (arr[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

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

    int comparisons_linear = 0;
    int result_linear = linearSearch(arr, n, key, &comparisons_linear);

    int comparisons_binary = 0;
    int result_binary = binarySearch(arr, 0, n - 1, key, &comparisons_binary);

    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
```

Experiment no. 7

CODE

```
#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;
}
```

OUTPUT

```
Original array:
64 34 25 12 22 11 90
Sorted array in ascending order:
11 12 22 25 34 64 90
```

Experiment no. 8

CODE

```
#include <stdio.h>

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;
   }
}

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);

   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
```

Experiment no. 9

CODE

```
#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: 5
Enter elements:
2 4 1 6 8
Sorted array:
1 2 4 6 8
```

Experiment no. 10 { **MERGE SORT** }
<u>CODE</u>

```
#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;
}
```

<u>OUTPUT</u>

```
Enter number of elements: 5
Enter elements:
3 6 4 9 1
Sorted array:
1 3 4 6 9
```

Experiment no. 10 { **QUICK SORT:** }
CODE

```
#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: 5
Enter elements:
2 6 4 9 0
Sorted array:
0 2 4 6 9
```

Experiment no. 11

<u>CODE</u>

```
#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
```

Experiment no. 12

CODE

```
#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
```

Experiment no. 13

CODE

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

#define MAX 100

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;
}
```

<u>OUTPUT</u>

```
Enter the number of vertices: 3
Enter the adjacency matrix:
2 4 6 1 3 7 8 5 9
Enter the starting vertex: 3
BFS Traversal starting from vertex 3: 3
```

Experiment no. 14

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;
}
```

OUTPUT

```
Index 0: EMPTY
Index 1: EMPTY
Index 2: 12
Index 3: 22
Index 4: 32
Index 5: EMPTY
Index 6: EMPTY
Index 7: EMPTY
Index 8: EMPTY
Index 9: EMPTY
Key 22 found at index 3
```

Experiment no. 15

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");
    if (file == NULL) {
        perror("Error opening file for writing");
        return;
    }
    fwrite(&student, sizeof(Student), 1, file);
    fclose(file);
}

int searchRecordSequential(const char *filename, int id, Student *student) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        perror("Error opening file for reading");
        return 0;
    }
    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;

    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);
    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;
}
```

<u>OUTPUT</u>

```
Record found: ID=2, Name=Bob, Age=21
```

Experiment no. 16 { *Prim's Algorithm* }

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;
}
```

OUTPUT

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Experiment no. 16{ *Kruskal's Algorithm* }

<u>CODE</u>

```
#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;
}
```

<u>OUTPUT</u>

```
Following are the edges in the constructed MST
0 -- 1 == 2
1 -- 2 == 3
1 -- 4 == 5
0 -- 3 == 6
```

Experiment no. 16 { *Dijkstra's Algorithm* }

<u>CODE</u>

```
#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;

}
```

OUTPUT

```
Vertex    Distance from Source
0         0
1         10
2         15
3         17
4         16
```