Week - 5

PRACTICAL 1 : Write a program to implement Circular Queue using arrays that performs following operations:

a) INSERT
b) DELETE
c) DISPLAY

ALGORITHM :

1) Initialise an empty array with a fixed size and set front and rear pointers to -1.

2) INSERT operation:
a. Check if the queue is full by comparing (rear + 1) % array_size with front. b. If the queue is full, display an overflow message.
c. If the queue is not full:
i. Set rear to (rear + 1) % array_size.
ii. Read the element to be inserted.
iii. Store the element at the rear position in the array.

3) DELETE operation:
a. Check if the queue is empty by comparing front with rear.
b. If the queue is empty, display an underflow message.
c. If the queue is not empty:
i. Retrieve and display the element at the front position from the array. ii. Set front to (front + 1) % array_size.

4) DISPLAY operation:
a. Check if the queue is empty by comparing front with rear.
b. If the queue is empty, display a message indicating that the queue is empty. c. If the queue is not empty:
i. Set current to front.
ii. Start a loop from current to rear.
iii. Retrieve and display each element at the current position in the array.
iv. Set current to (current + 1) % array_size.

5) Repeat steps 2-4 as needed.

PROGRAM :

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10 // Maximum size of the queue void insert(int element);
void delete();
void display();
int front = -1, rear = -1;int queue[MAX_SIZE];
```

```c
int main()
{
   int choice, element;
   while (1)
   {
       printf("\nCircular Queue Operations\n");
       printf("1. INSERT\n");
       printf("2. DELETE\n");
       printf("3. DISPLAY\n");
       printf("4. QUIT\n");
       printf("\nEnter your choice: ");
       scanf("%d", &choice);
       switch (choice)
       {
       case 1:
           printf("\nEnter the element to be inserted: ");
           scanf("%d", &element);
           insert(element);
           break;
       case 2:
           delete ();
           break;
       case 3:
           display();
           break;
       case 4:
           exit(0);
       default:
           printf("\nInvalid choice!");
       }
   }
   return 0;
}
void insert(int element)
{
   if ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1))
   {
       printf("\nQueue is full!");
   }
   else if (front == -1 && rear == -1)
```

```
    {
        front = rear = 0;
        queue[rear] = element;
        printf("\nElement %d is inserted into the queue", element);
    }
    else if (rear == MAX_SIZE - 1 && front != 0)
    {
        rear = 0;
        queue[rear] = element;
        printf("\nElement %d is inserted into the queue", element);
    }
    else
    {
        rear++;
        queue[rear] = element;
        printf("\nElement %d is inserted into the queue", element);
    }
}
void delete()
{
    if (front == -1 && rear == -1)
    {
        printf("\nQueue is empty!");
    }
    else if (front == rear)
    {
        printf("\nElement %d is deleted from the queue", queue[front]);
        front = rear = -1;
    }
    else if (front == MAX_SIZE - 1)
    {
        printf("\nElement %d is deleted from the queue", queue[front]);
        front = 0;
    }
    else
    {
        printf("\nElement %d is deleted from the queue", queue[front]);
        front++;
    }
}
```

```c
void display()
{
   int i;
   if (front == -1 && rear == -1)
   {
       printf("\nQueue is empty!");
   }
   else
   {
       printf("\nElements of the queue are: ");
       if (rear >= front)
       {
           for (i = front; i <= rear; i++)
           {
               printf("%d ", queue[i]);
           }
       }
       else
       {
           for (i = front; i < MAX_SIZE; i++)
           {
               printf("%d ", queue[i]);
           }
           for (i = 0; i <= rear; i++)
           {
               printf("%d ", queue[i]);
           }
       }
   }
}
```

OUTPUT :

Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 1
Enter the element to be inserted: 10

Element 10 is inserted into the queue Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 1
Enter the element to be inserted: 20
Element 20 is inserted into the queue is Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 3
Elements of the queue are: 10
Element 20 is inserted into the queue is Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 2
Element 10 is deleted from the queue Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 3
Elements of the queue are: 20 Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 1
Enter the element to be inserted: 30
Element 30 is inserted into the queue Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 1
Enter the element to be inserted: 40
Element 40 is inserted into the queue Circular Queue Operations

1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 3
Elements of the queue are: 20 30 40 Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 2
Element 20 is deleted from the queue Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 3
Elements of the queue are: 30 40 Circular Queue Operations
1. INSERT
2. DELETE
3. DISPLAY
4. QUIT
Enter your choice: 4

PRACTICAL 2 : Write a menu driven program to implement following operations on singly linked lists.

a) Insert a node at the front of the linked list
b) Insert a node at the end of the linked list.
c) Insert a node such that the linked list is in ascending order.
d) Delete the first node of a linked list.
e) Delete a node before specified position.
f) Delete a node after specified position.

ALGORITHM :

1) Define the Node class:
Create a Node class with two attributes: data to store the value of the node and next to store the reference to the next node.

Define a constructor to initialise the data and next attributes.

2) Initialise the linked list:
a. Create a Linked List class with a head attribute initialised as None. b. Define a constructor that sets the head to None.

3) Insert a node at the front of the linked list:
a. Create a new node with the given value.
b. Set the next pointer of the new node to the current head. c. Set the head pointer to the new node.

4) Insert a node at the end of the linked list:
a. Create a new node with the given value.
b. If the linked list is empty (head is None), set the head to the new node.
c. Otherwise, traverse to the last node using a loop until the next pointer is None.
d. Set the next pointer of the last node to the new node.

5) Insert a node such that the linked list is in ascending order:
a. Create a new node with the given value.
b. If the linked list is empty (head is None) or the value of the new node is less than the value of the head:

6) Set the next pointer of the new node to the current head.

7) ii. Set the head pointer to the new node.
Otherwise, traverse the linked list using a loop until finding the appropriate position to insert the new node.
Set the next pointer of the new node to the next pointer of the previous node.
Set the next pointer of the previous node to the new node.

8) Delete the first node of the linked list:
a. If the linked list is empty (head is None), display an underflow message. b. Otherwise, set the head to the next pointer of the current head.

9) Delete the node before the specified position:
a. If the linked list is empty (head is None) or the position is less than or equal to 1, display an error message.
Traverse the linked list using a loop until reaching the node before the specified position.
Set the next pointer of the previous node to the node after the specified Position.

10) Delete the node after the specified position:
If the linked list is empty (head is None) or the position is greater than or equal to the length of the linked list, display an error message.
Traverse the linked list using a loop until reaching the node at the specified position.
Set the next pointer of the current node to the node after the next node.

PROGRAM :

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

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

void printList(struct Node *head)
{
   printf("Linked List: ");
   while (head != NULL)
   {
       printf("%d ", head->data);
       head = head->next;
   }
   printf("\n");
}
struct Node *insertFront(struct Node *head, int newData)
{
   struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
   newNode->data = newData;
   newNode->next = head;
   head = newNode;
   printf("Node with data %d is inserted at the front of the list.\n",
newData);
   return head;
}
struct Node *insertEnd(struct Node *head, int newData)
{
   struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
   newNode->data = newData;
   newNode->next = NULL;
   if (head == NULL)
   {
       head = newNode;
   }
   else
   {
       struct Node *current = head;
```

```
        while (current->next != NULL)
        {
            current = current->next;
        }
        current->next = newNode;
    }
    printf("Node with data %d is inserted at the end of the list.\n",
newData);
    return head;
}
struct Node *insertAscending(struct Node *head, int newData)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = newData;
    if (head == NULL || newData < head->data)
    {
        newNode->next = head;
        head = newNode;
    }
    else
    {
        struct Node *current = head;
        while (current->next != NULL && current->next->data < newData)
        {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
    printf("Node with data %d is inserted in ascending order.\n", newData);
    return head;
}
struct Node *deleteFront(struct Node *head)
{
    if (head == NULL)
    {
        printf("Linked list is already empty.\n");
    }
    else
    {
```

```
        struct Node *temp = head;
        head = head->next;
        free(temp);
        printf("First node is deleted from the linked list.\n");
    }
    return head;
}
struct Node *deleteBefore(struct Node *head, int pos)
{
    if (head == NULL)
    {
        printf("Linked list is empty.\n");
    }
    else if (pos <= 1)
    {
        printf("Position should be greater than 1.\n");
    }
    else
    {
        struct Node *current = head;
        struct Node *prev = NULL;
        int i = 1;
        while (current != NULL && i < pos - 1)
        {

            prev = current;
            current = current->next;
            i++;
        }
        if (prev == NULL)
        {
            head = head->next;
            free(current);
        }
        else if (current == NULL)
        {
            printf("Position is beyond the end of the linked list.\n");
        }
        else
        {
```

```
            prev->next = current->next;
            free(current);
        }
        printf("Node before position %d is deleted from the linked list.");
    }
}
```

WEEK - 6

PRACTICAL 1 : Write a program to implement stack using linked list.

ALGORITHM :

1) Define the Node class:

- **Create a Node class with two attributes: data to store the value of the node and next to store the reference to the next node.**
- **Define a constructor to initialise the data and next attributes.**

2) Initialise the stack:

- **Create a LinkedList class with a top attribute initialised as None.**
- **Define a constructor that sets the top to None.**

3) PUSH operation:

- **Create a new node with the given value.**
- **Set the next pointer of the new node to the current top.**
- **Set the top pointer to the new node.**

4) POP operation:

- **Check if the stack is empty by checking if top is None.**
- **If the stack is empty, display an underflow message.**
- **If the stack is not empty:**
- **5) Retrieve the value of the top node.**
- **6) ii. Set the top pointer to the next node.**
- **7) iii. Return the retrieved value.**
- **8) PEEK operation:**

a. Check if the stack is empty by checking if the top is None.

- **If the stack is empty, display an underflow message.**
- **If the stack is not empty, return the value of the top node.**

9) Display the stack:

- **Check if the stack is empty by checking if the top is None.**
- **If the stack is empty, display a message indicating that the stack is empty.**
- **If the stack is not empty, start from the top node and traverse the stack using a loop.**
- **Display the value of each node as you traverse the stack.**

PROGRAM :

22BT04075

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

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

struct stack
{
   struct node *top;
};
struct stack *createStack()
{
   struct stack *newStack = (struct stack *)malloc(sizeof(struct stack));
   newStack->top = NULL;
   return newStack;
}
int isEmpty(struct stack *myStack)
{
   return myStack->top == NULL;
}
void push(struct stack *myStack, int data)
{
   // Create a new node
   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->next = NULL;
   if (isEmpty(myStack))
   {
       myStack->top = newNode;
   }
   else
   {
       newNode->next = myStack->top;
       myStack->top = newNode;
   }
}
int pop(struct stack *myStack)
{
   if (isEmpty(myStack))
   { // The stack is empty
       return -1;
   }
```

```
    int data = myStack->top->data;
    struct node *temp = myStack->top;
    myStack->top = myStack->top->next;
    free(temp);
    return data;
}
int peek(struct stack *myStack)
{
    if (isEmpty(myStack))
    { // The stack is empty
        return -1;
    }
    return myStack->top->data;
}
int main()
{
    struct stack *myStack = createStack();
    push(myStack, 1);
    push(myStack, 2);
    push(myStack, 3);
    push(myStack, 4);
    push(myStack, 5);
    printf("Elements of the stack: ");
    while (!isEmpty(myStack))
    {
        printf("%d ", pop(myStack));
    }
    printf("\n");
    return 0;
}
```

OUTPUT :

Elements of the stack: 5 4 3 2 1

WEEK - 7

PRACTICAL 1 : Write a program to implement following operations on the doubly linked list using C language a) Insert a node at the front of the linked list.

b) Insert a node at the end of the linked list.
c) Delete a node of the linked list.
d) Delete a node before specified position.

ALGORITHM :

1) Define the Node structure:
a. Create a structure named Node with three members: data to store the value of the node, prev to store the pointer to the previous node, and next to store the pointer to the next node.
2) Initialise the doubly linked list:
a. Create a structure named LinkedList with two members: head and tail, both initialised to NULL.
3) Insert a node at the front of the linked list:

- **Create a new node with the given value.**
- **Set the prev pointer of the new node to NULL.**
- **Set the next pointer of the new node to the current head.**
- **If the head is not NULL, set the prev pointer of the current head to the new node. e. Set the head pointer to the new node.**
- **If the tail is NULL, set the tail pointer to the new node.**

4) Insert a node at the end of the linked list:

- **Create a new node with the given value.**
- **Set the prev pointer of the new node to the current tail.**
- **Set the next pointer of the new node to NULL.**
- **If the tail is not NULL, set the next pointer of the current tail to the new node. e. Set the tail pointer to the new node.**
- **If the head is NULL, set the head pointer to the new node.**

5) Delete a node of the linked list:

- **Check if the linked list is empty by checking if the head is NULL.**
- **If the linked list is empty, display an underflow message.**
- **If the linked list is not empty:**

6) If the node to be deleted is the head node, set the head pointer to the next node.

7) ii. If the node to be deleted is the tail node, set the tail pointer to the previous node.

8) iii. Update the next pointer of the previous node to the next node.

9) iv. Update the prev pointer of the next node to the previous node.

10) v. Free the memory of the node to be deleted.

11) Delete a node before the specified position:

- **Check if the linked list is empty by checking if the head is NULL.**
- **If the linked list is empty, display an underflow message.**
- **If the linked list is not empty:**

12) Traverse the linked list until reaching the node at the specified position.
13) ii. If the node at the specified position is the head node, display an error message as there is no node before the head.
14) iii. If the node at the specified position is the second node, set the next pointer of the head to the next node.
15) iv. Otherwise, set the next pointer of the previous node to the node after the specified position.
16) v. Update the previous pointer of the next node to the previous node.
17) vi. Free the memory of the node before the specified position.

PROGRAM :

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

struct node {
int data;
struct node *next;
struct node *prev;
};

struct node *createNode(int data)
{
   struct node *newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->next = NULL;
   newNode->prev = NULL;
   return newNode;
}
void insertFront(struct node **headRef, int data)
{
   struct node *newNode = createNode(data);
   newNode->next = *headRef;
   if (*headRef != NULL)
   {
        (*headRef)->prev = newNode;
   }
   *headRef = newNode;
}
void insertEnd(struct node **headRef, int data)
{
   struct node *newNode = createNode(data);
   struct node *current = *headRef;
   if (*headRef == NULL)
```

```
    {
        *headRef = newNode;
        return;
    }
    while (current->next != NULL)
    {
        current = current->next;
    }
    current->next = newNode;
    newNode->prev = current;
}
void deleteNode(struct node **headRef, struct node *delNode)
{
    if (*headRef == NULL || delNode == NULL)
    {
        return;
    }
    if (*headRef == delNode)
    {
        *headRef = delNode->next;
    }
    if (delNode->next != NULL)
    {
        delNode->next->prev = delNode->prev;
    }
    if (delNode->prev != NULL)
    {
        delNode->prev->next = delNode->next;
    }
    free(delNode);
}
void deleteNodeBefore(struct node **headRef, int position)
{
    if (*headRef == NULL)
    {
        return;
    }
    struct node *current = *headRef;
    int count = 1;
    while (current != NULL && count < position)
    {
        current = current->next;
        count++;
    }
    if (current == NULL || current->prev == NULL)
```

```
    {
        return;
    }
    deleteNode(headRef, current->prev);
}
void printList(struct node *head)
{
    printf("Doubly linked list: ");
    while (head != NULL)
    {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
int main()
{
    struct node *head = NULL;
    insertFront(&head, 3);
    insertFront(&head, 2);
    insertFront(&head, 1);
    printList(head);
    insertEnd(&head, 4);
    insertEnd(&head, 5);
    insertEnd(&head, 6);
    printList(head);
    deleteNode(&head, head->next->next);

    printList(head);
    deleteNodeBefore(&head, 3);
    printList(head);
    return 0;
}
```

OUTPUT :

Doubly linked list: 1 2 3
Doubly linked list: 1 2 3 4 5 6
Doubly linked list: 1 2 4 5 6
Doubly linked list: 1 4 5 6

WEEK - 8

PRACTICAL - 1 : Write a program to implement linear search.

ALGORITHM :

1) Start from the beginning of the array or list.
2) Compare the target value with the current element.
a. If they match, return the current position.
b. If they don't match, move to the next element.
3) Repeat step 2 until a match is found or the end of the array or list is reached. 4) If the end of the array or list is reached without finding a match, return a value indicating that the target value was not found.

PROGRAM :

```c
#include <stdio.h>
int linearSearch(int arr[], int n, int x)
{
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == x)
        {
            return i;
        }
    }
    return -1;
}
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 5;
    int result = linearSearch(arr, n, x);
    if (result == -1)
    {
        printf("Element not found in array\n");
    }
    else
    {
        printf("Element found at index %d\n", result);
```

```
    }
    return 0;
}
```

OUTPUT :

Element found at index 4

PRACTICAL - 2 : Write a Program to implement Binary Search.

ALGORITHM :

1) Initialise two variables, start and end, to represent the range of the search space. Initially, start is set to the first element's index (0) and end is set to the last element's index.
2) Repeat the following steps until start is less than or equal to end:
a. Calculate the middle index as mid using the formula: mid = (start + end) / 2. b. Compare the target value with the element at index mid.
i. If they match, return mid as the position of the target value.
ii. If the target value is less than the element at index mid, set end to mid - 1 to search in the left half of the array.
iii. If the target value is greater than the element at index mid, set start to mid + 1 to search in the right half of the array.
3) If the entire array has been traversed and the target value is not found, return a value indicating that the target value was not found.

PROGRAM :

```
#include <stdio.h>
int binarySearch(int arr[], int n, int x)
{
    int left = 0;
    int right = n - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (arr[mid] == x)
        {
            return mid;
        }
        else if (arr[mid] < x)
        {
```

```
            left = mid + 1;
        }
        else
        {
            right = mid - 1;
        }
    }
    return -1;
}
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 5;
    int result = binarySearch(arr, n, x);
    if (result == -1)
    {
        printf("Element not found in array\n");
    }
    else
    {
        printf("Element found at index %d\n", result);
    }
}
```

OUTPUT :

Element found at index 4.

WEEK - 9

PRACTICAL - 1 : Write a program to implement, Bubble sort, Merge sort, Quick sort.

ALGORITHM :

1) Bubble Sort: 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.

2) sql

3) Merge Sort: Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts the two halves, and then merges them back together to obtain a sorted array.
4) Quick Sort: Quick Sort is another divide-and-conquer algorithm that selects a pivot element, partitions the array around the pivot, and recursively sorts the two sub- arrays on either side of the pivot.

PROGRAM :

Bubble Short -

```c
#include <stdio.h>
   void bubble_sort(int arr[], int n)
{
   int i, j, temp;
   for (i = 0; i < n - 1; i++)
   {
       for (j = 0; j < n - i - 1; j++)
       {
           if (arr[j] > arr[j + 1])
           {
               temp = arr[j];
               arr[j] = arr[j + 1];
               arr[j + 1] = temp;
           }
       }
   }
}
int main()
{
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   bubble_sort(arr, n);
   printf("Sorted array: \n");

   for (int i = 0; i < n; i++)
       printf("%d ", arr[i]);
}
```

Merge Short -

```c
#include <stdio.h>
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;
   int L[n1], R[n2];
```

```
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    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 merge_sort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main()
```

```
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    merge_sort(arr, 0, n - 1);

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

Quick Short -

```
#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 - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quick_sort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
        int main()
        {
            int arr[] = {64, 34, 25, 12, 22, 11, 90};
```

```
            int n = sizeof(arr) / sizeof(arr[0]);
            quick_sort(arr, 0, n - 1);
            printf("Sorted array: \n");
            for (int i = 0; i < n; i++)
                printf("%d ", arr[i]);
        }
    }
}
```

WEEK - 10

PRACTICAL - 1 : Write a program to create a binary search tree Using C language.

ALGORITHM :

1) Define the structure for a BST node:
a. Create a structure named Node with three members: data to store the value of the node, left to store the pointer to the left child node, and right to store the pointer to the right child node.
2) Create a function to allocate memory for a new node and initialise it:

- **Create a function named create Node that takes an integer value as a parameter.**
- **Allocate memory for a new node using malloc or calloc.**
- **Set the data member of the new node to the given value.**
- **Set the left and right members of the new node to NULL.**
- **Return the pointer to the new node.**

3) Create a function to insert a node into the BST:

- **Create a function named insertNode that takes two parameters: a pointer to the root of the BST and an integer value to be inserted.**
- **If the root is NULL, create a new node using the createNode function and set it as the root.**
- **If the value to be inserted is less than the value of the current node, recursively call insertNode with the left child of the current node as the new root.**
- **If the value to be inserted is greater than the value of the current node, recursively call insertNode with the right child of the current node as the new root.**
- **Repeat steps b-d until an appropriate position to insert the new node is found.**
- **Return the modified root of the BST.**

4) Create a function to traverse the BST in-order (left-root-right):

- **Create a function named inorder Traversal that takes a pointer to the root of the BST.**
- **If the root is not NULL:**
  5) Recursively call inorder Traversal with the left child of the root as the new root.

6) **ii. Print the value of the root. iii. Recursively call inorder Traversal with the right child of the root as the new root.**

7) **Create a main function to test the BST :**

- **Declare a variable root to store the root of the BST and initialise it to NULL.**
- **Insert nodes into the BST using the insert Node function.**
- **Traverse the BST in-order using the inorder Traversal function to display the sorted values.**

PROGRAM :

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

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

struct node *new_node(int data)
{
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   temp->data = data;
   temp->left = temp->right = NULL;
   return temp;
}
struct node *insert(struct node *root, int data)
{
   if (root == NULL)
   {
       return new_node(data);
   }
   if (data < root->data)
   {
       root->left = insert(root->left, data);
   }
   else if (data > root->data)
   {
       root->right = insert(root->right, data);
   }
   return root;
}
void inorder_traversal(struct node *root)
{
```

```
    if (root != NULL)
    {
        inorder_traversal(root->left);
        printf("%d ", root->data);
        inorder_traversal(root->right);
    }
}
int main()
{
    struct node *root = NULL;
    int n, data;
    printf("Enter the number of elements to be inserted: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &data);
        root = insert(root, data);
    }
    printf("Inorder traversal of the BST: ");
    inorder_traversal(root);
}
```

OUTPUT :

Enter the number of elements to be inserted: 2
Enter the elements: 3 2
Inorder traversal of the BST: 2 3

WEEK - 11

PRACTICAL - 1 : Implement recursive and non-recursive tree traversing methods in order, preorder and post order.

PROGRAM :

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

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

struct node *new_node(int data)
{
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   temp->data = data;
   temp->left = temp->right = NULL;
   return temp;
}
void inorder_recursive(struct node *root)
{
   if (root != NULL)
   {
       inorder_recursive(root->left);
       printf("%d ", root->data);
       inorder_recursive(root->right);
   }
}
void inorder_nonrecursive(struct node *root)
{
   struct node *stack[100];
   int top = -1;
   while (root != NULL || top != -1)
   {
       while (root != NULL)
       {
           stack[++top] = root;
           root = root->left;
       }
       root = stack[top--];
       printf("%d ", root->data);
```

```
        root = root->right;
    }
}
void preorder_recursive(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->data);
        preorder_recursive(root->left);
        preorder_recursive(root->right);
    }
}
void preorder_nonrecursive(struct node *root)
{
    struct node *stack[100];
    int top = -1;
    stack[++top] = root;
    while (top != -1)
    {
        root = stack[top--];
        printf("%d ", root->data);
        if (root->right != NULL)
        {
            stack[++top] = root->right;
        }
        if (root->left != NULL)
        {
            stack[++top] = root->left;
        }
    }
}
void postorder_recursive(struct node *root)
{
    if (root != NULL)
    {
        postorder_recursive(root->left);
        postorder_recursive(root->right);
        printf("%d ", root->data);
    }
}
void postorder_nonrecursive(struct node *root)
{
    struct node *stack1[100];

    struct node *stack2[100];
```

```
    int top1 = -1;
    int top2 = -1;
    stack1[++top1] = root;
    while (top1 != -1)
    {
        root = stack1[top1--];
        stack2[++top2] = root;
        if (root->left != NULL)
        {
            stack1[++top1] = root->left;
        }
        if (root->right != NULL)
        {
            stack1[++top1] = root->right;
        }
    }
    while (top2 != -1)
    {
        root = stack2[top2--];
        printf("%d ", root->data);
    }
}
int main()
{
    struct node *root = NULL;
    root = new_node(1);
    root->left = new_node(2);
    root->right = new_node(3);
    root->left->left = new_node(4);
    root->left->right = new_node(5);
    printf
        printf("Inorder traversal (recursive): ");
    inorder_recursive(root);
    printf("\n");
    printf("Inorder traversal (non-recursive): ");
    inorder_nonrecursive(root);
    printf("\n");
    printf("Preorder traversal (recursive): ");
    preorder_recursive(root);
    printf("\n");
    printf("Preorder traversal (non-recursive): ");
    preorder_nonrecursive(root);
    printf("\n");
    printf("Postorder traversal (recursive): ");
    postorder_recursive(root);
```

```
    printf("\n");
    printf("Postorder traversal (non-recursive): ");
    postorder_nonrecursive(root);
    printf("\n");
    return 0;
}
```

OUTPUT :

Inorder traversal (recursive): 4 2 5 1 3
Inorder traversal (non-recursive): 4 2 5 1 3
Preorder traversal (recursive): 1 2 4 5 3
Preorder traversal (non-recursive): 1 2 4 5 3
Postorder traversal (recursive): 4 5 2 3 1
Postorder traversal (non-recursive): 4 5 2 3 1

WEEK - 12

PRACTICAL - 1 : Write a program to implement hashing using C language.

ALGORITHM :

1) Define the size of the hash table:
a. Decide on the size of the hash table array based on the number of expected elements and the desired load factor.
2) Define a hash function:

- **Create a hash function that takes a key as input and returns an index within the range of the hash table size.**
- **The hash function should aim to distribute the keys uniformly across the hash table to minimize collisions.**
- **Common hash functions include modulo division, multiplication, and bit manipulation.**

3) Define the structure for a hash table element:
a. Create a structure named HashElement with two members: key to store the key of the element, and value to store the corresponding value.

- **4) Create a hash table array: a. Declare an array of HashElement structures with the size determined in step 1.**
- **5) Implement key-value pair insertion:**
- **Use the hash function to calculate the index where the key-value pair should be inserted.**
- **Check if the calculated index is within the bounds of the hash table array.**
- **If the index is vacant, insert the key-value pair at that index.**
- **If there is a collision (another key-value pair already present at the index), handle it based on your collision resolution strategy (e.g., linear probing, chaining, etc.).**

6) Implement key-based value retrieval:

- **Use the hash function to calculate the index where the desired key is expected to be stored.**
- **Check if the calculated index is within the bounds of the hash table array.**
- **If the key is found at the calculated index, return the corresponding value.**
- **If the key is not found at the calculated index, handle it based on your retrieval strategy (e.g., returning a default value, reporting an error, etc.).**

7) Implement key-based element deletion:
a. Use the hash function to calculate the index where the key to be deleted is expected to be stored.

- **Check if the calculated index is within the bounds of the hash table array.**
- **If the key is found at the calculated index, delete the corresponding key-value pair. d. If the key is not found at the calculated index, handle it based on your deletion strategy (e.g., reporting an error, ignoring the request, etc.**

PROGRAM :

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

```c
#define TABLE_SIZE 10

struct node
{
   int key;
   int value;
   struct node *next;
};
struct node *new_node(int key, int value)
{
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   temp->key = key;
   temp->value = value;
   temp->next = NULL;
   return temp;
}
void insert(struct node *table[], int key, int value)
{
   int index = key % TABLE_SIZE;
   struct node *new_node = new_node(key, value);
   if (table[index] == NULL)
   {
       table[index] = new_node;
   }
   else
   {
       struct node *temp = table[index];
       while (temp->next != NULL)
       {
           temp = temp->next;
       }
       temp->next = new_node;
   }
}
int get(struct node *table[], int key)
{
   int index = key % TABLE_SIZE;
   struct node *temp = table[index];
   while (temp != NULL)
   {
       if (temp->key == key)
       {
           return temp->value;
       }
       temp = temp->next;
```

```
    }
    return -1;
}
int main()
{
    struct node *table[TABLE_SIZE] = {NULL};
    insert(table, 10, 100);
    insert(table, 20, 200);
    insert(table, 30, 300);
    printf("Value for key 10: %d\n", get(table, 10));
    printf("Value for key 20: %d\n", get(table, 20));
    printf("Value for key 30: %d\n", get(table, 30));
    return 0;
}
```

OUTPUT :

Value for key 10: 100
Value for key 20: 200
Value for key 30: 300