Shri Vaishnav Vidyapeeth Vishwavidyala, Indore

Shri Vaishnav Institute of Information Technology

Department of Computer Science & Engineering

LAB FILE

Jan-Jun2023

Name of the Student:

Enrollment:

Program:

Section:

Year/Sem: I / II

Subject Code: BTCS201N

Subject Name: Data Structure and Algorithms

Name of Subject Teacher: Dr. Sandeep Kumar Jain

INDEX

Name:

Enrollment No.

14	WAP for implementation of Bubble Sort.		
15	WAP for implementation of Insertion Sort.		

Program No:- 5

Aim/Title: WAP to perform various operation in the single linked list.

a) Create First Node
b) Insert At Beginning
c) Insert At Last
d) Delete From Beginning
e) Delete From Last
f) Display List

Date:

Tool:

Algorithm/Flow Chart:-

Algorithm:

1. Start the program.

2. Create a struct Node to represent the linked list node.

3. Define a function createNode(data) to create a new node with the given data.

- Allocate memory for a new node.

- Set the data of the new node.

- Set the next pointer of the new node to NULL.

- Return the new node.

4. Define a function insertAtBeginning(head, data) to insert a new node at the beginning of the list.

- Create a new node using createNode(data).

- Set the next pointer of the new node to the current head.

- Set the head pointer to the new node.

- Print "Node inserted at the beginning."

5. Define a function insertAtLast(head, data) to insert a new node at the end of the list.

- Create a new node using createNode(data).

- If the head is NULL, set the head to the new node and print "Node inserted at the last."

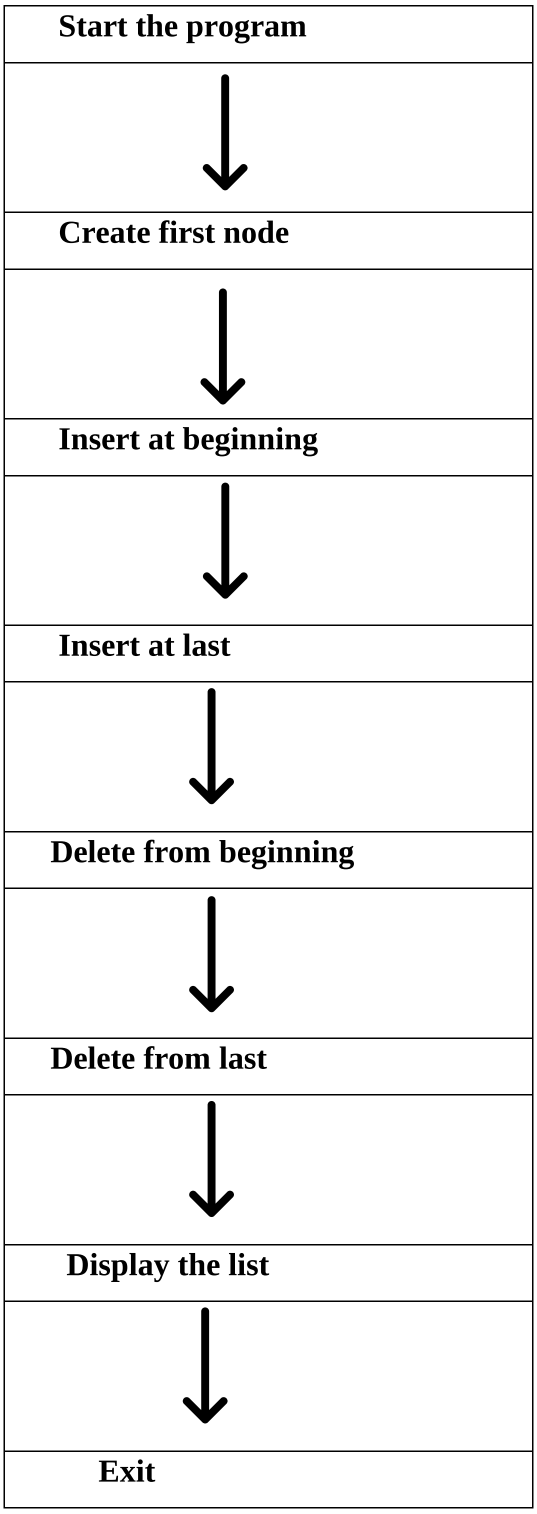
- Otherwise, iterate through the list until the last node.

- Set the next pointer of the last node to the new node.

- Print "Node inserted at the last."

```
6. Define a function deleteFromBeginning(head) to delete the first node from the list.
   - If the head is NULL, print "List is empty. Cannot delete from beginning."
   - Otherwise, store the head in a temporary variable.
   - Set the head to the next node.
   - Free the memory of the temporary variable.
   - Print "Node deleted from the beginning."
7. Define a function deleteFromLast(head) to delete the last node from the list.
   - If the head is NULL, print "List is empty. Cannot delete from last."
   - If the head is the only node, free the head and set it to NULL.
   - Otherwise, iterate through the list until the second-to-last node.
   - Set the next pointer of the second-to-last node to NULL.
   - Free the last node.
   - Print "Node deleted from the last."
8. Define a function displayList(head) to display the elements of the list.
   - If the head is NULL, print "List is empty."
   - Otherwise, iterate through the list and print each node's data.
9. Start the main function.
10. Declare the head pointer and set it to NULL.
11. Declare variables choice and data.
12. Start a do-while loop.
   - Print the menu options.
   - Read the choice from the user.
   - Use a switch statement based on the choice.
      - Case 1: Call insertAtBeginning(&head, data).
      - Case 2: Call insertAtLast(&head, data).
      - Case 3: Call deleteFromBeginning(&head).
      - Case 4: Call deleteFromLast(&head).
      - Case 5: Call displayList(head).
      - Case 0: Print "Exiting..." and exit the loop.
      - Default: Print "Invalid choice!".
   - Continue the loop until the choice is not 0.
13. End the main function.
14. End the program.
```

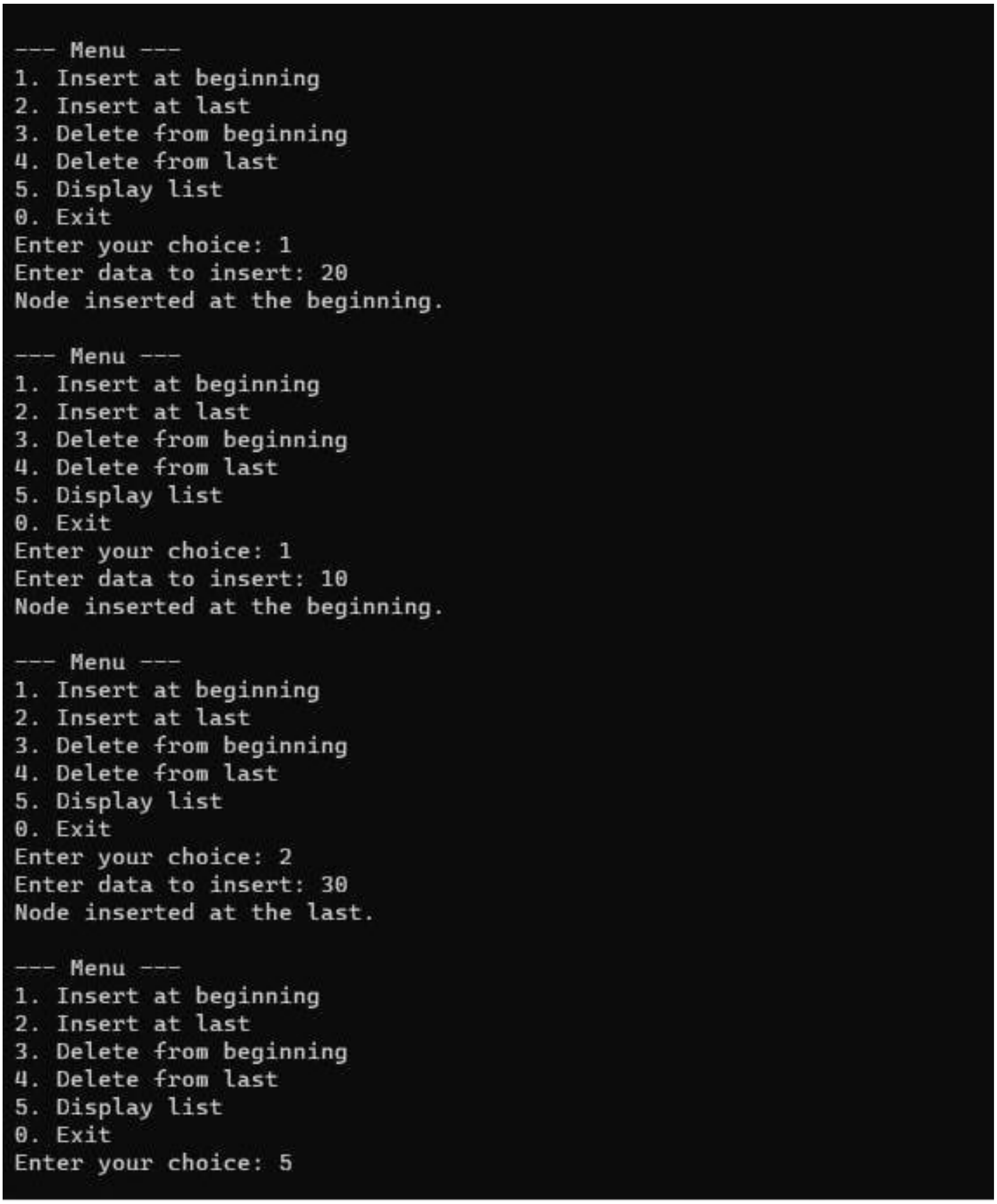
Flow char:-


Start the program
Create first node
Insert at beginning
Insert at last
Delete from beginning
Delete from last
Display the list
Exit

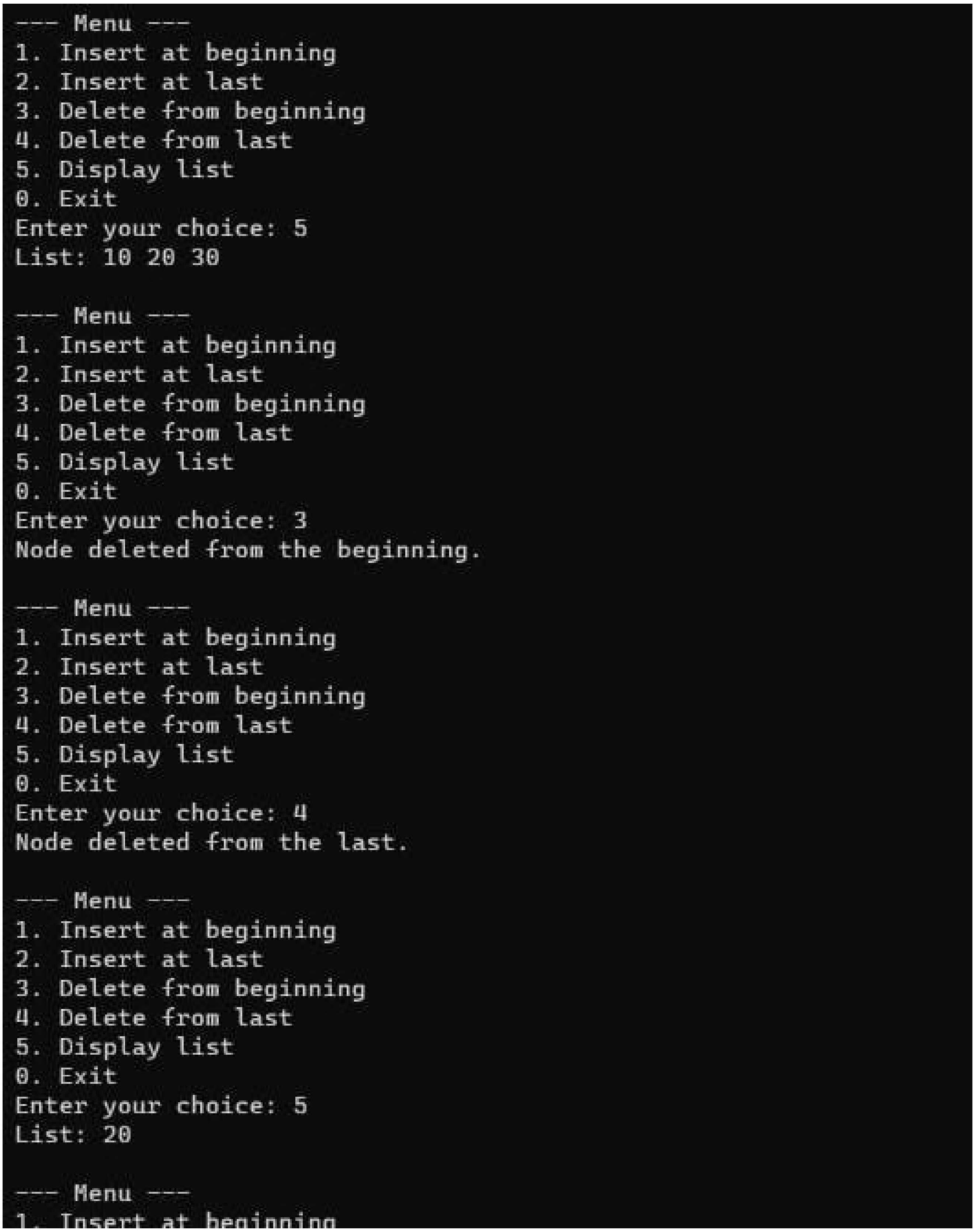
Source Code:-

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

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

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
    printf("Node inserted at the beginning.\n");
}

void insertAtLast(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        printf("Node inserted at the last.\n");
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    printf("Node inserted at the last.\n");
}

void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete from beginning.\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
    printf("Node deleted from the beginning.\n");
}

void deleteFromLast(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete from last.\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        printf("Node deleted from the last.\n");
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;
    free(temp);
    printf("Node deleted from the last.\n");
}

void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

    do {
        printf("\n--- Menu ---\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at last\n");
        printf("3. Delete from beginning\n");
        printf("4. Delete from last\n");
        printf("5. Display list\n");
    printf("0. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            insertAtBeginning(&head, data);
            break;
        case 2:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            insertAtLast(&head, data);
            break;
        case 3:
            deleteFromBeginning(&head);
            break;
        case 4:
            deleteFromLast(&head);
            break;
        case 5:
            displayList(head);
            break;
        case 0:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please enter a valid option.\n");
    }
} while (choice != 0);

return 0;
```

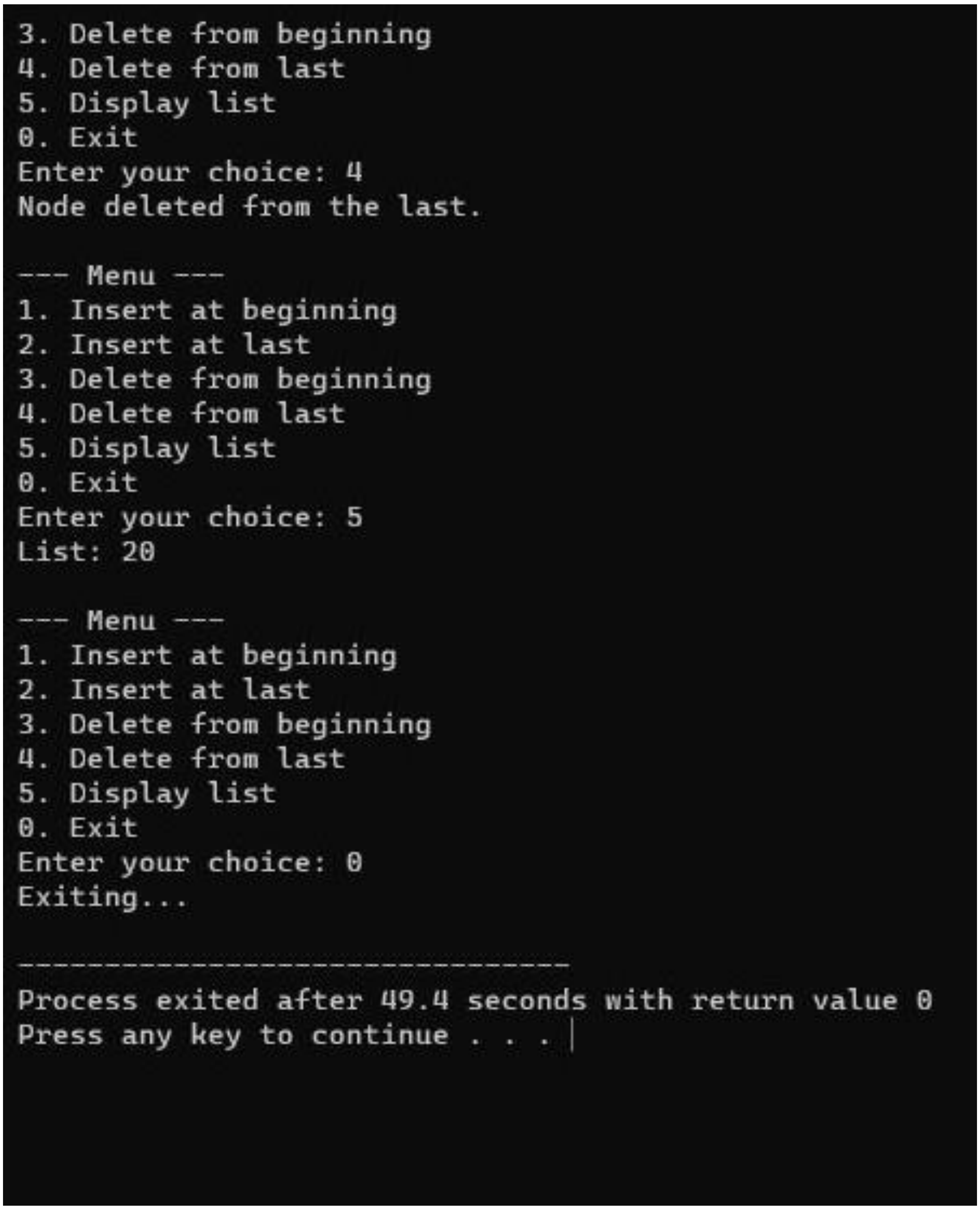
Outcomes :-



```
--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 1
Enter data to insert: 20
Node inserted at the beginning.

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 1
Enter data to insert: 10
Node inserted at the beginning.

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 2
Enter data to insert: 30
Node inserted at the last.

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 5
```



```
--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 5
List: 10 20 30

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 3
Node deleted from the beginning.

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 4
Node deleted from the last.

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 5
List: 20

--- Menu ---
1. Insert at beginning
```



```
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 4
Node deleted from the last.

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 5
List: 20

--- Menu ---
1. Insert at beginning
2. Insert at last
3. Delete from beginning
4. Delete from last
5. Display list
0. Exit
Enter your choice: 0
Exiting...

--------------------------------
Process exited after 49.4 seconds with return value 0
Press any key to continue . . .
```

Program No:- 09

Aim/Title: WAP for Tower of Hanoi using recursion.

Date:

Tool:

Algorithm/Flow Chart:-

Algorithm:-

1. Define a function towerOfHanoi that takes four parameters:

- `n`: the number of disks to be moved
- `source`: the tower from which disks are initially placed
- `auxiliary`: the auxiliary tower used for intermediate moves
- `destination`: the tower where disks are to be moved

2. Inside the towerOfHanoi function:

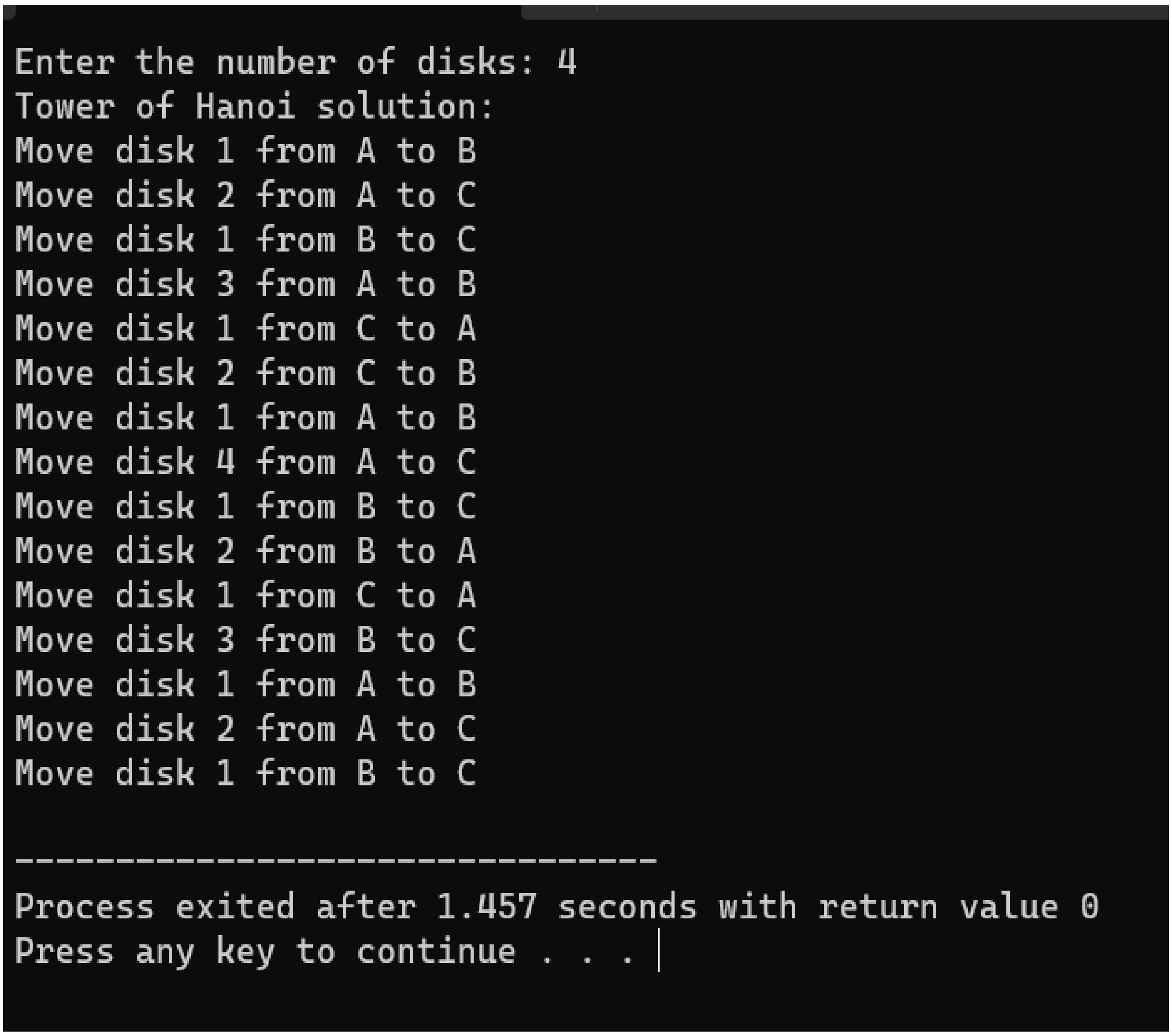
- If n is equal to 1 (base case), then:
  - I. Print the move of the disk from the source tower to the destination tower
  - II. Return from the function.
- Otherwise (recursive case), do the following:
  - I. Recursively call the towerOfHanoi function with n-1 disks, moving them from the source tower to the auxiliary tower.
  - II. Print the move of the nth disk from the source tower to the destination tower.
  - III. Recursively call the towerOfHanoi function with n-1 disks, moving them from the auxiliary tower to the destination tower.

3. In the main function:
   - Prompt the user to enter the number of disks.
   - Read the input value and store it in a variable `numDisks`.
   - Print the message indicating the start of the Tower of Hanoi solution.
   - Call the `towerOfHanoi` function with `numDisks` disks, using towers A, B, and C as the source, auxiliary, and destination towers, respectively.

Source Code:-

```c
#include <stdio.h>

void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }

    towerOfHanoi(n - 1, source, destination, auxiliary);
    printf("Move disk %d from %c to %c\n", n, source, destination);
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int numDisks;
    printf("Enter the number of disks: ");
    scanf("%d", &numDisks);

    printf("Tower of Hanoi solution:\n");
    towerOfHanoi(numDisks, 'A', 'B', 'C');

    return 0;
}
```

Outcomes :-



```
Enter the number of disks: 4
Tower of Hanoi solution:
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
Move disk 4 from A to C
Move disk 1 from B to C
Move disk 2 from B to A
Move disk 1 from C to A
Move disk 3 from B to C
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C

--------------------------------
Process exited after 1.457 seconds with return value 0
Press any key to continue . . .
```

Program No:- 10

Aim/Title: iilustrate queue implementation using array in c with following operation as enQueue, deQueue, isEmpty, displayQueue.

Date:

Tool:

Algorithm/Flow Chart:-

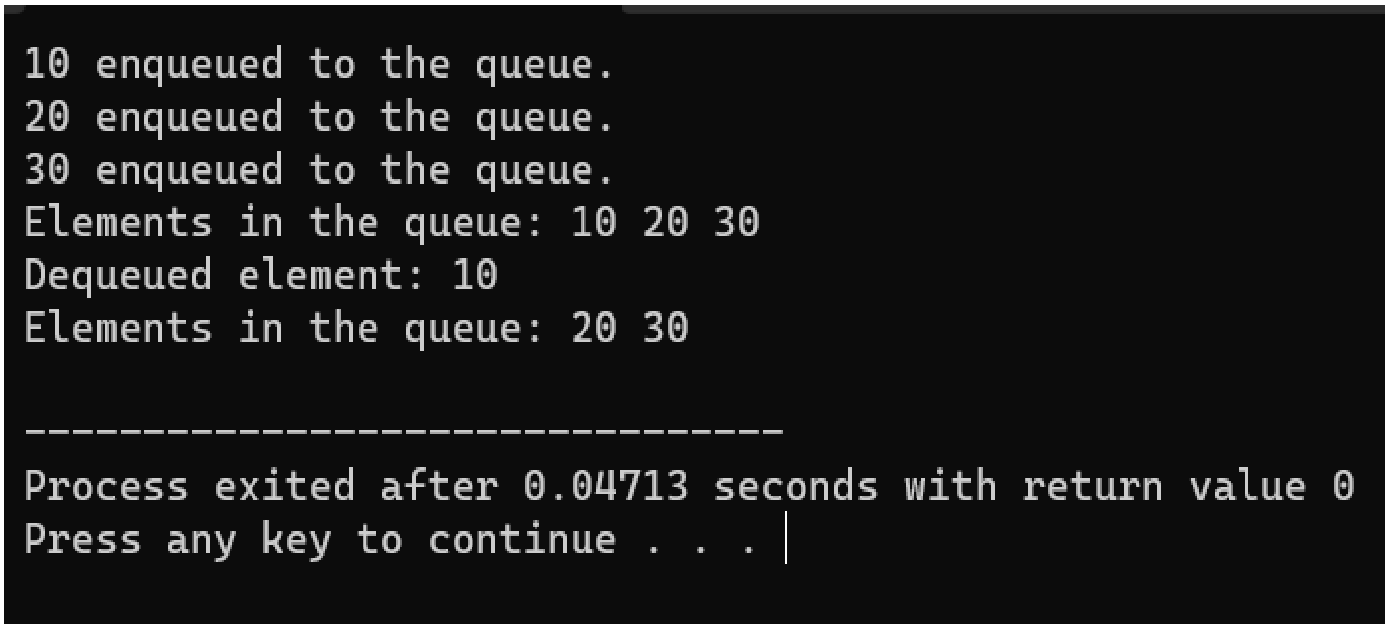
Algorithm:-

1. Create a structure Queue with an array items, front, and rear variables.
2. Initialize the queue by setting front and rear to -1.
3. Implement the isEmpty function to check if the queue is empty by comparing front with -1.
4. Implement the isFull function to check if the queue is full by comparing rear with the maximum size minus 1.
5. Implement the enQueue function to add an element to the queue:
    - Check if the queue is full. If it is, display an error message and return.
    - If the queue is empty, set `front` to 0.
    - Increment `rear` by 1 and assign the given value to `items[rear]`.
6. Implement the deQueue function to remove an element from the queue:
    - Check if the queue is empty. If it is, display an error message and return -1.
    - Retrieve the value at items[front].
    - Increment front by 1.
    - If front is greater than rear, reset the queue by setting front and rear to -1.
    - Return the dequeued value.
7. Implement the displayQueue function to display the elements in the queue:
    - Check if the queue is empty. If it is, display an appropriate message.
    - Iterate from `front` to `rear` and print each element.

Source Code:-

```
#include <stdio.h>

#define MAX_SIZE 100

// Structure to represent a queue
typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
} Queue;

// Function to initialize a queue
void initializeQueue(Queue* queue) {
    queue->front = -1;
    queue->rear = -1;
}

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

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

// Function to add an element to the queue (enQueue)
void enQueue(Queue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue %d\n", value);
        return;
    }

    if (isEmpty(queue)) {
        queue->front = 0;
    }

    queue->rear++;
    queue->items[queue->rear] = value;
    printf("%d enqueued to the queue.\n", value);
}

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

    int value = queue->items[queue->front];
    queue->front++;

    if (queue->front > queue->rear) {
        // Reset the queue
        queue->front = -1;
        queue->rear = -1;
    }

    return value;
}

// Function to display the elements in the queue
void displayQueue(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Elements in the queue: ");
    for (int i = queue->front; i <= queue->rear; i++) {
        printf("%d ", queue->items[i]);
    }
    printf("\n");
}

int main() {
    Queue queue;
    initializeQueue(&queue);

    enQueue(&queue, 10);
    enQueue(&queue, 20);
    enQueue(&queue, 30);
    displayQueue(&queue);

    int dequeuedElement = deQueue(&queue);
    if (dequeuedElement != -1) {
        printf("Dequeued element: %d\n", dequeuedElement);
    }

    displayQueue(&queue);

    return 0;
}
```

Outcomes :-



```
10 enqueued to the queue.
20 enqueued to the queue.
30 enqueued to the queue.
Elements in the queue: 10 20 30
Dequeued element: 10
Elements in the queue: 20 30

--------------------------------
Process exited after 0.04713 seconds with return value 0
Press any key to continue . . .
```

Program No:- 11

Aim/Title: Illustrate queue implementation using linked list with following operation as enQueue, deQueue, isEmpty, displayQueue.

Date:

Tool:

Algorithm/Flow Chart:-

Algorithm:-

1. Start
2. Create a structure for each node in the linked list (Node) with data and next pointer.
3. Create a structure for the Queue with front and rear pointers.
4. Create a function ‘createQueue’ to create an empty queue:

- Allocate memory for the Queue structure.
- Set the front and rear pointers to NULL.
- Return the created queue.

5. Create a function isEmpty to check if the queue is empty:

- If the front pointer is NULL, return 1 (true); otherwise, return 0 (false).

6. Create a function enQueue to insert an element into the queue:

- Create a new node.
- Set the data of the new node to the given element.
- Set the next pointer of the new node to NULL.
- If the queue is empty, set both front and rear pointers to the new node.
- Otherwise, set the next pointer of the current rear node to the new node.
- Set the rear pointer to the new node.

7. Create a function ‘deQueue’ to remove an element from the queue:

- If the queue is empty, print an appropriate message and return.
- Store the front node in a temporary variable.
- Move the front pointer to the next node.
- If the front becomes NULL, set the rear pointer to NULL as well.
- Free the memory occupied by the dequeued node.

8. Create a function displayQueue to print the elements of the queue:

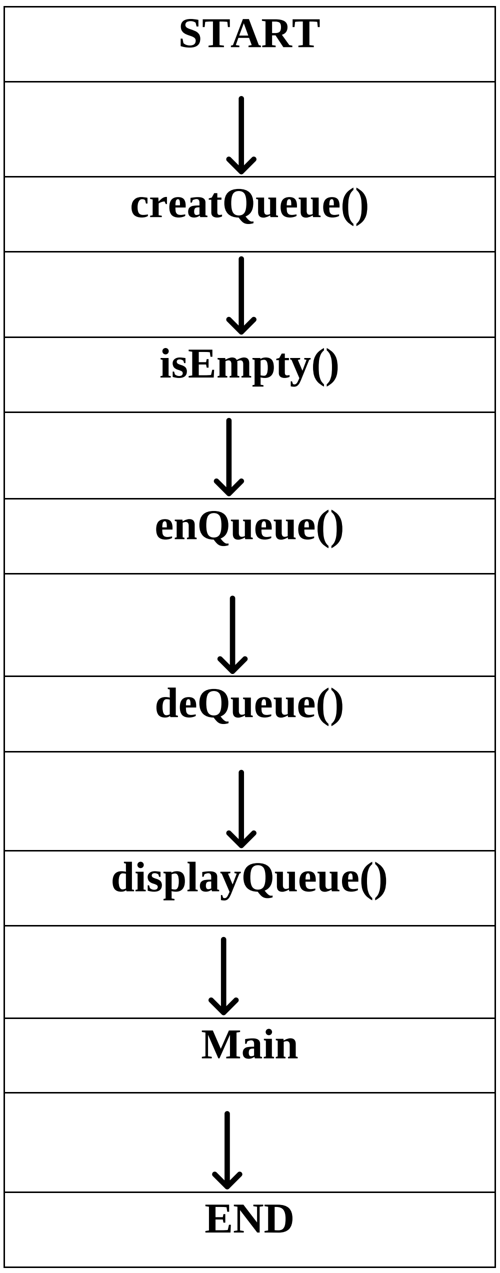
- If the queue is empty, print an appropriate message and return.
- Traverse the queue from the front to the rear.
- Print each element.

9. Main function:

- Create an empty queue using createQueue.
- Perform enqueue, dequeue, and display operations on the queue to demonstrate its functionality.

10. End

Flow Chart:-

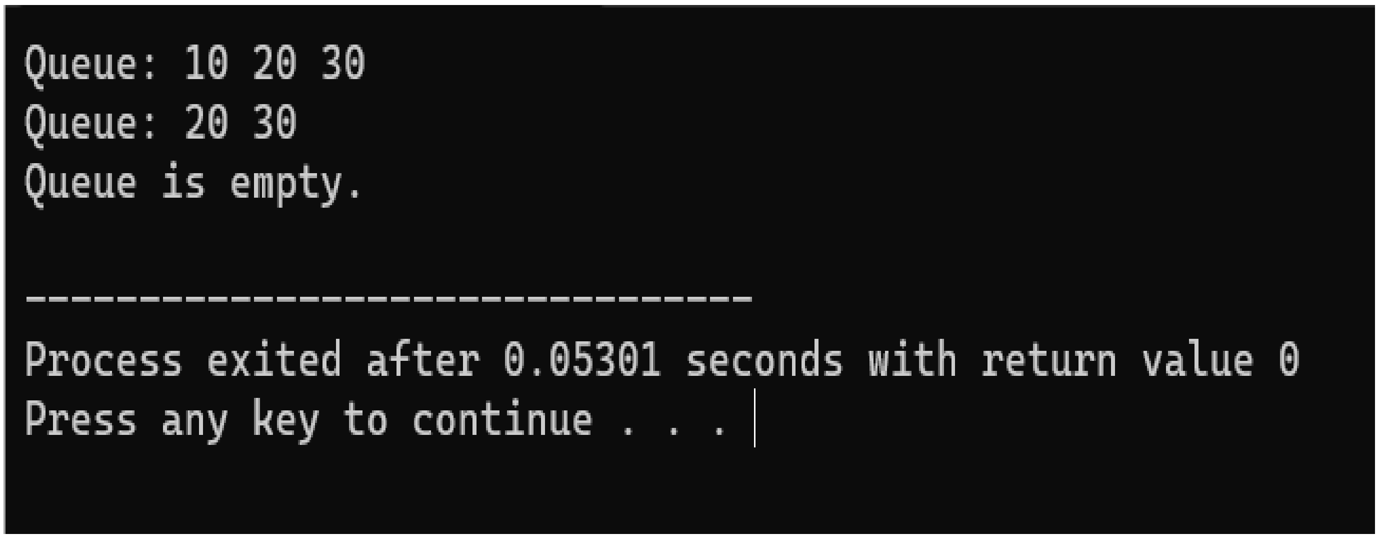

START
creatQueue()
isEmpty()
enQueue()
deQueue()
displayQueue()
Main
END

Source Code:-

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

// Structure for each node in the linked list
struct Node {
    int data;
    struct Node* next;
};

// Structure for the Queue
struct Queue {
    struct Node* front;
    struct Node* rear;
};

// Function to create an empty queue
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = NULL;
    queue->rear = NULL;
    return queue;
}

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

// Function to enqueue an element into the queue
void enQueue(struct Queue* queue, int data) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    // If the queue is empty, make the new node both front and rear
    if (isEmpty(queue)) {
        queue->front = newNode;
        queue->rear = newNode;
        return;
    }

    // Otherwise, add the new node at the end and update rear
    queue->rear->next = newNode;
    queue->rear = newNode;
}

// Function to dequeue an element from the queue
void deQueue(struct Queue* queue) {
    // If the queue is empty, no elements to dequeue
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return;
    }

    // Store the front node and move front to the next node
    struct Node* temp = queue->front;
    queue->front = queue->front->next;

    // If the front becomes NULL, update rear as NULL as well
    if (queue->front == NULL) {
        queue->rear = NULL;
    }

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

```
}

// Function to display the elements of the queue
void displayQueue(struct Queue* queue) {
    // If the queue is empty, display appropriate message
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

    // Traverse the queue and print each element
    struct Node* current = queue->front;
    printf("Queue: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Main function
int main() {
    // Create an empty queue
    struct Queue* queue = createQueue();

    // Demonstrate queue operations
    enQueue(queue, 10);
    enQueue(queue, 20);
    enQueue(queue, 30);

    displayQueue(queue);  // Output: Queue: 10 20 30

    deQueue(queue);
    displayQueue(queue);  // Output: Queue: 20 30

    deQueue(queue);
    deQueue(queue);

    displayQueue(queue);  // Output: Queue is empty.

    return 0;
}
```

Outcomes :-



```
Queue: 10 20 30
Queue: 20 30
Queue is empty.

--------------------------------
Process exited after 0.05301 seconds with return value 0
Press any key to continue . . .
```

Program No:- 12

Aim/Title: WAP to construct a binary search tree and perform deletion and inorder traversal on it.

Date:

Tool:

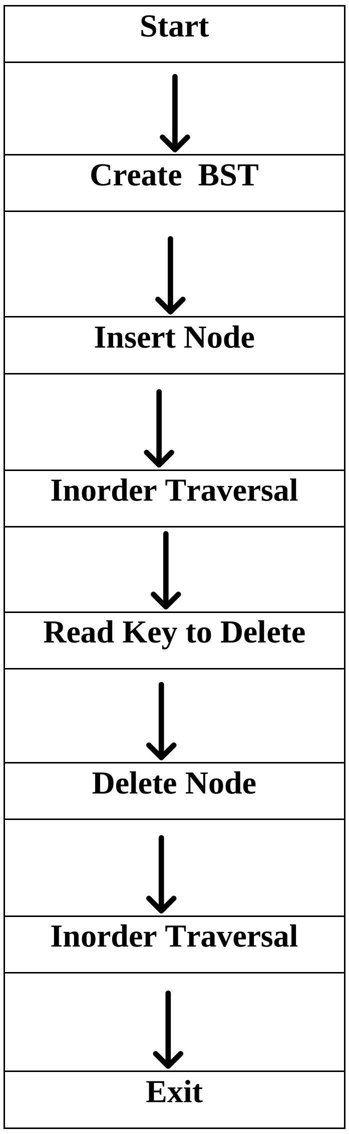
Algorithm/Flow Chart:-

Algorithm:-

Start with an empty binary search tree (BST).
Create a function createNode to allocate memory for a new node and initialize its data and pointers.
Create a function insertNode to insert a node into the BST:
If the tree is empty, create a new node with the given data and return it as the root.
If the data is less than the root's data, recursively insert it into the left subtree.
If the data is greater than the root's data, recursively insert it into the right subtree.
Return the modified root.
Create a function inorderTraversal to perform inorder traversal of the BST:
If the current node is not null:
Recursively traverse the left subtree.
Print the data of the current node.
Recursively traverse the right subtree.
Create a function minValueNode to find the minimum value node in a given BST:
Start from the given node.
Traverse down the left subtree until reaching the leftmost node.
Return the leftmost node.
Create a function deleteNode to delete a node from the BST:
If the root is null, return null.
If the data to be deleted is less than the root's data, recursively delete it from the left subtree.
If the data to be deleted is greater than the root's data, recursively delete it from the right subtree.
If the data to be deleted is equal to the root's data:
If the node has no left child, replace it with its right child (if any) and free the node.
If the node has no right child, replace it with its left child and free the node.
If the node has both left and right children, find the minimum value node from the right subtree.
Copy the data of the minimum value node to the node to be deleted.
Recursively delete the minimum value node from the right subtree.
Return the modified root.
Create the main function:
Initialize the root of the BST as null.

Insert nodes into the BST using the insertNode function.
Perform inorder traversal before deletion using the inorderTraversal function.
Read the key to be deleted from the user.
Delete the node with the given key using the deleteNode function.
Perform inorder traversal after deletion using the inorderTraversal function.
Return 0 to exit the program.

Flow Chart:-



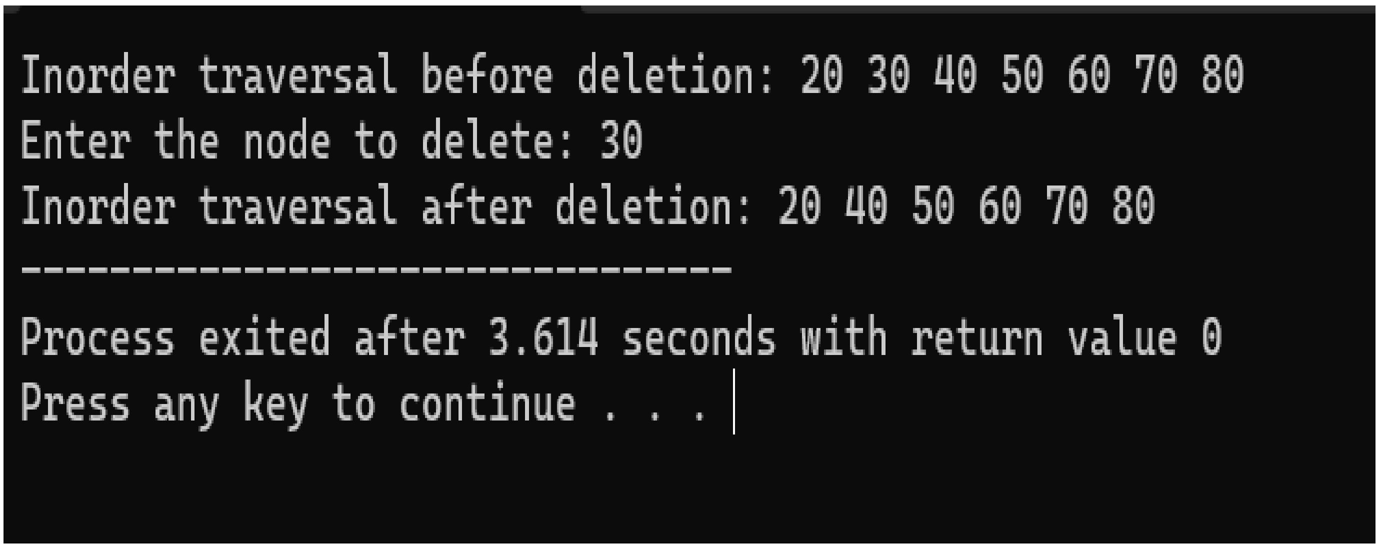
Source Code:-

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

// Structure for a node in the binary search tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node with the given data
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 insert a node into the binary search tree
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    }
    else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }
    return root;
}
```

```c
// Function to perform inorder traversal of the binary search tree
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Function to find the minimum value node in a given binary search tree
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}

// Function to delete a node from the binary search tree
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    }
    else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    }
    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 = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

int main() {
    struct Node* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
    insertNode(root, 40);
    insertNode(root, 70);
    insertNode(root, 60);
    insertNode(root, 80);

    printf("Inorder traversal before deletion: ");
    inorderTraversal(root);
    int key;
    printf("\nEnter the node to delete: ");
    scanf("%d", &key);
    root = deleteNode(root, key);
    printf("Inorder traversal after deletion: ");
    inorderTraversal(root);

    return 0;
```

```
}
```

Outcomes :-


```
Inorder traversal before deletion: 20 30 40 50 60 70 80
Enter the node to delete: 30
Inorder traversal after deletion: 20 40 50 60 70 80
--------------------------------
Process exited after 3.614 seconds with return value 0
Press any key to continue . . .
```

Program No:- 13

Aim/Title:- WAP to develop an algorithm for binary search, implement and test it.

Date:

Tool:

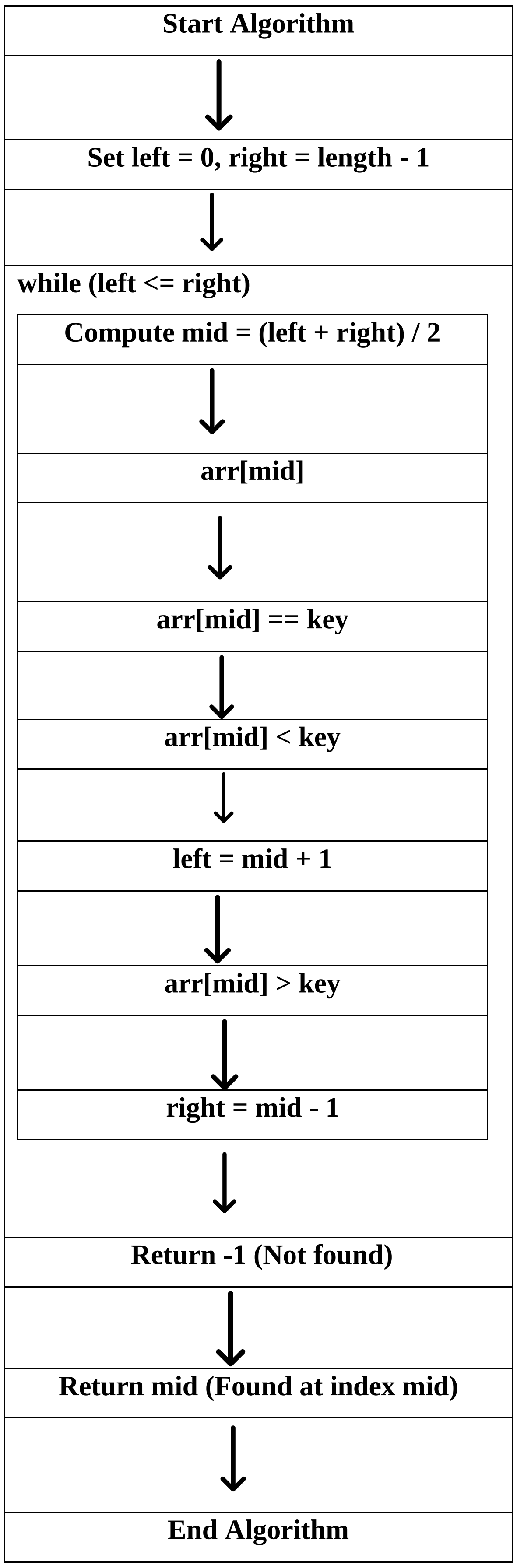
Algorithm/Flow Chart:-

Algorithm:-

Steps:

```
1. Set left = 0 and right = length of arr - 1.
2. Repeat the following steps until left <= right:
    a. Compute mid as (left + right) / 2.
    b. If arr[mid] equals key, return mid.
    c. If arr[mid] is less than key, set left = mid + 1.
    d. If arr[mid] is greater than key, set right = mid - 1.
3. If the key is not found after the loop, return -1.
```

Flow chart:-

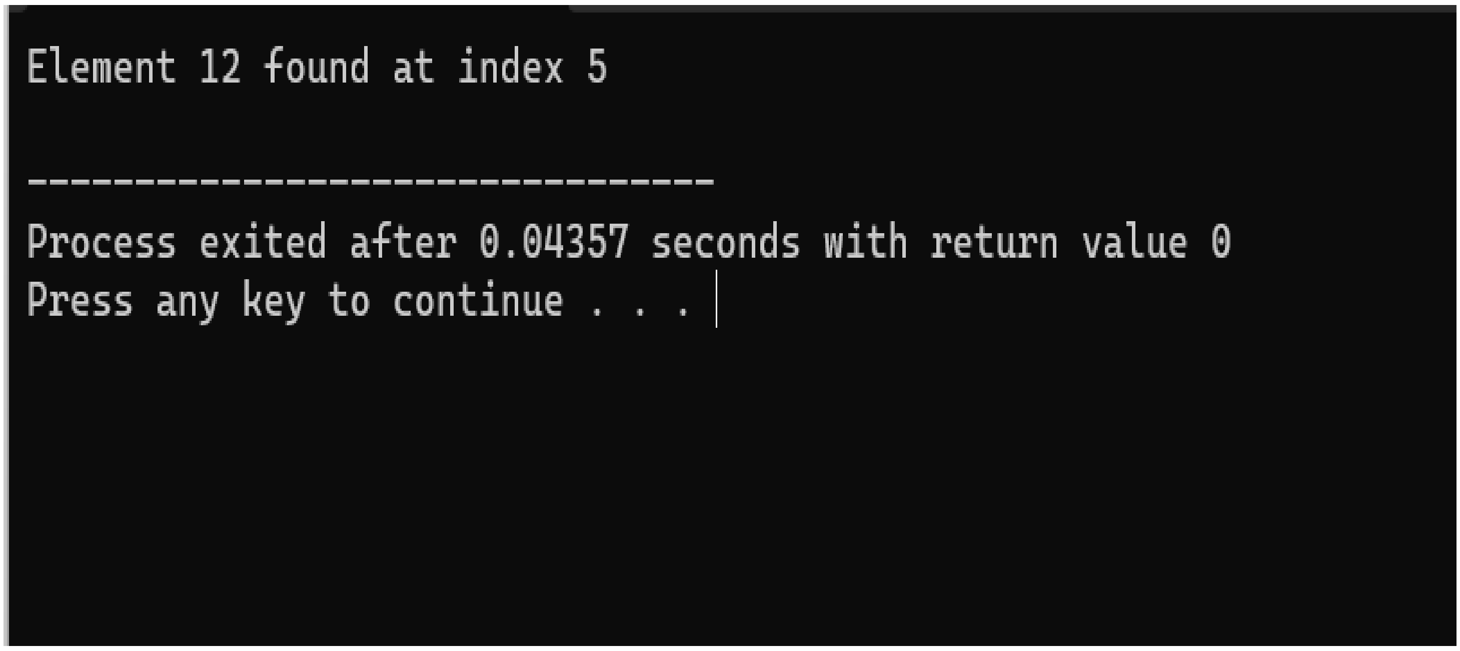

Start Algorithm
Set left = 0, right = length - 1
while (left <= right)
Compute mid = (left + right) / 2
arr[mid]
arr[mid] == key
arr[mid] < key
left = mid + 1
arr[mid] > key
right = mid - 1
Return -1 (Not found)
Return mid (Found at index mid)
End Algorithm

Source Code:-

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int key) {
   while (left <= right) {
      int mid = left + (right - left) / 2;

      // Check if the key is present at the middle
      if (arr[mid] == key)
         return mid;

      // If the key is greater, ignore the left half
      if (arr[mid] < key)
         left = mid + 1;

      // If the key is smaller, ignore the right half
      else
         right = mid - 1;
   }

   // Key not found
   return -1;
}

int main() {
   int arr[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key = 12;

   int index = binarySearch(arr, 0, n - 1, key);
```

```
    if (index != -1)
        printf("Element %d found at index %d\n", key, index);
    else
        printf("Element %d not found in the array\n", key);

    return 0;
}
```

Outcomes :-



```
Element 12 found at index 5


--------------------------------
Process exited after 0.04357 seconds with return value 0
Press any key to continue . . .
```