MODULE -3

1. INSERT A NODE AT THE HEAD OF A LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
// A linked list node
struct Node {
  int data;
  struct Node *next;
};
// Inserts a new node at the beginning
void push(struct Node** head_ref, int new_data) {
  struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
  new_node->data = new_data;
  new_node->next = (*head_ref);
  (*head_ref) = new_node;
}

void insertAfter(struct Node* prev_node, int new_data) {
  if (prev_node == NULL) {
    printf("The given previous node cannot be NULL\n");
    return;
  }
  struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
  new_node->data = new_data;
  new_node->next = prev_node->next;
  prev_node->next = new_node;
}
void append(struct Node** head_ref, int new_data) {
  struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
  struct Node *last = *head_ref;
  new_node->data = new_data;
  new_node->next = NULL;
  if (*head_ref == NULL) {
    *head_ref = new_node;
    return;
  }
  while (last->next != NULL)
    last = last->next;
  last->next = new_node;
}
void printList(struct Node *node) {
  while (node != NULL) {
    printf(" %d ", node->data);
    node = node->next;
  }
}
int main() {
  struct Node* head = NULL;
  append(&head, 6);     // List: 6
  push(&head, 7);       // List: 7->6
  push(&head, 1);       // List: 1->7->6
  append(&head, 4);     // List: 1->7->6->4
  insertAfter(head->next, 8); // List: 1->7->8->6->4

  printf("Created Linked list is: ");
  printList(head);

  return 0;
}
```

2. INSERT A NODE AT THE TAIL OF A LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node *next;
};
void insertAtTail(struct Node **head, int data) {
   struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed!\n");
      return;
   }
   newNode->data = data;
   newNode->next = NULL;
   if (*head == NULL) {
      *head = newNode;
      return;
   }
   struct Node *current = *head;
   while (current->next != NULL) {
      current = current->next;
   }
   current->next = newNode;
}
void printList(struct Node *head) {
   struct Node *current = head;
   while (current != NULL) {
      printf("%d -> ", current->data);
      current = current->next;
   }
   printf("NULL\n");
}
int main() {
   struct Node *head = NULL;
   insertAtTail(&head, 10);
   insertAtTail(&head, 20);
   insertAtTail(&head, 30);

   printf("Linked List: ");
   printList(head);
   struct Node *current = head;
   while (current != NULL) {
      struct Node *temp = current;
      current = current->next;
      free(temp);
   }
   return 0;
}
```

3. INSERT A NODE AT A SPECIFIC POSITION IN A LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
struct slinklist {
   int data;
   struct slinklist *next;
};
typedef struct slinklist node;
node *start = NULL;
int menu() {
   int ch;
   printf("\n 1.Create a list ");
   printf("\n--------------------------");
   printf("\n 2.Insert a node at specified position");
   printf("\n--------------------------");
   printf("\n 3.Display");
   printf("\n--------------------------");
   printf("\n 4. Exit ");
   printf("\n\n Enter your choice: ");
   scanf("%d", &ch);
   return ch;
}
node* getnode() {
   node *newnode;
   newnode = (node *)malloc(sizeof(node));
   printf("\n Enter data: ");
   scanf("%d", &newnode->data);
   newnode->next = NULL;
   return newnode;
}
void createlist(int n) {
   int i;
   node *newnode;
   node *temp;
   for (i = 0; i < n; i++) {
      newnode = getnode();
      if (start == NULL) {
         start = newnode;
      } else {
         temp = start;
         while (temp->next != NULL)
            temp = temp->next;
         temp->next = newnode;
      } }}
int countnode(node *ptr) {
   int count = 0;
   while (ptr != NULL) {
      count++;
      ptr = ptr->next;
   }
   return count;
}

void display() {
   node *temp;
   temp = start;
   printf("\n The contents of List (Left to Right): \n");
   if (start == NULL) {
      printf("\n Empty List");
```

```
      return;
   } else {
      while (temp != NULL) {
         printf("%d-->", temp->data);
         temp = temp->next;
      }
   }
   printf(" X ");
}

void insert_at_pos() {
   node *newnode, *temp, *prev;
   int pos, nodectr, ctr = 1;
   newnode = getnode();
   printf("\n Enter the position: ");
   scanf("%d", &pos);
   nodectr = countnode(start);

   if (pos < 1 || pos > nodectr + 1) {
      printf("Position %d is invalid\n", pos);
      free(newnode);
      return;
   }
   if (pos == 1) { // Insert at the beginning
      newnode->next = start;
      start = newnode;
   } else {
      temp = start;
      while (ctr < pos - 1) {
         temp = temp->next;
         ctr++;
      }
      newnode->next = temp->next;
      temp->next = newnode;
   }
}
void main(void) {
   int ch, n;
   while (1) {
      ch = menu();
      switch (ch) {
         case 1:
            if (start == NULL) {
               printf("\n Number of nodes you want to create: ");
               scanf("%d", &n);
               createlist(n);
               printf("\n List created..");
            } else
               printf("\n List is already created..");
            break;
         case 2:
            insert_at_pos();
            break;
         case 3:
            display();
            break;
         default:
            exit(0);
      }}}
```

MODULE – 5

1. Lowest Common Ancestor in Binary Tree

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int data;
 struct node *left, *right;
};
struct node *lca (struct node *root, int n1, int n2)
{
 while (root != NULL)
  {
   if (root->data > n1 && root->data > n2)
 root = root->left;
   else if (root->data < n1 && root->data < n2)
 root = root->right;
   else
 break;
  }
 return root;
}
struct node *newNode (int data)
{
 struct node *node = (struct node *) malloc (sizeof (struct node));
 node->data = data;
 node->left = node->right = NULL;
 return (node);
}
int main ()
{
 struct node *root = newNode (20);
 root->left = newNode (8);
 root->right = newNode (22);
 root->left->left = newNode (4);
```

```
  root->left->right = newNode (12);
  root->left->right->left = newNode (10);
  root->left->right->right = newNode (14);
  int n1 = 10, n2 = 14;
  struct node *t = lca (root, n1, n2);
  printf ("LCA of %d and %d is %d \n", n1, n2, t->data);
  n1 = 14, n2 = 8;
  t = lca (root, n1, n2);
  printf ("LCA of %d and %d is %d \n", n1, n2, t->data);
  n1 = 10, n2 = 22;
  t = lca (root, n1, n2);
  printf ("LCA of %d and %d is %d \n", n1, n2, t->data);
  getchar ();
  return 0;
}
```

2. Height of a Binary Tree

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int data;
 struct node *left;
 struct node *right;
};
int height (struct node *node)
{
 if (node == NULL)
  return 0;
 else
  {
   int leftHeight = height (node->left);
   int rightHeight = height (node->right);
   if (leftHeight > rightHeight)
    return (leftHeight + 1);
   else
    return (rightHeight + 1); }}
struct node *newNode (int data) {
 struct node *node = (struct node *) malloc (sizeof (struct node));
 node->data = data;
 node->left = NULL;
 node->right = NULL;
 return (node); }
int main () {
 struct node *root = newNode (10);
 root->left = newNode (20);
 root->right = newNode (30);
 root->left->left = newNode (40);
 root->left->right = newNode (50);
 printf ("Height of tree is %d", height (root));
 return 0; }
```

3. BINARY SEARCH TREE INSERTION

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int val;
   struct node *left, *right;
};
struct node* newNode(int item)
{
   struct node* temp =  (struct node *)malloc(sizeof(struct node));
   temp->val = item;
   temp->left = temp->right = NULL;
   return temp;
}
void inorder(struct node* root)
{
   if (root != NULL)
   {
      inorder(root->left);
      printf("%d \n", root->val);
      inorder(root->right);
   }
}
struct node* insert(struct node* node, int val)
{
   if (node == NULL) return newNode(val);
   if (val < node->val)
      node->left  = insert(node->left, val);
   else if (val > node->val)
      node->right = insert(node->right, val);
   return node;
}
 int main()
{
   struct node* root = NULL;
   root = insert(root, 100);
   insert(root, 60);
   insert(root, 40);
   insert(root, 80);
   insert(root, 140);
   insert(root, 120);
insert(root, 160);
inorder(root);
return 0;
}
```

Additional programs

1. Merge Sort algorithm for linked lists:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
   int data;
   struct Node* next;
} Node;
Node* createNode(int data) {
   Node* newNode = (Node*) malloc(sizeof(Node));
   if (!newNode) {
      printf("Memory error\n");
      return NULL; }
   newNode->data = data;
   newNode->next = NULL;
   return newNode; }
void insertNode(Node** head, int data) {
   Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
      return; }
   Node* lastNode = *head;
   while (lastNode->next) {
      lastNode = lastNode->next; }
   lastNode->next = newNode; }
void printList(Node* head) {
   while (head) {
      printf("%d -> ", head->data);
      head = head->next; }
   printf("NULL\n"); }
Node* getMiddle(Node* head) {
   if (head == NULL) {
      return head; }
   Node* slow = head;
   Node* fast = head;
   while (fast->next && fast->next->next) {
```

```
        slow = slow->next;
        fast = fast->next->next; }
    return slow; }
Node* merge(Node* head1, Node* head2) {
    if (head1 == NULL) {
        return head2; }
    if (head2 == NULL) {
        return head1; }
    if (head1->data <= head2->data) {
        head1->next = merge(head1->next, head2);
        return head1;
    } else {
        head2->next = merge(head1, head2->next);
        return head2;}}
Node* mergeSort(Node* head) {
    if (head == NULL || head->next == NULL) {
        return head; }
    Node* mid = getMiddle(head);
    Node* midNext = mid->next;
mid->next = NULL;
Node* left = mergeSort(head);
Node* right = mergeSort(midNext);
Node* sortedList = merge(left, right);
return sortedList; }
int main() {
Node* head = NULL;
insertNode(&head, 5);   insertNode(&head, 2);  insertNode(&head, 8);
insertNode(&head, 3);  insertNode(&head, 1);
insertNode(&head, 6);
insertNode(&head, 4);
printf("Original Linked List: ");
printList(head);
head = mergeSort(head);
printf("Sorted Linked List: ");
printList(head);
return 0; }
```

2. Divide a linked list into two halves:

```
#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));
newNode->data = data;
newNode->next = NULL;
return newNode;
}
void insertEnd(struct Node** head, int data) {
  struct Node* newNode = createNode(data);
  if (*head == NULL) {
    *head = newNode;
    newNode->next = *head;
  } else {
    struct Node* temp = *head;
    while (temp->next != *head) {
      temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = *head;
  }
}
void splitCircularList(struct Node* head, struct Node** head1, struct Node** head2) {
  if (head == NULL) return;

  struct Node* slow = head;
  struct Node* fast = head;

  while (fast->next != head && fast->next->next != head) {
    slow = slow->next;
    fast = fast->next->next;
  }
  if (fast->next->next == head) {
    fast = fast->next;
  }
  *head1 = head;
  if (head->next != head) {
    *head2 = slow->next;
  }
  slow->next = *head1;
  fast->next = *head2;
}
void printList(struct Node* head) {
  if (head == NULL) return;

  struct Node* temp = head;
  do {
    printf("%d -> ", temp->data);
    temp = temp->next;
  } while (temp != head);
  printf("HEAD\n");
}
```

```
int main() {
    struct Node* head = NULL;
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;
    insertEnd(&head, 1);
    insertEnd(&head, 2);
    insertEnd(&head, 3);
    insertEnd(&head, 4);
    insertEnd(&head, 5);

    printf("Original Circular Linked List: ");
    printList(head);
    splitCircularList(head, &head1, &head2);

    printf("First Half: ");
    printList(head1);

    printf("Second Half: ");
    printList(head2);

    return 0;
}
```

3. Check if two trees are mirrors of each other

```
#include <stdio.h>
#include <stdlib.h>
struct TreeNode {
int val;
struct TreeNode *left, *right;
};
struct TreeNode* newNode(int val) {
struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
node->val = val;
node->left = node->right = NULL;
return node;
}
int areMirror(struct TreeNode* root1, struct TreeNode* root2) {
if (root1 == NULL && root2 == NULL) return 1;
if (root1 == NULL || root2 == NULL) return 0;
return (root1->val == root2->val) &&
areMirror(root1->left, root2->right) &&
areMirror(root1->right, root2->left);
}
int main() {
struct TreeNode* root1 = newNode(1);
root1->left = newNode(2);
root1->right = newNode(3);
struct TreeNode* root2 = newNode(1);
root2->left = newNode(3);
root2->right = newNode(2);
printf("%s\n", areMirror(root1, root2) ? "true" : "false"); // Output: true
return 0;
}
```

4. Check whether BST Contains Dead End.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
struct Node {
int data;
   struct Node *left, *right;
};
struct Node* newNode(int data) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->data = data;
   node->left = node->right = NULL;
   return node; }
struct Node* insert(struct Node* root, int key) {
   if (root == NULL)
      return newNode(key);
   if (key < root->data)
      root->left = insert(root->left, key);
   else
      root->right = insert(root->right, key);
   return root; }
int checkDeadEnd(struct Node* root, int min, int max) {
   if (root == NULL)
      return 0;
   if (min == max)
      return 1;
   return checkDeadEnd(root->left, min, root->data - 1) ||
         checkDeadEnd(root->right, root->data + 1, max); }
int containsDeadEnd(struct Node* root) {
   return checkDeadEnd(root, 1, INT_MAX); }
int main() {
   struct Node* root = NULL;
   root = insert(root, 8);
   root = insert(root, 5);
```

```
root = insert(root, 2);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 11);

if (containsDeadEnd(root))

printf("BST contains a dead end\n");

else

printf("BST does not contain a dead end\n");

return 0;

}
```