4. Implementation of Link List

    a. Creation of Singly link list, Doubly Linked list
    b. Concatenation of Link list
    c. Insertion and Deletion of node in link list
    d. Splitting the link list into two link list

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
```

```
  head = lk;
}
void insertatend(int data){

  //create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
  struct node *linkedlist = head;

  // point it to old first node
  while(linkedlist->next != NULL)
    linkedlist = linkedlist->next;

  //point first to new first node
  linkedlist->next = lk;
}
void insertafternode(struct node *list, int data){
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
  lk->next = list->next;
  list->next = lk;
}
void deleteatbegin(){
  head = head->next;
}
void deleteatend(){
  struct node *linkedlist = head;
  while (linkedlist->next->next != NULL)
    linkedlist = linkedlist->next;
  linkedlist->next = NULL;
}
void deletenode(int key){
  struct node *temp = head, *prev;
  if (temp != NULL && temp->data == key) {
    head = temp->next;
    return;
  }

  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
```

```
    prev = temp;
    temp = temp->next;
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;
}
int searchlist(int key){
  struct node *temp = head;
  while(temp != NULL) {
    if (temp->data == key) {
      return 1;
    }
    temp=temp->next;
  }
  return 0;
}
void main(){
  int k=0;
  insertatbegin(12);
  insertatbegin(22);
  insertatend(30);
  insertatend(44);
  insertatbegin(50);
  insertafternode(head->next->next, 33);
  printf("Linked List: ");

  // print list
  printList();
  deleteatbegin();
  deleteatend();
  deletenode(12);
  printf("\nLinked List after deletion: ");

  // print list
  printList();
  insertatbegin(4);
```

```
  insertatbegin(16);
  printf("\nUpdated Linked List: ");
  printList();
  k = searchlist(16);
  if (k == 1)
    printf("\nElement is found");
  else
    printf("\nElement is not present in the list");
}
```

1. Data Structure Definition:
    - The program defines a structure node that represents a node in a linked list. Each node contains an integer data element and a pointer to the next node in the list.
    - Two global pointers head and current are declared to manage the linked list.
2. Function Definitions:
    - printList(): Prints the elements of the linked list.
    - insertatbegin(int data): Inserts a new node with the given data at the beginning of the linked list.
    - insertatend(int data): Inserts a new node with the given data at the end of the linked list.
    - insertafternode(struct node *list, int data): Inserts a new node with the given data after a specified node in the list.
    - deleteatbegin(): Deletes the first node in the linked list.
    - deleteatend(): Deletes the last node in the linked list.
    - deletenode(int key): Deletes a node with a specific key value from the linked list.
    - searchlist(int key): Searches for a key in the linked list and returns 1 if found, otherwise returns 0.
3. Main Function:
    - The main() function initializes the linked list with some nodes, performs various operations such as insertion, deletion, and searching, and then prints the updated linked list.
    - The linked list is initially populated with nodes containing values: 50 -> 22 -> 30 -> 12 -> 33 -> 44.
    - Nodes are then deleted, updated, and searched within the linked list.
4. Output:
    - The output of the program should display the linked list at different stages: after initial insertion, after deletions, after further insertions, and the result of the search operation.

In summary, the program demonstrates basic linked list operations such as insertion, deletion, and searching in a singly linked list implemented in C.

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

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

void deleteStart (struct Node **head)
{
  struct Node *temp = *head;

  // if there are no nodes in Linked List can't delete
  if (*head == NULL)
    {
      printf ("Linked List Empty, nothing to delete");
      return;
    }

  // move head to next node
  *head = (*head)->next;

  printf ("\n%d deleted\n", temp->data);
  free (temp);
}

void insertStart (struct Node **head, int data)
{

  // dynamically create memory for this newNode
  struct Node *newNode = (struct Node *) malloc (sizeof (struct Node));

  // assign data value
  newNode->data = data;
  // change the next node of this newNode
  // to current head of Linked List
  newNode->next = *head;

  //re-assign head to this newNode
  *head = newNode;
  printf ("\n%d Inserted\n", newNode->data);
}
```

```
void display (struct Node *node)
{
  printf ("\nLinked List: ");

  // as linked list will end when Node is Null
  while (node != NULL)
    {
      printf ("%d ", node->data);
      node = node->next;
    }
  printf ("\n");
}

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

  // Need '&' i.e. address as we need to change head
  insertStart (&head, 100);
  insertStart (&head, 80);
  insertStart (&head, 60);
  insertStart (&head, 40);
  insertStart (&head, 20);

  // No Need for '&' as not changing head in display operation
  display (head);

  deleteStart (&head);
  deleteStart (&head);
  display (head);

  return 0;
}
```

The provided code is a C program that implements basic operations on a singly linked list. Below is a breakdown of the functionalities and operations performed within the program:

1. **Data Structure Definition:**
   - The program defines a structure Node that represents a node in a linked list. Each node contains an integer data element and a pointer to the next node in the list.
2. **Function Definitions:**
   - deleteStart(struct Node **head): Deletes the first node of the linked list. If the list is empty, it prints a message indicating that the list is empty.
   - insertStart(struct Node **head, int data): Inserts a new node with the given data at the beginning of the linked list.

- display(struct Node *node): Displays the elements of the linked list.

3. **Main Function:**
   - The main function initializes the linked list with some nodes and performs various operations such as inserting nodes at the beginning, displaying the linked list, deleting the first node, and then displaying the updated linked list.
4. **Output:** The output of the program displays the linked list at different stages, including after initial insertion and after nodes have been deleted from the start of the linked list.

The program enables inserting nodes at the start, deleting nodes from the start, and displaying the current state of the linked list.

1. **Creation of doubly linked list**

```
#include <stdio.h>

//Represent a node of the doubly linked list

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

//Represent the head and tail of the doubly linked list
struct node *head, *tail = NULL;

//addNode() will add a node to the list
void addNode(int data) {
   //Create a new node
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;

   //If list is empty
   if(head == NULL) {
      //Both head and tail will point to newNode
      head = tail = newNode;
      //head's previous will point to NULL
      head->previous = NULL;
      //tail's next will point to NULL, as it is the last node of the list
      tail->next = NULL;
   }
```

```
    else {
        //newNode will be added after tail such that tail's next will point to newNode
        tail->next = newNode;
        //newNode's previous will point to tail
        newNode->previous = tail;
        //newNode will become new tail
        tail = newNode;
        //As it is last node, tail's next will point to NULL
        tail->next = NULL;
    }
}

//display() will print out the nodes of the list
void display() {
    //Node current will point to head
    struct node *current = head;
    if(head == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Nodes of doubly linked list: \n");
    while(current != NULL) {
        //Prints each node by incrementing pointer.
        printf("%d ", current->data);
        current = current->next;
    }
}

int main()
{
    //Add nodes to the list
    addNode(1);
    addNode(2);
    addNode(3);
    addNode(4);
    addNode(5);

    //Displays the nodes present in the list
    display();
```

```
    return 0;
    }
```

C program. Let's break it down:

c

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

This line includes the standard input/output library in our program. It provides functions like printf and scanf for input/output operations.

c

```c
struct node{
    int data;
    struct node *previous;
    struct node *next;
};
```

This section defines a structure called "node" to represent a node in the doubly linked list. It contains an integer data field and two pointers: "previous" and "next," which point to the previous and next nodes in the list, respectively.

c

```c
struct node *head, *tail = NULL;
```

Here, we declare two pointers named "head" and "tail" of type "struct node." These pointers will serve as the head and tail of the doubly linked list and are initially set to NULL to indicate an empty list.

c

```c
void addNode(int data) {
    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = data;
```

The addNode function is defined with a parameter "data" representing the value to be stored in the new node. Inside the function, a new node is created using the malloc function to allocate memory for the node based on the size of the "struct node."

c

```c
if(head == NULL) {
    head = tail = newNode;
    head->previous = NULL;
    tail->next = NULL;
}
else {
    tail->next = newNode;
    newNode->previous = tail;
```

```
    tail = newNode;
    tail->next = NULL;
}
```

This block of code adds the new node to the doubly linked list. If the list is initially empty (head is NULL), the new node becomes both the head and tail. Otherwise, the new node is added after the current tail, and the tail is updated to point to the new node.

c

```
void display() {
    struct node *current = head;
    if(head == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Nodes of doubly linked list: \n");
    while(current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
}
```

The display function is defined to print the elements of the doubly linked list. It initializes a pointer "current" to the head of the list and then traverses the list, printing the data of each node.

c

```
int main()
{
    addNode(1);
    addNode(2);
    addNode(3);
    addNode(4);
    addNode(5);
    display();
    return 0;
}
```

In the main function, nodes with values 1, 2, 3, 4, and 5 are added to the list using the addNode function, and then the display function is called to print the list.

Program for Concatenate two Linked Lists in C Language:

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

```
#include <stdlib.h>
struct Node
{
    int data;
    struct Node *next;
} *temp = NULL, *first = NULL, *second = NULL;

struct Node* Create(int A[], int n)
{
    int i;
    struct Node *t, *last;
    temp = (struct Node *) malloc(sizeof(struct Node));
    temp->data = A[0];
    temp->next = NULL;
    last = temp;

    for (i = 1; i < n; i++)
    {
        t = (struct Node *) malloc(sizeof(struct Node));
        t->data = A[i];
        t->next = NULL;
        last->next = t;
        last = t;
    }
    return temp;
}

void Display(struct Node *p)
{
    while (p != NULL)
    {
        printf ("%d ", p->data);
        p = p->next;
    }
}


void Concat(struct Node *first, struct Node *second)
{
    struct Node *p = first;
```

```
    while (p->next != NULL)
    {
        p = p->next;
    }
    p->next = second;
    second = NULL;
}

int main()
{
    int A[] = { 9, 7, 4, 3 };
    int B[] = { 2, 5, 6, 8 };
    first = Create(A, 4);
    second = Create(B, 4);

    printf ("1st Linked List: ");
    Display (first);

    printf ("\n2nd Linked List: ");
    Display (second);

    Concat (first, second);

    printf ("\n\nConcantenated Linked List: \n");
    Display (first);
    return 0;
}
```

This C program implements operations on linked lists. Here's a line-by-line breakdown:

c

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node *next;
} *temp = NULL, *first = NULL, *second = NULL;
```

- The program starts by including necessary libraries and defining a structure Node to represent a node in the linked list.

- It also declares pointers temp, first, and second, initializing them to NULL.

```c
struct Node* Create(int A[], int n)
{
   int i;
   struct Node *t, *last;
   temp = (struct Node *) malloc(sizeof(struct Node));
   temp->data = A[0];
   temp->next = NULL;
   last = temp;

   for (i = 1; i < n; i++)
   {
      t = (struct Node *) malloc(sizeof(struct Node));
      t->data = A[i];
      t->next = NULL;
      last->next = t;
      last = t;
   }
   return temp;
}
```

- The Create function takes an array A and its length n as input and creates a linked list with the elements of A.
- It iterates through the array, creates nodes for each element, and links them to form a linked list. It returns a pointer to the first node of the list.

```c
void Display(struct Node *p)
{
   while (p != NULL)
   {
      printf ("%d ", p->data);
      p = p->next;
   }
}
```

- The Display function prints the elements of the linked list pointed to by p.

```c
void Concat(struct Node *first, struct Node *second)
{
   struct Node *p = first;
```

```
    while (p->next != NULL)
    {
        p = p->next;
    }
    p->next = second;
    second = NULL;
}
```

- The Concat function concatenates the linked list second to the end of the linked list first.

```
c
int main()
{
    // Code to create and concatenate linked lists
}
```

- In the main function, it creates two arrays A and B, creates linked lists using these arrays, displays the original lists, concatenates them, and displays the concatenated list.

Overall, the program creates, displays, and concatenates linked lists.

c.C program to split a singly linked list into two halves.

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

// Structure defining a node in a singly linked list
struct Node {
    int data;          // Data stored in the node
    struct Node *next;  // Pointer to the next node
};

// Function to create a new node in the singly linked list
struct Node *newNode(int data) {
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node)); // Allocate memory for a new node
    temp->data = data;  // Assign data to the new node
    temp->next = NULL;  // Initialize the next pointer as NULL
    return temp;  // Return the new node
}
```

```
// Function to print the elements of the linked list
void printList(struct Node *head) {
    while (head != NULL) {
        printf("%d ", head->data);  // Print the data of the current node
        head = head->next;  // Move to the next node
    }
    printf("\n");
}

// Function to split the singly linked list into two halves
void splitList(struct Node *head, struct Node **front_ref, struct Node **back_ref) {
    struct Node *slow = head;  // Initialize slow pointer to the head of the list
    struct Node *fast = head->next;  // Initialize fast pointer to the second node

    while (fast != NULL) {
        fast = fast->next;  // Move fast pointer by one step
        if (fast != NULL) {
            slow = slow->next;  // Move slow pointer by one step
            fast = fast->next;  // Move fast pointer by another step
        }
    }

    *front_ref = head;  // Set front_ref to the original head of the list
    *back_ref = slow->next;  // Set back_ref to the node next to the middle node
    slow->next = NULL;  // Break the link between the two halves
}

int main() {
    struct Node *temp;
    struct Node *head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    printf("Original List: ");
    printList(head);

    struct Node *firstHalf = NULL;
    struct Node *secondHalf = NULL;
```

```
    // Calling the splitList method for the first list
    splitList(head, &firstHalf, &secondHalf);
    printf("Split the said singly linked list into halves:\n");
    printf("First half: ");
    temp = firstHalf;
    printList(temp);
    printf("Second half: ");
    temp = secondHalf;
    printList(temp);

    // Creating a second list and performing the same splitting operation
    struct Node *head1 = newNode(1);
    head1->next = newNode(2);
    head1->next->next = newNode(3);
    head1->next->next->next = newNode(4);
    head1->next->next->next->next = newNode(5);
    head1->next->next->next->next->next = newNode(6);

    printf("\nOriginal List: ");
    printList(head1);

    firstHalf = NULL;
    secondHalf = NULL;

    // Calling the splitList method for the second list
    splitList(head1, &firstHalf, &secondHalf);
    printf("Split the said singly linked list into halves:\n");
    printf("First half: ");
    temp = firstHalf;
    printList(temp);
    printf("Second half: ");
    temp = secondHalf;
    printList(temp);

    return 0;
}
```

Let's break it down:

c

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

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

These lines include the standard input/output and standard library functions like malloc for dynamic memory allocation in our program.

c

```
struct Node {
    int data;           // Data stored in the node
    struct Node *next;  // Pointer to the next node
};
```

Here, a structure "Node" is defined to represent a node in a singly linked list. It contains an integer data field and a pointer "next" pointing to the next node in the list.

c

```
struct Node *newNode(int data) {
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node)); // Allocate
memory for a new node
    temp->data = data;  // Assign data to the new node
    temp->next = NULL;  // Initialize the next pointer as NULL
    return temp;  // Return the new node
}
```

This function "newNode" is defined to create a new node with the given data value. It allocates memory for the new node, initializes its data and next pointer, and returns the new node.

c

```
void printList(struct Node *head) {
    while (head != NULL) {
        printf("%d ", head->data);  // Print the data of the current node
        head = head->next;  // Move to the next node
    }
    printf("\n");
}
```

The "printList" function is defined to print the elements of the linked list starting from the given "head" node.

c

```
void splitList(struct Node *head, struct Node **front_ref, struct Node
**back_ref) {
```

```
    // Code to split the singly linked list into two halves
}
```

The "splitList" function is defined to split the singly linked list into two halves. It uses two pointers, "slow" and "fast", to achieve this.

c

```
int main() {
    // Code to create and split linked lists
}
```

In the main function, it creates two linked lists and splits each list into halves using the "splitList" function, and then prints the original and split lists.

Overall, this program demonstrates the creation of a singly linked list, splitting it into two halves, and printing the elements of the lists.