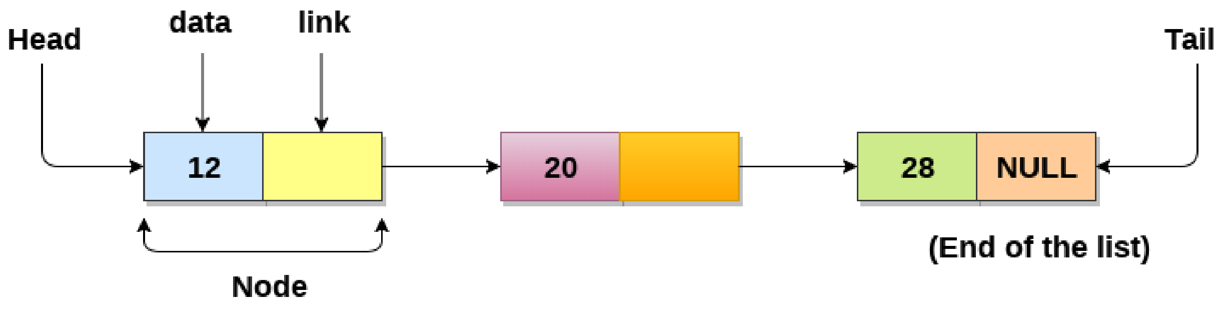
Unit – II

Syllabus:

- Singly linked lists: representation and operations
- Doubly linked lists
- Circular linked lists
- Comparing arrays and linked lists
- Applications of linked lists.

INTRODUCTION:

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



ARRAYS Vs LINKED LISTS:

- Array contains following limitations:
    - ➢ The size of array must be known in advance before using it in the program.
    - ➢ Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
    - ➢ All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.
- Linked list is the data structure which can overcome all the limitations of an array.
- Using linked list is useful because,
    - ➢ It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
    - ➢ Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows and shrinks as per the program's demand and limited to the available memory space.

- Singly linked list can be defined as the collection of ordered set of elements.
- The number of elements may vary according to need of the program.
- A node in the singly linked list consist of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- One way chain or singly linked list can be traversed only in one direction.
- Each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

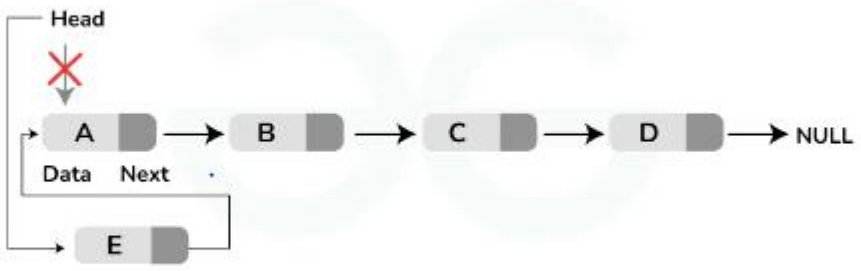
ADT OF LINKED LISTS:

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insert after a specified position	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .
4	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
5	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
6	Delete after a specified position	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
7	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
8	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .
9	Length	Find the length of the linked list.

Insert at beginning of a singly linked list

To insert a node at the beginning of a singly linked list, follow these steps:

- Create a new node using the given data.
- If the linked list's head is null, set the new node as the linked list's head and return.
- Set the new node's next pointer to the current head of the linked list.
- Set the linked list's head to point to the new node.

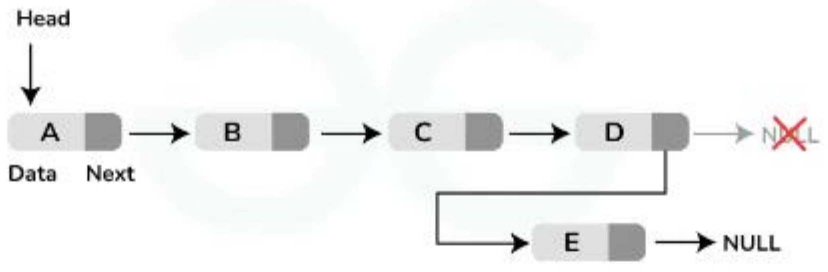


```
void insert_at_begin(struct Node** head_ref, int new_data)
{
    struct Node* new_node
        = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
     new_node->next = (*head_ref);
     (*head_ref) = new_node;
}
```

Insert at end of the singly linked list:

To insert a node at the end of a singly linked list, you can follow these steps:

1. Create a new node.
2. Store the head reference in a temporary variable.
3. Set the next pointer of the new node as NULL since it will be the last node.
4. If the Linked List is empty, make the new node as the head and return.
5. Else traverse till the last node.
6. Make the last node's next pointer the new node.

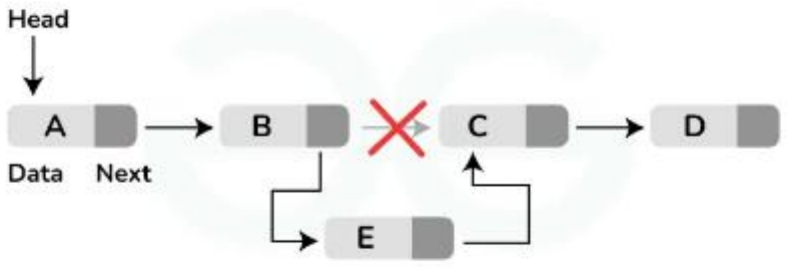


```
void insert_at_end(struct Node** head_ref, int new_data)
{
    struct Node* new_node
        = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head_ref;
    new_node->data = new_data;
    new_node->next = NULL;
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
    return;
}
```

Insert a node at specific position in singly linked list:

To insert a given data at a specified position, the below algorithm is to be followed:

- Traverse the Linked list upto *position-1* nodes.
- Once all the *position-1* nodes are traversed, allocate memory and the given data to the new node.
- Point the next pointer of the new node to the next of current node.
- Point the next pointer of current node to the new node.

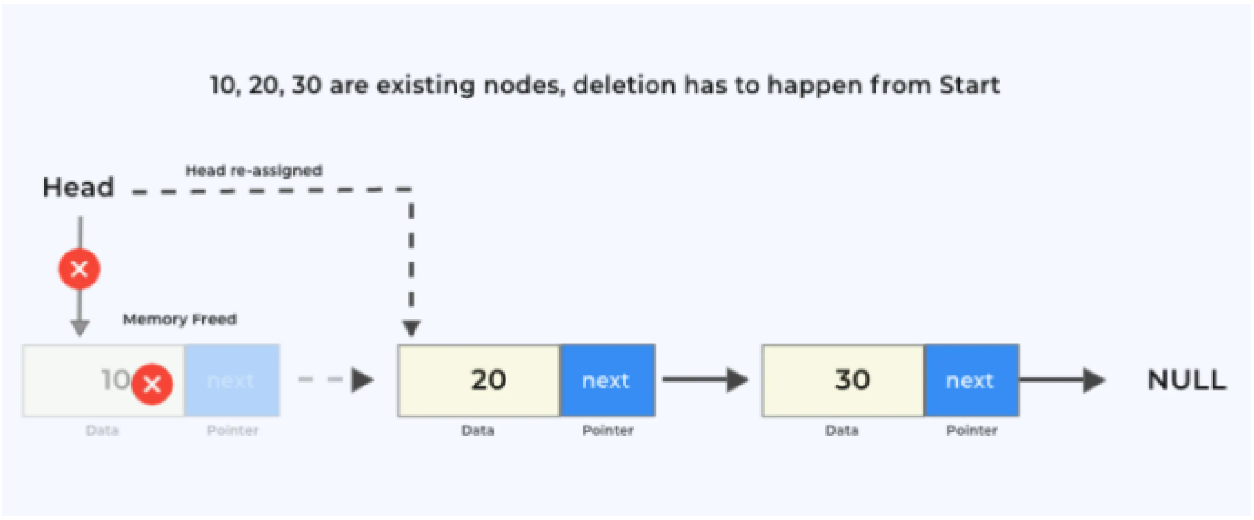


```
void insert_at_position(struct node **head_ref, int new_data, int position) {
struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
new_node->data = new_data;
new_node->next = NULL;
if (position == 1) {
  new_node->next = (*head_ref);
  (*head_ref) = new_node;
  return;
 }
struct Node *temp = *head_ref;
for (int i = 1; i < position - 1; i++) {
  temp = temp->next;
 }
new_node->next = temp->next;
temp->next = new_node;
}
```

Delete at beginning of a singly linked list

To delete a node at the beginning of a singly linked list, follow these steps:

1. Check if the linked list is empty. If it is, return.
2. Store the head node in a temporary variable.
3. Set the head node to the next node in the list.
4. Free the temporary variable.

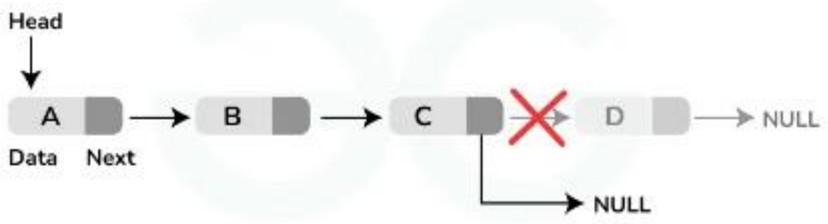


```
void delete_at_begin(struct node **head_ref) {
  if (*head_ref == NULL) {
    return;
  }
  struct node *temp = *head_ref;
  *head_ref = (*head_ref)->next;
  free(temp);
}
```

Delete at ending of a singly linked list

To delete a node at the ending of a singly linked list, follow these steps:

1. If the list has only one node, make the head node null.
2. Traverse to the end of the list.
3. Store the previous node, which is the second last node.
4. Change the next of the second last node to null.
5. Free or delete the memory of the last node.
6. The second last node becomes the last node

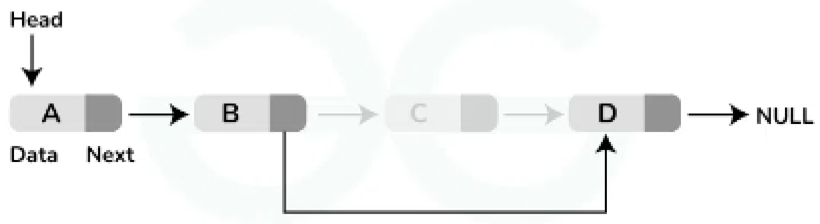


```
void delete_at_end(struct node **head_ref) {
    if (*head_ref == NULL) {
    return;
  }
  struct node *second_last = *head_ref;
  while (second_last->next->next != NULL) {
    second_last = second_last->next;
  }
  free(second_last->next);
  second_last->next = NULL;
}
```

Delete at specific position in a singly linked list

To delete a node at the specific position in a singly linked list, follow these steps:

1. Input: A pointer to the head node of the linked list and the value to be deleted.
2. If the linked list is empty, return NULL.
3. If the node to be deleted is the head node, set the head node to the next node and delete the original head node.
4. Otherwise, traverse the linked list from the head node until the node to be deleted is found.
5. If the node to be deleted is not found, return NULL.
6. Otherwise, set the previous node's next pointer to the node after the node to be deleted.
7. Delete the node to be deleted.
8. Return the head node of the linked list.



```
void delete_at_position(struct node **head_ref, int position) {
    if (*head_ref == NULL)
        return;
  struct node *temp = *head;
  if (position == 1) {
    *head_ref = temp->next;
    free(temp);
    return;
  }
  for (int i = 0; temp != NULL && i < position - 1; i++)
    temp = temp->next;
  if (temp == NULL || temp->next == NULL)
    return;
  struct node *next = temp->next->next;
  temp->next = next;
  free(temp->next);
}
```

Search an element in a singly linked list

To search an element in a singly linked list, follow these steps:

1. Start at the beginning of the list, at the first node.
2. While there is still a node to explore, check if the key matches the data in the current node.
3. If yes, return the index of the node.
4. If no, move to the next node in the list.
5. If the end of the list is reached and the key is not found, return -1.

```
int search(struct node *head_ref, int x) {
  struct node *current = head_ref;
  int index=0;
  while (current != NULL) {
    if (current->data == x) {
      return index;
    }
    current = current->next;
    index++;
  }
  return -1;
}
```

Traverse in forward direction in a singly linked list

To traverse in forward direction in a singly linked list, follow these steps:

1. Set a pointer to the head of the list.
2. While the pointer is not null, print the data in the current node and move the pointer to the next node.
3. Repeat step 2 until the pointer is null.

```
void traverse(struct Node* head_ref) {
  struct Node* temp = head_ref;
  while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
  }
}
```

Traverse in reverse direction in a singly linked list

To traverse in reverse direction in a singly linked list, follow these steps:

1. Create three pointers: current, next, and previous.
2. Initialize current to the head of the linked list.
3. Set next to the next node of current.
4. Set the next pointer of current to previous.
5. Set previous to current.
6. Set current to next.
7. Repeat steps 3-6 until current is null.

8. The previous pointer now points to the head of the reversed linked list.

```
static void reverse(struct Node** head_ref)
{
   struct Node* prev = NULL;
   struct Node* current = *head_ref;
   struct Node* next = NULL;
   while (current != NULL) {
      next = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
   *head_ref = prev;
}
```

Menu Driven Program to implement all operations in a singly linked list

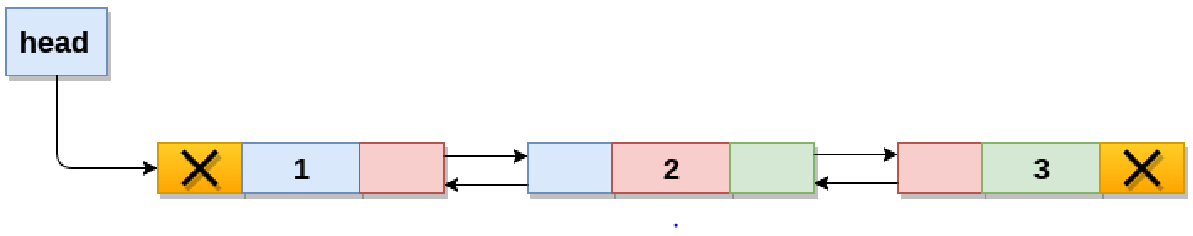
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
void insert_at_begin (struct node **head,int data);
void insert_at_end (struct node **head,int data);
void insert_at_position(struct node **head,int position,int data);
void delete_at_begin(struct node **head);
void delete_at_end(struct node **head);
void delete_at_position(struct node **head, int position);
int search(struct node **head,int key);
void traverse(struct node *head);
static void reverse_iter(struct node **head);
void reverse_rec(struct node *head);
int n,pos;
struct node *head=NULL;
void main ()
{
   int choice =0;
   while(choice != 11)
   {
      printf("\n\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from
Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an
element\n8.traverse\n9.iterative reverse\n10.recursive reverse\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         printf("Enter the node to be inserted:\n");
         scanf("%d",&n);
         insert_at_begin(&head,n);
```

```
        break;
        case 2:
        printf("Enter the node to be inserted:\n");
        scanf("%d",&n);
        insert_at_end(&head,n);
        break;
        case 3:
        printf("Enter the node to be inserted:\n");
        scanf("%d",&n);
        printf("Enter the position for insertion:\n");
        scanf("%d",&pos);
        insert_at_position(&head,pos,n);
        break;
        case 4:
        delete_at_begin(&head);
        break;
        case 5:
        delete_at_end(&head);
        break;
        case 6:
        printf("Enter the position from which element to be deleted:\n");
        scanf("%d",&pos);
        delete_at_position(&head,pos);
        break;
        case 7:
        int key,i;
        printf("Enter the element to be searched:\n");
        scanf("%d",&key);
        i=search(&head,key);
        if(i==-1){
          printf("Element not found\n");
        }
        else{
          printf("%d node found at %d location\n",key,i+1);
        }
        break;
        case 8:
        printf("Elements in the list are:\n");
        traverse(head);
        break;
        case 9:
        reverse_iter(&head);
        break;
        case 10:
        reverse_rec(head);
        break;
        case 11:
        exit(0);
        break;
        default:
        printf("Please enter valid choice..");
      }
   }
}
void insert_at_begin(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;
     printf("Node %d is inserted at beginning\n",new_data);
     return;
}
void insert_at_end(struct node** head_ref, int new_data)
{
     struct node* new_node
        = (struct node*)malloc(sizeof(struct node));
    new_node->data = new_data;
    struct node* last = *head_ref;
    new_node->next = NULL;
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = new_node;
     printf("Node %d is inserted at ending\n",new_data);
     return;
}
void insert_at_position(struct node** head_ref, int position, int new_data)
{
     struct node *newNode = (struct node *)malloc(sizeof(struct node));
  newNode->data = new_data;
  newNode->next = NULL;
  if (position == 1) {
    newNode->next = *head_ref;
    *head_ref = newNode;
    return;
  }
  struct node *prevNode = *head_ref;
  for (int i = 1; i < position - 1; i++) {
    prevNode = prevNode->next;
  }
  newNode->next = prevNode->next;
  prevNode->next = newNode;
     printf("Node %d is inserted at position %d \n",new_data,position);
     return;
}
void delete_at_begin(struct node** head_ref)
{
     struct node *temp = *head_ref;
     *head_ref = (*head_ref)->next;
     free(temp);
     return;
}
void delete_at_end(struct node** head_ref)
{
     if (*head_ref == NULL) {
    return;
  }
  struct node* secondLast = NULL;
  struct node* current = *head_ref;
  while (current->next != NULL) {
    secondLast = current;
    current = current->next;
  }
  if (secondLast == NULL) {
    *head_ref = NULL;
  } else {
    secondLast->next = NULL;
  }
  free(current);
     return;
}
void delete_at_position(struct node** head_ref,int position)
{
     if (*head_ref == NULL) {
    return;
  }
  struct node *temp = *head_ref;
  if (position == 1) {
    *head_ref = temp->next;
    free(temp);
    return;
  }
  for (int i = 0; temp != NULL && i < position - 1; i++) {
    temp = temp->next;
  }
  if (temp == NULL || temp->next == NULL) {
    return;
  }
  struct node *next = temp->next->next;
  temp->next = next;
  free(temp->next);
     return;
}

int search(struct node** head_ref,int key){
    struct node *current=*head_ref;
    int index=0;
    while (current != NULL) {
    if (current->data == key) {
      return index;
    }
    current = current->next;
    index++;
  }
  return -1;
}
void traverse(struct node* head_ref)
{
     while(head_ref!=NULL){
         printf("%d ",head_ref->data);
         head_ref=head_ref->next;
    }
}
Static void reverse_iter(struct node** head_ref)
{
   struct node* prev = NULL;
   struct node* current = *head_ref;
   struct node* next = NULL;
   while (current != NULL) {
       next = current->next;
       current->next = prev;
        prev = current;
       current = next;
   }
   *head_ref = prev;
    return;
}
void reverse_rec(struct node* head_ref)
{
    if(head_ref==NULL){
        return;
    }
    reverse_rec(head_ref->next);
    printf("%d ",head_ref->data);
}
```

DOUBLY LINKED LIST:

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).



Doubly Linked List

Singly Vs Doubly Linked List:

- In singly linked list traversing in reverse order takes more time than in doubly linked list.
- In singly linked list for inserting and deleting a node from end we need to traverse through the end of the list. But in doubly linked list we can do it easily.

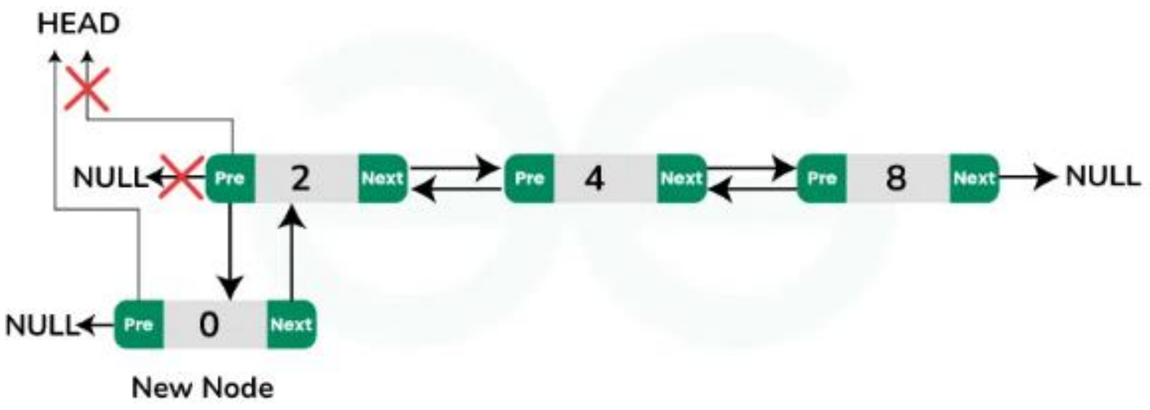
Operations of DLL:

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insert at a specified position	Adding the node into the linked list at a specified position.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Delete at a specified position	Removing the node which is present at a specified position.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insert at beginning of a doubly linked list

To insert a node at the beginning of a doubly linked list, follow these steps:

1. Create a new node and assign the data to be inserted to it.
2. Set the previous pointer of the new node to null.
3. Set the next pointer of the new node to the current head node.
4. Set the previous pointer of the current head node to the new node.
5. Update the head pointer to point to the new node.

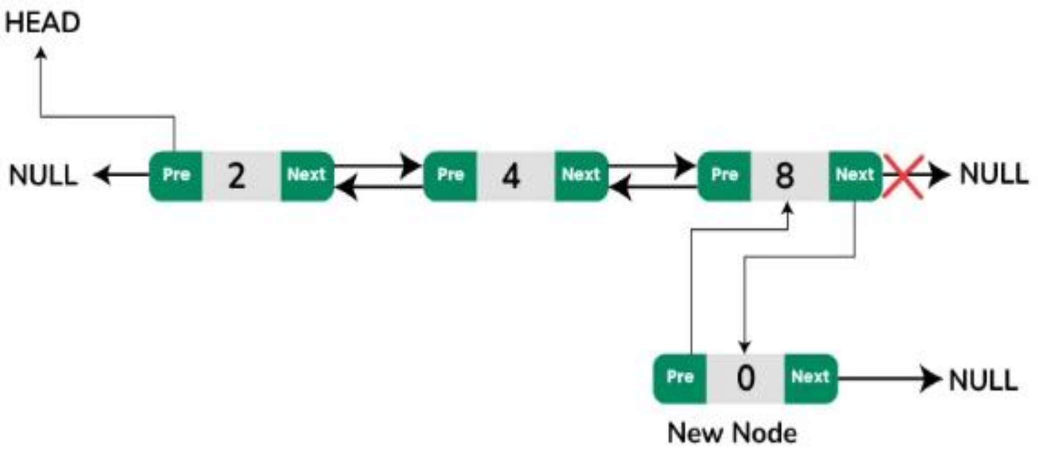


```
void insert_at_begin(struct Node** head_ref, int new_data)
{
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   new_node->data = new_data;
   new_node->next = (*head_ref);
   new_node->prev = NULL;
   if ((*head_ref) != NULL)
      (*head_ref)->prev = new_node;
   (*head_ref) = new_node;
}
```

Insert at end of the doubly linked list:

To insert a node at the end of a doubly linked list, you can follow these steps:

- Create a new node (say **new_node**).
- Put the value in the new node.
- Make the next pointer of **new_node** as null.
- If the list is empty, make **new_node** as the head.
- Otherwise, travel to the end of the linked list.
- Now make the next pointer of last node point to **new_node**.
- Change the previous pointer of **new_node** to the last node of the list.

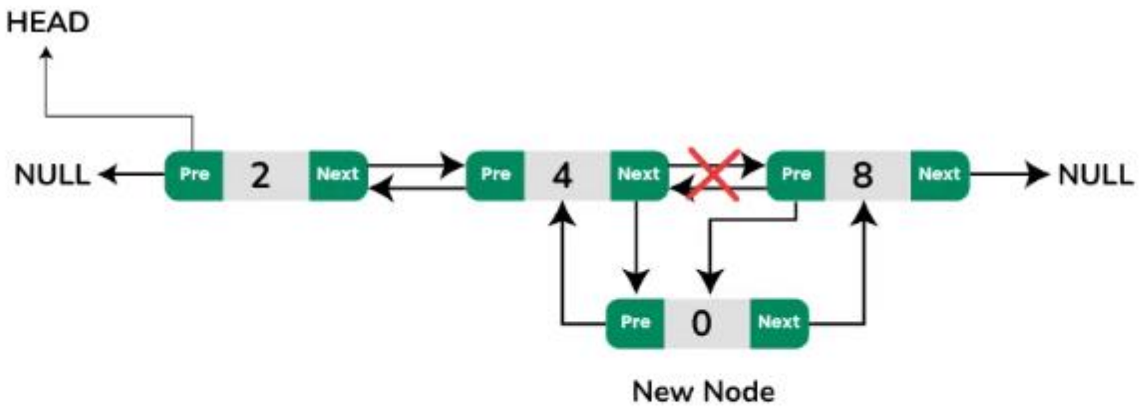


```
void insert_at_end(struct Node** head_ref, int new_data)
{
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
struct Node* last = *head_ref;
new_node->data = new_data;
new_node->next = NULL
if (*head_ref == NULL) {
   new_node->prev = NULL;
   *head_ref = new_node;
    return;
 }
while (last->next != NULL)
   last = last->next;
last->next = new_node;
new_node->prev = last;
return;
}
```

Insert a node at specific position in doubly linked list:

To insert a given data at a specified position, the below algorithm is to be followed:

1. Create a new node and assign it the data value you want to insert.
2. Traverse the linked list until you reach the node before the position where you want to insert the new node.
3. Set the new node's next pointer to the node after the current node.
4. Set the new node's previous pointer to the current node.
5. Set the current node's next pointer to the new node.
6. Set the previous node's next pointer to the new node.



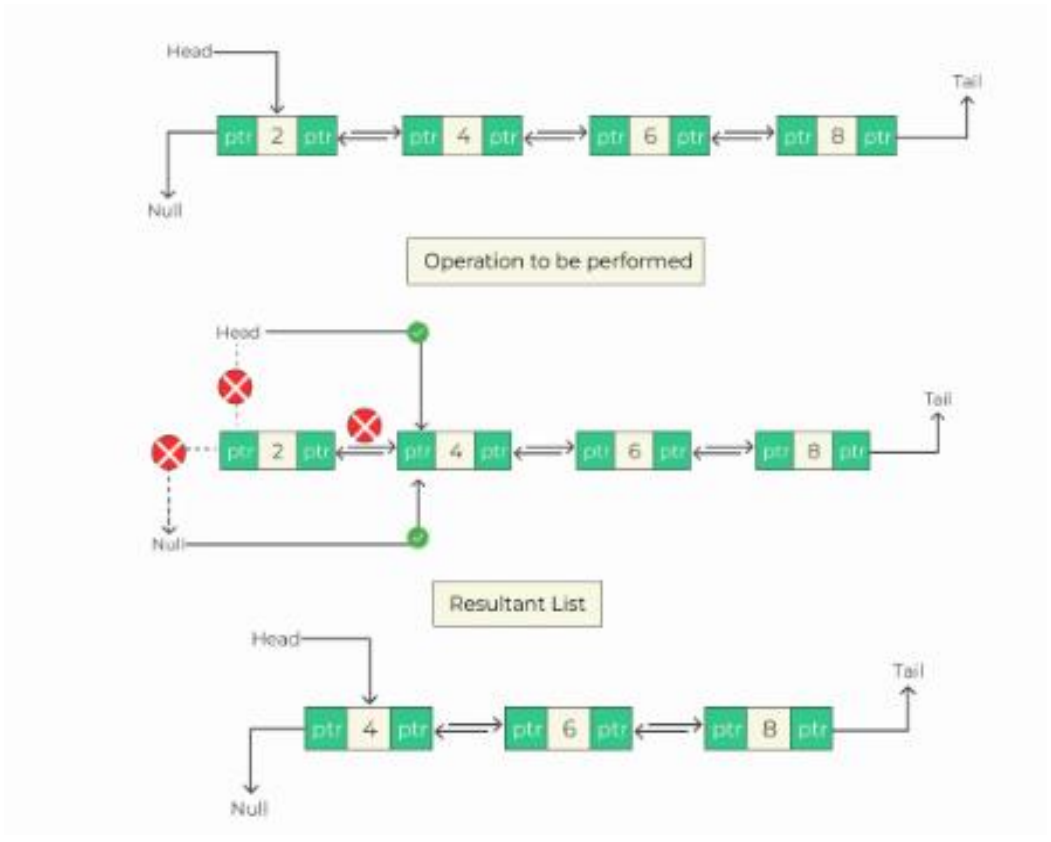
```
  void insert_at_position(struct node **head_ref, int new_data, int position) {
  struct node *new_node = (struct node *)malloc(sizeof(struct node));
new_node->data = new_data;
new_node->prev = NULL;
new_node->next = NULL;

if (position == 1) {
new_node->next = *head_ref;
(*head_ref)->prev = new_node;
(*head_ref) = new_node;
}
else {
struct node *temp = *head_ref;
for (int i = 1; i < position - 1; i++) {
   temp = temp->next;
}
new_node->next = temp->next;
temp->next->prev = new_node;
temp->next = new_node;
new_node->prev = temp;
}
}
```

Delete at beginning of a doubly linked list

To delete a node at the beginning of a doubly linked list, follow these steps:

1. Check if there is only one node
2. If there is one node, assign head to NULL
3. Free memory
4. Else, assign head to next node in the list
5. Assign prev pointer of head to NULL
6. Free memory

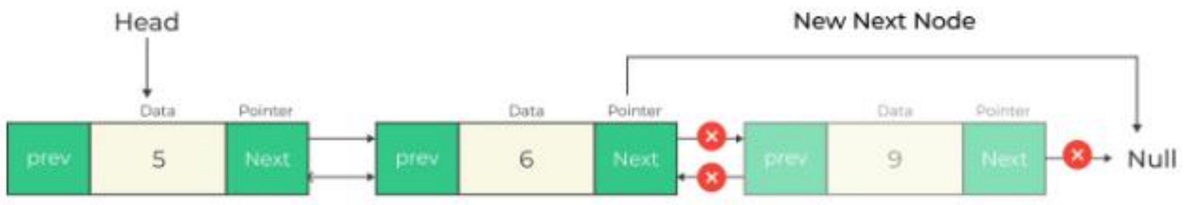


```
void delete_at_begin(struct node **head_ref) {
  if (*head_ref == NULL) {
    return;
}
struct node* temp = *head_ref;
(*head_ref)= (*head_ref)->next;
if (*head_ref != NULL) {
   (*head_ref)->prev = NULL;
}
free(temp);
}
```

Delete at ending of a doubly linked list

To delete a node at the ending of a doubly linked list, follow these steps:

1. Check if the list is empty. If it is, then there is nothing to delete and we can return.
2. If the list has only one node, then we can simply delete that node and return.
3. Otherwise, we need to find the last node in the list. We can do this by traversing the list until we reach a node whose next pointer is null.
4. Once we have found the last node, we need to update the previous pointer of the second-to-last node to point to null.
5. We can then free the memory allocated to the last node.



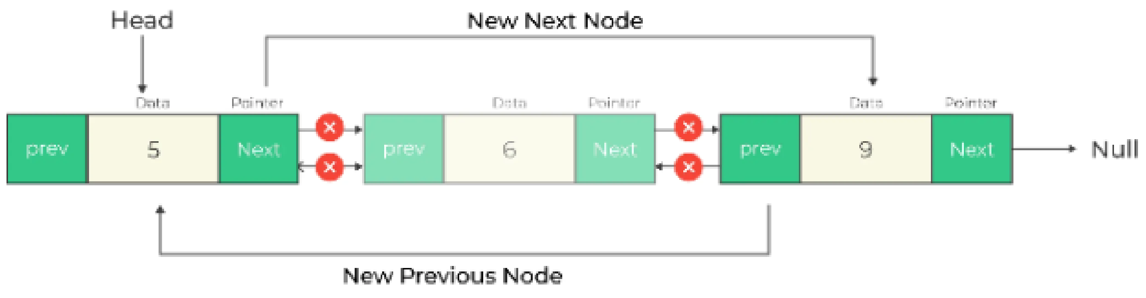
```
void delete_at_end(struct node **head_ref) {
struct Node *tempNode = *head_ref;
if (*head == NULL) {
   return;
}
if (tempNode->next == NULL) {
  *head = NULL;
   return;
}
while (tempNode->next != NULL)
   tempNode = tempNode->next;
struct Node *secondLast = tempNode->prev;
secondLast->next = NULL;
free (tempNode);
}
```

Delete at specific position in a doubly linked list

To delete a node at the specific position in a doubly linked list, follow these steps:

1. Check if the linked list is empty. If it is, then there is nothing to delete and we can return.
2. Check if the position to be deleted is less than 1 or greater than the length of the linked list. If it is, then we can return an error message.

3. Find the node at the given position. We can do this by traversing the linked list and keeping track of the current node and the previous node.
4. Once we have found the node to be deleted, we need to update the previous node's next pointer to point to the next node, and we need to update the next node's previous pointer to point to the previous node.
5. We can then free the memory allocated to the node to be deleted.



```
void delete_at_position(struct node **head_ref, int position) {
   if (*head_ref == NULL)
      return;
   if (position == 1) {
      *head_ref = *head_ref->next;
       if (*head_ref != NULL)
          *head_ref->prev = NULL;
       free(*head_ref);
       return;
   }
   struct node *current = *head_ref;
   for (int i = 1; current != NULL && i < position - 1; i++)
     current = current->next;
   if (current == NULL || current->next == NULL)
      return;
   struct node *next = current->next;
   current->next = next->next;
   if (next->next == NULL)
      current->next = NULL;
   next->next->prev = current;
   free(next);
}
```

Search an element in a doubly linked list

To search an element in a doubly linked list, follow these steps:

- Start at the beginning of the list, at the first node.
- While there is still a node to explore, check if the key matches the data in the current node.
- If yes, return the index of the node.
- If no, move to the next node in the list.

- If the end of the list is reached and the key is not found, return -1.

```
int search(struct node *head_ref, int x) {
  struct node *current = head_ref;
  int index=0;
  while (current != NULL) {
    if (current->data == x) {
      return index;
    }
    current = current->next;
    index++;
  }
  return -1;
}
```

Traverse in forward direction in a doubly linked list

To traverse in forward direction in a doubly linked list, follow these steps:

1. Start at the head node.
2. If the current node is null, then the traversal is complete.
3. Print the data of the current node.
4. Move to the next node.
5. Repeat steps 3 and 4 until the current node is null.

```
void traverse_forward(struct node **head_ref) {
    struct node *current = *head_ref;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
}
```

Traverse in reverse direction in a doubly linked list

To traverse in reverse direction in a doubly linked list, follow these steps:

- Take a pointer to point to head of the doubly linked list.
- Now, start traversing through the linked list till the end.
- After reaching last node, start traversing in backward direction and simultaneously print the node->data.

```
 void traverse_reverse(struct Node** head_ref)
 {
     struct Node* tail = *head_ref;
       while (tail->next != NULL) {
         tail = tail->next;
     }
      while (tail != *head_ref) {
         printf("%d ",tail->data);
         tail = tail->prev;
     }
     printf("%d ",tail->data);
 }
```

Menu Driven Program to implement all operations in a doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
   struct node *prev;
};
void insert_at_begin (struct node **head,int data);
void insert_at_end (struct node **head,int data);
void insert_at_position(struct node **head,int position,int data);
void delete_at_begin(struct node **head);
void delete_at_end(struct node **head);
void delete_at_position(struct node **head, int position);
int search(struct node **head,int key);
void traverse(struct node *head);
static void reverse_iter(struct node **head);
void reverse_rec(struct node *head);
int n,pos;
struct node *head=NULL;
void main ()
{
   int choice =0;
   while(choice != 11)
   {
      printf("\n\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from
Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an
element\n8.traverse\n9.iterative reverse\n10.recursive reverse\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         printf("Enter the node to be inserted:\n");
         scanf("%d",&n);
         insert_at_begin(&head,n);
         break;
         case 2:
         printf("Enter the node to be inserted:\n");
         scanf("%d",&n);
         insert_at_end(&head,n);
         break;
         case 3:
         printf("Enter the node to be inserted:\n");
         scanf("%d",&n);
         printf("Enter the position for insertion:\n");
         scanf("%d",&pos);
         insert_at_position(&head,pos,n);
         break;
         case 4:
         delete_at_begin(&head);
         break;
         case 5:
         delete_at_end(&head);
         break;
         case 6:
         printf("Enter the position from which element to be deleted:\n");
         scanf("%d",&pos);
         delete_at_position(&head,pos);
         break;
         case 7:
         int key,i;
         printf("Enter the element to be searched:\n");
         scanf("%d",&key);
         i=search(&head,key);
         if(i==-1){
             printf("Element not found\n");
         }
         else{
             printf("%d node found at %d location\n",key,i+1);
         }
         break;
         case 8:
         printf("Elements in the list are:\n");
         traverse(head);
         break;
         case 9:
         reverse_iter(&head);
         break;
         case 10:
         reverse_rec(head);
         break;
         case 11:
         exit(0);
         break;
         default:
         printf("Please enter valid choice..");
      }
   }
}
void insert_at_begin(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);
   new_node->prev = NULL;
   if ((*head_ref) != NULL)
      (*head_ref)->prev = new_node;
   (*head_ref) = new_node;
    printf("Node %d is inserted at beginning\n",new_data);
    return;
}
void insert_at_end(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) {
   new_node->prev = NULL;
   *head_ref = new_node;
 }
while (last->next != NULL)
   last = last->next;
last->next = new_node;
new_node->prev = last;
 printf("Node %d is inserted at ending\n",new_data);
    return;
}
void insert_at_position(struct node** head_ref, int position, int new_data)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
new_node->data = new_data;
new_node->prev = NULL;
new_node->next = NULL;

if (position == 1) {
new_node->next = *head_ref;
(*head_ref)->prev = new_node;
(*head_ref) = new_node;
} else {
struct node *temp = *head_ref;
for (int i = 1; i < position - 1; i++) {
temp = temp->next;
}
new_node->next = temp->next;
temp->next->prev = new_node;
temp->next = new_node;
new_node->prev = temp;
}
printf("Node %d is inserted at position %d \n",new_data,position);
    return;
}
void delete_at_begin(struct node** head_ref)
{
 if (*head_ref == NULL) {
return;
}
struct node* temp = *head_ref;
(*head_ref)= (*head_ref)->next;
if (*head_ref != NULL) {
(*head_ref)->prev = NULL;
}
free(temp);
return;
}
void delete_at_end(struct node** head_ref)
{
struct node *tempNode = *head_ref;
    if (*head_ref == NULL) {
return;
}
```

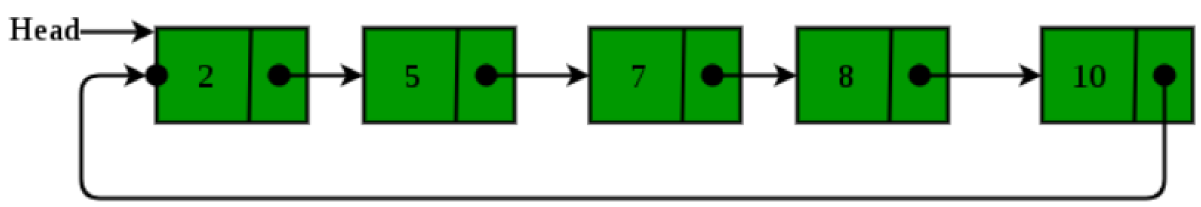
```
if (tempNode->next == NULL) {
  *head_ref = NULL;
return;
}
while (tempNode->next != NULL)
    tempNode = tempNode->next;
struct node *secondLast = tempNode->prev;
secondLast->next = NULL;
free (tempNode);
}
     return;
}
void delete_at_position(struct node** head_ref,int position)
{
     if (*head_ref == NULL)
    return;
if (position == 1) {
    *head_ref = *head_ref->next;
    if (*head_ref != NULL)
        *head_ref->prev = NULL;
    free(*head_ref);
    return;
}
struct node *current = *head_ref;
for (int i = 1; current != NULL && i < position - 1; i++)
      current = current->next;
if (current == NULL || current->next == NULL)
    return;
struct node *next = current->next;
current->next = next->next;
if (next->next == NULL)
   current->next = NULL;
next->next->prev = current;
free(next);
     return;
}

int search(struct node** head_ref,int key){
    struct node *current=*head_ref;
    int index=0;
    while (current != NULL) {
    if (current->data == key) {
      return index;
     }
    current = current->next;
    index++;
  }
  return -1;
}
void traverse(struct node** head_ref)
{
     struct node *current = *head_ref;
while (current != NULL) {
     printf("%d ", current->data);
    current = current->next;
}
```

```
}
static void reverse_iter(struct node** head_ref)
{
    struct node* tail = *head_ref;
     while (tail->next != NULL) {
       tail = tail->next;
    }
    while (tail != *head_ref) {
       printf("%d ",tail->data);
       tail = tail->prev;
    }
   printf("%d ",tail->data);
}
void reverse_rec(struct node* head_ref)
{
    if(head_ref==NULL){
        return;
    }
    reverse_rec(head_ref->next);
    printf("%d ",head_ref->data);
}
```

<u>**CIRCULAR LINKED LIST:**</u>

- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.
- We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly liked list has no beginning and no ending.
- There is no null value present in the next part of any of the nodes.



- Circular linked list are mostly used in task maintenance in operating systems.
- There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

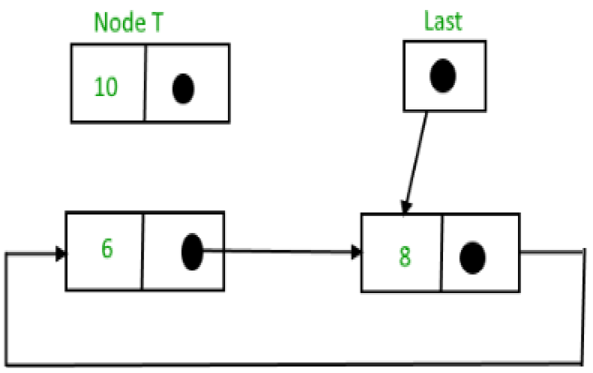
ADT of CIRCULAR LINKED LIST:

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insert at a specified position	Adding the node into the linked list at a specified position.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Delete at a specified position	Removing the node which is present at a specified position.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

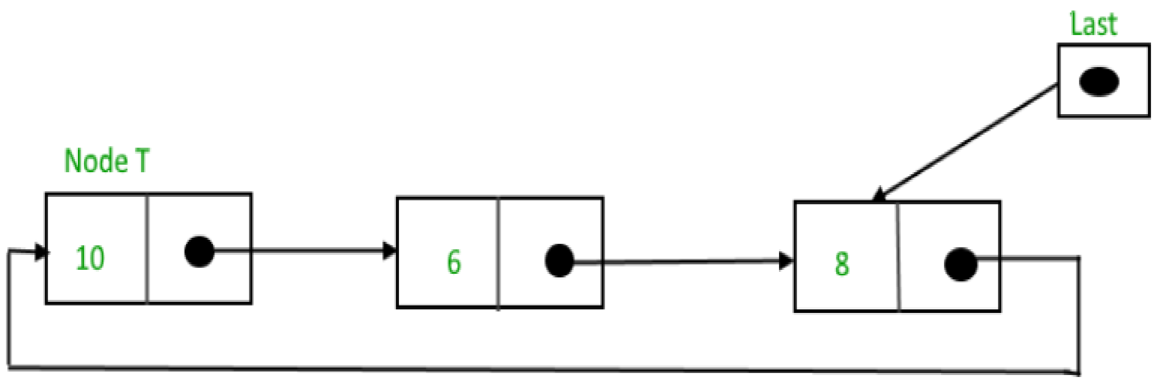
Insert at beginning of a circular linked list

To insert a node at the beginning of a circular linked list, follow these steps:

1. Create a new node with a value
2. Check if the list is empty
3. If the list is empty, set head to the new node and newNode → next to head
4. If the list is not empty, define a Node pointer 'temp' and initialize it with head
5. Move temp to its next node until it reaches the last node
6. Set newNode → next = head, head = newNode, and temp1 → next = head



After insertion.



```
void insert_at_begin(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);
   if ((*head_ref) == NULL) {
       (*head_ref) = new_node;
        new_node->next = (*head_ref);
   } else {
       struct Node *temp = (*head_ref);
       while (temp->next != (*head_ref)) {
            temp = temp->next;
        }
        temp->next = new_node;
   }
}
```

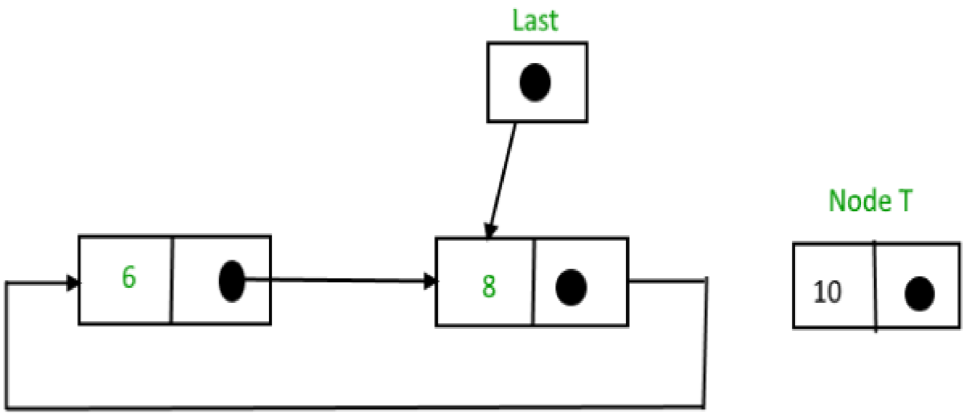
Insert at end of the circular linked list:

To insert a node at the end of a circular linked list, you can follow these steps:

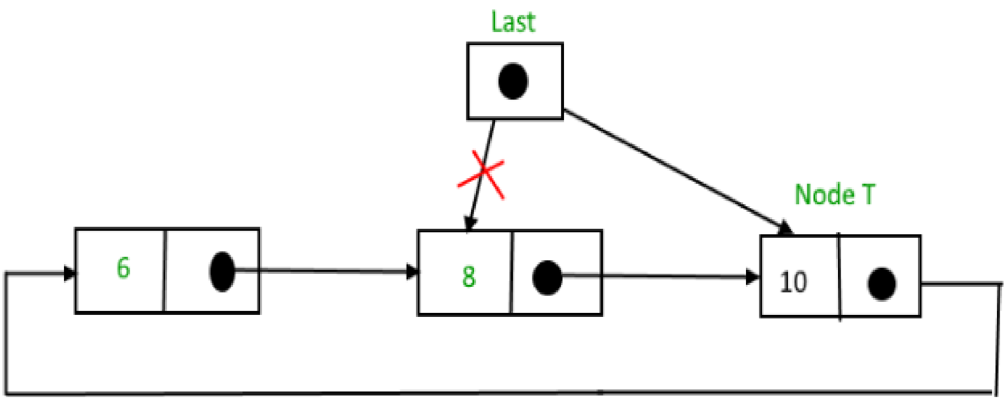
1. Allocate memory
2. If the list is empty:
   - Set head = ptr
   - Set ptr -> next = head
3. If the list contains at least one node:
   - Traverse the list to the last node
   - Set temp = head
   - While temp->next != head, set temp = temp->next

- At the end of the loop, set temp -> next = ptr and ptr -> next = head

4. Assign the new node next to the circular list
5. Assign the new node next to the front of the list
6. Assign tail next to the new node
7. Return the end node of the circular linked list



After insertion



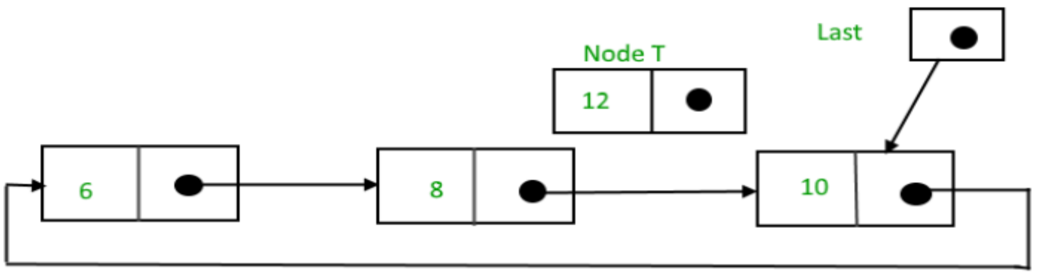
```
void insert_at_end(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
struct node* last = *head_ref;
new_node->data = new_data;
new_node->next = NULL;
if (*head_ref == NULL) {
    *head_ref = new_node;
     new_node->next=(*head_ref);
 }
while (last->next != (*head_ref))
    last = last->next;
last->next = new_node;
new_node->next = (*head_ref);
return;
```

}

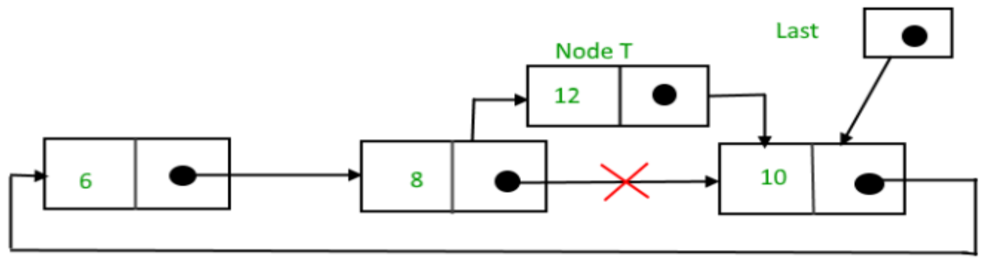
Insert a node at specific position in circular linked list:

To insert a given data at a specified position, the below algorithm is to be followed:

- Create a new node, say NewNode
- Traverse the list until the node with the given data is not found
- Store the node in P
- Make the next of NewNode point to the next of P, NewNode – > next = P – > next
- Make P point to the NewNode, P – > next = NewNode
- If the circular linked list is empty, terminate function
- Link the pointer of this new node to the node present at this place previously



After searching and insertion,

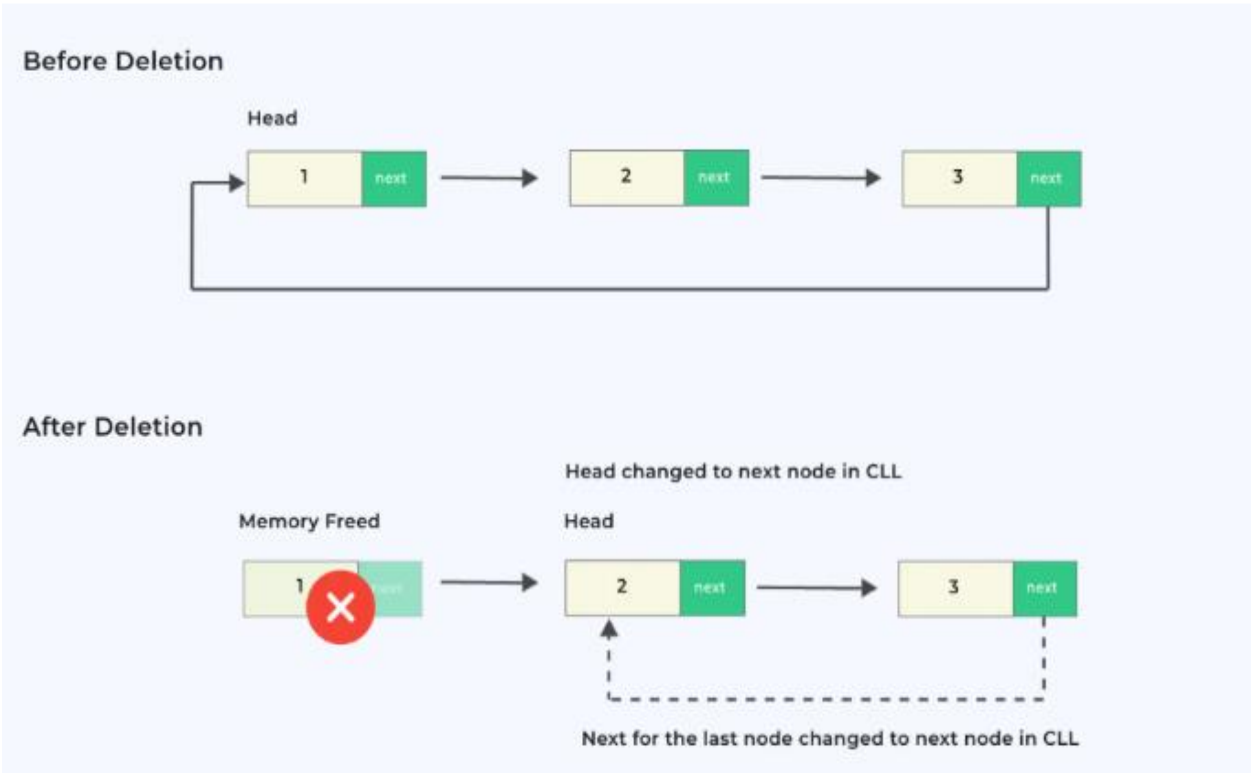


```
  void insert_at_position(struct node **head_ref, int new_data, int position) {
  struct node *new_node = (struct node *)malloc(sizeof(struct node));
new_node->data = new_data;
new_node->next = NULL;
if (position == 1) {
new_node->next = *head_ref;
(*head_ref) = new_node;
}
else {
struct node *temp = *head_ref;
for (int i = 1; i < position - 1; i++) {
temp = temp->next;
}
new_node->next = temp->next;
temp->next = new_node;
}
}
```

Delete at beginning of a circular linked list

To delete a node at the beginning of a circular linked list, follow these steps:

1. Define two node pointers
2. Initialize first with head
3. Traverse until pointer of p points to the head
4. Store head in the second node pointer
5. Make second node as new head of the list
6. Link the pointer of last node with new head
7. Free first


Before Deletion
Head
1
next
2
next
3
next
After Deletion
Head changed to next node in CLL
Memory Freed
Head
1
2
next
3
next
Next for the last node changed to next node in CLL

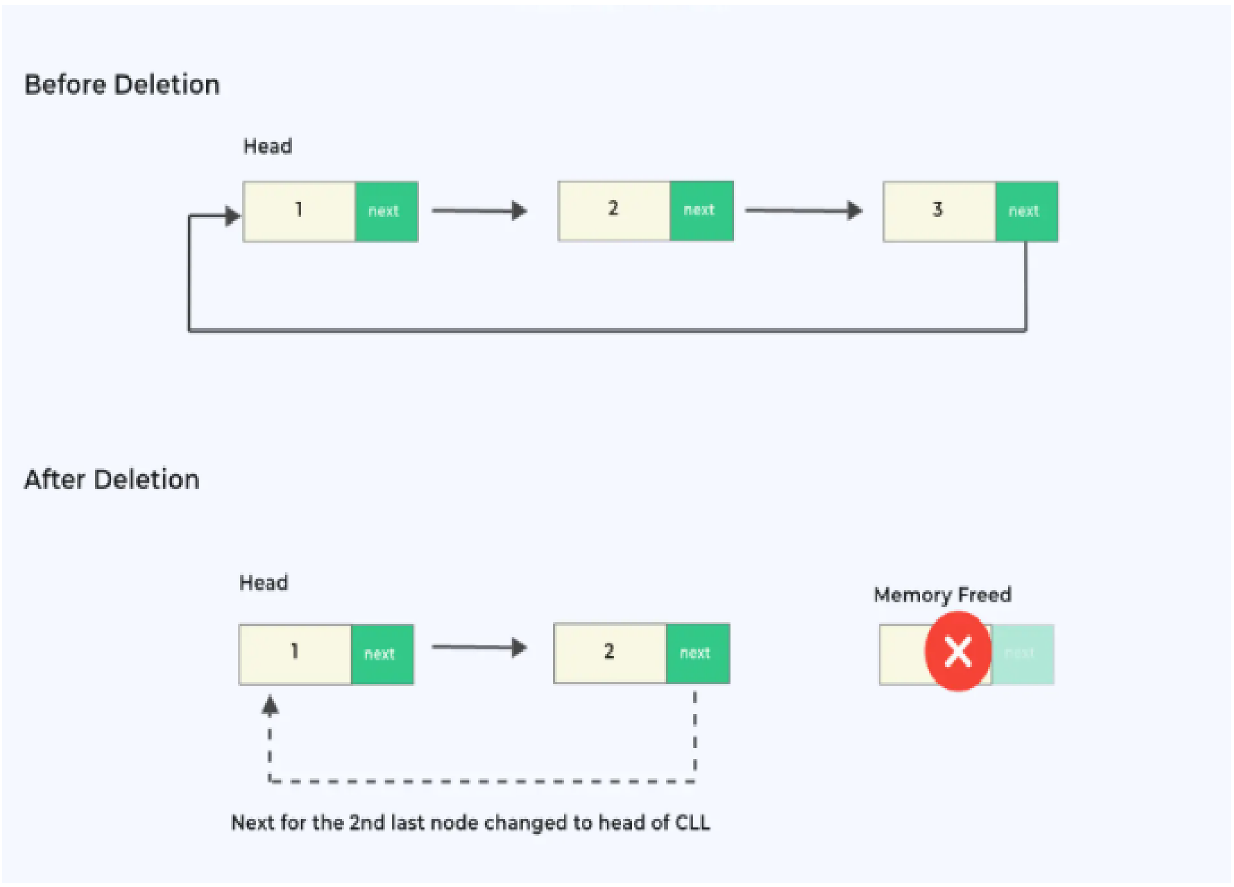
```
void delete_at_begin(struct node **head_ref) {
  if (*head_ref == NULL) {
return;
}
struct node* previous = *head_ref;
struct node* firstNode=*head_ref;
if (previous->next == previous) {
        *head_ref = NULL;
        return;
}
while (previous->next != (*head_ref)) {
        previous = previous->next;
}
previous->next = firstNode->next;
(*head_ref) = previous->next;
free(firstNode);
return;
```

}

Delete at ending of a circular linked list

To delete a node at the ending of a circuar linked list, follow these steps:

- check whether the head is null (empty list) then, it will return from the function as there is no node present in the list.
- If the list is not empty, it will check whether list has only one node.
- If the list has only one node, it will set both head and tail to null.
- If the list has more than one node then, iterate through the loop till current.next!= tail.
- Now, current will point to the node previous to tail. Make current as new tail and tail will point to head thus, deletes the node from last.

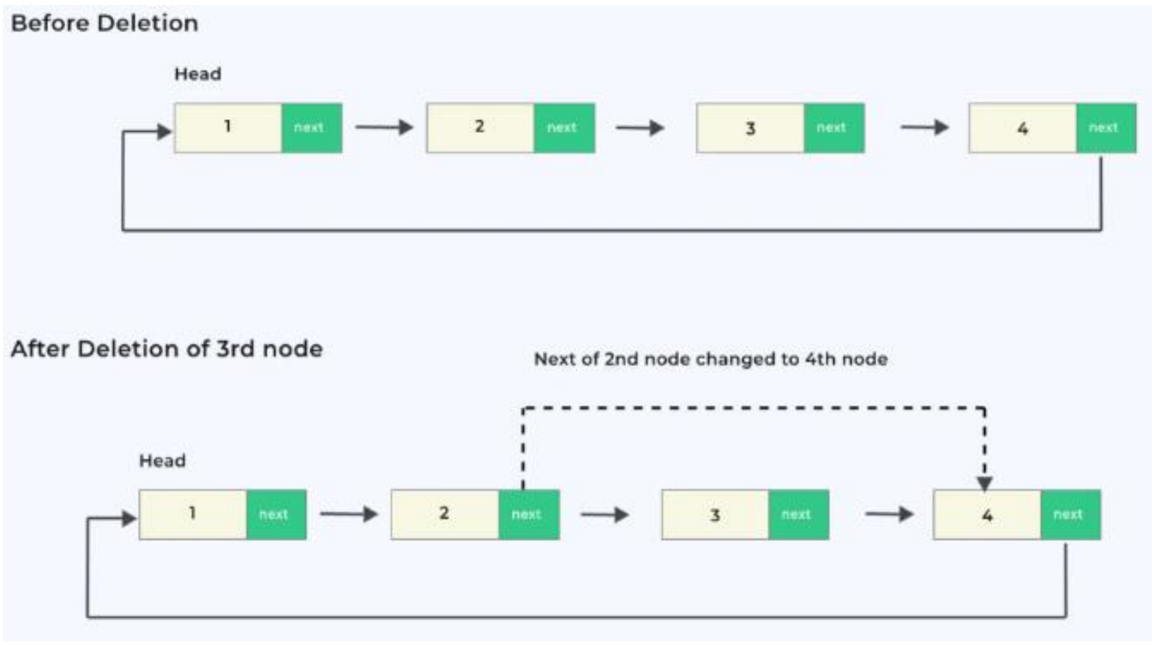


```
void delete_at_end(struct node **head_ref) {
   struct Node *current=*head_ref,*prev;
if (*head_ref) == NULL) {
return;
}
if (current->next == current) {
      *head_ref = NULL;
      return;
 }
while (current->next != (*head_ref)) {
      previous = current;
      current = current->next;
}
previous->next = current->next;
    *head = previous->next;
    free(current);
    return;
}
```

Delete at specific position in a circular linked list

To delete a node at the specific position in a circular linked list, follow these steps:

1. Initialize two pointers, curr and prev, and set curr to the head of the list.
2. Traverse the list until curr points to the node at the specified position.
3. If curr is the head of the list, set the head to curr->next.
4. If curr is the only node in the list, set the head to NULL.
5. Otherwise, set prev->next to curr->next.
6. Delete the node pointed to by curr.



```
void delete_at_position(struct node **head_ref, int position) {
    if (*head_ref == NULL)
    return;
if (position == 1) {
    *head_ref = *head_ref->next;
    free(*head_ref);
    return;
}
struct node *current = *head_ref;
Struct node *prev=NULL;
for (int i = 1; current != NULL && i < position - 1; i++){
      previous = current;
      current = current->next;
}
```

```
        previous->next = current->next;
        free(current);

}
```

Search an element in a circular linked list

To search an element in a circular linked list, follow these steps:

6. Start at the beginning of the list, at the first node.
7. While there is still a node to explore, check if the key matches the data in the current node.
8. If yes, return the index of the node.
9. If no, move to the next node in the list.
10. If the end of the list is reached and the key is not found, return -1.

```
int search(struct node **head_ref, int x) {
  struct node *current = *head_ref;
  int index=0;
  while (current != NULL) {
    if (current->data == x) {
      return index;
    }
    current = current->next;
    index++;
  }
  return -1;
}
```

Traverse in forward direction in a circular linked list

To traverse in forward direction in a circular linked list, follow these steps:

4. Set a pointer to the head of the list.
5. While the pointer is not null, print the data in the current node and move the pointer to the next node.
6. Repeat step 2 until the pointer is null.

```
void traverse(struct node* head_ref) {
  struct node* temp = head_ref;
  while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
  }
}
```

Traverse in reverse direction in a circular linked list

To traverse in reverse direction in a circular linked list, follow these steps:

- Create three pointers: current, next, and previous.
- Initialize current to the head of the linked list.
- Set next to the next node of current.
- Set the next pointer of current to previous.

- Set previous to current.
- Set current to next.
- Repeat steps 3-6 until current is null.
- The previous pointer now points to the head of the reversed linked list.

```
static void reverse(struct node** head_ref)
{
   if (*head_ref == NULL)
       return;
   struct node* prev = NULL;
   struct node* current = *head_ref;
   struct node* next;
   while (current != (*head_ref)){
       next = current->next;
       current->next = prev;
       prev = current;
       current = next;
   }
   (*head_ref)->next = prev;
   *head_ref = prev;
}
```

Menu Driven Program to implement all operations in a circular linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
void insert_at_begin (struct node **head,int data);
void insert_at_end (struct node **head,int data);
void insert_at_position(struct node **head,int position,int data);
void delete_at_begin(struct node **head);
void delete_at_end(struct node **head);
void delete_at_position(struct node **head, int position);
int search(struct node **head,int key);
void traverse(struct node *head);
static void reverse_iter(struct node **head);
void reverse_rec(struct node *head);
int n,pos;
struct node *head=NULL;
void main ()
{
   int choice =0;
   while(choice != 11)
   {
       printf("\n\n*********Main Menu*********\n");
       printf("\nChoose one option from the following list ...\n");
       printf("\n===============================================\n");
       printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from
Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an
```

```
element\n8.traverse\n9.iterative reverse\n10.recursive reverse\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            printf("Enter the node to be inserted:\n");
            scanf("%d",&n);
            insert_at_begin(&head,n);
            break;
            case 2:
            printf("Enter the node to be inserted:\n");
            scanf("%d",&n);
            insert_at_end(&head,n);
            break;
            case 3:
            printf("Enter the node to be inserted:\n");
            scanf("%d",&n);
            printf("Enter the position for insertion:\n");
            scanf("%d",&pos);
            insert_at_position(&head,pos,n);
            break;
            case 4:
            delete_at_begin(&head);
            break;
            case 5:
            delete_at_end(&head);
            break;
            case 6:
            printf("Enter the position from which element to be deleted:\n");
            scanf("%d",&pos);
            delete_at_position(&head,pos);
            break;
            case 7:
            int key,i;
            printf("Enter the element to be searched:\n");
            scanf("%d",&key);
            i=search(&head,key);
            if(i==-1){
                printf("Element not found\n");
            }
            else{
                printf("%d node found at %d location\n",key,i+1);
            }
            break;
            case 8:
            printf("Elements in the list are:\n");
            traverse(head);
            break;
            case 9:
            reverse_iter(&head);
            break;
            case 10:
            reverse_rec(head);
            break;
            case 11:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insert_at_begin(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);
    if ((*head_ref) == NULL) {
         (*head_ref) = new_node;
          new_node->next = (*head_ref);
    } else {
        struct node *temp = (*head_ref);
        while (temp->next != (*head_ref)) {
               temp = temp->next;
         }
         temp->next = new_node;
}
 printf("Node %d is inserted at beginning\n",new_data);
     return;
}
void insert_at_end(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;
     new_node->next=(*head_ref);
 }
while (last->next != (*head_ref))
    last = last->next;
last->next = new_node;
new_node->next = (*head_ref);
printf("Node %d is inserted at ending\n",new_data);
     return;
}
void insert_at_position(struct node** head_ref, int position, int new_data)
{
     struct node *new_node = (struct node *)malloc(sizeof(struct node));
new_node->data = new_data;
new_node->next = NULL;
if (position == 1) {
new_node->next = *head_ref;
(*head_ref) = new_node;
} else {
struct node *temp = *head_ref;
for (int i = 1; i < position - 1; i++) {
temp = temp->next;
```

```
}
new_node->next = temp->next;
temp->next = new_node;
}
     printf("Node %d is inserted at position %d \n",new_data,position);
     return;
}
void delete_at_begin(struct node** head_ref)
{
     if (*head_ref == NULL) {
return;
}
struct node* previous = *head_ref;
struct node* firstNode=*head_ref;
if (previous->next == previous) {
        *head_ref = NULL;
        return;
}
while (previous->next != (*head_ref)) {
        previous = previous->next;
}
previous->next = firstNode->next;
(*head_ref) = previous->next;
free(firstNode)
return;
}
void delete_at_end(struct node** head_ref)
{
     struct node *current=*head_ref,*prev;
if (*head_ref) == NULL) {
return;
}
if (current->next == current) {
        *head_ref = NULL;
        return;
 }
while (current->next != (*head_ref)) {
        previous = current;
        current = current->next;
}
previous->next = current->next;
    *head = previous->next;
    free(current);
 return;
}
void delete_at_position(struct node** head_ref,int position)
{
     if (*head_ref == NULL)
    return;
if (position == 1) {
    *head_ref = *head_ref->next;
    free(*head_ref);
    return;
}
struct node *current = *head_ref;
Struct node *prev=NULL;
```

```
for (int i = 1; current != NULL && i < position - 1; i++){
      previous = current;
      current = current->next;
}
previous->next = current->next;
free(current);
     return;
}

int search(struct node** head_ref,int key){
    struct node *current=*head_ref;
    int index=0;
    while (current != NULL) {
    if (current->data == key) {
      return index;
    }
    current = current->next;
    index++;
  }
  return -1;
}
void traverse(struct node* head_ref)
{
     while(head_ref!=NULL){
         printf("%d ",head_ref->data);
         head_ref=head_ref->next;

     }
}
Static void reverse_iter(struct node** head_ref)
{
     if (*head_ref == NULL)
        return;
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != (*head_ref)){
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    (*head_ref)->next = prev;
    *head_ref = prev;
     return;
}
void reverse_rec(struct node* head_ref)
{
     if(head_ref==NULL){
          return;
     }
     reverse_rec(head_ref->next);
     printf("%d ",head_ref->data);
}
```

APPLICATIONS OF LINKED LISTS:

1. Implementation of stacks and queues
2. Implementation of graphs: Adjacency list representation of graphs is the most popular which uses a linked list to store adjacent vertices.
3. Dynamic memory allocation: We use a linked list of free blocks.
4. Maintaining a directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of the linked list
7. Representing sparse matrices

ADDITIONAL PROGRAMS:

Program to remove duplicate elements from a list

```
#include <stdio.h>
struct node{
   int data;
   struct node *next;
};
struct node *head, *tail = NULL;
void addNode(int data) {
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->next = NULL;
   if(head == NULL) {
      head = newNode;
      tail = newNode;
   }
   else {
      tail->next = newNode;
      tail = newNode;
   }
}
void removeDuplicate() {
   struct node *current = head, *index = NULL, *temp = NULL;
   if(head == NULL) {
      return;
   }
   else {
      while(current != NULL){
         //Node temp will point to previous node to index.
         temp = current;
         //Index will point to node next to current
         index = current->next;
         while(index != NULL) {
            //If current node's data is equal to index node's data
            if(current->data == index->data) {
               //Here, index node is pointing to the node which is duplicate of current node
```

```
                    //Skips the duplicate node by pointing to next node
                    temp->next = index->next;
                }
                else {
                    //Temp will point to previous node of index.
                    temp = index;
                }
                index = index->next;
            }
            current = current->next;
        }
    }
}
void display() {
    struct node *current = head;
    if(head == NULL) {
        printf("List is empty \n");
        return;
    }
    while(current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main()
{
   int n,i,ele;
   printf("Enter number of elements in array:\n");
   scanf("%d",&n);
   for(i=0;i<n;i++){
       printf("Enter %d element: \n",i+1);
       scanf("%d",&ele);
       addNode(ele);
   }
    printf("Originals list: \n");
    display();
    removeDuplicate();
    printf("List after removing duplicates: \n");
    display();
    return 0;
}
```

Program to represent polynomials and perform addition.

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

struct Node {
    int coef;
    int exp;
    struct Node* next;
};
```

```
typedef struct Node Node;

void insert(Node** poly, int coef, int exp) {
    Node* temp = (Node*) malloc(sizeof(Node));
    temp->coef = coef;
    temp->exp = exp;
    temp->next = NULL;

    if (*poly == NULL) {
        *poly = temp;
        return;
    }

    Node* current = *poly;

    while (current->next != NULL) {
        current = current->next;
    }

    current->next = temp;
}

void print(Node* poly) {
    if (poly == NULL) {
        printf("0\n");
        return;
    }

    Node* current = poly;

    while (current != NULL) {
        printf("%dx^%d", current->coef, current->exp);
        if (current->next != NULL) {
            printf(" + ");
        }
        current = current->next;
    }

    printf("\n");
}

Node* add(Node* poly1, Node* poly2) {
    Node* result = NULL;

    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp == poly2->exp) {
            insert(&result, poly1->coef + poly2->coef, poly1->exp);
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->exp > poly2->exp) {
            insert(&result, poly1->coef, poly1->exp);
            poly1 = poly1->next;
        } else {
            insert(&result, poly2->coef, poly2->exp);
            poly2 = poly2->next;
        }
    }

    while (poly1 != NULL) {
        insert(&result, poly1->coef, poly1->exp);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insert(&result, poly2->coef, poly2->exp);
        poly2 = poly2->next;
    }

    return result;
}

int main() {
    Node* poly1 = NULL;
    insert(&poly1, 5, 4);
    insert(&poly1, 3, 2);
    insert(&poly1, 1, 0);

    Node* poly2 = NULL;
    insert(&poly2, 4, 4);
    insert(&poly2, 2, 2);
    insert(&poly2, 1, 1);

    printf("First polynomial: ");
    print(poly1);

    printf("Second polynomial: ");
    print(poly2);

    Node* result = add(poly1, poly2);
    printf("Result: ");
    print(result);

    return 0;
}
```

Program to sort a linked list

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

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

struct node *newNode(int data) {
  struct node *new_node = (struct node *)malloc(sizeof(struct node));
  new_node->data = data;
  new_node->next = NULL;
  return new_node;
}
```

```
void insertNode(struct node **head, int data) {
  struct node *new_node = newNode(data);
  if (*head == NULL) {
    *head = new_node;
  } else {
    new_node->next = *head;
    *head = new_node;
  }
}

void printList(struct node *head) {
  struct node *temp = head;
  while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
  }
  printf("\n");
}

void sortList(struct node **head) {
  struct node *current = *head;
  struct node *index = NULL;

  while (current != NULL) {
    index = current->next;
    while (index != NULL) {
      if (current->data > index->data) {
        int temp = current->data;
        current->data = index->data;
        index->data = temp;
      }
      index = index->next;
    }
    current = current->next;
  }
}

int main() {
  struct node *head = NULL;
  int n,i,ele;
  printf("Enter number of elements: \n");
  scanf("%d",&n);
  for(i=0;i<n;i++){
      printf("Enter %d element to insert:\n",i+1);
      scanf("%d",&ele);
      insertNode(&head,ele);
  }


  printf("Unsorted Linked List: ");
  printList(head);

  sortList(&head);
  printf("Sorted Linked List: ");
  printList(head);

  return 0;
}
```

Program to represent sparse matrix

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int column;
    int element;
    struct Node *next;
};
void CreateSparseMatrix(struct Node **X){
    int row, col, ele;
    struct Node *p;
    scanf("%d %d %d", &row, &col, &ele);
    if(X[row] == NULL){
        X[row] = (struct Node *) malloc(sizeof(struct Node));
        p = X[row];
        p->column = col;
        p->element = ele;
        p->next = NULL;
    }
    else{
        p = X[row];
        while(p->next != NULL){
            p=p->next;
        }
        struct Node *temp = (struct Node *) malloc(sizeof(struct Node));
        temp->column = col;
        temp->element = ele;
        temp->next = NULL;
        p->next = temp;
    }
}
void DisplaySparseMatrix(struct Node **X, int m, int n){
    for(int i = 0; i < m; i++){
        struct Node *p = X[i];
        for(int j = 0; j < n; j++){
            if(p == NULL){
                printf("0 ");
            }
            else if(j == p->column){
                printf("%d ", p->element);
                if(p->next != NULL)
                    p = p->next;
            }
            else{
                printf("0 ");
            }
        }
        printf("\n");
    }
}
int main(){
    int m, n, x;
    printf("Enter matrix dimensions:");
    scanf("%d %d", &m, &n);
    printf("Enter total number of non-zero elements: \n");
    scanf("%d", &x);

    struct Node** A = (struct Node **) malloc(sizeof(struct Node) * m);

    printf("Enter all non-zero Elements:\n");
    for(int i = 1; i <= x; i++){
        printf("Enter row number, column number and non-zero element:\n");
        CreateSparseMatrix(A);
    }

    printf("\nSparse Matrix: \n");
    DisplaySparseMatrix(A, m, n);
}
```