UNIT II ARRAYS & LINKED LIST

Arrays - Operations on Arrays – Insertion and Deletion- Applications on Arrays-Multidimensional Arrays- Sparse Matrix- Linked List Implementation – Insertion-Linked List-Deletion and Search- Applications of Linked List-Polynomial Arithmetic-Cursor Based Implementation – Methodology-Cursor Based Implementation-Circular Linked List-Circular Linked List – Implementation-Applications of Circular List -Joseph Problem-Doubly Linked List-Doubly Linked List Insertion-Doubly Linked List Insertion variations-Doubly Linked List Deletion -Doubly Linked List Deletion

LINEAR (OR ORDERED) LISTS ADT

List ADT instances are of the form [l_0, l_1, l_2, …, l_{n-1}], where l_i denotes a list of elements with n >= 0 is finite list where size is n.

LINEAR LIST OPERATIONS

Some of the linear list operations are as follows:

1. printList
   This function is used to print the element in a list.
2. makeEmpty
   This is used to create an empty list.
3. Find
   This function is used to find an element in a given list.
   Example:
   list: [34,12, 52, 16, 12]
   find(52) $\rightarrow$ 3
4. Insert
   This is used to insert a new element into the list.
   For example, insert(x,3) will insert an element 'x' at the 3rd index position as follows, [34, 12, 52, x, 16, 12]
5. Remove
   This function is used to remove an element from the given list.
   For example, remove(52) will remove the element 52 from the list as follows, [34, 12, x, 16, 12].
6. findKth
   This is used to find/search an element at the kth index in a given list.
   - remove(52) $\rightarrow$ 34, 12, x, 16, 12

- : retrieve the element at a certain position

IMPLEMENTATION OF LINEAR LISTS

A linear list ADT is implemented using two standard techniques namely array-based techniques and linked list-based techniques.

ARRAY IMPLEMENTATION OF LIST

Array is a collection of specific number of data stored in a consecutive memory location.

- Insertion and Deletion operation are expensive as it requires more data movement
- Find and Printlist operations takes constant time.
- Even if the array is dynamically allocated, an estimate of the maximum size of the

list is required which considerably wastes the memory space.

LINKED LIST IMPLEMENTATION

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.

Insertion and deletion operations are easily performed using linked list.

TYPES OF LINKED LIST

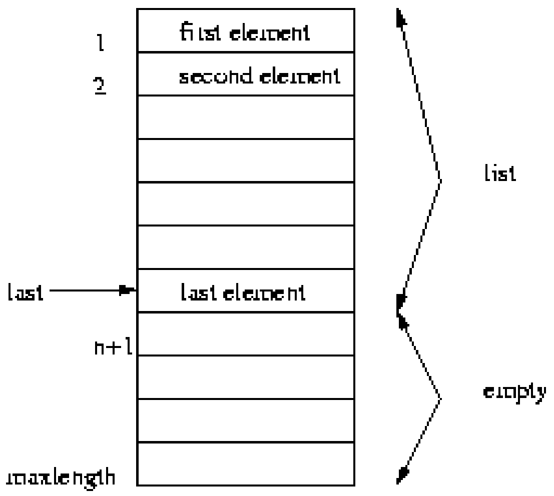
1. Singly Linked List

2. Doubly Linked List

3. Circular Linked List.

ARRAY IMPLEMENTATION

In an array implementation, elements are stored in contiguous array positions as shown in the below figure.

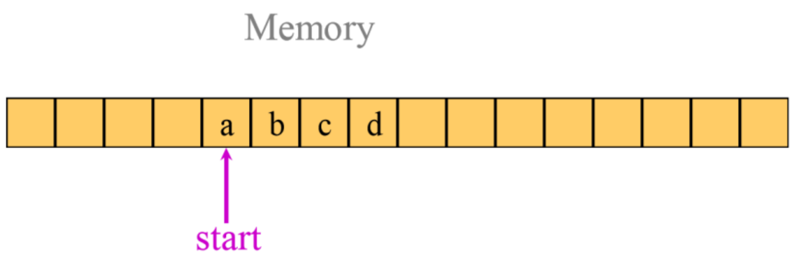


Arrays could be represented in the following forms:

1. 1-D representation
2. 2-D representation
3. Multi-dimensional representation

1D ARRAY REPRESENTATION

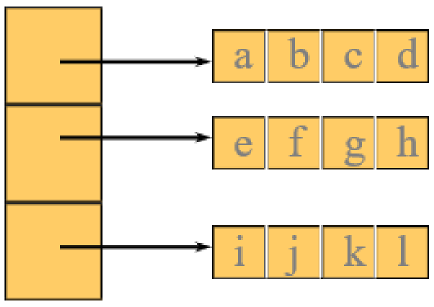
In 1D array representation, the elements will be mapped into a single dimensional contiguous memory location. Let x be an array with elements [a, b, c, d]. The below figure shows how the array elements will be stored in memory.



A 2D array 'a' of size 3 X 4 will be declared as follows: int a[3][4]. The tabular representation of the 2D array 'a' is shown as follows:

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

The method of storing elements in a 2D array is called as array-of-arrays representation. This will require 4 1D arrays of size 3,4,4,4 to store the array 'a'. The below figure shows how elements of 'a' of size 3 X 4 will be stored in a memory.



APPLICATIONS ON ARRAYS

An Arrays in data Structure is a container which can hold a fixed number of items and these items should be of the same type.

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of one-dimensional arrays whose elements are records.
- Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues and stacks.
- One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.
- Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple "if" statements. They are known

in this context as control tables and are used in conjunction with a purpose-built interpreter whose control flow is altered according to values contained in the array. The array may contain subroutine pointers (or relative subroutine numbers that can be acted upon by SWITCH statements) that direct the path of the execution.

IMPLEMENTATION OF ARRAYS – INSERTION AND DELETION

//To Insert An Element In An Array

```
#include<stdio.h>
int insertElement(int arr[], int elements, int keyToBeInserted, int size)
 {
    if (elements >= size)
     return elements;
     arr[elements] = keyToBeInserted;
     return (elements + 1);
 }
int main()
{
   int array[20] = { 23,35,45,76,34,92 };
   int size = sizeof(array) / sizeof(array[0]);
   int elements = 6;
   int i, keyToBeInserted = 65;
   printf("\n Before Insertion: ");
   for (i = 0; i < elements; i++)
   {
      printf("%d  ", array[i]);
    }
    elements = insertElement(array, elements, keyToBeInserted, size);
    printf("\n After Insertion: ");
    for (i = 0; i < elements; i++)
    {
       printf("%d  ",array[i]);
     }
   return 0;
 }
```

Output:

```
Before Insertion: 23 35 45 76 34 92
After Insertion: 23 35 45 76 34 92 65
```

Explanation:

Main function call the insert function to insert an element in an array by passing the parameters arr[] = array, elements = number of elements present in the array and keyToBeInserted = element to be inserted in the array with the size of the array. It checks if the maximum space of the array is already full and return the elements. If not then the element is inserted at the last index and the new array size is returned.

//To Delete An Element In An Array

```
#include<stdio.h>
int deleteElement(int array[], int size, int keyToBeDeleted)
{
   int pos = findElement(array, size, keyToBeDeleted);
   int i;
   if (pos == - 1)
  {
    printf("Element not found");
    return size;
  }
  for (i = pos; i < size - 1; i++)
 {
   array[i] = array[i + 1];
   return size - 1;
 }
}
int findElement(int array[], int size, int keyToBeDeleted)
{
  int i;
  for (i = 0; i < size; i++)
```

```
   if (array[i] == keyToBeDeleted)
     return i;
   else
   return - 1;
}
int main()
{
   int array[] = { 31, 27, 3, 54, 67, 32 };
   int size = sizeof(array) / sizeof(array[0]);
   int i, keyToBeDeleted = 67;
   printf("n Before Deletion: ");
   for (i = 0; i < size; i++)
     printf("%d  ", array[i]);
  deleteElement(array, size, keyToBeDeleted);
  printf("n After Deletion: ");
 for (i = 0; i < size; i++)
   printf("%d  ",array[i]);
 return 0;
}
```

Output:

```
Before Deletion: 31 27 3 54 67 32
After Insertion: 31 27 3 54 32
```

Explanation:

Main function call the delete function to delete an element by passing the parameters array[] is the array from which element needs to be deleted ,size of the array and keyToBeDeleted is the element to be deleted from the array. If element is not found then it prints Element not found. Else it deletes the element & moves rest of the element by one position.

//To Search an Element in an Array

```
#include<stdio.h>
int findElement(int array[], int size, int keyToBeSearched)
{
  int i;
 for (i = 0; i < size; i++)
  if (array[i] == keyToBeSearched)
   return i;
 else
   return - 1;
}
int main()
{
  int array[] = { 41, 29, 32, 54, 19, 3};
  int size = sizeof(array) / sizeof(array[0]);
  int keyToBeSearched = 67;
  int pos = findElement(array, size, keyToBeSearched);
  if(pos==-1)
  {
     printf("n Element %d not found", keyToBeSearched);
  }
  else
 {
   printf("n Position of %d: %d", keyToBeSearched ,pos+1);
 }
   return 0;
}
```

Output:

```
Position of 19:5
```

EXPLANATION

Main function call sub function to search the element in the given array by passing the parameters the array from which element needs to be deleted, size of the array and keyToFind is the element to be search from the array. Running a for loop for Finding & returning the position of the element.

MULTI-DIMENSIONAL SPARSE MATRIX

A matrix is a two-dimensional data object made of m rows and n columns. Generally, a matrix is represented to have m X n values. When the value of m is equal to n, then it is called as a **square matrix**.

There may be a situation in which a matrix may contains a greater number of ‘0’ values (elements) than NON-ZERO values. Such matrix is known as **sparse matrix**. But when a sparse matrix is represented with 2-dimensional array with all ZERO and non-ZERO values, lot of space is wasted to represent that matrix.

For example, consider a matrix of size 200 X 200 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero values. That means, totally we allocate 200 X 200 X 2 = 80000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 80000 times. This results in an increased time and space complexity. An example sparse matrix is given below:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 4 & 0 \end{vmatrix}$$

SPARSE MATRIX REPRESENTATIONS

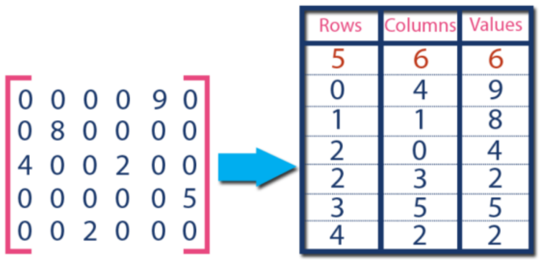
In order to make it simple, a sparse matrix can be represented using two representations.

1. Triplet Representation (Array Representation)
2. Linked Representation

TRIPLET REPRESENTATION (ARRAY REPRESENTATION)

In this representation, only non-zero values are considered along with their row and column index values. The array index [0,0] stores the total number of rows, [0,1] index stores the total number of columns and [0,1] index has the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right-side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

LINKED LIST REPRESENTATION

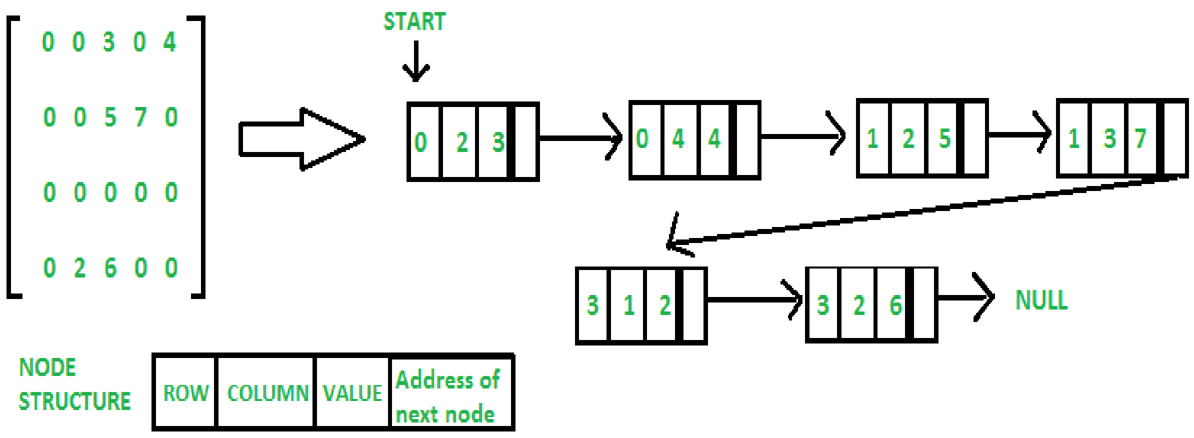
A sparse matrix could be represented using a linked list data structure also. In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row, column)
- **Next node:** Address of the next node

For example, consider a 4 X 5 matrix,

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

The above sparse matrix could be represented as follows:



The first field of the node holds the row index, second field will have the column index, third field will have the value of the element and the address of the next node will be stored in the next field.

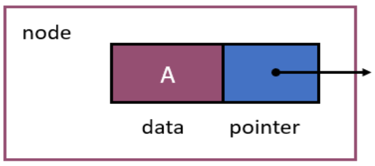
A node in a sparse matrix could be declared as follows:

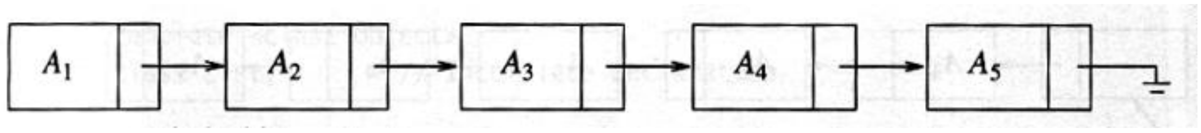
```
struct Node
{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};
```

LINKED LIST

Linked list consists of series of connected nodes. Each node contains the element (a piece of data**)** and a pointer to its successor node. The pointer of the last node points to NULL. A linked list can grow or shrink in size as the program runs. A linked list representation requires no estimate of the maximum size of the list. Space wastage is reduced in linked list representation.

The below figure shows a node structure with data and a pointer to the next node and a list of elements (A_1, A_2, $A_{3,}$ $A_{4,}$ A_5) stored using linked list representation. Insertion and deletion operations are easily performed using linked list, when compared to using arrays.



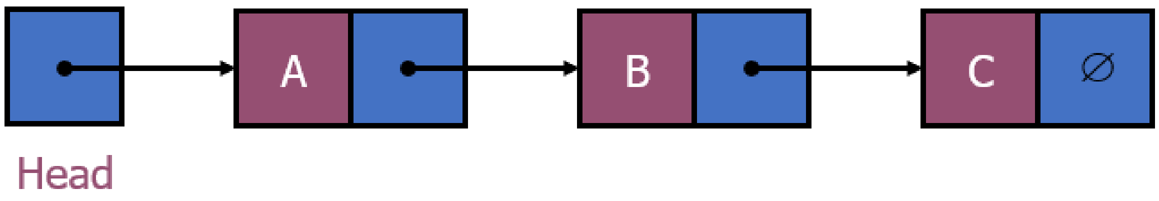


There are different types of linked list available.

TYPES OF LINKED LIST

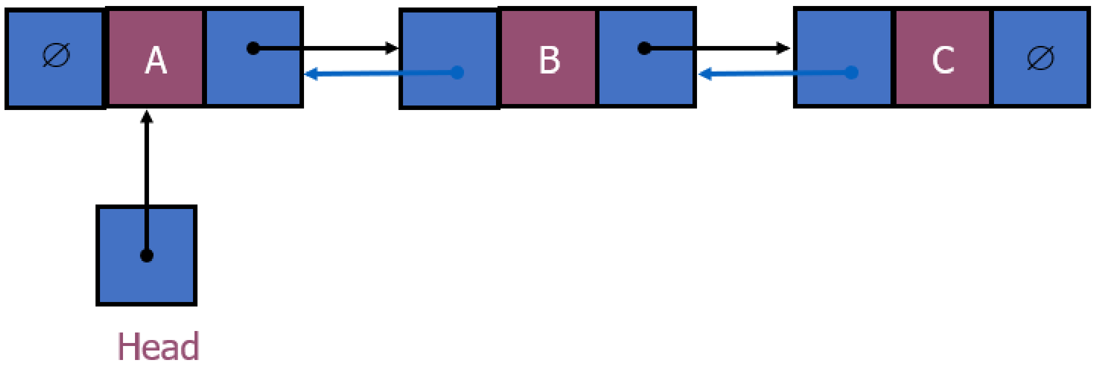
1. Singly Linked List

Each node points in a singly linked list has a successor. There will be NULL value in the next pointer field of the last node in the list



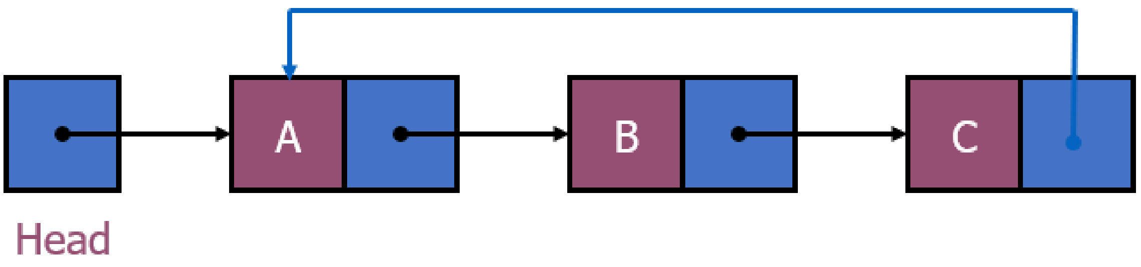
2. Doubly Linked List

In a doubly linked list, each node points to not only successor but the predecessor also. There are two NULL: at the first and last nodes in the list. Advantage of having a doubly linked list is that in given a list, it is easy to visit its predecessor. It will be convenient to traverse lists backwards



3. Circular Linked List.

In a circular linked list, The last node points to the first node of the list.



SINGLY LINKED LIST IMPLEMENTATION

A node in a linked list is usually of structure data type. The declaration of the node structure using pointers is as follows:

```
struct Node
{
   int data;
   struct Node *next;
}*head = NULL;
```

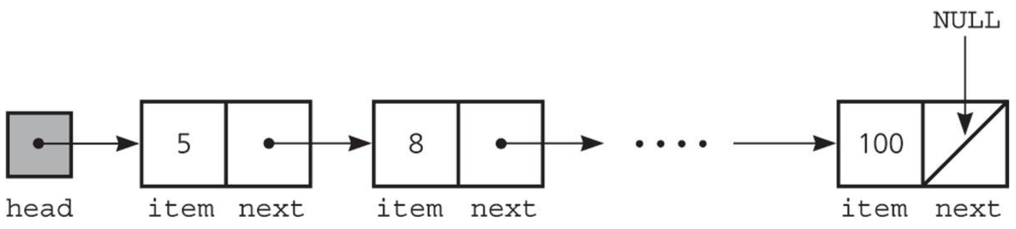
A node is dynamically allocated as follows:

node *p;

p = malloc(sizeof(Node));

By using malloc function, memory will be allocated for the node with size of type struct. The head pointer points to the first node in a linked list. If head is NULL, the linked list is empty and generally represented as **head=NULL.**

A sample linked list is shown below:



Linked lists provide flexibility in allowing the items to be rearranged efficiently.

– Insert an element.

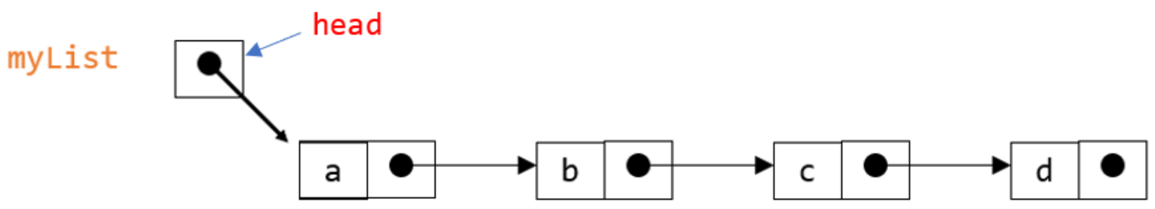
– Delete an element.

INSERTING A NEW NODE

In a linked list, inserting a new node has the following possible cases.

1. Insert into an empty list
2. Insert in front
3. Insert at back
4. Insert in middle

A sample linked list is shown below:

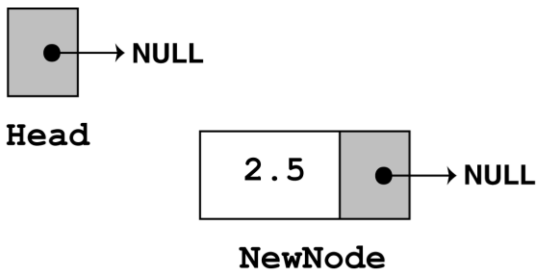


- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)
- The entry point into a linked list is called the **head** of the list.
- It should be noted that head is not a separate node, but the reference to the first node.
- If the list is empty then the head is a null reference.

INSERTING AT BEGINNING OF THE LIST

// Creating a new node

```
newNode = new ListNode;
newNode->value = num;
```



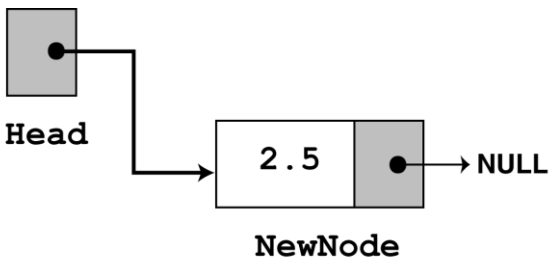
// If it’s the first node set it to point header

head=newNode;
head->next = NULL;

//If already elements are there in the list

newNode->next = head;

head=newNode;

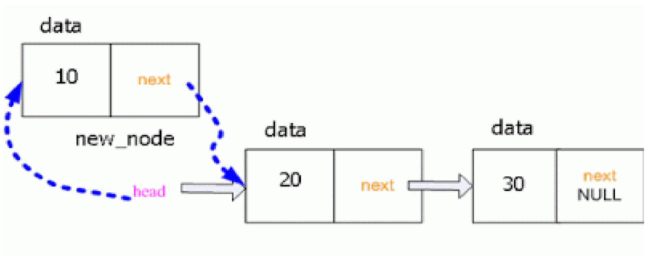


```
void insertAtBeginning(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(head == NULL)
   {
      newNode->next = NULL;
      head = newNode;
   }
   else
   {
      newNode->next = head;
      head = newNode;
   }
   printf("\nOne node inserted!!!\n");
}
```

Routine to Insert a new node at the beginning

Initially, allocate memory for node using malloc function and insert the value in the data field. Now if the list is empty, make the head point to the new node created. Else, copy the address of the existing linked list from the head and store it in the new node’s next pointer. Then store the address of the new node in the head. An example figure is shown below.

Example:



Initially, the linked list has elements 20 and 30. Node with data 20 is the first node, whose address is stored in the head. Now to insert a new node with data 10, copy the address of node with data 20 from head and store in new nodes next pointer. Then make the head point to the new node.

INSERTING AT END OF THE LIST

Steps involved in inserting a new node at the end in the list are as follows:

Step 1: Create a **newNode** with given value and **newNode → next** as **NULL**.

Step 2: Check whether list is **Empty** (**head** == **NULL**).

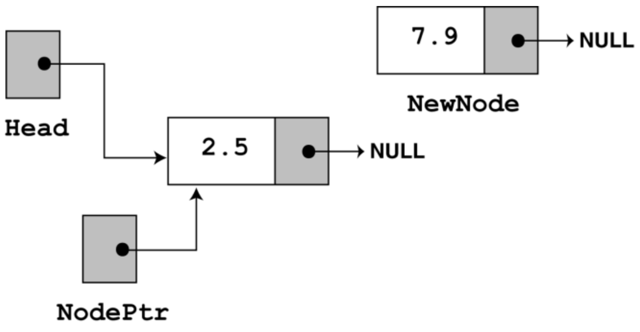
Step 3: If it is **Empty** then, set **head** = **newNode**.

Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5: Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6: Set **temp → next** = **newNode**.

1. Use a Node Pointer to trace to the current position



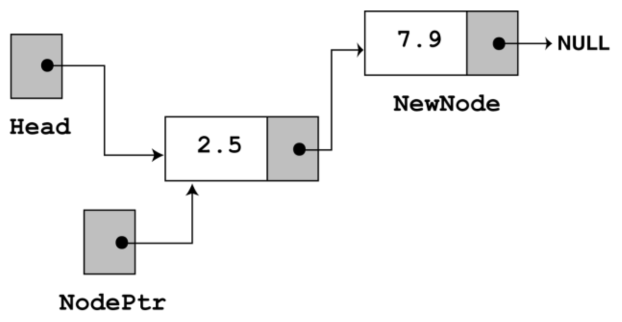
2. If the Node pointer does not points to NULL then move the Node pointer to the next node until it points to NULL

nodePtr=nodePtr->next;

3. When it reaches NULL append the newNode at the last

nodePtr ->next=newNode;

newNode->next=NULL;



```
void insertAtEnd(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if(head == NULL)
    head = newNode;
   else
   {
      struct Node *temp = head;
      while(temp->next != NULL)
    temp = temp->next;
      temp->next = newNode;
   }
   printf("\nOne node inserted!!!\n");
}
```

Routine To Insert An Element At The End

INSERTING A NEW NODE IN THE MIDDLE

Steps involved in inserting a new node in the middle of the list are as follows:

Step 1: Create a **newNode** with given value**.**

Step 2: Check whether list is **Empty (head == NULL)**

Step 3: If it is Empty then, set **newNode** → **next = NULL** and **head = newNode**.

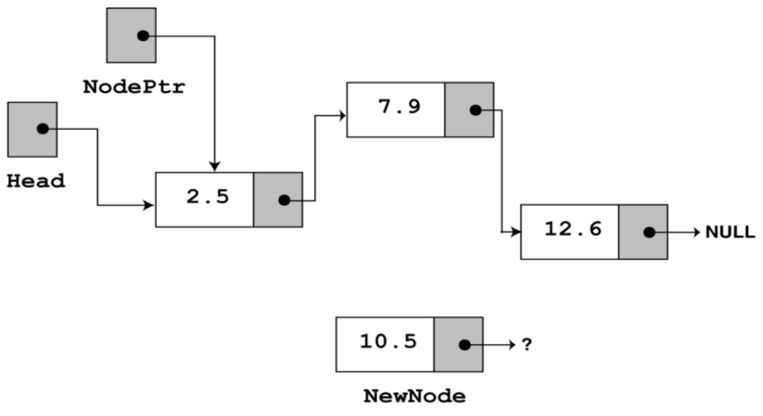
Step 4: If it is **Not Empty** then, **define a node pointer temp** and initialize with head.

Step 5: Keep **moving the temp to its next node** until it reaches to the node after which we want to insert the newNode (until **temp** → **data is equal to location**, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether **temp is reached to last node or not**. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp to next node.**

Step 7: Finally, Set **'newNode** → **next = temp** → **next'** and **'temp** → **next = newNode'**

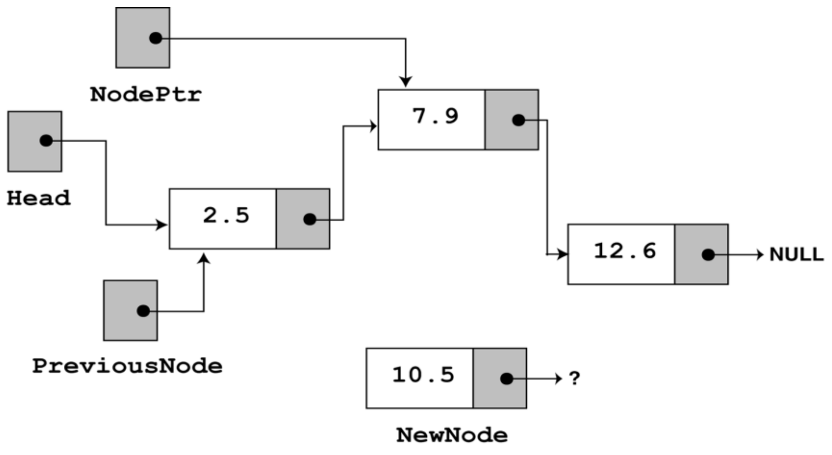
1. Use a Node Pointer to trace to the current position and to store the value of the next address.



2. Use a Previous Node Pointer to trace to the current position

previousNode=nodePtr;

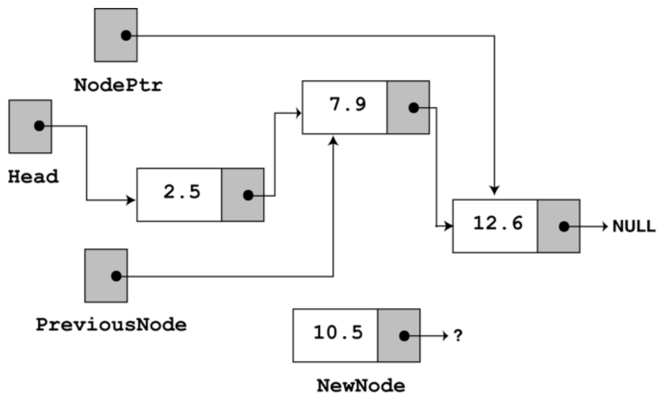
nodePtr=nodePtr->next;

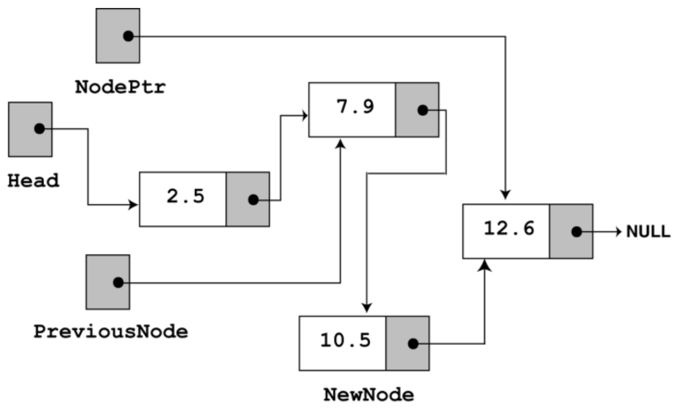


3. The New Node Pointer find the next node to insert the node

previousNode->next=newNode;

newnode->next=nodePtr;





```
void insertBetween(int value, int loc)
{
   struct Node *newNode,*prev_ptr,*cur_ptr;
   newNode = (struct Node*)malloc(sizeof(struct
Node));
   newNode->data = value;
    cur_ptr=head;
   if(head == NULL)
   {
      newNode->next = NULL;
      head = newNode;
   }
   else
   {
      for(i=1;i<loc;i++)
    {
   prev_ptr=cur_ptr;
   cur_ptr = cur_ptr->next;
    }
    prev_ptr->next = newNode;
    newNode->next = cur_ptr;
    }
printf("\nOne node inserted!!!\n");
}
```

Routine to Insert a New Node in Between

DELETING A NODE

In a linked list, deleting a node has the following possible cases.

1. Delete at the front
2. Delete at the back
3. Delete in the middle

DELETING A NODE AT THE FRONT (Deleting from Beginning of the list)

Steps involved in deleting a node from the beginning of the list are as follows:

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

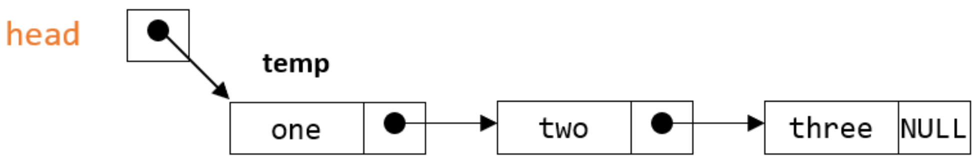
Step 3: If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4: Check whether list is having only one node (**temp** → **next** == **NULL**)

Step 5: If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6: If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

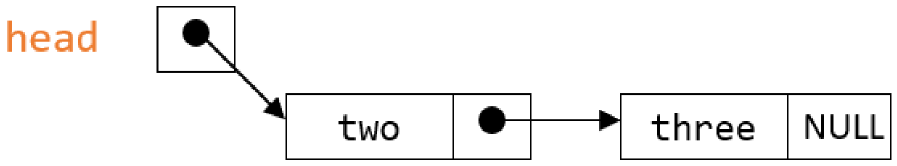
1. To delete the first element, change the link in the header



2. Assign head to a temp. **temp=head;**
3. If the temp is not NULL, check whether the data to be deleted is head.
4. If true then delete the head node and make the next node as head.

head=temp->next;

free(temp);



```
void removeBeginning()
{
    if(head == NULL)
     printf("\n\nList is Empty!!!");
    else
    {
        struct Node *temp = head;
        if(head->next == NULL)
        {
       head = NULL;
       free(temp);
        }
        else
        {
      head = temp->next;
      free(temp);
      printf("\nOne node deleted!!!\n\n");
        }
    }
}
```

Routine to Delete a Node From the Beginning

DELETING FROM END OF THE LIST

Steps involved in deleting a node from the end of the list are as follows:

Step 1: Check whether list is Empty **(head == NULL)**

Step 2: If it is Empty then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

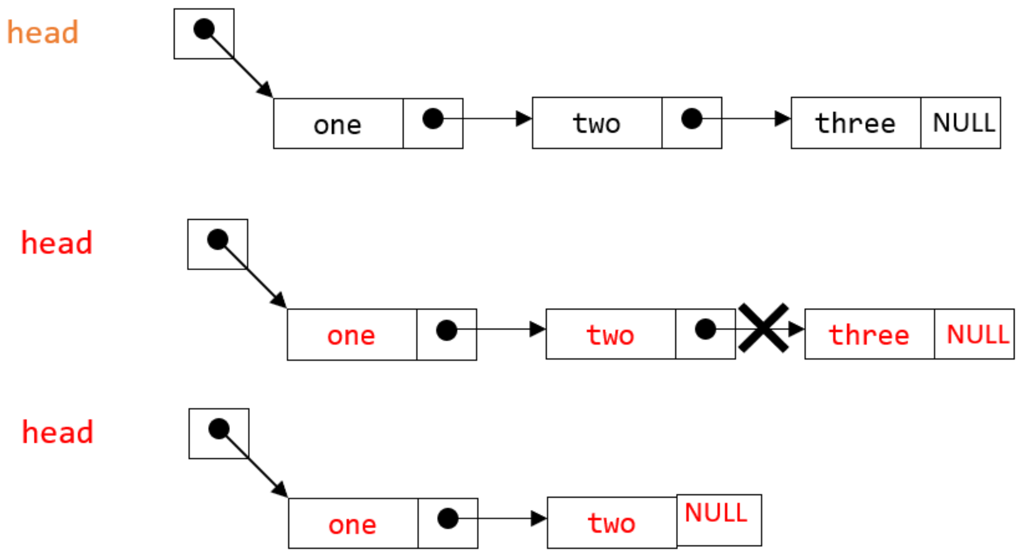
Step 3: If it is **Not Empty** then, define two Node pointers **'temp1' and 'temp2'** and initialize **'temp1' → head.**

Step 4: Check whether list has only one Node **(temp1 → next == NULL)**

Step 5: If it is **TRUE.** Then, set **head = NULL** and **delete temp1.** And terminate the function. (Setting Empty list condition)

Step 6: If it is **FALSE.** Then, set **'temp2 = temp1 '** and **move temp1 to its next node.** Repeat the same until it reaches to the last node in the list. **(until temp1 → next == NULL)**

Step 7: Finally, Set **temp2 → next = NULL** and **delete temp1.**



```
void removeEnd()
{
   if(head == NULL)
   {
      printf("\nList is Empty!!!\n");
   }
   else
   {
      struct Node *temp1 = head,*temp2;
      if(head->next == NULL)
    head = NULL;
      else
      {
     while(temp1->next != NULL)
     {
        temp2 = temp1;
        temp1 = temp1->next;
     }
     temp2->next = NULL;
      }
      free(temp1);
      printf("\nOne node deleted!!!\n\n");}}
```

Routine to delete a node from the end

DELETING A SPECIFIC NODE FROM THE LIST

Steps involved in deleting a specific node from the list is are as follows:

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4: Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5: If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8: If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9: If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

Step 10: If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).

Step 11: If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).

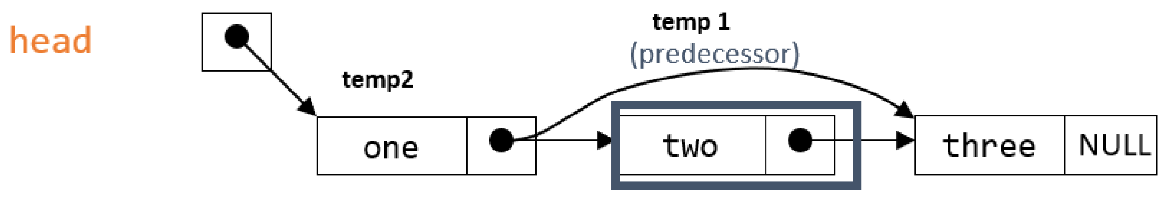
Step 12: If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

1. Use a Node Pointer to trace to the current position and also to store the value of the next address
2. Use a Previous Node Pointer to trace to the current position

```
temp2=temp1;
temp1=temp1->next;
```

3. If the element to be deleted is found then

```
temp2->next=temp1->next;
free( temp1);
```



```
void removeSpecific(int delValue)
{
   struct Node *temp1 = head, *temp2;
   while(temp1->data != delValue)
   {
     if(temp1 -> next == NULL){
   printf("\nGiven node not found in the list!!!");

     }
     temp2 = temp1;
     temp1 = temp1 -> next;
   }
   temp2 -> next = temp1 -> next;
   free(temp1);
   printf("\nOne node deleted!!!\n\n");
}
```

Routine to delete a specific node

APPLICATIONS OF LINKED LIST

The applications of a linked list are as follows. They are

- Used to implement **stacks and queues.**
- Used to implement the **Adjacency list representation** of graphs.
- Used to perform **dynamic memory allocation.**
- Used to **maintain directory of names.**
- Used to **perform arithmetic operations** on long integers.
- Used to **manipulate polynomials** to store constants.
- Used to **represent sparse matrices.**

POLYNOMIAL ARITHMETIC

The manipulation of symbolic polynomials, has a classic example of list processing. In general, we want to represent the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

Where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geqq 0$.

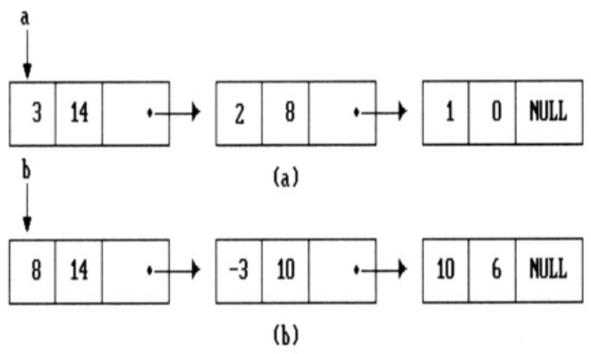
We will represent each term as a node containing **coefficient** and **exponent** fields, as well as a pointer to the next term. A node structure for a polynomial is shown below.

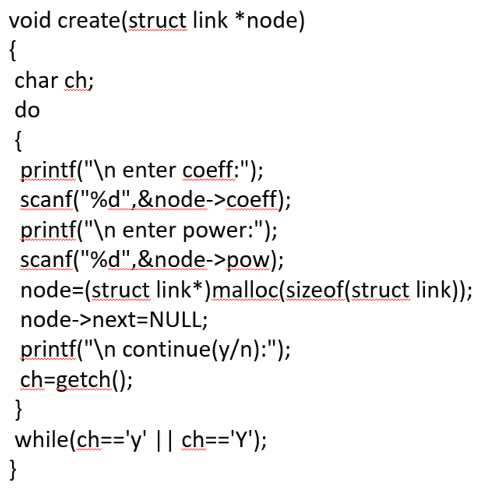
coef	expon	link

Assuming that the coefficients are integers, the node structure will be declared as follows:

```
struct link
        {
         int coeff;
        int pow;
        struct link *next;
        };
```

Consider the polynomials $a = 3x^{14} + 2x^{8} + 1$ and $b = 8x^{14} - 3x^{10} + 10x^{6}$. The linked list representation of the polynomials a and b are shown below.





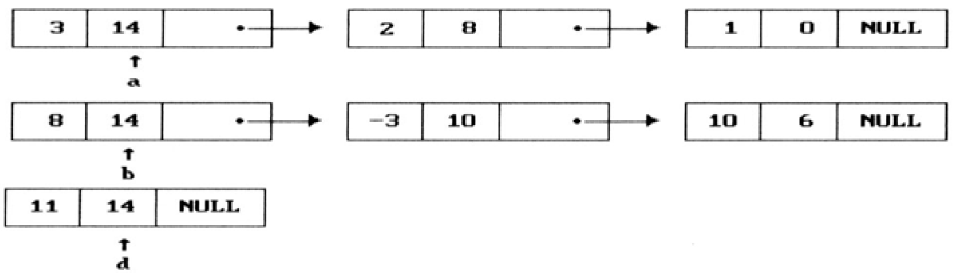
```
void create(struct link *node)
{
 char ch;
 do
 {
  printf("\n enter coeff:");
  scanf("%d",&node->coeff);
  printf("\n enter power:");
  scanf("%d",&node->pow);
  node=(struct link*)malloc(sizeof(struct link));
  node->next=NULL;
  printf("\n continue(y/n):");
  ch=getch();
 }
 while(ch=='y' || ch=='Y');
}
```

Routine to Create a Polynomial Linked List

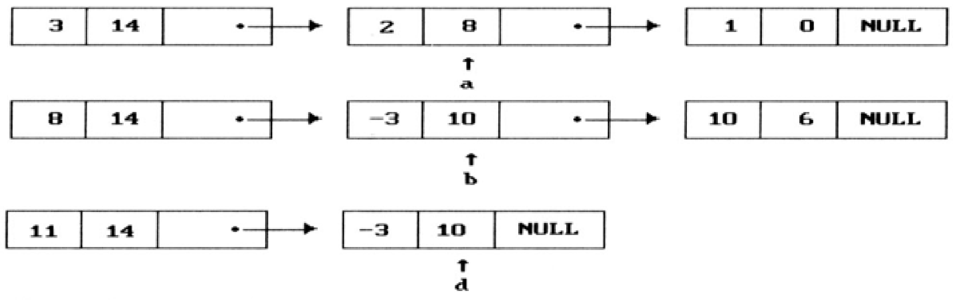
POLYNOMIAL ADDITION

- To add two polynomials, we examine their terms starting at the nodes pointed to by *a* and *b*.
    - If the exponents of the two terms are equal
        - add the two coefficients and create a new term for the result.
    - If the exponent of the current term in *a* is less than *b*
        - create a duplicate term of *b* and attach this term to the result, called *d*
        - advance the pointer to the next term in *b*.
    - We take a similar action on *a* if *a->expon* > *b->expon.*

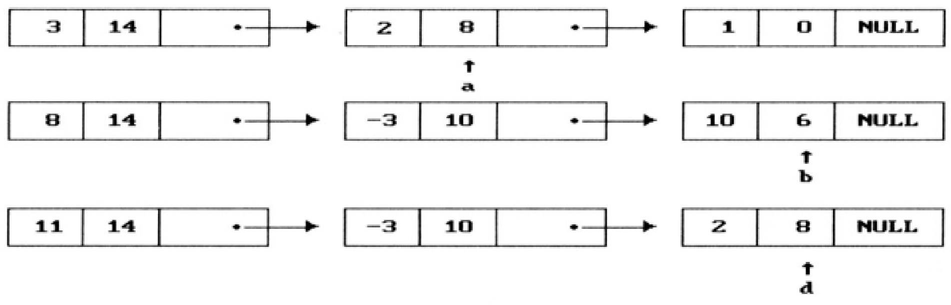
The figure shows generating the first three term of d = a+b below.



(a) a -> expon == b -> expon

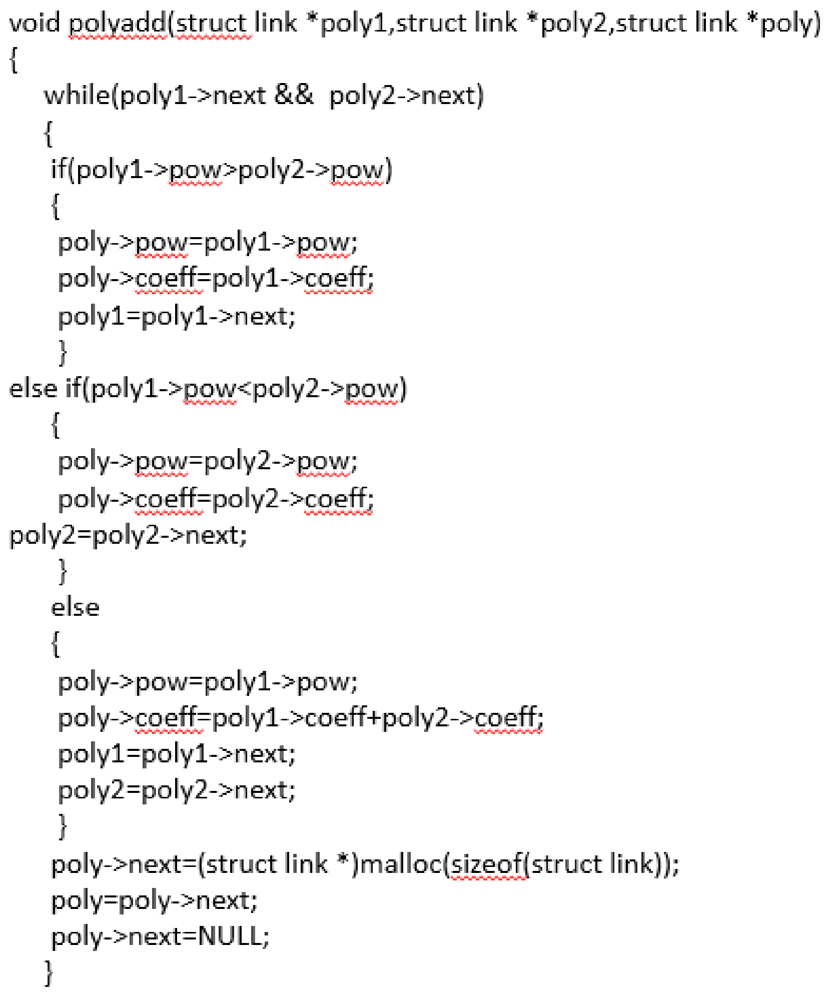


(b) a -> expon < b -> expon

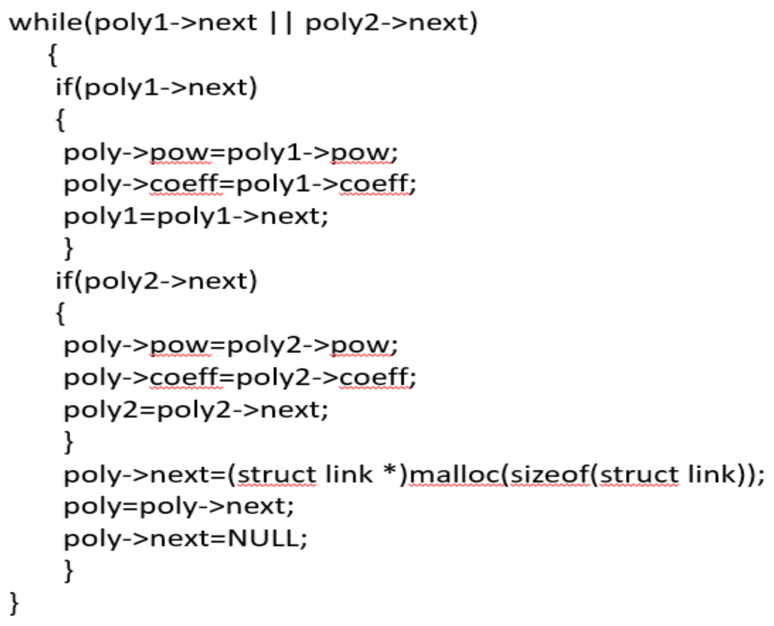


(c) a -> expon > b -> expon

From the above figure it is observed that when the exponents of a and b are equal, coefficients of a and b are added in the new polynomial d. Else if the exponent of a is bigger than exponent of b, link the corresponding node of a to d and vice versa. Routine to create a linked list for a polynomial is give below.



```
void polyadd(struct link *poly1,struct link *poly2,struct link *poly)
{
   while(poly1->next &&  poly2->next)
   {
    if(poly1->pow>poly2->pow)
    {
     poly->pow=poly1->pow;
     poly->coeff=poly1->coeff;
     poly1=poly1->next;
     }
else if(poly1->pow<poly2->pow)
    {
     poly->pow=poly2->pow;
     poly->coeff=poly2->coeff;
poly2=poly2->next;
     }
    else
    {
     poly->pow=poly1->pow;
     poly->coeff=poly1->coeff+poly2->coeff;
     poly1=poly1->next;
     poly2=poly2->next;
     }
    poly->next=(struct link *)malloc(sizeof(struct link));
    poly=poly->next;
    poly->next=NULL;
   }
```



```
  while(poly1->next || poly2->next)
     {
      if(poly1->next)
      {
       poly->pow=poly1->pow;
       poly->coeff=poly1->coeff;
       poly1=poly1->next;
       }
      if(poly2->next)
      {
       poly->pow=poly2->pow;
       poly->coeff=poly2->coeff;
       poly2=poly2->next;
       }
       poly->next=(struct link *)malloc(sizeof(struct link));
       poly=poly->next;
       poly->next=NULL;
       }
  }
```

Routine to Add Two Polynomials

CURSOR BASED - IMPLEMENTATION METHODLOGY

If linked lists are required and pointers are not available, then an alternate implementation must be used. The alternate method we will describe is known as a cursor implementation.

The two important items present in a pointer implementation of linked lists are

1. The data is stored in a collection of structures. Each structure contains the data and a pointer to the next structure.

2. A new structure can be obtained from the system's global memory by a call to malloc and released by a call to free.

Our cursor implementation must be able to simulate this. The logical way to satisfy condition 1 is to have a global array of structures. For any cell in the array, its array index can be used in place of an address. The type declarations for a cursor implementation of linked lists is given below.

```
typedef unsigned int node_ptr;

struct node
{
  element_type element;
  node_ptr next;
};

typedef node_ptr LIST;
typedef node_ptr position;
struct node CURSOR_SPACE[ SPACE_SIZE ];
```

Declarations For Cursor Implementation Of Linked Lists

We must now simulate condition 2 by allowing the equivalent of malloc and free for cells in the CURSOR_SPACE array. To do this, we will keep a list (the freelist) of cells that are not in any list. The list will use cell 0 as a header. The initial configuration is shown in the figure below.

Slot	Element	Next
0	?	1
1	?	2
2	?	3
3	?	4
4	?	5
5	?	6
6	?	7
7	?	8
8	?	9
9	?	10
10	?	0

An Initialized Cursor Space

A value of 0 for next is the equivalent of a pointer. The initialization of CURSOR_SPACE is a straightforward loop, which we leave as an exercise. To perform an malloc, the first element (after the header) is removed from the freelist.

To perform a free, we place the cell at the front of the freelist. The below routine shows the cursor implementation of malloc and free.

```
position cursor_alloc( void )
{
  position p;
  p = CURSOR_SPACE[O].next;
  CURSOR_SPACE[0].next = CURSOR_SPACE[p].next;
  return p;
}

void cursor_free( position p)
{
  CURSOR_SPACE[p].next = CURSOR_SPACE[O].next;
  CURSOR_SPACE[O].next = p;
}
```

Routines: cursor-alloc and cursor-free

Notice that if there is no space available, our routine does the correct thing by setting p = 0. This indicates that there are no more cells left, and also makes the second line of cursor_new a nonoperation (no-op). Given this, the cursor implementation of linked lists is straightforward. For consistency, we will implement our lists with a header node.

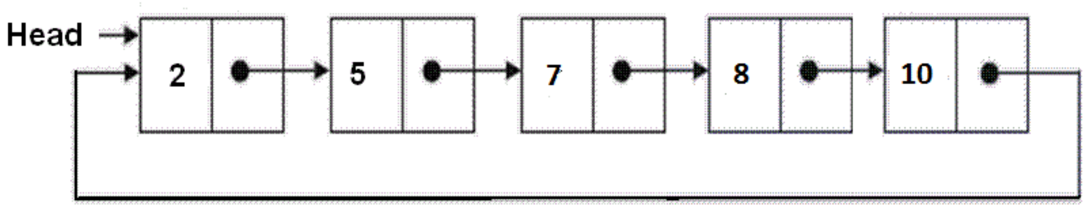
An example of cursor implementation of linked list is shown below. In the figure, if the value of L is 5 and the value of M is 3, then L represents the list a, b, e, and M represents the list c, d, f.

Slot	Element	Next
0	-	6
1	b	9
2	f	0
3	header	7
4	-	0
5	header	10
6	-	4
7	c	8
8	d	2
9	e	0
10	a	1

Example of a cursor implementation of linked lists

CIRCULAR LINKED LIST

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



ADVANTAGES OF CIRCULAR LINKED LISTS

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

OPERATIONS ON CIRCULARLY LINKED LIST

In a circular linked list, following operations are performed,

1. Insertion
2. Deletion
3. Display

First perform the following steps before implementing actual operations.

Step 1 - Include all the header files which are used in the program.

Step 2 - Declare all the user defined functions.

Step 3 - Define a Node structure with two members data and next

Step 4 - Define a Node pointer 'head' and set it to NULL.

Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Circular linked list – Creation

```
head = new;
new->next=head;
```

Insertion in a Circular Linked List

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting at Beginning of the list
2. Inserting at End of the list
3. Inserting at Specific location in the list

INSERTING AT BEGINNING OF THE LIST

Steps to **insert a new node at beginning** of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .

Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').

Step 6 - Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

INSERTING AT END OF THE LIST

Steps to insert a new node at end of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**).

Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** == **head**).

Step 6 - Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

INSERTING AT SPECIFIC LOCATION IN THE LIST (AFTER A NODE)

Steps to insert a new node at a specific location of the circular linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).

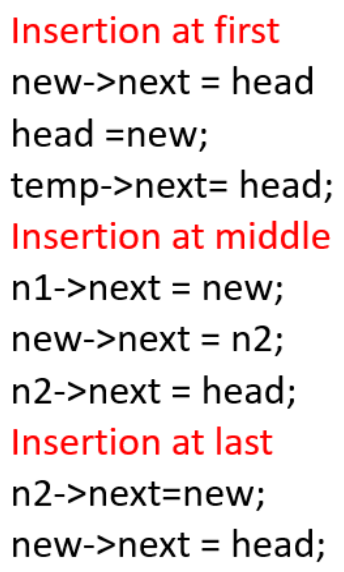
Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8 - If **temp** is last node then set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

Step 8 - If **temp** is not last node then set **newNode** → **next** = **temp** → **next** and **temp** → **next** = **newNode**.

CLL – INSERTION (ROUTINE)



Insertion at first

```
new->next = head
head =new;
temp->next= head;
```

Insertion at middle

```
n1->next = new;
new->next = n2;
n2->next = head;
```

Insertion at last

```
n2->next=new;
new->next = head;
```

DELETION

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

DELETING FROM BEGINNING OF THE LIST

The following steps to delete a node from beginning of the circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

Step 4 - Check whether list is having only one node (**temp1** → **next** == **head**)

Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1** → **next** == **head**)

Step 7 - Then set **head** = **temp2** → **next**, **temp1** → **next** = **head** and delete **temp2**.

DELETING FROM END OF THE LIST

The following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1** → **next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1** → **next** == **head**)
- **Step 7 -** Set **temp2** → **next** = **head** and delete **temp1**.

DELETING A SPECIFIC NODE FROM THE LIST

The following steps to delete a specific node from the circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

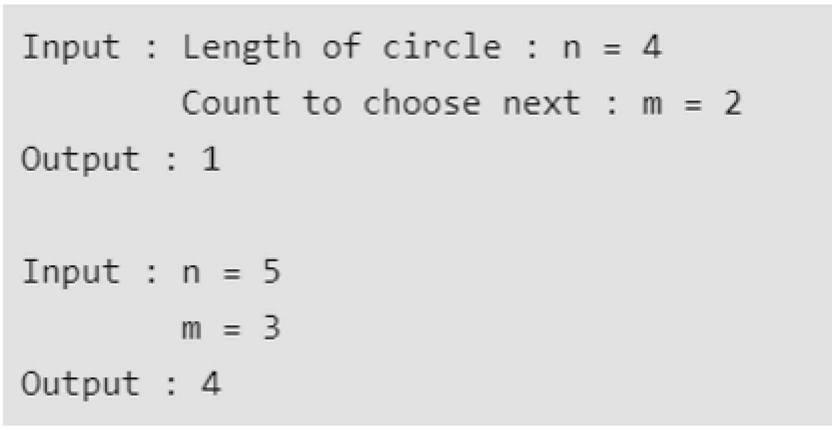
Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1** → **next** == **head**)
Step 7 - If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
Step 9 - If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head** → **next**, **temp2** → **nex**t = **head** and delete **temp1**.
Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1** → **next == head**).
Step 1 1- If **temp1** is last node then set **temp2** → **next** = **head** and delete **temp1** (**free(temp1)**).
Step 12 - If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

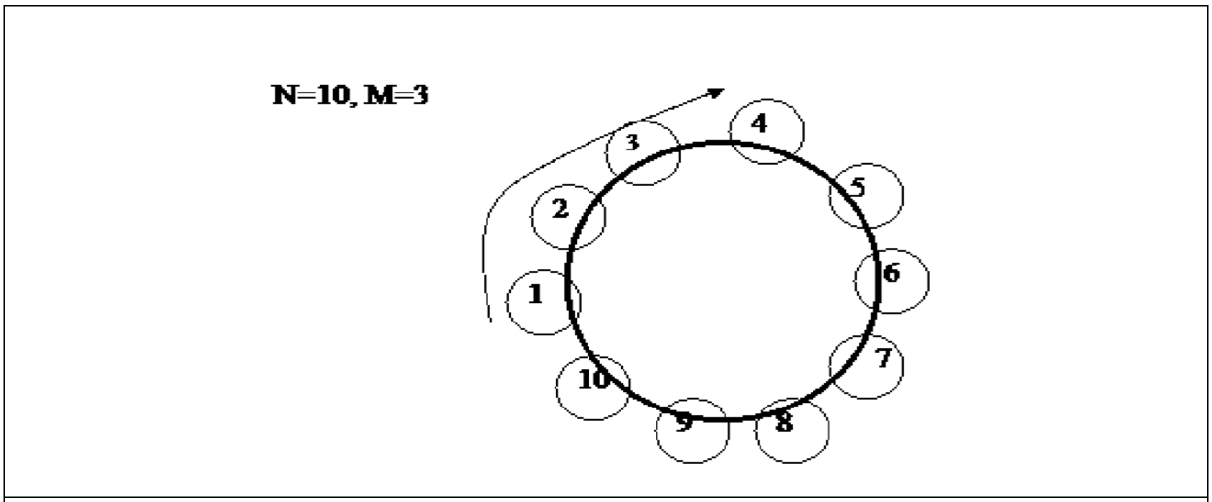
APPLICATIONS OF CIRCULAR LIST -JOSEPH PROBLEM

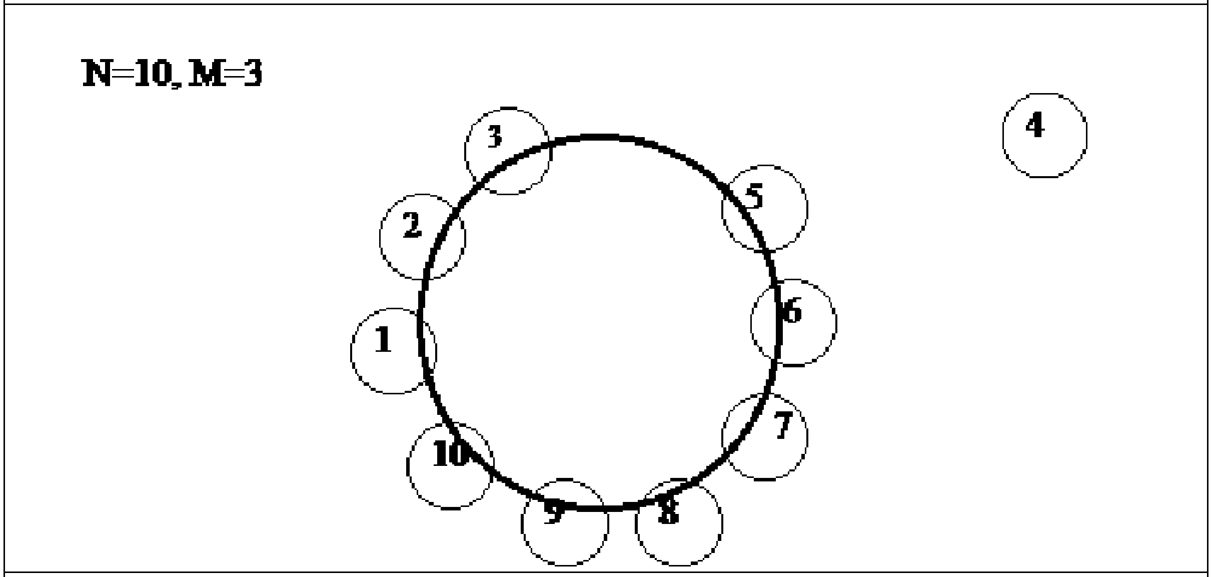
There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number m which indicates that m-1 persons are skipped and m-th person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

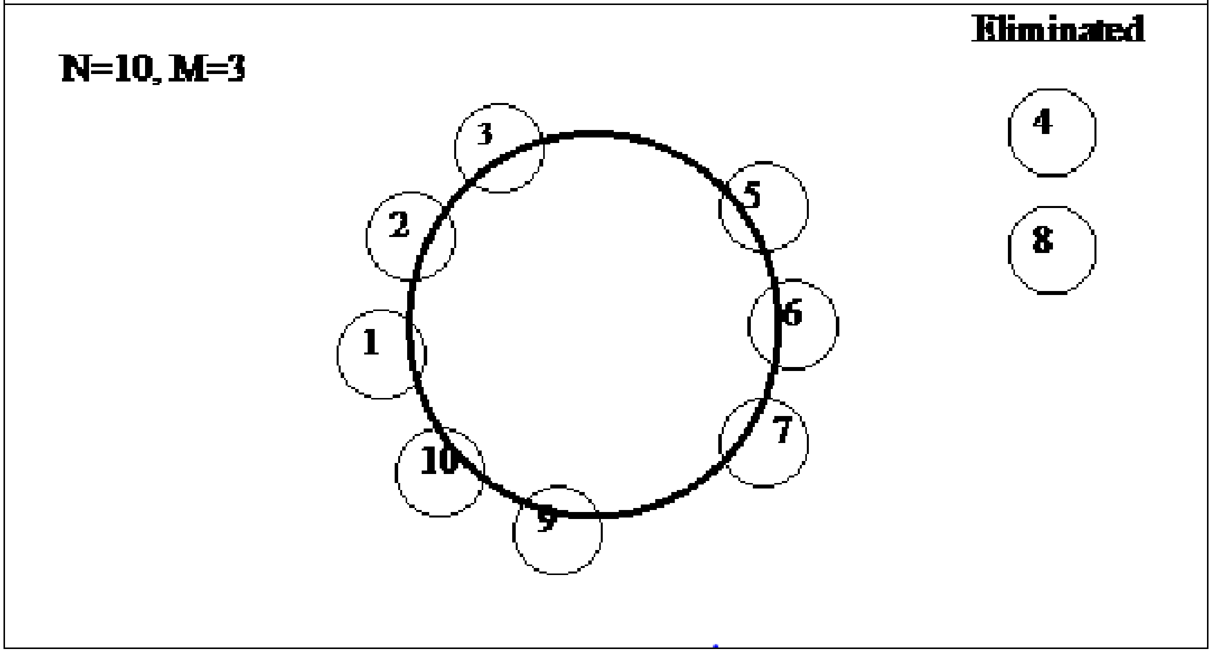
Examples:

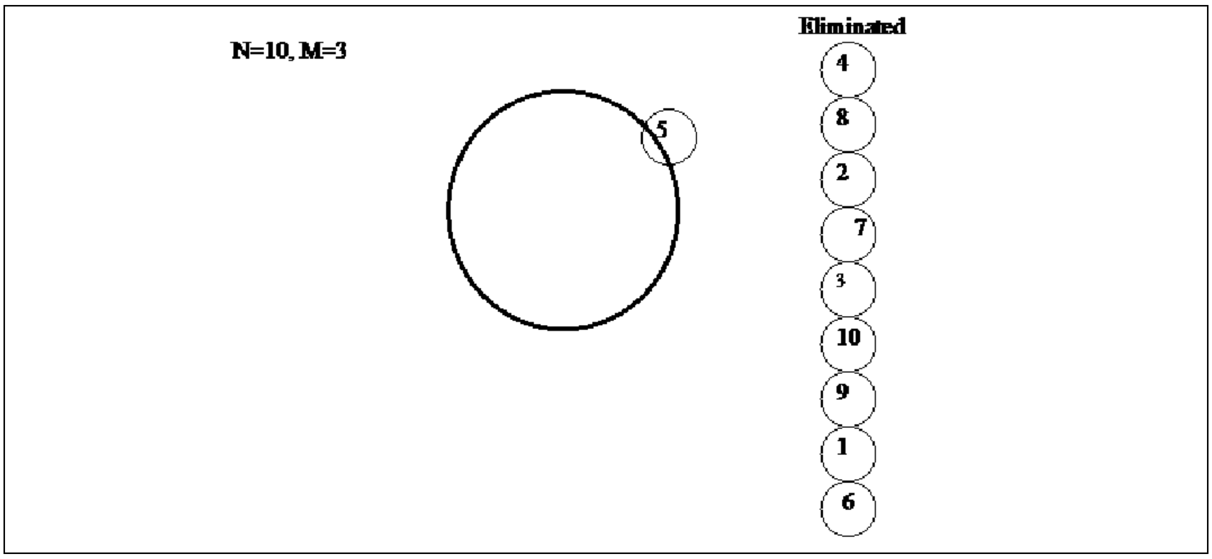


```
Input : Length of circle : n = 4
        Count to choose next : m = 2
Output : 1

Input : n = 5
        m = 3
Output : 4
```


N=10, M=3
1
2
3
4
5
6
7
8
9
10


N=10, M=3
1
2
3
5
6
7
8
9
10
4


Eliminated
N=10, M=3
1
2
3
5
6
7
9
10
4
8

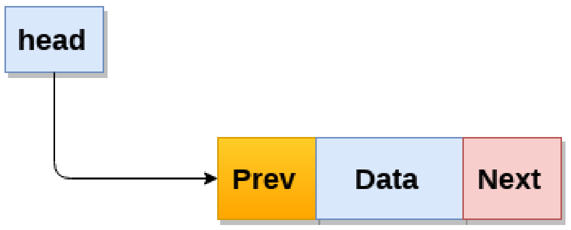


```
int josephus(int m,node *head)
{
        node *f;
        int c=1;
        while(head->next!=head)
        {
                c=1;
                while(c!=m)
                {
                        f=head;
                        head=head->next;
                        c++;
                }
                f->next=head->next;
                printf("%d->",head->data); //sequence in which nodes getting deleted
                head=f->next;
        }
        printf("\n");
        printf("Winner is:%d\n",head->data);
        return;
}
```

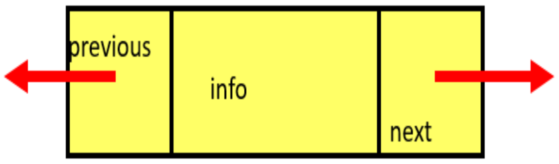
ROUTINE FOR JOSEPHUS PROBLEM

DOUBLY LINKED LIST

- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list and to the last node in the list (or contains null links if the list is empty)
- A simple node structure of a doubly linked list is shown below:



Node



A node structure in doubly linked list is declared as follows:

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

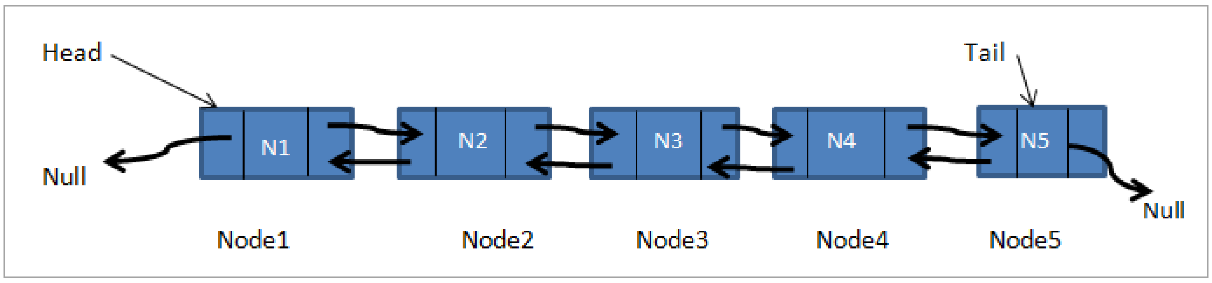
OPERATIONS IN A DOUBLY LINKED LIST ARE AS FOLLOWS

1. Node Creation
2. Insertion at beginning
3. Insertion at end
4. Insertion after specified node
5. Deletion at beginning
6. Deletion at end
7. Deletion of the node having given data
8. Searching
9. Traversing

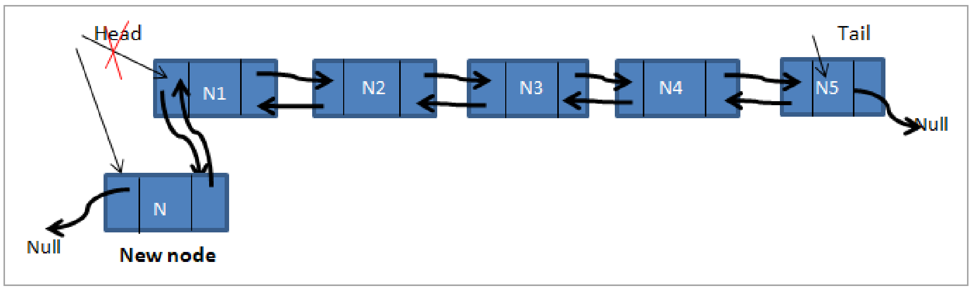
NODE CREATION

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

INSERTION AT BEGINNING



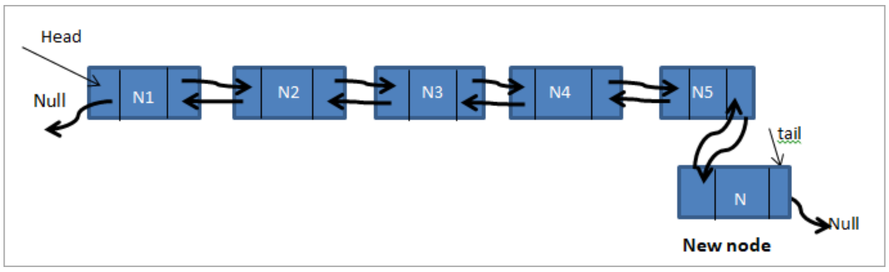
```
void insert_at_begin()
{
  struct node *tempptr;
  int item;
  tempptr = (struct node *)malloc(sizeof(struct node));
   printf("\nEnter Item value");
   scanf("%d",&item);

  if(head==NULL)
  {
      tempptr->right = NULL;
      tempptr->left=NULL;
      tempptr->data=item;
      head=tempptr;
  }
  else
  {
      tempptr->data=item;
      tempptr->left=NULL;
      tempptr->right = head;
      head->left=tempptr;
      head=tempptr;
 } printf("\nInsertion
completed\n");
}

}
```



INSERTION AT END

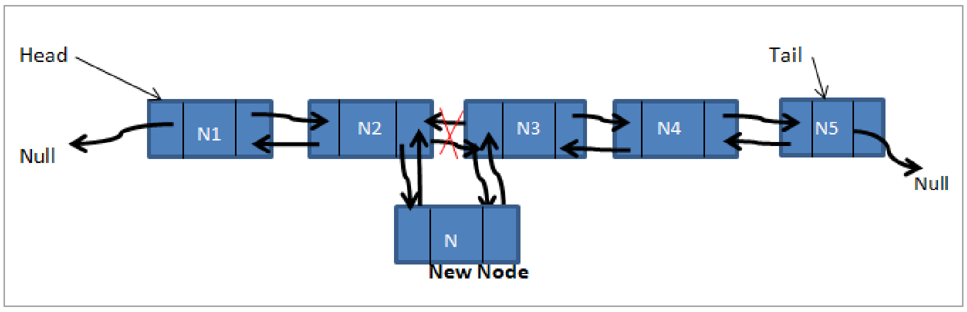
```
void insert_at_last()
{
  struct node *tempptr,*temp;
  int item;
  tempptr = (struct node *) malloc(sizeof(struct node));
      printf("\nEnter value");
      scanf("%d",&item);
```

```
     tempptr->data=item;
    if(head == NULL)
    {
       tempptr->right = NULL;
       tempptr->left = NULL;
       head = tempptr;
    }
    else
    {
      temp = head;
      while(temp->right!=NULL)
      {
          temp = temp->right;
      }
      temp->right = tempptr;
      tempptr ->left=temp;
      tempptr->right = NULL;
      }
         }
  printf("\n insertion completed \n");
 }
```



INSERTION AFTER SPECIFIED NODE

```
void insert_after_spcific_node()
{
  struct node *tempptr,*temp;
  int item,loc,i;
  tempptr = (struct node *)malloc(sizeof(struct node));

    temp=head;
    printf("Enter the location");
    scanf("%d",&loc);
    for(i=0;i<loc;i++)
    {
       temp = temp->right;
       if(temp == NULL)
       {
          printf("\n There are less than %d elements", loc);
          return;
       }
    }
    printf("Enter data");
    scanf("%d",&item);
    tempptr->data = item;
```



```
        tempptr->right = temp->right;
        tempptr -> left = temp;
        temp->right = tempptr;
        temp->right->left=tempptr;
        printf("\n insertion completed \n");

}
```

DELETION AT BEGINNING

```
void delete_at_begin()
{
   struct node *tempptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->right == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
   else
   {
      tempptr = head;
      head = head -> right;
      head -> left = NULL;
      free(tempptr);
      printf("\nnode deleted\n");
   }

}
```

DELETION AT END

```
void delete_at_last()
{
   struct node *tempptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->right == NULL)
   {
      head = NULL;
```

```
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        tempptr = head;
        if(tempptr->right != NULL)
        {
            tempptr = tempptr -> right;
        }
        tempptr -> left -> right = NULL;
        free(tempptr);
        printf("\nnode deleted\n");
    }
}
```

DISPLAY_FULL_LIST LIST

```
void display_full_list()
{
    struct node *tempptr;
    printf("\n printing values...\n");
    tempptr = head;
    while(tempptr != NULL)
    {
        printf("%d\n",tempptr->data);
        tempptr=tempptr->right;
    }
}
```

MAIN FUNCTION:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};
struct node *head;
void insert_at_begin();
void insert_at_last();
void insert_after_spcific_node();
void delete_at_begin();
void delete_at_last();
void deletion_specified();
void display_full_list();
void search();
void main ()
{
int option =0;
   while(option != 9)
   {
      printf("\nChoose option from below menu\n");
            printf("\n1.Insert begining\n 2.Insert last\n 3.Insert any random location \n 4.Delete
the first node \n 5.Delete the last node\n6.Delete the node after the given
data\n7.Search\n8.Show\n9.Exit\n");
      printf("\nEnter your option?\n");
      scanf("\n%d",&option);
      switch(option)
      {
         case 1:
         insert_at_begin();
         break;
         case 2:
               insert_at_last();
         break;
         case 3:
         insert_after_specific_node();
         break;
         case 4:
         delete_at_begin();
         break;
         case 5:
         delete_at_last();
         break;
         case 6:
         deletion_specified();
         break;
         case 7:
         search();
         break;
         case 8:
         display_full_list();
         break;
         case 9:
         exit(0);
         break;
         default:
            printf("Please enter valid option..");
        }
    }
}
```

OUTPUT:

```
Choose option from below menu
1.Insert begining
2.Insert last
3.Insert any random location
4.Delete the first node
5. Delete the last node
6.Delete the node after the given data
7.Search
8.Show
9.Exit
Enter your option?
1
Enter Item value 10
Insertion completed.
Choose option from below menu
1.Insert begining
2.Insert last
3.Insert any random location
4.Delete the first node
5. Delete the last node
6.Delete the node after the given data
7.Search
8.Show
9.Exit
Enter your option?
1
Enter Item value 20
Insertion completed.
Choose option from below menu
1.Insert begining
2.Insert last
3.Insert any random location
4.Delete the first node
5. Delete the last node
6.Delete the node after the given data
7.Search
8.Show
9.Exit
```

```
Enter your option?
8
Printing values…
20
10

Choose option from below menu
1.Insert begining
2.Insert last
3.Insert any random location
4.Delete the first node
5. Delete the last node
6.Delete the node after the given data
7.Search
8.Show
9.Exit
Enter your option?
4
Node deleted
Choose option from below menu
1.Insert begining
2.Insert last
3.Insert any random location
4.Delete the first node
5. Delete the last node
6.Delete the node after the given data
7.Search
8.Show
9.Exit
Enter your option?
8
Printing values…
10
Choose option from below menu
1.Insert begining
2.Insert last
3.Insert any random location
4.Delete the first node
5. Delete the last node
6.Delete the node after the given data
7.Search
8.Show
9.Exit
Enter your option?
```

Explanation:

This program having many functions each performing one operation in doubly linked list. When the program is executed, list of options or operation that we can perform in the linked list displayed. Each option is associated with one function which is called by the switch case based on the user input. When the input 1 is given after executing program, the function insert_at_begin(); is called. It creating the new node with the user entered data 10. Again new node is inserted at the beginning using the same function with value 20. Right now the doubly list having the two node. The first node data is 20 and second is 10. We can display the list by calling display full list function through option 8. Then the first node is deleted from the list by giving option 4 that calling deleting_at_begin function. After this operation, the list would have only one value that is 10. Likewise, all the operation in the doubly linked can be performed using above writing function.