Linear Arrays

- An array is a list of finite , ordered, homogenous data elements stored under a single name.
- Suppose the number n denotes the length or size of the array.
- The elements of the array can be referenced using a subscript or index which consists of consecutive numbers (0 to n-1)
- Elements of the array are stored in successive memory locations.
- UB(Upper Bound) – Largest index
- LB(Lower Bound) – Smallest index
- Length = UB – LB +1
- If we start from 1, i.e If LB=1 then length = UB
- An array is also called subscripted variable because the elements are accessed using subscripts.eg: a[1],a[2], ……..

Declaring arrays

An array is declared as ;

datatype arrayname[size];

eg: int a[10];

Initialization of arrays

- We can initialize array elements at the time of declaration.

Syntax

- Datatype arrayname[size] = {list of values};
- eg: int a[3]={10,4,6};
- int b[5] ={1,2} means int b[5]={1,2,0,0,0};
- int a[] = {1,2,3,4 ,5};
- int a[3]= {}; or int a[3]={0};-> All elements will be initialized to 0.
- We can assign values of individual elements of arrays using assignment operator.
- Eg: a[5]=10;

Representation of Linear Array in Memory

- If we store an integer data, it takes 2 bytes to store a data element.
- char takes one byte, float 4 bytes and double 8 bytes.
- if ‘a’ is an array with 10 integer elements, then it is represented in memory as :
- 1000 1002 1004 1006 1008 1010 1012 1014 1016 1018
- a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

Calculating the address of array element

- The starting address of array is called the base address.
- The address of a particular array element is calculated using the formula
- a[k]= base address + size of data item(index of element – LB)
- Suppose we have float a[10];
- base address is 1000
- a[5]= 1000+ 4(5-0)
- = 1000 + 20

  = 1020

Data Structure Array Operations

Inserting a new element in an array

- A new element can be inserted at any position of an array if there is enough memory space allocated to accommodate the new element.
- Some of the elements should be shifted forward to keep the order of the elements.
- Suppose we want to add an element 93 at the 3rd position in the following array

- 0 1 2 3 4 5 6
- After inserting all elements comes after 32 must be moved to the next location to accommodate the new element and to keep the order of the elements as follows.

0 1 2 3 4 5 6 7

```
#include <stdio.h>
#define MAX_SIZE 100
int main() {
   int array[MAX_SIZE], size, i, position, element;
printf("Enter the size of the array (max %d): ", MAX_SIZE);
   scanf("%d", &size);
printf("Enter %d elements: ", size);
   for (i = 0; i < size; i++) {
      scanf("%d", &array[i]);
```

```
    }
 printf("Enter the position to insert (0-based index): ");
    scanf("%d", &position);
printf("Enter the element to insert: ");
    scanf("%d", &element); // Shift elements to make space for the new element
    for (i = size; i > position; i--) {
        array[i] = array[i - 1];
    }
    // Insert the element at the specified position
    array[position] = element;
        size++; // Increment the size of the array
        printf("Updated array:\n"); // Print the updated array
       for (i = 0; i < size; i++) {
        printf("%d ", array[i]);    }
return 0;
}
```

Output

Enter the size of the array : 7

Enter 7 elements : 10 22 32 23 25 64 44

Enter the position to insert : 3

Enter the elements to insert : 66

Updated array : 10 22 66 32 23 25 64 44

Deleting an element from an array

Suppose we want to delete element 44 at the 3rd position in the following array.

22	32	44	51	65	71	80
0	1	2	3	4	5	6

After deletion, all elements coming after 44 must be moved to the previous location to fill the free space and to keep the order of the elements as follows

22	32	51	65	71	80
0	1	2	3	4	5

Algorithm

- Find the position of the element to be deleted in the array.
- If the element is found, shift all elements after the position of the element by 1 position.
- Decrement array size by 1.
- If the element is not found: Print" element is not found"

```
#include <stdio.h>
#define MAX_SIZE 100
  int main() {
   int array[MAX_SIZE], size, i, position;
   // Input the size of the array
   printf("Enter the size of the array (max %d): ", MAX_SIZE);
   scanf("%d", &size);
  printf("Enter %d elements: ", size); // Input elements of the array
   for (i = 0; i < size; i++) {
      scanf("%d", &array[i]);
   }
printf("Enter the position to delete (0-based index): "); // Input the position to delete
   scanf("%d", &position);
if (position < 0 || position >= size) {
      printf("Invalid position. Please enter a position between 0 and %d.\n", size - 1);
      return 1; // Exit program with an error code
   }
   // Shift elements to remove the specified element
   for (i = position; i < size - 1; i++) {
      array[i] = array[i + 1];   }
   // Decrement the size of the array
   size--;
   // Print the updated array
   printf("Updated array:\n");
   for (i = 0; i < size; i++) {
      printf("%d ", array[i]);  }
   return 0;
```

}

Output

Enter the size of the array (max %d): 5

Enter 5 elements: 23 24 35 65 78

Enter the position to delete (0-based index): 3

Updated array : 23 24 65 78

Multidimensional Arrays in C

Declaring Multidimensional Arrays in C

- The declaration of an n-dimensional array takes the following form,
- datatype arrayName[size1][size2]……[sizeN];
- For example, int X[3][4];
- Here X is a 2-dimensional array with 3 rows and 4 columns . The total number of elements that can be stored in a multidimensional array is the product of the size of all the dimensions. The size of this two dimensional array will be 12.
- Elements in this array can be reffered by A[i][j] where i is row number and j is the coumn.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Initializing Multidimensional Arrays

- A 2- dimensional array can be initialized in any of the following ways:
- int A[3][3]={{1,2,3},{9,8,7},{3,4,5}};
- int A[][3]={{1,2,3},{9,8,7},{3,4,5}};
- intA[3][3]={1,2,3,9,8,7,3,4,5};

A 3-dimensional array can be initialized in a similar way. For example

```
int A[3][2][4]={ {{1,7,8,2},{5,6,1,0}},{{5,3,0,2},{4,6,1,8}},
                                        {{7,3,9,1},{4,7,2,5}},{{6,4,1,8},{5,2,6,7}};
```

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

```
int main() {
  int r, c, a[100][100], b[100][100], sum[100][100], i, j;
  printf("Enter the number of rows (between 1 and 100): ");
  scanf("%d", &r);
  printf("Enter the number of columns (between 1 and 100): ");
  scanf("%d", &c);
  printf("\nEnter elements of 1st matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &a[i][j]);  }
  printf("Enter elements of 2nd matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element b%d%d: ", i + 1, j + 1);
      scanf("%d", &b[i][j]); }
// adding two matrices
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      sum[i][j] = a[i][j] + b[i][j];
    }  // printing the result
  printf("\nSum of two matrices: \n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("%d   ", sum[i][j]);
      if (j == c - 1) {
        printf("\n\n");
      }   }
  return 0;  }
```

```
Enter the number of rows (between 1 and 100): 2
Enter the number of columns (between 1 and 100): 3
Enter elements of 1st matrix: Enter element a11: 2    Enter element a12: 3
```

Enter element a13: 4 Enter element a21: 5

Enter element a22: 2 Enter element a23: 3

Enter elements of 2nd matrix: Enter element b11: -4 Enter element b12: 5

Enter element b13: 3 Enter element b21: 5

Enter element b22: 6 Enter element b23: 3

Sum of two matrices:

-2 8 7

10 8 6

Parallel Array

- A parallel array is a structure of an array.
- It contains multiple arrays of the same size, in which the i-th elements of each array is related to each other.
- All the elements in a parallel array represent a common object or entity.
- In the example below, we store the role number and marks of five persons in two different arrays.
- Roll_no = {1,2,3,4,5};
- Marks ={25,20,32,30,18};

Example: Parallel Array

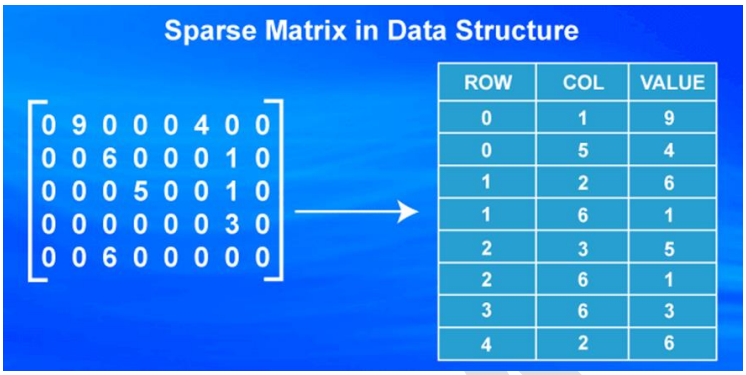
```
#include<stdio.h>
 int main() {
  int max=0,index=0,i,n=5;
  int roll_no[]={1,2,3,4,5};
   int marks[]={25,20,32,30,18};
   for(int i=0;i<n;i++) {
    if(marks[i]>max) {
     max = marks[i];
    index=i;  }}
    printf("Roll no %d has highest marks",roll_no[index]); return 0; }
```

Output

Rll no 3 has highest marks

Sparse Matrix

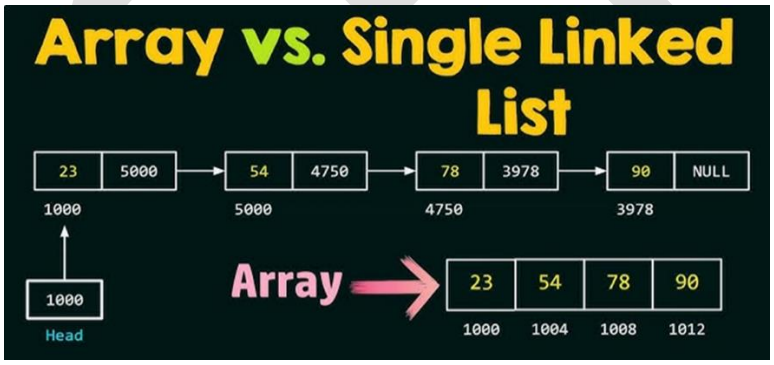
- A matrix is a two dimensional array made of m rows and n columns, and so m*n elements.
- A matrix is called a sparse matrix if most of its elements are 0
- An example of a sparse matrix of order 5*8 is shown below



- The usage of a 2D array to represent a sparse matrix wastes a lot of memory because the zeroes in the matrix are useless in most cases.
- This leads to seek an alternate representation that could reduce the memory space consumed by zero entries.
- A sparse matrix can be easily represented by a list of triplets of the form(row,column,element)
- We store only the index of the row, index of column and values of the non zero.

Linked List

Ways to maintain a list in memory



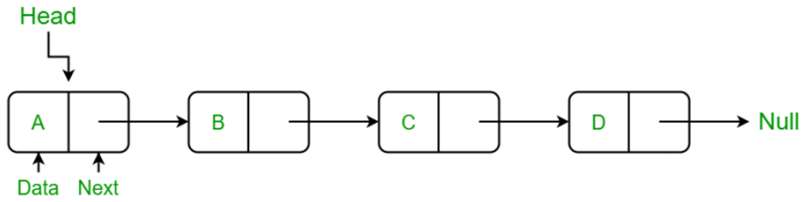
Types of Linked List

- **Single Linked List**:- Navigation is forward only
- **Doubly Linked List**:- Forward and backward navigation is possible.
- **Circular Linked List**:-Last element is linked to the first element.

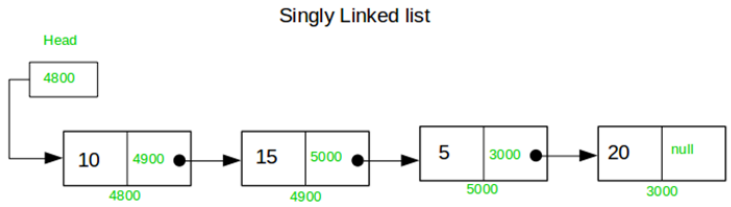
Single Linked List

A single linked list is a list made up of nodes that consists of two parts.

- Data
- Link



Representation of a single linked list



- First node accessed using head

Array Vs Linked list representation

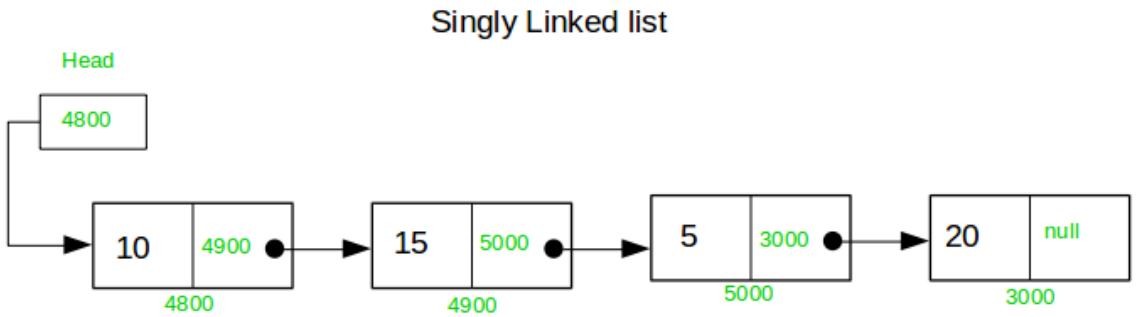
We want to store a list of numbers : 10,15,5,20

Array

10	15	5	20
1000	1004	1008	1012

- In an array, elements are stored in consecutive memory locations.
- Array is the sequential representation of list.
- We want to store a list of numbers : 10,15,5,20

Nodes are scattered here and there in the memory but they are still connected with each other



- Linked list is the linked representation of list

Node representation in C

- struct node {

int data;

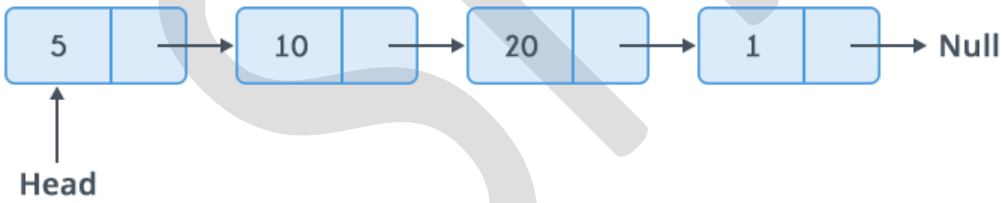
struct node,*link;

};

Node

Array	Linked List
An array is a collection of elements of a similar data type	A linked list is a collection of objects known as a node where node consists of two parts.i.e, data and address
Array elements are stored in a contiguous memory location	Linked list elements can be stored anywhere in the memory or randomly stored
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The linked list works with dynamic memory. Here dynamic memory means that the memory size can be changed at the run time according to our requirements
Array elements are independent of each other	Linked list elements are dependent on each other.
Array takes more time while performing any operation like insertion, deletion etc.	Linked list takes less time while performing any operation like insertion, deletion etc.
Accessing elements in an array is faster as the elements in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time	In the case of a linked list, memory is allocated at run time
Memory utilization is inefficient in the array.	Memory utilization is efficient in the case of a linked list.

How to create a node of the list?

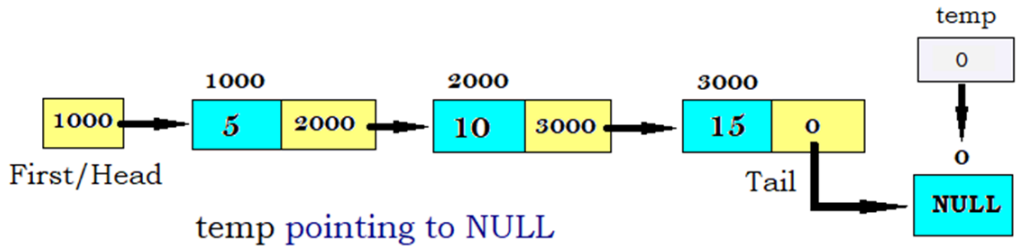



```
struct LinkedList{
   int data;
   struct LinkedList *next;
 };
#include <stdio.h>
#include <stdlib.h>
  struct Node {
   int data;
   struct Node* link;
```

```
};
 int main() {
 struct node *head = malloc(sizeof(struct node));
            head->data = 5;
             head->link = Null;
           return 0;
}
```

Traversing a Single Linked List

- Traversing a single linked list means visiting each node of a single linked list until the end node is reached.



Program to Count the number of Nodes

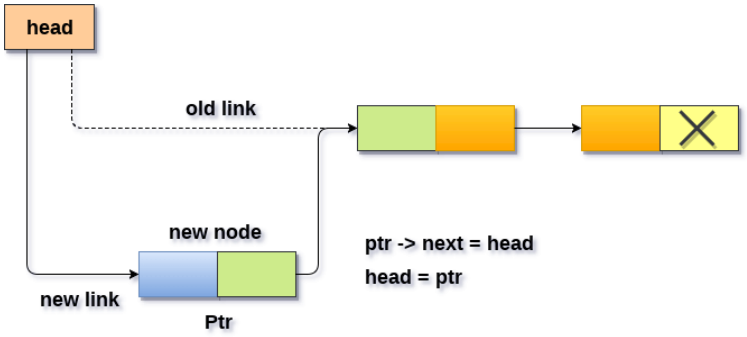
```
#include <stdio.h>
#include <stdlib.h>
struct node {
                              int data;
                 struct node *link;
               };
          int main() {
                 count_of_nodes(head);
                  }
```

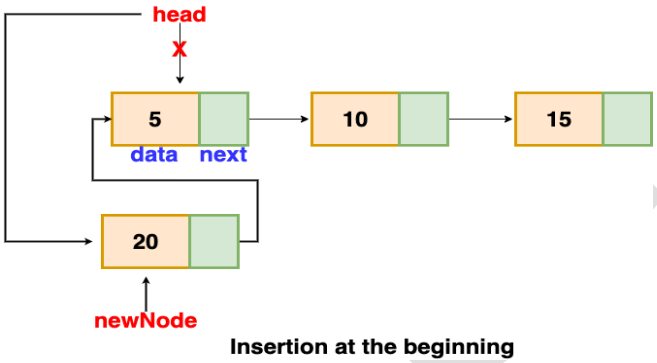
Program to Count the number of Nodes

```
 void count_of_nodes(struct node *head) {
    int count=0;
     if(head==NULL)
      printf("Linked list is empty");
```

```
    struct node *ptr = NULL;
    ptr=head;
    while(prt!= NULL) {
      count++;
       ptr = ptr->link;
       }
 printf("%d", count);
}
```

Inserting Node at the Beginning of the List





Insertion at the beginning

Adding node at the beginning of the list

```
 struct node* add_beg(struct node* head,int d)
  {
      struct node *ptr = malloc(sizeof(struct node));
       ptr->data = d;
       ptr->link = NULL;
       ptr->link = head;
       head = ptr;
```

```
    return head;
}
```

Inserting node at the end of the list

```
#include<stdio.h>
#include<stdlib.h>
struct node{
  int data;
   struct node *link;
   };
    int main() {
       add_at_end(head,67);     //ptr->link=temp;
       }
```

Inserting node at the end of the list

Allocate memory for new node

Store data

Traverse to last node

Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;}
temp->next = newNode;
```

Insert at the Middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
```

```
struct node *temp = head;
for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
     temp = temp->next;}  }
newNode->next = temp->next;
temp->next = newNode;
```

Delete from a Linked List

Delete from beginning

Point head to the second node(head = head->next;)

```
#include<stdio.h>
#include<stdlib.h>
Struct node{  int data;
                struct node *link;  };
          int main() {
                head = del_frist(head);
                ptr=head;
                 while(ptr!=NULL){
                          printf("%d", ptr->data);
                          ptr = ptr->link; }    return 0;
                    }
```

Delete from end

Traverse to second last element

Change its next pointer to null

```
      struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

Delete from end

```
#include<stdio.h>
#include<stdlib.h>
Struct node{  int data;
```

```
        struct node *link;  };
    int main() {
        head = del_last(head);
        ptr=head;
         while(ptr!=NULL){
                printf("%d", ptr->data);
                ptr = ptr->link; }     return 0;  }
```

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
  if(temp->next!=NULL) {
    temp = temp->next;  }
}  temp->next = temp->next->next;
```

Linked list : Searching

- Finding an element is similar to a traversal operation. Instead of displaying data, we have to check whether the data matches with the item to find.
- Initialize Ptr with the address of Head. Now the Ptr points to the first node of the linked list.
- A while loop is executed which will compare data of every node with them.
- If item has been found then control goes to last step.

Algorithm : Search

- Step1: [INITIALIZE] SET PTR=HEAD
- Step2: Repeats Steps3 and 4 while PTR!= NULL
- Step3: If ITEM = PTR -> DATA
- SET POS= PTR
- Go to Step5
- ELSE
- SET PTR=PTR->NEXT
- [END OF IF]
- [END OF LOOP]
- Step 4: SET POS=NULL

- Step 5: EXIT

Algorithm : Traverse

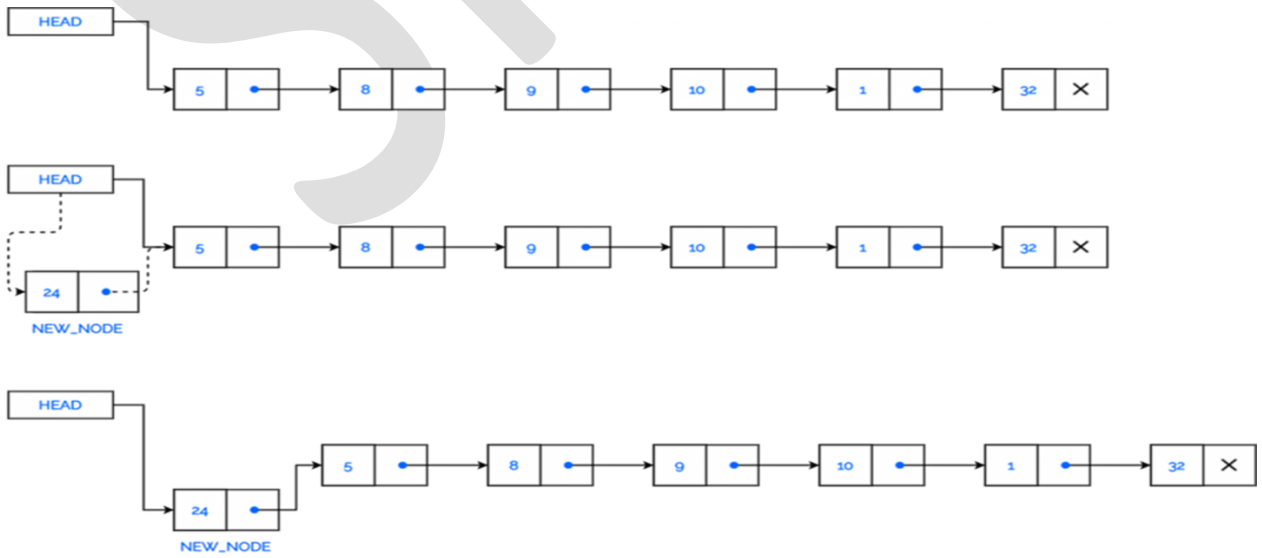
- Step1: [INITIALIZE] SET PTR=HEAD
- Step2: Repeats Steps3 and 4 while PTR!= NULL
- Step3: Apply process to PTR->DATA
- Step4: SET PTR = PTR->NEXT
- [END OF LOOP]
- Step5: EXIT

Inserting Elements to a Linked List

- We will see how a new node can be added to an existing linked list in the following cases.
- The new node is inserted at the beginning.
- The new node is inserted at the end.
- The new node is inserted after a given node.

Insert a Node at the beginning of a Linked list

- Consider the linked list shown in the figure. Suppose we want to create a new node with data 24 and add it as the first node of the list. The linked list will be modified as follows.
- inserting an element in the beginning of a linked list 1



Allocate memory for new node and initialize its DATA part to 24.

Add the new node as the first node of the list by pointing the NEXT part of the new node to HEAD.

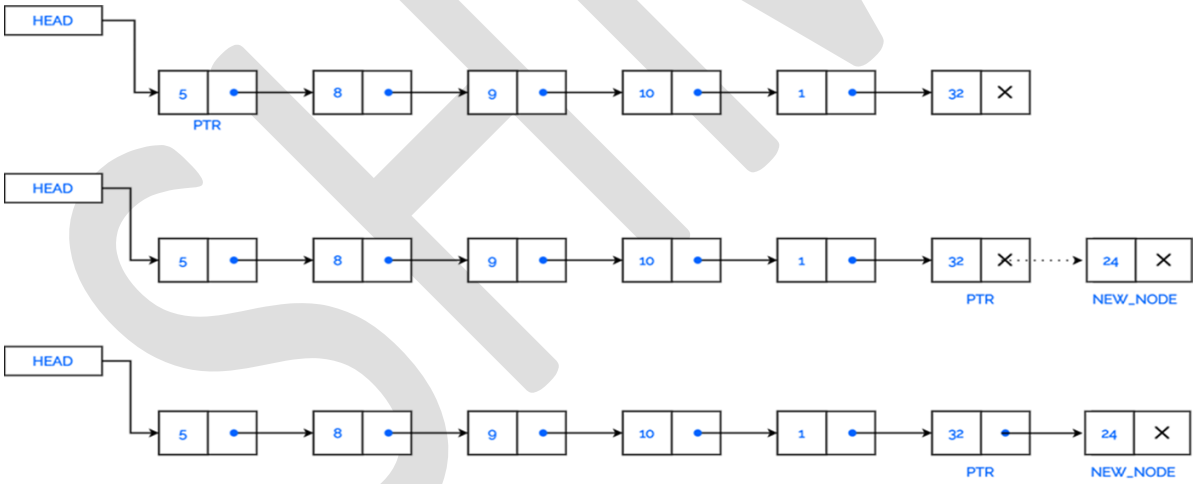
Make HEAD to point to the first node of the list.

Algorithm: InsertAtBeginning

- Step 1: IF AVAIL = NULL
- Write OVERFLOW
- Go to Step 7
- [END OF IF]
- Step 2: SET NEW_NODE = AVAIL
- Step 3: SET AVAIL = AVAIL -> NEXT
- Step 4: SET NEW_NODE -> DATA = VAL
- Step 5: SET NEW_NODE -> NEXT = HEAD
- Step 6: SET HEAD = NEW_NODE
- Step 7: EXIT

Insert a Node at the end of a Linked list

- Take a look at the linked list in the figure. Suppose we want to add a new node with data 24 as the last node of the list



- Allocate memory for new node and initialize its DATA part to 24.
- Traverse to last node.
- Point the NEXT part of the last node to the newly created node.
- Make the value of next part of last node to NULL.

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 10

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET PTR = HEAD

Step 7: Repeat Step 8 while PTR -> NEXT != NULL

Step 8: SET PTR = PTR -> NEXT

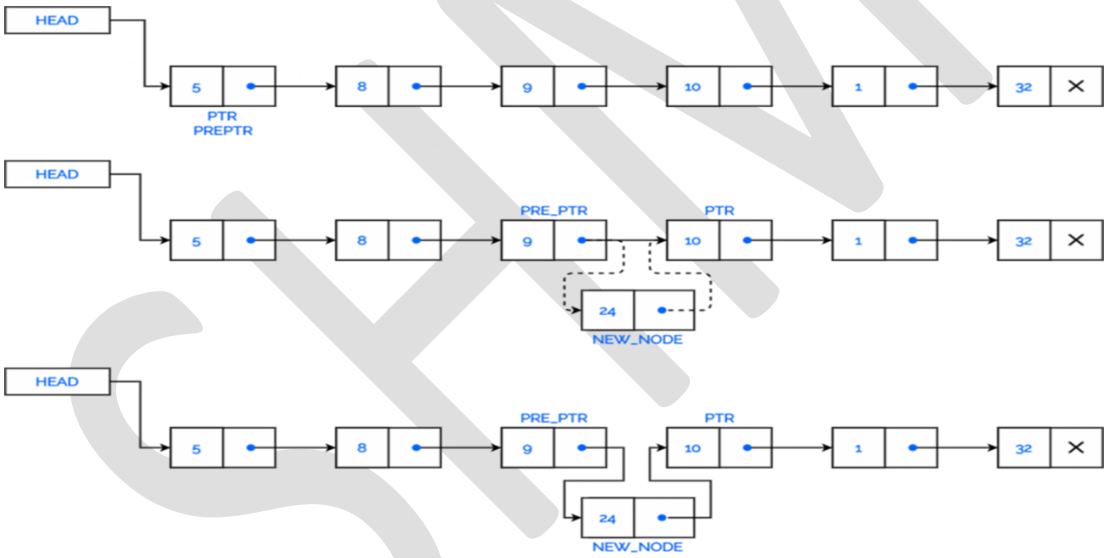
[END OF LOOP]

Step 9: SET PTR -> NEXT = NEW_NODE

Step 10: EXIT

Insert a Node after a given Node in a Linked list

- The last case is when we want to add a new node after a given node. Suppose we want to add a new node with value 24 after the node having data 9. These changes will be done in the linked list.



Allocate memory for new node and initialize its DATA part to 24.

- Traverse the list until the specified node is reached.
- Change NEXT pointers accordingly.

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET PTR = HEAD

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 10 : PREPTR -> NEXT = NEW_NODE

Step 11: SET NEW_NODE -> NEXT = PTR

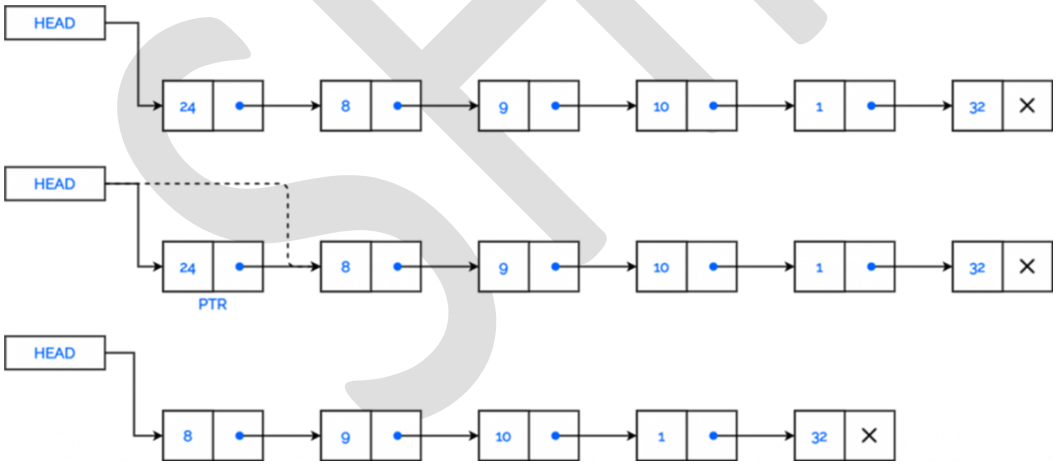
Step 12: EXIT

Deleting Elements from a Linked List

- Let's discuss how a node can be deleted from a linked listed in the following cases.
- The first node is deleted.
- The last node is deleted.
- The node after a given node is deleted.

Delete a Node from the beginning of a Linked list

- Suppose we want to delete a node from the beginning of the linked list. The list has to be modified as follows:



Check if the linked list is empty or not. Exit if the list is empty.

Make HEAD points to the second node.

Free the first node from memory.

Algorithm: DeleteFromBeginning

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 5

[END OF IF]

Step 2: SET PTR = HEAD

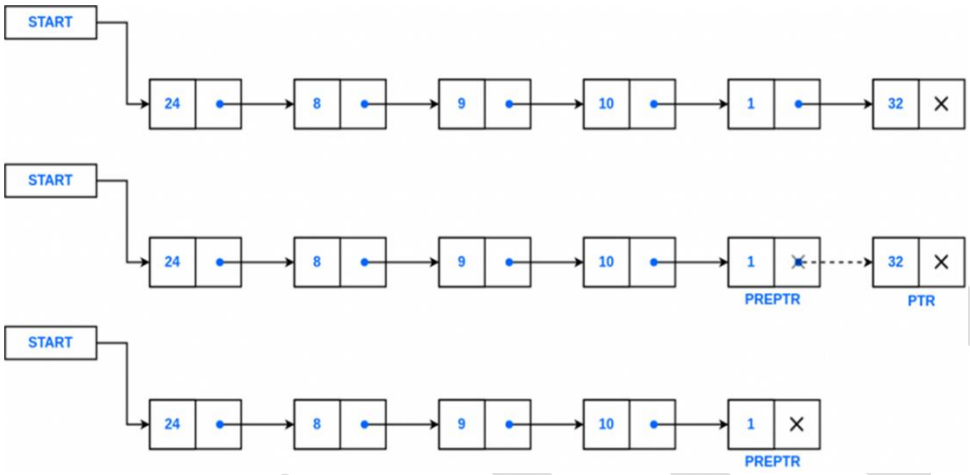
Step 3: SET HEAD = HEAD -> NEXT

Step 4: FREE PTR

Step 5: EXIT

Delete last Node from a Linked list

Suppose we want to delete the last node from the linked list. The linked list has to be modified as follows



- Traverse to the end of the list.
- Change value of next pointer of second last node to NULL.
- Free last node from memory.

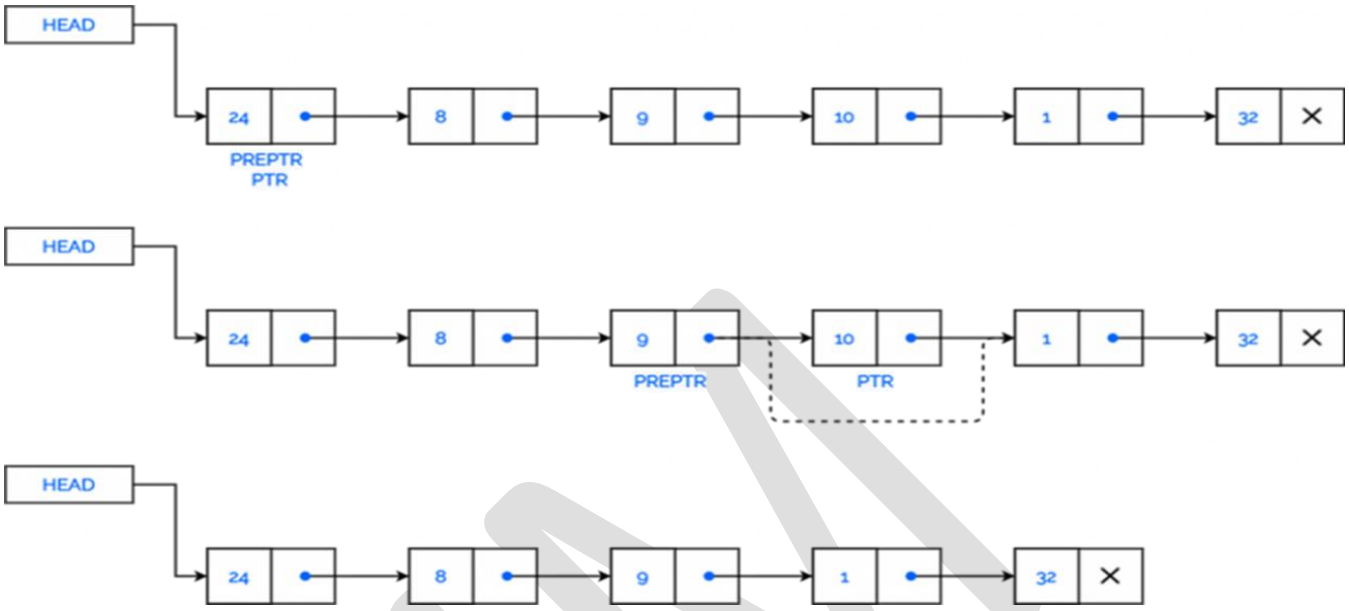
Algorithm: DeleteFromEnd

- Step 1: **IF** HEAD = **NULL**
- Write UNDERFLOW
- Go to Step 8
- [**END OF IF**]
- Step 2: SET PTR = HEAD
- Step 3: Repeat Steps 4 and 5 **while** PTR -> NEXT != **NULL**
- Step 4: SET PREPTR = PTR
- Step 5: SET PTR = PTR -> NEXT
- [**END OF** LOOP]
- Step 6: SET PREPTR -> NEXT = **NULL**
- Step 7: FREE PTR
- Step 8: EXIT

- Here we use two pointers PTR and PREPTR to access the last node and the second last node

Delete the Node after a given Node in a Linked list

- Suppose we want to delete the that comes after the node which contains data 9.



- Traverse the list upto the specified node.
- Change value of next pointer of previous node(9) to next pointer of current node(10).

Algorithm: DeleteAfterANode

Step 1: IF HEAD = NULL

Write UNDERFLOW Go to Step 10

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

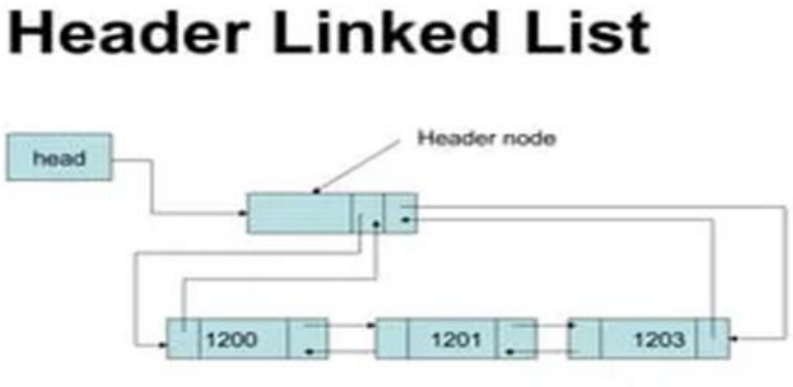
Step 8: SET PREPTR -> NEXT = PTR -> NEXT

Step 9: FREE TEMP

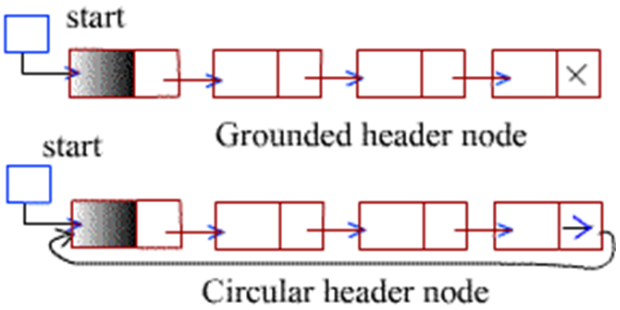
Step 10 : EXIT

Header Linked List

- A header linked list is a type of linked list that has a header node at the beginning of the list. In a header linked list, HEAD points to the header node instead of the first node of the list.

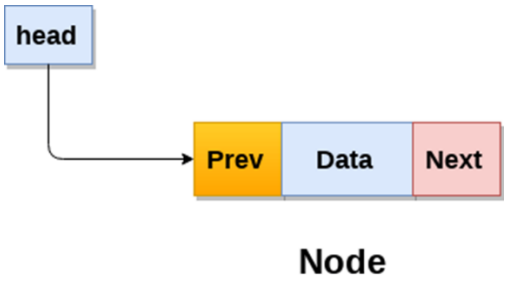


- The header node does not represent an item in the linked list.
- This data part of this node is generally used to hold any global information about the entire linked list.
- The next part of the header node points to the first node in the list.
- A header linked list can be divided into two types:
- Grounded header linked list that stores NULL in the last node's next field.
- Circular header list that stores the address of the header node in the next part of the last node of the list.



Doubly Linked List

- A doubly linked list is a type of linked list that contains a pointer to the next node as well as the previous node in the sequence.



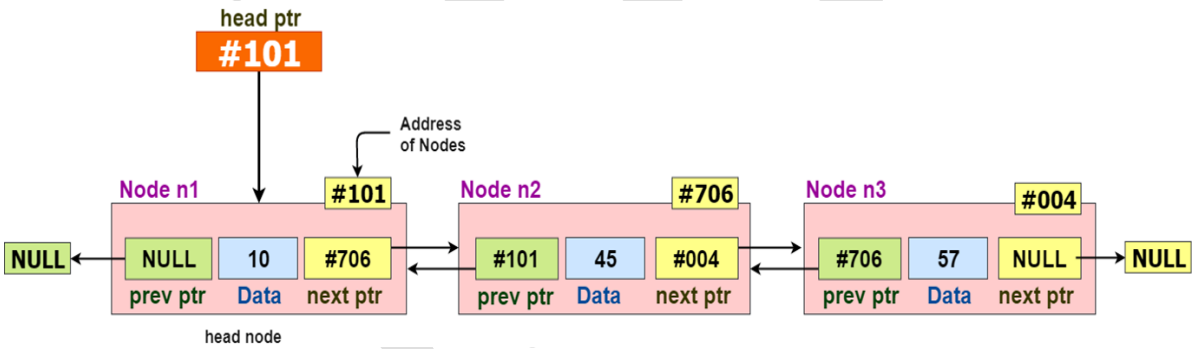
Doubly Linked List

- That is, each node in a doubly-linked list consists of :
- Data:- Data Item
- Prev:-Address of the previous node.
- Next:-Address of the next node

Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

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



Insertion on a Doubly Linked List

- Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.
- We can insert element at 3 different positions of a doubly linked list:
- Insertion at the beginning
- Insertion in-between nodes
- Insertion at the end.

1.Insertion at the beginning

- Create a new node
- Set prev and next pointers of new node
- Make new node as head node

2. Insertion in-between two nodes

- Create a new node
- Set the next pointer of new node and previous node
- Set the prev pointer of new node and the next node.

3. Insertion at the end

- Create a new node
- Set prev and next pointers of new node and the previous node

Deletion from a Doubly Linked List

1.Delete the First Node of Doubly Linked List

- If the node to be deleted is at the beginning.
- Reset value node after the del_node(i.e.node two)

2.Deletion of the Inner Node

For the node before the del_node(first node)

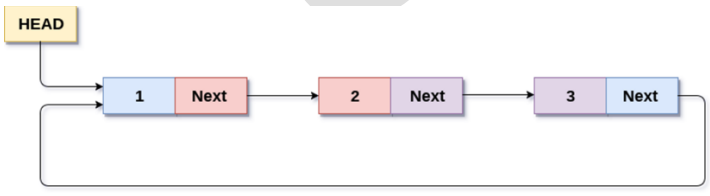
- Assign the value of next node of de_node to the next of the first node.
- **For the node after the del_node(third node)**
- Assign the value of prev of del_node to the prev of the third node.

3.Delete the Last Node of Doubly Linked List

- In this case, we are deleting the last node with value 3 of the doubly linked list.
- Here, we can simply delete the del_node and make the next of node. Before del_node point to NULL

Circular Linked List

- A circular linked list is a type of linked list in which the last node is also connected to the first node to form a circle.
- Thus circular linked list has no end.



Circular Singly Linked List

There are two types of circular linked lists:

- Circular singly linked list
- Circular Doubly linked list

Representation of Circular Linked List

- Each of the nodes in the linked list consists of two parts
- A data item.
- An address that points to another node.
- A single node can be represented using structure as

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

Step 1: IF PTR = NULL//**Insert node at the beginning**

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = HEAD

Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 7: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD

Step 9: SET TEMP → NEXT = NEW_NODE

Step 10: SET HEAD = NEW_NODE

Step 11: EXIT

Step 1: IF PTR = NULL//**Insert node at the end**

Write OVERFLOW

Go to Step 1

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET TEMP = HEAD

Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10: EXIT

// **Delete node at the beginning**

1. Step 1: IF HEAD = NULL.
2. Step 2: SET PTR = HEAD.
3. Step 3: Repeat Step 4 while PTR → NEXT != HEAD.
4. Step 4: SET PTR = PTR → next.
5. Step 5: SET PTR → NEXT = HEAD → NEXT.
6. Step 6: FREE HEAD.
7. Step 7: SET HEAD = PTR → NEXT.
8. Step 8: EXIT.

//**Delete node at the end of circular linked list**

- Step 1: IF HEAD = NULL.
- Step 2: SET PTR = HEAD.
- Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD.
- Step 4: SET PREPTR = PTR.
- Step 5: SET PTR = PTR -> NEXT.
- Step 6: SET PREPTR -> NEXT = HEAD.
- Step 7: FREE PTR.
- Step 8: EXIT.

Applications of Linked List

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