Linked List

Categories of data structures

Data structures are categories in two classes –

1. **<u>Linear data structure:</u>**

   - organized into sequential fashion

   - elements are attached one after another

   - easy to implement because follow the computer memory organization

   - Traversing is very easy

   - Examples – array, linked list, stack, queue etc.

2. **<u>Non - Linear data structure:</u>**

   -**every item is attached with many other items.**

   - data elements are not organized in a sequential fashion

   - data cannot be traversed in a single run.

   - Implementation is difficult

   - Examples – tree, graph etc.

Array vs Linked Lists

1. **Access :Random / Sequential**

➢Array elements can be randomly accessed using **subscript variable.**
e.g. a[0],a[1],a[3] etc.
➢While random access of any node is not possible in linked list we have to traverse through the linked list for accessing element. So **O(n)** time is required for accessing particular element .

2 . Memory Structure :

➢Array is stored in **contiguous Memory** locations , i.e. suppose first element is stored at 2000 then second integer element will be stored at 2004 .
➢But in linked list it is not necessary to store **next element** at the consecutive memory location .
➢Element in linked lists are stored at any available location , but the **pointer** to that memory location is stored in previous Node.

Array vs Linked Lists

3 . Insertion / Deletion

➢As the **array** elements are stored in **consecutive memory** locations , so while inserting elements we have to create space for insertion.

✓So more time is required for **creating space** and inserting element

➢Similarly if we have to delete the element from given location and then **shifting of all successive elements towards left is required**

➢In linked list we have to just change the pointer address field (pointer),so insertion and deletion operations are quite easy to implement

4 . Memory Allocation :

➢In case of array, generally memory allocated is done at **compile-time in stack** area.

➢In **linked list,** memory is allocated at **run-time** , using dynamic memory allocation in heap area using functions like **malloc, calloc and new** etc.

Lists Implementation

<u>Linked List</u> –

- Collection of similar data elements stored at discrete memory locations and connected by pointer (link). Hence size can grow or shrink at any point of time.
- Data elements are called **nodes**

Types –

1. Linear linked list or singly linked list or one way list
2. Doubly linked list or two way list
3. Circular linked list
4. Header linked list

Linear Linked list

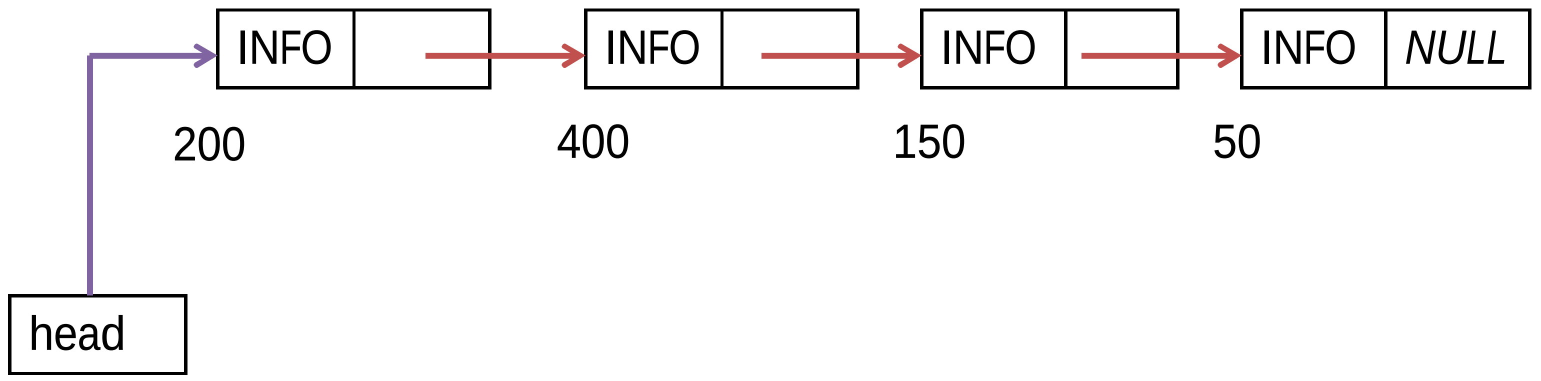
- Each node have two fields – 1. Information 2. Pointer to next node

Representation of node –

```
struct Node{
        int INFO;
        struct Node *NEXT;
};
```

For head –

```
struct Node *head
```



Operations on linear linked list

1. Creating empty linked list
2. Traversing
3. Insertion
4. Deletion
5. Searching
6. Sorting

Creating a linked list

```
struct Node{
            int INFO;
            struct Node *NEXT;
};
```

Algorithm Create_List()

```
1. Node * START
2. START= NULL
3. Return START
```

```
//C program
struct Node*  Create_List()
{   struct Node *  START;
    START= NULL;
    return START;
  }
```

This function creates an empty linked list where START is a pointer pointing to created list.

Traversing of linked list

Traversing the Linked List:

This function visits each node of linked list only once. Temp is a temporary pointer to node. START is the pointer pointing to starting of list.

```
Algorithm Traverse_List(START)
1.  Node *  Temp
2.  Temp = START
3.  While (Temp != NULL)
4.          Print Temp -> INFO
5.          Temp = Temp -> NEXT
```

```
void Traverse_List(Node START)
{
 struct Node * Temp; //Temp pointer
Temp= START;
  while (Temp!= NULL)
  {
     printf(" %d ", Temp -> INFO );
      Temp = Temp ->next;
  }
}
```

Insertion

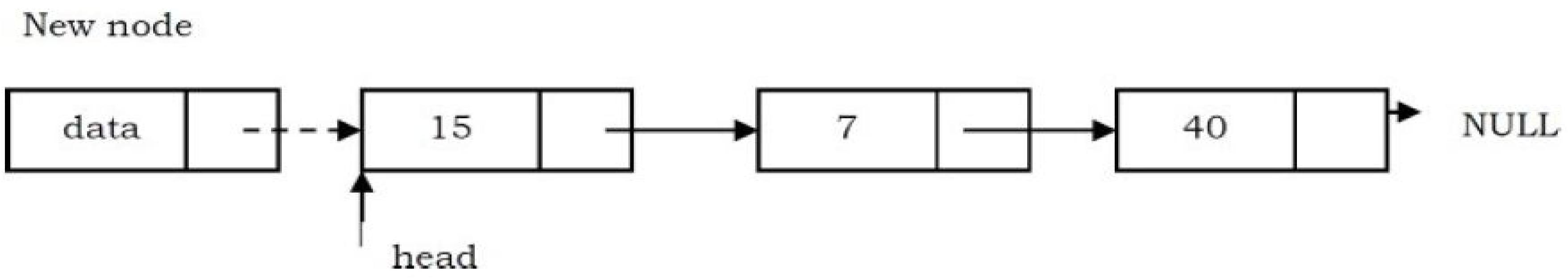
Three ways to insert a node into linked list –

1. ***Insert at beginning***
2. ***Insert after specified location***
3. ***Insert at the end (appending node)***

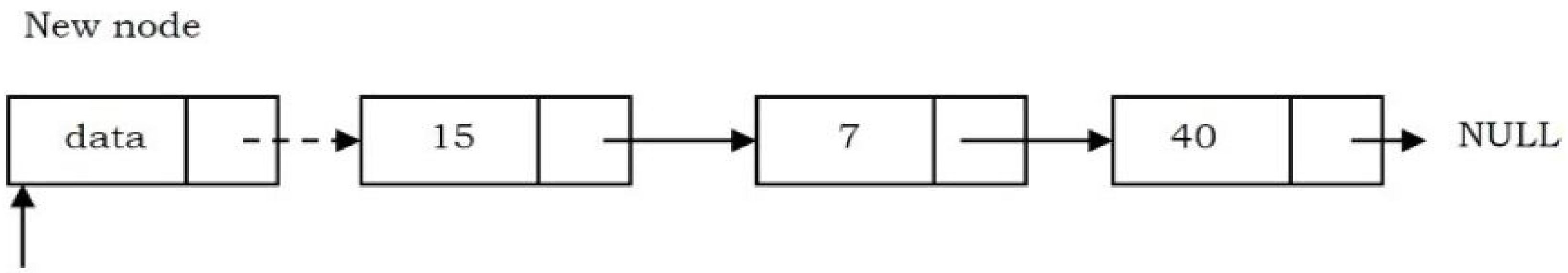
Insertion at beginning of linked list

In this case, a new node is inserted before the current head node. *Only one next pointer needs to* be modified (*new node's next pointer) and* It can be done in two steps:

Step 1. Update the next pointer of new node, to point to the current head



Step 2. Update head pointer to point to the new node.



Insertion at beginning of linked list

Insert at the Beginning of the Linked List:

This function adds the node at the starting of linked list. **New_Node** is temporary pointer to node to be inserted into list. **START** is the pointer pointing to starting of list and **info** is the information of node.

```
Algorithm InsertBeg_List(START, info)
1.  Node * New_node
2.  New_Node = Allocate memory   \\allocate memory for new node
3.  New_Node -> INFO =info     \\put info in INFO field of new node
4.  New_node -> NEXT = START
5.  START = New_node
```

Insertion at after specified location of linked list

This algorithm adds the node after the specified location of linked list. **Temp** is temporary pointers and **New_Node** is the pointer to the node to be inserted. **START** is the pointer pointing to starting of list and **info** is the information of node. **Loc** is the numbered location after which node is to be inserted. **i** is a loop variable.

```
Algorithm InsertAfter_List(START, info, Loc)
1.  Node * Temp, *New_node
2.  New_node = Allocate memory
3.  New_node -> INFO = info
4.  Temp = START
5.  If (START == NULL)          \\  if there is no node in list then make new node START
6.          START = New_Node
7.          New_node -> NEXT =NULL
8.  Else
9.          For i=1 to Loc-1
10.                 Temp  = Temp -> NEXT
11.                 If (Temp == NULL)  //if Loc is greater than number of nodes
12.                         Print "Loc is greater than number of nodes"
13.                         Return
14.         New_node -> NEXT = Temp -> NEXT
15.         Temp -> NEXT = New_node
```

Insertion at End of linked list (Appending node)

Append node in a Linked List: *Below* function adds the node at the end of linked list. **New_Node** is the node to be appended and Temp is temporary pointer. START is the pointer pointing to starting of list and info is the information of node.

```
Algorithm Append_List(START, info)
1.  Node* New_Node, *Temp
2.  New_Node = Allocate memory   \\allocate memory for new node
3.  New_Node -> INFO =info     \\put info in INFO field of new node
4.  New_Node -> NEXT = NULL    \\make it last node by keeping NULL in NEXT field
5.  If (START == NULL)         \\  if there is no node in list then make new node START
6.          START = New_Node
7.  Else
8.          Temp = START
9.          While (Temp -> NEXT != NULL)
10.                 Temp = Temp ->NEXT
11.         Temp -> NEXT = New_Node
```

Deletion

Three ways to delete a node from linked list –

1. ***Delete from beginning***
2. ***Delete after specified location***
3. ***Delete from the end***

NOTE: Deletion could be done on the basis of given data. Here, first find that node and then delete it.

Deletion from the beginning of linked list

This algorithm deletes the node from the starting of linked list. **Temp** is temporary pointer.

START is the pointer pointing to starting of list.

```
Algorithm DeleteBeg_List(START)
1.  Node* Temp = START
2.  If START ==NULL
3.          Print "Empty Linked List"
4.          Return
5.  START = START -> NEXT
6.  Free Temp
```

Deletion after specified location of linked list

This algorithm deletes the node after the specified location of linked list. **Temp** and **Temp1** are temporary pointers. **START** is the pointer pointing to starting of list. **Loc** is the numbered location from where next node is to be deleted. i is a loop variable.

```
Algorithm DeleteAfter_List(START, Loc)
1.  Node* Temp, *Temp1
2.  Temp = START
3.  For i=1 to Loc-1
4.          Temp  = Temp -> NEXT
5.          If (Temp->NEXT==NULL)
6.                  Print "Either Loc is Last node or greater than Total number of nodes"
7.                  Return
8.  Temp1 = Temp ->NEXT
9.  Temp -> NEXT = Temp1 -> NEXT
10. Free Temp1
```

Deletion from the End of linked list

This algorithm deletes the node from the end of the linked list. **Temp** and **Temp1** are temporary pointers. **START** is the pointer pointing to starting of list.

```
Algorithm DeleteEnd_List(START) –
1.  Node * Temp, *Temp1
2.  Temp1 = START
3.  If START==NULL   //checking Empty Linked list
4.          Print "Empty Linked List"
5.          Return
6.  If(START->NEXT==NULL) // check single node list
7.          START=NULL
8.          Free Temp1
9.          Return
10. While (Temp1 -> NEXT != NULL)
11.         Temp =Temp1
12.         Temp1 = Temp1 -> NEXT
13.  Temp -> NEXT = NULL
14.  Free Temp1
```

Searching element in singly linked list

- **Only linear search – As there is no way to find out the middle of linked list.**

This algorithm searches the node in linked list. Temp is temporary pointer. **START** is the pointer pointing to starting of list and **info** is the information of node to be searched. If element is found then **Loc** is used to returned the location of node in list.

```
Algorithm SearchUnsorted_List(START,info) –
1.  Node *Temp
2.  Initialize Loc = 0
3.  Temp = START
4.  If Temp==NULL //empty linked list
5.          Exit
6.  While (Temp != NULL)
7.          Loc = Loc +1
8.          If (Temp->INFO == info)
9.                  Return Loc OR Temp
10.         Temp = Temp -> NEXT
11. Return NULL // return NULL when element node found
```

Searching element in **sorted** singly linked list

This function will search the node in linked list. **Temp** is temporary pointer. **START** is the pointer pointing to starting of list and **info** is the information of node to be searched. If element is found then **Loc** is used to returned the location of node in list.

```
Algorithm SearchSorted_List(START,info) –
1.  Node *Temp
2.  Initialize Loc = 0
3.  Temp = START
4.  If Temp==NULL
5.          Exit
6.  While (Temp  != NULL)
7.          Loc = Loc+1
8.          If (Temp->INFO == info)
9.                  Return Loc OR Temp
10.         Else If (Temp->INFO > info)
11.                 Return NULL
12.         Temp = Temp ->NEXT
13. Return NULL
```

Reverse the singly linked list

This algorithm reverses the linked list. **Pnode, Cnode and Nnode** are temporary pointers. **START** is the pointer pointing to starting node of the linked list.

```
Algorithm Reverse_List(START) –
1.  Node *Cnode, *Pnode, *Nnode
2.  Cnode =START
3.  Nnode = Cnode ->NEXT
4.  Cnode -> NEXT =NULL
5.  While (Nnode != NULL)
6.          Pnode = Cnode
7.          Cnode = Nnode
8.          Nnode = Nnode ->NEXT
9.          Cnode -> NEXT = Pnode
10. START = Cnode
```

Arrays Vs. Linked List

Array	Linked List
Elements stored at contiguous memory locations Size of array is static and can not be changed at later stage if need arise.	Elements stored at discrete memory locations No fixed size, can grow and shrink very efficiently
Traversing takes O(n) Time	Traversing takes O(n) Time
Searching – Unsorted list – Linear O(n) Sorted list – Binary O($\log_2 n$)	Searching – Only Linear O(n)
Insertion – Shift all element one position right, Less efficient than in linked list	Insertion – Only some pointers need to be managed, more efficient than in array
Deletion – Shift all element one position left, Less efficient than in linked list	Deletion – Only some pointers need to be managed, more efficient than in array

Doubly Linked list

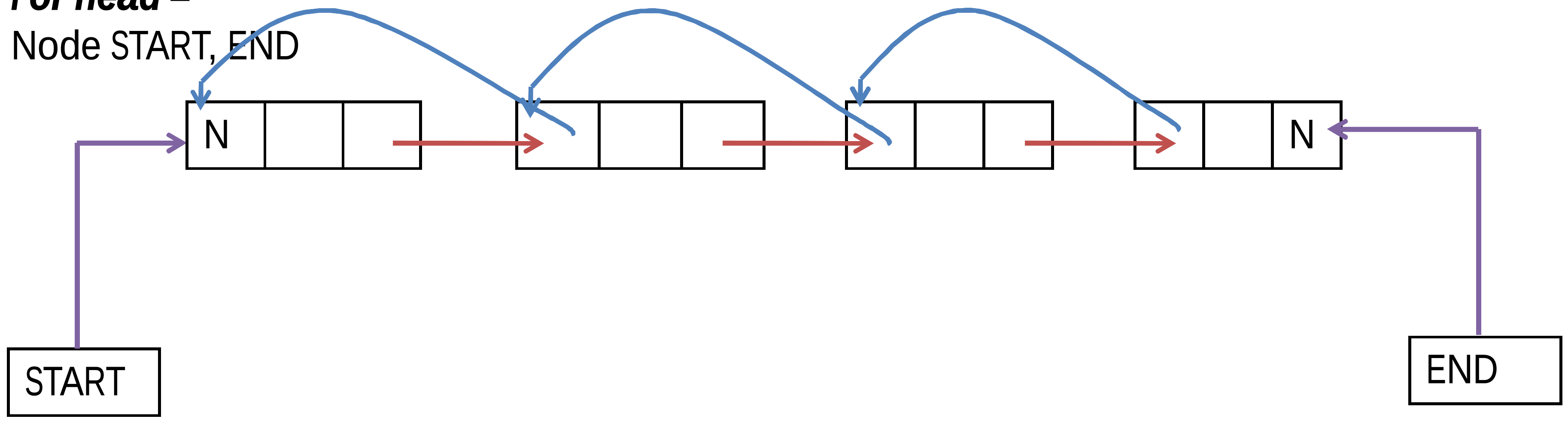
- Each node have three fields – 1. Information 2. Pointer to next node 3. Pointer to previous node

Representation of node –

```
struct Node{
            struct Node *PREV;
            int INFO;
            struct Node *NEXT;
 };
typedef struct Node * Node;
```

For head –

Node START, END



Creating a doubly linked list

This function creates an empty linked list and START is a pointer pointing to created list.

Algorithm Create_DList() –

1. Node START, LAST
2. START= *NULL*
3. END = *NULL*

Traversing of Doubly linked list

This function visits each node of linked list only once. Temp is a temporary pointer to node.

START and END are the pointers pointing to starting of list and end of list respectively.

Algorithm Traverse_DList(START, END) –

```
1.  Node Temp
2.  Temp = START
3.  While (Temp != NULL)
4.          Print Temp -> INFO
5.          Temp = Temp -> NEXT
```

OR (from last to first)

```
1.  Node Temp
2.  Temp = END
3.  While (Temp != NULL)
4.          Print Temp -> INFO
5.          Temp = Temp -> PREV
```

Insertion

Four ways to insert a node into doubly linked list –

1. ***Insert at beginning***
2. ***Insert after specified location***
3. ***Insert before specified location***
4. ***Insert at the end (also known as appending)***

Insertion at beginning of Doubly linked list

This function add the node at the starting of linked list. New_node is temporary pointer to node to be inserted into list. START is the pointer pointing to starting of list and info is the information of node.

```
Algorithm InsertBeg_DList(START, info) –
1.  Node New_node
2.  New_node = Allocate memory
3.  New_node -> INFO =info
4.  New_node -> NEXT = START
5.  New_node ->PREV = NULL
6.  If  START!=NULL    //  linked list is not empty
7.          START -> PREV=  New_node
8.  START = New_node
```

Insertion after specified location of doubly linked list

This function add the node after the specified location of linked list. Temp and Temp1 are temporary pointers. START is the pointer pointing to starting of list and info is the information of node. Loc is the numbered location after which node is to be inserted. i is a loop variable.

```
Algorithm InsertAfter_DList(START, info, Loc) –
1.  Node Temp, New_node
2.  Temp = START
3.  For i=1 to Loc-1
4.          Temp  = Temp -> NEXT
5.          If (Temp == NULL)
6.                  Exit
7.  New_node = Allocate memory
8.  New_node -> INFO = info
9.  New_node -> NEXT = Temp -> NEXT
10. New_node ->PREV = Temp
11.  (Temp -> NEXT) -> PREV = New_node
12. Temp -> NEXT = New_node
```

Insertion before specified location of D linked list

This function add the node before the specified location of linked list. Temp and New_node are temporary pointers. START is the pointer pointing to starting of list and info is the information of node. Loc is the numbered location before which node is to be inserted. i is a loop variable.

```
Algorithm InsertBefore_DList(START, info, Loc) –
1.  Node Temp, New_node
2.  Temp = START
3.  For i=1 to Loc-1
4.          Temp  = Temp -> NEXT
5.          If (Temp == NULL)
6.                  Exit
7.  New_node = Allocate memory
8.  New_node -> INFO = info
9.  New_node -> NEXT = Temp
10. New_node ->PREV = Temp ->PREV
11.  (Temp -> PREV) -> NEXT = New_node
12. Temp -> PREV= New_node
```

Insertion at End(Appending) of Doubly linked list

This function add the node at the end of linked list. New_node is temporary pointers. START and END are the pointers pointing to starting of list and end of list respectively. info

Algorithm InsertEnd_DList(START ,END, info) –

```
1.  Node  New_node
2.  New_node = Allocate memory
3.  New_node -> INFO =info
4.  New_node -> NEXT = NULL
5.  If (START == NULL  OR  END == NULL)  \\  if there is no node in list
6.          New_node -> PREV = NULL
7.          START = New_node
8.          END = New_node
9.  Else
10.         New_node -> PREV = END
11.         END -> NEXT = New_node
12.         END = New_node
```

Deletion

Four ways to Delete a node from linked list –

1. ***Delete from beginning***
2. ***Delete after specified location***
3. ***Delete before specified location***
4. ***Delete from the end***

NOTE: Deletion could be done on the base information (to be deleted) available at any node or not. Here, first find that node and then delete it.

Deletion at beginning of linked list

This function delete the node at the starting of linked list. Temp is temporary pointer. START and are the pointers pointing to starting of list and end of list.

```
Algorithm DeleteBeg_DList(START,END) –
1.  Temp = START
2.  If Temp ==NULL
3.          Exit
4.  If (START == END) // Single Node
5.          START = END =NULL
6.  ELSE
7.          (Temp ->NEXT) -> PREV =NULL
8.          START = START -> NEXT
9.  Free Temp
```

Deletion after specified location of linked list

This function delete the node after the specified location of linked list. Temp and Temp1 are temporary pointers. START is the pointer pointing to starting of list and info is the information of node. Loc is the numbered location from where next node is to be deleted. i is a loop variable.

```
Algorithm DeleteAfter_DList(START, Loc) –
1.  Node Temp, Temp1
2.  Temp = START
3.  For i=1 to Loc-1
4.          Temp  = Temp -> NEXT
5.          If (Temp == NULL)
6.                  Exit
7.  Temp1 = Temp ->NEXT
8.  Temp -> NEXT = Temp1 -> NEXT
9.  (Temp1->NEXT) ->PREV = Temp
10. Free Temp1
```

Deletion before specified location of linked list

Algorithm DeleteBefore_DList(START, Loc) –

This function delete the node before the specified location of linked list. Temp and Temp1 are temporary pointers. START is the pointer pointing to starting of list and info is the information of node. Loc is the numbered location from where previous node is to be deleted. i is a loop variable.

```
1.  Node Temp, Temp1
2.  Temp = START
3.  For i=1 to Loc-1
4.          Temp  = Temp -> NEXT
5.          If (Temp == NULL)
6.                      Exit
7.  Temp1 = Temp ->PREV
8.  Temp -> PREV = Temp1 -> PREV
9.  (Temp1->PREV) ->NEXT = Temp
10. Free Temp1
```

Deletion at End of linked list

This function delete the node at the end of linked list. Temp and Temp1 are temporary pointers. END is the pointer pointing to end of list.

```
Algorithm DeleteEnd_List(END) –
1.  Node Temp, Temp1
2.  Temp = END
3.  If Temp==NULL
4.          Exit
5.  Temp1 = Temp -> PREV
6.  Temp1 -> NEXT = NULL
7.  END = Temp1
8.  Free Temp
```

Singly Vs. Doubly Linked List

Singly Linked List	Doubly Linked List
Node has two fields.	Node has three fields.
Reverse traversing is not possible.	Reverse traversing is possible.
Insertion at the end of linked list takes O(n) time.	Insertion at the end of linked list take constant time
Deletion at the end of linked list take O(n) time and deletion before specified location is not possible.	Deletion at the end of linked list take constant time and deletion before specified location is also possible.

Exercises

1. Write an algorithm for searching in doubly linked list.
2. Write an algorithm to sorting a singly linked list.
3. Why reverse algorithm is not needed in doubly linked list. Justify your answer comparing with singly linked list.
4. Write an algorithm to delete the entire singly linked list or doubly linked list.

Circular Linked list

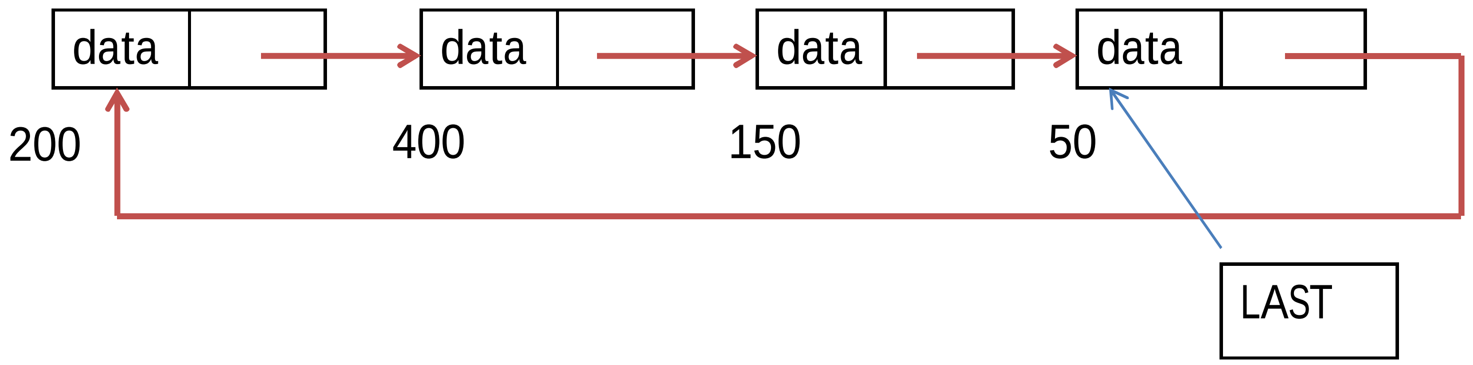
It could be singly or doubly linked list.

For singly circular linked list

- Each node have two fields – 1. Information 2. Pointer to next node

Representation of node –

```
struct Node{
        int INFO;
        struct Node *NEXT;
};
```



Supported operations

- All operation which can be performed on singly or doubly linked list can be extended to circular by maintaining last node next pointer to first node.

1. Creating empty linked list
2. Appending nodes to linked list
3. Traversing
4. Insertion
5. Deletion
6. Searching
7. Sorting

How do we know end of list ?

Compare next pointer field with address of first node or head

Circular Linked list

Advantages:

- We can go to any node, in linear linked list it is not possible to go to previous node.
- It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in doubly linked list, we will have to go through in between nodes.

Disadvantages:

1. It is not easy to reverse the linked list. If proper care is not taken, then the problem of infinite loop can occur.
2. If we want to go back to the previous node, then we can not do it in single step. Entire circle to be covered.

Circular Linked list(Insert at beginning)

Algorithm InsertBeg_List(LAST, info)

```
1.  Node New_node
2.  New_Node = Allocate memory   \\allocate memory for new node
3.  New_Node -> INFO =info     \\put info in INFO field of new node
4.  IF LAST==NULL
5.          New_node -> NEXT = New_node
6.          LAST=New_node
7.  ELSE
8.          New_node -> NEXT = LAST->NEXT
9.          LAST->NEXT = New_node
```

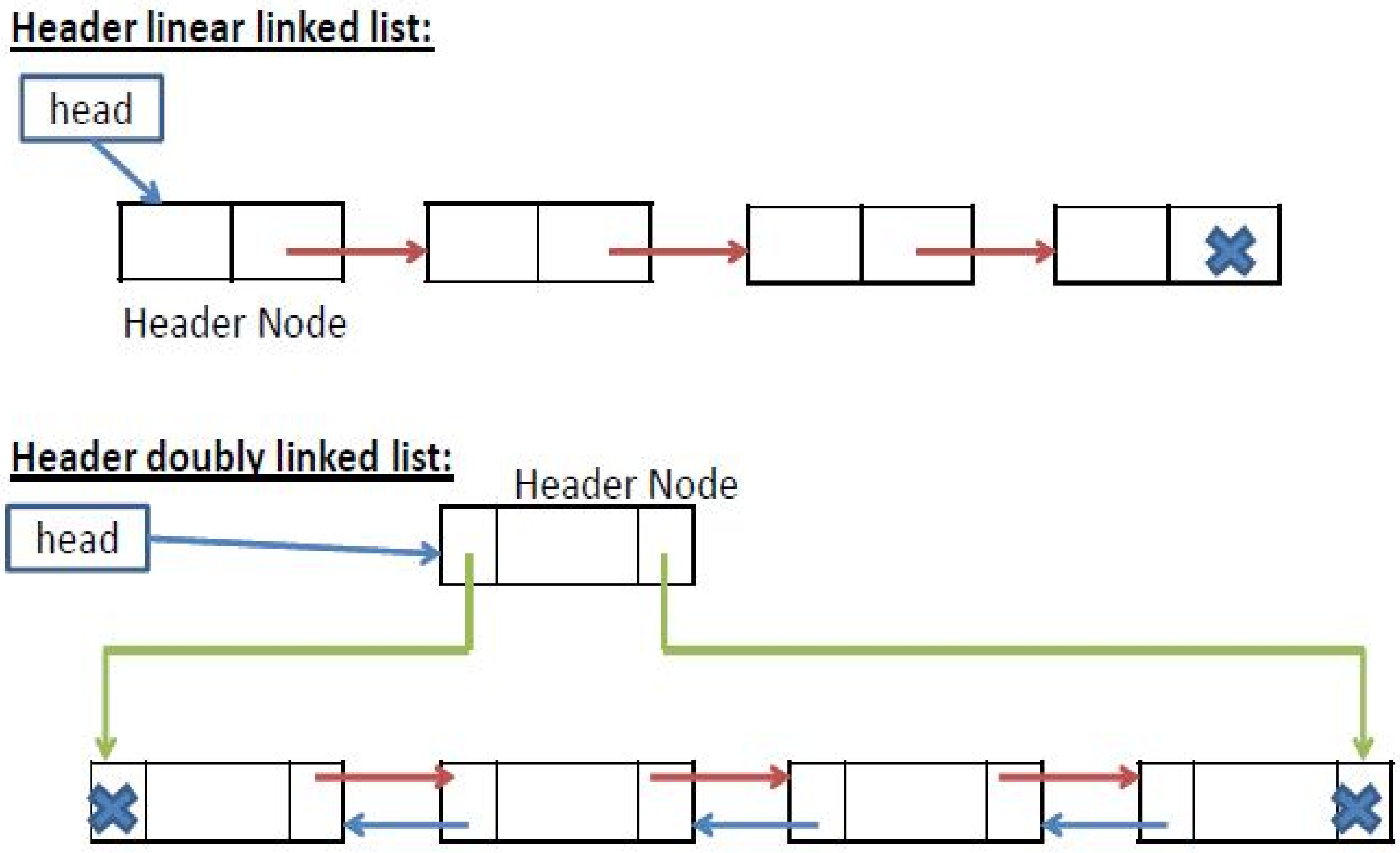
Circular Linked list(Insert at the end)

Algorithm InsertAt_Last_List(LAST, info)

```
1.  Node New_node
2.  New_Node = Allocate memory   \\allocate memory for new node
3.  New_Node -> INFO =info     \\put info in INFO field of new node
4.  IF LAST==NULL
5.          New_node -> NEXT = New_node
6.          LAST=New_node
7.  ELSE
8.          New_node -> NEXT = LAST->NEXT
9.          LAST->NEXT = New_node
10.         LAST=New_node;
```

Header Linked list

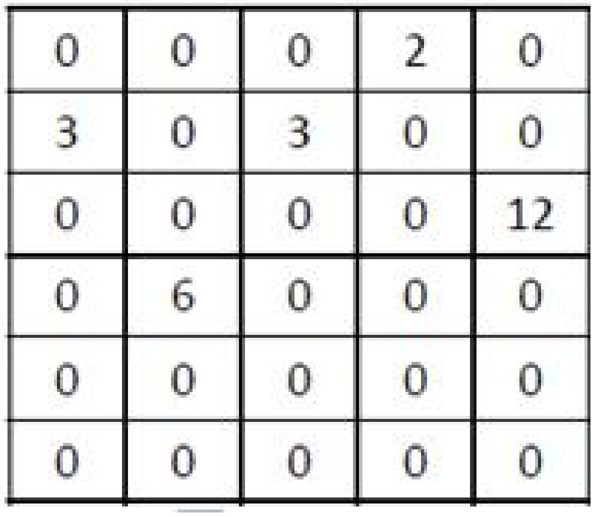
➢A Header linked list is one more variant of linked list. In Header linked list, we have a special node present at the beginning of the linked list.
➢Header node keeps information about list like number of nodes, sorted or not sorted etc.



Application of Linked list

- To implement scheduling algorithms in Operating Systems
- To represent sparse matrices
- To implement other data structures like stack, queue, tree etc.
- To manipulate polynomials
- Open a folder that has images. Use Microsoft Image Viewer to view the images. In the User Interface, '->' button is your **node->next**; The '<-' button is your **node->prev.**
- The cache in browser. (allows you to hit back and fwd links URL's through linked list)
- Undo functionality in photo editors, text editors. (a linked list of different states)

Sparse matrix representation

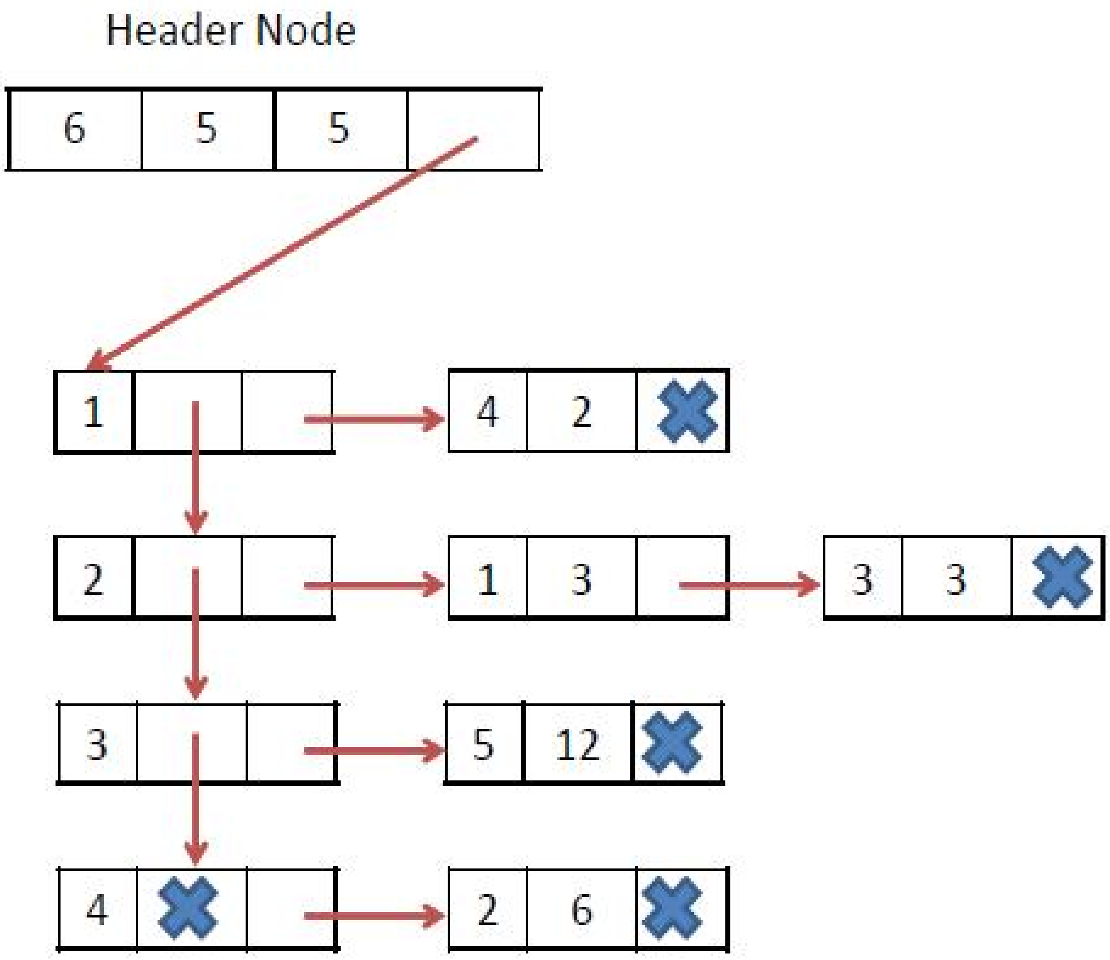


0	0	0	2	0
3	0	3	0	0
0	0	0	0	12
0	6	0	0	0
0	0	0	0	0
0	0	0	0	0

Sparse matrix representation

0	0	0	2	0
3	0	3	0	0
0	0	0	0	12
0	6	0	0	0
0	0	0	0	0
0	0	0	0	0

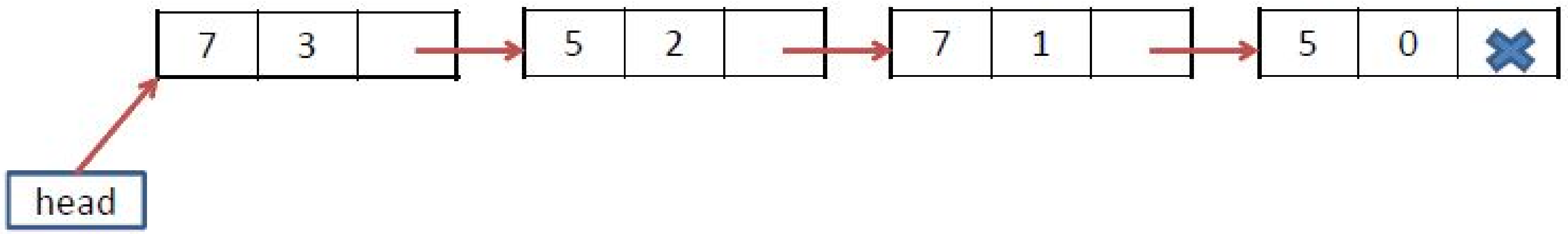
6	5	5
1	4	2
2	1	3
2	3	3
3	5	12
4	2	6



Polynomial representation

Let polynomial is

$7x^3 + 5x^2 + 7x + 5$



Representation of node –

```
struct Node{
        int coef;
        int pow;
        struct Node *NEXT;
};
```

For head –

```
Node *head
```