UNIT –II

Objective:

- To gain knowledge on linked lists.

Syllabus:

Unit-II: Linked lists

Linked Lists- Basic concepts and operations of Single linked list, Circular linked list, Double linked list.

Learning Outcomes:

At the end of the unit student will be able to:

1. Define a self referential structure.
2. Describe about linked lists.
3. Implement the operations on linked lists.
4. Choose an appropriate linked list for a given problem.
5. Distinguish between single, double and circular linked lists.

Learning Material

➢ OPERATIONS ON DATA STRUCTURES:

The basic operations that are performed on data structures are as follows:

1) ***Traversing:***It means to access each data item exactly once so that it can be processed.

   For example, to print the names of all the students in a class.

2) ***Searching:***It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items.

   For example, to find the names of all the students who secured 100 marks in mathematics?

3) ***Inserting: It*** is used to add new data items to the given list of data items.

   For example, to add the details of a new student who has recently joined the course?

4) ***Deleting: It*** means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

5) ***Sorting:*** Dataitems can be arranged in some order like ascending order or descending order depending on the type of application.

For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

6) ***Merging:*** *Lists* of two sorted data items can be combined to form a single list of sorted data items.

➢ ***AbstractDataType:***

- Abstract data type is a Mathematical model or concept that defines a data type logicly.
- **ADT** specifies a set of data and collection of operations that can be performed on that data.
- The definition of ADT only mentions ***What Operations are to be Performed but not how it is implemented***.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "**abstract**" because it gives an implementation independent view (overview).
- The process of providing only the essentials and hiding the details is known as **Abstraction.**

➢ LINKED LISTS:

- In arrays once memory is allocated, it can't extended any more. So array is known as static Data Structure.
- Linked List is dynamic Data Structure, where amount of memory required can be vary during it use.

Definition: A Linked List is an ordered collection of finite, homogeneous data elements called nodes.

Where the linear order is maintained by means of links or pointers.

- The representation of node is as follows:

Data	Link

Node: an element in Linked List

- A node consists of two parts. i.e. **Data part** and **Link part**.
- The data part contains actual data to be represented.
- The link part is also referred as address field, which contains address of the next node.

- **Representation of Linked List in memory**

  There are two ways to represent a Linked List in memory.

  1. Static memory allocation using arrays.
  2. Dynamic allocations using pool of storage.

Dynamic allocations using pool of storage

- The efficient way of representation of linked list is using pool of storage.
- In this method memory bank, memory manager and garbage collector is available.

Memory bank: is a collection of free memory spaces.

Memory manager: is a program.

- Whenever a linked list requires a anode, then request is placed to memory manager.
- If the required node is available in the memory bank, then that node is send to caller.
- If the required node is not available in the memory bank, then memory manager send NULL to caller.

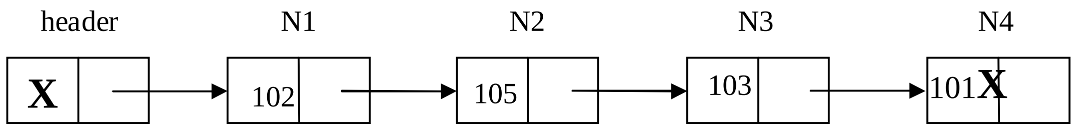
Garbage collector:collect the unused nodes in the linked list and send back to memory bank.

Types of linked lists

1. Single Linked List (SLL)
2. Circular Linked List (CLL)
3. Double Linked List (DLL)

1.Single Linked List (SLL):

- In SLL, each node contains only one link, which points to the next node in the list.
- The pictorial representation of SLL is as follows.



SLL with 4 nodes

- Here header is an empty node, i.e. data part is NULL, represented by X mark.
- The link part of the header node contains address of the first node in the list.
- In SLL, the last node link part contains NULL.
- In SLL we can move from left to right only. So SLL is called as one way list.

- **Operations on Single Linked List**

    1. Traversing a SLL
    2. Insertion of a node in to SLL
    3. Deletion of a node from SLL
    4. Search for a node in SLL
    5. Reversing a SLL

1. Traversing a SLL

Traversing a SLL means, visit every node in the list starting from first node to the last node.

```
AlgoritmSLL_Traverse(header)
Input: header is a header node.
Output: Visiting of every node in SLL.
    1. ptr=header
    2. while(ptr.link != NULL)
        a) ptr=ptr.link go to step(b)
        b) print "ptr.data"
    3. end loop
End SLL_Traverse
```

2. Insertion of a node into SLL

- The Insertion of a node in to SLL can be done in various positions.
    - i) Insertion of a node into SLL at beginning.
    - ii) Insertion of a node into SLL at ending.
    - iii) Insertion of a node into SLL at any position.
- For insertion of a node into SLL, we must get node from memory bank.
- The procedure for getting node from memory bank is as follows:

Procedure for getnewnode( )

```
    1. Check for availability of node in memory bank
    2. if ( AVAIL = NULL)
        a) print "Required node is not available in memory"
        b) return NULL
    3. else
        a) return address of node to the caller
        4. end if
```

End Procedure for getnewnode

i) <u>Insertion of a node into SLL at beginning</u>

AlgoritmSLL_Insert_Begin(header,x)

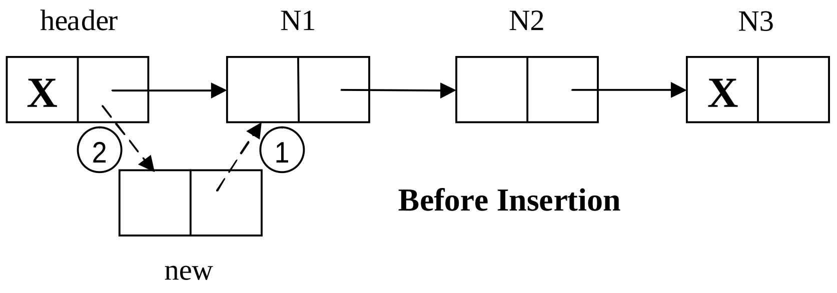
Input: header is a pointer to the header node, x is data part of new node to be inserted.

Output: SLL with new node inserted at beginning.

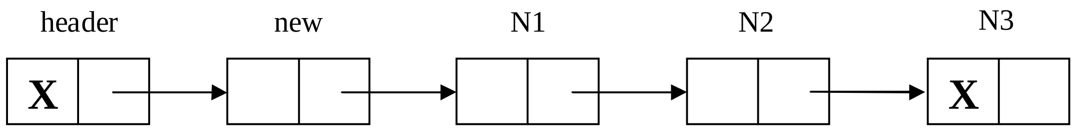
```
        1. new=getnewnode( )
        2. if(new = = NULL)
                a) print "required node was not available in memory, so unable to process"
        3. else
                a) new.link=header.link          /* 1 */
                b) header.link=new               /* 2 */
                c) new.data=x
        4. end if
```

End SLL_Insert_Begin

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address



Before Insertion



After Insertion

.

<u>ii) Insertion of a node into SLL at ending</u>

- To insert a node into SLL at beginning first we need to traverse to last node, then insert as new node as last node.

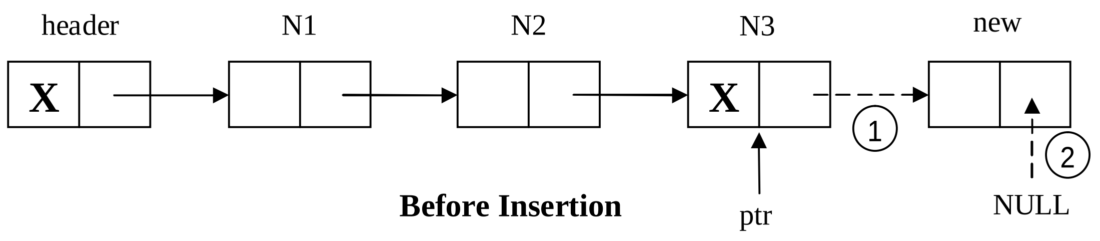
Algorithm SLL_Insert_Ending(header,x)

Input: header is header node, x is data part of new node to be insert.

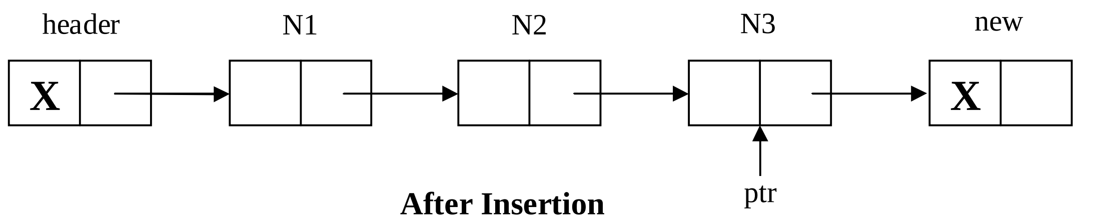
Output: SLL with new node at ending.

```
1. new=getnewnode()
2. if(new = NULL)
        a) print "Required node was not available in memory bank, so unable to process"
3. else
        a) ptr=header
        b) while(ptr.link!=NULL)
                i) ptr=ptr.link go to step(b)
        c) end loop
        d) ptr.link=new              /* 1 */
        e) new.link=NULL             /* 2 */
        f) new.data=x
4. end if
```

End_SLL_Insert_Ending



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new node becomes the last node.

<u>iii) Insertion of a node into SLL at any position.</u>

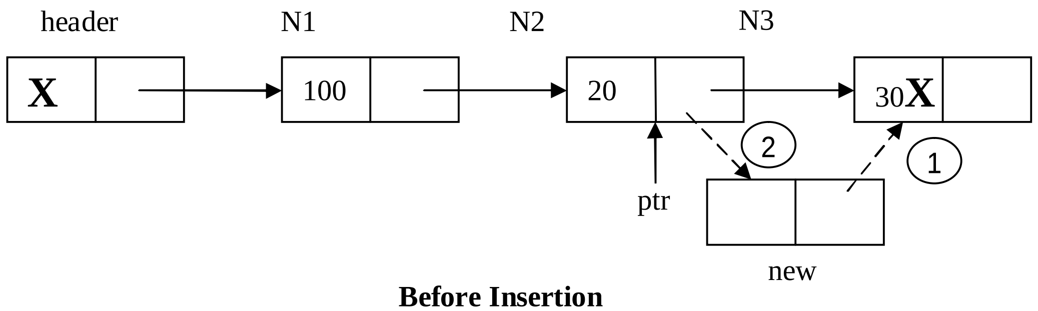
- For insertion of a node at any position in SLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

Algorithm SLL_Insert_ANY(header,x,key)

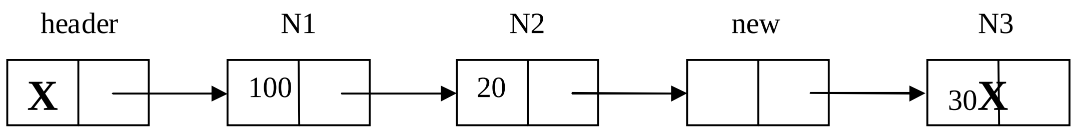
Input: header is header node, x is data pat of new node to be inserting, key is the data part of a node, after this node we want to insert new node.

Output: SLL with new node at ANY

```
1. new=Getnewnode( )
2 .if(new = = NULL)
        a. print "required node was not available in memory bank, so unable to process"
3. else
        i. ptr=header
        ii. while(ptr.data!=key and ptr.link!=NULL)
                a) ptr=ptr.link
        iii. end loop
        iv. if(ptr.link=NULL and ptr.data!=key)
        a) print "required node with data part as key value is not available, so unable to process"
        v. else
                a) new.link=ptr.link            /* 1 */
                b) ptr.link=new                 /* 2 */
                c) new.data=x
        vi. end if
4. end if.
End SLL_insert_ANY
```



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

3. Deletion of a node from SLL

The deletion of a node in from SLL can be done in various positions.

i) Deletion of a node from SLL at beginning.

ii) Deletion of a node from SLL at ending.

iii) Deletion of a node from SLL at any position.

<u>i) Deletion of a node from SLL at beginning</u>

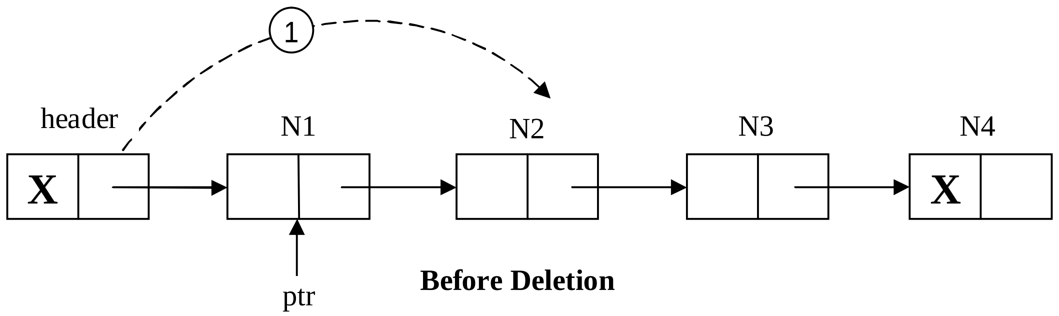
Algorithm SLL_Delete_Begin(header)

Input: Header is a header node.

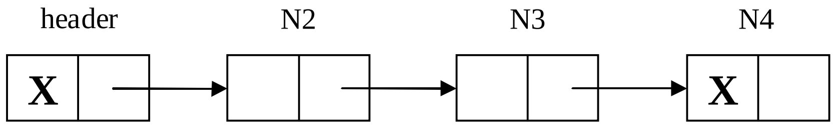
Output: SLL with node deleted at Beginning.

```
1. if(header.link = = NULL)
        a) print “SLL is empty, so unable to delete node from list”
2. else                         /*SLL is not empty*/
        i. ptr=header.link                /* ptr points to first node into list*/
        ii. header.link=ptr.link          /* 1 */
        iii. return(ptr)           /*send back deleted node to memory bank*/
3. end if
```

End SLL_Delete_Begin



Before Deletion



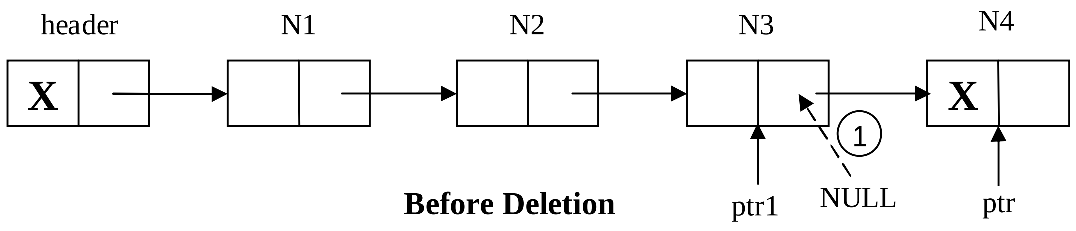
After Deletion

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

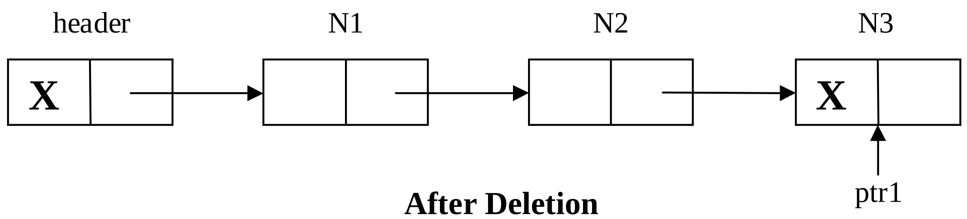
ii) Deletion of a node from SLL at ending

- To delete a node from SLL at ending, first we need to traverse to last node in the list.
- After reach the last node in the list, last but one node link part is replaced with NULL.

```
Algorithm   SLL_ Delete_End (header)
Input:  header is a header node
Output: SLL with node deleted at ending.
    1. if(header.link = = NULL)
        a)print "SLL is empty, so unable to delete the node from list"
    2. else                                   /*SLL is not empty*/
        a) ptr=header                         /*ptr  initially points to header node*/
        b) while(ptr.link!=NULL)
            i) ptr1=ptr
            ii) ptr=ptr.link                  /*go to step b*/
        c) end loop
        d) ptr1.link=NULL       /* 1 */
        e) return(ptr)
    3. end if
End   SLL_ Delete_ End
```



Before Deletion



After Deletion

1. Link part of last but one node is replaced with NULL. Because after deletion of last node in the list, last but one node become the last node.

iii) Deletion of a node from SLL at any position

- For deletion of a node from SLL at any position, a key value is specified.
- Where key being the data part of a node to be deleting.

Algorithm SLL_ Delete_ ANY (header,key)

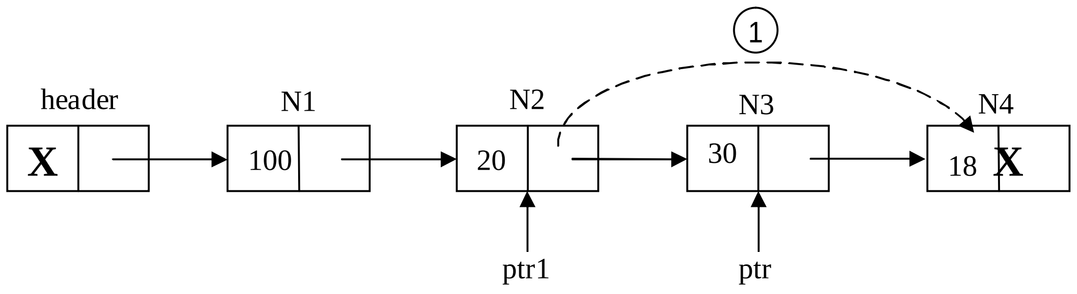
Input: header is a header node, key is the data part of the node to be delete.

Output: SLL with node deleted at Any position. i.e. Required element.

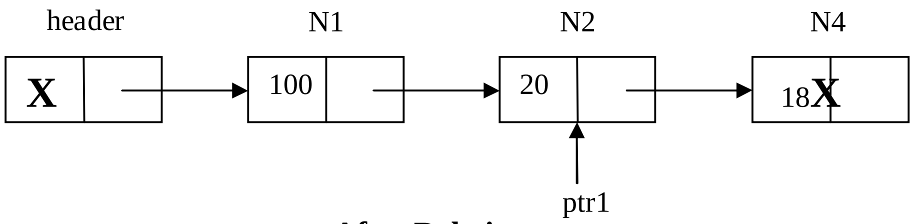
```
1. if(header.link = = NULL)
        a)print "SLL is empty, so unable to delete the node from list"
2. else                                   /*SLL is not empty*/
        a) ptr=header                     /*ptr  initially points to header node*/
        b) while(ptr.link!=NULL and ptr.data!=key)
                i) ptr1 = ptr
                ii) ptr=ptr.link   go to step b
        c) end loop
        d) if(ptr.link = = NULL and ptr.data!=key)
                i) print "Required node with data part as key value is not available"
        e) else                           /* node with data part as key value available */
                i) ptr1.link = ptr.link   /* 1 */
                ii) return(ptr)
        f) end if
3. end if
```

End SLL_ Delete_ ANY

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.



Before Deletion



After Deletion

4. Search for a node in SLL:

- For searching a node in SLL, a key value is specified by the user.
- If any node's data part is equal to key value, then search is successful.
- Otherwise the required node is not available or unsuccessful search.

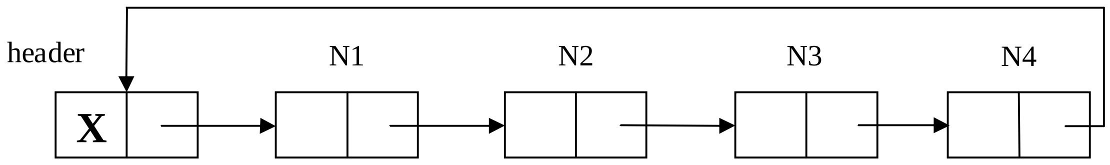
Algorithm SLL_Search(header,key)

Input: header is a header node
Output: Location is pointer to a node with data part as key value.

```
1. ptr = header
2. flag = 0
3. Location = NULL
4. while(ptr.link != NULL)
        a) ptr = ptr.link
        b)if(ptr.data==key)
          i)  flag=1
          ii) Location=ptr
          iii) break
c)end if
5. end loop
6. if (ptr.data = = key)
        a) flag = 1
        b) Location = ptr
        c) return (Location)
    7. else
        a) print "required node was not available"
    8. end if
End SLL_Search
```

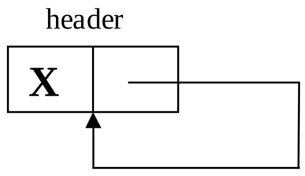
2. Circular Linked List :

- Circular linked list is a special type of linked list.
- In a Circular Linked list, the field of the last node points to the first node of the list.
- It is mainly used in lists that allow to access to nodes in the middle of the list without starting at the
  beginning.



Circular Linked List

- If a CLL is empty, then the link part of the header node points to itself.



Empty Circular Linked List

➢ **Operations on Circular Linked List:**

1. Insertion of a node in to CLL
2. Deletion of a node from CLL

1. Insertion of a node in to CLL

The Insertion of a node in to CLL can be done in various positions.

i) Insertion of a node into CLL at beginning.

ii) Insertion of a node into CLL at ending.

iii) Insertion of a node into CLL at any position.

i) Insertion of a node into CLL at beginning.

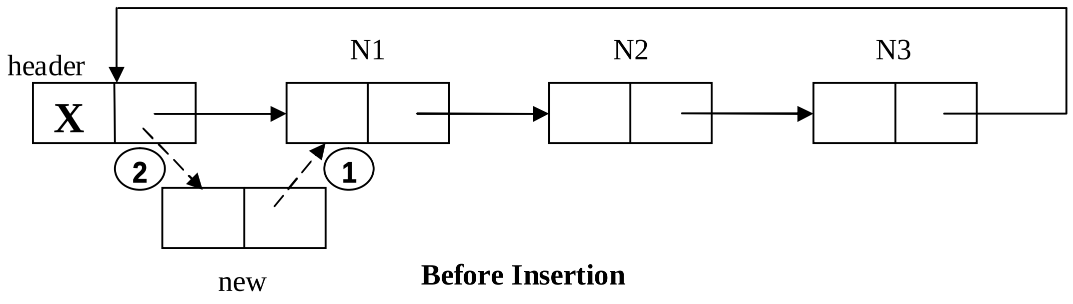
AlgorithmCLL_Insert_Begin(header,x)

Input: header is a header node, x is data part of new node to be insert.

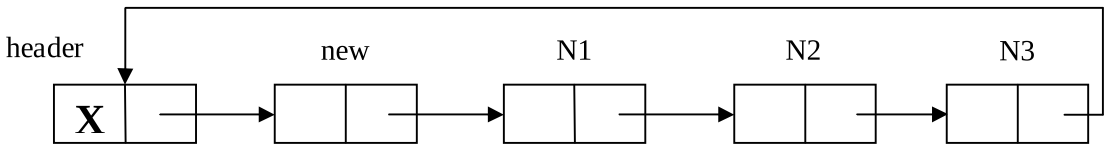
Output: CLL with new node inserted at beginning.

```
1. new=getnewnode( )
2. if(new = = NULL)
      a) print "required node was not available in memory, so unable to process"
3. else
      a) new.link = header.link                /* 1 */
      b) header.link = new                     /* 2 */
   c )new.data = x
4. end if
```

End CLL_Inset_Begin



Before Insertion



After Insertion

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address.

ii) Insertion of a node into CLL at ending.

AlgorithmCLL_Insert_END(header,x)

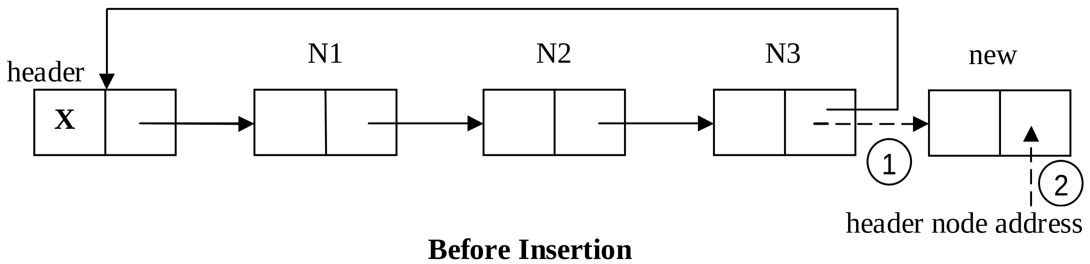
Input: header is a header node, x is data part of new node to be insert.

Output: CLL with new node inserted at ending.

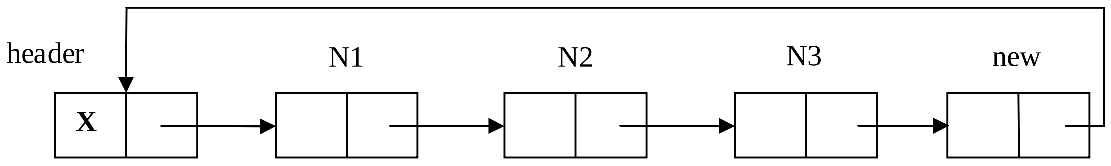
```
1. new=getnewnode()
```

```
2 .if(new = = NULL)
        a) print “required node was not available at memory bank, so unable to process”
3. else
        a) ptr=header
        b) while(ptr.link != header)
                i) ptr=ptr.link      //goto step b
        c) end loop
        d) ptr.link=new
        e) new.link=header
        f) new.data=x
4. end if
```

End CLL_Insertion_END



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with address of header node, because new node becomes the last node.

iii) Insertion of a node into CLL at any position.

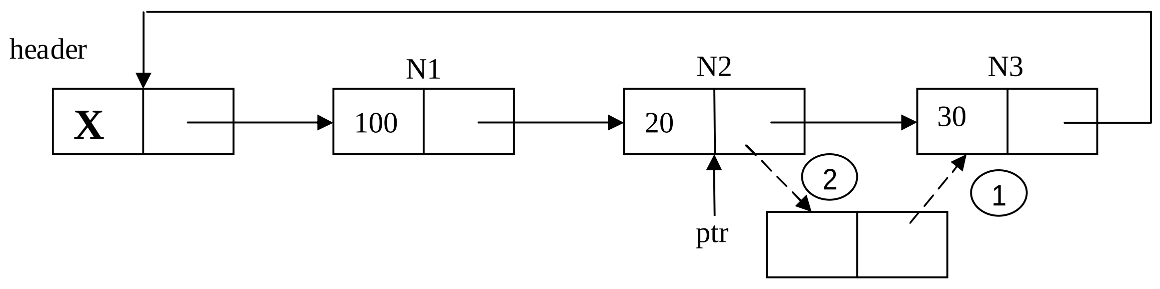
Algorithm CLL_Insert_ANY(header,x,key)

Input: header is header node, x is data pat of new node to be insert, key is the data part of a node, after this node we want to insert new node.

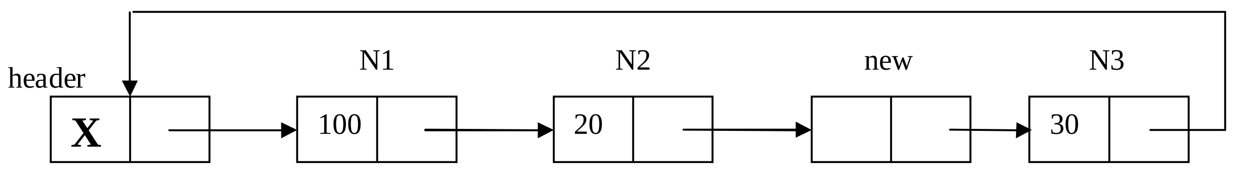
Output: CLL with new node at ANY

```
1. new=Getnewnode()
2 .if(new=NULL)
```

```
            a. print “required node was not available in memory bank, so unable to process”
    3. else
            i. ptr=header
            ii. while(ptr.data! =key and ptr.link!=header)
                    a) ptr=ptr.link
            iii. end loop
            iv. if(ptr.link=header and ptr.data!=key)
                    a) print “required node with data part as key value is not available, so unable to process”
            v. else
                    a) new.link=ptr.link
                    b) ptr.link=new
                    c) new.data=x
            vi. end if
    4. end if.
End CLL_insert_ANY
```



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

2. Deletion of a node from CLL:

The Deletion of a node from CLL can be done in various positions.

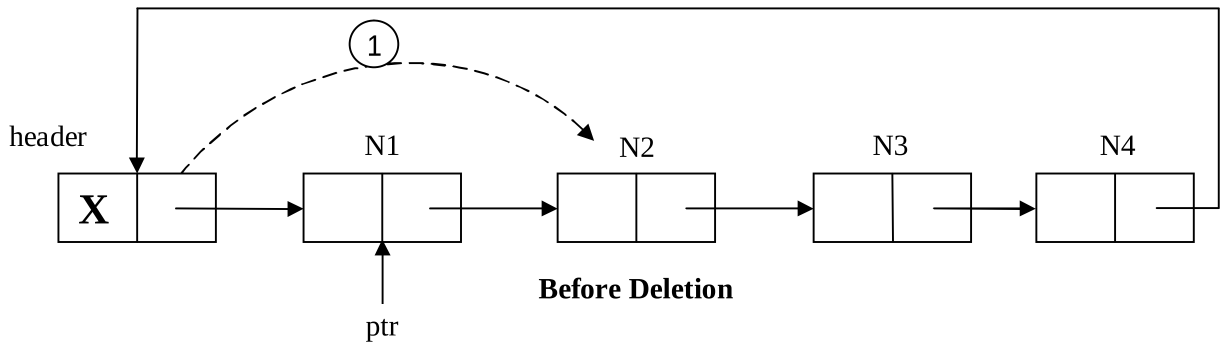
i) Deletion of a node from CLL at beginning.

ii) Deletion of a node from CLL at ending.

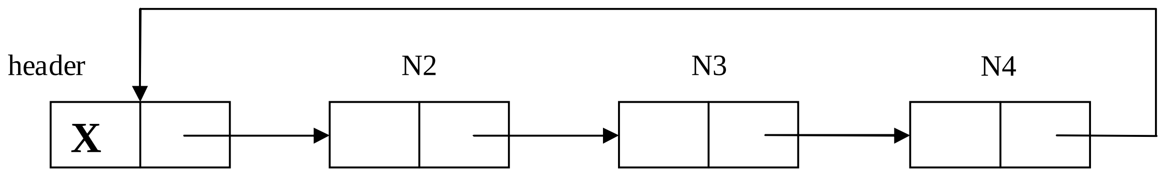
iii) Deletion of a node from CLL at any position

<u>i) Deletion of a node from CLL at beginning</u>

```
Algorithm CLL_Delete_Begin(header)
Input: Header is a header node.
Output: CLL with node deleted at Beginning.
    1. if(header.link = = header)
        a) print "CLL is empty, so unable to delete node from list"
    2. else                         /*DLL is not empty*/
        i. ptr=header.link                      /* ptr points to first node into list*/
        ii. header.link=ptr.link    /* 1 */
        iii. return(ptr)            /*send back deleted node to memory bank*/
    3. end if
End CLL_Delete_Begin
```



Before Deletion



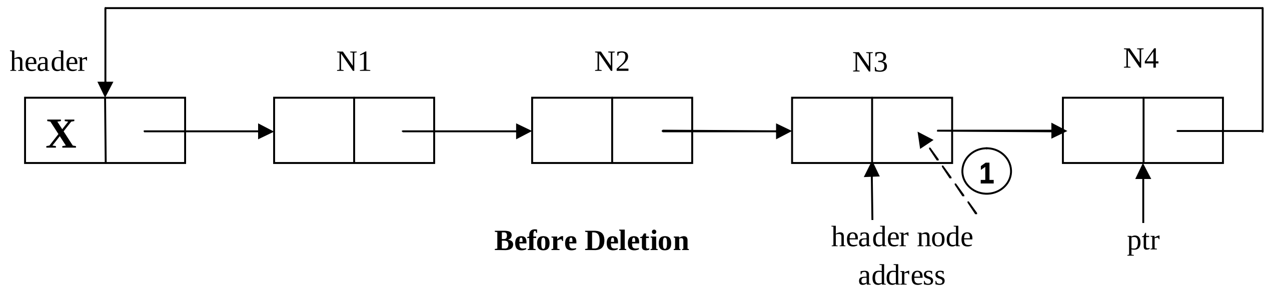
After Deletion

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

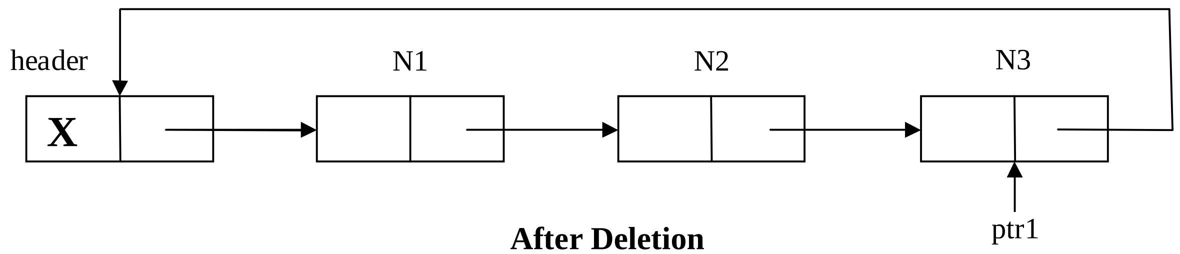
ii) Deletion of a node from CLL at ending

```
Algorithm   CLL_ Delete_ End (header)
Input:  header is a header node
Output: CLL with node deleted at ending.
    1. if(header.link = = header)
        a)print "CLL is empty, so unable to delete the node from list"
    2. else                                  /*CLL is not empty*/
        a) ptr=header  /*ptr  initially points to header node*/
        b) while(ptr.link!=header)
            i) ptr1=ptr
            ii) ptr=ptr.link   /*go to step b*/
        c) end loop
        d) ptr1.link=header                  /* 1 */
        e) return(ptr)
    3. end if
End   CLL_ Delete_ End
```

1. Link part of last but one node is replaced with address of header node. Because after deletion of last node in the list, last but one node become the last node.



Before Deletion



After Deletion

iii) Deletion of a node from CLL at any position

- For deletion of a node from CLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

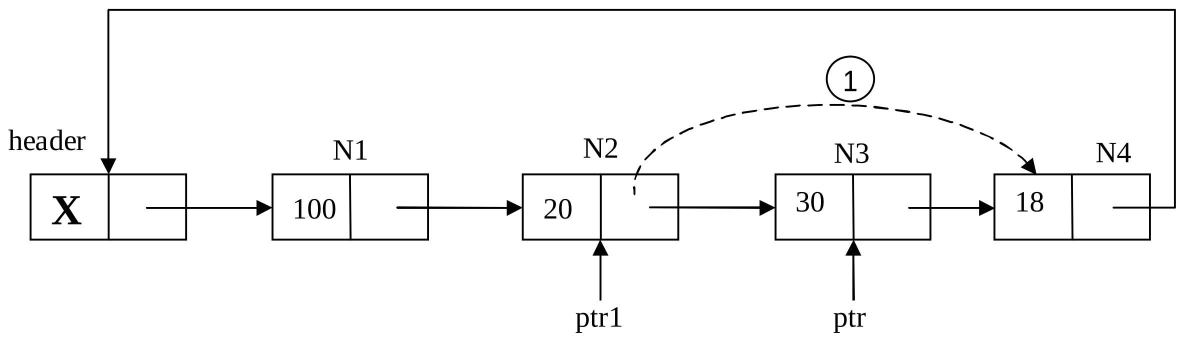
Algorithm CLL_ Delete_ ANY (header,key)

Input: header is a header node, key is the data part of the node to be delete.

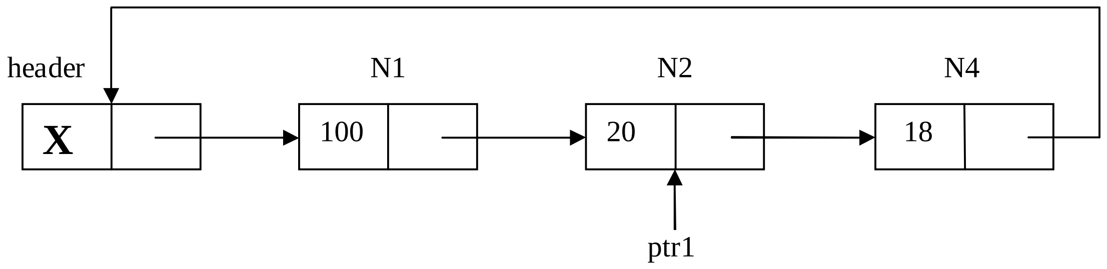
Output: CLL with node deleted at Any position. i.e. Required element.

```
1. if(header.link = = header)
        a)print “SLL is empty, so unable to delete the node from list”
2. else                                      /*CLL is not empty*/
        a) ptr=header  /*ptr  initially points to header node*/
        b) while(ptr.link!=header and ptr.data!=key)
                i) ptr1 = ptr
                ii) ptr=ptr.link   go to step b
        c) end loop
        d) if(ptr.link = = header and ptr.data!=key)
                i) print “Required node with data part as key value is not available”
        e) else                              /*node with data part as key value available
                i) ptr1.link = ptr.link               /* 1 */
                ii) return(ptr)
        f) end if
3. end if
```

End CLL_ Delete_ ANY



Before Deletion

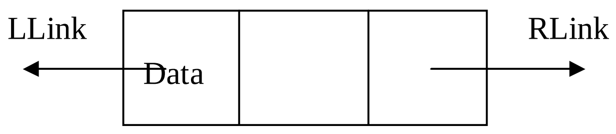


After Deletion

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.

3. Double Linked List:

- In a SLL one can move from the header node t o any node in one direction only. i.e. from left to right.
- A DLL is a two way list. Because one can move either from left to right or right to left.
- In DLL, each node maintains two links.



Structure of a node in DLL

- Here**LLink** refers Left Link and **RLink** refers Right Link.
- The LLink part of a node in DLL always points to the previous node. i.e. LLink part of a node Consists address of previous node.

- The RLink part of a node in DLL always points to the next node. i.e. RLink part of a node Consists address of next node.

➢ <u>Operations on Double Linked List:</u>

1. Insertion of a node in to DLL
2. Deletion of a node from DLL

1. Insertion of a node in to DLL:

The Insertion of a node in to DLL can be done in various positions.

i) Insertion of a node into DLL at beginning.

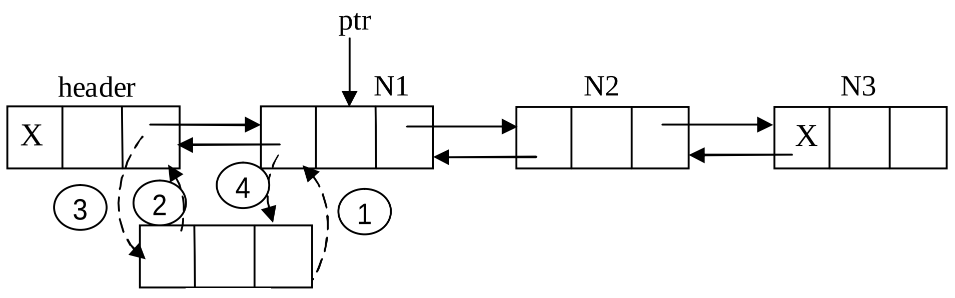
ii) Insertion of a node into DLL at ending.

iii) Insertion of a node into DLL at any position.

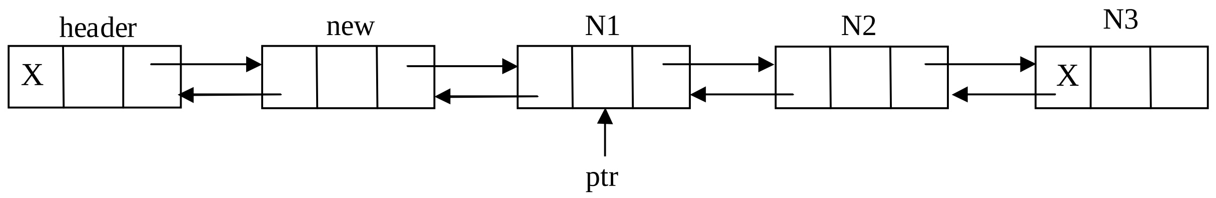
- For insertion of a node into DLL, we must get node from memory bank. The procedure for getting node from memory bank is same as getting node for SLL from memory bank.

<u>i) Insertion of a node into DLL at beginning</u>

```
Algorithm DLL_insertion_Begin(header,X)
Input: header is a header node.
Output: DLL with new node at begin.
    1. new=getnewnode()
    2. if(new = = NULL)
        a) print "required node is not available in memory"
    3. else
        a) ptr=header.rlink
        b) new.rlink=ptr              /* 1 */
        c) new.llink=header           /* 2 */
        d) header.rlink=new           /* 3 */
        e) ptr.llink=new              /* 4 */
    4. end if
End DLL_insertion_Begin
```



Before Insertion



After Insertion

1. Rlink part of new node is replaced with the address of first node in the DLL. i.e. address of first node is available in Rlink part of header node.
2. Llink part of new node is replaced with the address of header.
3. Rlink part of header node is replaced with the address of new node.
4. Llink part of previous first node is replaced with the address of new node.

ii) Insertion of a node into DLL at ending.

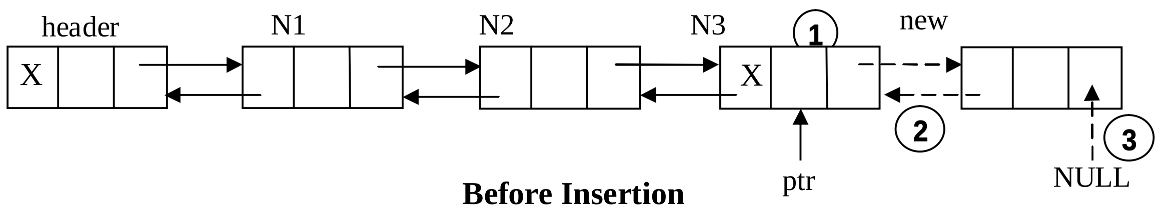
Algorithm DLL_Insert_Ending(Header,x)

Input: Header is the header node, x is the data part of new node to be inserted.

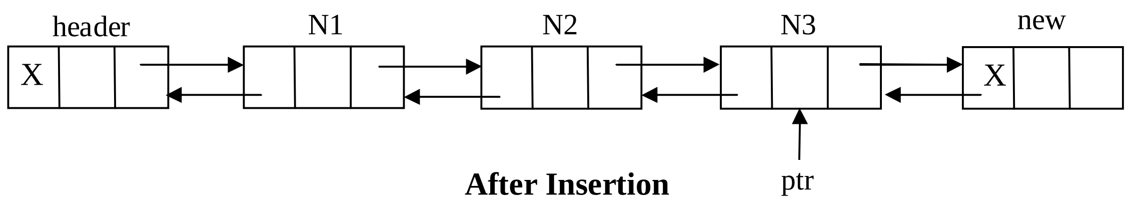
Output: DLL with new node inserted at the ending.

```
1. new=getnewnode()
2. if(new = = NULL)
        a)print "Required node was not available"
3. else
        a) ptr=header
        b) while(ptr.rlink != NULL)
                i) ptr=ptr.rlink          goto step(b)
        c) end while loop
        d) ptr.rlink=new                  /* 1 */
        e) new.llink=ptr                  /* 2 */
        f) new.rlink=NULL                 /* 3 */
        g) new.data=x
    4. end if
```

End DLL_Insertion_Ending



Before Insertion



After Insertion

1. RLink part of last node in the DLL is replaced with address of new node.
2. LLink part of new node is replaced with address of previous last node.
3. RLink part of new node is replaced with NULL. Because newly inserted node becomes the last node in the list.

<u>iii) Insertion of a node into DLL at any position</u>

- For insertion of a node at any position in DLL, a key value is specified.
- Where key being the data part of a node, after this node new node has to be inserting.

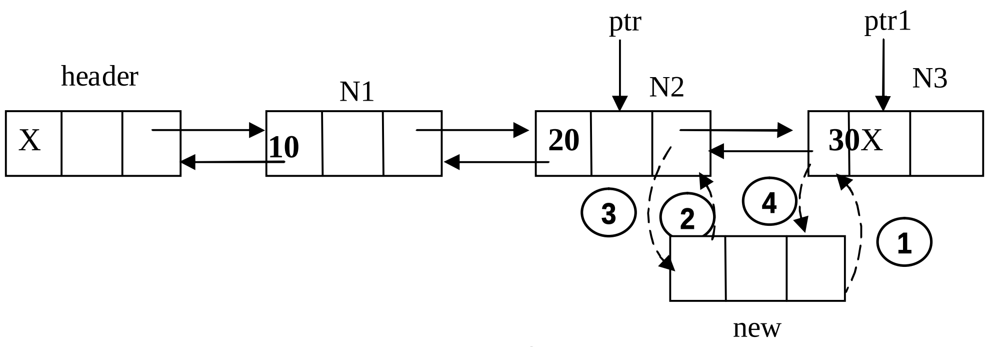
Algorithm DLL_Insertion_ANY(header,x,key)

Input: Header is a header node, key is the data part of a node, after that node new node is inserted, x is data part of new node to be insert.

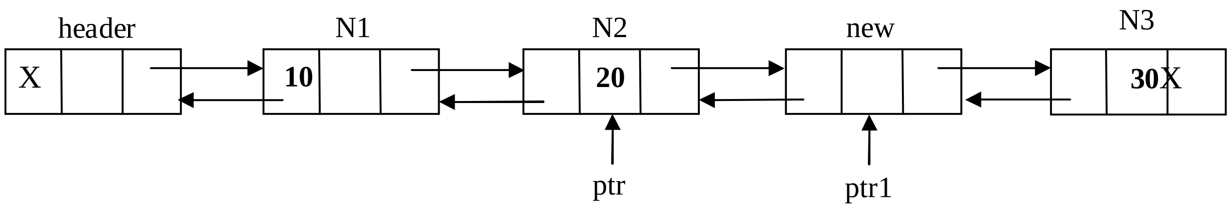
Output: DLL with new node inserted after the node with data part as key value

```
1. new=getnewnode()
2. if(new = = NULL)
        a) print“required node is not available in memory”
3. else
        a) ptr=header
        b) while(ptr.data!=key and ptr.rlink!=NULL)
                i) ptr=ptr.rlink     go to step(b)
        c) end loop
        d) if(ptr.rlink = = NULL and ptr.data != key)
                i) print “required node with key value was not available”
        e) else
                i) ptr1=ptr.rlink
                ii) new.rlink=ptr1              /* 1 */
                iii) new.llink=ptr              /* 2 */
                iv) ptr.rlink=new               /* 3 */
                v) ptr1.llink=new               /* 4 */
                vi)new.data=x
        f) end if
    4. end if
```

EndDLL_Insertion_ANY



Before Insertion



After Insertion

1. RLink part of new node is replaced with the address of next node. i.e. in the above example N3 becomes the next node for newly inserting node.
2. LLink part of new node is replaced with the address of previous node. i.e. in the above example N2 becomes the previous node for newly inserting node.
3. RLink part of previous node is replaced with address of new node.
4. LLink part of next node is replaced with address of new node

2. Deletion of a node from DLL

The deletion of a node in from DLL can be done in various positions.

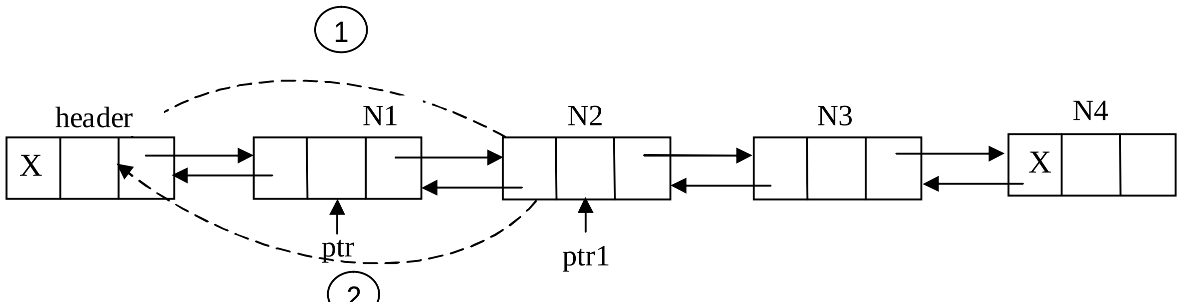
i) Deletion of a node from DLL at beginning.

ii) Deletion of a node from DLL at ending.

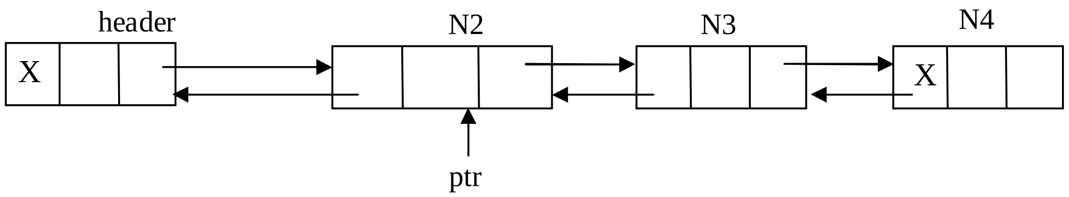
iii) Deletion of a node from DLL at any position.

i) Deletion of a node from DLL at beginning

```
Algorithm DLL_Deletion_Begin(header)
Input: header is a header node
Output:  DLL with node deleted at begin
    1. if(header.rlink = = NULL)
        a)  Print "DLL is empty, not possible to perform deletion operation"
    2. else
        a)  ptr=header.rlink
        b)  ptr1=ptr.rlink
        c)  header.rlink=ptr1           /* 1 */
        d)  ptr1.llink=header           /* 2 */
        e)  return(ptr)
    3. End if
End DLL_Deletion_Begin
```



Before Deletion



After Deletion

1. RLink part of header node is replaced with the address of second node. i.e. address of second node is available RLink part of first node.
2. LLink part of second node is replaced with the address of header node.

<u>ii) Deletion of a node from DLL at ending.</u>

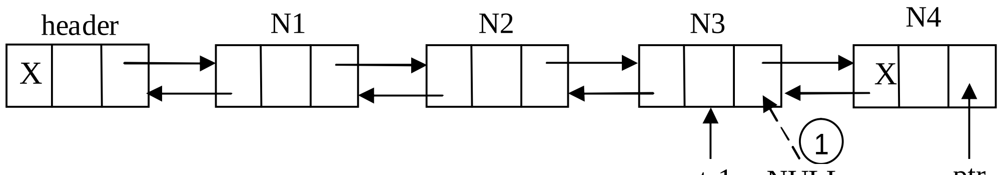
Algorithm DLL_Deletion_End(header)

Input: header is a header node.

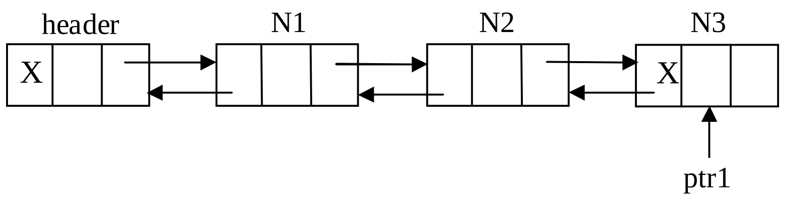
Output: DLL with deleted node at ending.

```
1.  if(header.rlink=NULL)
        a)  Print "DLL is empty, not possible to perform deletion operation"
2. else
        a) ptr=header
        b) while(ptr.rlink != NULL)
                i) ptr1=ptr
                ii) ptr=ptr.rlink
        c) end loop
        d) ptr1.rlink=NULL                    /* 1 */
        e) return(ptr)
3. end if
```

End DLL_Deletioon_Ending



Before Deletion



AfterDeletion

1. RLink part of last but one node in DLL is replaced with NULL. Because last but one node becomes last node.

<u>iii) Deletion of a node from DLL at any position.</u>

- For deletion of a node from DLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

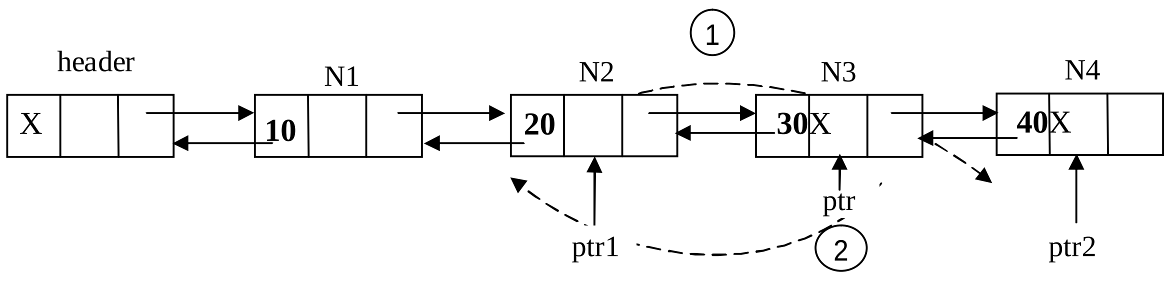
Algorithm DLL_Deletion_Any(header,key)

Input: header is header node, key is the data part of a node to be delete.

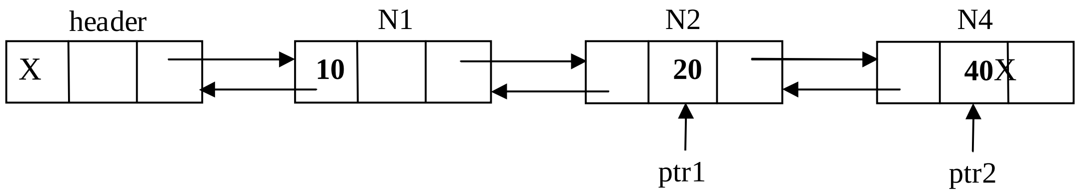
Output: DLL without node as data part is key value.

```
1. if(header.rlink = = NULL)
        a) print "DLL is empty, not possible for deletion operation"
2. else
        i) ptr=header
        ii) while(ptr.data!=key and ptr.rlink!=NULL)
                a) ptr=ptr.rlink
        iii) end loop
        iv) if(ptr.rlink=NULL and ptr.data != key)
                a) print "required node was not available in list"
        v) else
                a) ptr1 = ptr.llink
                b) ptr2 = ptr.rlink
                c) ptr1.rlink = ptr2            /* 1 */
                d) ptr2.rlink = ptr1            /* 2 */
        vi) end if
3. end if
```

End DLL_Deletion_Any



Before Deletion



After Deletion

1. RLink part of previous node is replaced with the address of next node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.
2. LLink part of next node is replaced with the address of previous node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.

UNIT-II
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. The logical or mathematical model of a particular organization of data is defined as ______________.

2. An ordered collection of finite, homogeneous data elements where the linear order is maintained by means of links or pointers is defined as ___________________.

3. In single linked list each node contain minimum of two fields. One field is data field to store the data and select for what purpose the second field is used to store _____________? [ ]

a) Pointer to character b) Pointer to integer c) Pointer to next node d) None

4. Identify the memory allocation process in Linked list [ ]

a)Dynamic b)Compile Time c)Static d)None of these

5. A variant of linked list, identify in which last node of the list points to the first node of the list is? [ ]

a)Singly linked list b) Doubly linked list c)Circular linked list d) Multiply linked list

6. In doubly linked lists, identify which type of traversal can be performed? [ ]

a)Only in forward direction b) Only in reverse direction c)In both directions d) None

7. A variant of the linked list, identify in which none of the node contains NULL pointer is? [ ]

a)Singly linked list b) Doubly linked list c)Circular linked list d) None

8. Identify non-linear Data Structure from the following [ ]

a. Array b. Stack c. Graph d. Linked list

9.A node in single linked list can reference the previous node. [True/False]

10. Choose, Which type of structure is used to create a linked list? [ ]

a) Nested structure b) Self referential structure c) Array of structure d)pointers to structure

11. Predict, Which type of linked list occupies more memory? [ ]

a)SLL b) DLL c)CLL d)None

12. Compute how many pointers need to modify in inserting a node at the beginning of the single linked list

a) 1 b) 2 c) 3 d) 0 [ ]

13. What does the following function do for a given Linked List with first node as head? [ ]

```
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

a) Prints all nodes of linked lists

b) Prints all nodes of linked list in reverse order

c) Prints alternate nodes of Linked List

d) Prints alternate nodes in reverse order

14. Deleting a node at any position (middle) of the single linked list needs to modify __________pointers.

a) 1 b) 2 c) 3 d) 0 [ ]

15. A double linked list is declared as follows: [ ]

```
struct dllist
{
struct dllist *fwd, *bwd;
int data;
}
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list. Which among the following segments of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to neither the first nor last element of the list?

a. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> bwd = X -> bwd

b. X -> bwd -> fwd = X -> bwd;
   X -> fwd -> bwd = X -> fwd

c. X -> bwd -> bwd = X -> fwd;
   X -> fwd -> fwd = X -> bwd

d. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd

15. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the double linked list. The start pointer points to beginning of the list? [ ]

a. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd;

b. X -> fwd = start;
   start -> bwd = X;
   start = X;

```
c. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd;
   start = X;
d. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd
```

16. Does C perform array out of bound checking? What is the output of the following program? [ ]

```
int main()
{
    int i;
    int arr[5] = {0};
    for (i = 0; i <= 5; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

a) Compiler Error: Array index out of bound.
b) The always prints 0 five times followed by garbage value
c) The program always crashes.
d) The program may print 0 five times followed by garbage value, or may crash if address (arr+5) is invalid.

17. Which boolean expression indicates whether the numbers in two nodes (p and q) are the same. Assume that neither p nor q is null. [ ]

a) p == q
b) p.data == q.data
c) p.link == q.link
d) None of the above.

18. Which of the following statement is true [ ]

I. Using single linked list it is not possible to traverse the list in backward direction.

II. To find the predecessor it is required to traverse the list from the first node in case of single linked list.

a) I only b) II only c) Both I and II d) None of the above

19. Suppose each set is represented as a linked list with elements in arbitrary order. Which of the operations among union, intersection, membership, cardinality will be the slowest? [ ]

a) union only
b) intersection, membership
c) membership, cardinality
d) union, intersection

20. The following C function takes a singly linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank.

```
Typedefstruct node
    {
     Intvalue;
     Structnode *next;
}Node;
Node *move_to_front(Node *head)
{
       Node *p, *q;
       If ((head == NULL: || (head->next == NULL))
                   Return head;
        Q = NULL;
         p = head;
       While (p->next !=NULL)
      {
          Q = p;
          P = p->next;
         }
           ----------------------------------------
     Return head;
}
```

Choose the correct alternative to replace the blank line. **[ ]**

a) q = NULL; p->next = head; head = p;

b) q->next = NULL; head = p; p->next = head;

c) head = p; p->next = q; q->next = NULL;

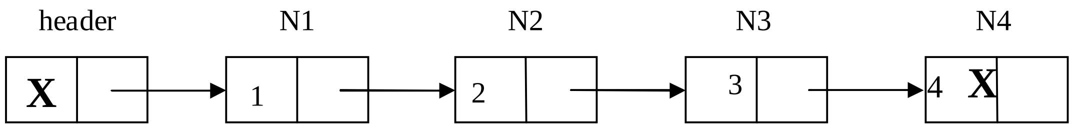
d) q->next = NULL; p->next = head; head = p;

SECTION-B

SUBJECTIVE QUESTIONS

1. Explain about delete operation in singly linked list.
2. Compare single linked list and circular single linked list.
3. Write an algorithm to perform deletion operation on circular linked list.
4. Write an algorithm to perform insertion operation on a double linked list.
5. Write an algorithm to perform deletion operation on a double linked list.
6. Write short notes on data structures.

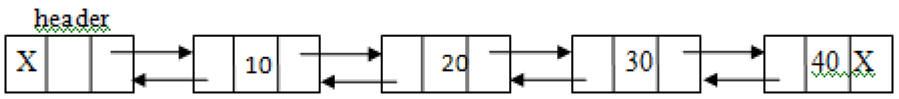
7. **Consider the following single linked list.**



Demonstrate the following operations on this list and draw the updated single linked listafter each operation.

1. Insert 5 at end	2. Insert 6 at begin	3.Insert 9 after 2
4. Delete 6	5. Delete 5	6. Delete 3

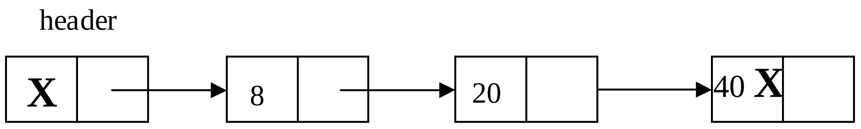
8. Consider the following double linked list.



Illustrate the following operations on this list and draw the updated single linked list after each operation.

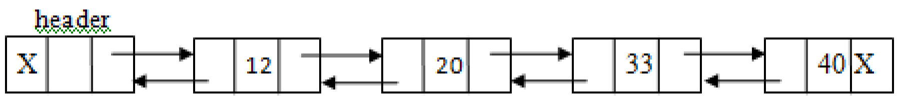
1. Insert 50 at end	2. Insert 60 at begin	3.Insert 90 after 20
4. Delete 60	5. Delete 50	6. Delete 30

9. Consider the following single linked list.



Insert the following elements into the list **2,15,30,50**. Such that the list will be in **ascending**order and draw the updated single linked list after each insertion operation.

10. Consider the following double linked list.



Insert the following elements into the list **2,15,30,50**. Such that the list will be in **ascending**order and draw the updated single linked list after each insertion operation.

11. Write a program to implement insert operation in a doubly linked List.

12. Write a program to perform deletion operation in the middle of a doubly linked list.

13. Develop a program to delete an element of a single linked list.

14. Develop a program to merge two single linked lists into one list so that the resultant list will be inascending order.

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. Consider the function f defined below. **(GATE 2003)**

```
    struct item
    {
      int data;
      struct item * next;
    };
    int f(struct item *p)
    {
  return( (p == NULL) || (p->next == NULL) || (( P->data <= p->next->data) && f(p->next)) );
    }
```

For a given linked list p, the function f returns 1 if and only if

a) the list is empty or has exactly one element

b) the elements in the list are sorted in non-decreasing order of data value

c) the elements in the list are sorted in non-increasing order of data value

d) not all elements in the list have the same data value.

2. A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enQueue and deQueue can be performed in constant time? **(GATE 2004)**

a) rear node b) front node c)not possible with a single pointer d) node next to front

3. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is (GATE 2002)

a) log 2 n b) n/2 c) log 2 n – 1 d) n