4 Linked List

4.1 BASICS OF LINKED LIST

Introduction:

- A brief about why linked list came into the picture:

Basically, there are two reasons behind taking the linked list into the picture.

1) Insertion and deletion is easier:

In Linked List, insertion and deletion take O(1).

2) Size:

When a conventional array is used, the size of the array is fixed once declared. We cannot increase or decrease the array size based on the elements inserted or deleted.

- Linked list is a data structure which is in the form of list of nodes, as shown in the diagram below:

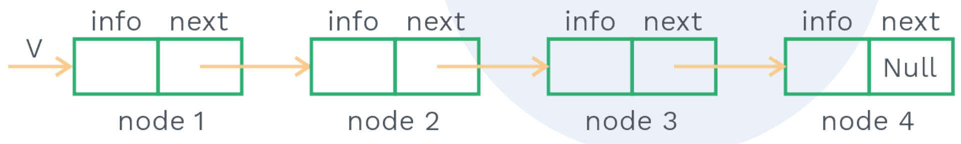


Fig. 4.1

- A node contains two fields:
  1) An information field (Head)
  2) Next address field (Tail)
- Information field contains actual data, and the next address field contains the address of the next node of the list.
- An external pointer 'V', is a pointer variable which contains an address for the first node of the list, and points to the first node.
- We can access the entire list with this external pointer 'V'.
- Linked list is a flexible data structure as we can insert any number of nodes and delete nodes from the given linked list.
- Next field of last node of linked list contains null, thus null pointer is used to signal the end of a list.

Note:

The list with no node is called the empty list or the null list.

Note:

The value of external pointer 'V' to an empty list is the null pointer.

4.2 OPERATIONS ON LINKED LIST

Insertion operation:

1) Insertion of a node to the front of a linked list:

Consider a Linked List:

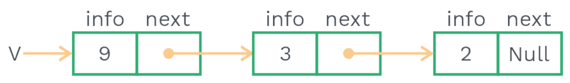


Fig. 4.2

Now we need to insert a node having info as '4' to the front of the above list.

Steps to insert:

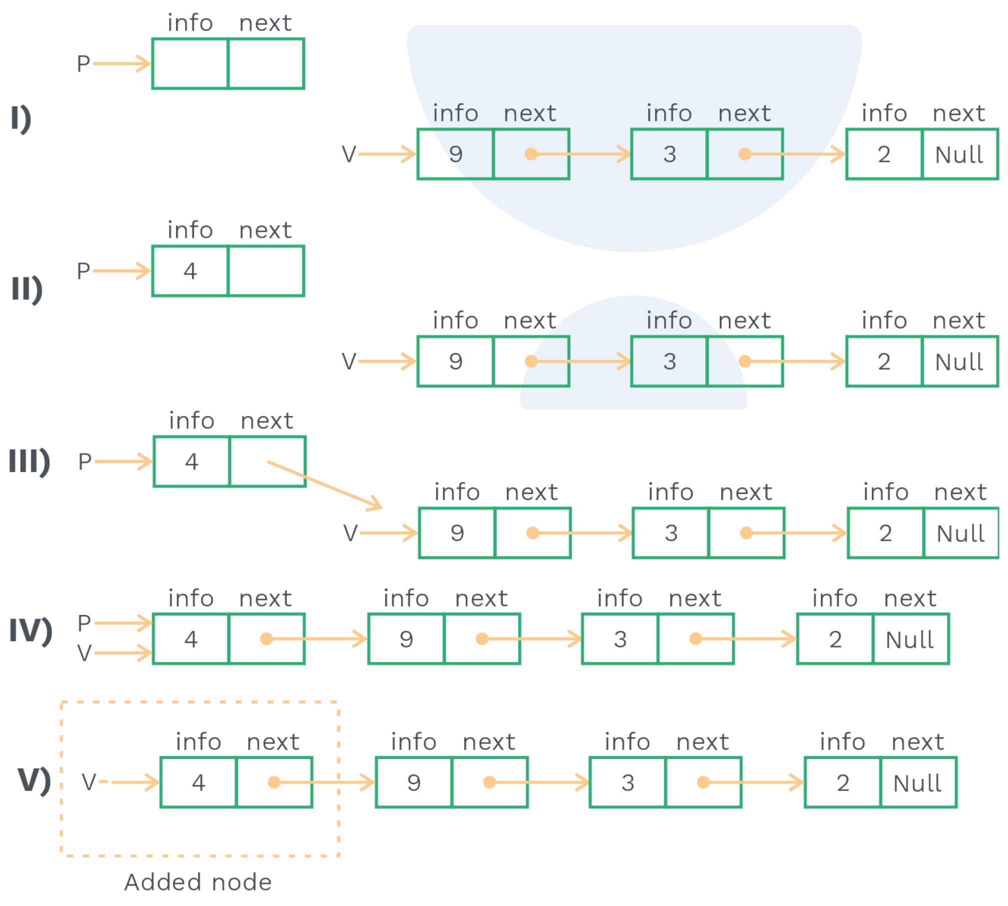
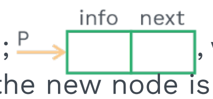


Fig. 4.3

I) We create an empty node ; , which is pointed by a variable P, means the address of the new node is set to variable P.

II) Integer '4' is inserted into the info field of the newly allocated node.
III) Next field of the new node is updated with the address of the first node of the given list i.e. V.
IV) P–value is assigned to V–value so that V is pointing now to the newly attached node.
V) We get the final list after inserting a node with an info field having integer '4'.

* **Algorithm for insertion of a node to the front of a linked list:**

```
1) P = getnode( ) ;
2) info(P) = 4 ;
3) next(P) = V ;
4) V = P ;
5) return (V) ;
```

Meaning:

1) P = getnode( ) ;	//	This operation obtains an empty node and sets the content of a variable–'P' to the address of that node.
2) info(P) = 4 ;	//	By this operation, integer–'4' is inserted into the info field of new node. info(P) implies the info field of a node pointed by P).
3) next(P) = V ;	//	next(P) implies next field of a node pointed by P, and the given statement of code implies node pointed by P be added to the front of the list.
4) V = P ;	//	the address contained in P is now assigned to the variable V ; hence previous address residing in V is replaced by the address residing in P.
5) return (V) ;	//	It will return the linked list, pointed by 'V'.

Note:

```
getnode( )
{
Struct node
{
  int info ;
  Struct node* next ;
   A ;
  return &A ;
  }
```

(We will see detailed code in the **section–4.4 Implementation of Linked List.)**

Generalised algorithm to add any object ‘x’ to the front of a linked list:

```
1) P = getnode( ) ;
2) info(P) = x ;
3) next(P) = V ;
4) V = P ;
5) return (V) ;
```

2) Insertion of a node to the end of a list:

Consider the previous list only:

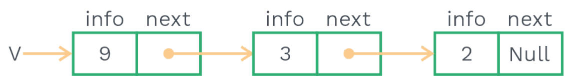


Fig. 3.4

Now, we have to insert a node having info as 4 to the end of the above list.

Steps to insert:

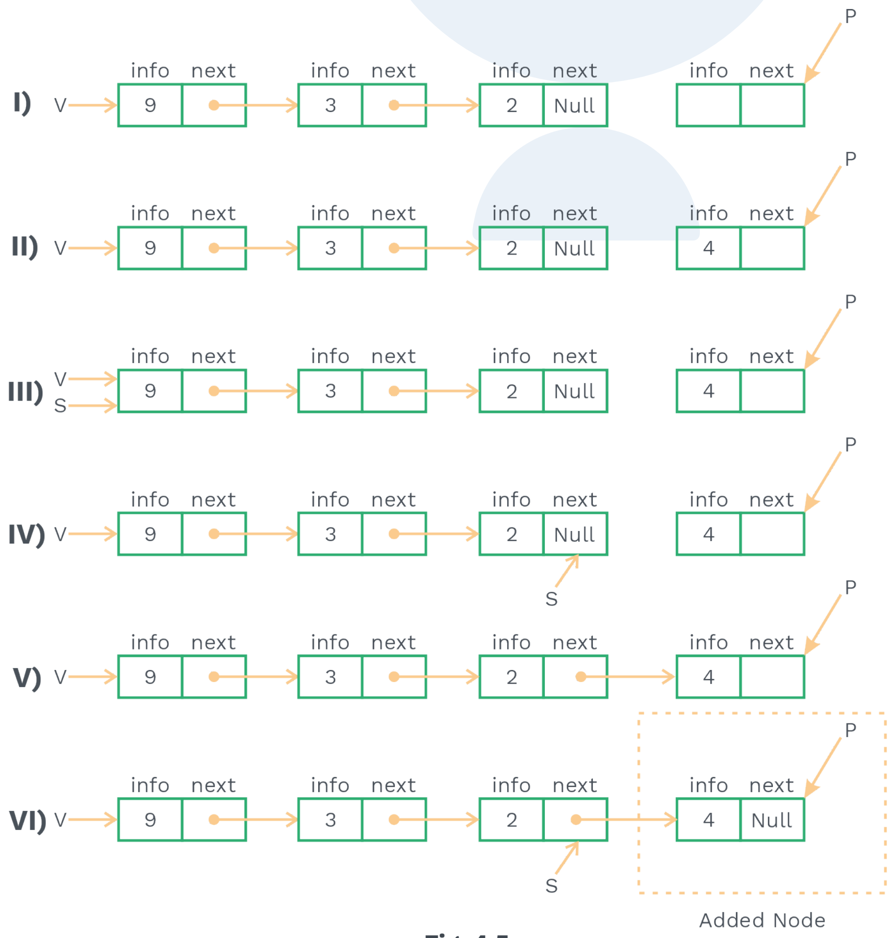


Fig. 4.5

I) An empty node is created, which is pointed by a variable ‘P’.
II) Integer ‘4’ is inserted into the info field of the newly allocated node.
III) The value of ‘V’ is assigned to a new pointer variable ‘S’.
IV) ‘S’ traverses the linked list till the end of the linked list.
V) Last node of the given linked list now points to the newly created node pointed by P.
VI) next(P) is set to ‘null’, which implies it is last node of the given linked list.

* **Algorithm for insertion of a node to the end of a given linked list:**

```
P = getnode( ) ;
info(P) = 4 ;
S = V ;
while (next(S)! = Null)
S = next(S) ;
next(S) = P ;
next(P) = Null ;
return (V) ;
```

Generalised algorithm to add any object ‘x’ to the end of a linked list:

```
P = getnode( ) ;
info(P) = 4 ;
S = V ;
while (next(S)! = Null)
S = next(S) ;
next(S) = P ;
next(P) = Null ;
return (V) ;
```

Time complexity:
1) Insertion at beginning = O(1).
2) Insertion at the end when we have only head pointer = O(n).
3) Insertion at the end when we have both head and tail pointer = O(1).

Deletion operation:

1) **Deletion of a node from the front of a linked list:**

Consider a linked list:

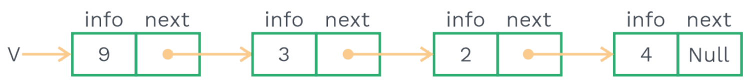


Fig. 4.6

We have to delete the first node of the given linked list.

Steps to delete:

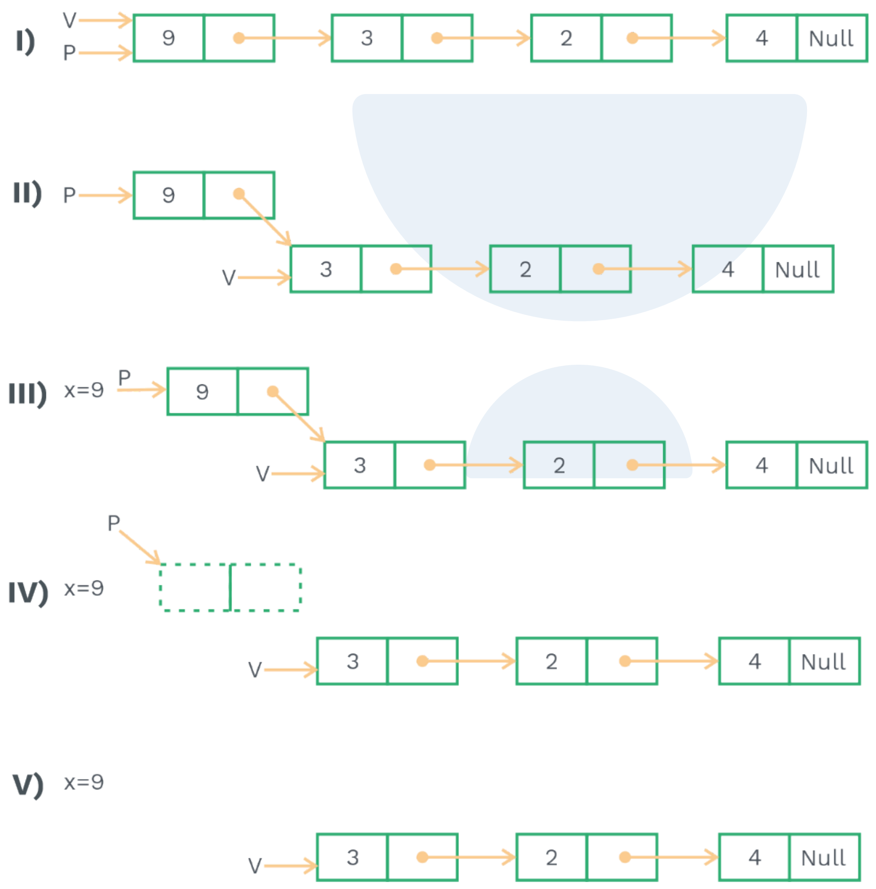


Fig. 4.7

I) The first node of the given linked list, which is pointed by ‘V’, is pointed by one more variable, ‘P’.

II) Next field of ‘P’ is now assigned to ‘V’.

III) Info field is copied to a variable ‘x’.

IV) Node pointed by ‘P’ is now free, which means deleted.

V) Final linked list after deleting the first node is returned.

* **Algorithm for deletion of a node from the front of a linked list:**

```
P = V ;
V = next (P) ;
x = info (P) ;
free node (P) ;
return (V) ;
```

Note:

freenode(P) ; means the node pointed by ‘P’ is now not allocated. The corresponding memory space is ready for reuse.

2) Deletion of a node from the end of a linked list:

Consider the previous linked list:

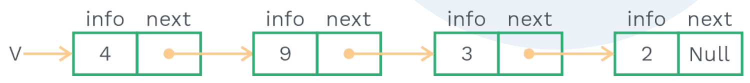


Fig. 4.8

We have to delete the last node from this linked list.

Steps to delete:

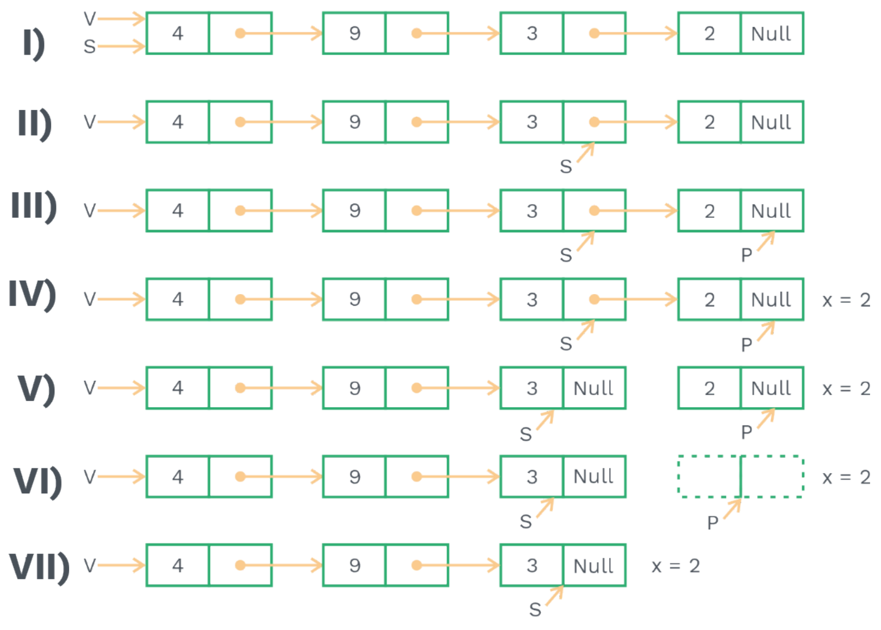


Fig. 4.9

I) First node of the given linked list, which is pointed by 'V', is pointed by one more variable, 'S'.
II) 'S' traverses the given linked list till the second last node, and points it.
III) One variable, 'P', points to last node of the linked list.
IV) info field of the node pointed by P is copied to a variable 'x'.
V) next(S) is set to null, to make it the last node.
VI) node pointed by P is deallocated.
VII) Final linked list after deleting the last node is returned.

* **Algorithm for deletion of a node from the end of a linked list:**

```
S = V ;
while (next(next(S)) ! = Null)
S = next(S) ;
P = next(S) ;
x = info(P) ;
next(S) = Null ;
freenode(P) ;
return(V) ;
```

Rack Your Brain

Write an algorithm to perform each of the following operations:
a) Append an element to the end of a list.
b) Concatenate two lists.
c) Delete the last element from a list.
d) Delete the nth element from a list.

Time complexity:
1) Deletion from beginning = O(1).
2) Deletion of given node = O(n).
3) Deletion of node at end = O(n).

Previous Years' Question

Q. Let P be a singly linked list, Let Q be the pointer to an intermediate node x in the list. What is the worst–case time complexity of the best known algorithm to delete the node x from the list?

a) O(n)	**b)** $O(\log^2 n)$
b) O(log n)	**d)** O(1)

Sol: d) **(GATE CSE: 2004)**

Previous Years' Question

Q. What is the worst case the time complexity of inserting n elements into an empty linked list, if the linked list needs to be maintained in sorted order?

a) $\Theta(n^2)$	**b)** $\Theta(n)$
c) $\Theta(n\log n)$	**d)** $\Theta(1)$

Sol: a) **(GATE CSE: 2020)**

Searching operation:

Linear search on a linked list:

Consider a linked list, pointed by V, we need to find the address of the node having value '4'.

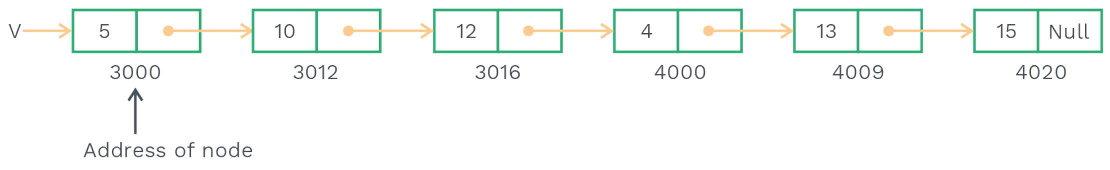


Fig. 4.10

Finding integer '4':

I) Start traversing nodes one by one.

II) Compare the info field of nodes with integer '4'.

III) Return the address of the node containing integer '4'.

Time complexity:

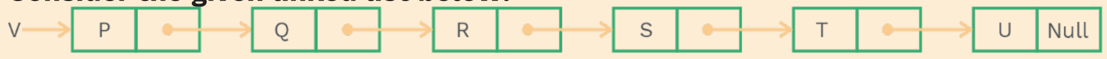
Linear search on a single linked list will take O(n) [worst case] because at the worst case, we need to traverse the whole linked list with unsorted elements of it, to find the desired element.

Note:

Binary search on linked list is possible but not efficient.

PRACTICE QUESTIONS

Q1 **Consider the given linked list below:**



Find the result of the code given below:

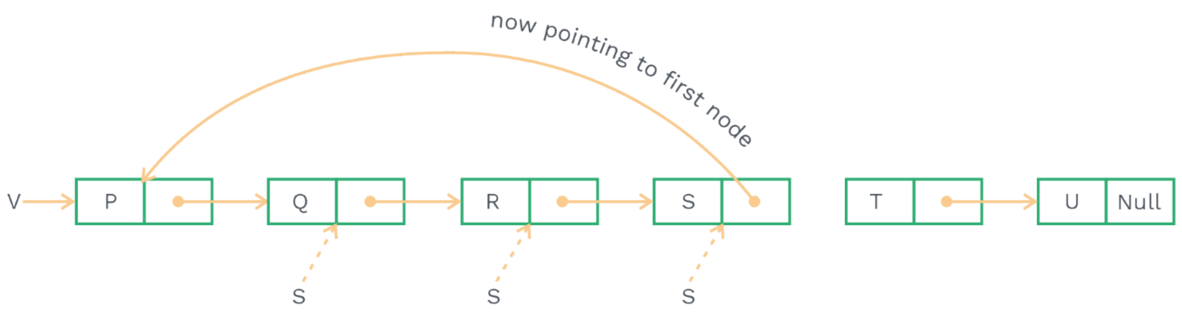
Struct node * S ;

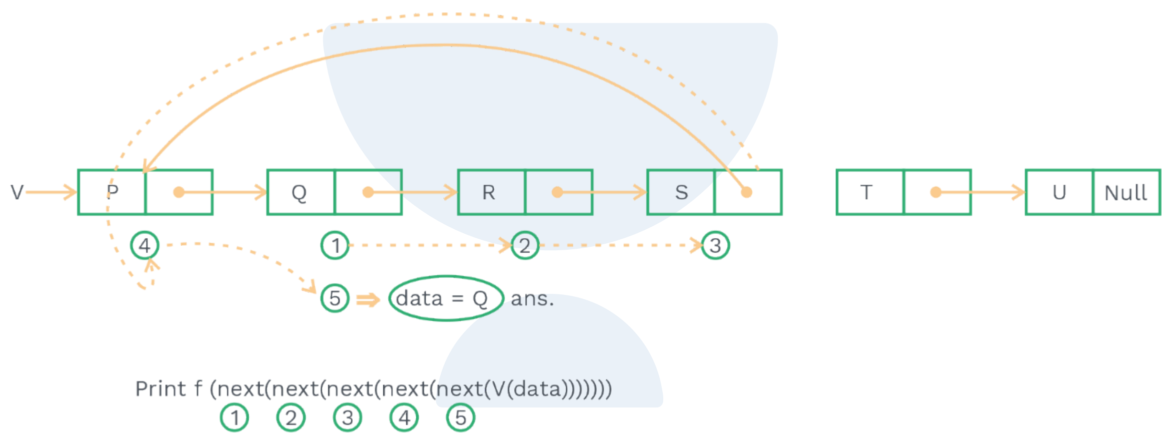
S = next(next(next(V)))

next(S) = V

printf(next(next(next(next(next(V)))))(data))

Sol:

I)


II)


Previous Years' Question

Q. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is

a) $\log_2 n$ **b)** n/2
c) $\log_2 n-1$ **d)** n

Sol: d) **(GATE CSE: 2002)**

Previous Years' Question

Q. Linked lists are not suitable data structures of which one of the following problems?

a) Insertion sort
b) Binary search
c) Radix sort
d) Polynomial manipulation

Sol: b) **(GATE CSE: 1994)**

4.3 TYPES OF LINKED LIST

Singly linked list:

- Singly linked list is a type of linked list in which there is one next field which points to the next node of the linked list.
- Representation of singly linked list.

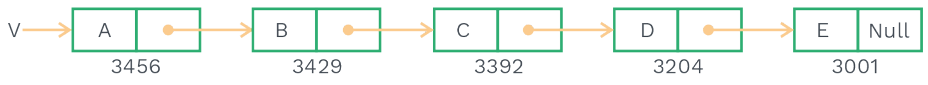


Fig. 4.11

- Each node contains two fields
  1) Information field: Contains objects
  2) Next Field: Contains the address of the next node means points to the next node.
- Last node next field contains null, which implies that it is the last node.
- It is flexible in size means, we can insert any number of nodes to the existing linked list, and also can delete nodes.

Doubly linked list:

- Doubly linked list is a type of linked list in which two fields are reserved for addresses other than the information field, i.e. one for the previous node address, and one for the next node address.

Representation of doubly linked list:

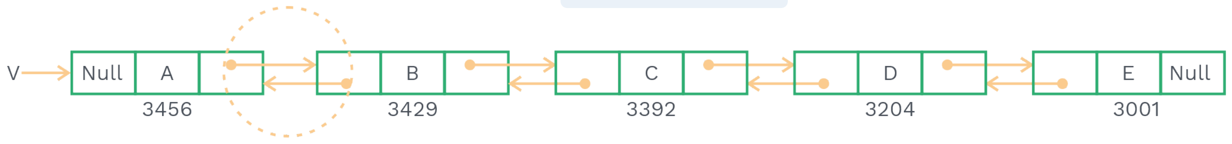


Fig. 4.12

- Each node contains three fields
  1) Previous field: Contains the address of the previous node means points to previous node.
  2) Information field: Contains objects
  3) Next field: Contains the address of the next node means points to the next node.
- The first node, previous address field posses null value. If some node contains the previous field with a null value, it means it is the first node of a doubly linked list.
- Last node, next address field posses a null value. If some node contains the next field with null value, it means it is the last node of a Doubly linked list.
- Doubly linked list is also flexible in size means we can add any number of nodes to the existing linked list and also can remove nodes.

Previous Years' Question

Q. N items are stored in a sorted doubly linked list. For a delete operation, a pointer is provided to the record to be deleted. For a decrease–key operation, a pointer is provided to the record on which the operation is to be performed.
An algorithm performs the following operations on the list in this order :
Θ(N), delete, O(logN) insert,
O(log N) fund, and Θ(N) decrease–key.
What is the time complexity of all these operations put together?

a) $O(\log^2 N)$ **b)** O(N)
c) $O(N^2)$ **d)** $\Theta(N^2 \log N)$

Sol: c) **(GATE CSE: 2016 (Set–2))**

Circular singly linked list:

- Circular single linked list is a type of linked list in which the last node next field contains first node address.
- Representation of circular single linked list.

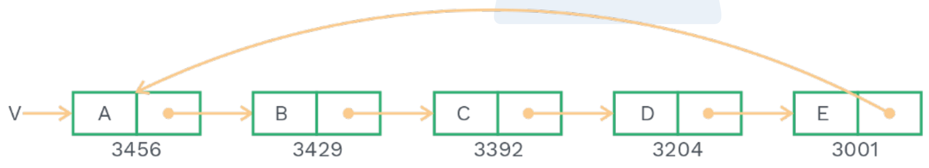


Fig. 4.13

- Each node similar to a single linked list contains two fields
  1) information field
  2) next field
- Last node next field contains the address of the first node of the linked list, which makes it circular.
- Similar to a single linked list, it is also flexible in size means we can add any no. of nodes to the existing linked list, which is circular and also can remove nodes by changing node links.

Previous Years' Question

Q. In a circular linked list organization, insertion of a record involves modification of:

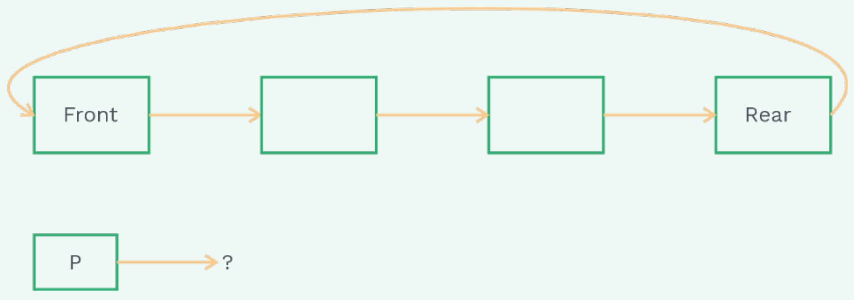
a) One pointer
b) Two pointer
c) Multiple pointers
d) No pointer

Sol: b) **(GATE CSE: 1987)**

- Advantage: We can go back in the list, wherever we want to reach in the circular single linked list.
- We can go back in the circular single linked list which takes O(n) worst case time complexity. 'n' is no. of elements.

Previous Years' Question

Q. A circularly linked list is used to represent a queue. A single variable p is used to access the queue. To which node should point p such that both the operations enqueue and dequeue can be performed in constant time?



a) rear node **b)** front node
c) not possible with a single pointer **d)** node next to front
Sol: a) **(GATE CSE: 2004)**

Circular doubly linked list:

- Circular doubly linked list follows two conditions on doubly linked list, which are:
  1) Last node next field contains first node address.
  2) First node previous field contains last node address.

Above two conditions on the doubly linked list make it circular doubly linked list.

- Representation of circular doubly linked list:

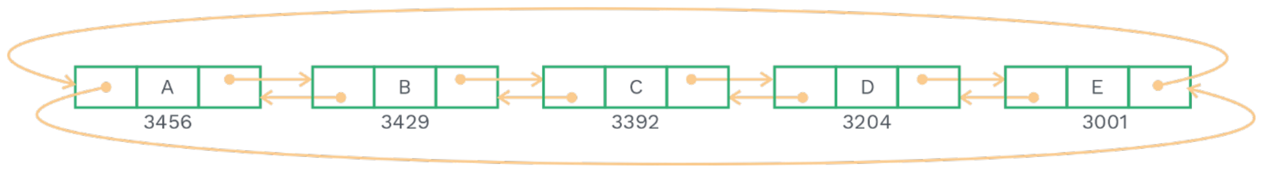


Fig. 4.14

- Each node similar to doubly linked list contains three fields:
  1) Previous field
  2) Information field
  3) Next field

- Similar to doubly linked list, it is also flexible in size means we can add any no. of nodes to the existing circular doubly linked list and also can remove nodes by changing node links.

4.4 IMPLEMENTATION OF LINKED LIST

Implementation of linked list using structures and pointers:

*** A brief note on structures and pointers:**

1) Structure:

It is a group of items in which each item is identified by its own identifier and its datatype.

- Each of the items is known as a member of the structure.
- Consider the following declaration:

```
Struct        {
        int a ;
        char b ;
        float c ;
    } A, B, C ;
```

- ‘Struct’ it a keyword used to declare a structure.
- There are three members of the structure they are:
  i) a)
  ii) b)
  iii) c)
- A, B, and C are structure variables, each of A, B, and C contains three members a, b, c.
- Using structure, one can define a datatype, which can be referred a user defined data type.

2) Pointer:

In C–language, programmers are allowed to reference the location of objects as well as the objects themselves.

- For example: Suppose there is a variable x of int type or float type or any type. Then, & x refers to the location which contains x. Here x is called a pointer.

Creating linked list:

Consider a linked list that we have to create,

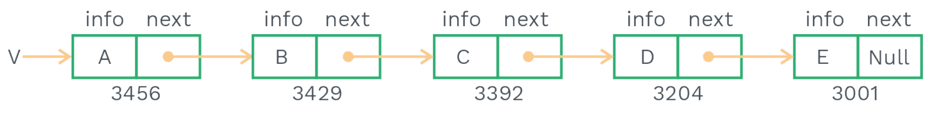


Fig. 4.15

Code:

```
Struct node
{
   Char info ;
   Struct node * next ;
}  A1, B1, C1, D1, E1 ;
```

// Creating node structure and defining 5–variables A1, B1, C1, D1 and E1 of it. Datatype of these variables is struct node.

```
A1 = {'A', Null} ;
B1 = {'B', Null} ;
C1 = {'C', Null} ;
D1 = {'D', Null} ;
E1 = {'E', Null} ;
```

// Assigning values to each node.

```
A1. next = & B1 ;
B1. next = & C1 ;
C1. next = & D1 ;
D1. next = & E1 ;
```

// Updating address field i.e. next field of each node with the address of next node.

```
Struct node * V = & A1 ;
```

// Assigning address of the first node to a pointer variable of type 'struct node'.

```
return (V) ;
```

// return the linked list.

Note:

We can create linked lists according to our needs means, whatever type of object, we are required to keep in the info field of a node, we can keep it; for example – int data, char data, float data etc.

Creating doubly linked list:

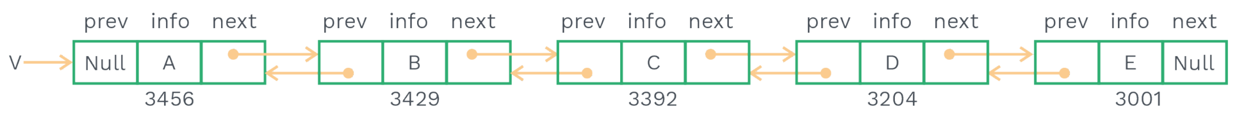


Fig. 4.16

```
Code:
Struct node
{
      Struct node* prev ;
      Char info ;
      Struct node * next ;
}     A1, B1, C1, D1, E1 ;

A1 = { Null, 'A', Null} ;
B1 = { Null, 'B', Null} ;
C1 = { Null, 'C', Null} ;
D1 = { Null, 'D', Null} ;
E1 = { Null, 'E', Null} ;

A1. next = & B1 ;

B1. prev = & A1 ;
B1. next = & C1 ;

C1. prev = & B1 ;
C1. next = & D1 ;

D1. prev = & C1 ;
D1. next = & E1 ;

E1. prev = & D1 ;

Struct node * V = & A1 ;
return (V) ;
```

Creating circular singly linked list:

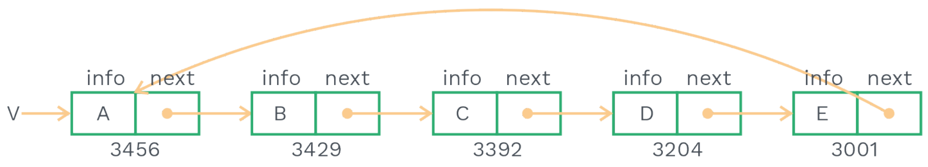


Fig. 4.17

Code:

```
Struct node
{
      Char info ;
      Struct node *next ;
}     A1, B1, C1, D1, E1 ;
A1 = {‘A’, Null} ;
B1 = {‘B’, Null} ;
C1 = {‘C’, Null} ;
D1 = {‘D’, Null} ;
E1 = {‘E’, Null} ;
A1. next = & B1 ;
B1. next = & C1 ;
C1. next = & D1 ;
D1. next = & E1 ;
E1. next = & A1 ;
Struct node * V = & A1 ;
return (V) ;
```

Creating circular doubly linked list:

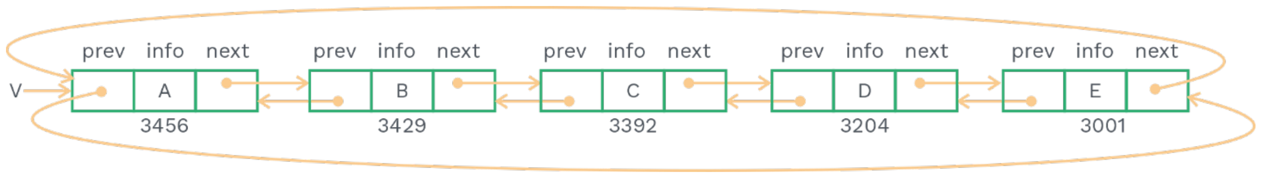


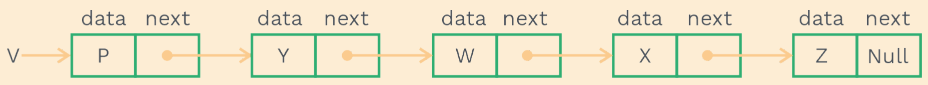
Fig. 4.18

Code:

```
Struct node
{
      Struct node * prev ;
      Char info ;
      Struct node *next ;
}     A1, B1, C1, D1, E1 ;
A1 = {Null, ‘A’, Null} ;
B1 = {Null, ‘B’, Null} ;
C1 = {Null, ‘C’, Null} ;
D1 = {Null, ‘D’, Null} ;
```

```
E1 = {Null, 'E', Null} ;
A1. prev = & E1 ;
A1. next = & B1 ;
B1. prev = & A1 ;
B1. next = & C1 ;
C1. prev = & B1 ;
C1. next = & D1 ;
D1. prev = & C1 ;
D1. next = & E1 ;
E1. prev = & D1 ;
E1. next = & A1 ;
Struct node *V = & A1 ;
return (V) ;
```

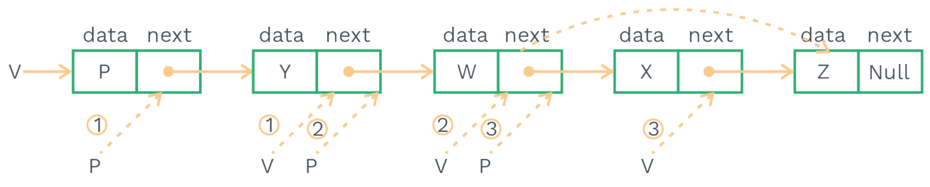
Q2 **Write a function for the given linked list to delete a node which contains the data 'x'.**



Sol:

```
Void delete_data_x( )
{ while (V → data ! = x && V → next ! = Null)
{
      Q = V ;
      P = V ;          // P is a new pointer variable.
      V = V → next;
}
      if (V → data == x)
{
      P → next = V → next ;

freenode (V);
V = Null ;
return (Q) ;
}
```



① ② ③ are sequence of execution of conditions in the function.

Resulted linked list is now:

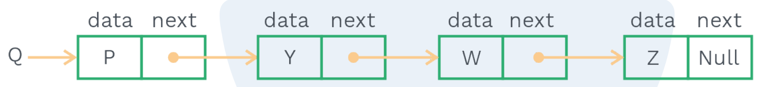


Fig. 4.19

Previous Years' Question

Q. Consider the function f defined below.

```
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

Sol: b)

4.4 USES OF LINKED LIST

- Different uses of linked list are as follows:

  1) Implementation of stacks:
  We can implement stacks by using a linked list, to satisfy the LIFO property by adding and removing nodes.

  2) Implementation of queues:
  Queues can be implemented by using a linked list to satisfy the FIFO property by adding nodes at last or by removing nodes from the start of the linked list.

  3) Representation of graphs:
  Graphs are represented by an adjacency list.
  For example:
  Let's take a graph:

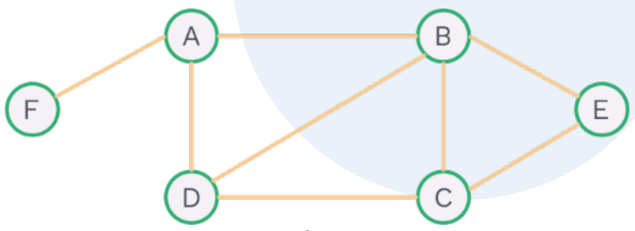


Fig. 4.20

Adjacency list representation of above graph:

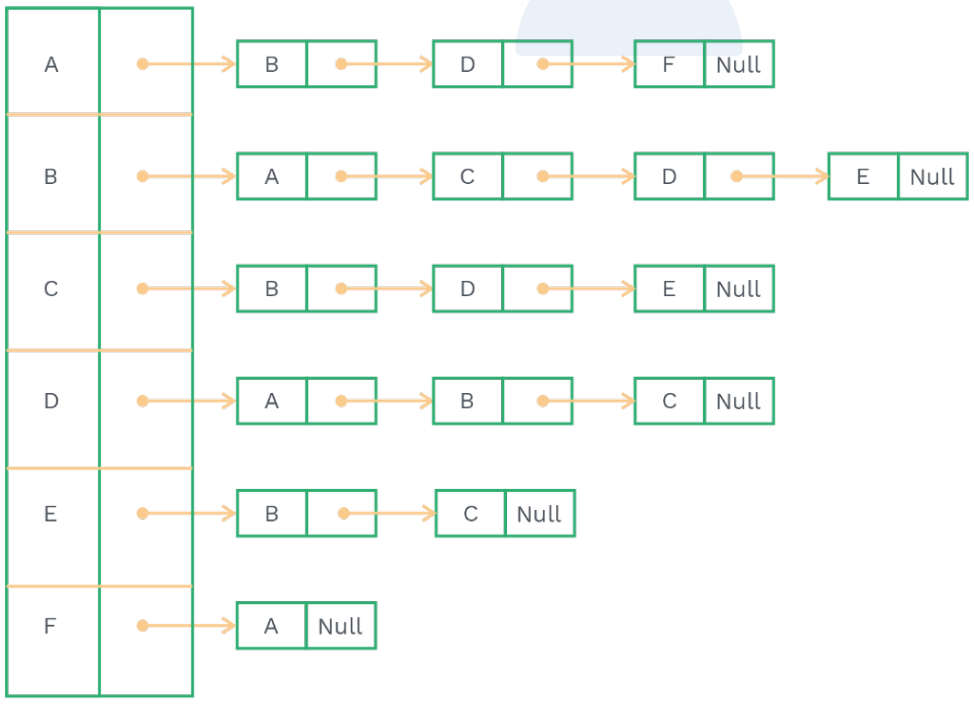


Fig. 4.21

4) Representation of sparse matrices:
To represent or to store sparse matrices, an array is not a good option. We use linked list to save space.
5) Memory allocation:
Dynamically, memory is allocated by using linked list by maintaining list of free blocks.

Chapter Summary

- Linked List is a data structure which is in the form of list of nodes, and contains two fields:

  1) Information field (Head): Information field contains actual data.
  2) Next–address field (Tail): Next–address field contains the address of the next node of the list.

- Linked list is a flexible data structure as we can insert any number of nodes and delete nodes from the given linked list.

- **Operations on linked list:**
  1) Insertion of nodes:
  Steps: **I)**– Create an empty node
  II)– Insert data
  III)– Update the next field of the node
  IV)– Update pointer

  2) Deletion of nodes:
  Steps: **I)** – Add one more pointer to the first node.
  II) – Next field of the new pointer is updated with the pointer of first node previously.
  III)– Take data to some variable.
  IV)– Free the node.

  3) Searching:
  Steps: **I)**– Start traversing nodes one by one.
  II) Compare into the field of nodes with the required data.

III) – Return the address of the node containing the data.

- **Types:** **1)** Singly Linked List
  2) Doubly Linked List
  3) Circular Singly Linked List
  4) Circular Doubly Linked List
- Searching time in linked list is O(n) [Worst case]
- **Usage:** **1)** Implementation of Stacks
  2) Implementation of Queues
  3) Representation of graph by adjacency list.
  4) For sparse matrices
  5) Memory allocation