LINKED LIST

- Linked list is a linear data structure that includes a series of connected nodes.
- Linked list can be defined as the nodes that are randomly stored in the memory.
- A node in the linked list contains two parts, i.e., first is the data part and second is the address part.
- The last node of the list contains a pointer to the null.
- After array, linked list is the second most used data structure.
- In a linked list, every link contains a connection to another link

Representation of a Linked list

- Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below.

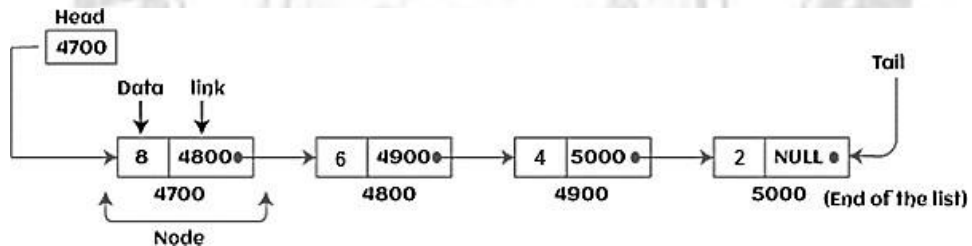


Fig. 3.4: Representation of Linked List

Why use linked list over array?

- Linked list is a data structure that overcomes the limitations of arrays (Refer 3.3.6)
- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration.
- List grows as per the program's demand and limited to the available memory space.

Declare a linked list

- It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e.,
  - Simple variable,
  - Pointer variable.

We can declare the linked list by using the user-defined data type structure.

The declaration of linked list is given as follows

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

In the above declaration, we have defined a structure named as node that contains two variables, one is data that is of integer type, and another one is next that is a pointer which contains the address of next node.

Types of Linked list

Linked list is classified into the following types

- Singly-Linked List
- Doubly Linked List
- Circular Singly Linked List
- Circular Doubly Linked List

Singly-Linked List

- Singly linked list can be defined as the collection of an ordered set of elements.
- A node in the singly linked list consists of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node
- Link part of the node stores the address of its immediate successor.

Doubly Linked List

- Doubly linked list is a complex type of linked list
- Here, a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly-linked list, a node consists of three parts:
  - Node data,
  - Pointer to the next node in sequence (next pointer), and
  - Pointer to the previous node (previous pointer).

Circular Singly Linked List

- In a circular singly linked list, the last node of the list contains a pointer to the first node of the list.
- We can have circular singly linked list as well as circular doubly linked list.

Circular Doubly Linked List

- Circular doubly linked list is a more complex type of data structure.
- Here a node contains pointers to its previous node as well as the next node.
- Circular doubly linked list doesn't contain NULL in any of the nodes.
- The last node of the list contains the address of the first node of the list.
- The first node of the list also contains the address of the last node in its previous pointer.

Advantages of Linked list

- **Dynamic data structure** - The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion** - Unlike arrays, insertion, and deletion in linked list is easier. Elements in the linked list are stored at a random location.
- **Memory efficient** - The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation** - We can implement both stacks and queues using linked list.

Disadvantages of Linked list

- **Memory usage** - In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Traversal** - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly.
- **Reverse traversing** - Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked list

- Using linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

- Insertion - This operation is performed to add an element into the list.
- Deletion - It is performed to delete an operation from the list.
- Display - It is performed to display the elements of the list.
- Search - It is performed to search an element from the list using the given key.

Complexity of Linked list

1. Time Complexity

Operation	Average Case	Worst Case
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

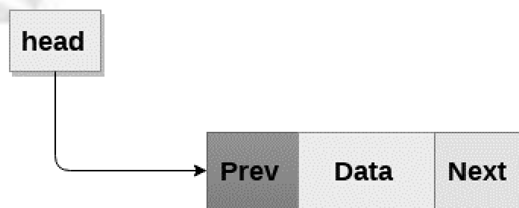
2. Space Complexity

Operation	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

DOUBLY LINKED LIST

- Doubly linked list is a complex type of linked list
- Here, a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly-linked list, a node consists of three parts:
  - Node data,
  - Pointer to the next node in sequence (next pointer), and
  - Pointer to the previous node (previous

pointer).A sample node in a doubly linked list is shown in



Node

Fig. 3.5

Fig. 3.5: Sample node in a doubly linked list

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in Fig 3.6

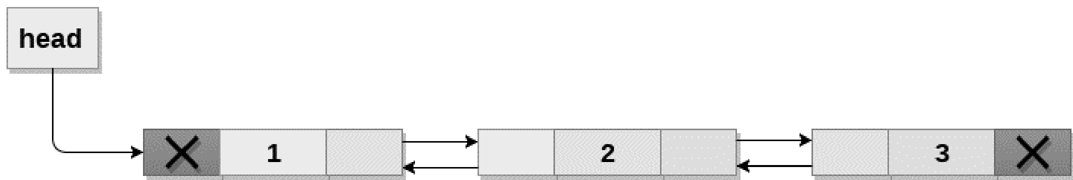


Fig. 3.6: Doubly Linked List

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

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

- The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.
- In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list.
- Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

- Memory Representation of a doubly linked list is shown in Fig. 3.7. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).
- In Fig 3.7, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added

to the list therefore the **prev** of the list contains null. The **next** node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

- We can traverse the list in this way until we find any node containing null or -1 in its next part.

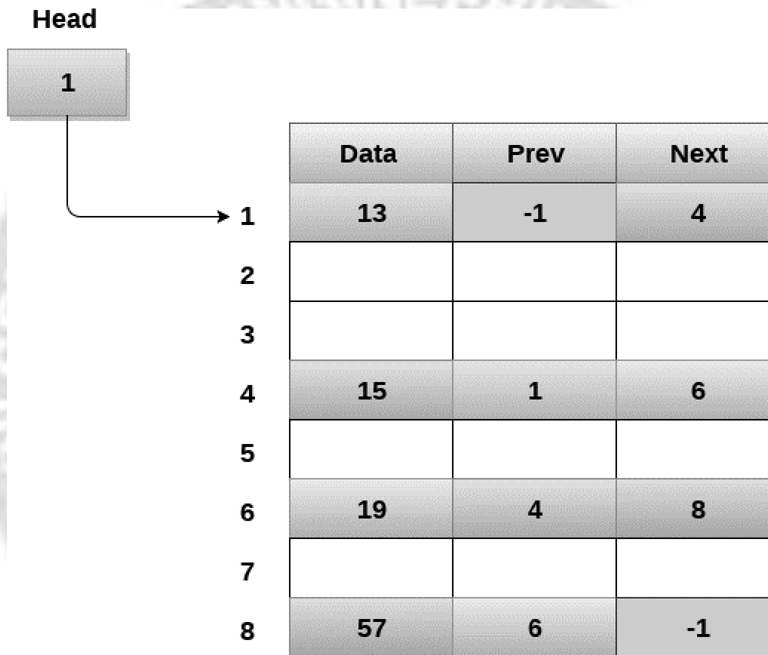


Fig. 3.7: Memory Representation of a doubly linked list

Operations on doubly linked list

Table 3.1 describes all the operations performed on Doubly Linked List

Table 3.1: Operations on Doubly Linked List

Sl. No.	Operation	Description
1.	Insertion at beginning	Adding the node into the linked list at beginning.
2.	Insertion at end	Adding the node into the linked list to the end.
3.	Insertion after specified node	Adding the node into the linked list after the specified node.
4.	Deletion at beginning	Removing the node from beginning of the list
5.	Deletion at the end	Removing the node from end of the list.

6.	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7.	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null
8.	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion at beginning

- There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element.

The following steps to be performed for insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory.
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds.
- In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
- In the second scenario, the condition head == NULL become false and the node will be inserted in beginning.
- The next pointer of the node will point to the existing head pointer of the node.
- The prev pointer of the existing head will point to the new node being inserted.
- Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer.
- Hence assign null to its previous part and make the head point to this node.

Algorithm 3.1

Step 1: IF ptr = NULL

Write OVERFLOW

Go to Step 9

[END OF IF]

Step 2: SET NEW_NODE = ptr

Step 3: SET ptr = ptr -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> PREV = NULL

Step 6: SET NEW_NODE -> NEXT = START

Step 7: SET head -> PREV = NEW_NODE

Step 8: SET head = NEW_NODE

Step 9: EXIT

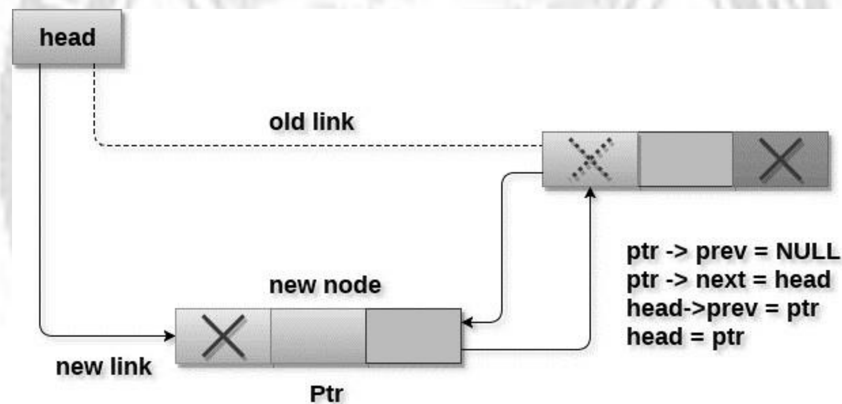


Fig. 3.8: Insertion into Doubly Linked List at the beginning

Insertion at end

- To insert a node in doubly linked list at the end, make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.
- Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds.
- In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list.

- For this reason, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.
- Make the next pointer of temp point to the new node being inserted i.e. ptr.
- Make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.
- Make the next pointer of the node ptr point to the null as it will be the new last node of the list.

Algorithm 3.2

Step 1: IF PTR = NULL

Write OVERFLOWGo to Step 11 [END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET TEMP = START

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

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10C: SET NEW_NODE -> PREV = TEMP

Step 11: EXIT

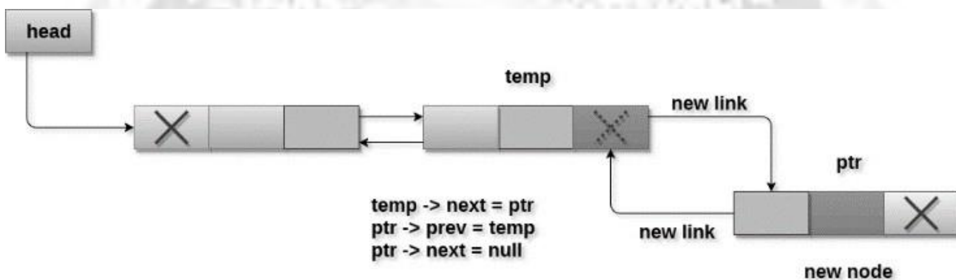


Fig. 3.9: Insertion into Doubly Linked List at the end

Insertion after specified node

To insert a node after the specified node in the list, we need to skip the required

number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

The following steps are used for this purpose.

- Allocate the memory for the new node.
- Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.
- The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node. Make the next pointer of ptr point to the next node of temp.
- Make the prev of the new node ptr point to temp.
- Make the next pointer of temp point to the new node ptr.
- Make the previous pointer of the next node of temp point to the new node.

Algorithm 3.3

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 15

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = START

Step 6: SET I = 0

Step 7: REPEAT 8 to 10 until I

Step 8: SET TEMP = TEMP -> NEXT

STEP 9: IF TEMP = NULL

STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 15

[END OF IF]

[END OF LOOP]

Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT

Step 12: SET NEW_NODE -> PREV = TEMP

Step 13 : SET TEMP -> NEXT = NEW_NODE

Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE

Step 15: EXIT

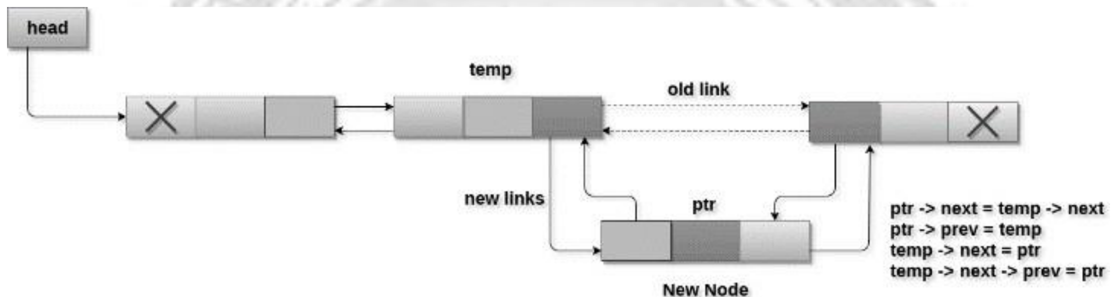


Fig. 3.10: Insertion of Doubly Linked List after specified node

Deletion at beginning

- Deletion in doubly linked list at the beginning is the simplest operation.
- Just need to copy the head pointer to pointer ptr and shift the head pointer to itsnext.
- Make the prev of this new head node point to NULL.
- Now free the pointer ptr by using the free function.

Algorithm 3.4

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 6

STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = HEAD →

NEXTSTEP 4: SET HEAD → PREV =

NULLSTEP 5: FREE PTR

STEP 6: EXIT

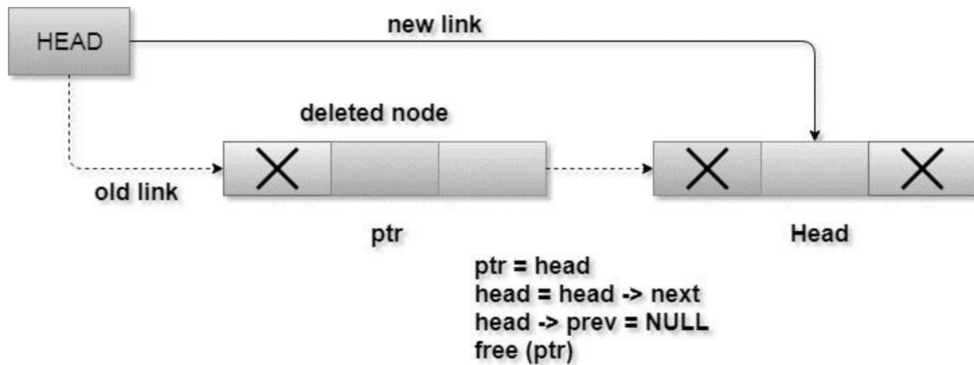


Fig. 3.11: Deletion in Doubly Linked List from beginning

Deletion at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position. To delete the last node of the list, following steps are performed.

- If the list is already empty then the condition head == NULL will become true and therefore the operation cannot be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list.
- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of ptr to NULL.
- Free the pointer as this the node which is to be deleted.

Algorithm 3.5:

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 7

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL

Step 4: SET TEMP = TEMP->NEXT

[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT

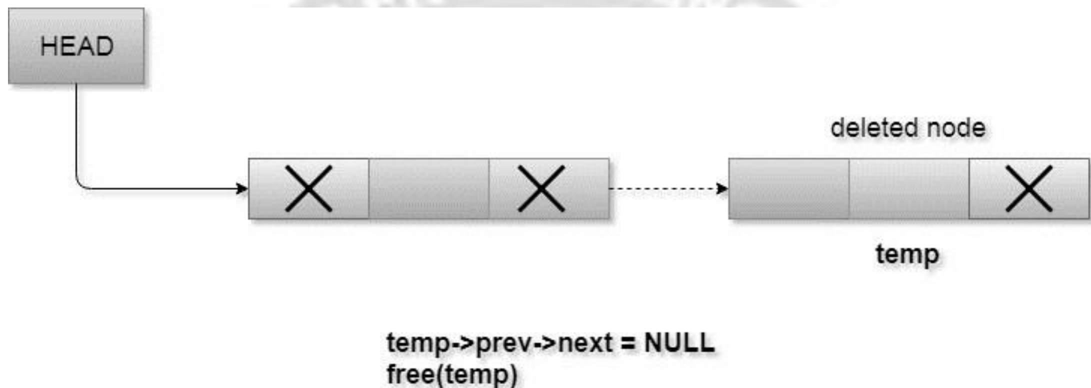


Fig. 3.12: Deletion in Doubly Linked List at the end

Deletion of the node having given data

- Copy the head pointer into a temporary pointer temp.
- Traverse the list until we find the desired data value.
- Check if this is the last node of the list. If it is so then we can't perform deletion.
- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
- Otherwise, make the pointer ptr point to the node which is to be deleted.
- Make the next of temp point to the next of ptr.
- Make the previous of next node of ptr point to temp. free the ptr.

Algorithm 3.6

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> DATA !=
ITEMStep 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 5: SET PTR = TEMP -> NEXT
Step 6: SET TEMP -> NEXT = PTR ->
NEXTStep 7: SET PTR -> NEXT -> PREV =
TEMPStep 8: FREE PTR
Step 9: EXIT

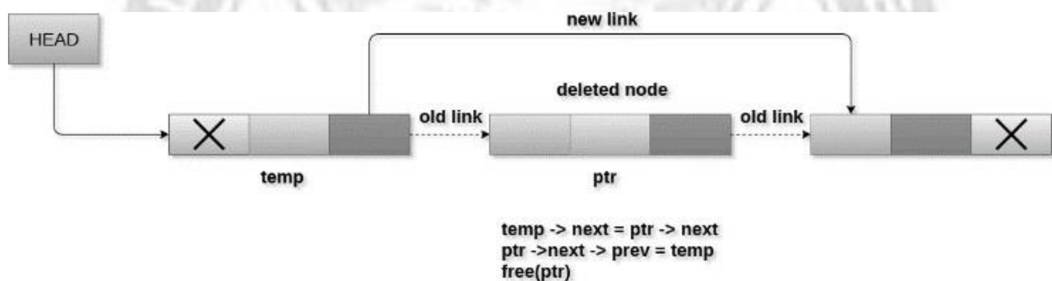


Fig. 3.13: Deletion of a specified node in Doubly Linked List at the end

Searching for a specific node

To search for a specific element in the list, traverse the list in order. The following operations to be performed in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm 3.7

Step 1: IF HEAD == NULL
WRITE "UNDERFLOW"
GOTO STEP 8
[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Set i = 0

Step 4: Repeat step 5 to 7 while PTR != NULL

Step 5: IF PTR → data = item

return i

[END OF IF]

Step 6: i = i + 1

Step 7: PTR = PTR → next

Step 8: Exit

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose,

- Copy the head pointer in any of the temporary pointer ptr.
- Traverse through the list by using while loop.
- Keep shifting value of pointer variable ptr until we find the last node.
- The last node contains null in its next part.

Algorithm 3.8

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW"

GOTO STEP 6

[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

Step 4: Write PTR → data

Step 5: PTR = PTR → next

Step 6: Exit

Advantages of Doubly Linked Lists

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.

- As with a singly linked list, it is the easiest data structure to implement.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List.

Disadvantages of Doubly Linked Lists

- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.