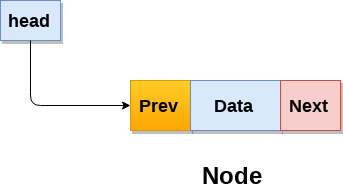
Doubly linked list

Doubly linked list is a complex type of linked list in which 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) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



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.

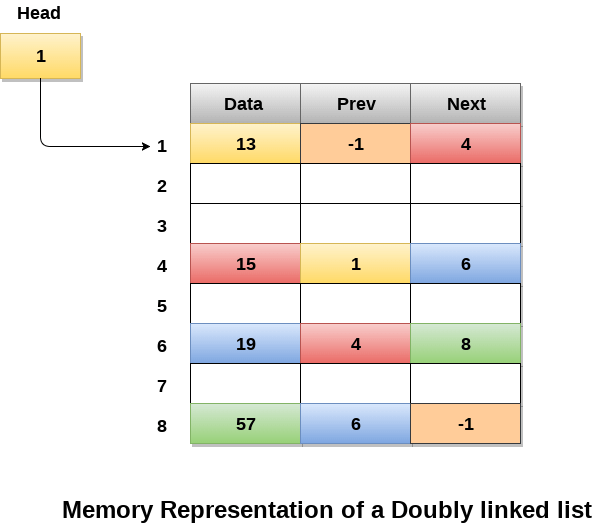
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains 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 the following image. 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 the following image, 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.



## **Operations on doubly linked list**

**Node Creation**

struct node

{

    struct node \*prev;

**int** data;

    struct node \*next;

};

struct node \*head;

# A complete working Python program to demonstrate all

# insertion methods

# A linked list node

class Node:

    # Constructor to create a new node

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

        self.prev = None

# Class to create a Doubly Linked List

class DoublyLinkedList:

    # Constructor for empty Doubly Linked List

    def \_\_init\_\_(self):

        self.head = None

    # Given a reference to the head of a list and an

    # integer, inserts a new node on the front of list

    def push(self, new\_data):

        # 1. Allocates node

        # 2. Put the data in it

        new\_node = Node(new\_data)

        # 3. Make next of new node as head and

        # previous as None (already None)

        new\_node.next = self.head

        # 4. change prev of head node to new\_node

        if self.head is not None:

            self.head.prev = new\_node

        # 5. move the head to point to the new node

        self.head = new\_node

    # Given a node as prev\_node, insert a new node after

    # the given node

    def insertAfter(self, prev\_node, new\_data):

        # 1. Check if the given prev\_node is None

        if prev\_node is None:

            print "the given previous node cannot be NULL"

            return

        # 2. allocate new node

        # 3. put in the data

        new\_node = Node(new\_data)

        # 4. Make net of new node as next of prev node

        new\_node.next = prev\_node.next

        # 5. Make prev\_node as previous of new\_node

        prev\_node.next = new\_node

        # 6. Make prev\_node ass previous of new\_node

        new\_node.prev = prev\_node

        # 7. Change previous of new\_nodes's next node

        if new\_node.next is not None:

            new\_node.next.prev = new\_node

    # Given a reference to the head of DLL and integer,

    # appends a new node at the end

    def append(self, new\_data):

        # 1. Allocates node

        # 2. Put in the data

        new\_node = Node(new\_data)

        # 3. This new node is going to be the last node,

        # so make next of it as None

        new\_node.next = None

        # 4. If the Linked List is empty, then make the

        # new node as head

        if self.head is None:

            new\_node.prev = None

            self.head = new\_node

            return

        # 5. Else traverse till the last node

        last = self.head

        while(last.next is not None):

            last = last.next

        # 6. Change the next of last node

        last.next = new\_node

        # 7. Make last node as previous of new node

        new\_node.prev = last

        return

    # This function prints contents of linked list

    # starting from the given node

    def printList(self, node):

        print "\nTraversal in forward direction"

        while(node is not None):

            print " % d" %(node.data),

            last = node

            node = node.next

        print "\nTraversal in reverse direction"

        while(last is not None):

            print " % d" %(last.data),

            last = last.prev

# Driver program to test above functions

# Start with empty list

llist = DoublyLinkedList()

# Insert 6. So the list becomes 6->None

llist.append(6)

# Insert 7 at the beginning.

# So linked list becomes 7->6->None

llist.push(7)

# Insert 1 at the beginning.

# So linked list becomes 1->7->6->None

llist.push(1)

# Insert 4 at the end.

# So linked list becomes 1->7->6->4->None

llist.append(4)

# Insert 8, after 7.

# So linked list becomes 1->7->8->6->4->None

llist.insertAfter(llist.head.next, 8)

print "Created DLL is: ",

llist.printList(llist.head)

**Output:**

Created DLL is:

Traversal in forward direction

1 7 8 6 4

Traversal in reverse direction

4 6 8 7 1