PYTHON PROJECT

DOUBLY LINKED LIST

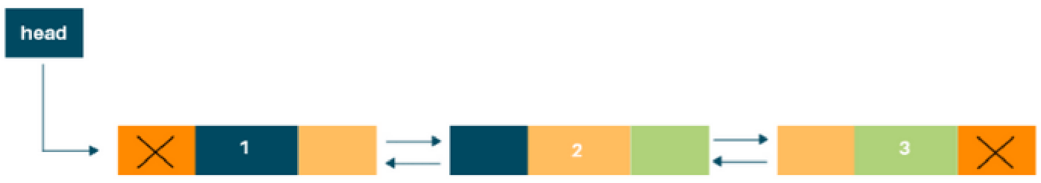
Submitted by

SUBRAMANYA H (20GACSE069)

SUCHIT PRIYADARSHI (20GACSE070)

SUJATHA BHAT (20GACSE071)

Doubly Linked List

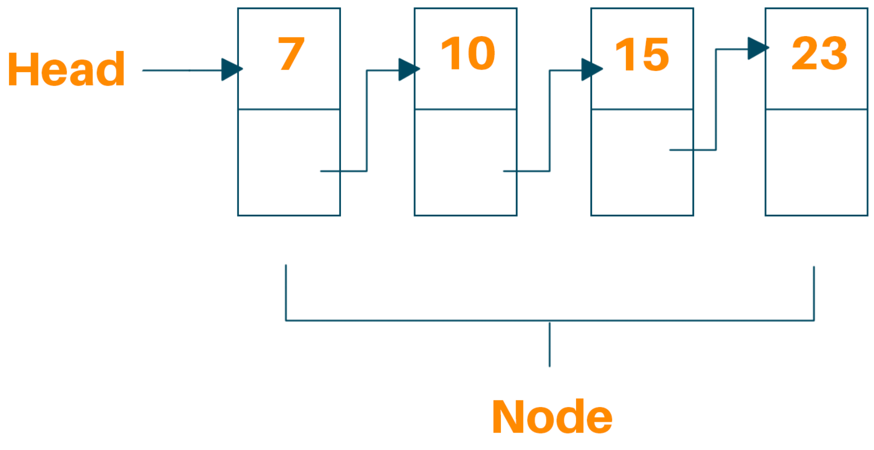


Ctrl+Z, can you even survive a day without these keys? Well, Apparently not, undo and redo features are one of the used functionalities in computer engineering. But have you ever thought about how is it even possible? I happen to have an answer this time! Undo and Redo functionality is one of the most important applications of a doubly linked list. So what exactly is a doubly linked list? Before that let's have a quick glance at Linked List:

What is Linked List?

Linked list is a linear data structure in which each node is an object. Different from general Arrays, the Linked list data structure does not have a contiguous memory structure. Every element is connected to the subsequent node through a pointer.

Every element present in the Linked list is known as Node. Every node contains a key or data element with an extra pointer pointing to the next element in the list. The initial pointer in the Linked list is called Head. It is necessary to state that "Head" pointer is not another node but a pointer to the first element of the Linked list. If the Linked list are empty value of Head will be null.



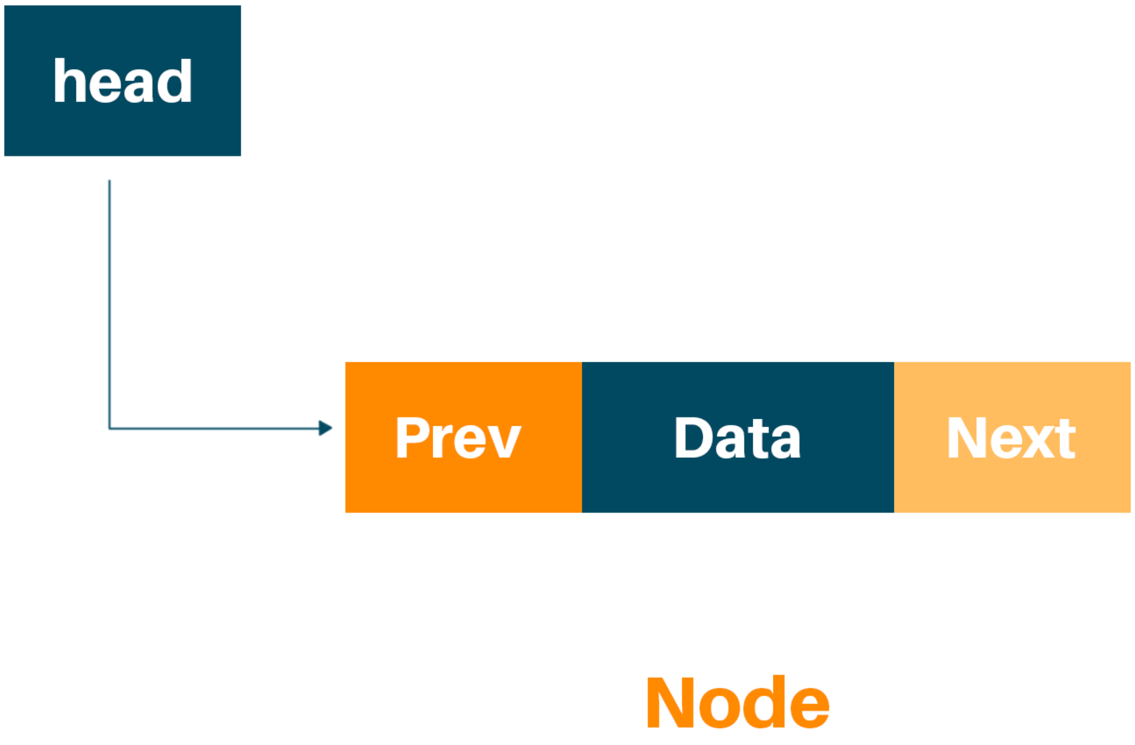
What is Doubly Linked List ?

It is easier to implement Singly Linked list or Linked list whereas it is pretty difficult to traverse that in reverse, to overcome this we can utilise Doubly LinkedList, where every node takes an additional pointer to point the former node to the element in addition to the pointer for the next node.

A doubly linked list has more efficient iteration, particularly if you need to ever iterate in reverse and more efficient deletion of particular nodes.

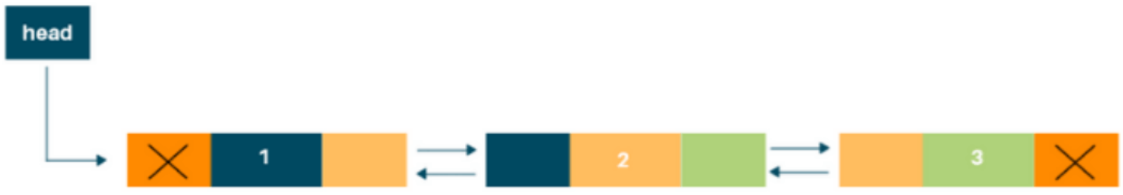
We can conclude that a doubly linked list is a complex type of linked list where a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three components: node data, pointer to the next node in node(next pointer), pointer to the former node (previous pointer).

```
class Node:

  def __init__(self, data, prev, next):

    self.data = data

    self.next = next

    self.prev = prev
```



Example

For example, consider a doubly linked list containing four nodes having, four numbers from 1 to 4 in their data part, illustrated in the following image



Step-by-Step Implementation of Doubly Linked List in Python

We add the following functionalities to the doubly Linked List class. We add these functions to the class.

1. Inserting Items to Empty List
2. Inserting Items at the End
3. Deleting Elements from the Start
4. Deleting Elements from the End
5. Traversing the Linked List

- **Inserting Items at the Front :**

The most permissive way to insert an element in a doubly linked list is to insert the element in the empty list. The following piece of code enters an element at the origin of the doubly linked list:

```
def insertFront(self, data) :

    newNode = Node(data, None, self._head)

    if(self._head != None) :

        self._head.prev = newNode

    self._head = newNode

    return True # insert success
```

Inserting Items at the End :

Inserting an element at the end of the doubly linked list is slightly alike to inserting an element at the start of the list. Before entering the elements, we need to verify and check if the doubly linked list is empty. If the list is empty then we can easily use the function insertFront() method to insert the element to the list. If the list already has added element, we iterate through the list till the address to the next node reaches NULL. When the next node address becomes NULL, it denotes that the current node is the last node.

The previous reference for the new node is set to the last node, and the next address for the last node is assigned to the newly inserted node. The function for inserting an item at the last node is as follows:

```
def insertRear(self, data) :

  curr = self.__head

  if(curr == None) : # List is empty

    self.__head = Node(data, None, None)

  else :

    # take curr to the last node

    while(curr.next != None) :

      curr = curr.next

    curr.next = Node(data, curr, None)

  return True # insert success
```

- **Deleting Elements from the Start**

The most natural way to delete an element from a doubly linked list is deleting from the start of the list. In order to achieve this, we will assign the value of the start node to the next node and then assign the previous address of the start node to NULL. But before doing this, we need to check two things. First, we need to check if the list is empty. And then we have to verify if the list has only one node or not. If the list has only one element then we can easily assign the start node to NULL. The following code can be utilised to delete node from the start of the doubly linked list.

```
def deleteFront(self) :

  if(self.__head == None) : # List is empty

    return False # delete failure

  # Shift the header to the leading node

  self.__head = self.__head.next

  if(self.__head != None) :

    self.__head.prev = None
    return True # delete success
```

- **Deleting Elements from the End**

To remove the element from the end, we have to again verify if the list is empty or if the list has a single element. If the list has a single element, we will assign the start node to NULL. If the doubly linked list has more than one element, we traverse through the list until the last node reaches NULL. Once we reach the last node, we assign the next address of the node previous to the last node, to NULL which actually deletes the last node. The following function can be used to delete the element from the end of the list.

```
  def deleteRear(self) :
    if(self.__head == None) : # List is empty
      return False # delete failure
    prev, curr = self.__head, self.__head
    #take curr to the last node and prev to the penultimate node
    while(curr.next != None) :
      prev = curr
      curr = curr.next
    if(curr == self.__head) : #  last node is the header node itself
      self.__head = None
    else :
      prev.next = None
    del curr
    return True # delete success
```

- **Traversing the Linked List**

Display the elements in the doubly linked list, while iterating through each element.

```
 def PrintList(self):
    curr = self.__head;
    # curr pointer traverses the linked list from header to the last node
```

```
while(curr != None):
   print(curr.data)
   curr = curr.next
return True
```

- **Sorting (Insertion Sort) :**

In this article, we will discuss the Insertion sort Algorithm. The working procedure of insertion sort is also simple. This article will be very helpful and interesting to students as they might face insertion sort as a question in their examinations. So, it is important to discuss the topic.

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted

Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is $O(n)$.
- Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is $O(n^2)$.
- Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending

order, but its elements are in descending order. The worst-case time complexity of **insertion sort is O(n^2).**

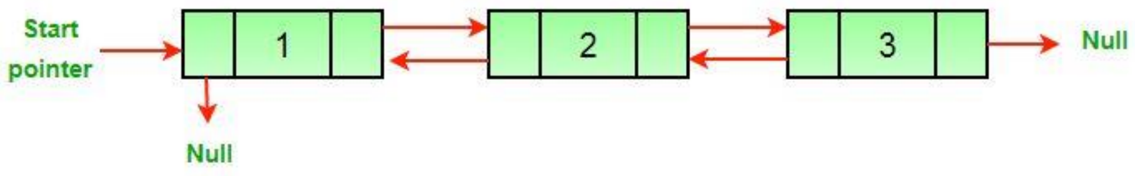
2. Space Complexity

Space Complexity	**O (1)**
Stable	**YES**

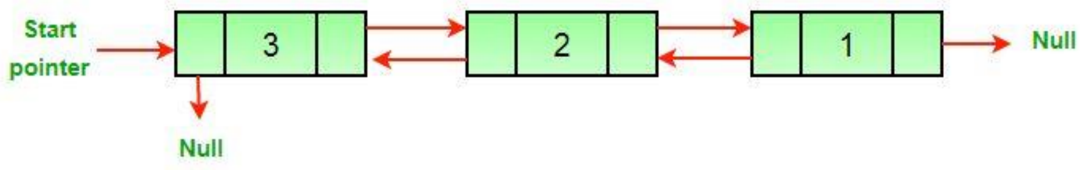
```
def sort(self) :

    curr = self.__head

    if(curr == None) : return True

    while(curr != None) :

      value, back = curr.data, curr

      while(back.prev != None and back.prev.data > value) :

        back.data = back.prev.data

        back = back.prev

      back.data = value

      curr = curr.next
```

- ***Reversing a Doubly Linked List :***

Before reversal :



After reversal :



```
def reverse(self) :
    if(self.__head == None) : return False
    curr = self.__head
    while(curr != None) :
      curr.prev , curr.next = curr.next, curr.prev
      self.__head = curr
      curr = curr.prev
    return True
```

- <u>**Complete python code of Doubly Linked List**</u>

```
class Node:
  def __init__(self, data, prev, next):
    self.data = data
    self.next = next
    self.prev = prev

class DoublyLinkedList :

  def __init__(self):
    self.__head = None #head is a private member(pointer)that points to the header node
of the linked list

  #Insert operations on the linked list :
  def insertFront(self, data) :
    newNode = Node(data, None, self.__head)
    if(self.__head != None) :
        self.__head.prev = newNode
    self.__head = newNode
    return True # insert success

def insertRear(self, data) :
    curr = self.__head
    if(curr == None) : # List is empty
        self.__head = Node(data, None, None)
    else :
        # take curr to the last node
        while(curr.next != None) :
            curr = curr.next
        curr.next = Node(data, curr, None)
    return True # insert success

# Delete operations on the linked list :

def deleteFront(self) :
    if(self.__head == None) : # List is empty
        return False # delete failure
    # Shift the header to the leading node
    self.__head = self.__head.next
    if(self.__head != None) :
        self.__head.prev = None
    return True # delete success
```

```
def deleteRear(self) :
   if(self.__head == None) : # List is empty
      return False # delete failure
   prev, curr = self.__head, self.__head
   #take curr to the last node and prev to the penultimate node
   while(curr.next != None) :
      prev = curr
      curr = curr.next
   if(curr == self.__head) : #  last node is the header node itself
      self.__head = None
   else :
      prev.next = None
   del curr
   return True # delete success


# Utility operations on the linked list :
def PrintList(self):
   curr = self.__head;
   # curr pointer traverses the linked list from header to the last node
   while(curr != None):
      print(curr.data)
      curr = curr.next
   return True


def reverse(self) :
    if(self.__head == None) : return False
    curr = self.__head
    while(curr != None) :
        curr.prev , curr.next = curr.next, curr.prev
        self.__head = curr
        curr = curr.prev
    return True

 # Insertion Sort (Non Decreasing Order)
def sort(self) :
    curr = self.__head
    if(curr == None) : return True
    while(curr != None) :
        value, back = curr.data, curr
        while(back.prev != None and back.prev.data > value) :
            back.data = back.prev.data
            back = back.prev
        back.data = value
        curr = curr.next

def removeDuplicates(self) :
    curr = self.__head
    if(curr == None) : return False
    # two pointer appraoch by sorting
    self.sort()
    slow, fast = curr, curr.next
    # slow + 1th node marks the end of unique numbers


    while(fast != None) :

      if(slow.data != fast.data) :

        slow = slow.next

        slow.data = fast.data

      fast = fast.next


    # slow + 1th node till end are redundant nodes therefore delete those

    curr = slow.next

    slow.next = None


    while(curr != None) :

      delNode = curr

      curr = curr.next

      del delNode


    return True


#Driver Code : main()

d = DoublyLinkedList()

# Perform operations on d here
```

Applications

- A music player that has next and previous buttons, like the concept of doubly linked list you'll have functionality to go back and the next element. .
- Undo Redo operations that we discussed at the beginning of the article.

- The browser cache functionality in common web browsers like Chrome, Microsoft edge, which allows us to move front and back within pages is also a great application of a doubly linked list.
- LRU cache, also Most Recently Used is also an instance of DLL.
- A deck of cards in a game is a perfect example of the applicability of DLL. It is additionally utilized to store the state of the game during play.