

4

Linked Lists

Python's list implementation is quite powerful and can encompass several different use cases. We have discussed the built-in data structures of **lists** in Python in *Chapter 1, Python Data Types and Structures*. Most of the time, Python's built-in implementation of a list data structure is used to store data using a linked list. In this chapter, we will understand how linked lists work along with their internals.

A linked list is a data structure where the data elements are stored in a linear order. Linked lists provide efficient storage of data in linear order through pointer structures. Pointers are used to store the memory address of data items. They store the data and location, and the location stores the position of the next data item in the memory.

The focus of this chapter will be the following:

- Arrays
- Introducing linked lists
- Singly linked lists
- Doubly linked lists
- Circular lists
- Practical applications of linked lists

Before discussing linked lists, let us first discuss an array, which is one of the most elementary data structures.

Arrays

An array is a collection of data items of the same type, whereas a linked list is a collection of the same data type stored sequentially and connected through pointers. In the case of lists, the data elements are stored in different memory locations, whereas the array elements are stored in contiguous memory locations.

An array stores the data of the same data type and each data element in the array is stored in contiguous memory locations. Storing multiple data values of the same type makes it easier and faster to compute the position of any element in the array using **offset** and **base address**. The term *base address* refers to the address of memory location where the first element is stored, and offset refers to an integer indicating the displacement between the first element and a given element.

Figure 4.1 demonstrates an array holding a sequence of seven integer values that are stored sequentially in contiguous memory locations. The first element (data value 3) is stored at index 0, the second element at index position 1, and so on.

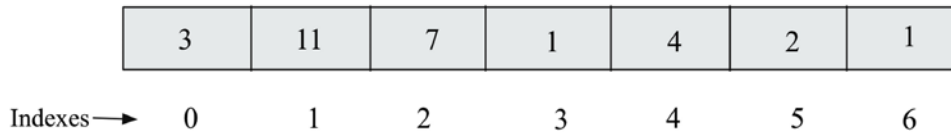


Figure 4.1: Representation of a one-dimensional array

To store, traverse, and access array elements is very fast as compared to lists since elements can be accessed randomly using their index positions, whereas in the case of a linked list, the elements are accessed sequentially. Therefore, if the data to be stored in the array is large and the system has low memory, the array data structure will not be a good choice to store the data because it is difficult to allot a large block of memory locations. The array data structure has further limitations in that it has a static size that has to be declared at the time of creation.

In addition, the insertion and deletion operations in array data structures are slow as compared to linked lists. This is because it is difficult to insert an element in an array at a given location since all data elements after that desired position must be shifted and then new elements inserted in between. Thus, array data structures are suitable when we want to do a lot of accessing of elements and fewer insertion and deletion operations, whereas linked lists are suitable in applications where the size of the list is not fixed, and a lot of insertion and deletion operations will be required.

Introducing linked lists

The linked list is an important and popular data structure with the following properties:

1. The data elements are stored in memory in different locations that are connected through pointers. A pointer is an object that can store the memory address of a variable, and each data element points to the next data element and so on until the last element, which points to None.
2. The length of the list can increase or decrease during the execution of the program.

Contrary to arrays, linked lists store data items sequentially in different locations in memory, wherein each data item is stored separately and linked to other data items using pointers. Each of these data items is called a node. More specifically, a node stores the actual data and a pointer. In *Figure 4.2*, nodes A and B store the data independently, and node A is connected to node B.

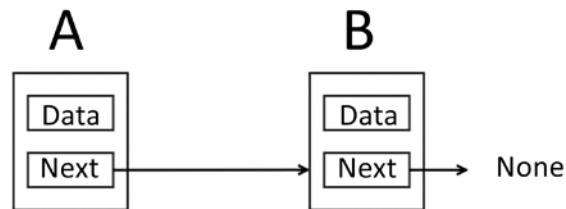


Figure 4.2: A linked list with two nodes

Moreover, the nodes can have links to other nodes based differently on how we want to store the data, and on which basis we will learn various kinds of data structures, such as singular linked lists, doubly linked lists, circular link lists, and trees.

Nodes and pointers

A node is a key component of several data structures such as linked lists. A node is a container of data, together with one or more links to other nodes where a link is a pointer.

To begin with, let us consider an example of creating a linked list of two nodes that contains data (for example, strings). For this, we first declare the variable that stores the data along with pointers that point to the next variable. Consider the example in the following *Figure 4.3*, in which there are two nodes. The first node has a pointer to the string (**eggs**), and another node pointing to the **ham** string.

Furthermore, the first node that points to the **eggs** string has a link to another node. Pointers are used to store the address of a variable, and since the string is not actually stored in the node, rather, the address of the string is stored in the node.

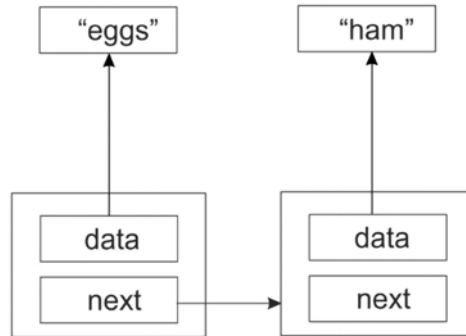


Figure 4.3: A sample linked list of two nodes

Furthermore, we can also add a new third node to this existing linked list that stores **spam** as a data value, while a second node points to the third node, as shown in Figure 4.4. Hence, Figure 4.3 demonstrates the structure of three nodes having data strings, in other words, **eggs**, **ham**, and **spam**, which are stored sequentially in a linked list.

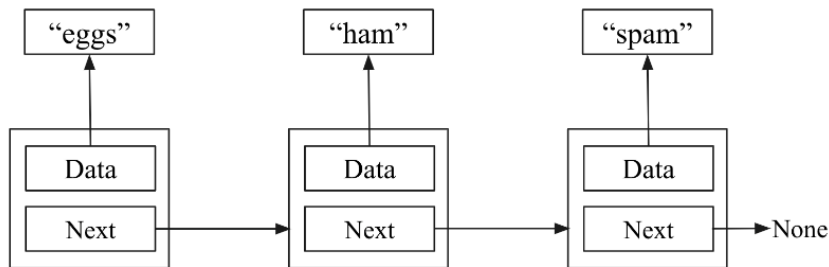


Figure 4.4: A sample linked list of three nodes

So, we have created three nodes—one containing **eggs**, one **ham**, and another **spam**. The **eggs** node points to the **ham** node, which in turn points to the **spam** node. But what does the **spam** node point to? Since this is the last element in the list, we need to make sure its next member has a value that makes this clear. If we make the last element point to nothing, then we make this fact clear. In Python, we will use the special value **None** to denote nothing. Consider Figure 4.5. Node **B** is the last element in the list, and thus it is pointing to **None**.

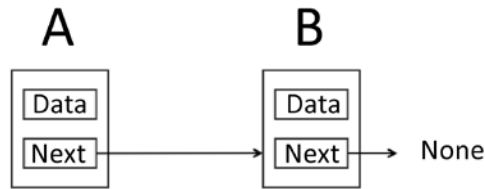


Figure 4.5: A linked list with two nodes

Let us first learn about the implementation of the node, as shown in the following code snippet:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

Here, the **next** pointer is initialized to **None**, meaning that unless we change the value of **next**, the node is going to be an endpoint, meaning that initially, any node that is attached to the list will be independent.

You can also add any other data items to the node class if required. If your node is going to contain customer data, then create a **Customer** class and place all the data there.

There are three kinds of list—a singly linked list, a doubly linked list, and a circular linked list. First of all, let's discuss singly linked lists.

We need to learn the following operations in order to use any linked lists in real-time applications.

- Traversing the list
- Inserting a data item in the list:
 - Inserting a new data item (node) at the beginning
 - Inserting a new data item (node) at the end of the list
 - Inserting a new data item (node) in the middle/or at any given position in the list
- Deleting an item from the list:
 - Deleting the first node
 - Deleting the last node
 - Deleting a node in the middle/or at any given position in the list

We will be discussing these important operations on different types of linked lists in subsequent subsections, along with their implementations, using Python. Let us start with singly linked lists.

Singly linked lists

A linked list (also called a singly linked list) contains a number of nodes in which each node contains data and a pointer that links to the next node. The link of the last node in the list is `None`, which indicates the end of the list. Refer to the following linked list in *Figure 4.6*, in which a sequence of integers is stored.

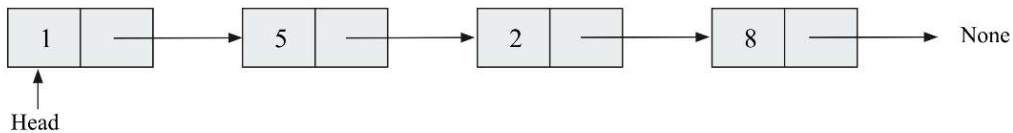


Figure 4.6: An example of a singly linked list

Next, we discuss how to create a singly linked list, and how to traverse it.

Creating and traversing

In order to implement the singly linked list, we can use the node class that we created in the previous section. For example, we create three nodes, `n1`, `n2`, and `n3`, that store three strings:

```
n1 = Node('eggs')
n2 = Node('ham')
n3 = Node('spam')
```

Next, we link the nodes sequentially to form the linked list. For example, in the following code, node `n1` is pointing to node `n2`, node `n2` is pointing to node `n3`, and node `n3` is the last node, and is pointing to **None**:

```
n1.next = n2
n2.next = n3
```

Traversal of the linked lists means visiting all the nodes of the list, from the starting node to the last node. The process of traversing the singly linked list begins with the first node, displaying the data of the current node, following the pointers, and finally stopping when we reach the last node.

To implement the traversal of the linked list, we start by setting the current variable to the first item (starting node) in the list, and then we traverse the complete list through a loop, traversing each node as shown in the following code:

```
current = n1
while current:
    print(current.data)
    current = current.next
```

In the loop, we print out the current element after which we set `current` to point to the next element in the list. We keep doing this until we reach the end of the list. The output of the preceding code for this example is:

```
eggs
ham
spam
```

There are, however, several problems with this simplistic list implementation:

- It requires too much manual work by the programmer
- Too much of the inner workings of the list is exposed to the programmer

So, let us discuss a better and more efficient way of traversing the linked list.

Improving list creation and traversal

As you will notice in the earlier example of the list traversal, we are exposing the node class to the client/user. However, the client node should not interact with the node object. We need to use `node.data` to get the contents of the node, and `node.next` to get the next node. We can access the data by creating a method that returns a generator, which can be done using the `yield` keyword in Python. The updated code snippet for list traversal is as follows:

```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val
```

Here, the `yield` keyword is used to return from a function while saving the states of its local variables to enable the function to resume from where it left off. Whenever the function is called again, the execution starts from the last `yield` statement. Any function that contains a `yield` keyword is termed a **generator**.

Now, list traversal is much simpler. We can completely ignore the fact that there is anything called a node outside of the list:

```
for word in words.iter():
    print(word)
```

Notice that since the `iter()` method yields the data member of the node, our client code doesn't need to worry about that at all.

A singly linked list can be created using a simple class to hold the list. We start with a constructor that holds a reference to the very first node in the list (that is head in the following code). Since this list is initially empty, we will start by setting this reference to `None`:

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None
```

In the preceding code, we start with an empty list that points to `None`. Now, new data elements can be appended/added to this list.

Appending items

The first operation that we need to perform is to append items to the list. This operation is also called an insertion operation. Here we get a chance to hide the `Node` class away. The user of the list class should never have to interact with `Node` objects.

Appending items to the end of a list

Let's have a look at the Python code for creating a linked list where we append new elements to the list using the `append()` method, as shown here:

The first shot at an `append()` method may look like this:

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0
    def append(self, data):
        # Encapsulate the data in a Node
        node = Node(data)
        if self.head is None:
            self.head = node
```

```

else:
    current = self.head
    while current.next:
        current = current.next
    current.next = node

```

Here, in this code, we encapsulate data in a node so that it has the next pointer attribute. From here, we check if there are any existing nodes in the list (that is, whether `self.head` points to a Node). If there is `None`, this means that initially, the list is empty and the new node will be the first node. So, we make the new node the first node of the list; otherwise, we find the insertion point by traversing the list to the last node and updating the next pointer of the last node to the new node. This working is depicted in *Figure 4.7*.

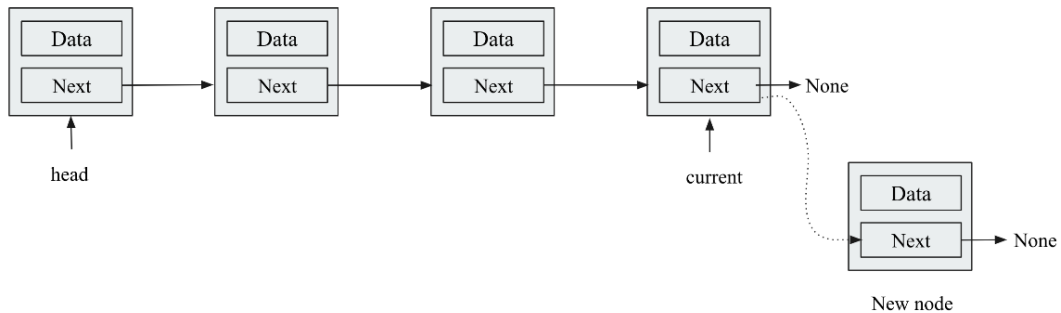


Figure 4.7: Inserting a node at the end of the list in a singly linked list

Consider the following example code to append three nodes:

```

words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

```

List traversal will work as we discussed before. You will get the first element of the list from the list itself, and then traverse the list through the next pointer:

```

current = words.head
while current:
    print(current.data)
    current = current.next

```

Still, this implementation is not very efficient, and there is a drawback with the append method. In this, we have to traverse the entire list to find the insertion point. This may not be a problem when there are just a few items in the list, but it will be very inefficient when the list is long, as it will have to traverse the whole list to add an item every time. Let us discuss a better implementation of the append method.

For this, the idea is that we not only have a reference to the first node in the list but also have one more variable in the node that references the last node of the list. That way, we can quickly append a new node at the end of the list. The worst-case running time of the append operation can be reduced from $O(n)$ to $O(1)$ using this method. We must ensure that the previous last node points to the new node that is to be appended to the list.

Here is our updated code:

```
class SinglyLinkedList:
    def __init__(self):
        self.tail = None
        self.head = None
        self.size = 0
    def append(self, data):
        node = Node(data)
        if self.tail:
            self.tail.next = node
            self.tail = node
        else:
            self.head = node
            self.tail = node
```



Take note of the convention being used. The point at which we append new nodes is through `self.tail`. The `self.head` variable points to the first node in the list.

In this code, a new node can be appended in the end through a `tail` pointer by making a link from the last node to the new node. *Figure 4.8* shows the workings of the preceding code.

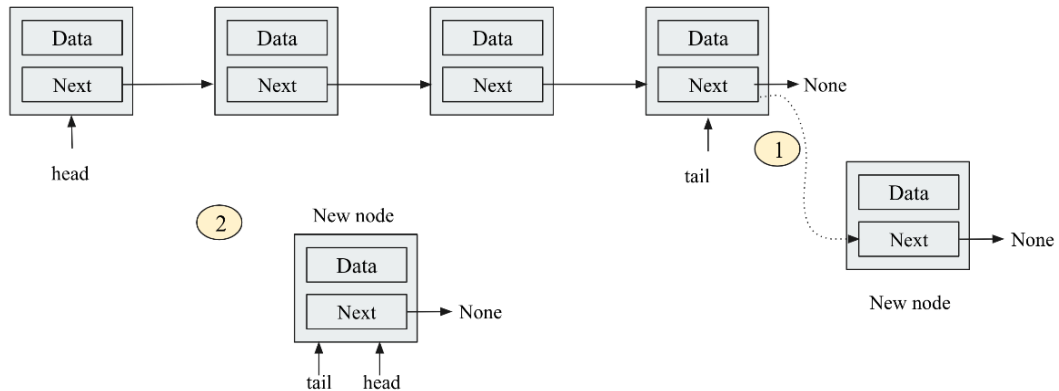


Figure 4.8: Illustrating the insertion of a node at the end of a linked list

In Figure 4.8, *step 1* shows the addition of the new node at the end, and *step 2* shows when the list is empty. In that case, `head` is made the new node, with `tail` pointing to that node.

Furthermore, the following code snippet shows the workings of the code:

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
egg
ham
spam
```

Appending items at intermediate positions

To append or insert an element in an existing linked list at a given position, firstly, we have to traverse the list to reach the desired position where we want to insert an element. An element can be inserted in between two successive nodes using two pointers (`prev` and `current`).

A new node can easily be inserted in between two existing nodes by updating these links, as shown in *Figure 4.9*.

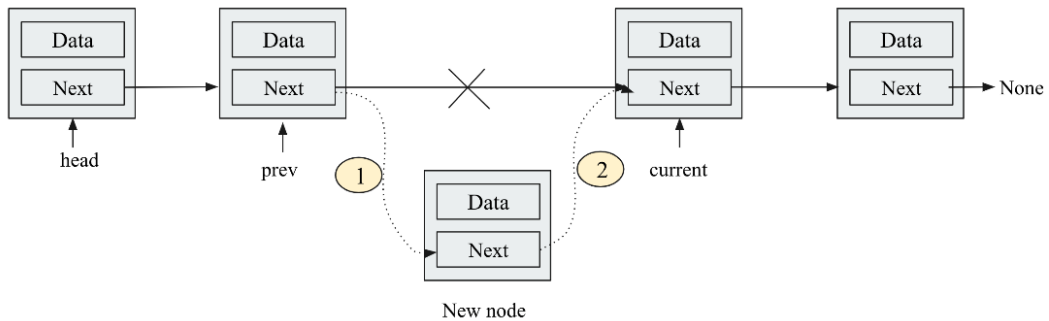


Figure 4.9: Insertion of a node between two successive nodes in a linked list

When we want to insert a node in between two existing nodes, all we have to do is update two links. The previous node points to the new node, and the new node should point to the successor of the previous node.

Let's look at the complete code below to add a new element at a given index position:

```
class SinglyLinkedList:
    def __init__(self):
        self.tail = None
        self.head = None
        self.size = 0

    def append_at_a_location(self, data, index):
        current = self.head
        prev = self.head
        node = Node(data)
        count = 1
        while current:
            if count == 1:
                node.next = current
                self.head = node
                print(count)
                return
            elif index == index:
                node.next = current
                prev.next = node
                return
            current = current.next
            prev = current
            count += 1
```

```
        count += 1
        prev = current
        current = current.next
    if count < index:
        print("The list has less number of elements")
```

In the preceding code, we start from the first node and move the current pointer to reach the index position where we want to add a new element, and then we update the node pointers accordingly. In the `if` condition, firstly, we check whether the index position is 1. In that case, we have to update the nodes as we are adding the new node at the start of the list. Therefore, we have to make the new node a head node. Next, in the `else` part, we check whether we have reached the required index position by comparing the value of `count` and `index`. If both values are equal, we add a new node in between nodes indicated by `prev` and `current` and update the pointers accordingly. Finally, we print an appropriate message if the required index position is greater than the length of the linked list.

The following code snippet uses the `append` method to add a “new” data element at an index position of 2 in the existing linked list:

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
current = words.head

while current:
    print(current.data)
    current = current.next

words.append_at_a_location('new', 2)
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
egg
new
ham
spam
```

It is important to note that the condition where we may want to insert a new element can change depending upon the requirement, so let's say we want to insert a new element just before an element that has the same data value. In that case, the code to `append_at_a_position` will be as follows:

```
def append_at_a_location(self, data):
    current = self.head
    prev = self.head
    node = Node(data)
    while current:
        if current.data == data:
            node.next = current
            prev.next = node
        prev = current
        current = current.next
```

We can now use the preceding code to insert a new node at an intermediate position:

```
words.append_at_a_location('ham')
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
egg
ham
ham
spam
```

The worst-case time complexity of the insert operation is $O(1)$ when we have an additional pointer that points to the last node. Otherwise, when we do not have the link to the last node, the time complexity will be $O(n)$ since we have to traverse the list to reach the desired position and in the worst case, we may have to traverse all the n nodes in the list.

Querying a list

Once the list is created, we may require some quick information about the linked list, such as the size of the list, and occasionally to establish whether a given data item is present in the list.

Searching an element in a list

We may also need to check whether a list contains a given item. This can be implemented using the `iter()` method, which we have already seen in the previous section while traversing the linked list. Using that, we write the search method as follows:

```
def search(self, data):
    for node in self.iter():
        if data == node:
            return True
    return False
```

In the above code, each pass of the loop compares the data to be searched with each data item in the list one by one. If a match is found, `True` is returned, otherwise `False` is returned.

If we run the following code for searching a given data item:

```
print(words.search('ssspam'))
print(words.search('spam'))
```

The output of the preceding code is as follows:

```
False
True
```

Getting the size of the list

It is important to get the size of the list by counting the number of nodes. One way to do it is by traversing the entire list and increasing the counter as we go along:

```
def size(self):
    count = 0
    current = self.head
    while current:
        count += 1
        current = current.next
    return count
```

The above code is very similar to what we did while traversing the linked list. Similarly, in this code, we traverse the nodes of the list one by one and increase the count variable. However, list traversal is potentially an expensive operation that we should avoid wherever we can.

So instead, we can opt for another method in which we can add a size member to the SinglyLinkedList class, initializing it to 0 in the constructor, as shown in the following code snippet:

```
class SinglyLinkedList:
    def __init__(self):
        self.head = data
        self.size = 0
```

Because we are now only reading the size attribute of the node object, and not using a loop to count the number of nodes in the list, we reduce the worst-case running time from $O(n)$ to $O(1)$.

Deleting items

Another common operation on a linked list is to delete nodes. There are three possibilities that we may encounter in order to delete a node from the singly linked list.

Deleting the node at the beginning of the singly linked list

Deleting a node from the beginning is quite easy. It involves updating the head pointer to the second node in the list. This can be done in two steps:

1. A temporary pointer (current pointer) is created that points to the first node (head node), as shown in *Figure 4.10*.

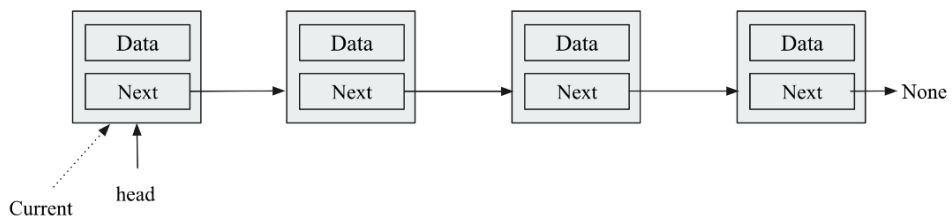


Figure 4.10: Illustration of the deletion of the first node from the linked list

2. Next, the current node pointer is moved to the next node and assigned to the head node. Now, the second node becomes the head node that is pointed to by the head pointer, as shown in *Figure 4.11*.

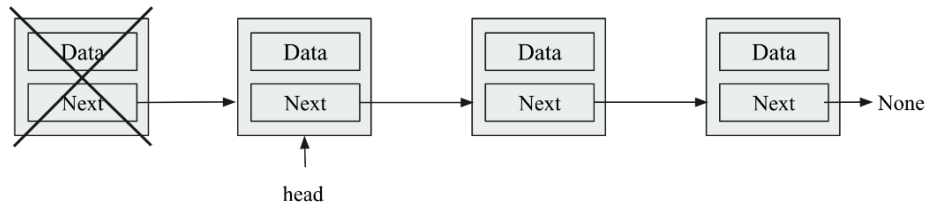


Figure 4.11: After deleting the first node, the head pointer now points to the new starting element

This can be implemented using the following Python code. In this code, initially, three data elements are added as we have done previously, and then the first node of the list is deleted:

```
def delete_first_node (self):
    current = self.head
    if self.head is None:
        print("No data element to delete")
    elif current == self.head:
        self.head = current.next
```

In the above code, we initially check if there is no item to delete from the list, and we print the appropriate message. Next, if there is some data item in the list, we assign the head pointer to the temporary pointer `current` as per *step 1*, and then the head pointer is now pointing to the next node, assuming that we already have a linked list of three data items – “eggs”, “ham”, and “spam”:

```
words.delete_first_node()
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the preceding code is as follows:

```
ham
spam
```

Deleting the node at the end in the singly linked list

To delete the last node from the list, we have to first traverse the list to reach the last node. At that time, we also need an extra pointer that points to just one node before the last node, so that after deleting the last node, the second last node can be marked as the last node. It can be implemented in the following three steps:

1. Firstly, we have two pointers, in other words, a current pointer that will point to the last node, and a prev pointer that will point to the node previous to the last node (second last node). Initially, we will have three pointers (current, prev, and head) pointing to the first node, as shown in *Figure 4.12*.

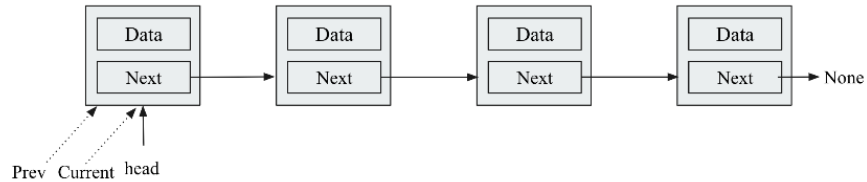


Figure 4.12: Illustration of the deletion of the end node from the linked list

2. To reach the last node, we move the current and prev pointers in such a way that the current pointer should point to the last node and the prev pointer should point to the second last node. So, we stop when the current pointer reaches the last node. This is shown in *Figure 4.13*.

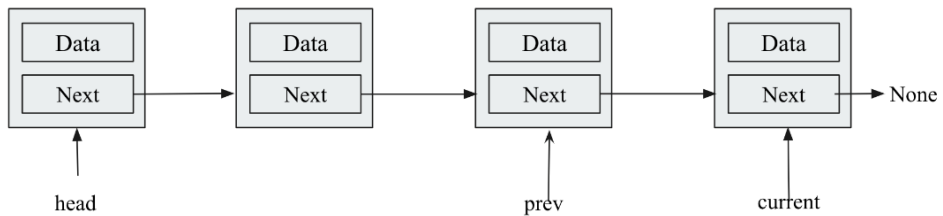


Figure 4.13: Traversal of the linked list to reach the end of the list

3. Finally, we mark the prev pointer to point to the second last node, which is rendered as the last node of the list by pointing this node to None, as shown in *Figure 4.14*.

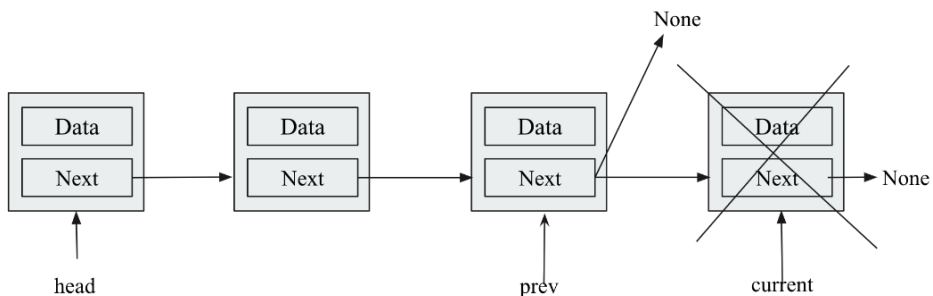


Figure 4.14: Deletion of the last node from the linked list

The implementation in Python for deleting a node from the end of the list is as follows:

```
def delete_last_node (self):
    current = self.head
    prev = self.head
    while current:
        if current.next is None:
            prev.next = current.next
            self.size -= 1
        prev = current
        current = current.next
```

In the preceding code, firstly, the current and prev pointers are assigned the head pointer as per *step 1*. Then, in the while loop, we check whether we reached the end of the list using the `current.next is None` condition. Once we reach the end of the list, we make the second last node, which is indicated by the prev pointer, the last node. We also decrement the size of the list. If we do not reach the end of the list, we increment the prev and current pointers in the while loop in the last two lines of code. Next, let us discuss how to delete any intermediate node in a singly linked list.

Deleting any intermediate node in a singly linked list

We first have to decide how to select a node for deletion. Identifying the intermediate node to be deleted can be determined by the index number or by the data the node contains. Let us understand this concept by deleting a node depending on the data it contains.

To delete any intermediary node, we need two pointers similar to the case when we learned to delete the last node; in other words, the current pointer and the prev pointer. Once we reach the node that is to be deleted, the desired node can be deleted by making the previous node point to the next node of the node that is to be deleted. The process is provided in the following steps:

1. *Figure 4.15* shows when an intermediate node is deleted from the given linked list. In this, we can see that the initial pointers point to the first node.

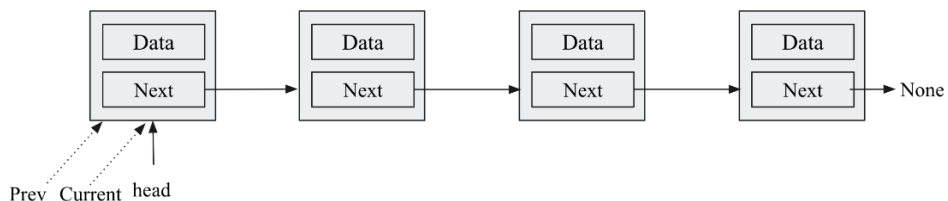


Figure 4.15: Illustration of the deletion of an intermediate node from the linked list

2. Once the node is identified, the prev pointer is updated to delete the node, as shown in *Figure 4.16*. The node to be deleted is shown along with the link to those to be updated in *Figure 4.16*.

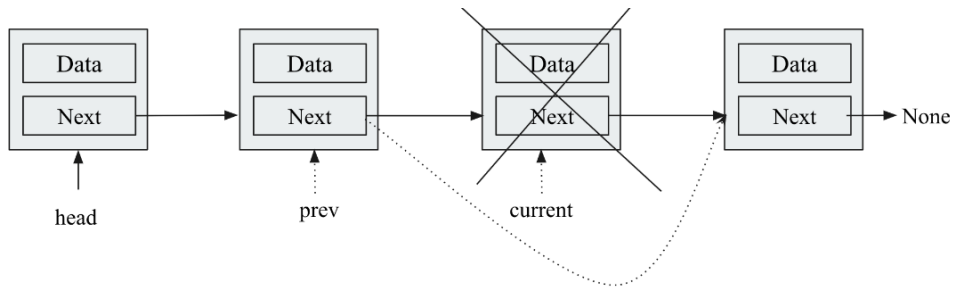


Figure 4.16: Traversing to reach the intermediate node that is to be deleted in the linked list

3. Finally, the list after deleting the node is shown in *Figure 4.17*.

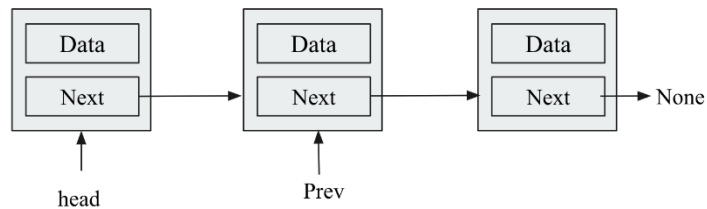


Figure 4.17: Deletion of an intermediate node from the linked list

Let's say we want to delete a data element that has the given value. For this given condition, we can first search the node to be deleted and then delete the node as per the steps discussed.

Here is what the implementation of the `delete()` method may look like:

```
def delete(self, data):
    current = self.head
    prev = self.head
    while current:
        if current.data == data:
            if current == self.head:
                self.head = current.next
            else:
                prev.next = current.next
            self.size -= 1
```

```
        return
    prev = current
    current = current.next
```

Assuming that we already have a linked list of three items – “eggs”, “ham”, and “spam”, the following code is for executing the delete operation, that is, deleting a data element with the value “ham” from the given linked list:

```
words.delete("ham")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the preceding code is as follows:

```
egg
spam
```

The worst-case time complexity of the delete operation is $O(n)$ since we have to traverse the list to reach the desired position and, in the worst-case scenario, we may have to traverse all the n nodes in the list.

Clearing a list

We may need to clear a list quickly, and there is a very simple way to do this. We can clear a list by simply clearing the pointer head and tail by setting them to None:

```
def clear(self):
    # clear the entire list.
    self.tail = None
    self.head = None
```

In the above code, we can clear the list by assigning None to the tail and head pointers.

We have discussed different operations for a singly linked list, and now we will discuss the concept of doubly linked list and learn how different operations can be implemented in a doubly linked list in the next section.

Doubly linked lists

A doubly linked list is quite similar to the singly linked list in the sense that we use the same fundamental concept of nodes along with how we can store data and links together, as we did in a singly linked list. The only difference between a singly linked list and a doubly linked list is that in a singly linked list, there is only one link between each successive node, whereas, in a doubly linked list, we have two pointers—a pointer to the next node and a pointer to the previous node. See the following *Figure 4.18* of a node; there is a pointer to the next node and the previous node, which are set to None as there is no node attached to this node.

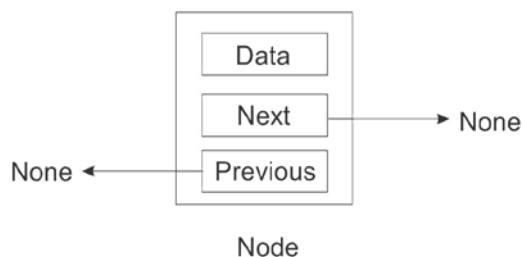


Figure 4.18: Represents a doubly linked list with a single node

A node in a singly linked list can only determine the next node associated with it. However, there is no link to go back from this referenced node. The direction of flow is only one way. In a doubly linked list, we solve this issue and include the ability not only to reference the next node, but also to reference the previous node. Consider the following *Figure 4.19* to understand the nature of the linkages between two successive nodes. Here, node **A** is referencing node **B**; in addition, there is also a link back to node **A**.

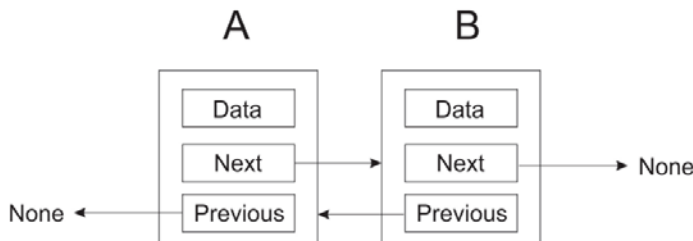


Figure 4.19: Doubly linked list with two nodes

Doubly linked lists can be traversed in any direction. A node in a doubly linked list can be easily referred to by its previous node whenever required without having a variable to keep track of that node.

However, in a singly linked list, it may be difficult to move back to the start or beginning of the list to make some changes at the start of the list, which is very easy now in the case of a doubly linked list.

Creating and traversing

The Python code to create a doubly linked list node includes its initializing methods, the prev pointer, the next pointer, and the data instance variables. When a node is newly created, all these variables default to None:

```
class Node:
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
```

The prev variable has a reference to the previous node, while the next variable keeps the reference to the next node, and the data variable stores the data.

Next, let's create a doubly linked list class.

The doubly linked list class has two pointers, head and tail, that will point to the start and end of the doubly linked list, respectively. In addition, for the size of the list, we set the count instance variable to 0. It can be used to keep track of the number of items in the linked list. Consider the following Python code for creating a doubly linked list class:

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0
```



Here, self.head points to the beginner node of the list, and self.tail points to the last node. However, there are no fixed rules as to the naming of the head and tail node pointers.

Doubly linked lists also require functionalities that return the size of the list, insert items into the list, and delete nodes from the list. Next, we discuss different operations that can be applied to the doubly linked list. Let's start with the append operation.

Appending items

The append operation is used to add an element at the end of a list. An element can be appended or inserted into a doubly linked list in the following instances.

Inserting a node at beginning of the list

Firstly, it is important to check whether the head node of the list is None. If it is None, this means that the list is empty, otherwise the list has some nodes, and a new node can be appended to the list. If a new node is to be added to the empty list, it should have the head pointer pointing to the newly created node, and the tail of the list should also point to this newly created node.

The following *Figure 4.20* illustrates the head and tail pointers of the doubly linked list when a new node is added to an empty list.

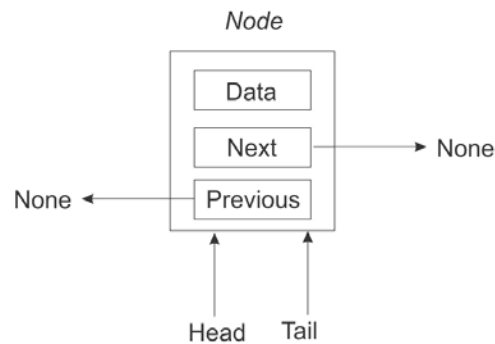


Figure 4.20: Illustration of inserting a node in an empty doubly linked list

Alternatively, we can insert or append a new node at the beginning of an existing doubly linked list, as shown in *Figure 4.21*.

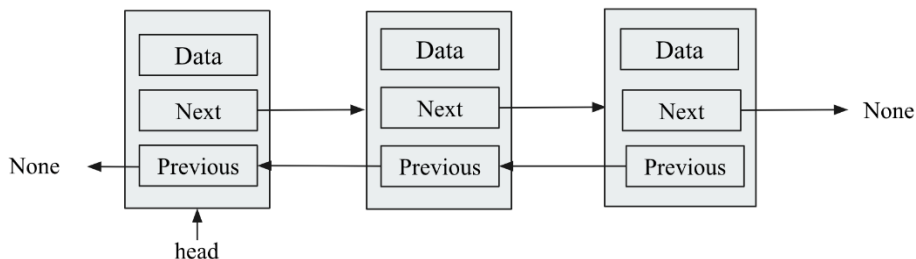


Figure 4.21: Illustration of inserting an element in a doubly linked list

The new node should be made as a new starting node of the list and that should now point to the previous head node.

It can be done by updating the three links, which are also shown with dotted lines in *Figure 4.22* and described as follows:

1. Firstly, the next pointer of a new node should point to the head node of the existing list
2. The prev pointer of the head node of the existing list should point to the new node
3. Finally, mark the new node as the head node in the list

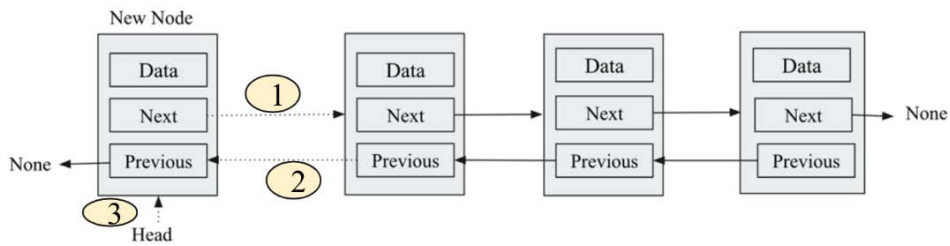


Figure 4.22: Inserting a node at the beginning of the doubly linked list

The following code is used to append/insert an item at the beginning when the list is initially empty and with an existing doubly linked list:

```
def append_at_start(self, data):
    #Append an item at beginning to the list.
    new_node = Node(data, None, None)
    if self.head is None:
        self.head = new_node
        self.tail = self.head
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node
    self.count += 1
```

In the above code, firstly, the `self.head` condition is checked irrespective of whether the list is empty. If it is empty, then the head and tail pointers point to the newly created node. In this case, the new node becomes the head node. Next, if the condition is not true, this means the list is not empty, and a new node has to be added at the beginning of the list. For this, three links are updated as shown in *Figure 4.22*, and also shown in the code in bold font. After updating these three links, finally, the size of the list is increased by 1. Furthermore, let us understand how to insert an element at the end of the doubly linked list.

Further, the following code snippet shows how we can create a double link list and append a new node at the starting of the list:

```
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

print("Items in doubly linked list before append:")
current = words.head
while current:
    print(current.data)
    current = current.next
words.append_at_start('book')

print("Items in doubly linked list after append:")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is:

```
Items in doubly linked list before append:
egg
ham
spam
Items in doubly linked list after append:
book
egg
ham
spam
```

In the output, we can see that the new data item “book” is added in the starting of the list.

Inserting a node at the end of the list

To append/insert a new element at the end of the doubly linked list, we will need to traverse through the list to reach the end of the list if we do not have a separate pointer pointing to the end of the list. Here, we have a *tail* pointer that points to the end of the list.

A visual representation of the append operation to an existing list is shown in the following *Figure 4.23*.

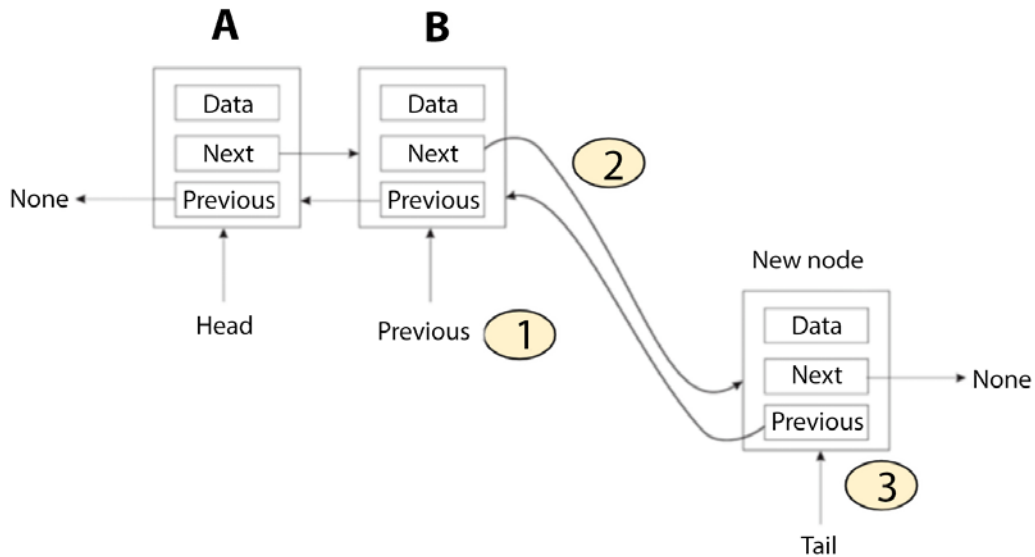


Figure 4.23: Inserting a node at the end of the list in a doubly linked list

To add a new node at the end, we update two links as follows:

1. Make the prev pointer of the new node point to the previous tail node
2. Make the previous tail node point to the new node
3. Finally, update the tail pointer so that the tail pointer now points to the new node

The following code is used to append an item at the end of the doubly linked list:

```
def append(self, data):
    #Append an item at the end of the List.
```

```

new_node = Node(data, None, None)
if self.head is None:
    self.head = new_node
    self.tail = self.head
else:
    new_node.prev = self.tail
    self.tail.next = new_node
    self.tail = new_node
self.count += 1

```

In the above code, the if part of the preceding program is for adding a node to the empty list; the else part of the preceding program will be executed if the list is not empty. If the new node is to be added to a list, the new node's previous variable is to be set to the tail of the list:

```
new_node.prev = self.tail
```

The tail's next pointer (or variable) has to be set to the new node:

```
self.tail.next = new_node
```

Lastly, we update the tail pointer to point to the new node:

```
self.tail = new_node
```

Since an append operation increases the number of nodes by one, we increase the counter by one:

```
self.count += 1
```

The following code snippet can be used to append a node at the end of the list:

```

print("Items in doubly linked list after append")
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

words.append('book')
print("Items in doubly linked list after adding element at end.")
current = words.head
while current:
    print(current.data)
    current = current.next

```

The output of the above code:

```
Items in doubly linked list after adding element at end.  
egg  
ham  
spam  
book
```

The worst-case time complexity of appending an element to the doubly linked list is $O(1)$ since we already have the tail pointer that points to the end of the list, and we can directly add a new element. Next, we will discuss how to insert a node at an intermediate position of the doubly linked list.

Inserting a node at an intermediate position in the list

Inserting a new node at any given position in a doubly linked list is similar to what we discussed in a singly linked list. Let us take an example in which we insert a new element just before the element that has the same data value as the given data.

Firstly, we traverse to the position where we want to insert a new element in that situation. The current pointer points to the target node, while the prev pointer just points to the previous node of the target node, as shown in *Figure 4.24*.

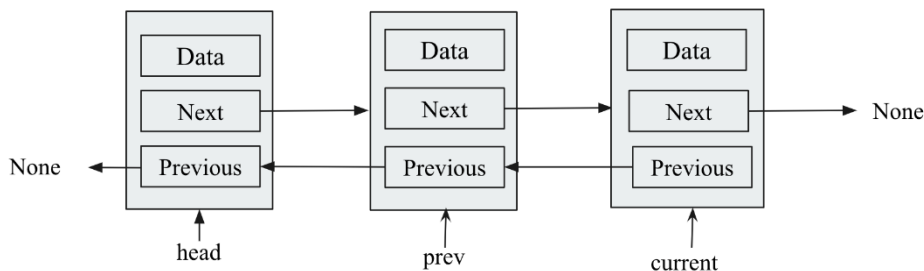


Figure 4.24: Illustration of pointers for inserting a node at an intermediate position in a doubly linked list

After reaching the correct position, a few pointers have to be added in order to add a new node. The details of these links that need to be updated (also shown in *Figure 4.25*) are as follows:

1. The next pointer of the new node points to the current node
2. The prev pointer of the new node should point to the previous node
3. The next pointer of the previous node should point to the new node

4. The prev pointer of the current node should point to the new node

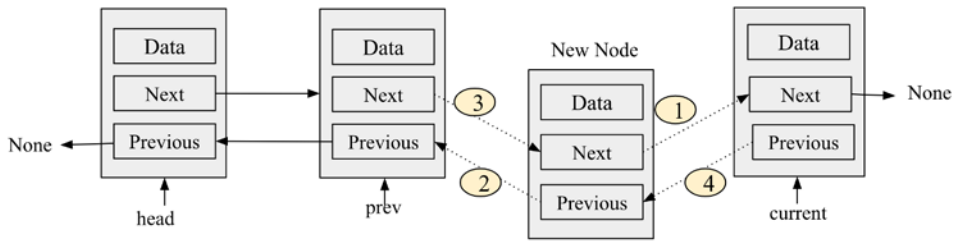


Figure 4.25: Demonstration of links that need to be updated in order to add a new node at any intermediate position in the list

Here is what the implementation of the `append_at_a_location()` method may look like:

```
def append_at_a_location(self, data):
    current = self.head
    prev = self.head
    new_node = Node(data, None, None)
    while current:
        if current.data == data:
            new_node.prev = prev
            new_node.next = current
            prev.next = new_node
            current.prev = new_node
            self.count += 1
        prev = current
        current = current.next
```

In the preceding code, firstly, the `current` and `prev` pointers are initialized by pointing to the head node. Then, in the `while` loop, we first reach the desired position by checking the condition. In this example, we check the `data` value of the `current` node against the `data` value provided by the user. Once we reach the desired position, we update four links as discussed, which are also shown in *Figure 4.25*.

The following code snippet can be used to insert an data element “ham” after the first occurrence of the word “ham” in the doubly linked list:

```
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
```

```
words.append('spam')

words.append_at_a_location('ham')

print("Doubly linked list after adding an element after word \"ham\" in
the list.")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code:

```
Doubly linked list after adding an element after word "ham" in the list.
egg
ham
ham
spam
```

Appending at the start and end positions in a doubly linked list will have a worst-case running time complexity of $O(1)$ since we can directly append the new node, and the worst-case time complexity for appending a new node at any intermediate position will be $O(n)$ since we may have to traverse the list of n items.

Next, let us learn how to search a given item if that is present in the doubly linked list or not.

Querying a list

The search for an item in a doubly linked list is similar to the way we did it in the singly linked list. We use the `iter()` method to check the data in all the nodes. As we run a loop through all the data in the list, each node is matched with the data passed in the `contain` method. If we find the item in the list, `True` is returned, denoting that the item is found, otherwise `False` is returned, which means the item was not found in the list. The Python code for this is as follows:

```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val
```

```
def contains(self, data):
    for node_data in self.iter():
        if data == node_data:
            print("Data item is present in the list.")
            return
    print("Data item is not present in the list.")
    return
```

The following code can be used to search if a data item is present in the existing doubly linked list:

```
words = DoublyLinkedList()

words.append('egg')
words.append('ham')
words.append('spam')

words.contains("ham")
words.contains("ham2")
```

The output of the above code is as follows:

```
Data item is present in the list.
Data item is not present in the list.
```

The search operation in a doubly linked list has a running time complexity of $O(n)$ since we have to traverse the list in order to reach the desired element and, in the worst case, we may have to traverse the whole list of n items.

Deleting items

The deletion operation is easier in the doubly linked list compared to the singly linked list. Unlike in a singly linked list, where we need to traverse the linked list to reach the desired position, and we also need one more pointer to keep track of the previous node of the target node, in a doubly linked list, we don't have to do that because we can traverse in both directions.

The delete operation in a doubly linked list can have four scenarios, which are discussed as follows:

1. The item to be deleted is located at the start of the list
2. The item to be deleted is found at the tail end of the list

3. The item to be deleted is located anywhere at an intermediate position in the list
4. The item to be deleted is not found in the list

The node to be deleted is identified by matching the data instance variable with the data that is passed to the method. If the data matches the data variable of a node, that matching node will be deleted:

1. For the first scenario, when we have found the item to be deleted at the first position, we will have to simply update the head pointer to the next node. It is shown in *Figure 4.26*.

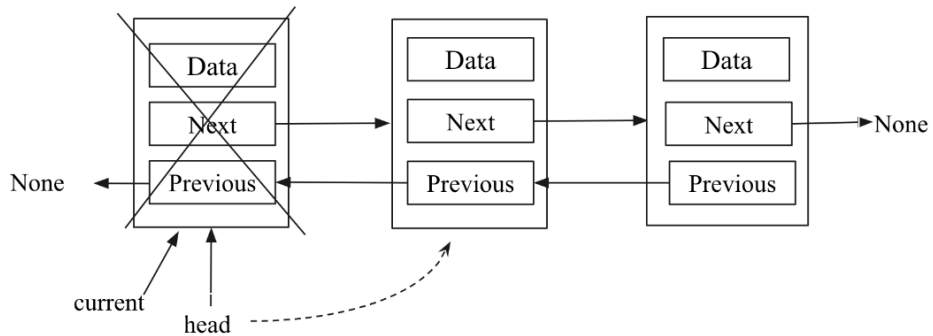


Figure 4.26: Illustration of the deletion of the first node in a doubly linked list

2. For the second scenario, when we found the item to be deleted at the last position in the list, we will have to simply update the tail pointer to the second last node. It is shown in *Figure 4.27*.

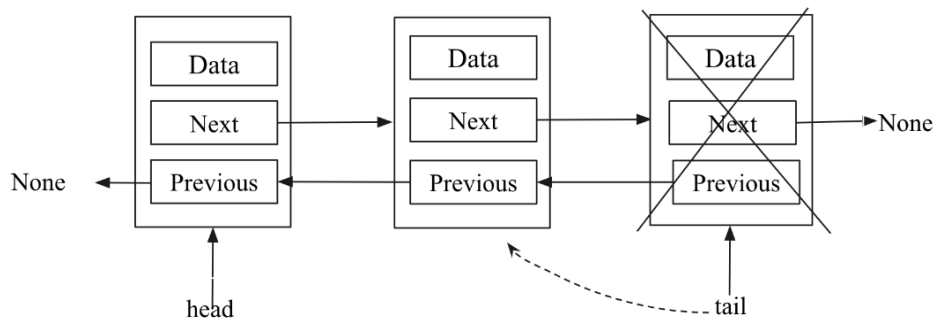


Figure 4.27: Illustration of the deletion of the last node in a doubly linked list

3. For the third scenario, we found the data item to be deleted at any intermediate position. To better understand this, consider the example shown in *Figure 4.28*. In this, there are three nodes, A, B, and C. To delete node B in the middle of the list, we will essentially make A point to node C as its next node, while making C point to A as its previous node.

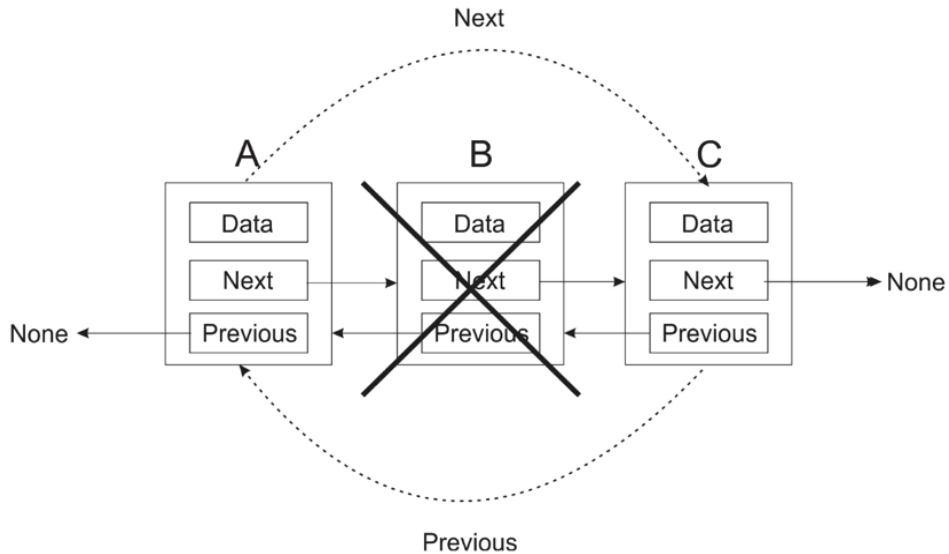


Figure 4.28: Illustration of the deletion of the intermediate node B from the doubly linked list

The complete implementation to delete a node from the doubly linked list in Python is as follows. We'll discuss each part of this code step by step:

```
def delete(self, data):
    # Delete a node from the list.
    current = self.head
    node_deleted = False
    if current is None:
        # List is empty
        print("List is empty")
    elif current.data == data:
        # Item to be deleted is found at starting of the list
        self.head.prev = None
        node_deleted = True
        self.head = current.next
    elif self.tail.data == data:
```

```

        #Item to be deleted is found at the end of list
        self.tail = self.tail.prev
        self.tail.next = None
        node_deleted = True
    else:
        while current:
            #search item to be deleted, and delete that node
            if current.data == data:
                current.prev.next = current.next
                current.next.prev = current.prev
                node_deleted = True
                current = current.next
            if node_deleted == False:
                # Item to be deleted is not found in the list
                print("Item not found")
        if node_deleted:
            self.count -= 1

```

Initially, we create a `node_deleted` variable to denote the deleted node in the list and this is initialized to `False`. The `node_deleted` variable is set to `True` if a matching node is found and subsequently removed.

In the `delete` method, the `current` variable is initially set to the head node of the list (that is, it points to the `self.head` node of the list). This is shown in the following code fragment:

```

def delete(self, data):
    current = self.head
    node_deleted = False

```

Next, we use a set of `if...else` statements to search various parts of the list to ascertain the node with the specified data that is to be deleted.

First of all, we search for the data to be deleted at the head node, and if the data is matched at the head node, this node would be deleted. Since `current` is pointing at head, if `current` is `None`, this means that the list is empty and has no nodes to find the node to be deleted. The following is its code fragment:

```

if current is None:
    node_deleted = False

```

However, if `current` (which now points to `head`) contains the data being searched for, this means that we found the data to be deleted at the head node, and `self.head` is then marked to point to the `current.next` node. Since there is now no node behind `head`, `self.head.prev` is set to `None`. Consider the following code snippet for this:

```
elif current.data == data:
    self.head.prev = None
    node_deleted = True
    self.head = current.next
```

Similarly, if the node that is to be deleted is found at the tail end of the list, we delete the last node by setting its previous node pointing to `None`. `self.tail` is set to point to `self.tail.prev`, and `self.tail.next` is set to `None` as there is no node afterward. Consider the following code fragment for this:

```
elif self.tail.data == data:
    self.tail = self.tail.prev
    self.tail.next = None
    node_deleted = True
```

Lastly, we search for the node to be deleted by looping through the entire list of nodes. If the data that is to be deleted is matched with a node, that node will be deleted.

To delete a node, we make the previous node of the current node point to the next node using the `current.prev.next = current.next` code. After that step, we make the current's next node point to the previous node of the current node using `current.next.prev = current.prev`. Furthermore, if we traverse the complete list, and the desired item is not found, we print the appropriate message. Consider the following code snippet for this:

```
else:
    while current:
        if current.data == data:
            current.prev.next = current.next
            current.next.prev = current.prev
            node_deleted = True
            current = current.next
        if node_deleted == False:
            # Item to be deleted is not found in the List
            print("Item not found")
```

Finally, the `node_delete` variable is then checked to ascertain whether a node is actually deleted. If any node is deleted, then we decrease the count variable by 1, and this keeps track of the total number of nodes in the list. See the following code fragment:

```
if node_deleted:
    self.count -= 1
```

This decrements the count variable by 1 in case any node is deleted.

Let's take an example to see how the delete operation works with the same example of adding three strings – “egg”, “ham”, and “spam”, and then a node with the value “ham” is deleted from the list. The code is as follows:

```
#Code to create for a doubly linked list
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

words.delete('ham')
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the preceding code is as follows:

```
egg
spam
```

The worst-case running time complexity of the delete operation is $O(n)$ since we may have to traverse the list of n items to search for the item to be deleted.

In the next section, we will learn different operations on a circular linked list.

Circular lists

A circular linked list is a special case of a linked list. In a circular linked list, the endpoints are connected, which means that the last node in the list points back to the first node. In other words, we can say that in circular linked lists, all the nodes point to the next node (and the previous node in the case of a doubly linked list) and there is no end node, meaning no node will point to `None`.

The circular linked lists can be based on both singly and doubly linked lists. Consider *Figure 4.29* for the circular linked list based on a singly linked list where the last node, C, is again connected to the first node A, thus making a circular list.

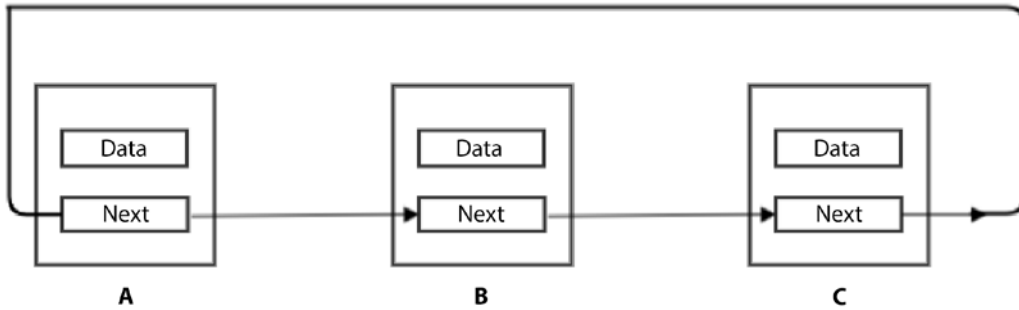


Figure 4.29: Example of a circular list based on a singly linked list

In the case of a doubly linked circular list, the first node points to the last node, and the last node points back to the first node. *Figure 4.30* shows the concept of the circular linked list based on a doubly linked list where the last node C is again connected to the first node A through the next pointer. Node A is also connected to node C through the previous pointer, thus making a circular list.

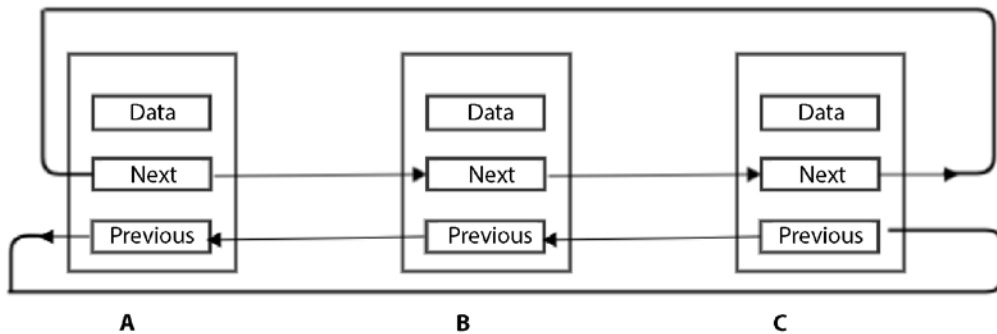


Figure 4.30: Example of a circular list based on a doubly linked list

Now, we are going to look at an implementation of a singly linked circular list. It is very straightforward to implement a doubly linked circular list once we understand the basic concepts of singly and doubly linked lists.

Almost everything is similar except that we should be careful in managing the link of the last node to the first node.

We can reuse the node class that we created in the singly linked lists subsection. We can reuse most parts of the `SinglyLinkedList` class as well. So, we are going to focus on where the circular list implementation differs from the normal singly linked list.

Creating and traversing

The circular linked list class can be created using the following code:

```
class CircularList:
    def __init__(self):
        self.tail = None
        self.head = None
        self.size = 0
```

In the above code, initially in the circular linked list class, we have two pointers; `self.tail` is used to point to the last node, and `self.head` is used to point to the first node of the list.

Appending items

Here, we want to add a node at the end of a circular linked list, as shown in *Figure 4.31*, in which we have four nodes, wherein the head is pointing to the starting node and the tail is pointing to the last node.

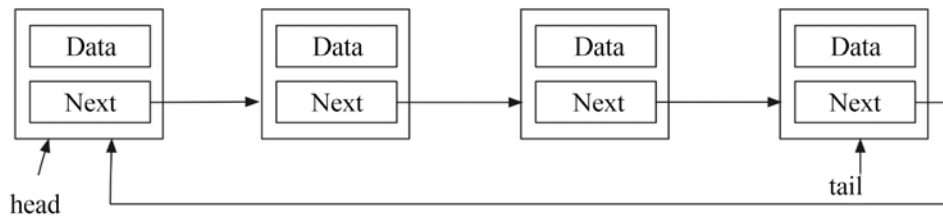


Figure 4.31: Example of a circular linked list for adding a node at the end

Figure 4.32 shows how a node is added to a circular linked list.

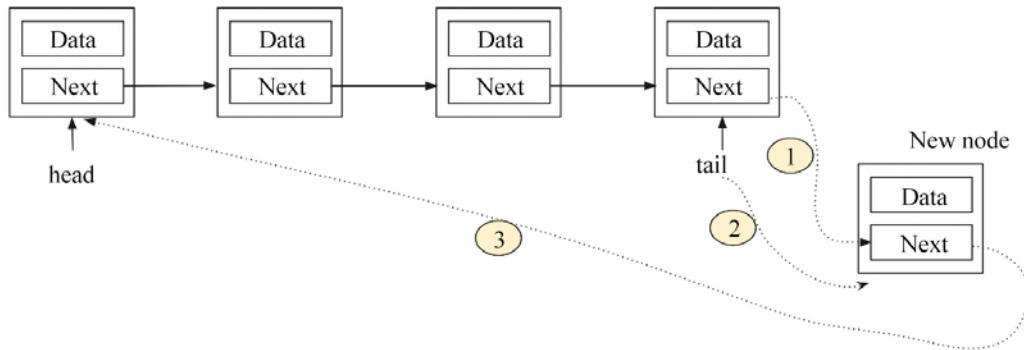


Figure 4.32: Inserting a node at the end of the singly circular list

To add a node at the end, we will update three links:

1. The next pointer of the last node to point to a new node
2. The next pointer of a new node to point to the head node
3. Update the tail pointer to point to the new node

The implementation of the circular linked list to append an element at the end of the circular list based on a singly linked list is as follows:

```
def append(self, data):
    node = Node(data)
    if self.tail:
        self.tail.next = node
        self.tail = node
        node.next = self.head
    else:
        self.head = node
        self.tail = node
        self.tail.next = self.tail
    self.size += 1
```

In the above code, firstly, we check whether the list is empty. If the list is empty, we go to the else part of the above code. In this case, the new node will be the first node of the list, and both the head and tail pointers will point to the new node, while the next pointer of the new node will again point to the new node.

Otherwise, if the list is not empty, we go to the `if` part of the preceding code. In this case, we update the three pointers as shown in *Figure 4.32*. This is similar to what we did in the case of the single linked list. Only one link is additionally added in this case, which is shown in bold font in the preceding code.

Further, we can use `iter()` method to traverse all the elements of the list. The `iter()` method described below should be defined in `CircularList` class:

```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
    yield val
```

The below code can be used to create a singly circular linked list, and then print all the data elements of the list, and then we stop when the counter becomes 3 which is the length of the list.

```
words = CircularList()
words.append('eggs')
words.append('ham')
words.append('spam')
```

```
counter = 0
for word in words.iter():
    print(word)
    counter += 1
    if counter > 2:
        break
```

The output of the preceding code is as follows:

```
eggs
ham
spam
```

Appending any element at an intermediate position in a circular list is exactly to its implementation in a singly linked list.

Querying a list

Traversing a circular linked list is very convenient as we don't need to look for the starting point. We can start anywhere, and we just need to carefully stop traversing when we reach the same node again. We can use the same `iter()` method, which we discussed at the start of this chapter. This will also be the case for the circular list; the only difference is that we have to mention an exit condition when we are iterating through the circular list, otherwise the program will get stuck in a loop, and it will run indefinitely. We can make any exit condition dependent upon our requirements; for example, we can use a counter variable. Consider the following example code:

```
words = CircularList()
words.append('eggs')
words.append('ham')
words.append('spam')
counter = 0

for word in words.iter():
    print(word)
    counter += 1
    if counter > 100:
        break
```

In the above code, we add three strings of data to the circular linked list, and then we print the data values iterating through the list 100 times.

In the next section, let us understand how the delete operation works in a circular linked list.

Deleting an element in a circular list

To delete a node in a circular list, it looks like we can do it similarly to how we did in the case of the append operation—simply make sure that the last node through the `tail` pointer points back to the starting node of the list through the `head` pointer. We have the following three scenarios:

1. When the item to be deleted is the head node:

In this scenario, we have to ensure that we make the second node of the list the new head node (shown as *step 1* in *Figure 4.33*), and the last node should be pointing back to the new head (shown as *step 2* in *Figure 4.33*).

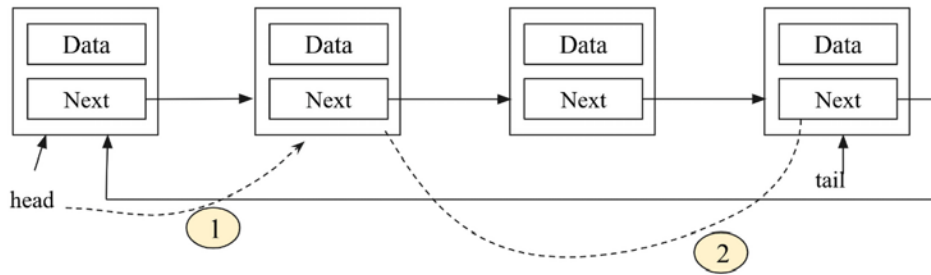


Figure 4.33: Deletion of a starting node in a singly circular list

2. When the item to be deleted is the last node:

In this scenario, we have to ensure that we make the second last node the new tail node (shown as *step 1* in Figure 4.34), while the new tail node should be pointing back to the new head (shown as *step 2* in Figure 4.34).

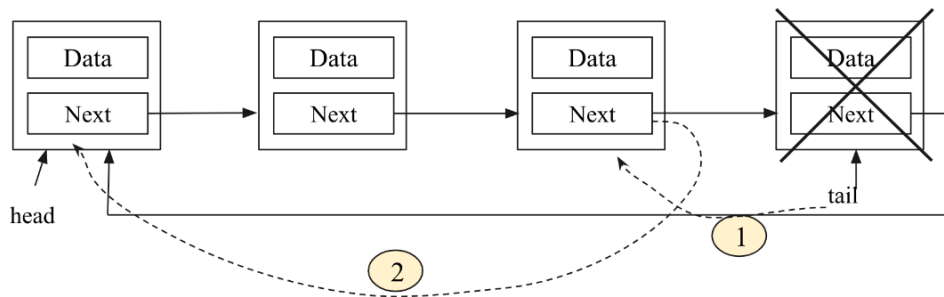


Figure 4.34: Deletion of the last node in a singly circular list

3. When the item to be deleted is an intermediate node:

This is very similar to what we did in the singly linked list. We have to make a link from the previous node of the target node to the next node of the target node, as shown in Figure 4.35.

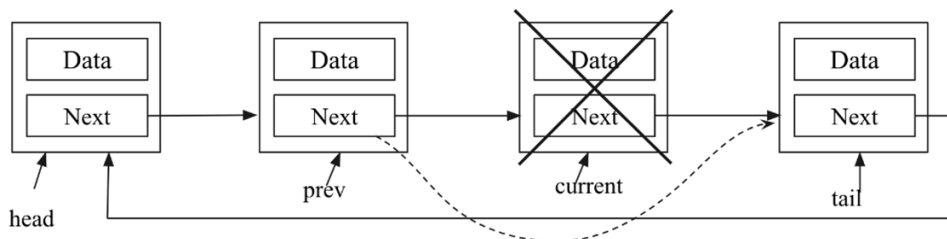


Figure 4.35: Deletion of any intermediate node in a singly circular list

The implementation of the delete operation is as follows:

```
def delete(self, data):
    current = self.head
    prev = self.head
    while prev == current or prev != self.tail:
        if current.data == data:
            if current == self.head:
                #item to be deleted is head node
                self.head = current.next
                self.tail.next = self.head
            elif current == self.tail:
                #item to be deleted is tail node
                self.tail = prev
                prev.next = self.head
            else:
                #item to be deleted is an intermediate node
                prev.next = current.next
        self.size -= 1
    return
    prev = current
    current = current.next
    if flag is False:
        print("Item not present in the list")
```

In the preceding code, firstly, iterate over all the elements to search the desired element to be deleted. Here, it is important to note the stopping condition. If we simply check the current pointer to be equal to None (which we did in the singly linked list), the program will go into an indefinite loop since the current node will never point to None in the case of circular linked lists.

For this, we cannot check whether current has reached tail because then it will never check the last node. So, the stopping criterion in the circular list is the fact that the prev and current pointers point to the same node. It will work fine except on one occasion when the first loop iteration, at that time, current and prev, will point to the same node, in other words, the head node.

Once, we enter the loop, we check the data value of the current pointer with the given data value to get the node to be deleted. We check whether the node to be deleted is the head node, tail node, or intermediate node, and then update the appropriate links shown in *Figures 4.33, 4.34, and 4.35*.

So, we have discussed the different scenarios while deleting any node in singly circular linked list, similarly, the doubly linked list based circular linked list can be implemented.

The following code can be used to create a circular linked list, and apply different delete operations:

```
words = CircularList()
words.append('eggs')
words.append('ham')
words.append('spam')
words.append('foo')
words.append('bar')

print("Let us try to delete something that isn't in the list.")
words.delete('socks')
counter = 0
for item in words.iter():
    print(item)
    counter += 1
    if counter > 4:
        break

print("Let us delete something that is there.")
words.delete('foo')
counter = 0
for item in words.iter():
    print(item)
    counter += 1
    if counter > 3:
        break
```

The output of the above code is as follows:

```
Let us try to delete something that isn't in the list.
Item not present in the list
eggs
ham
spam
foo
bar
```

```
Let us delete something that is there.  
eggs  
ham  
spam  
bar
```

The worst-case time complexity of inserting an element at a given location in the circular linked list is $O(n)$ since we have to traverse the list to the desired location. The complexity of insertion at the first and last locations of the circular list will be $O(1)$. Similarly, the worst-case time complexity to delete an element at a given location is $O(n)$.

So far, we have discussed the different scenarios while deleting any node in a singly circular linked list. Similarly, the doubly linked list can be implemented based on a circular linked list.

In a singly linked list, the traversal of nodes can be done in one direction, whereas, in a doubly linked list, it is possible to traverse in both directions (forward and backward). In both cases, the complexity of the insertion and deletion operations at a given location is $O(n)$ whenever we have to traverse the list in order to reach the desired location where we want to insert or delete any element. Similarly, the worst-case time complexity of the insertion or deletion of a node for a given desired location is $O(n)$. Whenever we need to save memory space, we should use a singly linked list since it only needs one pointer, whereas a doubly linked list takes more memory space to store double pointers. When a search operation is important, we should use a doubly linked list since it is possible to search in both directions. Furthermore, the circular linked list should be used when we have an application when we need to iterate over the nodes in the list. Let us now see more real-world applications of linked lists.

Practical applications of linked lists

As of now, we have discussed singly linked lists, circular linked lists, and doubly linked lists. Depending upon what kind of operations (insertion, deletion, updating, and so on) will be required in different applications, these data structures are used accordingly. Let's see a few real-time applications where these data structures are being used.

Singly linked lists can be used to represent any sparse matrix. Another important application is to represent and manipulate polynomials by accumulating constants in the node of linked lists.

It can also be used in implementing a dynamic memory management scheme that allows the user to allocate and deallocate the memory as per requirements during the execution of programs.

On the other hand, doubly linked lists are used by the thread scheduler in the operating system to maintain the list of processes running at that time. These lists are also used in the implementation of **MRU** (most recently used) and **LRU** (least recently used) cache in the operating system.

Doubly linked lists can also be used by various applications to implement **Undo** and **Redo** functionality. The browsers can use these lists to implement backward and forward navigation of the web pages visited.

A circular linked list can be used by operating systems to implement a round-robin scheduling mechanism. Another application of circular linked lists is to implement **Undo** functionality in Photoshop or Word software and use it in implementing a browser cache that allows you to hit the **BACK** button. Besides that, it is also used to implement advanced data structures such as the Fibonacci heap. Multiplayer games also use a circular linked list to swap between players in a loop.

Summary

In this chapter, we studied the concepts that underlie lists, such as nodes and pointers to other nodes. We have discussed singly linked lists, doubly linked lists, and circular linked lists. We have seen various operations that can be applied to these data structures and their implementations using Python.

These types of data structures have certain advantages over arrays. In the case of arrays, insertion and deletion are quite time-consuming as these operations require the shifting of elements downward and upward, respectively, due to contiguous memory allocations. On the other hand, in the case of linked lists, these operations require only changes in pointers. Another advantage of linked lists over arrays is the allowance of a dynamic memory management scheme that allocates memory during the runtime as and when needed, while the array is based on a static memory allocation scheme.

The singly linked list can traverse in a forward direction only, while traversal in doubly linked lists is bidirectional, hence the reason why the deletion of a node in a doubly linked list is easy compared to a singly linked list. Similarly, circular linked lists save time while accessing the first node from the last node as compared to the singly linked list. Thus, each list has its advantages and disadvantages. We should use them as per the requirements of the application.

In the next chapter, we are going to look at two other data structures that are usually implemented using lists—stacks and queues.

Exercise

1. What will be the time complexity when inserting a data element after an element that is being pointed to by a pointer in a linked list?
2. What will be the time complexity when ascertaining the length of the given linked list?
3. What will be the worst-case time complexity for searching a given element in a singly linked list of length n ?
4. For a given linked list, assuming it has only one head pointer that points to the starting point of the list, what will be the time complexity for the following operations?
 - a. Insertion at the front of the linked list
 - b. Insertion at the end of the linked list
 - c. Deletion of the front node of the linked list
 - d. Deletion of the last node of the linked list
5. Find the n^{th} node from the end of a linked list.
6. How can you establish whether there is a loop (or circle) in a given linked list?
7. How can you ascertain the middle element of the linked list?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/MEvK4>

