Linked Lists in Detail with Python Examples: Single Linked Lists

stackabuse.com/linked-lists-in-detail-with-python-examples-single-linked-lists/

By Usman Malik • 0 Comments

Linked lists are one of the most commonly used data structures in any programming language. In this article, we will study linked lists in detail. We will see what are the different types of linked lists, how to traverse a linked list, how to insert and remove elements from a linked list, what are the different techniques to sort a linked list, how to reverse a linked list and so on.

After reading this article, you should be able to crack all the linked list interview questions.

What is a Linked List?

Before we study what are linked lists, let's first briefly review how Arrays store data. In arrays, the data is stored at contiguous memory locations. For instance, if the first item in the array is stored at index 10 of the memory and is of size 15 bytes, the second item will be stored at index 10+15+1 = 26th index. Therefore, it is straight forward to traverse an array.

To find the third item in an array, you can simply use the starting index of the first item, plus the size of the first item, plus the size of the second item, plus 1.

How Linked Lists Store Data

Linked Lists, on the other hand, are different. Linked lists, do not store data at contiguous memory locations. For each item in the memory location, linked list stores value of the item and the reference or pointer to the next item. One pair of the linked list item and the reference to next item constitutes a node.

For instance, if a node consists of 34|10, it means that the value of the node is 30, while the next item is stored at the memory location "10". In order to traverse a linked list, you just need to know the memory location or reference of the first node, the rest of nodes can be sequentially traversed using the reference to the next element in each node.

The reference to the first node is also known as the start node.

Linked Lists vs Arrays:

- A linked list is a dynamic data structure which means that the memory reserved for the link list can be increased or reduced at runtime. No memory is allocated for a linked list data structure in advance. Whenever a new item is required to be added to the linked, the memory for the new node is created at run time. On the other hand, in case of the array, memory has to be allocated in advance for a specific number of items. In cases where sufficient items are not available to fill all array index, memory space is wasted.
- Since arrays require contiguous memory locations, it is very difficult to remove or insert an item in an array since the memory locations of a large number of items have to be updated. On the other hand, linked list items are not stored in a contiguous memory location, therefore you can easily update linked lists.
- Owing to its flexibility, a linked list is more suitable for implementing data structures like stacks, queues, and lists.

However, there are some downsides to the linked list as well.

- Since each linked list item has to store the reference to the next item, some extra memory is required.
- Unlike Arrays, where you can directly access an item, you cannot access a linked list item directly since the only information you have is the reference to the first item. In Big O terms, worst-case access time is O(n).

In this series of articles, we will study the following types of linked lists along with their different functionalities.

- Single Linked List
- Doubly Linked List
- Circular Linked List
- Linked List with Header
- Sorted Linked List

In this first part of the article, we will focus on single linked list and its different operations.

Single Linked List

A single linked list is the simplest of all the variants of linked lists. Every node in a single linked list contains an item and reference to the next item and that's it.

In this section, we will see how to create a node for the single linked list along with the functions for different types of insertion, traversal, and deletion.

Creating the Node Class

The first thing that you need to do is to create a class for the nodes. The objects of this class will be the actual nodes that we will insert in our linked list. We know that a node for a single linked list contains the item and the reference to the next node. Therefore, our node class will contain two member variables `item` and `ref`. The value of the `item` will be set by the value passed through the constructor, while the reference will be initially set to null.

Execute the following script:

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

Creating the Single Linked List Class

Next, we need to create a class for the Linked List. This class will contain the methods to insert, remove, traverse and sort the list. Initially, the class will only contain one member `start_node` that will point to the starting or first node of the list. The value of `start_node` will be set to null using the constructor since the linked list will be empty at the time of creation. The following script creates a class for the linked list.

```
class LinkedList:
    def __init__(self):
        self.start_node = None
```

Now we have created a class for our single list. The next step is to add insertion function to insert items into the linked list. But before that, we will add a function to traverse a linked list. This function will help us read the data in our list.

Traversing Linked List Items

The Python code for the traverse function is as follows. Add the function below to the `LinkedList` class that we created in the last section.

```
def traverse_list(self):
    if self.start_node is None:
        print("List has no element")
        return
    else:
        n = self.start_node
        while n is not None:
            print(n.item , " ")
            n = n.ref
```

Let's see what is happening in the above function. The function has two main parts. First, it checks whether the linked list is empty or not. The following code checks that:

```
 if self.start_node is None:
        print("List has no element")
        return
```

If the linked list is empty, that means there is no item to iterate. In such cases, the `traverse_list()` function simply prints the statement that the list has no item.

Otherwise if the list has an item, the following piece of code will execute:

```
    n = self.start_node
        while n is not None:
            print(n.item , " ")
            n = n.ref
```

As we said earlier, the `start` variable will contain a reference to the first nodes. Therefore, we initialize a variable `n` with `start` variable. Next, we execute a loop that executes until `n` becomes none. Inside the loop, we print the item stored at the current node and then set the value of `n` variable to `n.ref`, which contains the reference to the next node. The reference of the last node is `None` since there is no node after that. Therefore, when `n` becomes `None`, the loop terminates.

Now, we have a function to traverse a linked list, let's see how we can add items to a single linked list.

Inserting Items

Depending upon the location where you want to insert an item, there are different ways to insert items in a single linked list.

Inserting Items at the Beginning

The simplest way to insert an item in a single linked list is to add an item at the start of the list. The following function inserts item at the start of the list. Add this function to the `LinkedList` class that we created earlier.

```
    def insert_at_start(self, data):
        new_node = Node(data)
        new_node.ref = self.start_node
        self.start_node= new_node
```

In the script above, we create a method `insert_at_start()`, the method accepts one parameter, which is basically the value of the item that we want to insert. Inside the method, we simply create an object of the `Node` class and set its reference to the `start_node` since `start_node` was previously storing the first node, which after insertion of a new node at the start will become the second node.

Therefore, we add the reference of `start_node` to the `ref` variable of the new node. Now since the `new_node` is the first node, we set the value of the `start_node` variable to `new_node`.

Inserting Items at the End

The following function is used to add an item at the end of the linked list.

```
def insert_at_end(self, data):
    new_node = Node(data)
    if self.start_node is None:
        self.start_node = new_node
        return
    n = self.start_node
    while n.ref is not None:
        n= n.ref
    n.ref = new_node;
```

In the above script, we create a function `insert_at_end()`, which inserts the element at the end of the linked list. The value of the item that we want to insert is passed as an argument to the function. The function consists of two parts. First we check if the linked list is empty or not, if the linked list is empty, all we have to do is set the value of the `start_node` variable to `new_node` object.

On the other hand, if the list already contains some nodes. We initialize a variable `n` with the start node. We then iterate through all the nodes in the list using a while loop as we did in the case of `traverse_list` function. The loop terminates when we reach the last node. We then set the reference of the last node to the newly created `new_node`.

Add the `insert_at_end()` function to the `LinkedList` class.

Inserting Item after another Item

We may need to add item after another item in a single linked list. To do so, we can use the `insert_after_item()` function as defined below:

```
def insert_after_item(self, x, data):

    n = self.start_node
    print(n.ref)
    while n is not None:
        if n.item == x:
            break
        n = n.ref
    if n is None:
        print("item not in the list")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node
```

The `insert_after_item()` function accepts two parameters: `x` and `data`. The first parameter is the item after which you want to insert the new node while the second parameter contains the value for the new node.

We start by creating a new variable `n` and assigning `start_node` variable to it. Next, we traverse through the linked list using while loop. The while loop executes until `n` becomes `None`. During each iteration, we check if the value stored in the current node is equal to the value passed by the `x` parameter. If the comparison returns true, we break the loop.

Next, if the item is found, the `n` variable will not be `None`. The reference of the `new_node` is set to reference stored by `n` and the reference of `n` is set to `new_node`. Add the `insert_after_item()` function to the `LinkesList` class.

Inserting Item before another Item

```
def insert_before_item(self, x, data):
    if self.start_node is None:
        print("List has no element")
        return

    if x == self.start_node.item:
        new_node = Node(data)
        new_node.ref = self.start_node
        self.start_node = new_node
        return

    n = self.start_node
    print(n.ref)
    while n.ref is not None:
        if n.ref.item == x:
            break
        n = n.ref
    if n.ref is None:
        print("item not in the list")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node
```

In the script above we define the `insert_before_item()` function. The function has three parts. Let's look at each part in detail.

```
if self.start_node is None:
    print("List has no element")
    return
```

In the script above, we check if the list is empty. If it is actually empty, we simply print that the list has no element and return from the function.

Next, we check if the element is located at the first index. Look at the following script:

```
if x == self.start_node.item:
    new_node = Node(data)
    new_node.ref = self.start_node
    self.start_node = new_node
    return
```

If the element after which we want to insert a new node is located at the first index. We simply set the reference of the newly inserted node to the `start_node` and then set the value of `start_node` to `new_node`.

Finally, if the list is not `None` and the element is not found at the first index, we create a new variable `n` and assign `start_node` variable to it. Next, we traverse through the linked list using while loop. The while-loop executes until `n.ref` becomes `None`. During each iteration, we check if the value stored in the reference of the current node is equal to the value passed by the `x` parameter. If the comparison returns true, we break the loop.

Next, if the item is found, the `n.ref` variable will not be `None`. The reference of the `new_node` is set to reference of `n` and the reference of `n` is set to `new_node`. Look at the following script:

```
if n.ref is None:
    print("item not in the list")
else:
    new_node = Node(data)
    new_node.ref = n.ref
    n.ref = new_node
```

Add `insert_before_item()` function to the `LinkedList` class.

Inserting Item at Specific Index

Sometimes, we need to insert item at a specific index, we can do so with the help of the following script:

```
def insert_at_index (self, index, data):
    if index == 1:
        new_node = Node(data)
        new_node.ref = self.start_node
        self.start_node = new_node
    i = 1
    n = self.start_node
    while i < index-1 and n is not None:
        n = n.ref
        i = i+1
    if n is None:
        print("Index out of bound")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node
```

In the script, we first check if the index that we want to store item to is 1, then simply assign `start_node` to the reference of the `new_node` and then set the value of `start_node` to `new_node`.

Next, execute a while loop which executes until the counter `i` becomes greater than or equal to the `index-1`. For instance, if you want to add a new node to the third index. During the first iteration of the while loop, `i` will become 2 and the currently iterated node will be '2'. The loop will not execute again since `i` is now 2 which is equal to index-1 (3-1=2). Therefore, the loop will break. Next, we add a new node after the currently iterated node (which is node 2), hence the new node is added at index.

It is important to mention that if the index or the location passed as argument is greater than the size of the linked list, a message will be displayed to the user that index is out of range or out of bound.

Testing Insertion Functions

Now we have defined all our insertion functions, let's test them.

First, create an object of the linked list class as follows:

```
new_linked_list = LinkedList()
```

Next, let's first call the `insert_at_end()` function to add three elements to the linked list. Execute the following script:

```
new_linked_list.insert_at_end(5)
new_linked_list.insert_at_end(10)
new_linked_list.insert_at_end(15)
```

To see, if the items have actually been inserted, let's traverse through the linked list using traverse function.

```
new_linked_list.traverse_list()
```

You should see the following output:

```
5
10
15
```

Next, let's add an element at the start:

```
new_linked_list.insert_at_start(20)
```

Now, if you traverse the list, you should see the following output:

```
20
5
10
15
```

Let's add a new item 17 after item 10:

```
new_linked_list.insert_after_item(10, 17)
```

Traversing the list returns the following output now:

```
20
5
10
17
15
```

You can see 17 inserted after 10.

Let's now insert another item 25 before the item 17 using `insert_before_item()` function as shown below:

```
new_linked_list.insert_before_item(17, 25)
```

Now the list will contain the following elements:

```
20
5
10
25
17
15
```

Finally, let's add an element at the third location, which is currently occupied by 10. You will see that 10 will move one location forward and the new item will be inserted at its place. The `insert_at_index()` function can be used for this purpose. The following script inserts item `8` at index the third index of the list.

```
new_linked_list.insert_at_index(3,8)
```

Now if you traverse the list, you should see the following output:

```
20
5
8
10
25
17
15
```

And with that, we have tested all of our insertion functiond. We currently have 7 elements in our list. Let's write a function that returns the number of elements in a linked list.

Counting Elements

The following function counts the total number of elements.

```
    def get_count(self):
        if self.start_node is None:
            return 0;
        n = self.start_node
        count = 0;
        while n is not None:
            count = count + 1
            n = n.ref
        return count
```

In the script above we create `get_count()` function which simply counts the number of elements in the linked list. The function simply traverses through all the nodes in the array and increments a counter using while loop. At the end of the loop, the counter contains total number of elements in the loop.

Add the above function to the `LinkedList` class, compile the `LinkedList` class and then insert some elements in the `LinkedList` as we did in the last section. We had 7 items in our linked list, by the end of the last section.

Let's use the `get_count()` function to get the total number of items in the list:

```
new_linked_list.get_count()
```

You should see the number of items in your linked list in the output.

Alternatively, another way to get the 'count' of the list would be to track the number of items inserted and removed from the list in a simple counter variable belonging to the `LinkedList` class. This works well, and is faster than the `get_count` method above, if the underlying list data structure can not be manipulated from outside the class.

Searching Elements

Searching for an element is quite similar to counting or traversing a linked list, all you have to do is to compare the value to be searched with the value of node during each iteration. If the value is found, print that the value is found and break the loop. If the element is not found after all the nodes are traversed, simply print that the element not found.

The script for the `search_item()` is as follows:

```
def search_item(self, x):
    if self.start_node is None:
        print("List has no elements")
        return
    n = self.start_node
    while n is not None:
        if n.item == x:
            print("Item found")
            return True
        n = n.ref
    print("item not found")
    return False
```

Add the above function to the `LinkedList` class. Let's search an element in the previously created list. Execute the following script:

```
new_linked_list.search_item(5)
```

Since we inserted 5 in our linked list, the above function will return true. The output will look like this:

```
Item found
True
```

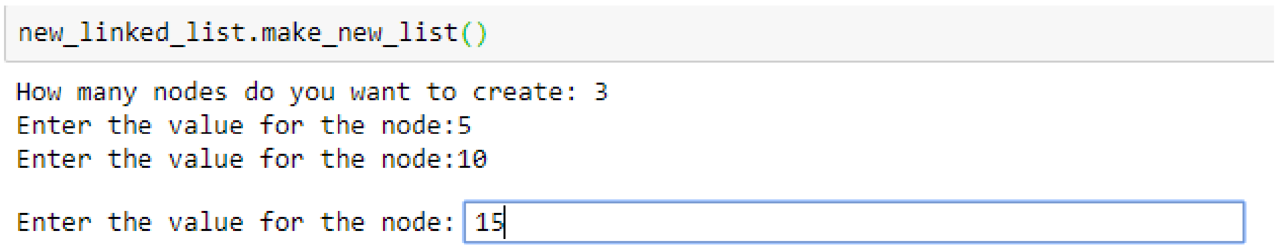
Creating a Linked List

Though we can add items one by one using any of the insertion functions. Let's create a function that asks the user to enter the number of elements in the node and then the individual element and enters that element in the linked list.

```
def make_new_list(self):
    nums = int(input("How many nodes do you want to create: "))
    if nums == 0:
        return
    for i in range(nums):
        value = int(input("Enter the value for the node:"))
        self.insert_at_end(value)
```

In the above script, the `make_new_list()` function first asks the user for the number of items in the list. Next using a for-loop, the user is prompted to enter the value for each node, which is then inserted into the linked list using the `insert_at_end()` function.

The following screenshot shows the `make_new_list()` function in action.



```
new_linked_list.make_new_list()
```

```
How many nodes do you want to create: 3
Enter the value for the node:5
Enter the value for the node:10

Enter the value for the node: 15
```

Deleting Elements

In this section, we will see the different ways to delete an element from a single linked list.

Deletion from the Start

Deleting an element or item from the start of the linked list is straightforward. We have to set the reference of the `start_node` to the second node which we can do by simply assigning the value of the reference of the start node (which is pointing to the second node) to the start node as shown below:

```
def delete_at_start(self):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    self.start_node = self.start_node.ref
```

In the script above, we first check if the list is empty or not. If the list is empty we display the message that the list has no element to delete. Otherwise, we assign the value of the `start_node.ref` to the `start_node`. The `start_node` will now point towards the second element. Add the `delete_at_start()` function to the `LinkedList` class.

Deletion at the End

To delete an element from the end of the list, we simply have to iterate through the linked list till the second last element, and then we need to set the reference of the second last element to none, which will convert the second last element to last element.

The script for the function `delete_at_end` is as follows:

```
  def delete_at_end(self):
    if self.start_node is None:
      print("The list has no element to delete")
      return

    n = self.start_node
    while n.ref.ref is not None:
      n = n.ref
    n.ref = None
```

Add the above script to the `LinkedList()` class.

Deletion by Item Value

To delete the element by value, we first have to find the node that contains the item with the specified value and then delete the node. Finding the item with the specified value is pretty similar to searching the item. Once the item to be deleted is found, the reference of the node before the item is set to the node that exists after the item being deleted. Look at the following script:

```
 def delete_element_by_value(self, x):
  if self.start_node is None:
    print("The list has no element to delete")
    return

  # Deleting first node
  if self.start_node.item == x:
    self.start_node = self.start_node.ref
    return

  n = self.start_node
  while n.ref is not None:
    if n.ref.item == x:
      break
    n = n.ref

  if n.ref is None:
    print("item not found in the list")
  else:
    n.ref = n.ref.ref
```

In the script above, we first check if the list is empty. Next, we check if the element to be deleted is located at the start of the linked list. If the element is found at the start, we delete it by setting the first node to the reference of the first node (which basically refers to the second node).

Finally, if the element is not found at the first index, we iterate through the linked list and check if the value of the node being iterated is equal to the value to be deleted. If the comparison returns true, we set reference of the previous node to the node which exists after the node which is being deleted.

Testing Deletion Functions

Let's test deletion functions that we just created. But before that add some dummy data to our linked list using the following script:

```
new_linked_list.insert_at_end(10)
new_linked_list.insert_at_end(20)
new_linked_list.insert_at_end(30)
new_linked_list.insert_at_end(40)
new_linked_list.insert_at_end(50)
```

The above script inserts 5 elements to a linked list. If you traverse the list, you should see the following items:

```
10
20
30
40
50
```

Let's first delete an item from the start:

```
new_linked_list.delete_at_start()
```

Now if you traverse the list, you should see the following output:

```
20
30
40
50
```

Let's delete an element from the end now:

```
new_linked_list.delete_at_end()
```

The list now contains the following items:

```
20
30
40
```

Finally, let's delete an element by value, say 30.

```
new_linked_list.delete_element_by_value(30)
```

Now if you traverse the list, you should not see the item 30.

Reversing a Linked List

To reverse a linked list, you need to have three variables, `prev`, `n` and `next`. The `prev` will keep track of the previous node, the `next` will keep track of the next node will the `n` will correspond to the current node.

We start a while-loop by assigning the starting node to the variable `n` and the `prev` variable is initialized to none. The loop executes until `n` becomes none. Inside the while loop, you need to perform the following functions.

- Assign the value of the reference of the current node to `next`.
- Set the value of reference of the current node `n` to the `prev`
- Set `prev` variable to current node `n`.
- Set current node `n` to the value of `next` node.

At the end of the loop, the `prev` variable will point to the last node, we need to make it the first node so we set the value `self.start_node` variable to `prev`. The while-loop will make each node point to its previous node, resulting in a reversed linked list. The script is as follows:

```
def reverse_linkedlist(self):
    prev = None
    n = self.start_node
    while n is not None:
        next = n.ref
        n.ref = prev
        prev = n
        n = next
    self.start_node = prev
```

Add the above function to the `LinkedList` class. Create a linked list of random numbers and then see if you can reverse it using the `reverse_linkedlist()` function.

Conclusion

In this article, we started our discussion about a single linked list. We saw what are the different functions that can be performed on the linked list such as traversing a linked list, inserting items to a linked list, searching and counting linked list items, deleting items from a linked list and reversing a single linked list.

This is Part 1 of the series of articles on the linked list. In the next part (*coming soon*), we will see how to sort a single linked list, how to merge sorted linked lists and how to remove cycles from a single linked list.

About Usman Malik
Paris (France) Twitter
Programmer | Blogger | Data Science Enthusiast | PhD To Be | Arsenal FC for Life

Previous Post Next Post