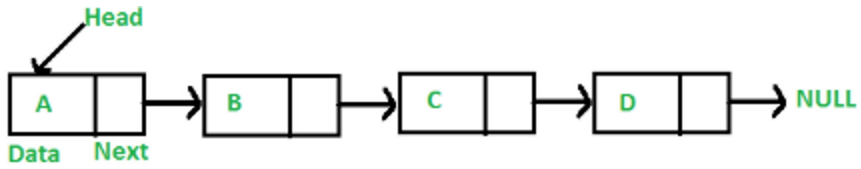
June8

07 June 2020 01:34

Link List

Linked lists are an ordered collection of objects.
Like arrays, Linked List is a linear data structure.
Unlike arrays, linked list elements are not stored at a contiguous location but are linked using pointers.



what makes them different from normal lists?
Linked lists differ from lists in the way that they store elements in memory. While lists use a contiguous memory block to store references to their data, linked lists store references as part of their own elements.

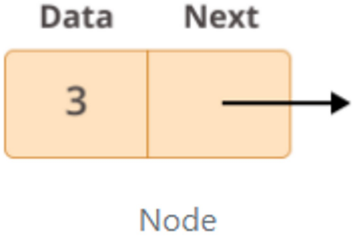
Why Linked List?
Arrays can be used to store linear data of similar types, but arrays have the following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

Advantage- Ease of insertion/deletion
Drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

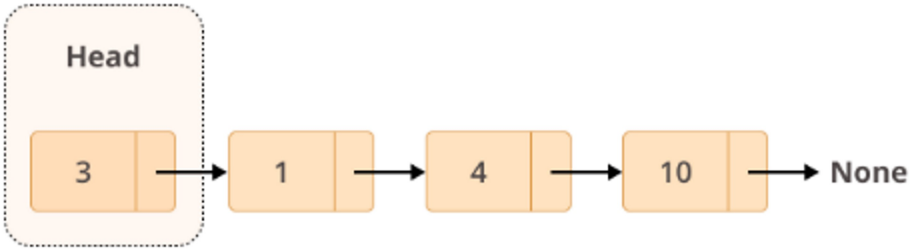
Main Concept -
Each element of a linked list is called a **node**, and every node has two different fields:

1. **Data** contains the value to be stored in the node.
2. **Next** contains a reference to the next node on the list.

what a typical node looks like:



A linked list is a collection of nodes. The first node is called the **head**, and it's used as the starting point for any iteration through the list. The last node must have its next reference pointing to **None** to determine the end of the list.



Linked List

How to Create a Linked List?
First things first, create a **class** to represent your linked list:

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

The only information you need to store for a linked list is where the list starts (the head of the list). Next, create another **class** to represent each node of the linked list:

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

In the above class definition, you can see the two main elements of every single node: **data** and **next**.
You can also add a __repr__ to both classes to have a more helpful representation of the objects

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

    def __repr__(self):
        return self.data

class LinkedList:
    def __init__(self):
        self.head = None

    def __repr__(self):
        node = self.head
        nodes = []
        while node is not None:
            nodes.append(node.data)
            node = node.next
        nodes.append("None")
        return " -> ".join(nodes)
```

Using above classes lets create Linked List with three nodes

```
llist = LinkedList() #Empty Linked List
print(llist)

first_node = Node("monday")
llist.head = first_node   # Linked List with head
print(llist)
second_node = Node("Tuesday")
third_node = Node("Wednesday")
first_node.next = second_node
second_node.next = third_node   # Linked List with end
print(llist)
```

```
None
monday -> None
monday -> Tuesday -> Wednesday -> None
```

How to Traverse a Linked List?

One of the most common things you will do with a linked list is to **traverse** it. Traversing means going through every single node, starting with the **head** of the linked list and ending on the node that has a next value of **None**.

Traversing is just a fancier way to say iterating.

```
#for traversal
def printList(self):
      node = self.head
      while (node):
        print (node.data)
        node = node.next
```

The method above goes through the list and yields every single node. The most important thing to remember about this is that you need to always validate that the current node is not **None**. When that condition is True, it means you've reached the **end** of your linked list.

After yielding the current node, you want to move to the next node on the list.
That's why you add node = node.next.

Questions to implement

1. Given a linked list consists of data, a next pointer and also a random pointer which points to random node of the list. Write code for cloning the list.
2. Find modular node from the end: Given a singly linked list, write a function to find the first from the end whose n%k= =0, where n is the number of elements in the list and k is an integer constant. If n = 19 and k = 3 then we should return 16th node.
3. Reversing a linked list in pairs If you have a linked list that holds 1 -> 2 --> 3->4->X then after reverse it should return 2->1->4->3->X
4. Suppose there are two singly linked lists both of which intersect at some point and become a
   single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. list 1 may have n nodes before it reaches the intersection point, and list 2. might have m nodes before it reaches the intersection point where m and n may be m = n, m < n or m > n.