Introduction

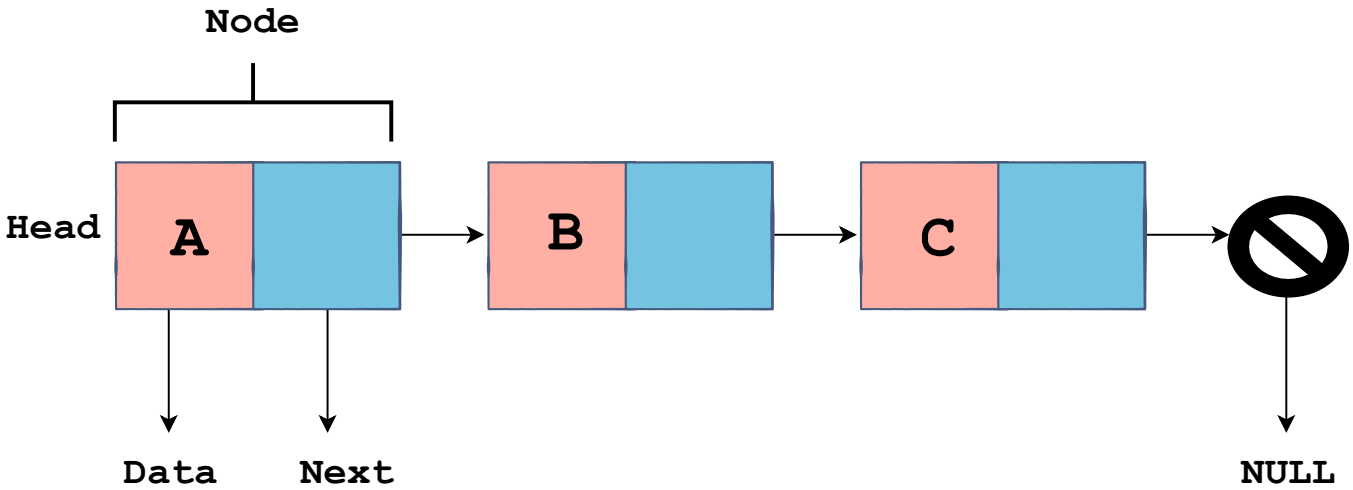
This lesson introduces you to the singly linked list and implements its structure in Python.

We'll cover the following

For the sake of this course, we will go over the following different types of linked lists and implement them in Python:

1. Singly Linked Lists
2. Doubly Linked Lists
3. Circular Linked List

Below is a simple depiction of a singly linked list:



Singly Linked List

Structure

Every linked list consists of nodes, as shown in the illustration above. Every node has two components:

1. Data
2. Next

The *data* component allows a node in the linked list to store an element of data that can be of type string, character, number, or any other type of object. In the illustration above, the data elements are `A`, `B`, and `C` which are of character type.

The *next* component in every node is a pointer that points from one node to another.

The start of the linked list is referred to as the **head**. `head` is a pointer that points to the beginning of the linked list, so if we want to traverse the linked list to obtain or access an element of the linked list, we'll start from `head` and move along.

The last component of a singly linked list is a notion of null. This null idea terminates the linked list. In Python, we call this `None`. The last node in a singly linked list points to a null object, and that tells you that it's the end of the linked list.

Arrays vs. Linked Lists

	Arrays	Linked Lists
Insertion/Deletion at the beginning of the array or linked list given a value	$O(n)$	$O(1)$
Access Element	$O(1)$	$O(n)$
Contiguous Memory	Yes	No

Insertion/Deletion

The insertion/deletion operation is in $O(n)$ operations for insertion/deletion of value at the beginning of the array. Now think if we are given an array and a value to insert at the beginning of an array. For insertion, we have to shift all the elements in the array to the right. The shifting makes room for the element at the beginning of the array. Due to the shifting of the elements, the time complexity is $O(n)$. The same is the case with the deletion of an element from the beginning of an array. Again, we shift all the elements back by one index. Therefore, this operation also has a cost of $O(n)$ time complexity.

Inserting a node at the head of a linked list given the head node is a constant-time operation as we need to change the orientation of a few pointers. If we are given the exact pointer after which we have to insert another node, it will be a constant-time operation.

Accessing Elements

Accessing any element given an index in arrays is better than accessing nth elements in linked lists. It is a constant time operation to access elements in arrays. If given an array and an index, it can immediately give you the element at which the entry is stored. This is because arrays are contiguous.

Accessing an nth element in a linked list is an $O(n)$ operation given that you have access to the head node of the linked list. If we want to access an element, we need to start from the head pointer and traverse the entire linked list before we can get to it.

Contiguous Memory

Arrays are contiguous in memory which allows the access time to be constant, whereas, in linked lists, you do not have the luxury of contiguous memory.

You should probably keep in mind the table above when it comes to trade-offs between arrays and linked lists. If you're given a problem, you might want to consider whether or not an array is the preferred data structure. For that assessment, you will need to be aware of the pros and cons of the two data structures.

Implementation

Now let's go ahead and create our classes in Python:

- `Node` class
- `LinkedList` class

```
1  class Node:
2    def __init__(self, data):
3      self.data = data
4      self.next = None
5
6  class LinkedList:
7    def __init__(self):
8      self.head = None
```

class Node and class LinkedList

We create a class called `Node`. In the constructor of this class, we give the argument of `self` and `data` on **line 2**. Every node is going to consist of `data` and `next`. We define `self.data` equal to `data` that is passed into the constructor of the object of class `Node` (**line 3**). We set `self.next` equal to `None` on **line 4**. This is something that we'll set again as we make use of the node, but for now, we just set it to `None`. That's pretty much all we need for the `Node` class right now.

On **line 6**, we define a `LinkedList` class, and in the constructor, we again pass `self`. On **line 8**, we define the `head` pointer, which will point to the first node in the linked list. Initially, we just set it equal to `None`.

Now that we have defined the classes in our implementation, we'll learn how to insert elements into a linked list in the next lesson.