Singly Linked List Insertion

Let's look at the Pythonic implementation for the insertion of a node in a linked list.

We'll cover the following

Types of Insertion

The three types of insertion strategies used in singly linked-lists are:

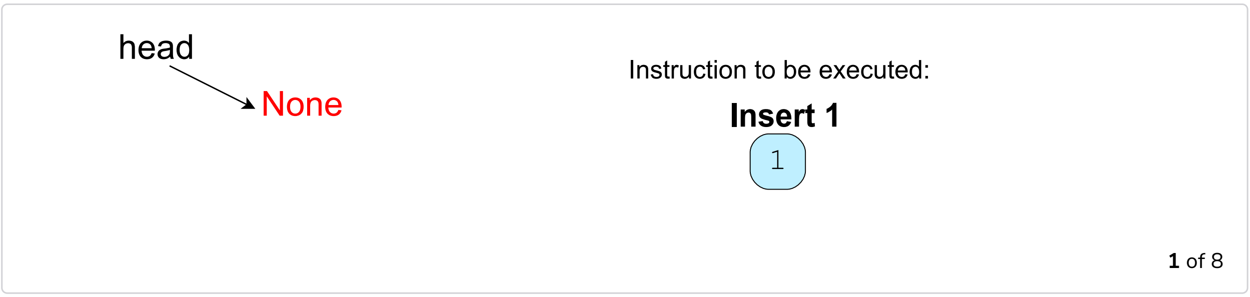
1. Insertion at the head
2. Insertion at the tail
3. Insertion at the kth index

Insertion at Head

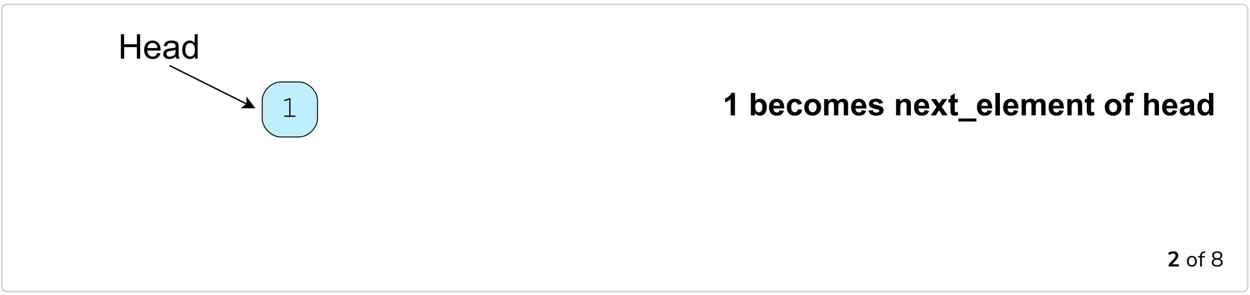
This type of insertion means that we want to insert a new element as the first element of the list.

As a result, the newly added node will become the **head**, which in turn will point to the previous first node.

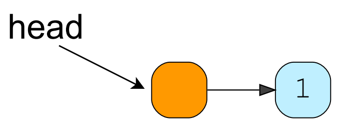
For a better understanding of the **Insertion At Head** method, check out the illustration below:



1 of 8

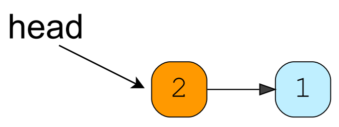


2 of 8



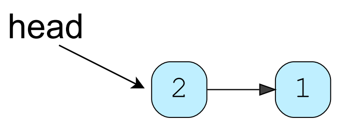
Insert 2

3 of 8

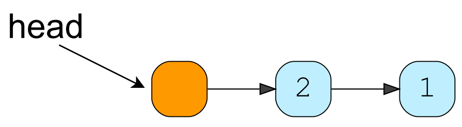


2 becomes next_element of head
1 becomes next_element of 2

4 of 8

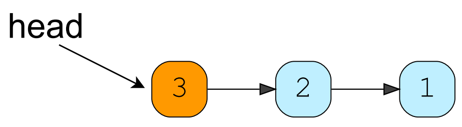


5 of 8



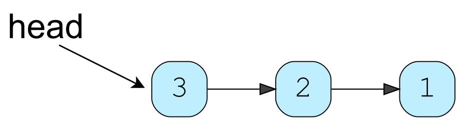
Insert 3

6 of 8



3 becomes next_element of head
2 becomes next_element of 3

7 of 8



8 of 8

Implementation

For this lesson, we are only dealing with insertion at head; the other approaches will be covered later.

The implementation of this operation is simple and straightforward. It is all about correctly manipulating the `next_element` of the node being inserted.

Take a look at the implementation for `insert_at_head` below:

LinkedList.py

Node.py

```python
from Node import Node


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

    # Insertion at Head
    def insert_at_head(self, data):
        # Create a new node containing your specified value
        temp_node = Node(data)
        # The new node points to the same node as the head
        temp_node.next_element = self.head_node
        self.head_node = temp_node  # Make the head point to the new
        return self.head_node  # return the new list

    def is_empty(self):
        if self.head_node is None:
            return True
        else:
            return False

# Supplementary print function
    def print_list(self):
        if(self.is_empty()):
            print("List is Empty")
            return False
        temp = self.head_node
        while temp.next_element is not None:
            print(temp.data, end=" -> ")
            temp = temp.next_element
        print(temp.data, "-> None")
        return True


list = LinkedList()
list.print_list()

print("Inserting values in list")
for i in range(1, 10):
    list.insert_at_head(i)
list.print_list()

```

Explanation

To start things off, let's explain the function called `print_list(self)`. It simply starts at the head node, and iterates through the nodes using temp and displays their value. Our iteration ends when `temp.next_element` is `None`, which means that we've reached the last node in the list.

The list that is created is going to look like this:

$$9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> NULL$$

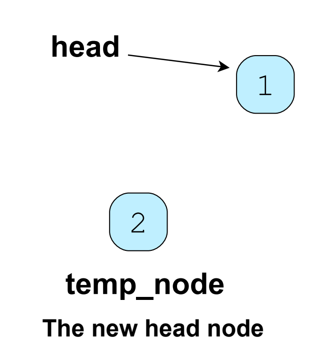
insert_at_head()

Now, we are at the main part of the code. `insert_at_head()` takes an integer value as `data` and inserts it just after `head` to make it the first element of the list.

The function follows these steps to insert a new node:

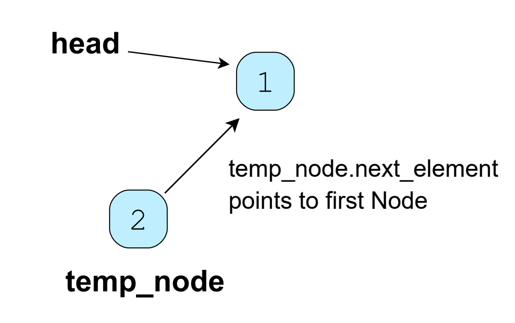
- Create a new Node object with the given value, called `temp_node`.
- Make the `next_element` of `temp_node` will become the new `head` .
- `temp_node` will become the `next_element` of `head`.

Here is a graphical representation of the whole process:

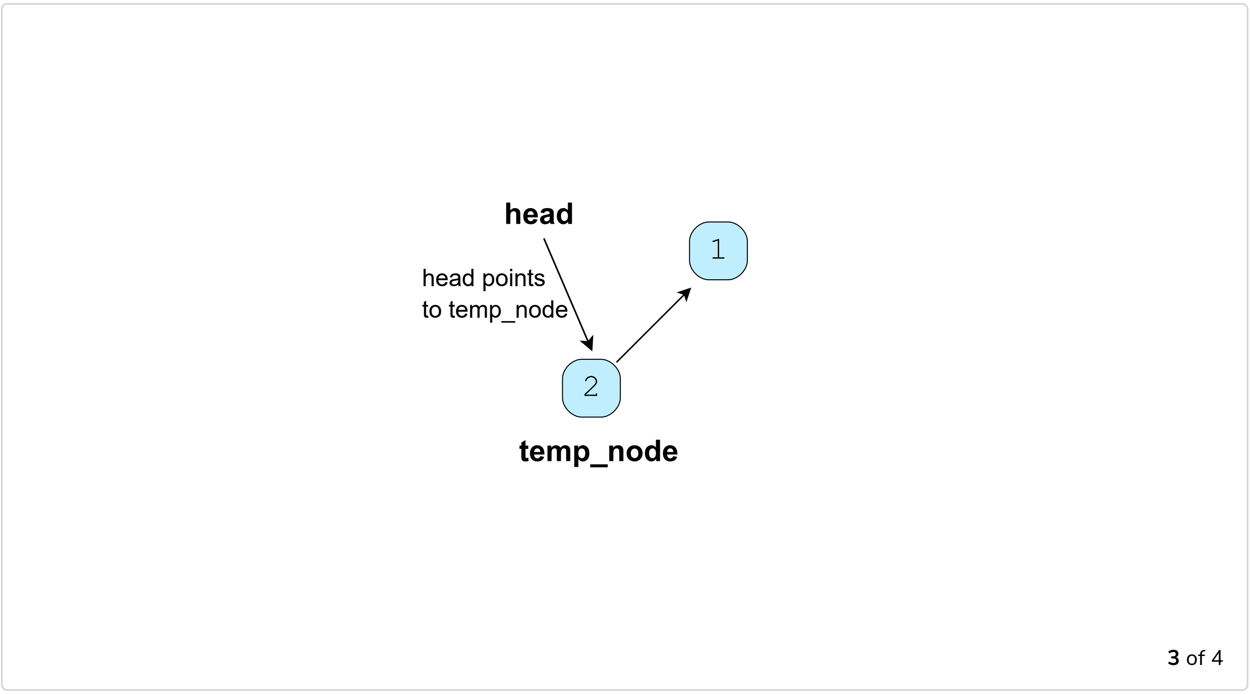


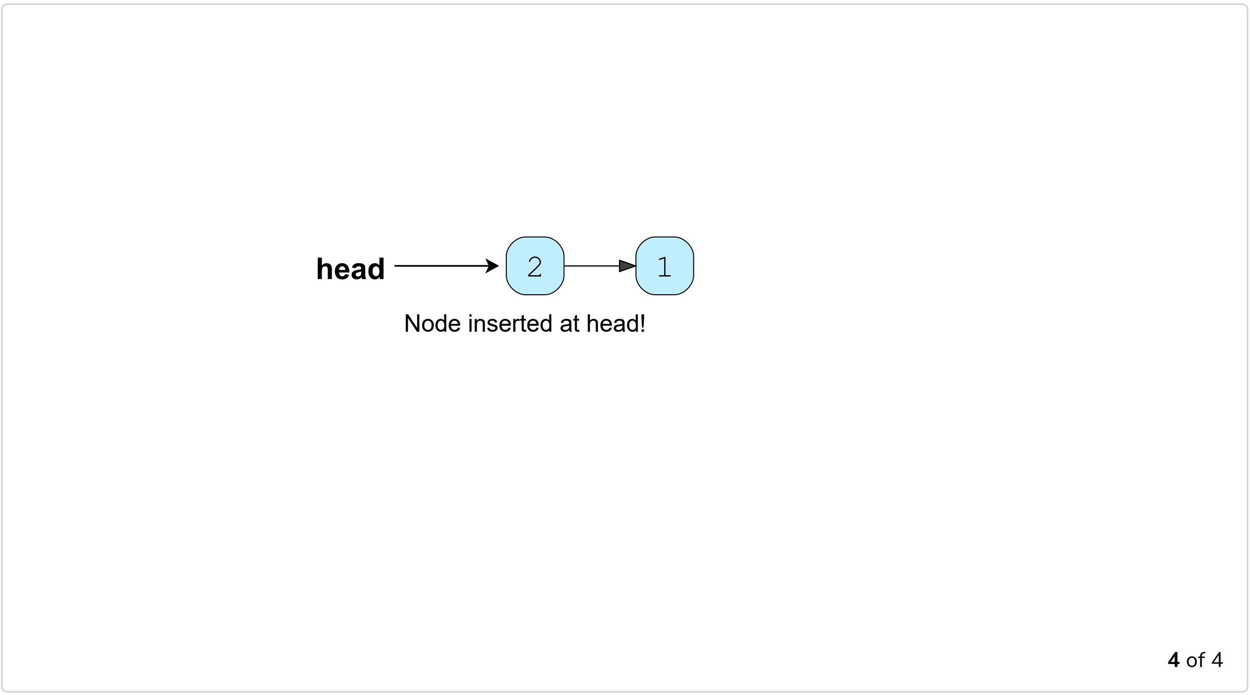
insert_at_head()

1 of 4



2 of 4





Time Complexity #

At every instance, we point the `head` to a new node. Therefore, the time complexity for **insertion at head** is *O(1)*.

Play around with the code and observe its functionality. The next lesson will cover the second insertion strategy, **Insertion at Tail**. By now, it shouldn't sound too intimidating.