# Linked Lists

## **Singly-Linked Lists**

A singly linked list is a fundamental data structure that consists of a sequence of elements, each of which contains a reference (or link) to the next element in the sequence. Because the next reference of a node can be viewed as a ***link*** or ***pointer*** to another node, the process of traversing a list is also known as ***link hopping*** or ***pointer hopping***. Here's an overview of its key aspects:

### **Structure**

1. **Nodes**: A singly linked list is made up of nodes. Each node typically contains two parts:
   * **Data**: The actual value or data stored in the node.
   * **Next Pointer**: A reference (or pointer) to the next node in the list.
2. **Head**: The first node in the list is known as the head. It is the entry point to access the list.
3. **Tail**: The last node, which points to **null** (or **None** in Python), indicating the end of the list.
4. **Length**: The number of nodes in the list.

### **Operations**

Singly linked lists support various operations, including:

1. **Insertion**:
   * At the beginning (head): O(1) time complexity.
   * At the end (tail): O(n) if the tail is not maintained, O(1) if the tail is maintained.
   * After/before a given node: O(1), if the previous node is known.
2. **Deletion**:
   * From the beginning: O(1).
   * From the end: O(n), as it requires traversing the list to find the penultimate node.
   * A specific node: O(n) in the worst case, as it may require traversing to find the node.
3. **Search**: To find a node or retrieve data, you need to traverse the list from the head, making the worst-case time complexity O(n).
4. **Traversal**: Singly linked lists can be traversed linearly, starting from the head and moving through each node until the end (null/None) is reached.

### **Advantages**

1. **Dynamic Size**: Unlike arrays, the size of a singly linked list can grow or shrink dynamically.
2. **Efficient Insertions/Deletions**: At the beginning or with a known reference node, these operations are more efficient than in an array.

### **Disadvantages**

1. **Memory Overhead**: Each node requires extra memory for the next pointer.
2. **No Random Access**: You cannot directly access a node by index, unlike arrays.
3. **Linear Search Time**: Searching for an element requires O(n) time.

### **Applications**

Singly linked lists are useful in scenarios where:

* The list size changes frequently.
* You primarily add/remove elements from the beginning of the list.
* Memory allocation is fragmented (since linked lists don’t require contiguous memory allocation).

## **ADT Operations**

### **Adding an element at the beginning of the singly-linked list**

when adding a new node at the beginning (i.e., as the new head), the steps you mentioned are slightly misstated. Here's the correct sequence of steps:

1. **Create the New Node**: This node will contain the data you want to add and initially, its **next** pointer will be **None**.
2. **Set the New Node’s Next Pointer**: Next, you need to link this new node to the existing list. To do this, set the **next** pointer of **NewNode** to the current head of the list (**Head**). Now, **NewNode** points to what was previously the first node of the list. This step links the next reference of the new node to the previous node.
3. **Update the Head of the List**: Finally, you update the head reference of the list to be this new node. The original head is no longer the first node; **NewNode** has taken its place.

Here's a breakdown:

* **Before Insertion**: Head -> Node A -> Node B -> ... -> None
* **Insert New Node (Node N)**: Node N -> None
* **Set Node N's Next**: Node N -> Node A
* **Update Head**: Head -> Node N

**Why This Update is Necessary:**

* **Maintaining the List Integrity**: Without updating the head, the list's starting point remains unchanged, and the new node isn't actually part of the list.
* **Access and Traversal**: All operations on the list (like traversal, search, etc.) start from the **head**. If the **head** isn't updated, these operations won't include the new node.
* **Reflecting Changes**: The head update ensures that any changes (like adding a new node at the beginning) are reflected across the entire list.

In a singly linked list, the **head** is not a node by itself but rather a reference or pointer to the first node in the list. When you add a new node to the beginning of the list, you need to update this **head** reference to point to the new node, making it the first node of the list.

### Here's the Process:

1. **Start with a Head Node**: Initially, **self.head** points to the first node in your list. If your list is empty, **self.head** is **None**.
2. **Create a New Node**: When you create a new node, this node is not yet part of the list. It's just an independent node.
3. **Set the New Node's Next Pointer**: You then set the **next** attribute of your new node to point to the current **self.head**. This step links your new node to the existing list.
4. **Update the Head Reference**: Finally, you update **self.head** to point to the new node. This step is crucial because it officially makes your new node the first node of the list. Without this step, the linked list still starts with the old first node, and the new node wouldn't be considered part of the list.

# **Implementing a Stack Using Singly Linked List**

In designing such an implementation, we need to decide whether to model the top of the stack at the head or at the tail of the list. There is clearly a best choice here; we can efficiently insert and delete elements in constant time only at the head. Since all stack operations affect the top, we orient the top of the stack at the head of our list.

Composition vs Inheritance

Whether to implement a stack using composition or inheritance when using a singly linked list (SLL) as the underlying data structure depends on the specific requirements of your application and your design philosophy. Both approaches have their advantages and considerations. Let's explore them:

**Composition**

In composition, your **Stack** class would have a **SinglyLinkedList** object as an attribute. You would then implement stack methods (**push**, **pop**, **peek**, etc.) that internally call the corresponding methods of the **SinglyLinkedList**.

Advantages:

* **Encapsulation**: Composition is often favored for encapsulation. It allows you to expose only the stack-related methods and hide the list-specific methods that aren't relevant to stack behavior.
* **Flexibility**: If you decide to change the underlying data structure in the future (e.g., from a linked list to an array), you can do so more easily without affecting the stack's public interface.

**Inheritance**

In inheritance, **Stack** extends **SinglyLinkedList**, meaning it inherits all of its methods and properties.

Advantages:

* **Code Reuse**: Inheritance allows you to reuse code from the base class, which can result in less code overall.
* **Is-A Relationship**: If your stack "is a" special kind of linked list (which can be a valid interpretation), inheritance can be a natural choice.

Considerations:

* **Interface Exposure**: With inheritance, all public methods of the **SinglyLinkedList** become part of the **Stack** interface. This might include methods that don't make sense for a stack (like **add\_last** or random access methods).

**Conclusion**

* **Composition** might be more suitable if you want to restrict the stack operations strictly to **push**, **pop**, **peek**, etc., and keep the option open to change the internal data structure later without affecting the external interface of the stack.
* **Inheritance** can be a simpler solution if you're comfortable with **Stack** having the extended interface of a **SinglyLinkedList**, and you see a clear "is-a" relationship between them.