Table of Contents

1. Introduction

A **linked list** is a linear data structure in which elements are stored in nodes, and each node points to the next node in the sequence. Unlike arrays, linked lists do not have a fixed size, and memory is allocated dynamically as nodes are added. The last node in a linked list points to `null` or a special sentinel node to indicate the end.

A node in a linked list typically contains two components:

1. **Data:** The actual value or payload stored in the node.
2. **Next (or Pointer/Reference):** A reference or link to the next node in the sequence.

The first node in a linked list is called the **head**, and the last node is often called the **tail** (if the linked list maintains a reference to the tail). If a linked list is singly linked, each node points to the next node. In a doubly linked list, each node has references to both the next and the previous nodes.

There are various types of linked lists, including:

1. **Singly Linked List:** Each node points to the next node in the sequence.
2. **Doubly Linked List:** Each node has references to both the next and the previous nodes.
3. **Circular Linked List:** The last node points back to the head, forming a loop.

2. Algorithm

The basic operations on a linked list include inserting a node, deleting a node, and searching for a node. Here's an overview of the algorithms for these operations:

Insertion:

1. Insert at the Beginning:

- Create a new node.

- Set the new node's `next` pointer to the current head.
- Update the head to the new node.

```
def insert_at_beginning(linked_list, data):
  new_node = Node(data)
  new_node.next = linked_list.head
  linked_list.head = new_node
```

2. Insert at the End:

- Create a new node.
- Traverse the list to find the last node.
- Set the last node's `next` pointer to the new node.

```
def insert_at_end(linked_list, data):
  new_node = Node(data)
  if not linked_list.head:
      linked_list.head = new_node
      return
  last_node = linked_list.head
  while last_node.next:
      last_node = last_node.next
  last_node.next = new_node
```

3. Insert at a Specific Position:

- Create a new node.
- Traverse the list to find the node at the desired position.
- Adjust pointers to insert the new node.

```
def insert_at_position(linked_list, data, position):
  if position == 0:
      insert_at_beginning(linked_list, data)
      return
  new_node = Node(data)
  current_node = linked_list.head
  for _ in range(position - 1):
```

```
        if not current_node:
            return  # Handle invalid position
        current_node = current_node.next
    new_node.next = current_node.next
    current_node.next = new_node
```

Deletion:

1. Delete at the Beginning:

- Update the head to the next node.

```
def delete_at_beginning(linked_list):
    if linked_list.head:
        linked_list.head = linked_list.head.next
```

2. Delete at the End:

- Traverse the list to find the last and second-to-last nodes.
- Set the second-to-last node's `next` pointer to `None`.

```
def delete_at_end(linked_list):
    if not linked_list.head:
        return
    if not linked_list.head.next:
        linked_list.head = None
        return
    last_node = linked_list.head
    while last_node.next.next:
        last_node = last_node.next
    last_node.next = None
```

3. Delete at a Specific Position:

- Traverse the list to find the node at the desired position.
- Adjust pointers to skip the node.

```
def delete_at_position(linked_list, position):
    if not linked_list.head:
```

```
            return
        if position == 0:
            linked_list.head = linked_list.head.next
            return
        current_node = linked_list.head
        for _ in range(position - 1):
            if not current_node.next:
                return  # Handle invalid position
            current_node = current_node.next
        if current_node.next:
            current_node.next = current_node.next.next
```

Searching:

1. Linear Search:

- Traverse the list and compare the data in each node with the target value.

```
def linear_search(linked_list, target):
    current_node = linked_list.head
    while current_node:
        if current_node.data == target:
            return True
        current_node = current_node.next
    return False
```

These are basic algorithms for a singly linked list. Doubly linked lists involve additional pointers for the previous node, and circular linked lists have the last node pointing back to the head. Understanding these basic operations helps in implementing and working with linked lists effectively.

3. Example Usage

Let's see a simple example of a singly linked list in Python:

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

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

# Example usage:
linked_list = LinkedList()
linked_list.head = Node(1)
second_node = Node(2)
third_node = Node(3)

linked_list.head.next = second_node
second_node.next = third_node
```

In this example, we have a linked list with three nodes, where each node contains a piece of data and a reference to the next node in the sequence.

4. Advantages

Linked lists offer several advantages:

1. **Dynamic Size:** Linked lists can grow or shrink in size during program execution, allowing for dynamic memory allocation.
2. **Efficient Insertion and Deletion:** Inserting or deleting elements in a linked list is generally more efficient than in an array because it involves updating references, and there is no need to shift elements.
3. **No Pre-allocation of Memory:** Memory is allocated as needed, and there is no need to pre-allocate a fixed-size block.