Linked Lists

A linked list is a data structure that consists of a sequence of nodes, where each node contains data and a reference (or link) to the next node in the sequence. Linked lists are commonly used in scenarios where dynamic memory allocation is necessary, or where frequent insertion and deletion of data is required, as they are more flexible than arrays in these respects.

**Types of Linked Lists**

1. **Singly Linked List**: Each node contains data and a reference to the next node. It allows traversal in one direction (forward).
2. **Doubly Linked List**: Each node contains data, a reference to the next node, and a reference to the previous node, allowing bidirectional traversal.
3. **Circular Linked List**: Similar to a singly linked list, but the last node points back to the first node, forming a circle.
4. **Doubly Circular Linked List**: A circular list with nodes that point to both the next and previous nodes.

**Components of a Linked List**

* **Node**: The fundamental building block of a linked list. A node contains the data (or value) and a reference (or link) to the next node.
* **Head**: The first node in a linked list. It’s often used as the starting point for traversing the list.
* **Tail**: The last node in a linked list (only in non-circular lists). The next reference in the tail node is None (in singly linked lists).

**Basic Operations on a Linked List**

* **Traversal**: Going through each node in the list.
* **Insertion**: Adding a new node to the list (at the beginning, end, or a specified position).
* **Deletion**: Removing a node from the list (from the beginning, end, or a specified position).
* **Searching**: Finding a node with a specific value.
* **Reversing**: Changing the order of nodes in the list.

**Implementation of a Singly Linked List in Python**

Here's a simple implementation of a singly linked list in Python:

python

Copy code

# Define a node

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

# Define a linked list

class LinkedList:

def \_\_init\_\_(self):

self.head = None

# Method to insert a new node at the end

def append(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

return

last\_node = self.head

while last\_node.next:

last\_node = last\_node.next

last\_node.next = new\_node

# Method to display the linked list

def display(self):

current = self.head

while current:

print(current.data, end=" -> ")

current = current.next

print("None")

# Method to insert a node at the beginning

def insert\_at\_beginning(self, data):

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

# Method to delete a node by value

def delete\_by\_value(self, value):

current = self.head

if current and current.data == value:

self.head = current.next

current = None

return

prev = None

while current and current.data != value:

prev = current

current = current.next

if current is None:

print("Value not found in the list.")

return

prev.next = current.next

current = None

# Example usage

ll = LinkedList()

ll.append(1)

ll.append(2)

ll.append(3)

ll.insert\_at\_beginning(0)

ll.display() # Output: 0 -> 1 -> 2 -> 3 -> None

ll.delete\_by\_value(2)

ll.display() # Output: 0 -> 1 -> 3 -> None

In this example:

* append() adds a node to the end of the list.
* insert\_at\_beginning() inserts a node at the beginning of the list.
* delete\_by\_value() removes the first node with a specific value.
* display() prints all nodes in the list.

**Advantages of Linked Lists**

* **Dynamic Size**: Linked lists can grow or shrink as needed.
* **Efficient Insertions/Deletions**: Insertion and deletion operations are generally O(1) if you have a reference to the node, unlike arrays where shifting is required.

**Disadvantages of Linked Lists**

* **No Random Access**: Linked lists don’t support efficient random access as arrays do.
* **Memory Overhead**: Each node needs additional memory to store a reference to the next (and potentially previous) node.

Linked lists are a foundational data structure and can be modified to create more complex data structures, like stacks, queues, and graphs.