# Doubly Linked List

This guide explains **Doubly Linked Lists (DLLs)** in a fun, relatable, and student-friendly way. It uses the exact code and variable names , maintaining its structure and enhancing the comments for clarity.

## 📘 What is a Doubly Linked List?

A **Doubly Linked List** is a linear data structure where each node links to both its previous and next neighbors.

🎯 Real-World Analogy: Think of your browser history—go back to the previous page or forward to the next. That’s a DLL in action!

## 🧱 Node Structure

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data # Value to store in the node  
 self.prev = None # Pointer to the previous node  
 self.next = None # Pointer to the next node

## 🛠️ DoublyLinkedList Operations

class DoublyLinkedList:  
 def \_\_init\_\_(self):  
 self.head = None # Head points to the first node in the list  
  
 def insert\_at\_beginning(self, data):  
 new\_node = Node(data)  
 if self.head is None:  
 self.head = new\_node  
 return  
  
 new\_node.next = self.head # New node's next is the old head  
 self.head.prev = new\_node # Old head's prev is the new node  
 self.head = new\_node # Update head to the new node  
  
 def insert\_at\_end(self, data):  
 new\_node = Node(data)  
 if self.head is None:  
 self.head = new\_node  
 return  
  
 last\_node = self.head  
 while last\_node.next is not None:  
 last\_node = last\_node.next  
  
 last\_node.next = new\_node  
 new\_node.prev = last\_node  
  
 def insert\_at\_position(self, data, target\_position):  
 print(f"\nInsert at position {target\_position} value {data}")  
 if target\_position <= 0:  
 print("Invalid position")  
 return  
  
 if self.head is None and target\_position != 1:  
 print("Invalid position")  
 return  
  
 if target\_position == 1:  
 self.insert\_at\_beginning(data)  
 return  
  
 current\_position = 1  
 current\_node = self.head  
 while current\_node is not None and current\_position < target\_position - 1:  
 current\_position += 1  
 current\_node = current\_node.next  
  
 if current\_node is None:  
 print("Invalid position")  
 return  
  
 new\_node = Node(data)  
  
 if current\_node.next is not None:  
 current\_node.next.prev = new\_node # Update next node's prev  
 new\_node.next = current\_node.next # Link to the next node  
  
 current\_node.next = new\_node # Link current to new  
 new\_node.prev = current\_node # Link new to current  
  
 def delete\_at\_beginning(self):  
 if self.head is None:  
 return  
  
 if self.head.next is None:  
 self.head = None  
 return  
  
 self.head = self.head.next  
 self.head.prev = None # Remove back reference  
  
 def delete\_at\_end(self):  
 if self.head is None:  
 return  
  
 if self.head.next is None:  
 self.head = None  
 return  
  
 last\_node = self.head  
 while last\_node.next.next is not None:  
 last\_node = last\_node.next  
  
 last\_node.next = None  
  
 def delete\_at\_position(self, target\_position):  
 if self.head is None:  
 print("List is empty")  
 return  
  
 if target\_position <= 0:  
 print(f"Invalid target position {target\_position}")  
 return  
  
 if self.head.next is None:  
 self.head = None  
 return  
  
 to\_be\_deleted = self.head  
 current\_position = 1  
  
 while current\_position < target\_position and to\_be\_deleted is not None:  
 current\_position += 1  
 to\_be\_deleted = to\_be\_deleted.next  
  
 if to\_be\_deleted is None:  
 print(f"Target position {target\_position} is invalid")  
 return  
  
 if to\_be\_deleted.next is None:  
 to\_be\_deleted.prev.next = None  
 return  
  
 to\_be\_deleted.next.prev = to\_be\_deleted.prev  
 to\_be\_deleted.prev.next = to\_be\_deleted.next  
  
 def search(self, key):  
 if self.head is None:  
 print("Key is not found in the list")  
 return  
  
 current\_node = self.head  
 while current\_node is not None:  
 if current\_node.data == key:  
 print("Key is found in the list")  
 return  
 current\_node = current\_node.next  
  
 print("Key is not found in the list")  
  
 def print\_all\_nodes(self):  
 print("\nPrinting all the nodes in the doubly linked list\n")  
 if self.head is None:  
 print("List is empty")  
 return  
  
 current\_node = self.head  
 while current\_node is not None:  
 print(f"<-- {current\_node.data} -->", end=" ")  
 current\_node = current\_node.next  
 print()

## ✅ Advantages vs Other Structures

| Feature | Array | Singly Linked List | Doubly Linked List |
| --- | --- | --- | --- |
| Dynamic Size | ❌ | ✅ | ✅ |
| Direct Index Access | ✅ (O(1)) | ❌ (O(n)) | ❌ (O(n)) |
| Forward/Backward Traverse | ❌ | ❌ | ✅ |
| Efficient Insert/Delete | ❌ | ✅ | ✅ |
| Memory Usage | ✅ | ✅ | ❌ (extra pointer) |

## ⏱️ Time Complexity Comparison

| Operation | Time Complexity |
| --- | --- |
| Insert at Beginning | O(1) |
| Insert at End | O(n) |
| Insert at Position | O(n) |
| Delete at Beginning | O(1) |
| Delete at End | O(n) |
| Delete at Position | O(n) |
| Search | O(n) |

## 💡 Real-World Use Cases

* Web browser forward/back buttons
* Undo-redo operations in editors
* Media playlists
* Navigation systems with backtracking

## 🎓 Summary

Doubly Linked Lists are:

* Flexible with two-way traversal
* Easy to insert/delete from both ends
* A bit heavier on memory

But totally worth it when direction and flexibility matter. 🔄

Happy coding! 🚀