**UNIT I**

**LINEAR DATA STRUCTURES – LIST**

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation – singly linked lists-doubly-linked lists – circularly-linked list-applications of lists –Polynomial ADT — Radix Sort—Multi lists.

**Data Structure**

A **Data Structure** is the method of storing and organizing data in the computer memory.

It is the branch of Computer Science that deals with arranging such large datasets in such a manner so that they can be accessed and modified as per the requirements.

In other words, a data structure is a fundamental building block for all critical operations to be performed on the data.

* Data structures are the data structured in a way for efficient use by the users.
* As the computer program relies hugely on the data and also requires a large volume

of data for its performance, therefore it is highly important to arrange the data.

* This arrangement of data in organized structures is known as a data structure.
* Storing of the data in data structures allows the access, modifications, and other

operations that can be carried over the data elements.

* The arrangement of the data is mainly done in a computer and therefore proper

algorithms are required to carry on operations with the data structures.

* Reducing space and decreasing the time complexity of different tasks is the main

aim of data structures.

**Data Structures examples**

* The following examples from our day-to-day life will help you understand the concept of data structures.
* Our name is a data structure. It comprises First name, Middle Name, and Last name.
* The date is a data structure. It includes three types of data viz date (numeric value), month (character or numeric value), and year (numeric value).

**Characteristics of data structures:**

* It depicts the logical representation of data in computer memory.
* It represents the logical relationship between the various data elements.
* It helps in efficient manipulation of stored data elements.
* It allows the programs to process the data in an efficient manner.

**The most important points in a data structure are:**

* A large amount of data is organized through every type of data structure.
* A particular principle is followed by every data structure.
* The basic principle of the data structure should be followed even if any operations are carried out over the data structure.
* Arrangement of the data within a data structure can follow different orders.

A data structure is therefore classified according to the way of arrangement of the data.

Basically, there are two types of data structure.

* Primitive data structure
* Non-primitive data structure

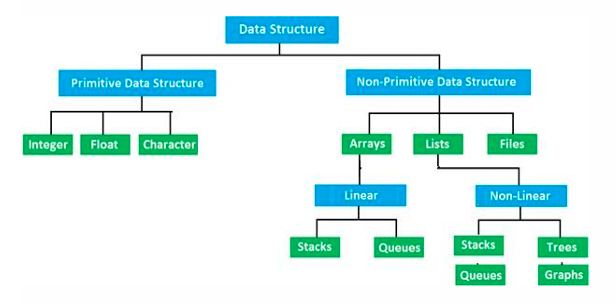
The primitive type of data structure includes the predefined such as char, float, int,

and double.

The non-primitive data structures are used to store the collection of elements. This

data structure can be further categorized into

* Linear data structure
* Non-Linear data structure.



**What is the Linear data structure?**

* A linear data structure is a structure in which the elements are stored sequentially,

and the elements are connected to the previous and the next element.

* As the elements are stored sequentially, so they can be traversed or accessed in a

single run.

* The implementation of linear data structures is easier as the elements are

sequentially organized in memory.

* The data elements in an array are traversed one after another and can access only

one element at a time.

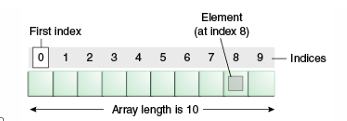
The types of linear data structures are Array, Queue, Stack, Linked List.

**Array:**

* An array consists of data elements of a same data type.
* For example, if we want to store the roll numbers of 10 students, so instead of

creating 10 integer type variables, we will create an array having size 10.

* Therefore, we can say that an array saves a lot of memory and reduces the length of the code.



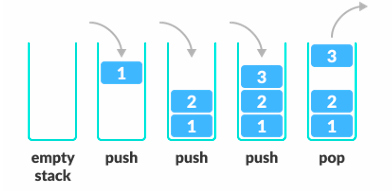
**Stack**

* It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data

added last will be removed first.

* The addition of data element in a stack is known as a push operation, and the

deletion of data element form the list is known as pop operation.

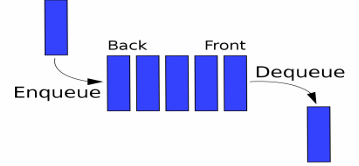


**Queue:**

* It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the

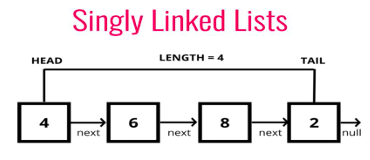
element which is added first will be removed first.

* There are two terms used in the queue front end and rear.
* The insertion operation performed at the back end is known as enqueue, and the deletion operation performed at the front end is known as dequeue.



**Linked list:**

It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.



**What is a Non-linear data structure?**

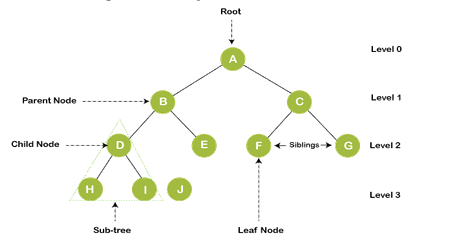
A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is non-sequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

**Trees and Graphs are the types of non-linear data structure.**

**Tree**

It is a non-linear data structure that consists of various linked nodes.

* It has a hierarchical tree structure that forms a parent-child relationship.
* The diagrammatic representation of a tree data structure is shown below:

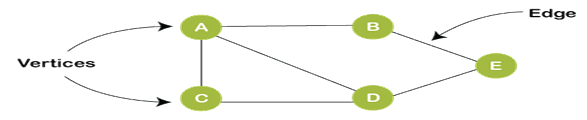


**Graph**

* A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices.
* The vertices are used to store the data elements, while the edges represent the

relationship between the vertices.

* A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn,
* Facebook. In the case of facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



|  |  |  |
| --- | --- | --- |
|  | Linear Data structure | Non-Linear Data structure |
| Basic | In this structure, the elements are arranged sequentially or linearly and attached to one another. | In this structure, the elements are arranged hierarchically or non-linear manner. |
| Types | Arrays, linked list, stack, queue are the types of a linear data structure. | Trees and graphs are the types of a non-linear data structure. |
| Implementation | Due to the linear organization, they are easy to implement. | Due to the non-linear organization, they are difficult to implement. |
| Traversal | As linear data structure is a single level, so it requires a single run to traverse each data item. | The data items in a non-linear data structure cannot be accessed in a single run. It requires multiple runs to be traversed. |
| Arrangement | Each data item is attached to the previous and next items. | Each item is attached to many other items. |
| Levels | This data structure does not contain any hierarchy, and all the data elements are organized in a single level. | In this, the data elements are arranged in multiple levels. |
| Memory utilization | In this, the memory utilization is not efficient. | In this, memory is utilized in a very efficient manner. |
| Time complexity | The time complexity of linear data structure increases with the increase in the input size. | The time complexity of non-linear data structure often remains same with the increase in the input size |
| Applications | Linear data structures are mainly used for developing the software. | Non-linear data structures are used in image processing and Artificial Intelligence. |

**Application of data structures:**

* Data structures are widely applied in the following areas:
* Compiler design
* Operating system
* Statistical analysis package
* DBMS
* Numerical analysis
* Simulation
* Artificial intelligence
* Graphics

**ABSTRACT DATA TYPES (ADTs)**

Data types specify the type of data structure. We can classify the data type into primitive data types like integer, float, double, etc., or abstract data types like list, stack, queue, etc.

Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviors for a data structure, without specifying how these operations are implemented or how data is organized in memory.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it provides an implementation-independent view.

**Why Use ADTs?**

* **Encapsulation:**Hides complex implementation details behind a clean interface.
* **Reusability**: Allows different internal implementations (e.g., array or linked list) without changing external usage.
* **Modularity:** Simplifies maintenance and updates by separating logic.
* **Security:**Protects data by preventing direct access, minimizing bugs and unintended changes.

**LIST ADT**

List ADT (Abstract Data Type) is implemented using the built-in list class, but conceptually, the List ADT supports a set of operations, regardless of how it is implemented internally (array-based or linked list).

**Core Operations of List ADT**

Create a list

my\_list = [] # Empty list

my\_list = [1, 2, 3] # List with elements

Insert elements

At the end:

my\_list.append(4)

At a specific position:

my\_list.insert(1, 10) # Insert 10 at index 1

Add multiple elements:

my\_list.extend ([5, 6])

Delete elements

By value:

my\_list.remove(10) # Removes first occurrence of 10

By index:

del my\_list[2]

Pop last element (or specific index):

my\_list.pop() # Removes last element

my\_list.pop(1) # Removes element at index 1

**Various operations performed on List**

* Insert (X, 5)- Insert the element X after the position 5.
* Delete (X) - The element X is deleted
* Find (X) - Returns the position of X.
* Next (i) - Returns the position of its successor element i+1.
* Previous (i) Returns the position of its predecessor i-1.
* Print list - Contents of the list is displayed.
* Makeempty- Makes the list empty.

**AbstractDataType List**

**Instances:** List is a collection of elements which are arranged in a linear manner.

**Operations:** Various operations that can be carried out on list are -

**1. Insertion:** This operation is for insertion of element in the list.

**2. Deletion:**This operation removed the element from the list.

**3. Searching:** Based on the value of the key element the desired element can be searched.

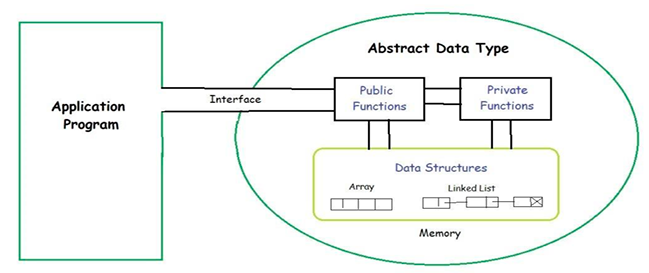
**4. Modification:** The value of the specific element can be changed without changing its location.

**5. Display:** The list can be displayed in forward or in backward manner.

**The List can be implemented by two ways**

1. Array based implementation.

2. Linked List based implementation.



**Difference Between ADTs and UDTs**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Abstract Data Types (ADTs)** | **User-Defined Data Types (UDTs)** |
| Definition | Defines a class of objects and the operations that can be performed on them, along with their expected behavior (semantics), but without specifying implementation details. | A custom data type created by combining or extending existing primitive types, specifying both structure and operations. |
| Focus | What operations are allowed and how they behave, without dictating how they are implemented. | How data is organized in memory and how operations are executed. |
| Purpose | Provides an abstract model to define | Allows programmers to create concrete implementations of data structures using primitive types. |
| Implementation Details | Does not specify how operations are implemented or how data is structured. | Specifies how to create and organize data types to implement the structure. |
| Usage | Used to design and conceptualize data structures. | Used to implement data structures that realize the abstract concepts defined by ADTs. |
| Example | List ADT, Stack ADT, Queue ADT | Structures, classes, enumerations, records. |

Examples of ADTs Now, let’s understand three common ADT’s: List ADT, Stack ADT, and Queue ADT.

**List ADT**

The List ADT (Abstract Data Type) is a sequential collection of elements that supports a set of operations without specifying the internal implementation. It provides an ordered way to store, access, and modify data.

The List ADT need to store the required data in the sequence and should have the following operations:

* get(): Return an element from the list at any given position.
* insert(): Insert an element at any position in the list.
* remove(): Remove the first occurrence of any element from a non-empty list.
* removeAt(): Remove the element at a specified location from a non-empty list.
* replace(): Replace an element at any position with another element.
* size(): Return the number of elements in the list.
* isEmpty(): Return true if the list is empty; otherwise, return false.
* isFull(): Return true if the list is full; otherwise, return false.

**Stack ADT**

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack. In Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

* push(): Insert an element at one end of the stack called the top.
* pop(): Remove and return the element at the top of the stack, if it is not empty.
* peek(): Return the element at the top of the stack without removing it, if the stack is
* not empty.
* size(): Return the number of elements in the stack.
* isEmpty(): Return true if the stack is empty; otherwise, return false.
* isFull(): Return true if the stack is full; otherwise, return false.

**Queue ADT**

The Queue ADT is a linear data structure that follows the FIFO (First In, First Out) principle. It allows elements to be inserted at one end (rear) and removed from the other end (front).

The Queue ADT follows a design similar to the Stack ADT, but the order of insertion and deletion changes to FIFO. Elements are inserted at one end (called the rear) and removed from the other end (called the front). It should support the following operations:

* enqueue(): Insert an element at the end of the queue.
* dequeue(): Remove and return the first element of the queue, if the queue is not empty.
* peek(): Return the element of the queue without removing it, if the queue is not empty.
* size(): Return the number of elements in the queue.
* isEmpty(): Return true if the queue is empty; otherwise, return false.

A Linked List is a data structure consisting of a sequence of nodes, each containing a value and a reference to the next node in the sequence. Unlike arrays, Linked Lists do not require contiguous memory allocation, making them more flexible and efficient for certain operations. In this article, we will explore the advantages and disadvantages of Linked Lists and how to implement them in Python.

**Advantages and Disadvantages of Linked Lists**

Linked Lists offer several advantages over other data structures. Firstly, they allow for efficient insertion and deletion of elements, as they only require updating the references of neighboring nodes. This makes Linked Lists ideal for scenarios where frequent modifications are expected. Additionally, Linked Lists can dynamically grow or shrink in size, unlike arrays, which have a fixed size. However, Linked Lists also have some disadvantages. Unlike arrays, Linked Lists do not support random access to elements, meaning that accessing an element at a specific index requires traversing the list from the beginning. This can result in slower performance for certain operations. Furthermore, Linked Lists require extra memory to store the references to the next nodes, which can be inefficient for small datasets.

**LINKED LIST IMPLEMENTATION IN PYTHON**

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node.

Basic Operations in Linked List

* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Display** − Displays the complete list.
* **Search** − Searches an element using the given key.
* **Delete** − Deletes an element using the given key.

**Types of Linked List**

Singly Linked Lists

Doubly Linked Lists

Circular Linked Lists

A**linked list** is a linear data structure in which all the elements, i.e., nodes, are stored in a sequence but not in contiguous memory locations. Each node of a linked list contains two parts:

**Data:** The actual value.

**Pointer:** A reference to the next node.

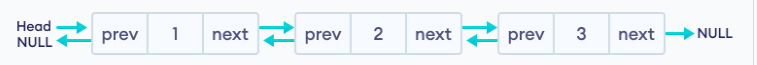
**A singly linked list** refers to a kind of linked list in which each node points to the following node in the sequence, and the last node points to NULL, indicating the end of the list.

Singly Linked Lists are more commonly used in data structures due to their dynamic allocation properties, which aid in better memory usage, node insertion, and deletion compared to arrays.



**Doubly Linked Lists**

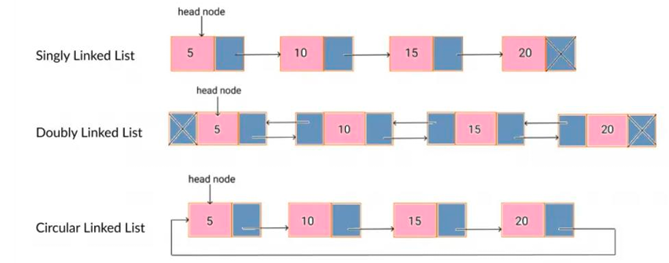
Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.



**Circular Linked Lists**

Circular linked lists can exist in both singly linked list and doubly linked list. Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

Python provides a flexible and intuitive way to implement Linked Lists. There are three main types of Linked Lists: Singly Linked List, Doubly Linked List, and Circular Linked List. Let’s explore each of them in detail.



**ARRAY IMPLEMENTATION OF LIST:**

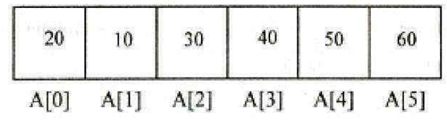
Array is a collection of specific number of same type of data stored in consecutive memory

locations. Array is a static data structure , the memory should be allocated in advance and the

size is fixed. This will waste the memory space when used space is less than the allocated space.

Insertion and Deletion operation are expensive as it requires more data movements

Find and Print list operations takes constant time.



The basic operations performed on a list of elements are

a. Creation of List.

b. Insertion of data in the List

c. Deletion of data from the List

d. Display all data‟s in the List

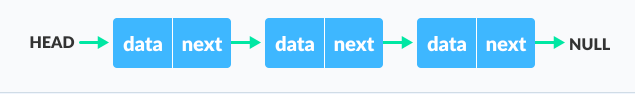
e. Searching for a data in the list

Declaration of Array:

The process of providing only the essentials and hiding the details is known as abstraction. For example, we use primitive values like int, float, and char with the understanding that these data types can operate and be performed on without any knowledge of their implementation details. ADTs operate similarly by defining what operations are possible without detailing their implementation.

**Linked List**

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node. For example,



**Linked list Data Structure**

* The first node has a special name called HEAD. Also, the last node in the linked list

can be identified because its next portion points to NULL.

* Linked lists can be of multiple types: singly, doubly, and circular linked list. In this

article, we will focus on the singly linked list. To learn about other types, visit

**Note:** You might have played the game Treasure Hunt, where each clue includes the

information about the next clue. That is how the linked list operates.

**Representation of Linked List**

Each node consists:

A data item

An address of another node

Linked list Representation

The power of a linked list comes from the ability to break the chain and rejoin it. E.g. if you

wanted to put an element 4 between 1 and 2, the steps would be:

* Create a new struct node and allocate memory to it.
* Add its data value as 4
* Point its next pointer to the struct node containing 2 as the data value
* Change the next pointer of "1" to the node we just created.
* Doing something similar in an array would have required shifting the positions of all

the subsequent elements.

**Linked List Utility**

Lists are one of the most popular and efficient data structures. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

# Linked list implementation in Python

class Node:

# Creating a node

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

self.item = item

self.next = None

class LinkedList:

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

self.head = None

if \_\_name\_\_ == '\_\_main\_\_':

linked\_list = LinkedList()

# Assign item values

linked\_list.head = Node(1)

second = Node(2)

third = Node(3)

# Connect nodes

linked\_list.head.next = second

second.next = third

# Print the linked list item

while linked\_list.head != None:

print(linked\_list.head.item, end=" ")

linked\_list.head = linked\_list.head.next

**Linked List Complexity**

|  |  |  |
| --- | --- | --- |
| Time Complexity | Worst case | Average Case |
| Search | O(n) | O(n) |
| Insert | O(1) | O(1) |
| Deletion | O(1) | O(1) |

Space Complexity: O(n)

**Linked List Applications**

* Dynamic memory allocation
* Implemented in stack and queue
* In undo functionality of softwares
* Hash tables, Graphs

**SINGLY LINKED LIST**

A Singly Linked List consists of nodes where each node contains a value and a

reference to the next node in the sequence. Here’s how you can create a Singly Linked

List in Python:

class Node:

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

self.value = value

self.next = None

class Linked List:

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

self.head = None

**Creating a Singly Linked**

List To create a Singly Linked List, we need to define a Node class that represents each node in the list. Each node contains a value and a reference to the next node. The Linked List class serves as the container for the nodes, with the head attribute pointing to the first node in the list.

**Inserting Nodes in a Singly Linked List**

Inserting nodes in a Singly Linked List involves updating the references of neighboring nodes. Here’s an example of inserting a node at the beginning of the list:

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

new\_node = Node(value)

new\_node.next = self.head

self.head = new\_node

**Deleting Nodes from a Singly Linked List**

Deleting nodes from a Singly Linked List requires updating the references of neighboring nodes. Here’s an example of deleting a node with a specific value:

def delete\_node(self, value):

current = self.head

if current.value == value:

self.head = current.next

else:

while current.next:

if current.next.value == value:

current.next = current.next.next

break

current = current.next

**Searching in a Singly Linked List**

Searching for a specific value in a Singly Linked List involves traversing the list until

the value is found or the end of the list is reached. Here’s an example of searching for a

value:

def search(self, value):

current = self.head

while current:

if current.value == value:

return True

current = current.next

return False

**Reversing a Singly Linked List**

Reversing a Singly Linked List requires updating the references of each node to point

to the previous node. Here’s an example of reversing a Singly Linked List:

def reverse(self):

previous = None

current = self.head

while current:

next\_node = current.next

current.next = previous

previous = current

current = next\_node

self.head = previous

**DOUBLY LINKED LIST**

A Doubly Linked List is similar to a Singly Linked List, but each node contains a reference to both the next node and the previous node in the sequence. This allows for efficient traversal in both directions. Here’s how you can create a Doubly Linked List in Python:

class Node:

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

self.value = value

self.next = None

self.previous = None

class DoublyLinked List:

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

self.head = None

**Creating a Doubly Linked List**

To create a Doubly Linked List, we define a Node class that contains a value, a reference to the next node, and a reference to the previous node. The DoublyLinked List class serves as the container for the nodes, with the head attribute pointing to the first node in the list.

**Inserting Nodes in a Doubly Linked List**

Inserting nodes in a Doubly Linked List involves updating the references of neighboring nodes. Here’s an example of inserting a node at the beginning of the list:

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

new\_node = Node(value)

if self.head:

self.head.previous = new\_node

new\_node.next = self.head

self.head = new\_node

**Deleting Nodes from a Doubly Linked List**

Deleting nodes from a Doubly Linked List requires updating the references of neighboring nodes. Here’s an example of deleting a node with a specific value:

def delete\_node(self, value):

current = self.head

if current.value == value:

self.head = current.next

if self.head:

self.head.previous = None

else:

while current.next:

if current.next.value == value:

current.next = current.next.next

if current.next:

current.next.previous = current

break

current = current.next

**SEARCHING IN A DOUBLY LINKED LIST**

Searching for a specific value in a Doubly Linked List involves traversing the list in

either direction until the value is found or the end of the list is reached. Here’s an

example of searching for a value:

current = next\_node

next\_node = next\_node.next

if current == self.head:

break

self.head = previous

Reversing a Doubly Linked List

Reversing a Doubly Linked List requires updating the references of each node to swap the next and previous pointers. Here’s an example of reversing a Doubly Linked List:

def reverse(self):

current = self.head

while current:

next\_node = current.next

current.next = current.previous

current.previous = next\_node

if not next\_node:

self.head = current

current = next\_node

**Circular Linked List**

A Circular Linked List is a variation of a Singly Linked List where the last node points back to the first node, creating a circular structure. This allows for efficient traversal from any node in the list. Here’s how you can create a Circular Linked List in Python: class Node:

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

self.value = value

self.next = None

class CircularLinked List:

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

self.head = None

**Creating a Circular Linked List**

To create a Circular Linked List, we define a Node class that contains a value and a reference to the next node. The CircularLinked List class serves as the container for the nodes, with the head attribute pointing to the first node in the list. Additionally, the last node’s next reference is set to the head, creating a circular structure.

Inserting Nodes in a Circular Linked List

Inserting nodes in a Circular Linked List involves updating the references of neighboring nodes. Here’s an example of inserting a node at the beginning of the list:

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

new\_node = Node(value)

if not self.head:

self.head = new\_node

new\_node.next = self.head

else:

current = self.head

while current.next != self.head:

current = current.next

current.next = new\_node

new\_node.next = self.head

self.head = new\_node

**Deleting Nodes from a Circular Linked List**

Deleting nodes from a Circular Linked List requires updating the references of neighboring nodes. Here’s an example of deleting a node with a specific value:

def delete\_node(self, value):

if not self.head:

return

current = self.head

if current.value == value:

while current.next != self.head:

current = current.next

if current == self.head:

self.head = None

else:

current.next = self.head.next

self.head = self.head.next

else:

previous = None

while current.next != self.head:

previous = current

current = current.next

if current.value == value:

previous.next = current.next

break

**Searching in a Circular Linked List**

Searching for a specific value in a Circular Linked List involves traversing the list until the value is found or the entire list is traversed. Here’s an example of searching for a value:

def search(self, value):

if not self.head:

return False

current = self.head

while True:

if current.value == value:

return True

current = current.next

if current == self.head:

break

return False

**Reversing a Circular Linked List**

Reversing a Circular Linked List requires updating the references of each node to reverse the circular structure. Here’s an example of reversing a Circular Linked List:

def reverse(self):

if not self.head:

return

previous = None

current = self.head

next\_node = current.next

while True:

current.next = previous

previous = current

current = next\_node

next\_node = next\_node.next

if current == self.head:

break

self.head = previous

**Common Operations on Linked Lists**

Linked Lists support various common operations that can be performed on the elements. Let’s explore some of these operations: Accessing Elements in a Linked List To access elements in a Linked List, we can traverse the list starting from the head node and move to the next node until we reach the desired position. Here’s an example of accessing an element at a specific index:

def get\_element(self, index):

current = self.head

count = 0

while current:

if count == index:

return current.value

count += 1

current = current.next

raise IndexError("Index out of range")

**Modifying Elements in a Linked List**

Modifying elements in a Linked List involves traversing the list to find the desired element and updating its value. Here’s an example of modifying an element at a specific index:

def modify\_element(self, index, new\_value):

current = self.head

count = 0

while current:

if count == index:

current.value = new\_value

return

count += 1

current = current.next

raise IndexError("Index out of range")Copy Code

**Finding the Length of a Linked List**

To find the length of a Linked List, we can traverse the list and count the number of

nodes. Here’s an example of finding the length of a Linked List:

def get\_length(self):

current = self.head

count = 0

while current:

count += 1

current = current.next

return count

**Checking if a Linked List is Empty**

To check if a Linked List is empty, we can simply check if the head node is None. Here’s an example of checking if a Linked List is empty: def is\_empty(self): return self.head is None

**Concatenating Linked Lists**

To concatenate two Linked Lists, we can traverse the first list to find the last node and update its next reference to the head of the second list. Here’s an example of concatenating two Linked Lists:

def concatenate(self, other\_list):

if not self.head:

self.head = other\_list.head

else:

current = self.head

while current.next:

current = current.next

current.next = other\_list.head

**Linked List vs. Array Linked**

Lists and arrays are both commonly used data structures, but they have different characteristics that make them suitable for different scenarios. Let’s compare

Linked Lists and arrays in terms of memory efficiency, insertion and deletion efficiency, and random access efficiency.

**Memory Efficiency**

Linked Lists are more memory-efficient than arrays because they do not require contiguous memory allocation. Each node in a Linked List only needs to store the value and a reference to the next node, whereas arrays need to allocate memory for all elements, even if they are not used.

**Insertion and Deletion Efficiency**

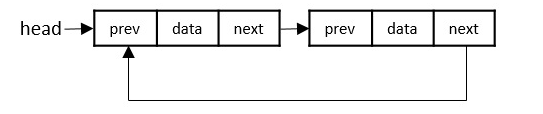
Linked Lists excel in insertion and deletion operations, especially when elements are frequently added or removed from the middle of the list. Inserting or deleting an element in a Linked List only requires updating the references of neighboring nodes, whereas arrays may require shifting elements to accommodate the change.

**Random Access Efficiency**

Arrays provide efficient random access to elements based on their indices, as they allow direct memory addressing. In contrast, Linked Lists require traversing the list from the beginning to access an element at a specific index, resulting in slower performance for random access operations.

**Choosing the Right Data Structure**

The choice between Linked Lists and arrays depends on the specific requirements of the application. If frequent modifications and dynamic resizing are expected, Linked Lists is a better choice. On the other hand, if random access and memory efficiency are crucial, arrays are more suitable.

****

**APPLICATIONS OF LISTS**

* Image viewer - Previous and next images are linked and can be accessed by the next and previous buttons.
* Previous and next page in a web browser - We can access the previous and next URL searched in a web browser by pressing the back and next buttons since they are linked as a linked list.
* Music Player - Songs in the music player are linked to the previous and next songs. So you can play songs either from starting or ending of the list.
* GPS navigation systems- Linked lists can be used to store and manage a list of locations and routes, allowing users to easily navigate to their desired destination.
* Robotics- Linked lists can be used to implement control systems for robots, allowing them to navigate and interact with their environment.
* Task Scheduling- Operating systems use linked lists to manage task scheduling, where each process waiting to be executed is represented as a node in the list.
* Image Processing- Linked lists can be used to represent images, where each pixel is represented as a node in the list.
* File Systems- File systems use linked lists to represent the hierarchical structure of directories, where each directory or file is represented as a node in the list.
* Symbol Table- Compilers use linked lists to build a symbol table, which is a data structure that stores information about identifiers used in a program.
* Undo/Redo Functionality- Many software applications implement undo/redo functionality using linked lists, where each action that can be undone is represented as a node in a doubly linked list.
* Speech Recognition- Speech recognition software uses linked lists to represent the possible phonetic pronunciations of a word, where each possible pronunciation is represented as a node in the list.
* Polynomial Representation- Polynomials can be represented using linked lists, where each term in the polynomial is represented as a node in the list.

**POLYNOMIAL ADT**

A Polynomial ADT (Abstract Data Type) is a data structure that represents a polynomial and provides a set of operations to perform on it.

**Node of Polynomial**

Coefficient

Exponent

Next

**Example:**

**3x2 + 5x +7**



**Consider the following two Polynomials**

Polynomial 1: 3x3 + 4x2 + 2x + 5

Polynomial 2: 5x3 + 2x2 + 3

Polynomial Linked List 1



**Polynomial Linked List 2**



**5x3+ 2x2 + 0x1 +3**

**Steps to Add Polynomials**

**1.Initial Pointer**

**Ptr1 points to (3,3) in polynomial 1**

**Ptr2 points to (5,3) in polynomial 2**

**2.Compare Exponent**

**Both exponents are equal**

**Add the coefficients: 3+5=8**

**Results: 8x3**

3.Move to the next terms

Ptr 1 points to (4,2)

Ptr 2 points to (2,2)

Both exponents are equal

Add the coefficients: 4+2=6

**Results: 8x3 + 6x2**

4.Move to the next terms

Ptr 1 points to (2,1)

Ptr 2 points to null

Both exponents are equal

**Results: 8x3 + 6x2 + 2x**

**5.Move to the next terms**

Ptr 1 points to (5,0)

Ptr 2 points to (3,0)

Both exponents are equal

**Results: 8x3 + 6x2 + 2x + 8**

**Steps for the Algorithm**

1. **Create two Linked List**

Represent each polynomial using a linked list

Each node in the linked list contains: Coefficient, Exponent

1. **Initialize Pointers**

Set two pointers ptr1,ptr2 to the head nodes of the two linked lists representing the polynomials.

1. **Traverse the Lists.**

While both pointers are not null

If exponent is not equal

Add the coefficient of the current node and store the results in the new linked list.

Move both pointers to the next node.

If exponent of ptr1 is greater

Copy the term at ptr1 into the result list because it has a higher component.

Move ptr1 to the next node.

If exponent of ptr2 is greater

Copy the term at ptr2 into the result list because it has a higher component.

Move ptr2 to the next node.

1. **Handle remaining terms**

After one list is exhausted, append the remaining terms form the other list to the result.

1. **Return the result**

The new linked list will represent the sum of the two polynomials.

**PSEUDOCODE**

Function addpolynomials(poly1,poly2);

Set ptr1 = poly1.head

Set ptr2 = poly2.head

While ptr1 is not null AND ptr2 is not null;

If ptr1.exponnet == ptr2.exponnet;

Sum = ptr1.coefficient + ptr2.coefficient

If sum !=0;

Append (sum,ptr1.exponnet) to result

Move ptr1 and ptr2 to the next nodes

Else if ptr1.exponnet > ptr2.exponent;

Append (ptr1.coefficient,ptr1.exponent) to result

Move ptr1 to the next node

Else

Append (ptr2.coefficient, ptr2.exponent) to result

Move ptr2 to the next node

While ptr1 is not null;

Append (ptr1. Coefficient,ptr1.exponent ) to result

Move ptr1 to the next node

While ptr2 is not null;

Append (ptr2. Coefficient,ptr2.exponent ) to result

Move ptr to the next node

Return result

**Declaration of Linked list implementation of Polynomial:**

struct poly{

{ int coeff;

int power;

struct poly \*next;

} \*list1, \*list2, \*list3;

**RADIX SORT**

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

Example:

For the given unsorted list of elements, 236, 143, 26, 42, 1, 99, 765, 482, 3, 56, we need to perform the radix sort and obtain the sorted output list

**Step 1**

Check for elements with maximum number of digits, which is 3. So we add leading zeroes to the numbers that do not have 3 digits. The list we achieved would be

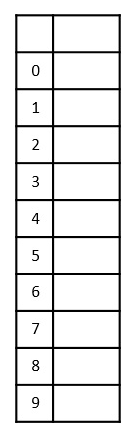
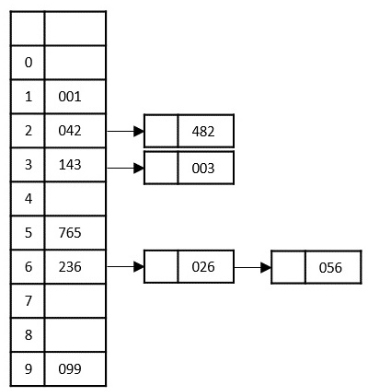


**Step 2**

Construct a table to store the values based on their indexing. Since the inputs given are decimal numbers, the indexing is done based on the possible values of these digits, i.e., 0-9.

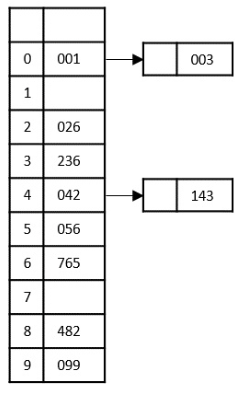
**Step 3**

Based on the least significant digit of all the numbers, place the numbers on their respective indices.

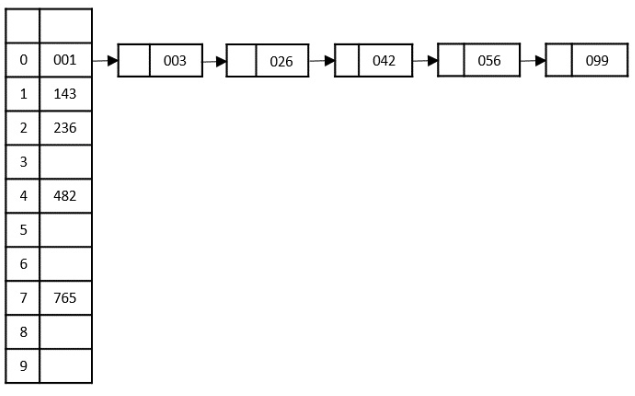
**Step 4**

The order of input for this step would be the order of the output in the previous step. Now, we perform sorting using the second least significant digit.

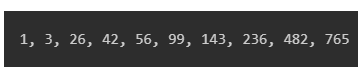


The order of the output achieved is 001, 003, 026, 236, 042, 143, 056, 765, 482, 099.

**Step 5**



The final sorted output is



**MULTI LIST**

A Multilist in data structures is a linked data structure where each node can have multiple pointers (links), not just one "next" pointer like in a regular linked list. It’s often used when data needs to be organized in multiple dimensions rows and columns or when each node needs to be part of more than one linked list simultaneously.

* It is a special type of List
* It contains two or more pointers to other nodes.
* It is used to organize multiple orders of one set of elements.

Properties

* It is an integrated list (Single,Double,Circular)
* All the nodes are integrated in using links of pointers.
* Linked nodes are connected with related data.
* Nodes contain pointers from one structure to the other.