DIPLOMA WALLAH

(Your **One Stop Hub** For Diploma Resources)

DATA STRUCTURES

WITH PYTHON

Complete Notes Based on Full Syllabus

- Diploma Engineering

4th Semester

www.diplomawallah.in

Notes prepared by **Sangam**

Diploma Wallah Sangam

unit-05

* Linear (arrays)

Data is arranged in a sequential manner, and each element has a unique predecessor and successor (except first & last).

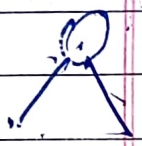
Examples:
- Arrays
- Linked Lists (though linked list uses pointers, logically it's still linear)
- Stack
- Queues.

Runtime Requirements
- Access by index : O(1) (fast - direct indexing)
- Insertion / Deletion (middle) : O(n) (because shifting elements is needed)
- Search (unsorted) : O(n)
- Search (sorted) : O(log(n)) with binary search.

* Space Requirements:
- Arrays :- fixed size, continuous memory block.
- Waste memory if unused space exists.
- Need reallocation if full.

* When to use
- When the number of elements is known in advance.
- When random access is required (eg. accessing 5th element instantly).
- When cache-friendliness is important (arrays are stored in contiguous memory.)

Non-linear Data Structure (pointer-based)
Data is not stored sequentially; elements are connected via pointers/references. One elements can connect to multiple others.



Ex.
- Trees (Binary Tree, BST, Heap)
- Graphs
- Linked Structure with branching.

* Run Time Requirements
- Traversal/Search: O(n) for general trees; O(log n) for balanced BST.
- Insertion/deletion: O(log n) for balanced BST, O(1) for some heap operations.
- No direct indexing - must follow pointers.

* Space Requirements:
- Required extra memory for storing pointers/references.
- may use non-contingous memory blocks, so less cache friendly.

* When to use
- When data has hierarchial or network relationship (eg- family tree)
- When the number of elements is dynamic (keep changing)
- When quick insertion/deletion is more important than random access.

Feature	Linear (Array)	Non-Linear (pointer)
memory	Contiguous	Non-Contiguous
Access Time	O(1) direct access	O(n) traversal
insertion/del.	O(n) shifting required	O(log n) or O(1) (depends)
Extra space	No extra space for pointers.	Extra space for pointers.
Structure when to use	Sequential / Fixed size, random access	Hierarchical / Network / Dynamic, complex relations.

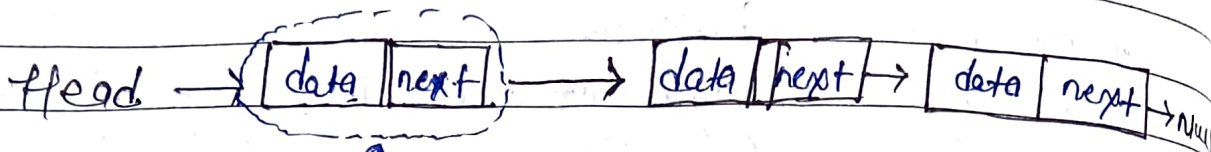
* Linked List *

A linked list is a linear data structure where data elements (nodes) are stored in separate memory locations and are connected to each other using pointers.

Each node contains:

1. Data :- the actual value
2. pointer :- (link) - the address of the next node in the sequence.

* unlike arrays, linked lists do not require contigous memory.



ex -

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

Types:

1. Singly Linked List :- each node points to the next node ; last node points to Null.
2. Doubly linked list :- each node has pointers to both previous and next nodes.
3. Circular Linked list :- last node points back to the first node.

Examples :-

- Images are arranged in a sequence.
- clicking Next shows the next image (follows next pointer).
- clicking previous (if using doubly linked list) show the previous image.

image1 → image2 - image3 → Null.

Application

- Dynamic Memory Allocation:— Linked list can grow/shrink at runtime without reallocation.
- Implementation of data structures:— Stacks, Queues, Graph adjacency lists.
- Undo/Redo in Editors:→ maintain history of operations.
- Navigation System:— Forward/backward page navigation in browsers.
- media players:— playlist where songs/videos are linked.
- Image Viewers:— Browse forward/backward through images.

Diploma Wallah

Sayan

unit-06

Singly Linked List

1. Creating Nodes

→ A node contains two parts:-

- data
- Next

Examples:-

```
class Node:
    def __init__(self, data):
        self.data = data  # Store value
        self.next = None  # pointer to next node

# Create nodes
node1 = Node(10)
node2 = Node(20)
node3 = Node(30)

# Link nodes
node1.next = node2
node2.next = node3

# Head points to first node
head = node1
```

[10 | next] → [20 | next] → [30 | None]

Eg

```
def print_list(head):
    temp = head
    while(temp):
        print(temp.data, end = " -> ")
        temp = temp.next
    print("None")
```

print(head)
list

2. Traversing nodes

Traversal means visiting each node in sequence. Starting from head until None is reached.

Ex.

```
def traverse(head):
    current = head
    while current:
        print(current.data, end = " → ")
        current = current.next
    print("None")
traverse(head)
```

O/P → 10 → 20 → 30 → None.

3. Searching for a Node

We check each node's data until we find the key or reach the end.

Ex

```
def search(head, key):
    current = head
    while current:
        if current.data == key:
            return True
        current = current.next
    return False
print(search(head, 20))   # True
print(search(head, 50))   # false.
```

4. prepending Nodes (Insert at Beginning)

Concept:

To insert at the start.

1. Create a new node
2. point its next to current head.
3. update head to the new node.

Ex

```
def prepend(head, data):
    new_node = Node(data)
    new_node.next = head
    return new_node   # new head
head = prepend(head, 5)
traverse(head)
```

o/p →

5 → 10 → 20 → 30 → None

5. Removing a Node

Concept:- To delete a node:

1. find the node with the matching value.
2. update the previous node's next to skip it
3. If deleting the head, just move head to head.next.

Ex-

```
def delete_node(head, key):
    current = head
    prev = None
    while current:
        if current.data == key:
            if prev:    # deleting middle / last node
                prev.next = current.next
            else:
                head = current.next
            return head
        prev = current
        current = current.next
    return head
head = delete_node(head, 20)
traverse(head)
```

O/P → 5 → 10 → 30 → None

6. Linked List Iterator

We can create an iterator to loop through a linked list using for.

```
Class LinkedListIterator:
    def __init__(self, head):
        self.current = head
    def __iter__(self):
        return self

    def __next__(self):
        if self.current:
            data = self.current.data
            self.current = self.current.next
            return data
        else:
            raise StopIteration
# usage
for value in LinkedListIterator(head):
    print(value)
```

O/P → 5
10
30

Advantage
- Dynamic size, grows / shrinks easily.
- Efficient insertion / deletion at any position.

Disadvantage
- Extra memory for next pointer.
- Sequential access only (no random access like arrays)