Day 04

D3 Recap

- Implementation of Stack and Queue

Stack -> LIFO

- Push
- Pop
- Peek
- isFull - Array (Static Data type)
- Isempty -

Queue -> FILO

- Enqueue
- Dequeue
- Peek
- size
- isempty/isnull

In [10]:

```python
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, data):
        self.queue.append(data)

    def dequeue(self):
        data = self.queue[0]
        del self.queue[0]
        return data

#     what is the element added first or enqueued
    def peek(self):
        return self.queue[0]

    def size(self):
        return len(self.queue)

    def isempty(self):
        return self.queue == []

    def isnull(self):
        if self.size() == 0:
            return True
        return False
```

In [11]:

```python
qu = Queue()
```

In [12]:

```python
qu.size()
```

Out[12]:

```
0
```

In [13]:

```python
qu.isempty()
```

Out[13]:

```
True
```

In [14]:

```python
qu.isnull()
```

Out[14]:

```
True
```

In [7]:

```python
qu.enqueue(1)
qu.enqueue(2)
qu.enqueue(3)
qu.enqueue(4)
```

In [8]:

```python
qu.peek()
```

Out[8]:

```
1
```

In [9]:

```
qu.size()
```

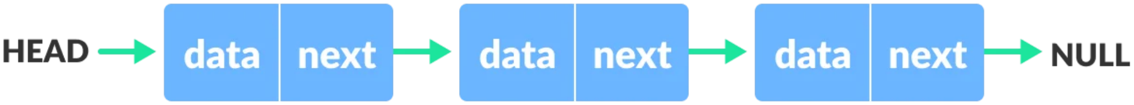
Out[9]: 4

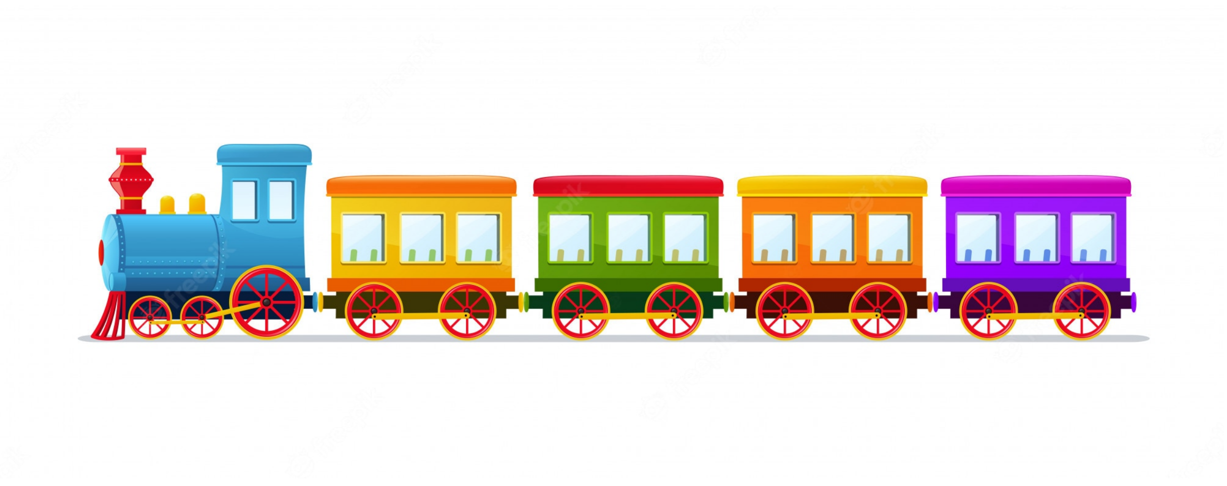
Linked List

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

In [16]:

```
li = [1,2,3,4]
```





li = [1, 2, 3, 4, 5]

li.pop(2)

li = [1, 2, 4, 5]

li[2] li.insert(1, 6)

[1, 6, 2, 4, 5]

1 add1, 2 add2, 3 add3, 4 None

1 add2, 3 add3, 4 None

Advantages Of Linked List:

- **Dynamic data structure:** A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory.
- **No memory wastage:** In the Linked list, efficient memory utilization can be achieved since the size of the linked list increase or decrease at run time so there is no memory wastage and there is no need to pre-allocate the memory.
- **Implementation:** Linear data structures like stacks and queues are often easily implemented using a linked list.
- **Insertion and Deletion Operations:** Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element only the address present in the next pointer needs to be updated.

Disadvantages Of Linked List:

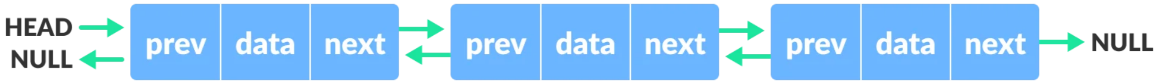
- **Memory usage:** More memory is required in the linked list as compared to an array. Because in a linked list, a **pointer** is also required to store the address of the next element and it requires extra memory for itself.
- **Traversal:** In a Linked list traversal is more time-consuming as compared to an array. Direct access to an element is not possible in a linked list as in an array by index.
- **Reverse Traversing:** In a singly linked list reverse traversing is not possible, but in the case of a doubly-linked list, it can be possible as it contains a pointer to the previously connected nodes with each node. For performing this extra memory is required for the back pointer hence, there is a wastage of memory.
- **Random Access:** Random access is not possible in a linked list due to its random memory allocation.

Types of Linked List

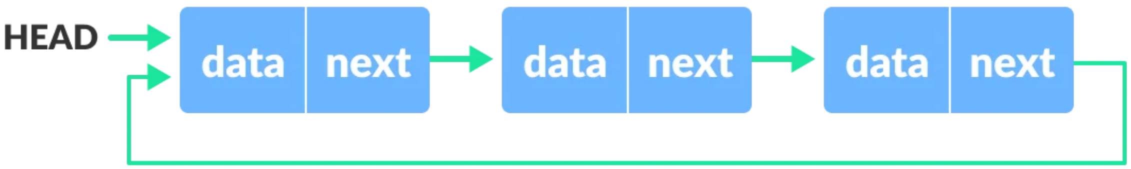
There are three common types of Linked List.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Doubly Linked List



Circular Linked List



Basic operations on Linked Lists:

- Deletion
- Insertion
- Search
- Display

In [26]:

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

# class LinkedList:
```

In [27]:

```python
ele = Node(1)
```

In [38]:

```python
ele
```

Out[38]:

```
<__main__.Node at 0x1c4fa908c70>
```

In [28]:

```python
ele.data
```

Out[28]:

```
1
```

In [30]:

```python
print(ele.next_node)
```

```
None
```

In [34]:

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

class LinkedList:
    def __init__(self):
        self.head = None
        self.num_of_nodes = 0
    def append(self, data):

        self.num_of_nodes += 1

        # pointer of every node
        new_node = Node(data)

        # no elements in linked list
        if not self.head:
            self.head = new_node

        # add 1 ele with none and self.head 1st ele address
        # add 2 ele add 1 ele address 2nd ele then 2 nd none and data of
        # atleast 1 item in linked list
        else:
            new_node.next_node = self.head
            self.head = new_node

    def size(self):
        return self.num_of_nodes
```

In [35]:

```
ll = LinkedList()

ll.size()
```

Out[35]:

```
0
```

In [36]:

```
ll.append(5)
ll.append(6)
ll.append(4)


ll.size()
```

Out[36]:

```
3
```

In [37]:

```
ll
```

Out[37]:

```
<__main__.LinkedList at 0x1c4fa2416a0>
```

In [ ]:

```
li = [4 add6, 6 add5, 5 none]
```