Linked Lists

Created	@August 3, 2022 8:00 PM
Class	
Type	
Materials	
Reviewed	☐

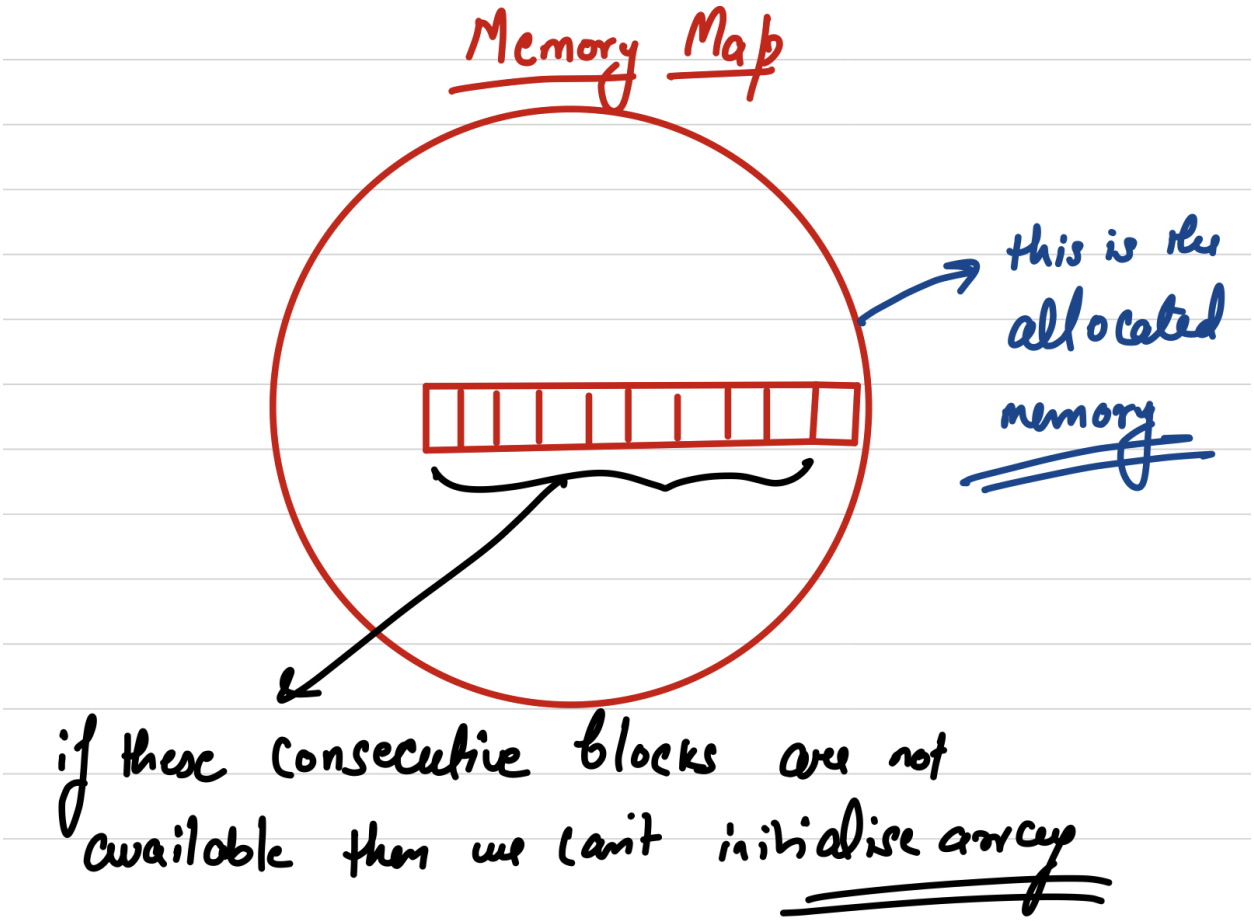
We already know that Arrays as a data structure exists. Using arrays we can implement a lot of algorithms and we can extend the capabilities of an array in multiple dimensions as well.

Then why do we need another data structure ?

There are cases when arrays might not perform in the best way possible. So for handling these use cases we need to introduce more data structures that might help us to overcome problems of arrays.

Problems with arrays:

- Arrays consume contiguous cross-section of memory. Due to this, there can space issues with the program. So, we know that if we initialise a 10 length array, it will consume 10 consecutive adjacent blocks of memory. But let’s say we don’t have 10 consecutive blocks of memory available but instead 10 blocks are scattered, then arrays will not be helpful as they can’t work with scattered memory blocks.



- In an array, we can efficiently (average constant) add a new element or remove an old element to the last of the array. But we don't have any efficient way to add a new element to the start or remove an old element from the start of the array. The inbuilt functions available in most of the languages which can add or remove an element from start of the array works in `O(n)` time to add / remove a single element.

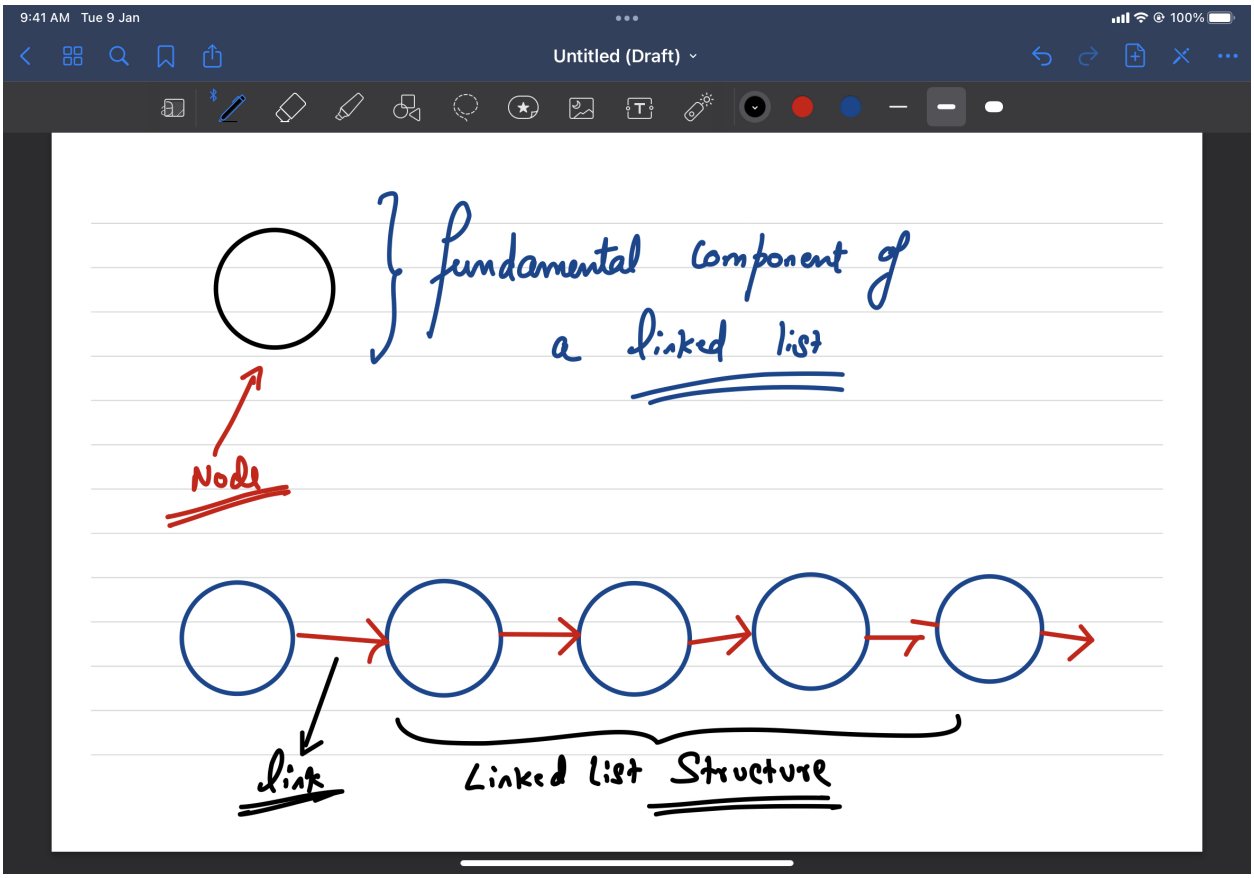
Linked Lists to the rescue

In order to overcome the above few disadvantages of the arrays, we introduce a new data structure called as Linked Lists.

As the name suggests, `Linked Lists` → refers to linkage of different memory blocks leading to a list like structure. By saying Linked we refer to some kind of a chain like

structure. So just like any ordinary chain structure in Linked List also we have linkage of memory blocks.

Let's see how a normal linked List might look like



In a linked list, `Node` is the fundamental component which actually stores data. Node is just like any ordinary object (just like of a person class we can have objects, similarly objects of a Node class will create linked lists)

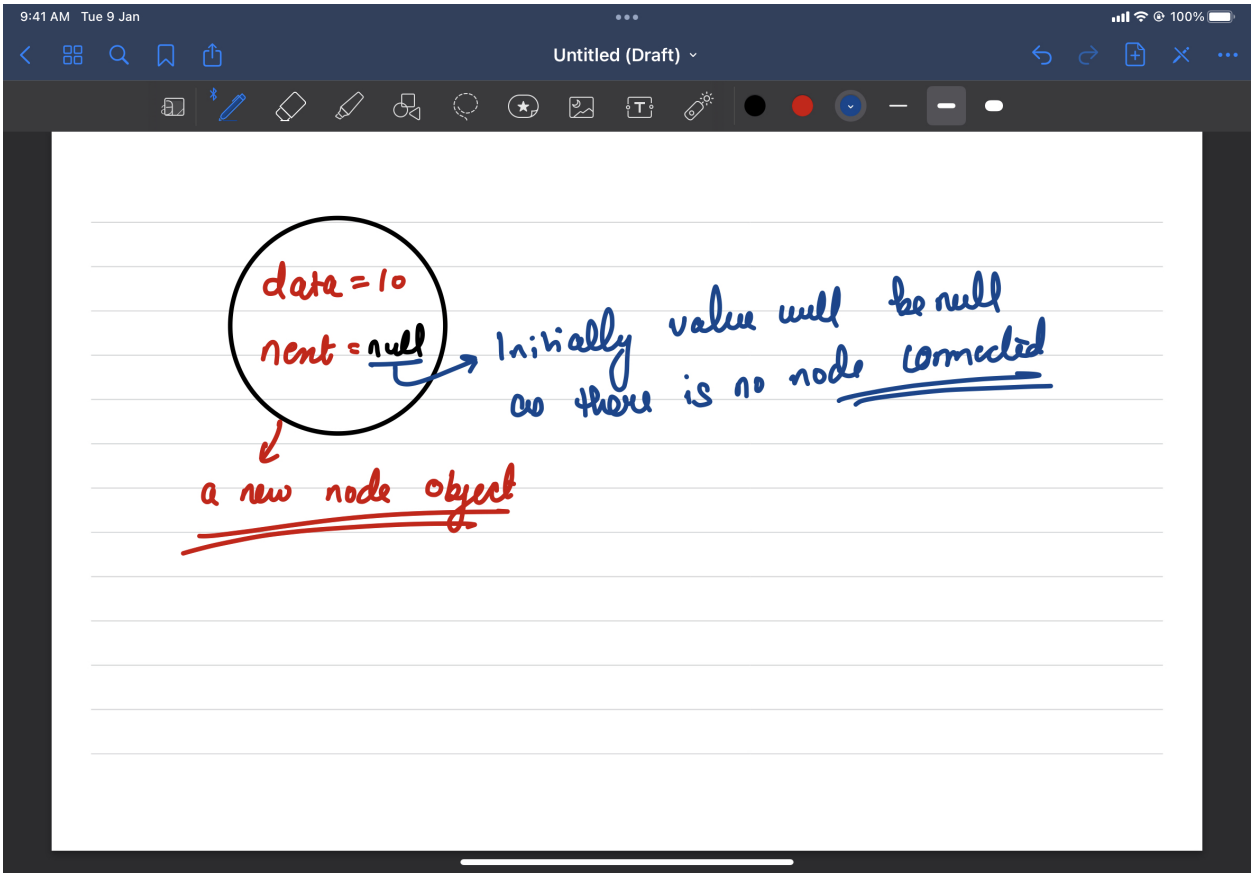
Node is an object, that is the fundamental building block of a linked list and multiple nodes linked together forms a linked list. Because it's an ordinary object we can define a blue print for it through classes, and using the node class we can create multiple instances of node, then connect them and form linked list.

What properties node will be having ?

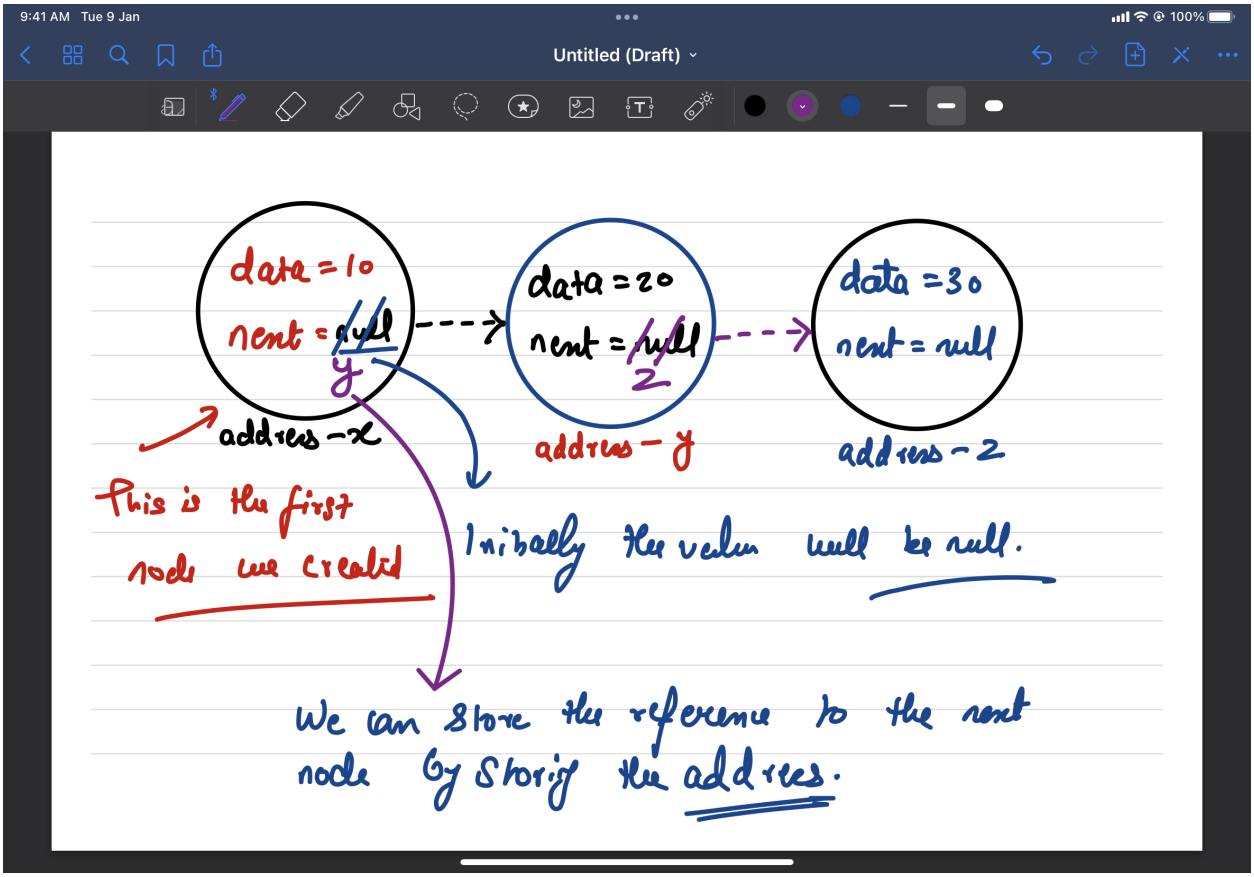
Because node is the fundamental block that will store the data, so there should be a `data` property. Apart from this, we need a mechanism to store the link to the next node.

Because linked lists are chain of nodes, so we need a property to establish a chain or a link between two nodes.

We can store the reference to the next chained node object in side the class, let's call this property `next`




```
class Node {
  constructor(d) {
    this.data = d; // data parameter represents the actual data stored in node
    this.next = null; // this will be a ref to the next node connected to the curr node
  }
}
```

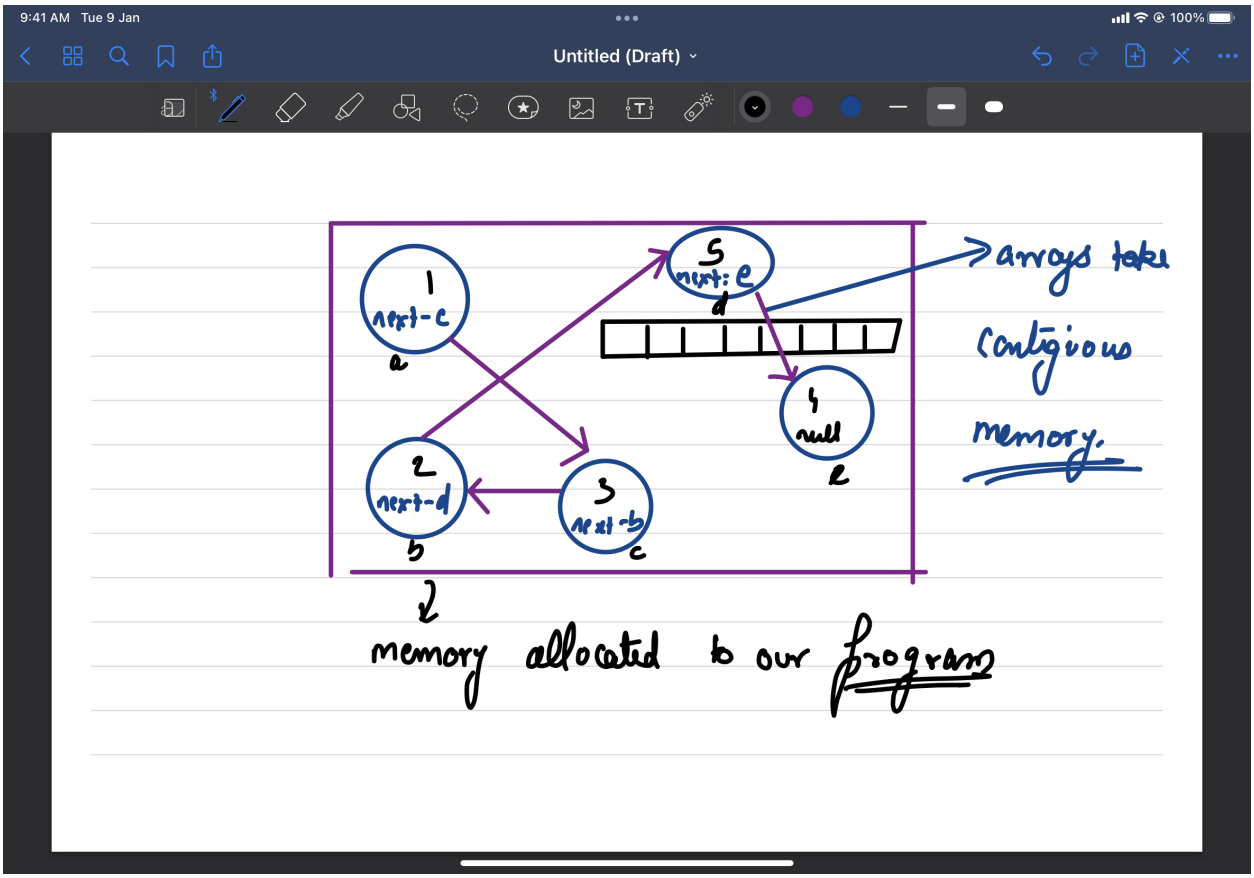


When we want to connect multiple nodes with each other, we can store the address of the next node in the current node's `next` property. So, whenever we say `node.next` we are actually referring to the next node to the current node.

If we want to move from one node to other, we can use this `next` property to do the same.

How does it help ?

So now we know the structure of a linked list that how it looks like, so because we are creating new nodes every time to store new data values, and these nodes are just ordinary objects created as instance of `Node` class, they will be always created at some random available memory location. So, no more restriction of contiguous memory requirement.



So this is going to help us in optimal usage of the memory space available. For example, if we have enough memory to store 10 data values but not in contiguous locations, instead the memory might be scattered so in this case `Linked Lists` will be very helpful because every node of a linked list is create at a random available memory location as it is just an ordinary object hence it does optimal and sensible usage of memory.

Advantage with addition of a node at start

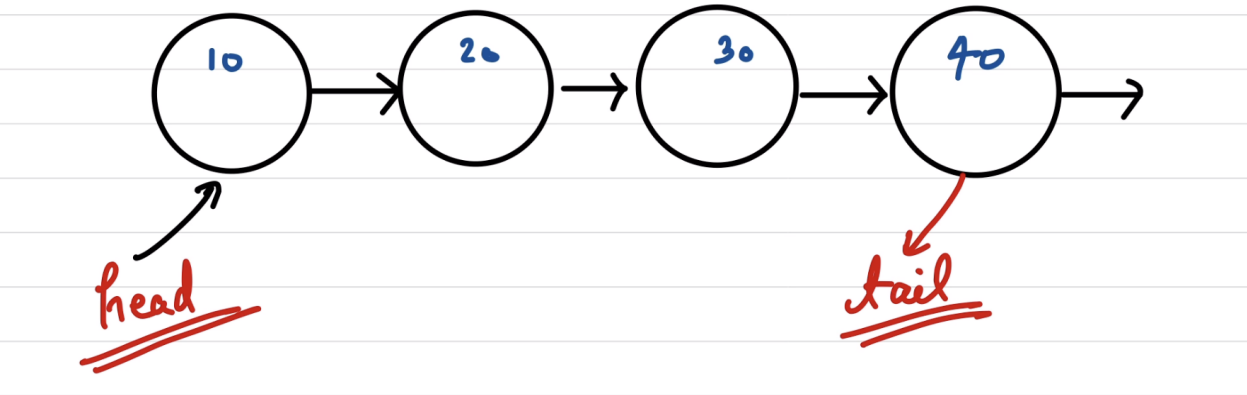
So in arrays, addition or deletion of a data point from starting was in-efficient because, arrays have strict space constraint, so we need to shift all the values forward / backward for doing addition / removal of a data.

But this is not the case with Linked List, because every node is created at a new memory location, we don’t need to do the shiftings. All we have to do is create a new

node, and store the reference of the previously available first node to the next property of current node.

Head and Tail of Linked List

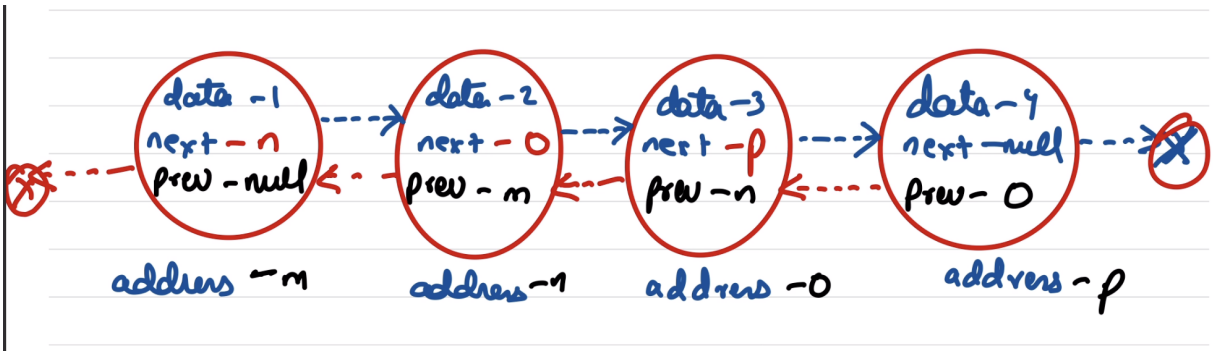
The first node of a Linked List which is accessible to us, is called as `Head` of the linked list and the last node after which no other node is connected is called as the `Tail` of the linked list.



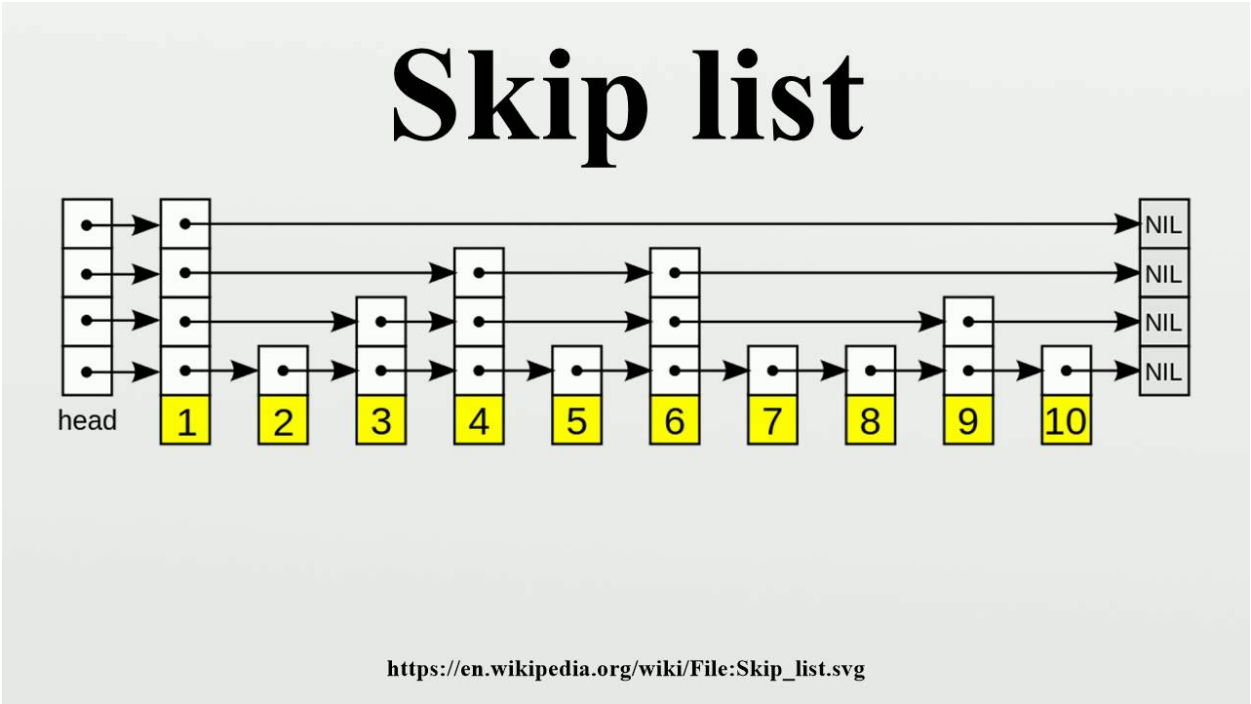
Types of Linked Lists

There are multiple different type of Linked Lists available

- Singly Linked List : In a singly linked list there is only one direction of the chain. We can move from head to the next node then to the next node and so on till the node, i.e. from any current node we have access to the next node only, so we can only move forward and we cannot come backwards, i.e. there is no way to come to the previous node of the linked lists. The above all images are examples of singly linked lists.
- Doubly Linked List: In a double linked lists there is two directions of the chain. We can move from head to the next node then to the next node and even come back to the previous node. So for a doubly linked lists inside the node class we have one more property apart from data and next called as `prev`. This prev property stores reference to the previous node.



- Circular Linked List: In a circular linked lists, tail is connected to head, so there is no node with next as null.
- Skip lists: In a skip list, one node is not only connected to the next node but there will be some connection to few later available nodes as well.



Implementation of Linked Lists

For implementing Linked Lists we need to write the following operations:

- AddAtHead (addition of a node at head)
- AddAtTail (addition of a node at tail)

- AddAt (addition of a node in between)
- RemoveHead
- RemoveTail
- RemoveAt (remove a particular node in between)
- Display

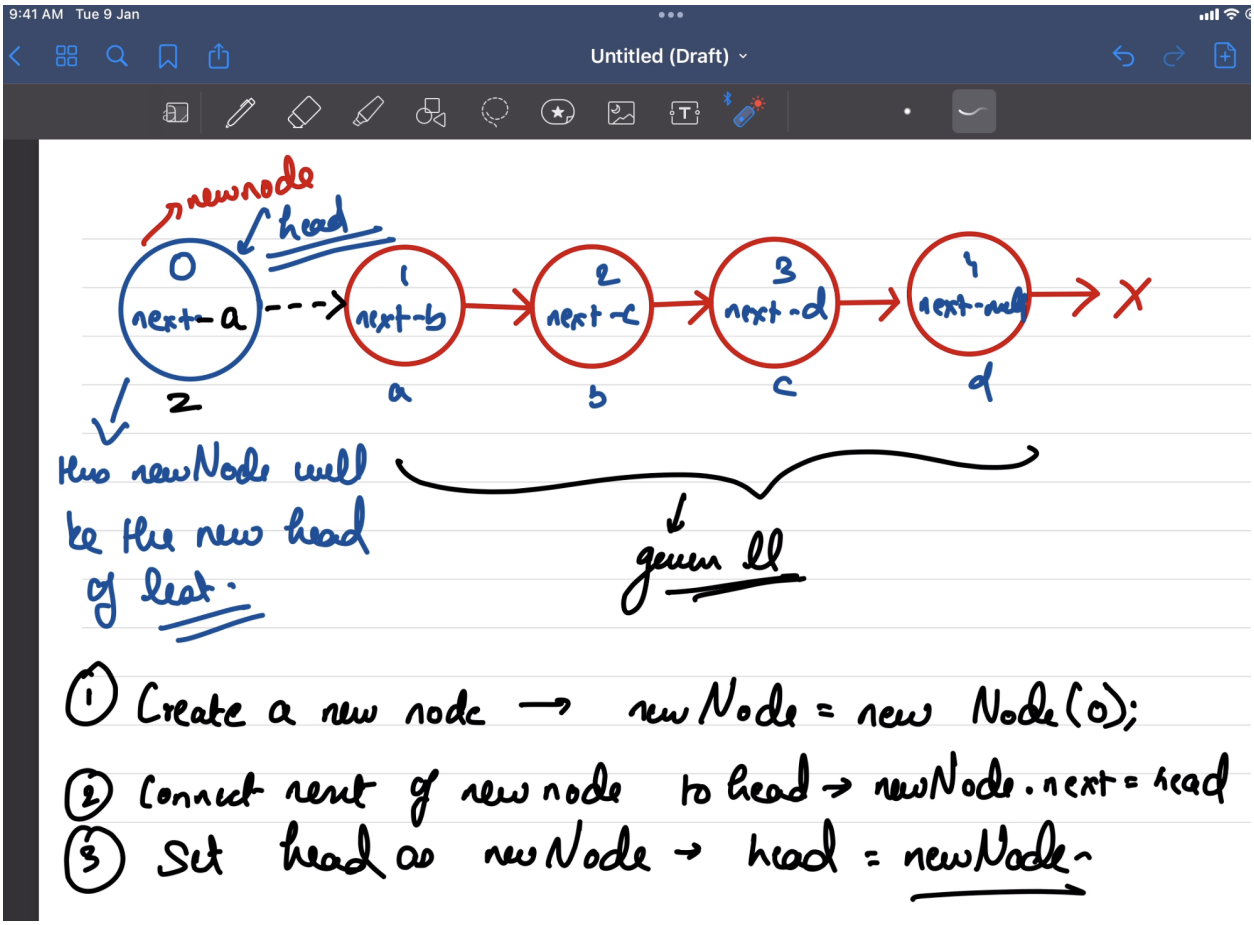
These functions we need to implement for implementing a linked list.

Linked List Class

In most of the situation while operating on a linked list we will be having access to the head only. And initially when we have not attached any node to the linked list head will be null.

```
class LinkedList {
  constructor() {
    // when we initialise a new linked list head will be empty
    this.head = null;
  }
}
```

Addition of Node at head of a Linked List



To add a new node to the head of a linked list,

- we will first create a new node object, this object will be having next property as null.
- set the next property of the new node object to the previous head
- update the head of the linked list to be equal to the new node object

```
class Node {
  constructor(d) {
    this.data = d; // data parameter represents the actual data stored in node
    this.next = null; // this will be a ref to the next node connected to the curr node
  }
}

class LinkedList {
  // singly
  constructor() {
    // when we initialise a new linked list head will be empty
    this.head = null;
  }
```

```
  addAtHead(data) {
    let newNode = new Node(data); // created a new node
    newNode.next = this.head; // set the next of new node to head
    this.head = newNode; // update the head to the new node
  }

  display() {
    console.log(this.head);
  }
}

let ll = new LinkedList();
ll.addAtHead(5);
ll.addAtHead(4);
ll.addAtHead(3);
ll.addAtHead(2);
ll.addAtHead(1);
ll.display();
```

give me 2 mins guys