Linked Lists

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

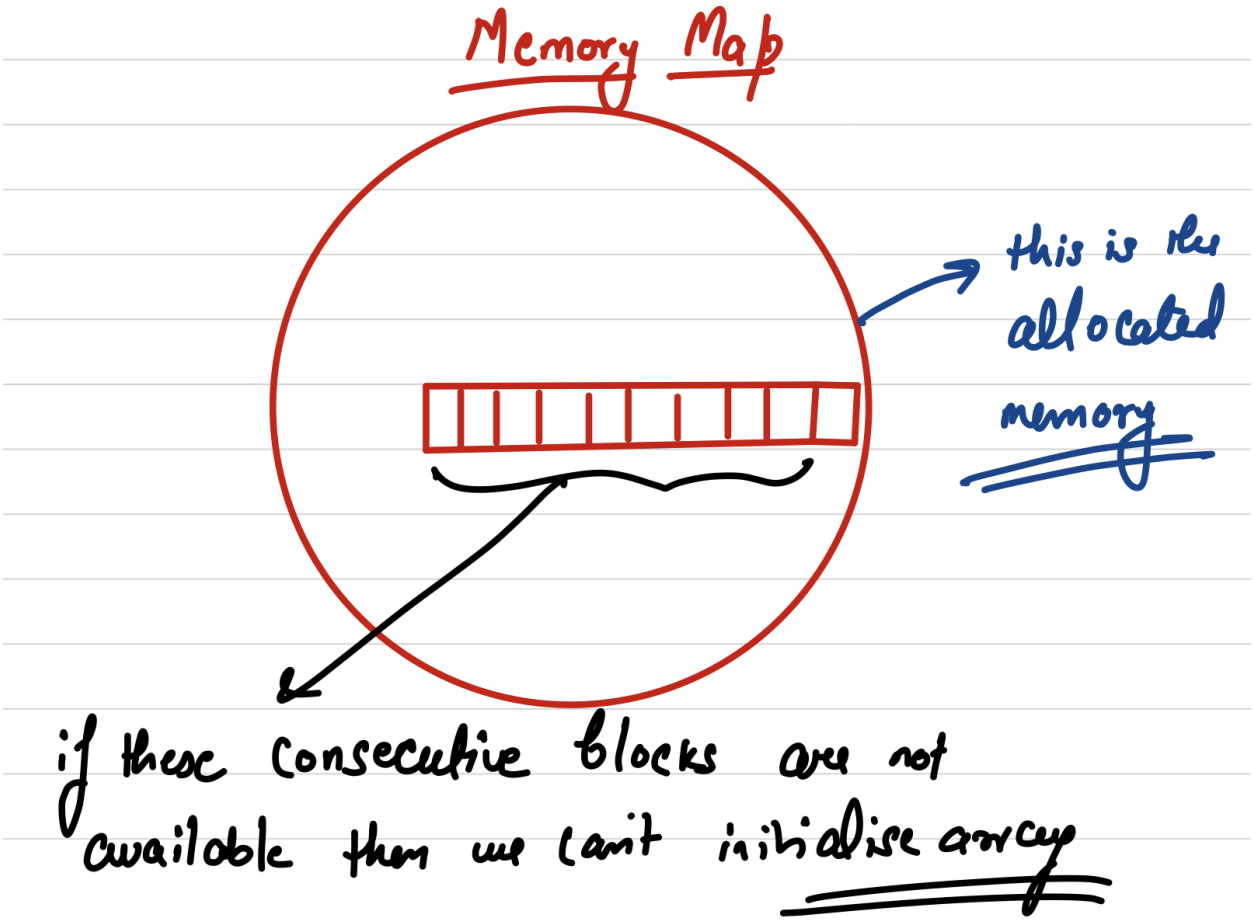
We already know that Arrays as a data structure exists. Using arrays we can implement a lot of algorithms and we can extend the capabilities of an array in multiple dimensions as well.

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.

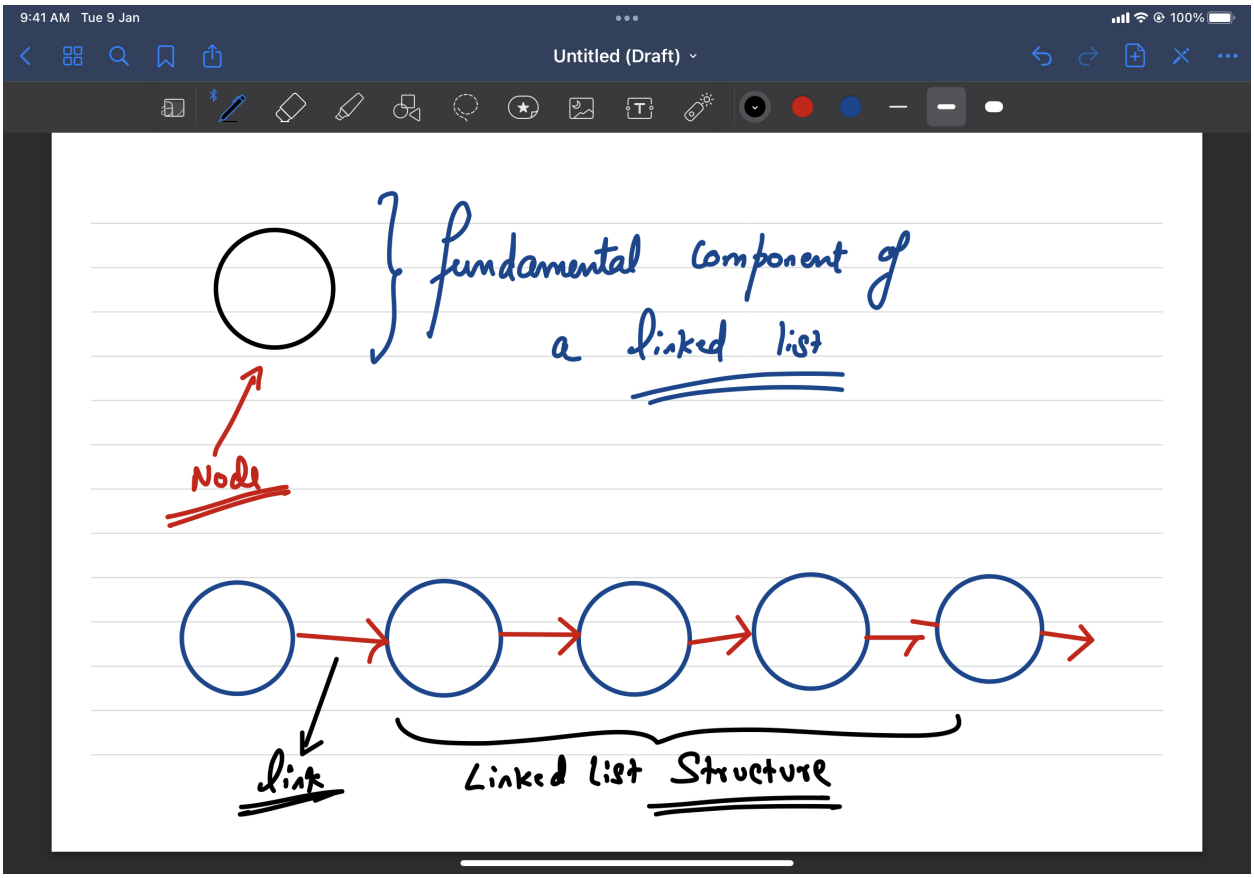
Linked Lists to the rescue

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)

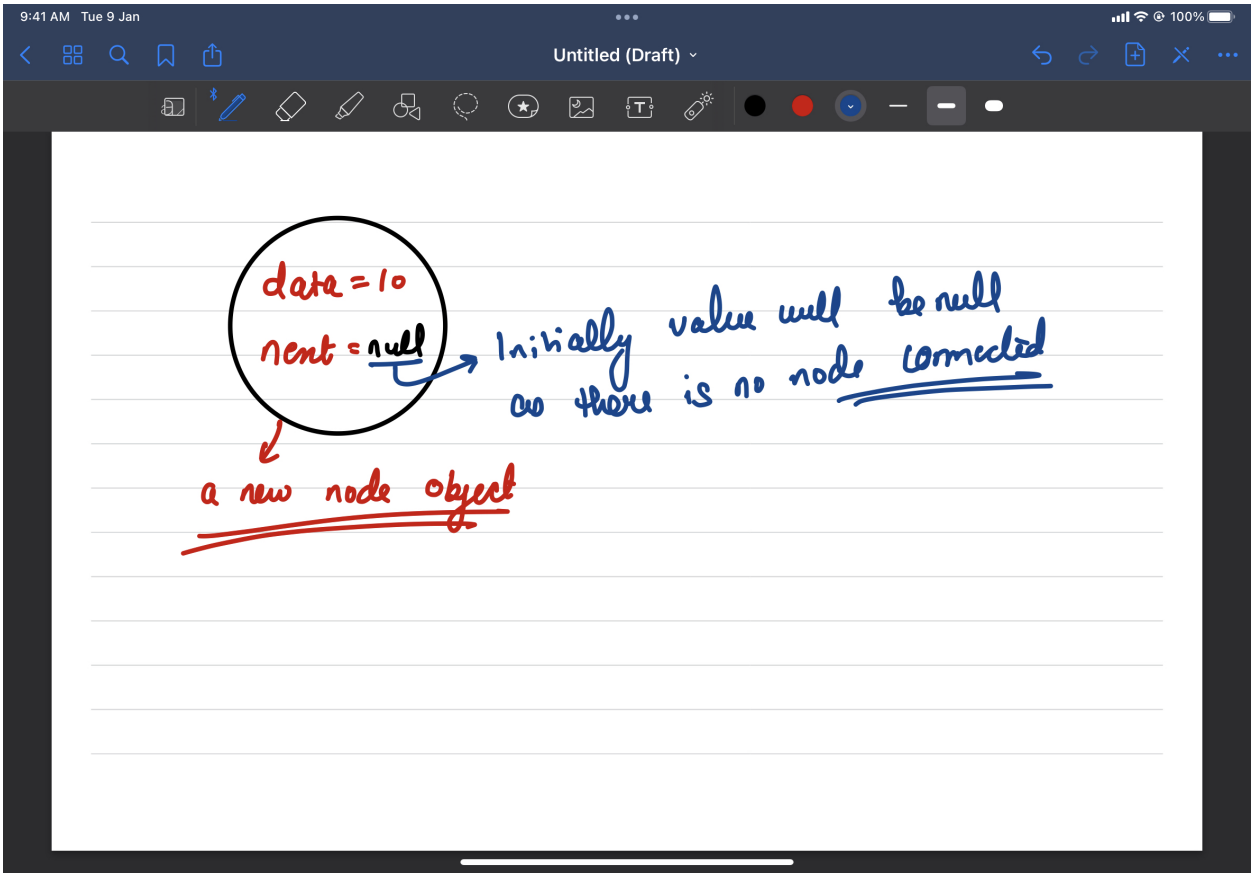
Node is an object, that is the fundamental building block of a linked list and multiple nodes linked together forms a linked list. Because it's an ordinary object we can define a blue print for it through classes, and using the node class we can create multiple instances of node, then connect them and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

What properties node will be having ?

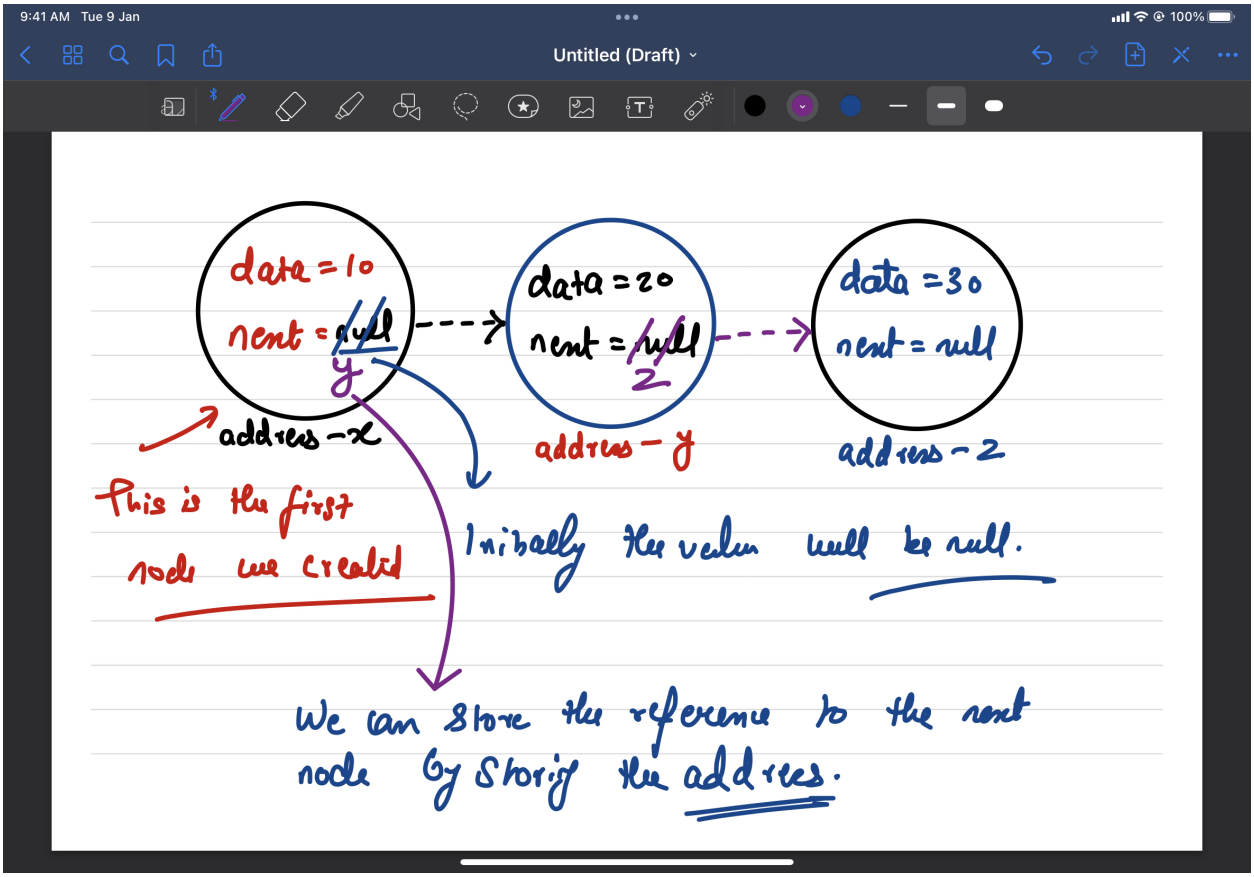
Because node is the fundamental block that will store the data, so there should be a `data` property. Apart from this, we need a mechanism to store the link to the next node.

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
  }
}
```

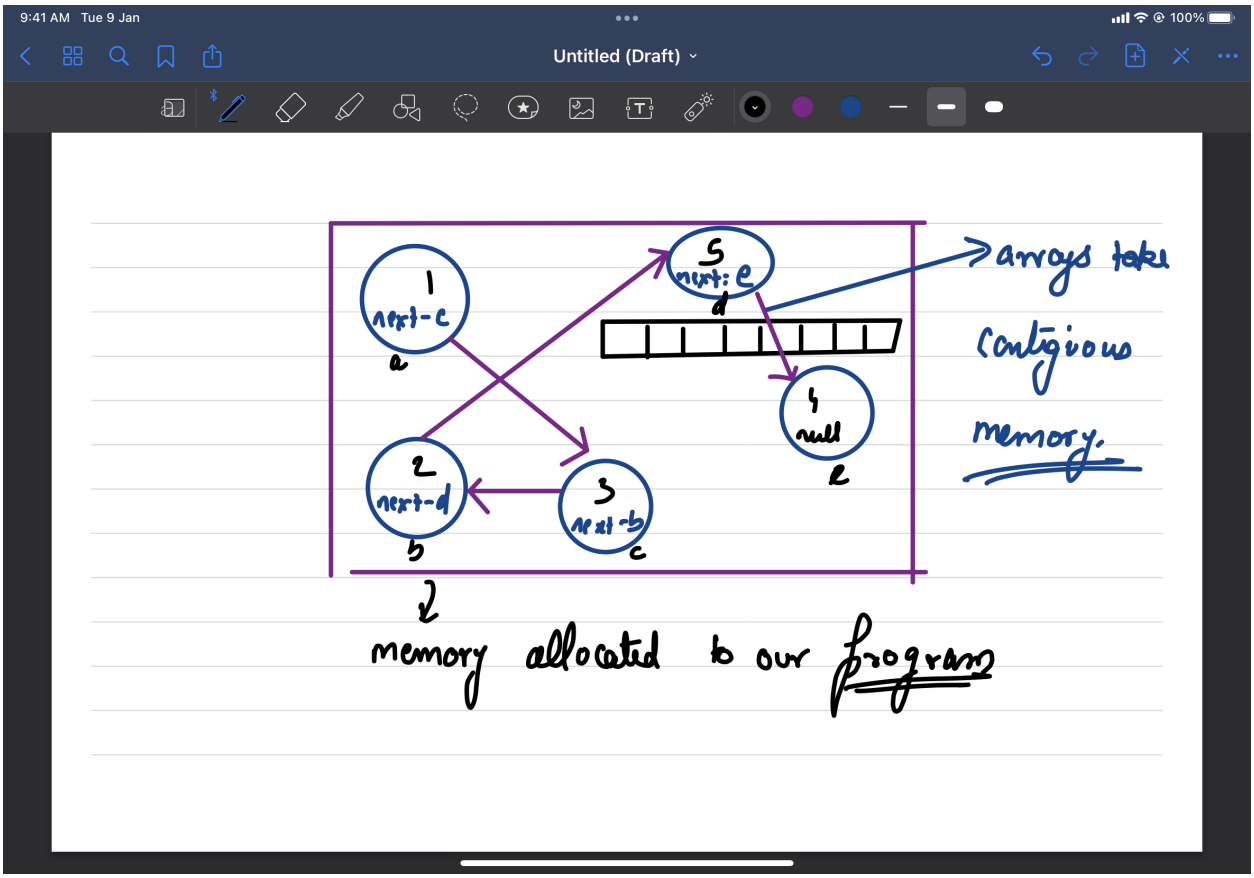


When we want to connect multiple nodes with each other, we can store the address of the next node in the current node's `next` property. So, whenever we say `node.next` we are actually referring to the next node to the current node.

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

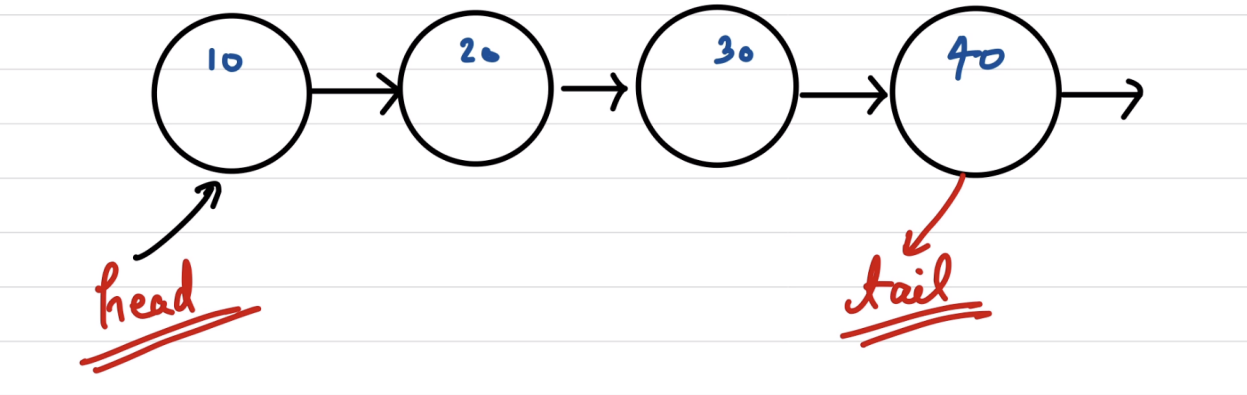
So in arrays, addition or deletion of a data point from starting was in-efficient because, arrays have strict space constraint, so we need to shift all the values forward / backward for doing addition / removal of a data.

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

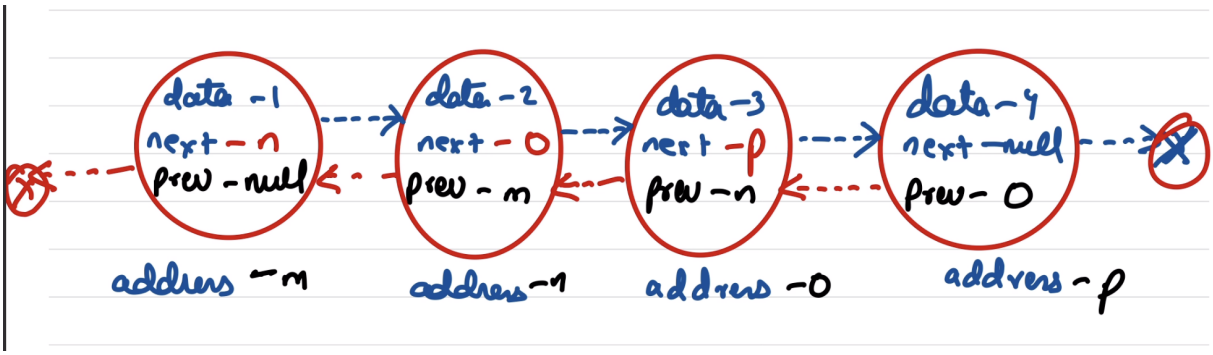
The first node of a Linked List which is accessible to us, is called as `Head` of the linked list and the last node after which no other node is connected is called as the `Tail` of the linked list.



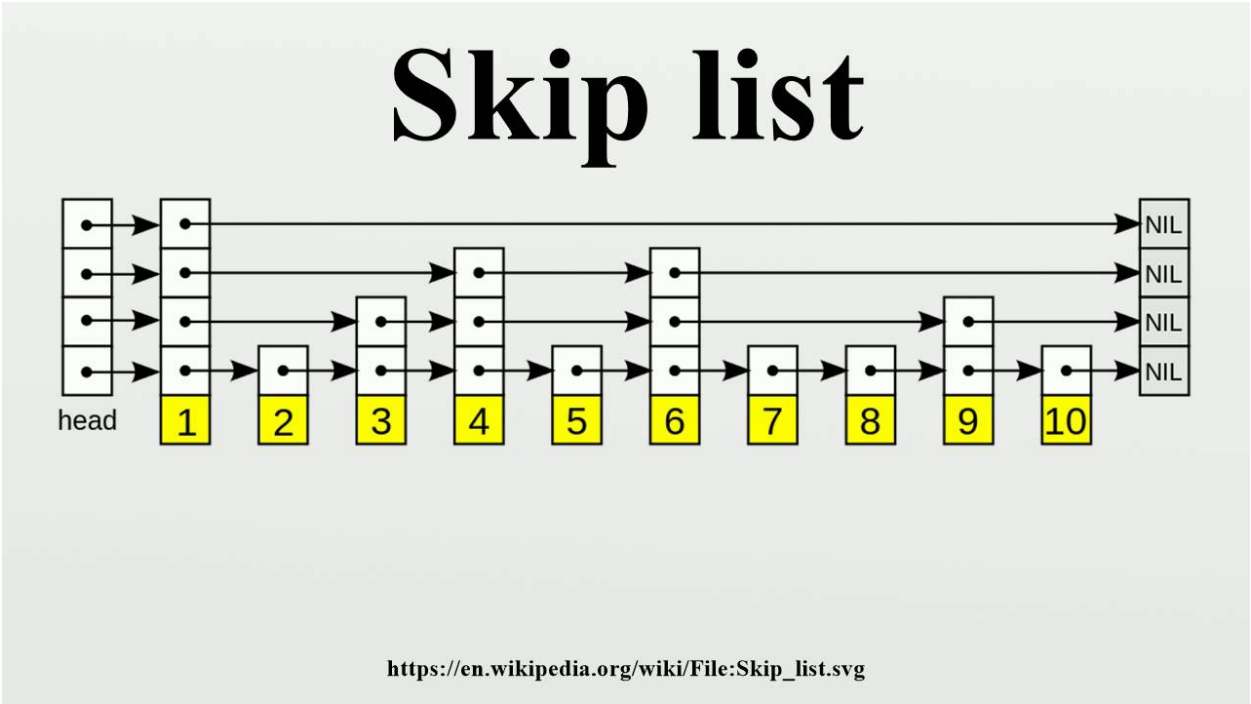
Types of Linked Lists

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

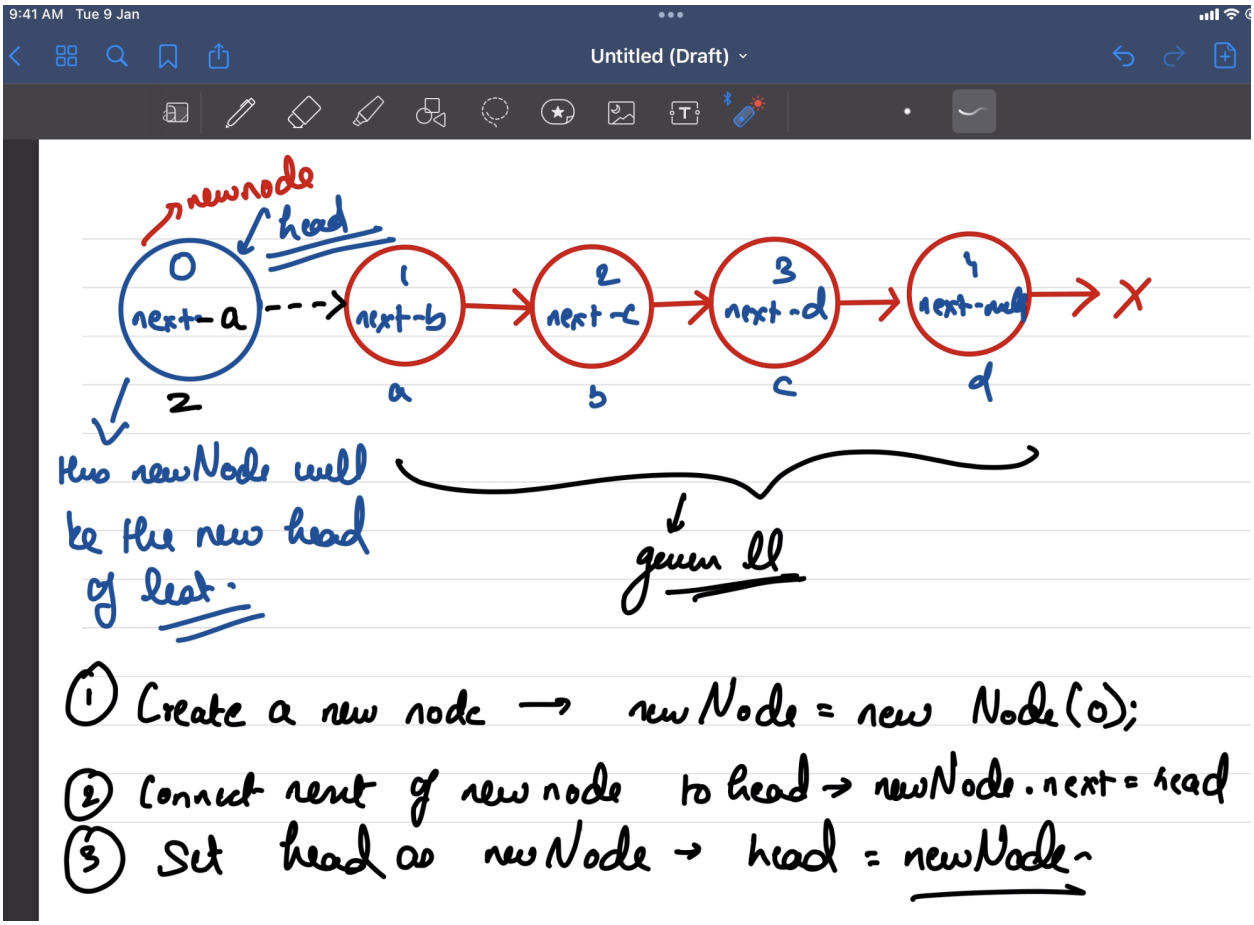
These functions we need to implement for implementing a linked list.

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,

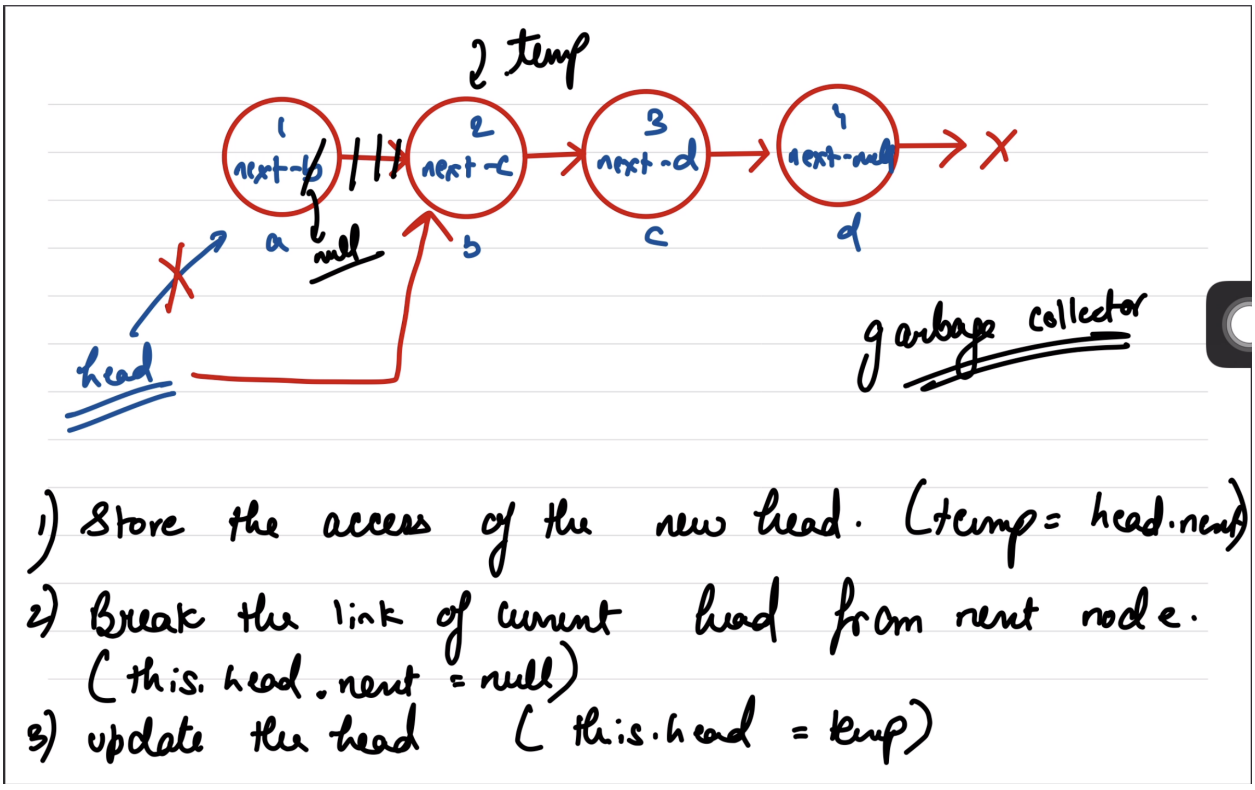
- we will first create a new node object, this object will be having next property as null.
- set the next property of the new node object to the previous head
- update the head of the linked list to be equal to the new node object

Time complexity will be `O(1)` as we are just creating an object and then manipulating pointers and constant space because no other data structure is used

```
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();
```

Removing an element from head of the list:



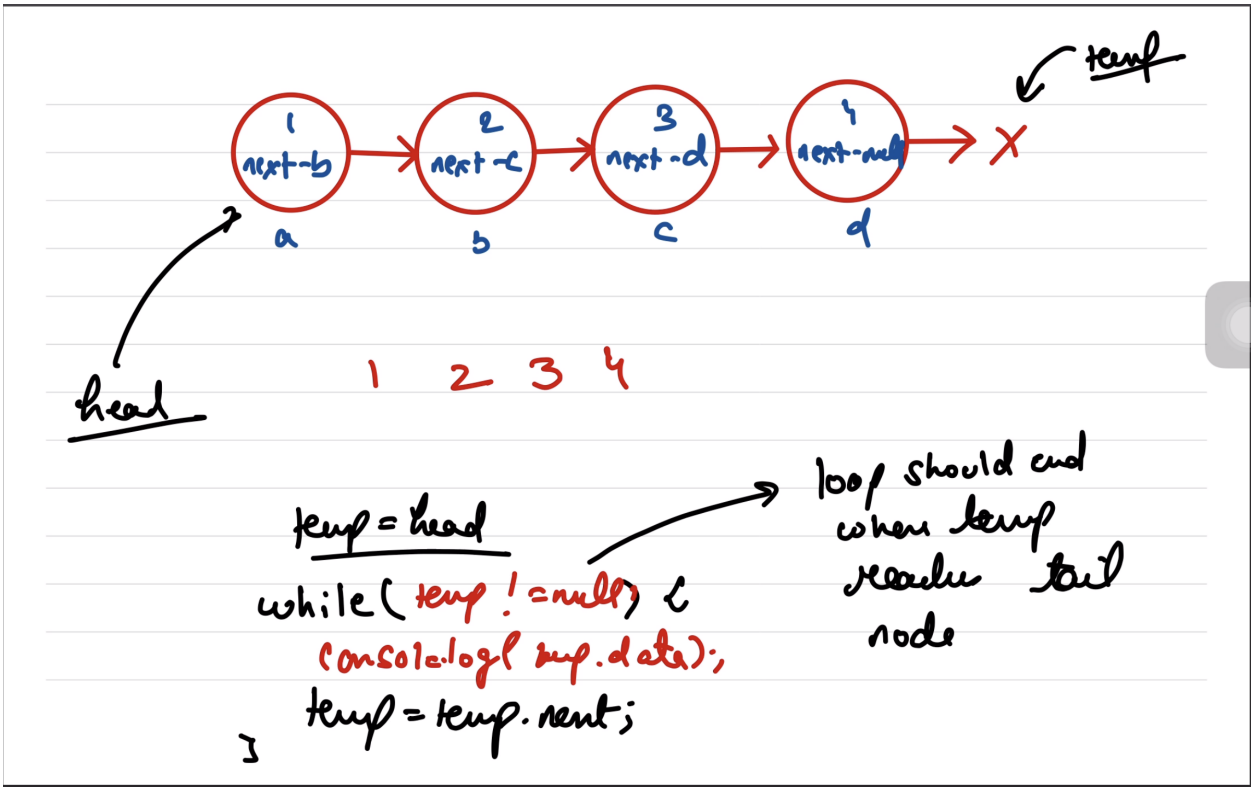
- First store access of the node that will become the next head.
- Break the link of the current head from the remaining list
- update the head

Time complexity will be `O(1)` as we are just manipulating pointers and constant space because no other data structure is used

```
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
  }

  removeAtHead() {
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }

  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();
```

Print the list

If we want to print the data of the nodes of the list, We should now move the head anywhere, as it can lead to loss of access of the current head if we are not careful. Instead prepare a temp variable starting from head, and till the tile temp is not null keep on printing the temp variable and move the temp to the next node.

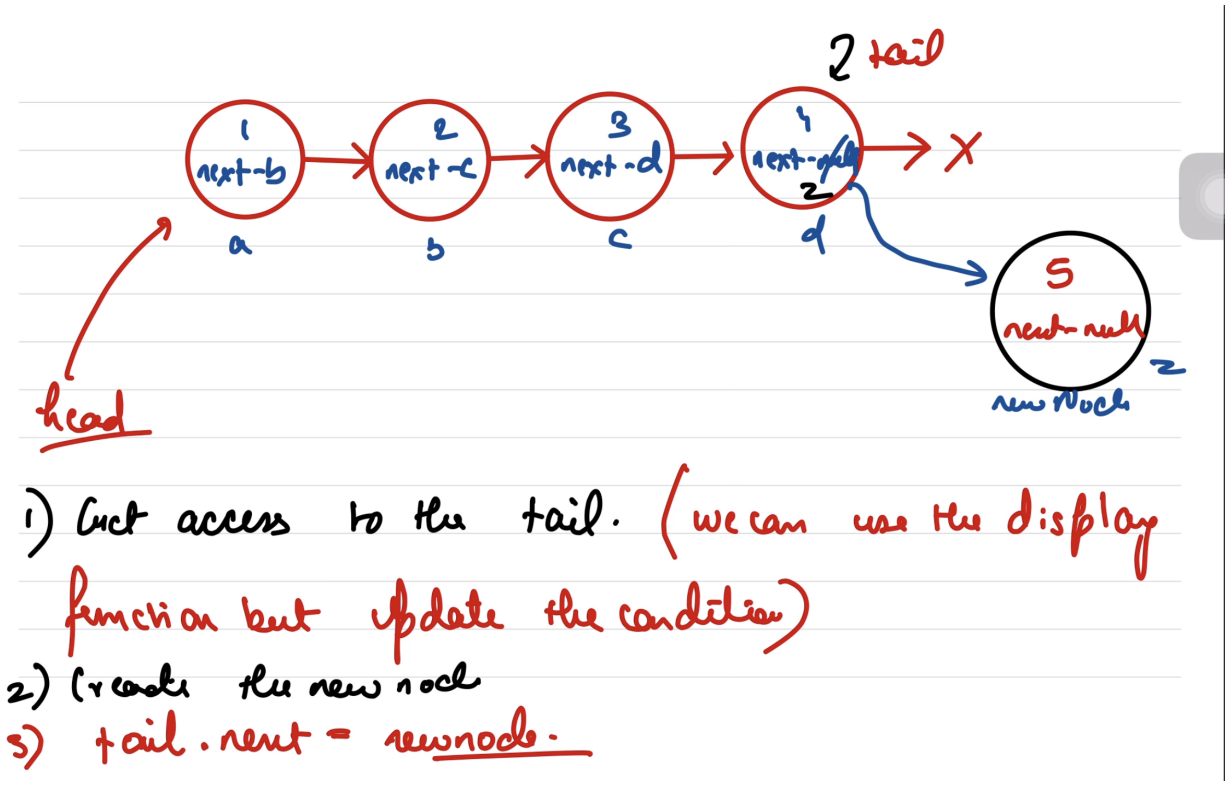



```
class Node {
  constructor(d) {
    this.data = d; // data parameter represents the actual data stored in node
    this.next = null; // this will be a ref to the next node connected to the curr node
  }
}

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
  }

  removeAtHead() {
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }

  display() {
    let temp = this.head;
    while(temp != null) {
      console.log(temp.data);
      temp = temp.next;
    }
  }
}

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

If we change the while condition to `temp.next ≠ null`, then the loop will stop at the tail, and won't print the tail.

Adding a new node at the tail



We can just iterate to the tail of the linked list. In order to do that, we can use the display function's idea with an updated condition that is `temp.next != null` so that we stop the iteration at the tail.

Then we create the new node and attach the next of the tail on the new node.

Time complexity will be `O(n)` because we have to iterate over all the n nodes to get access to tail and space will be constant as no other data structure is used

```
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
  }

  removeAtHead() {
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }

  addAtTail(data) {
    if(this.head == null) { // if ll is empty, addattail is equal to addathead
      this.addAtHead(data);
      return;
    }
    let tail = this.head;
    while(tail.next != null) tail = tail.next;
    let newNode = new Node(data);
    tail.next = newNode;
  }

  display() {
    let temp = this.head;
    while(temp != null) {
      console.log(temp.data);
      temp = temp.next;
    }
  }
}

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

Remove the node at tail

In order to delete the tail node, we need the access to the second last node of the list, and mark it’s next reference to null.

Time complexity will be `O(n)` because we have to iterate over all the n nodes to get access to tail and space will be constant as no other data structure is used

```
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
  }

  removeAtHead() {
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }

  addAtTail(data) {
    if(this.head == null) { // if ll is empty, addattail is equal to addathead
      this.addAtHead(data);
      return;
    }
    let tail = this.head;
    while(tail.next != null) tail = tail.next;
    let newNode = new Node(data);
    tail.next = newNode;
  }

  removeAtAtail() {
    if(this.head == null) return; // empty ll
    if(this.head.next == null) { // only one node in the list
      this.head = null;
      return;
    }
    let temp = this.head;
    while(temp.next.next != null) {
      // the above condition gives us access to second last node
      temp = temp.next;
    }
    temp.next = null;
  }

  display() {
    let temp = this.head;
    while(temp != null) {
      console.log(temp.data);
```

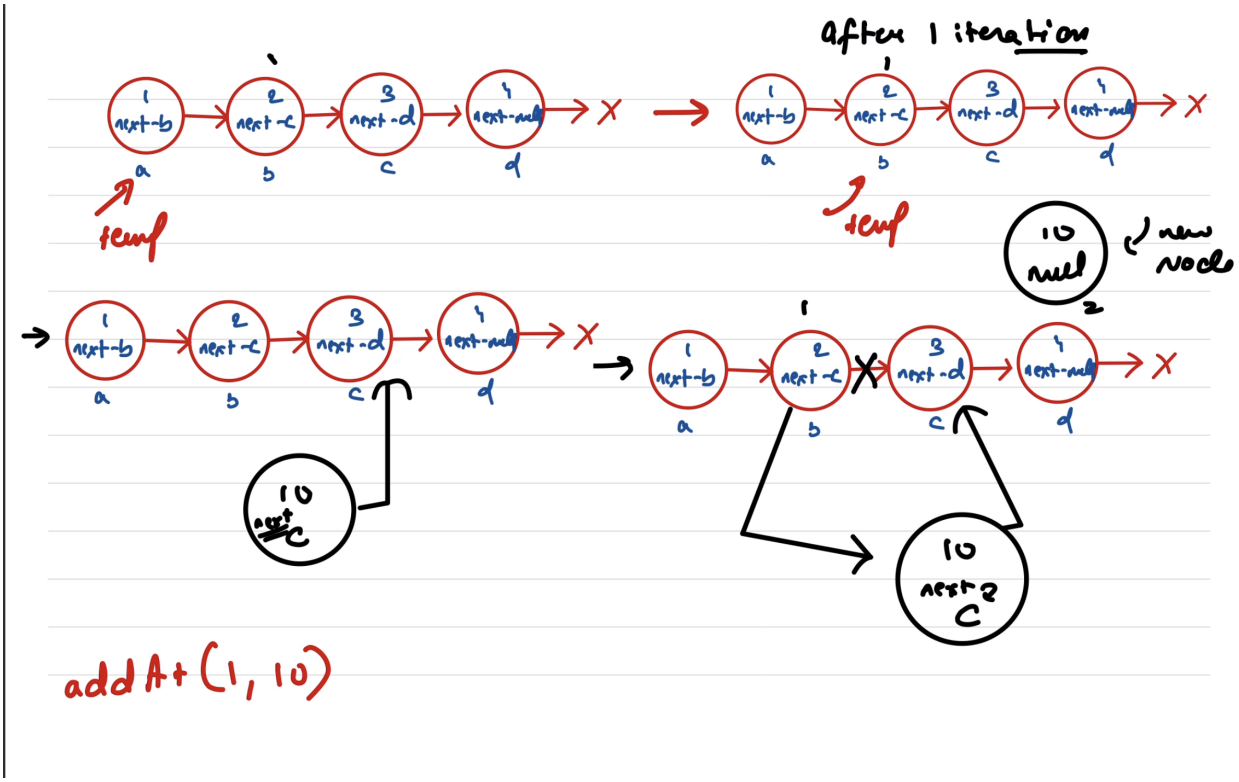
```
      temp = temp.next;
    }
  }
}

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

Add at a position x in the LL

In order to get access to the position x of the LL, we need to iterate x times on the LL, and maintain a pointer temp, that can help us to reach there. While iterating we need to keep a check also that we have not crossed the tail of the LL in case if position of greater than length of the list.

The moment we have access to the position x's node, we can create a new node, and then mark the next of the new node to the next of the xth node, and then update the xth node's next with new node.

Time complexity will be `O(n)` because we might have to iterate over all the n nodes and space will be constant as no other data structure is used




```
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
  }

  removeAtHead() {
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }
```

```
  addAtTail(data) {
    if(this.head == null) { // if ll is empty, addattail is equal to addathead
      this.addAtHead(data);
      return;
    }
    let tail = this.head;
    while(tail.next != null) tail = tail.next;
    let newNode = new Node(data);
    tail.next = newNode;
  }

  removeAtAtail() {
    if(this.head == null) return; // empty ll
    if(this.head.next == null) { // only one node in the list
      this.head = null;
      return;
    }
    let temp = this.head;
    while(temp.next.next != null) {
      // the above condition gives us access to second last node
      temp = temp.next;
    }
    temp.next = null;
  }

  addAt(pos, data) {
    if(this.head == null) {
      this.addAtHead(data);
      return;
    }
    let temp = this.head;
    for(let i = 0; i < pos && temp.next != null; i++) {
      temp = temp.next;
    }
    // now inside temp we have access to the node at pos
    let newNode = new Node(data);
    newNode.next = temp.next;
    temp.next = newNode;
  }

  display() {
    let temp = this.head;
    while(temp != null) {
      console.log(temp.data);
      temp = temp.next;
    }
  }
}

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

```
ll.addAtHead(1);
ll.display();
```

remove at a given position

If we want to remove a node at a particular position, we need to make sure that we have access to the node present on the previous position i.e. pos-1

The moment we have access to the prev node, then `prev.next` is the node to be deleted.

We can mark the `prev.next` as `nodeToBeDeleted` and then mark the next of prev node i.e. `prev.next` equal to `nodeToBeDeleted.next`

And then to isolate the node to be deleted we can do `nodeToBeDeleted.next = null`

Time complexity will be `O(n)` because we might have to iterate over all the n nodes and space will be constant as no other data structure is used

```
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
  }

  removeAtHead() {
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }

  addAtTail(data) {
    if(this.head == null) { // if ll is empty, addattail is equal to addathead
```

```
      this.addAtHead(data);
      return;
    }
    let tail = this.head;
    while(tail.next != null) tail = tail.next;
    let newNode = new Node(data);
    tail.next = newNode;
  }

  removeAtAtail() {
    if(this.head == null) return; // empty ll
    if(this.head.next == null) { // only one node in the list
      this.head = null;
      return;
    }
    let temp = this.head;
    while(temp.next.next != null) {
      // the above condition gives us access to second last node
      temp = temp.next;
    }
    temp.next = null;
  }

  addAt(pos, data) {
    if(this.head == null) {
      this.addAtHead(data);
      return;
    }
    let temp = this.head;
    for(let i = 0; i < pos && temp.next != null; i++) {
      temp = temp.next;
    }
    // now inside temp we have access to the node at pos
    let newNode = new Node(data);
    newNode.next = temp.next;
    temp.next = newNode;
  }

  removeAt(pos) {
    if(this.head == null) return; // LL was empty
    if(this.head.next == null || pos == 0) {
      // either you have a single node or pos is 0
      this.removeAtHead();
      return;
    }
    let prev = this.head;
    for(let i = 0; i < pos - 1; i++) {
      prev = prev.next;
    }
    let nodeToBeDeleted = prev.next;
    prev.next = nodeToBeDeleted.next;
    nodeTobeDeleted.next = null;

  }
```

```
  display() {
    let temp = this.head;
    while(temp != null) {
      console.log(temp.data);
      temp = temp.next;
    }
  }
}

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

Till now we have successfully implemented a full fledged Singly Linked List, what if we have to implement a doubly LL ?

```
class Node {
    constructor(d) {
        this.data = d;
        this.next = null;
        this.prev = null; // this new parameter will get added to the node to have access
 of the prev node
    }
}
```

To add a node at head of DLL

```
addAtHead(data) {
        /**
         * Time: O(1)
         * Space: O(1)
         */
        if(this.head == null) {
            let newNode = new Node(data);
            this.head = newNode;
            return;
        }
    let newNode = new Node(data); // created a new node
    newNode.next = this.head; // set the next of new node to head
        this.head.prev = newNode;
```

```
    this.head = newNode; // update the head to the new node
}
```

This time, we have to mark the previous of the head to the new node and then update the head, because in DLL, we have to arrange prev pointers as well. Now, because we are trying to access `this.head.prev` then there will be a corner case i.e. DLL can be empty, so we have to handle it separately.