CSA201 – Applied Data Structures and Algorithms

Unit 4 Linked List Data Structure

GYALPOZHING
COLLEGE OF INFORMATION TECHNOLOGY

What is an Array?

- An *array* is a data structure consisting of a collection of elements, each identified by at least one array *index* or *key*. Elements are stored in contiguous memory locations, allowing for efficient random access.

4	3	2	7	8	11
[0]	[1]	[2]	[3]	[4]	[5]

- ***Why do we need an Array?***

3 variables

number1

number2

number3

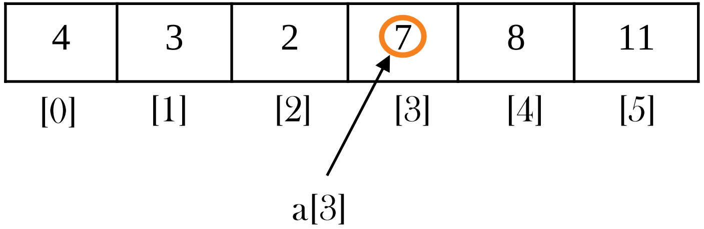
- What if 500 integer?
- Are we going to use 500 variables?

The answer is an **ARRAY**

Types of Arrays

One dimensional array : an array with a bunch of values having been declared with a single index.

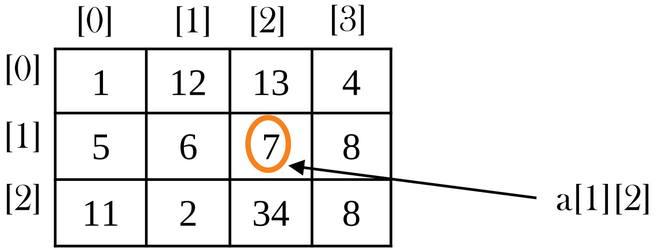
a[i] —> i between 0 and n



4	3	2	7	8	11
[0]	[1]	[2]	[3]	[4]	[5]

Two dimensional array : an array with a bunch of values having been declared with double index.

a[i][j] —> i and j between 0 and n



	[0]	[1]	[2]	[3]
[0]	1	12	13	4
[1]	5	6	7	8
[2]	11	2	34	8

Creating an Array

When we create an array, we:

- Declare - creates a reference to array
- Instantiation of an array - creates an array
- **Initialization/Insertion** - assigns values to cells in array

```
dataType[] arr;
arr = new dataType[];
arr[0] = 1;
arr[1] = 2;
```

```
class ArrayOfIntegers {
    int[] arr;
    ArrayOfIntegers(){
        arr = new int[2];
    }
}

class Main{
    public static void main(String[] args) {
        ArrayOfIntegers obj = new ArrayOfIntegers();
        obj.arr[0] = 1;
1       obj.arr[1] = 2;
        System.out.println(obj.arr[0]);
    }
}
```

Insertion in Array

arr =

4	3	2	7	8	11
[0]	[1]	[2]	[3]	[4]	[5]

arr[3]= 7
arr[5] = 11

```
class ArrayOfIntegers {
    int[] arr;
    int size;
    ArrayOfIntegers(){
        arr = new int[6];
        size = 0;
    }

    void insertion(int element){
2              arr[size++] = element;
    }
}
```

Insertion in Array

arr =

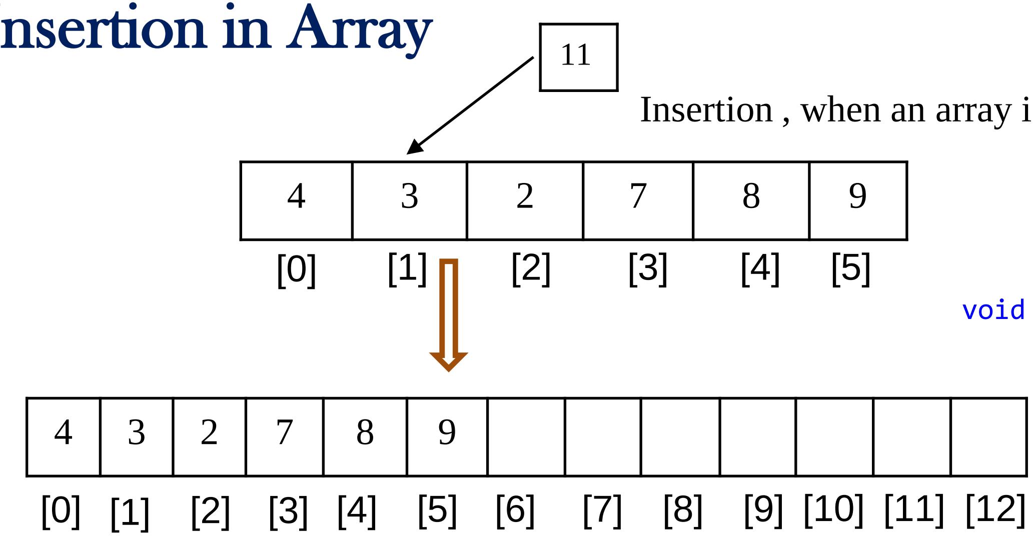
4	3	2	7	8	11
[0]	[1]	[2]	[3]	[4]	[5]

arr[3]= 7
arr[5] = 11

```
class Main{
    public static void main(String[] args) {
        ArrayOfIntegers obj = new ArrayOfIntegers();
        obj.insertion(4);
        obj.insertion(3);
        obj.insertion(2);
        obj.insertion(7);
        obj.insertion(8);
        obj.insertion(11);
    }
}
```

Insertion in Array

Insertion , when an array is full.



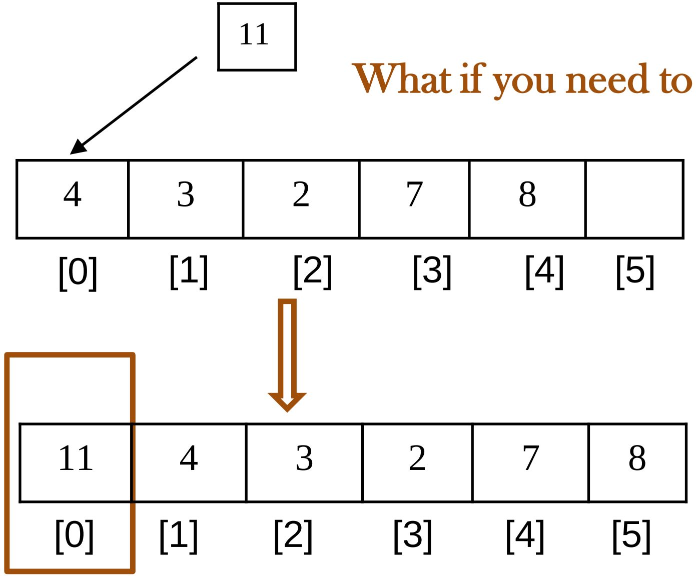
4	3	2	7	8	9
[0]	[1]	[2]	[3]	[4]	[5]

4	3	2	7	8	9							
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

```
void insertion(int element){
        if(arr.length == size){
            int newArr[] = new int[arr.length * 2];
            int i;
            for (i = 0; i < arr.length; i++) {
                newArr[i] = arr[i];
            }
            arr = newArr;
        }
        arr[size++] = element;
    }
```

Insertion in Array

What if you need to add an element at the start?



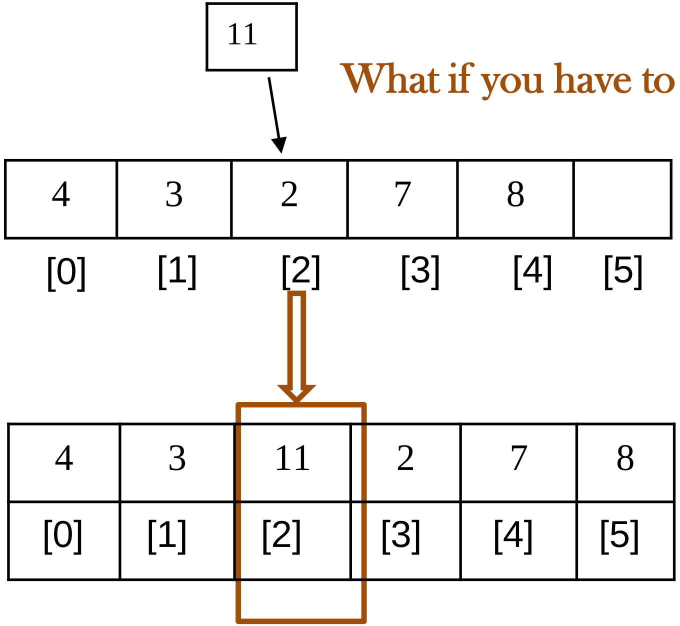
11

4	3	2	7	8	
[0]	[1]	[2]	[3]	[4]	[5]

11	4	3	2	7	8
[0]	[1]	[2]	[3]	[4]	[5]

Insertion in Array

What if you have to add an element at any arbitrary position within an array?

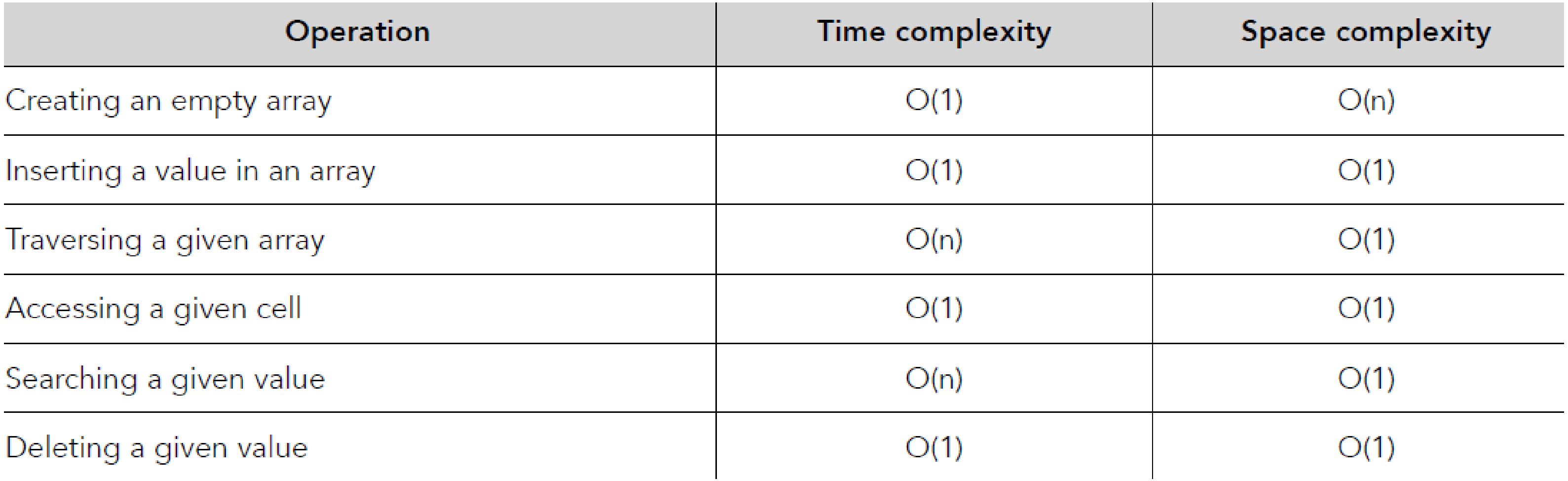


11

4	3	2	7	8	
[0]	[1]	[2]	[3]	[4]	[5]

4	3	11	2	7	8
[0]	[1]	[2]	[3]	[4]	[5]

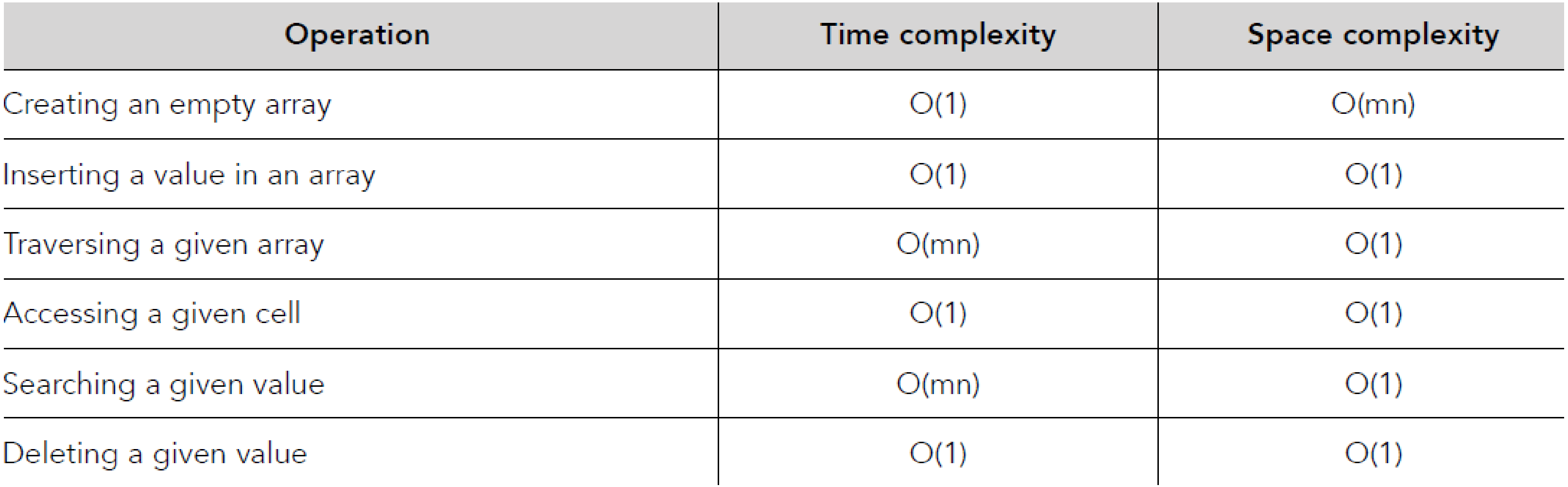
Time and Space Complexity of 1D Arrays



Operation	Time complexity	Space complexity
Creating an empty array	O(1)	O(n)
Inserting a value in an array	O(1)	O(1)
Traversing a given array	O(n)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(n)	O(1)
Deleting a given value	O(1)	O(1)

Note: Complete the implementation of the remaining operations for any given array.

Time and Space Complexity of 2D Arrays



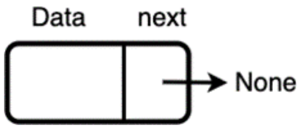
Operation	Time complexity	Space complexity
Creating an empty array	O(1)	O(mn)
Inserting a value in an array	O(1)	O(1)
Traversing a given array	O(mn)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(mn)	O(1)
Deleting a given value	O(1)	O(1)

Note: Complete the implementation of the remaining operations for any given array.

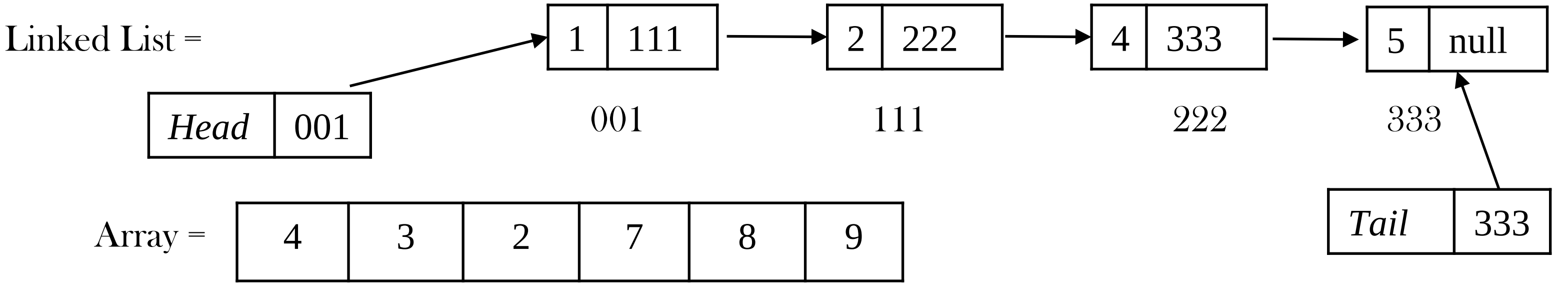
Linked List

What is a Linked List?

- Linked List is a form of a *sequential collection* and *it does not have to be in order.*
- A Linked list is made up of independent *nodes* that may contain any type of *data* and each *node* has a *reference* to the *next node* in the link.
- *Main Concepts*
- Each element of a linked list is called *node*, and every node has two different fields:
  - *Data* - contains the value to be stored in the node.
  - *Next* - contains a reference to the next node on the list.



Linked List vs Array



Linked List =

1	111

001

2	222

111

4	333

222

5	null

333

Head	001

Tail	333

Array =

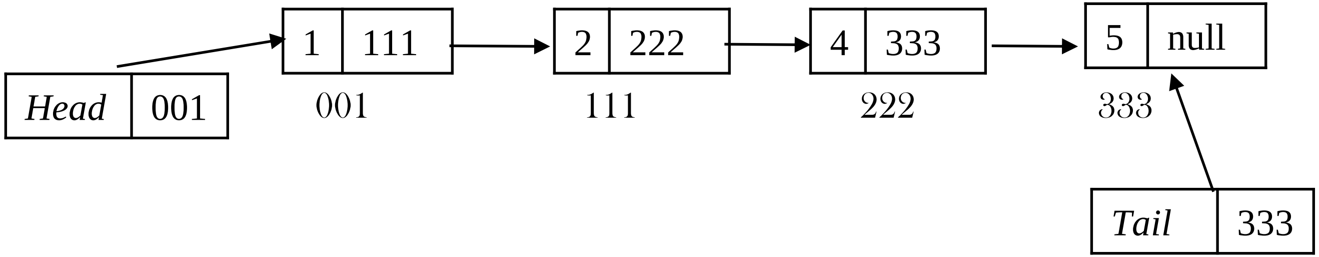
4	3	2	7	8	9

- *Elements* of Linked list are independent objects
- *Variable size* - the size of a linked list is not predefined
- *Random access* - accessing an element is very efficient in ***arrays***

Types of Linked List

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

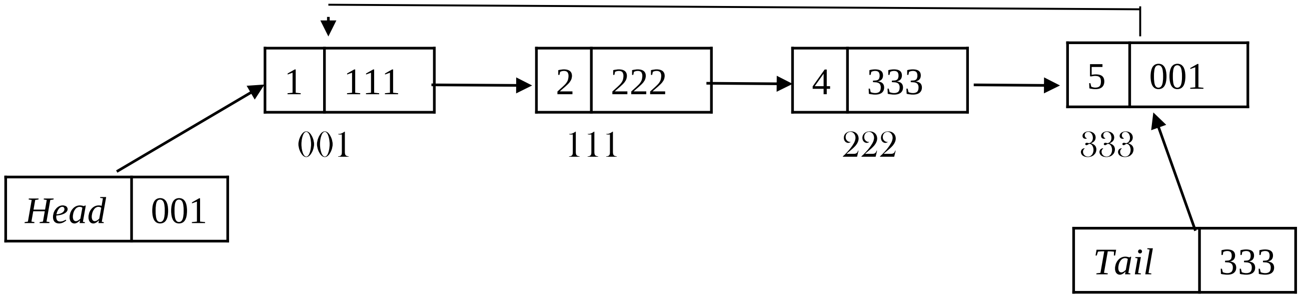
Singly Linked List



Types of Linked List

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

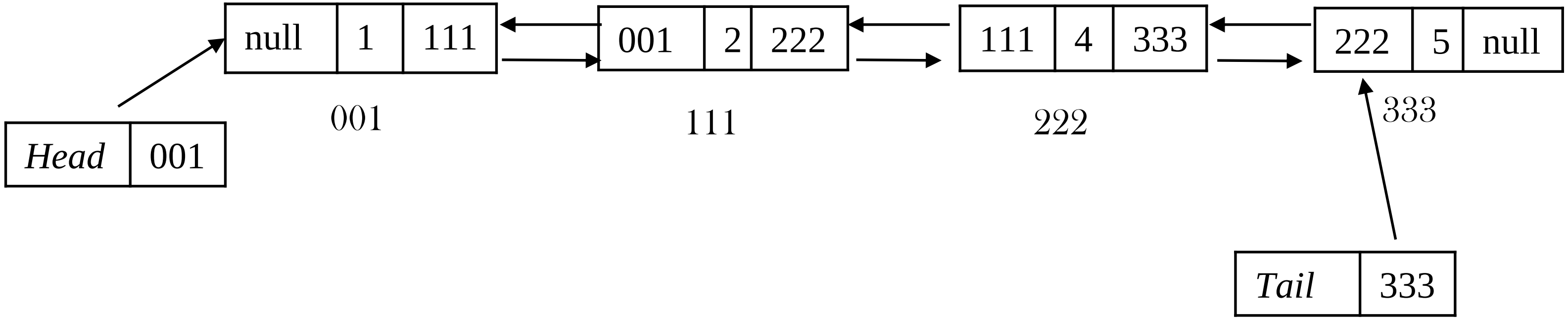
Circular Singly Linked List



Types of Linked List

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

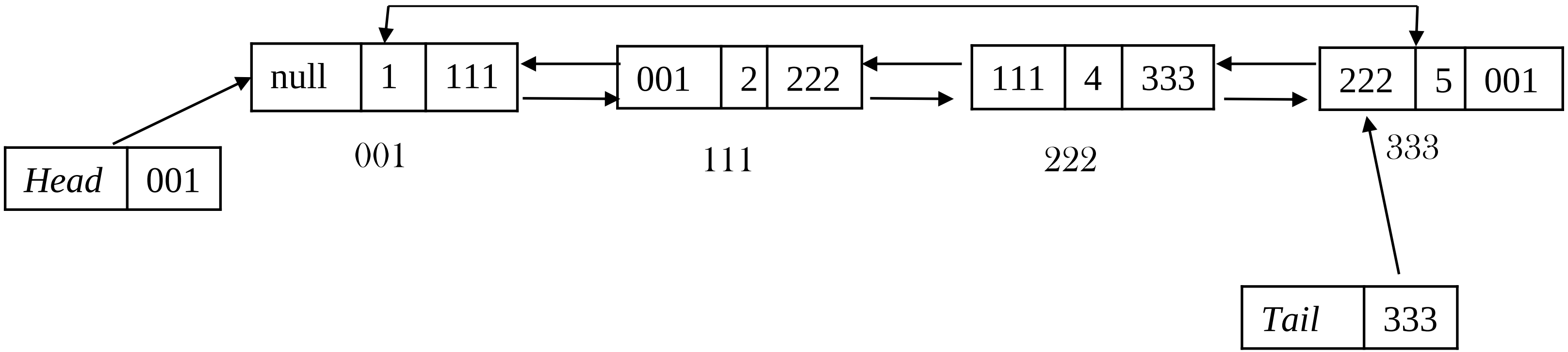
Doubly Linked List



Types of Linked List

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

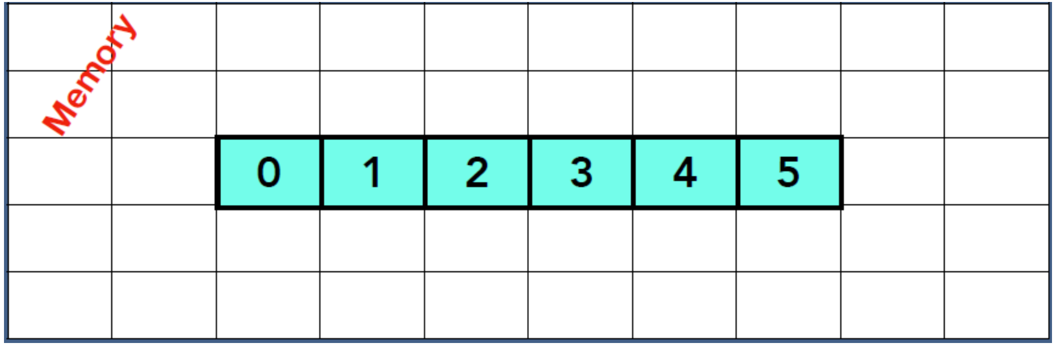
Circular Doubly Linked List



Linked List in Memory

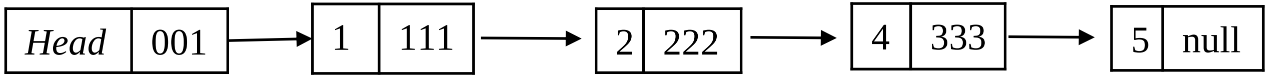
Arrays in memory :

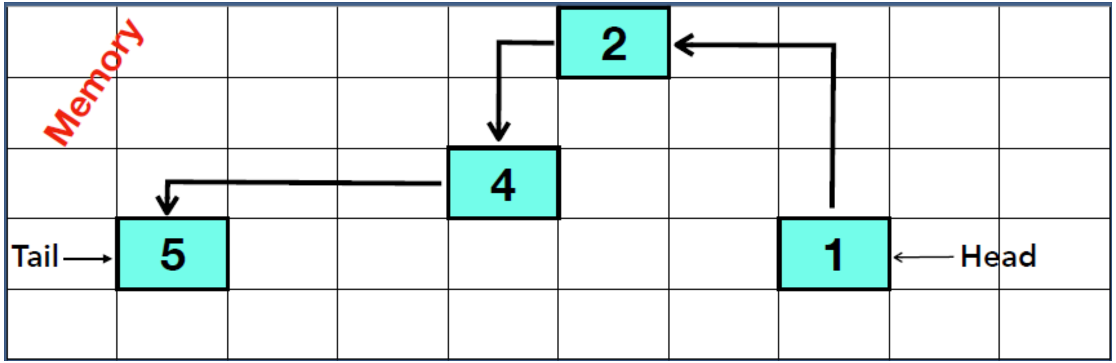
0	1	2	3	4	5



Linked List in Memory

Linked List:



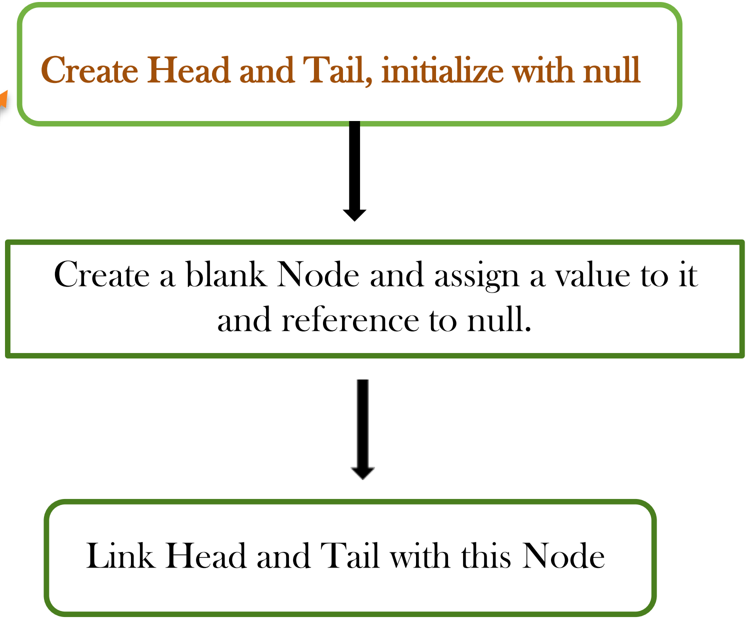


Creation of Singly Linked List

Head	null

Tail	null

```
class Node{
    int data;
    Node next;
    Node(int data){
        this.data = data;
        next = null;
    }
}
class SinglyLinkedList {
    Node head, tail;
    SinglyLinkedList(){
        head = tail = null;
    } }
```

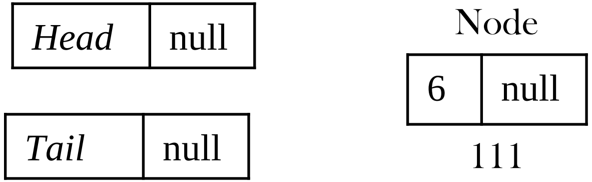


Create Head and Tail, initialize with null

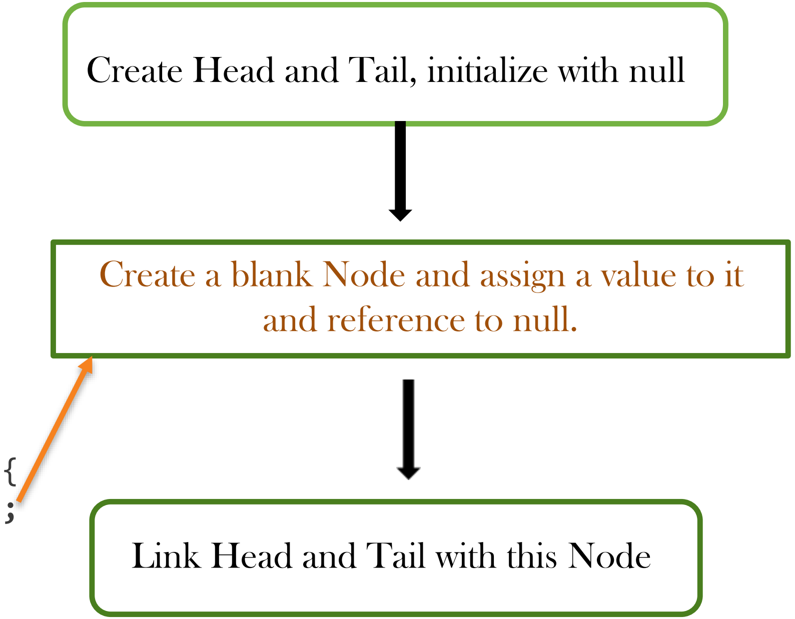
Create a blank Node and assign a value to it and reference to null.

Link Head and Tail with this Node

Creation of Singly Linked List



```
class SinglyLinkedList {
    Node head, tail;
    SinglyLinkedList(){
        head = tail = null;
    }
    void insertFirstElement(int element){
        Node newNode = new Node(element);
    }
}
```

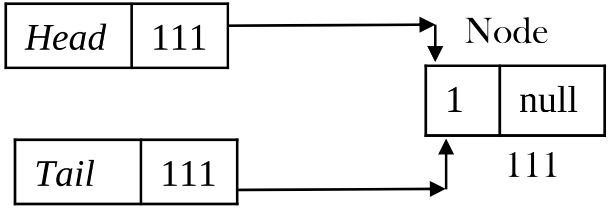


Create Head and Tail, initialize with null

Create a blank Node and assign a value to it and reference to null.

Link Head and Tail with this Node

Creation of Singly Linked List



Head 111

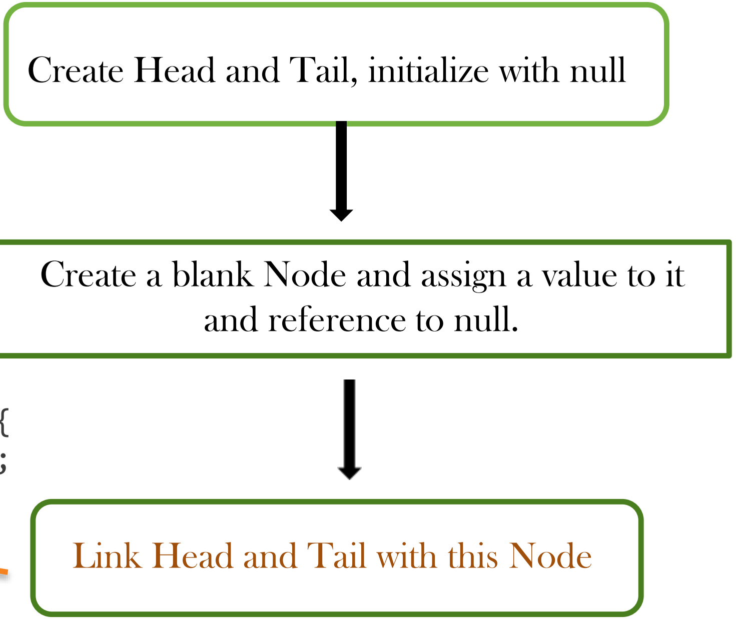
Node

1 null

111

Tail 111

```
SinglyLinkedList {
    Node head, tail;
    SinglyLinkedList(){
        head = tail = null;
    }
    void insertFirstElement(int element){
        Node newNode = new Node(element);
        head = newNode;
        tail = newNode;

    }
}
```



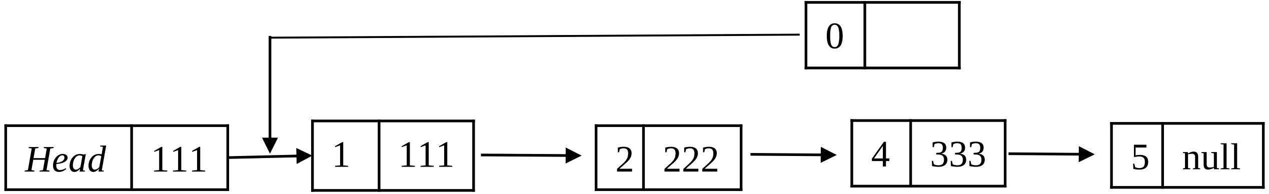
Create Head and Tail, initialize with null

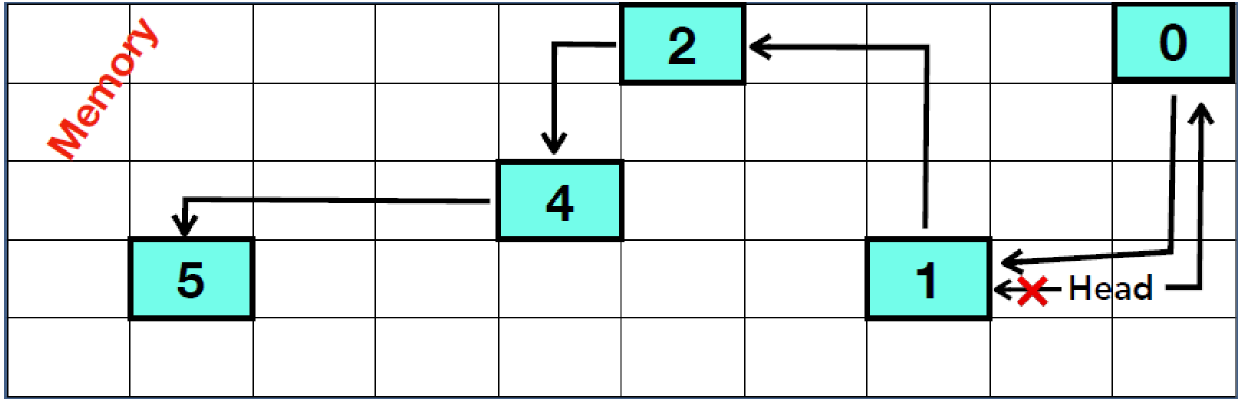
Create a blank Node and assign a value to it and reference to null.

Link Head and Tail with this Node

Insertion to Linked List in Memory

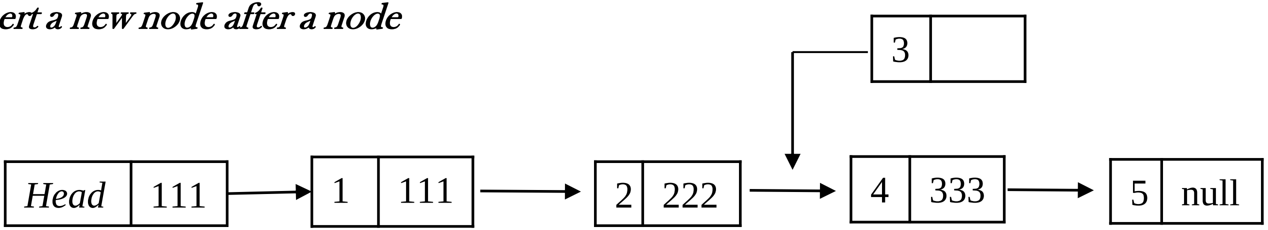
Insert a new node at the beginning

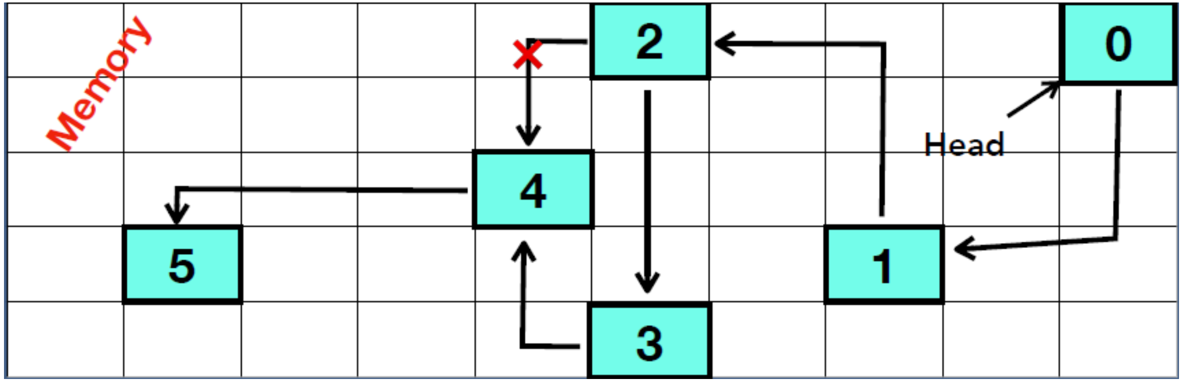




Insertion to Linked List in Memory

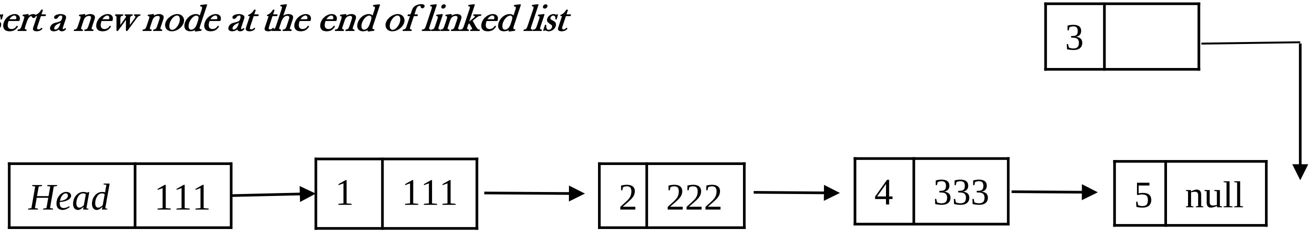
Insert a new node after a node

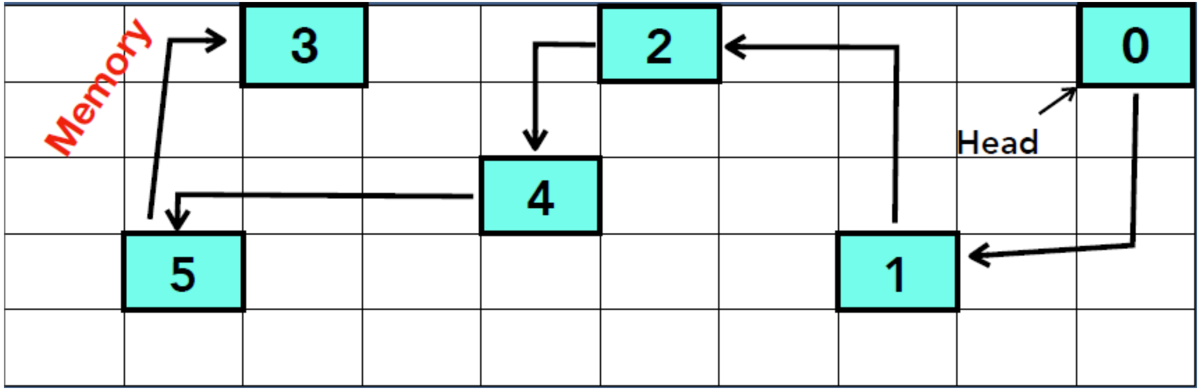




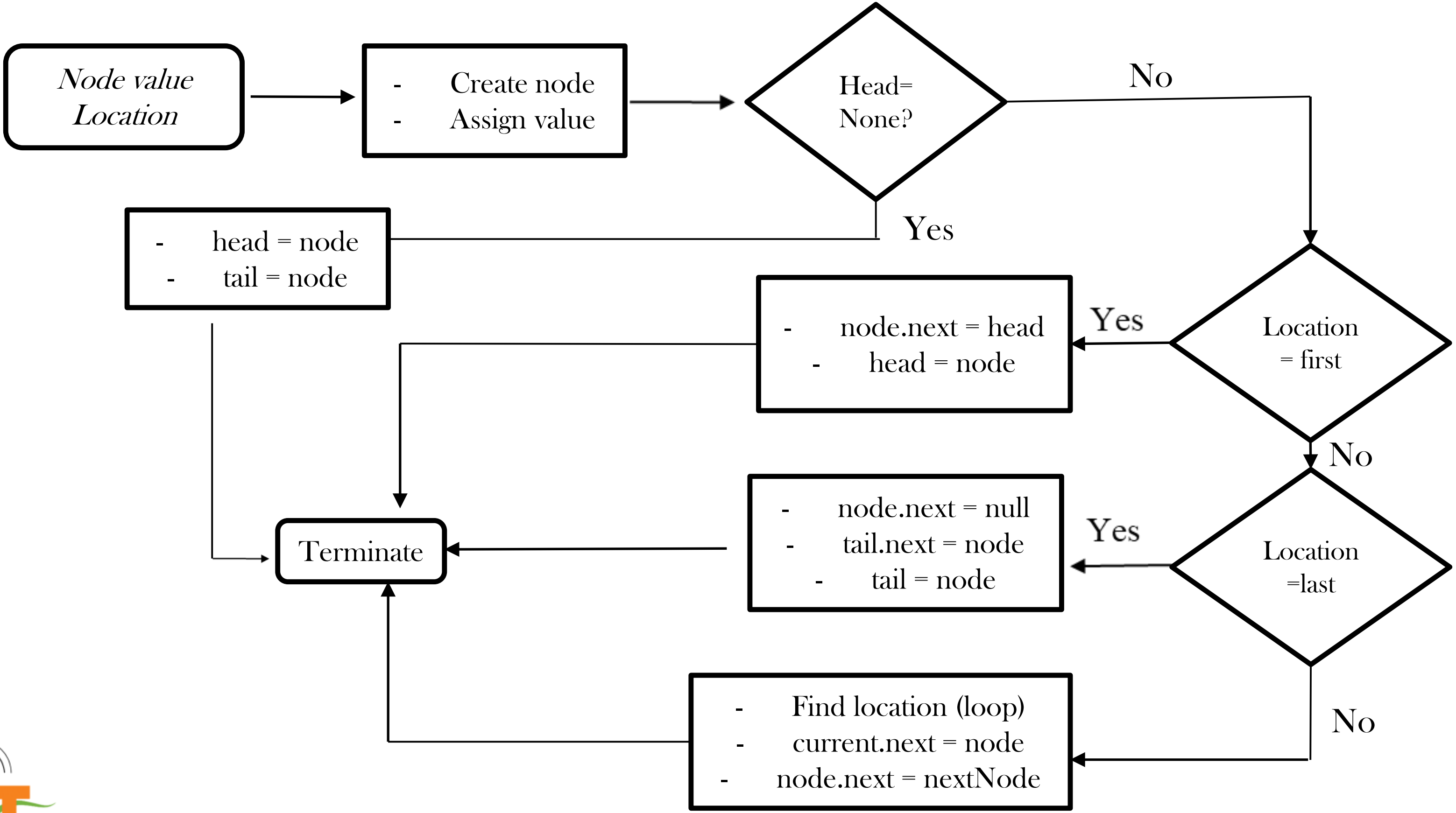
Insertion to Linked List in Memory

Insert a new node at the end of linked list





Insertion Algorithm



Node value
Location

- Create node
- Assign value

Head= None?

No

Yes

- head = node
- tail = node

Location = first

Yes

- node.next = head
- head = node

No

Location =last

Yes

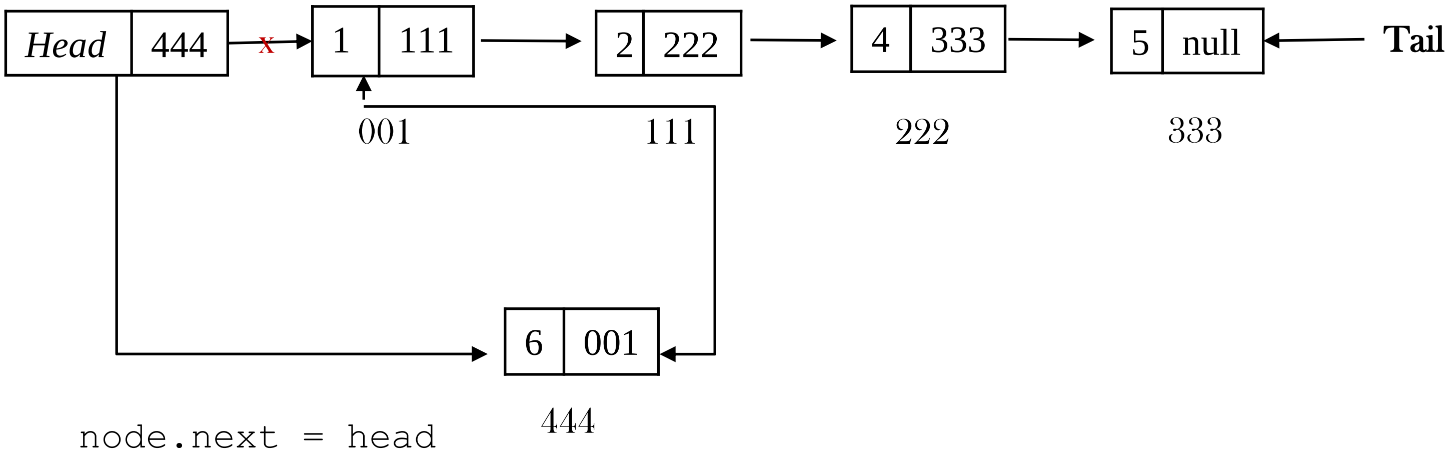
- node.next = null
- tail.next = node
- tail = node

No

- Find location (loop)
- current.next = node
- node.next = nextNode

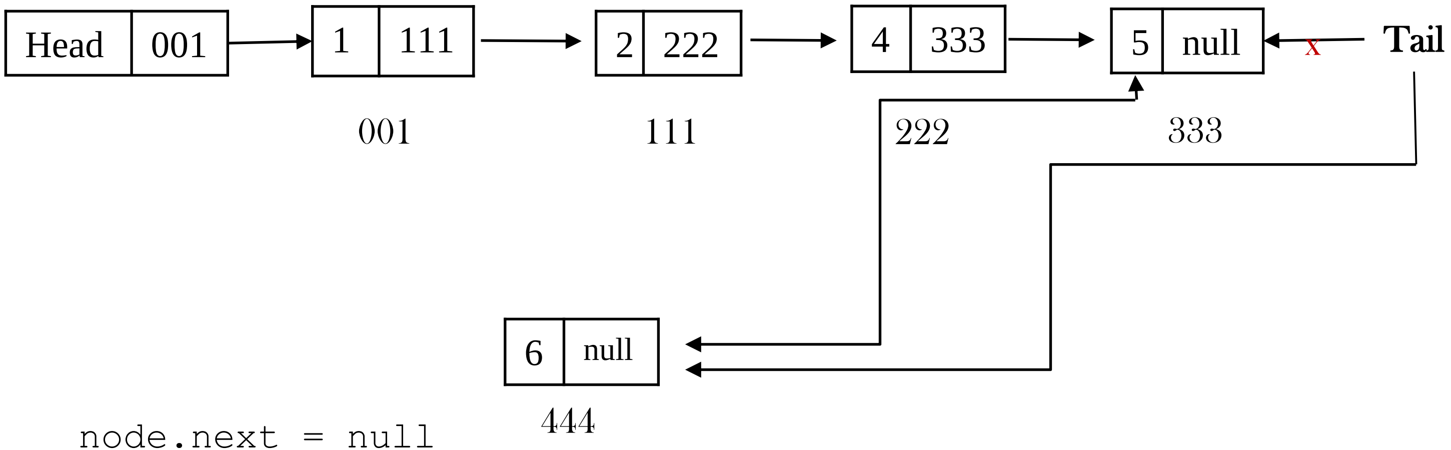
Terminate

Singly Linked List Insertion at the beginning



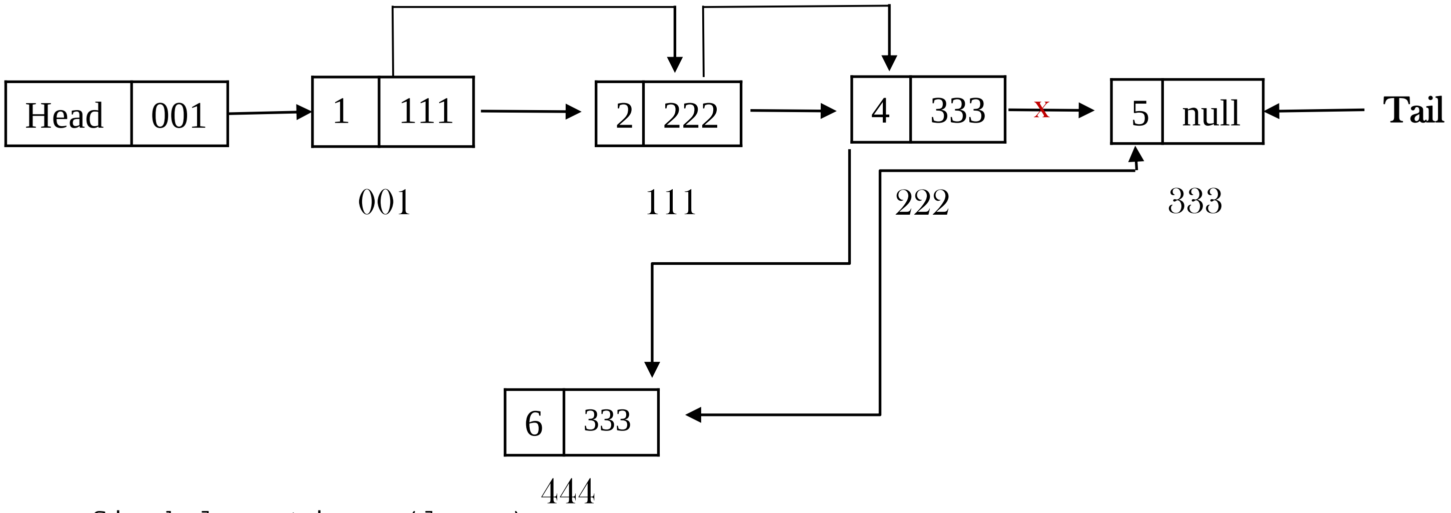
```
node.next = head

head = node
```

Singly Linked List Insertion at the end



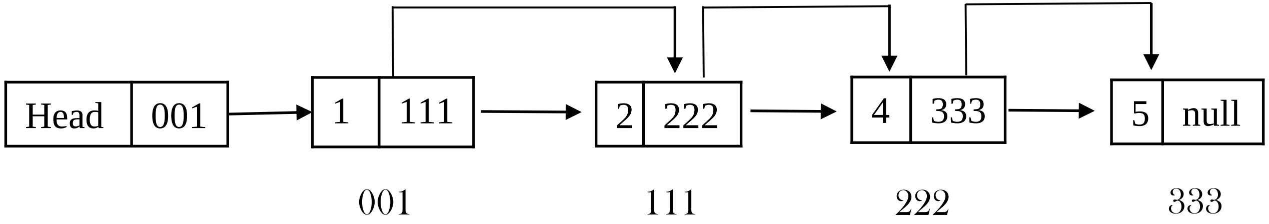
```
node.next = null
tail.next = node
tail = node
```

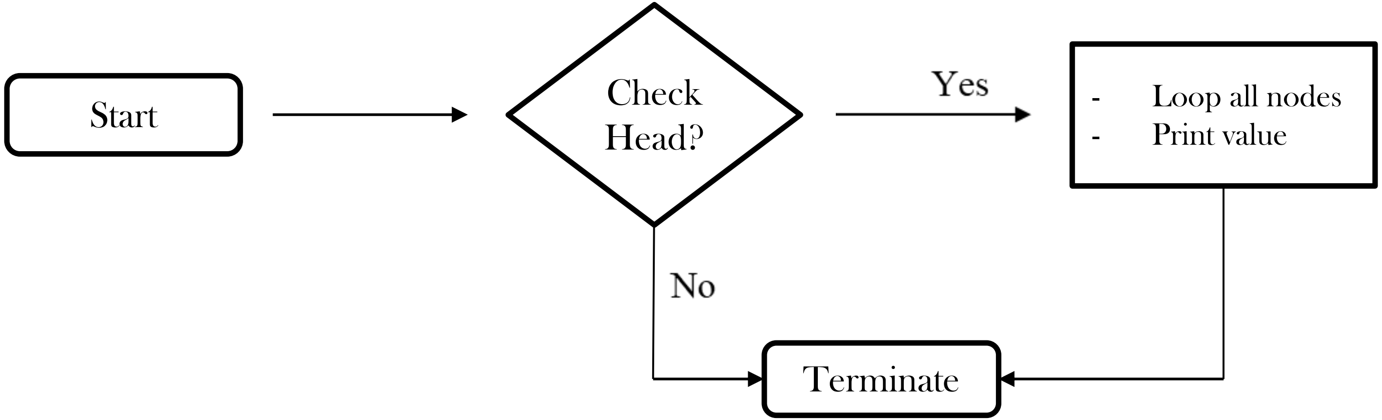
Singly Linked List Insertion in the middle



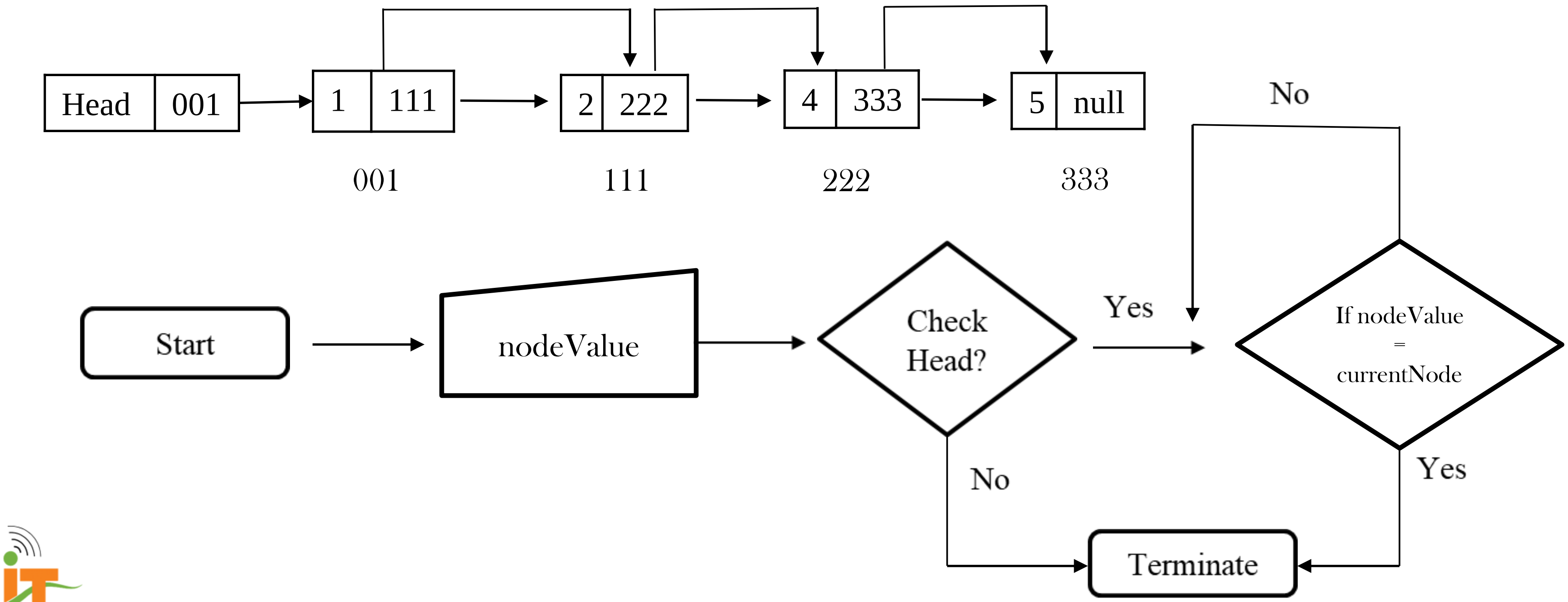
- find location (loop)
- current.next = node
- node.next =nextNode

Traversal of Singly Linked List





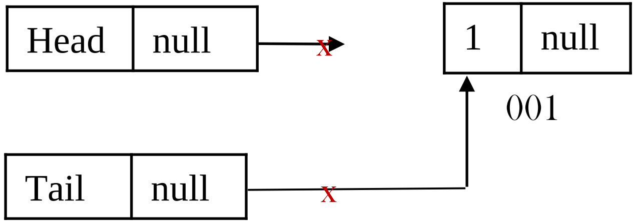
Search in Singly Linked List



Singly Linked List Deletion

- Deleting the first node ⟵
- Deleting any given node
- Deleting the last node

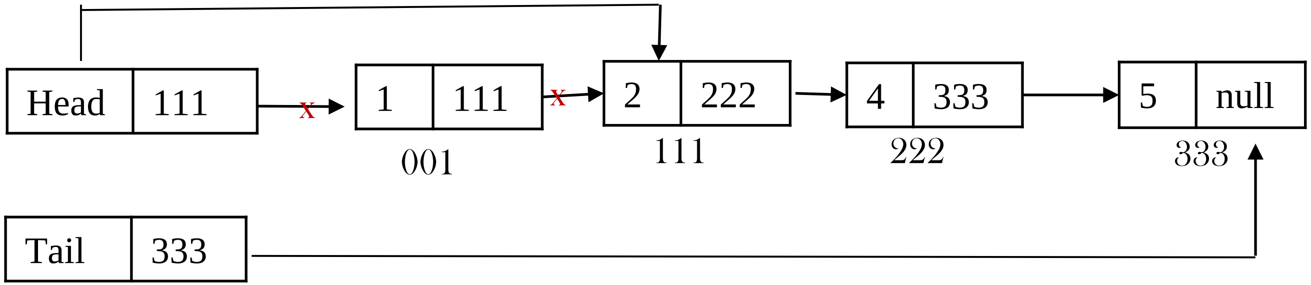
Case 1 - one node



Singly Linked List Deletion

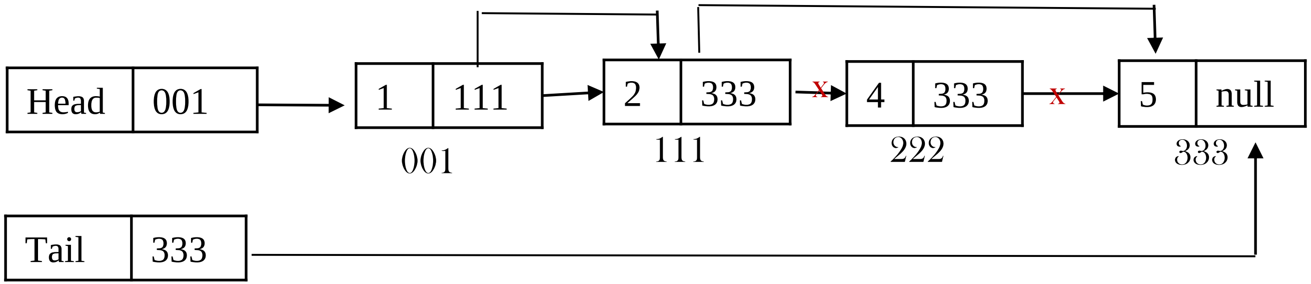
- Deleting the first node
- Deleting any given node
- Deleting the last node

Case 2 – more than one node



Singly Linked List Deletion

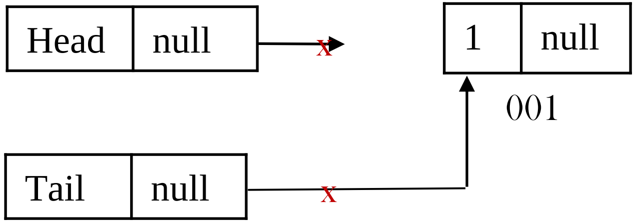
- Deleting the first node
- Deleting any given node
- Deleting the last node



Singly Linked List Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node ⟵

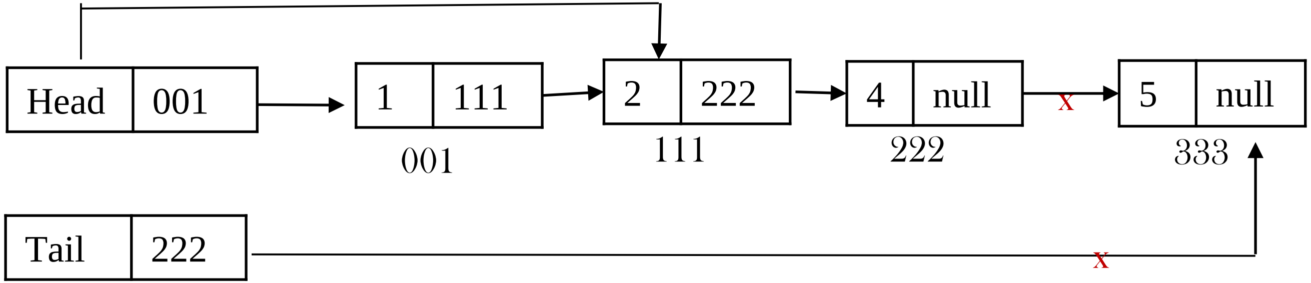
Case 1 - one node



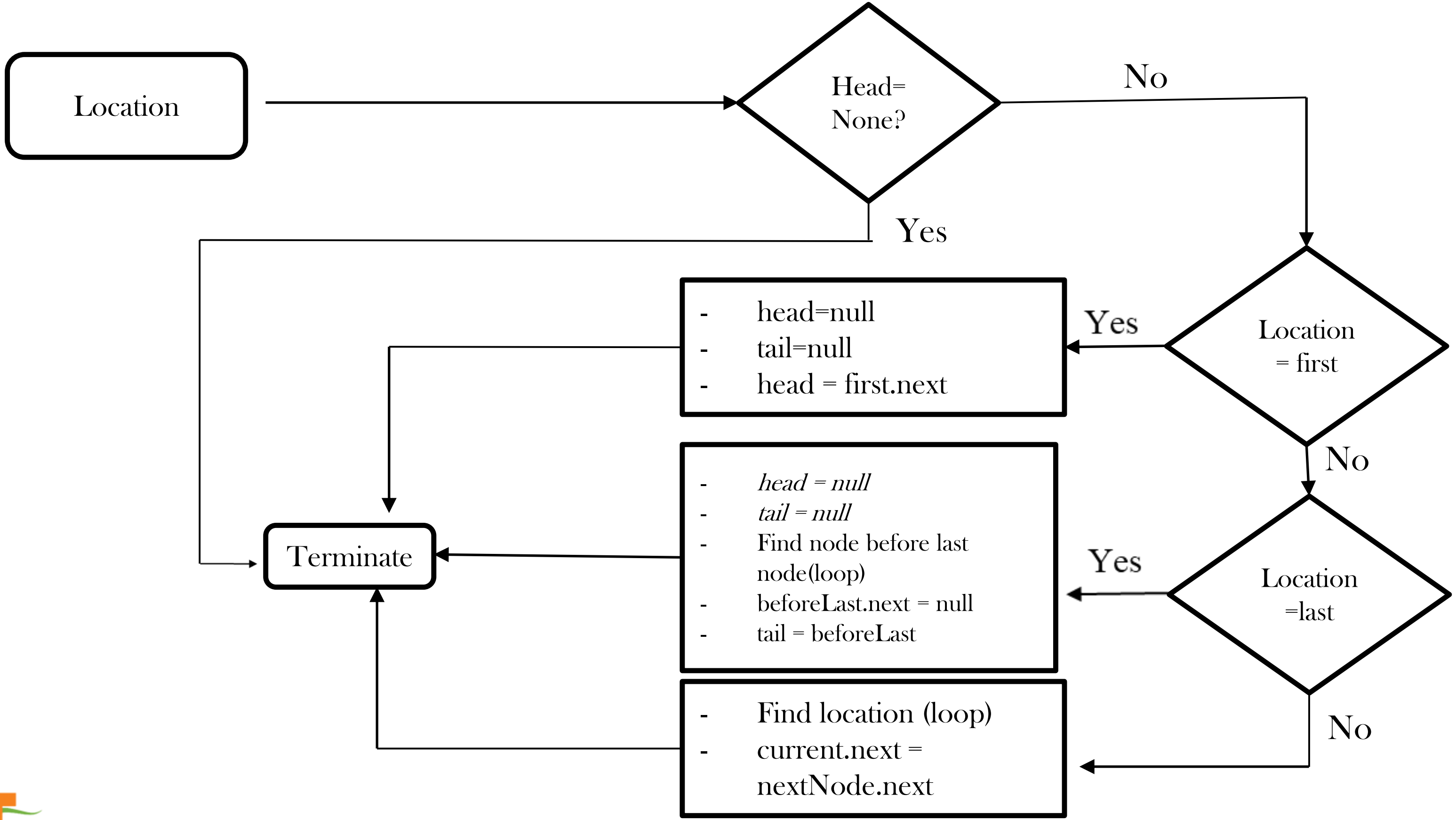
Singly Linked List Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node ⟸

Case 2 – more than one node



Singly Linked List Deletion Algorithm

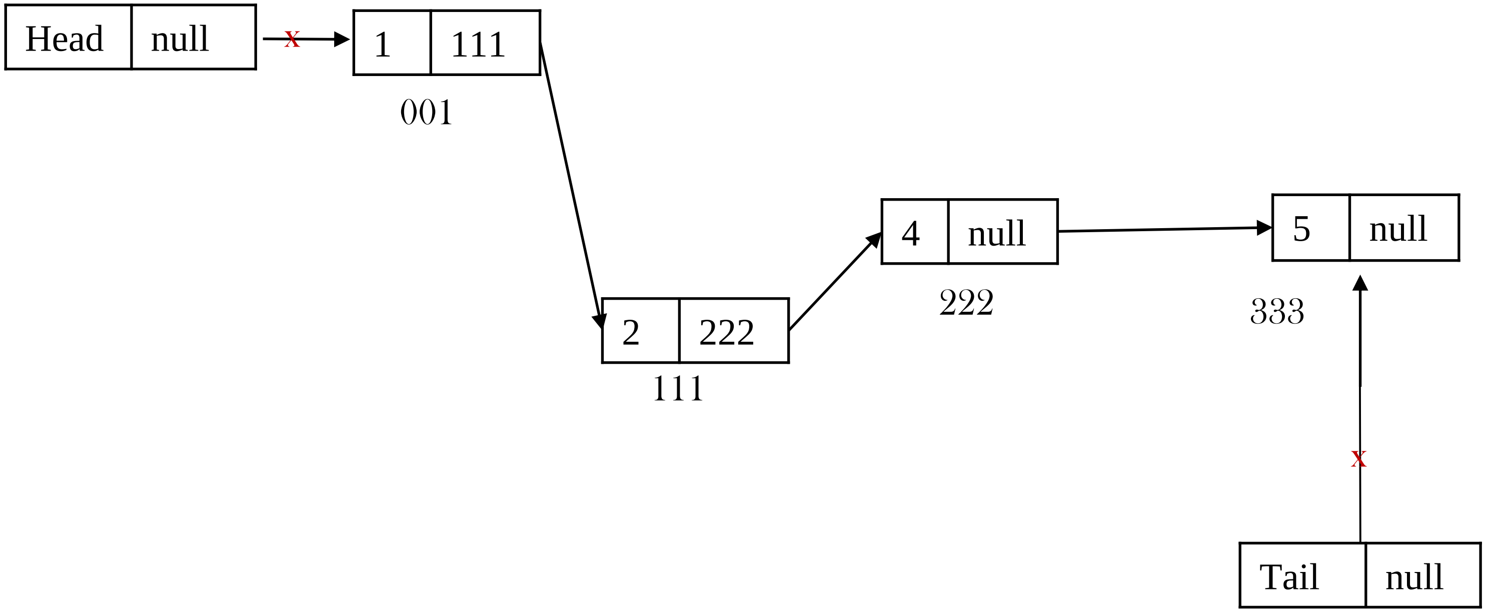


- head=null
- tail=null
- head = first.next

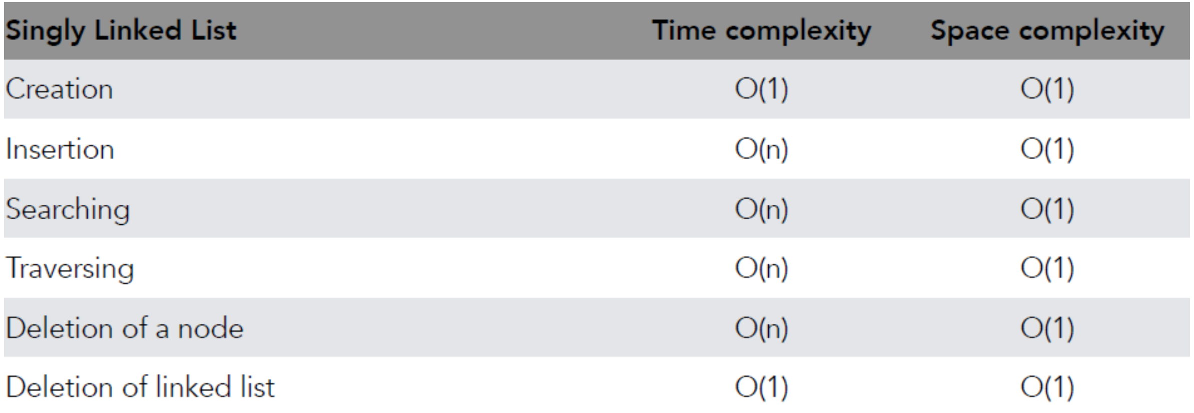
- *head = null*
- *tail = null*
- Find node before last node(loop)
- beforeLast.next = null
- tail = beforeLast

- Find location (loop)
- current.next = nextNode.next

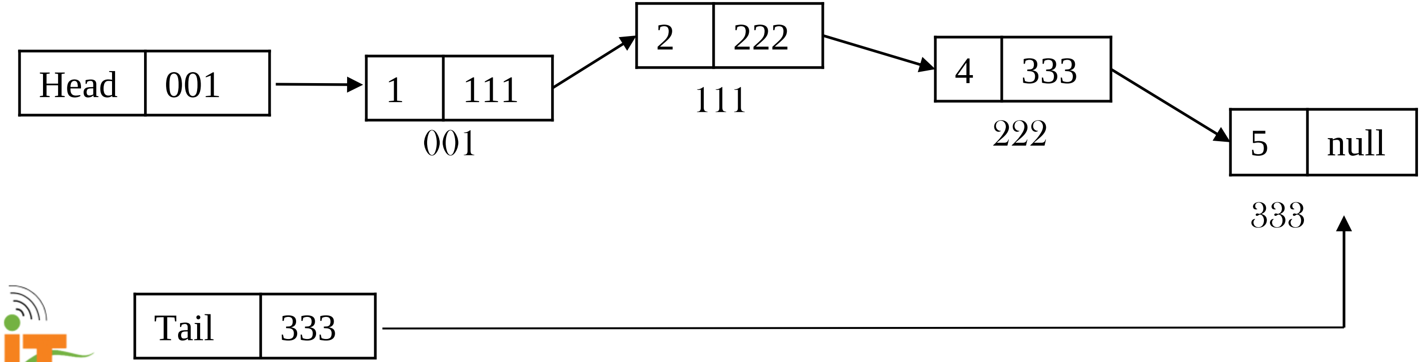
Delete entire Singly Linked List



Time and Space Complexity of Singly Linked List



Singly Linked List	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(1)	O(1)


Head
001
1
111
001
2
222
111
4
333
222
5
null
333
Tail
333