ArrayList and LinkedList

ArrayList(C)

Ways to create an Arraylist

```
//1
List<Integer> l = new ArrayList();

//2 - default capacitty is 100
List<Integer> la = new ArrayList(100);

//3
List<Integer> list = Arrays.asList(1,2,3,4,5);

// just a concept
Integer[] arr = {1,2,3,4,5};
List<Integer> list2 = Arrays.asList(arr);
```

Introduction and Basics

- An arraylist is a resizable implementation of the List interface.
- Unlike Arrays in Java , which have a fixed size, an arraylist can change its size dynamically as elements are added or removed. This flexible makes it a popular choice when the number of element in a list isn't known in advance.

```
List<Integer> list = new ArrayList();
list.add(11);
list.add(21);
list.add(31);

         System.out.println(list.get(2));
        System.out.println(list.size());
        for(int i = 0; i < list.size(); i++){
            System.out.println(list.get(i));
        }
        for(int x: list){
            System.out.println(x);
        }
        System.out.println(list.contains(5));
        System.out.println(list.contains(50));

        list.remove(2);
        for(int x: list){
            System.out.println(x);
        }
        list.add(2, 50);

        for(int x: list){
            System.out.println(x);
```

```
        }

        list.add(1); // 0
        list.add(5); // 1
        list.add(80); // 2

        list.set(2, 50);

        System.out.println(list);
```

Internal working of ArrayList

- Unlike a regular array, which has a fixed size, an ArrayList can grow and shrink as elements are added or removed.
- This dynamic resizing is achieved by creating a new array when the current array us full and copying the elements to the new array.
- Internally, the arrayList is implemented as an array of Object references. when you add elements to arrayList, you're essentially storing these elements in this internal array.
- When an arraylist is created , it has initial capacity of 10. this capacity refers to the internal array that can hold elements before needing to resize.

```
ArrayList<Integer> l = new ArrayList();
//size = 0 , no elements in it
//capactiy = 10,
```

Adding Elements to ArrayList

- ***Check Capacity*** : before adding the new element, Arraylist checks if there is enough space in the internal array. If Array is full, it needs to be resized.
- ***Resize if necessary*** : If the internal array is full, the arrayList will create a new array with a larger capacity(usually 1.5 times the current capacity) and copy the elements from the old array to the new array.
- ***Add the Element*** : The new element is then added to the internal array at the appropriate index, and the table size is increased.

Resizing the Array in ArrayList

- ***Initial capacity :*** By default, **initial capacity is 10.** This means internal array can hole up to 10 elements before it need to grow
- ***Growth Factor :*** When the internal array is full, ***a new array is created with a size 1.5 times the old array***. This growth factor balances memory efficiency and resizing cost.
- ***Copying element :*** When resizing occurs, all the elements from the old array are copied to the new array.

Removing Elements

- ***Check bounds :*** The arraylist first checks if the index is within the valid range;
- ***Remove the Element :*** The element is removed , and all the elements to the right of the removed element are shifted one position to the left to fill the gap.
- ***Reduce size :*** The size is decremented by 1.

💡 Initial capacity of the arraylist can be set more than 10. Consider we are aware that our arraylist will store 100 elements. then we can create an arraylist with initial capacity of 100. which will not resize.

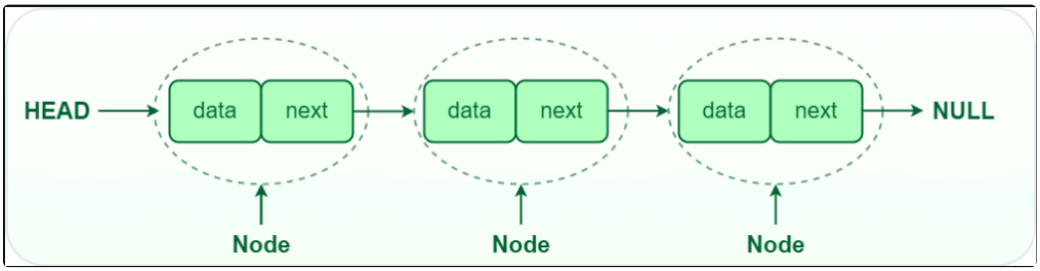
```
List<Integer> l = new ArrayList(100);
//100 is inital capacity now, so till 100th element arraylist will not resize
```

LinkedList (C)

- A linked list is a linear data structure where each element is separate object called as node. Each node contains two parts.
  - Data : Value stored in node
  - Pointers(next) : pointing to next node or one pointing to next and another to previous
- A linked list stores an element in form of nodes

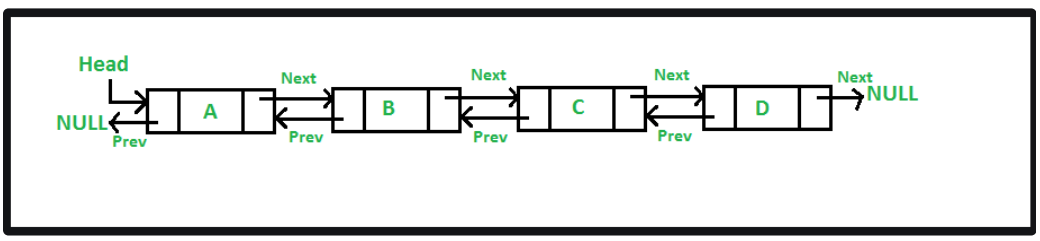
Singly Linked list

- For singly linked list → node contains data and address of next node

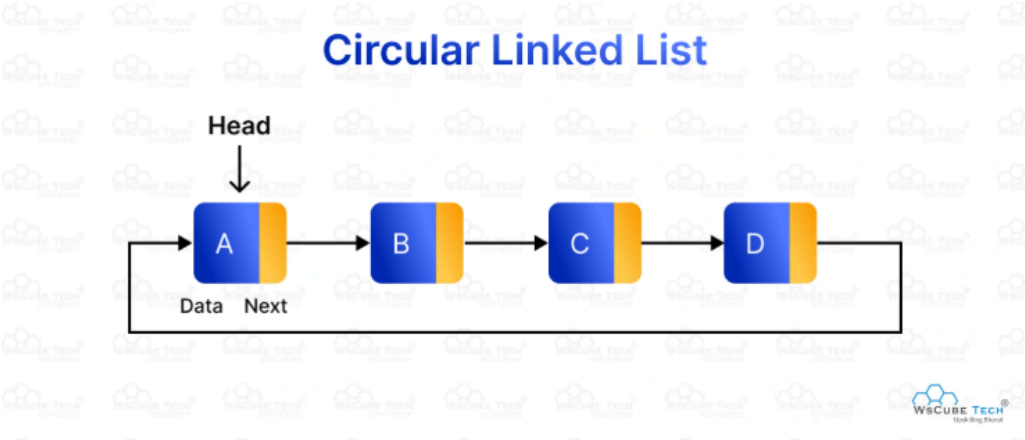


Doubly Linked list

- For singly linked list → node contains data ,address of next node and address of previous node



Circular linked list



- *In linked List, If you want to find value of element at any particular index then complete list need to be iterated as it is not a indexed base collection.*
- *When we want to add data in linked list we can just change pointers address to the new nodes and no shifting of elements is required*

Performance Consideration *:*

- ***Insertion and deletion :*** *Linked list is better for frequent insertion and deletion in the middle of the list because it does not require shifting elements, as int Arraylist.*
- ***Random access*** : LinkedList has slower random access (get(int index)) compared to arrayList because it has to traverse the list from the beginning to reach the desired index.
- ***Memory Overhead*** : *LinkedList require more memory than array list because each node in LinkedList requires extra memory to store references to the next and previous nodes.*

ArrayList vs LinkedList

ArrayList	LinkedList
Yes, **Allows multiple null value**	Yes, **Allows multiple null value**
Yes, **Allows duplicate**	Yes**, Allows duplicate**
Yes, **insertion order is preserved**	Yes, **insertion order is preserved**
Searching Operation is **fast** compare to LinkedList	**Searching** Operation is **slow** compare to ArrayList
Get() performance is **fast** compare to LinkedList because of ArrayList maintain Index base system.	**Get()** performance is **Slow** compare to ArrayList because of LinkedList implements **Doubly** LinkedList.

Deleting is **Slow** compare to LinkedList.	**Deleting** is **fast** compare to ArrayList.
ArrayList need to shifted to fill the space created by remove operation.	LinkedList pointer location in two neighbor node of the node which is going to be remove.
Add() is high time complexity for insertion.	Add() is low time complexity for insertion.
Memory consumption is **low** in ArrayList because of indexing.	Memory consumption is **High** in LinkedList because of maintain data and two pointers for neighbor node.
Implements **dynamic** array internally to store elements	Implements **doubly** linked list internally to store elements
Manipulation of elements is **slower**	Manipulation of elements is **faster**
ArrayList act only as a List	LinkedList act as a List and a Queue
Not thread safe	Not thread safe

Summary :

For random access →	**ArrayList** ✅
For frequent insert/remove →	**LinkedList** ✅

Document by ***Suyash***😇