BIG O notation

Arrays

Arrays are the simplest data structure and we use them to store list of items like a list of strings, list of integers, list of objects etc. These items are stored in sequence in memory.

Example:

Memory address: 100 104 108 112 ……………….so on

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array: | 27 | 7 | 44 | 12 | 54 |

Here we have an array with list of integers like 27, 7, 44, 12, 54 and their corresponding memory address for example starts with 100 and in java each integer value takes 4 bytes of memory so the next address will be 104, 108 and so on. Now because of this, the run time complexity of array for searching any item is **O(1)**. Here the calculation of memory address doesn’t take much time since no loops are involved or complex logic. So if we want to store items and access them by their index then arrays are the optimum ones.

In java and others Arrays are static, which means we have to specify the size of array while defining them. If we don’t the size upfront, then we have to guess. If we guess either low or more size then we will have either shortage of memory or more unused memory.

So for example, if we have an array with less size defined but we have more items to add and if we create a new array with bigger size and copy the previous array items to this new array, then the run time complexity will be linear which is O(n) (as the input grows so does the number insertions or copies to the new array linearly).

Now coming to the delete of an item from array, For the best case scenario if we have to delete an item from the last index then it is easy for array to check the index and delete. So best case scenario has run time complexity of O(1). But there is worst case scenario where we may have to delete an item from the first index, then the item has to be removed and all the rest of items indices has to be readjusted. So the run time complexity for the worst case scenario of deletion of item in array is O(n).

Sample code for deletion of an item from index 3 which is 40. Here we loop from the index=3 and shift the next to index item and same for the rest of items. So here if the index that we want to delete is in the beginning then we have to shift all the elements.

*int[] array = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};*

*int index=3;*

*for (int i = index; i < array.length - 1; i++) {*

*array[i] = array[i + 1];*

*}*

*Output: 10, 20, 30, 50, 60, 70, 80, 90, 100, 100*

If we observe here item value is removed but since array size is static the last value is still there and we have now duplicates. So the only way we have is to copy these items to a new array using a loop.

Alternatively we can use Arrays.copyOf(original array, new length) to copy to a new array.

*int[] array = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};*

*int [] array2 = Arrays.copyOf(array,9) // this is to remove last item*

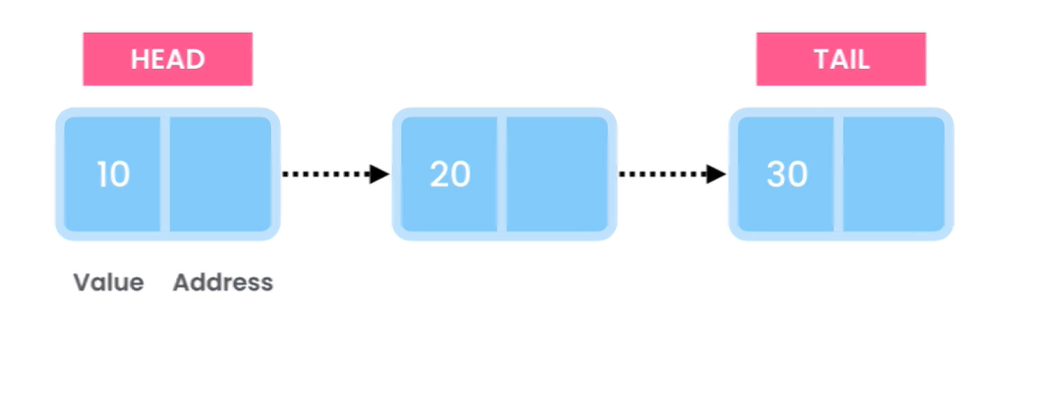
*int [] array3 = Arrays.copyOf(array, 1); // to remove from start*

*System.out.println(Arrays.toString(array));*

So for lookup we use Arrays and for insert and delete we have to use Linked List.

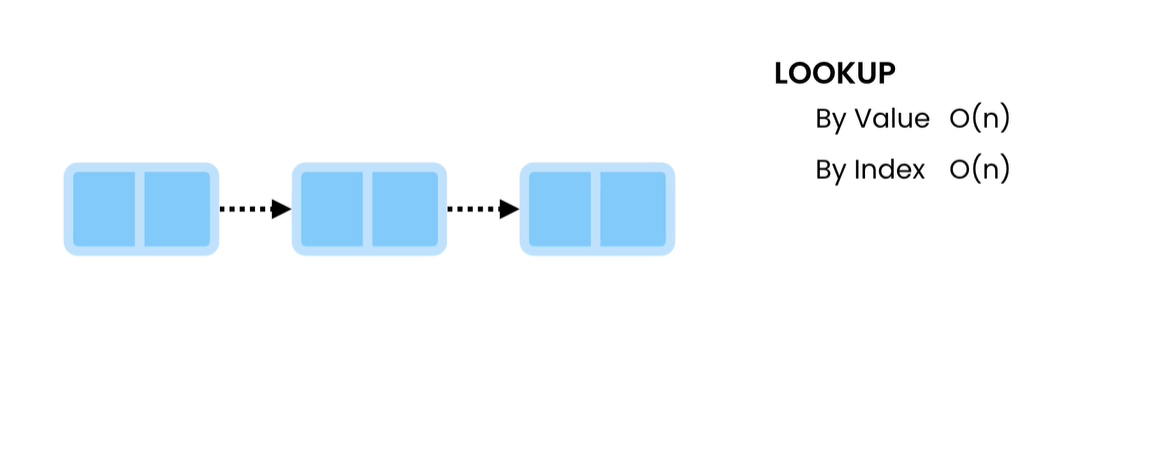
LinkedList

We use Linked Lists to store items or objects in sequence but unlike Arrays, LinkedList are capable of growing and shrinking when we try to add new item or remove an item. LinkedList consists of a group of nodes in sequence. Each node consists of value and address to the next node. The first node is called head and last one is tail.

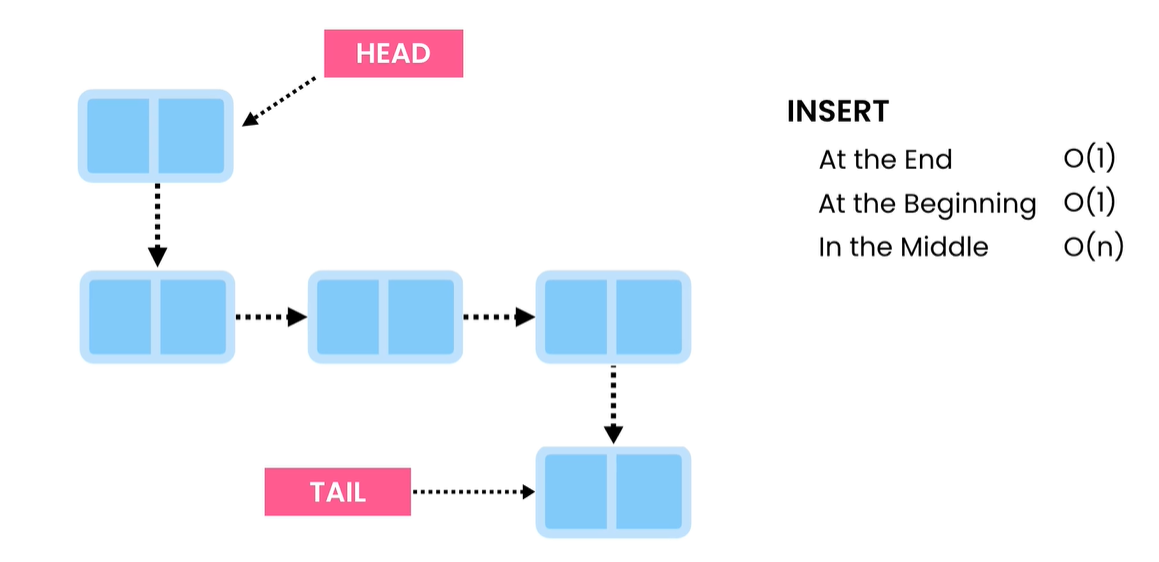


Time complexity

Now lets say we want to lookup a given number in LinkedList, here in worst case scenario we have to traverse all the nodes hence the time complexity is O(n). If we are looking for a value by index, time complexity will be same O(n). Unlike arrays, nodes in linked list are stored in different memory address, that’s why they keep the address of next node. So if we look up either by value or index the time complexity is O(n).



Time complexity for Insertions depends on where we are going to insert an item. If we are inserting at the end of linkedlist, then a new node is created and the tail is pointed to the new node and last node will be pointed to the new node. Similarly if we are inserting in the beginning a new node will be created and head will be pointed to the new node and the new node points to the first node in the sequence. So for both these cases the time complexity will be O(1). Now if we have to insert in the middle then first we have to find out the address when the new item has to be inserted which is O(n) and then point the node addresses properly which is O(1). Since we have to lookup for the insert address the time complexity will be O(n) in the worst case.

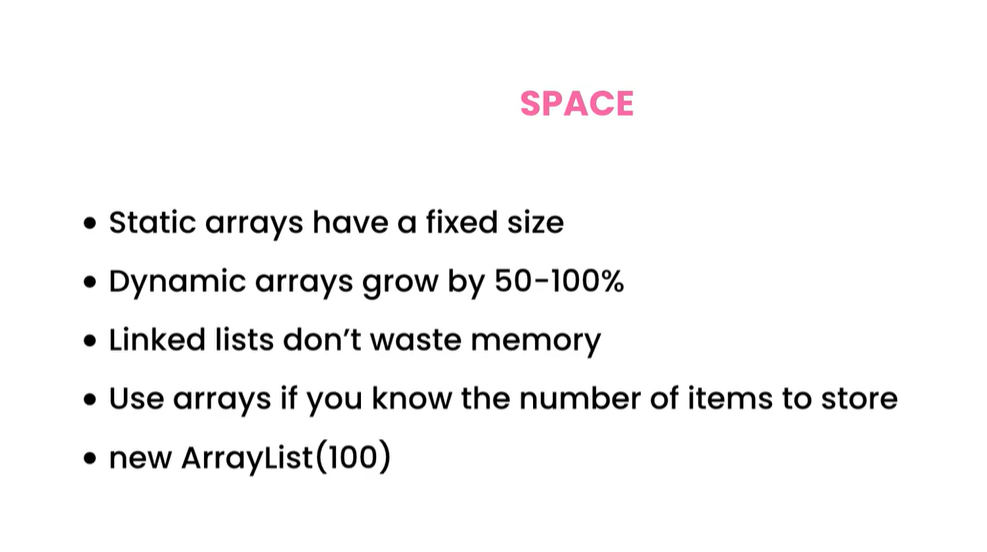


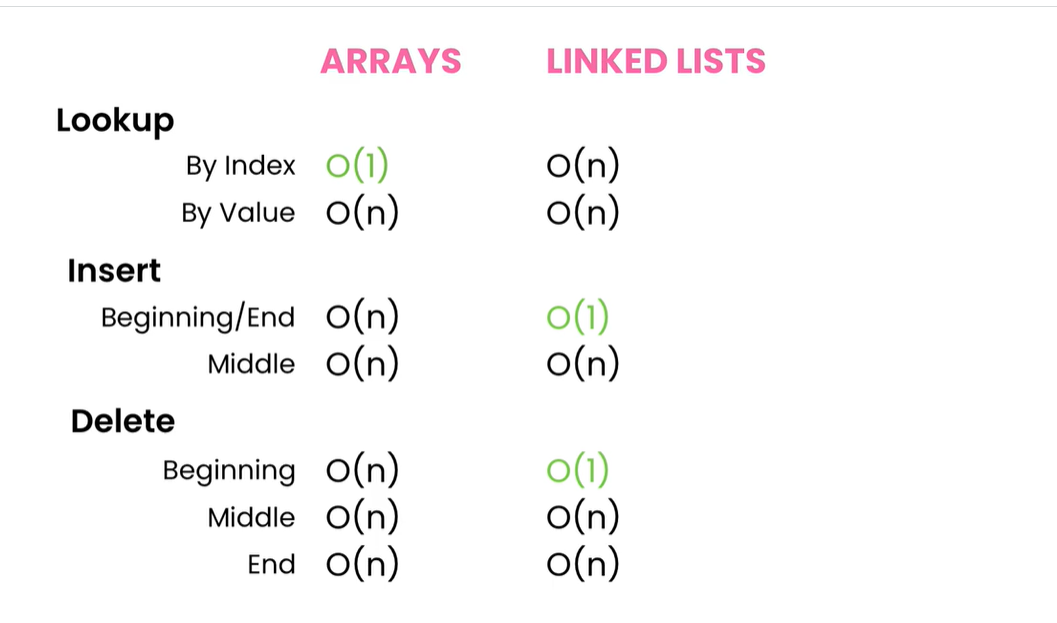
Timecomplexity for the deletion operation on linkedlist.

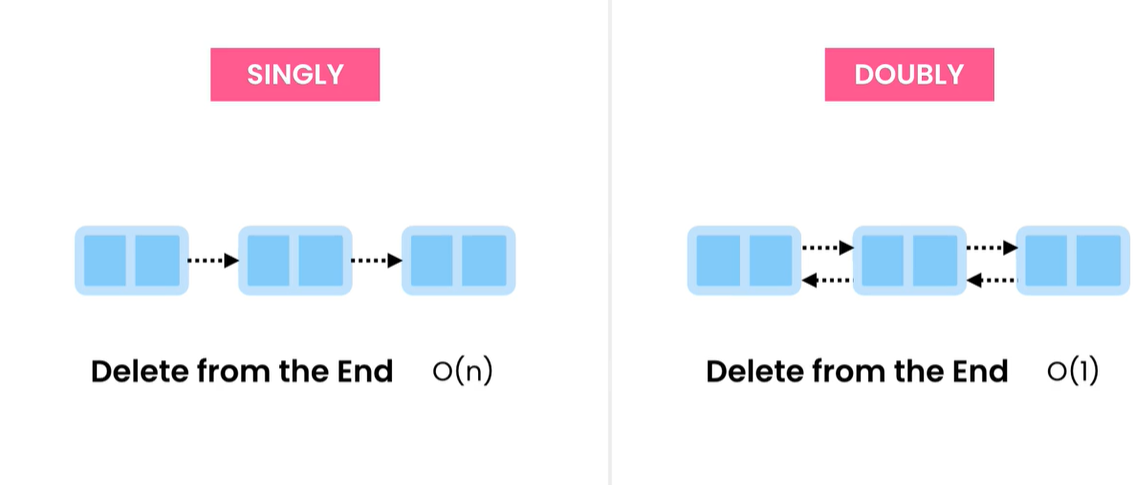
Delete from the beginning: We have to simply point head to the second node. Since the previous first node doesn’t have head reference, java’s garbage collection will remove it from the memory. Hence the time complexity is O(1).

From the end: we have to traverse all the node until the last node and get the address of last but one node and point tail to that node and the java’s garbage collection will remove the previous last node. Hence here the time complexity is O(n)

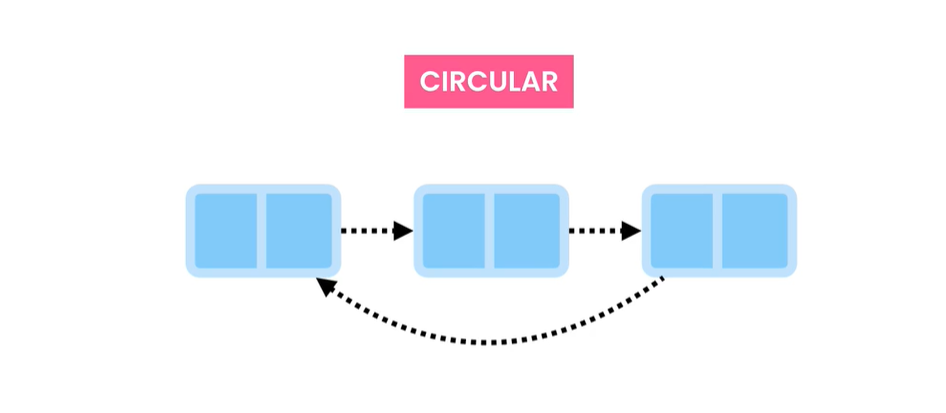
From the middle: We have to first lookup for the node and previous node which has time complexity of O(n) and then repoint the node addresses.







Only disadvantage with doubly linked list is it take more space but good for delete from the end in regards to performance. LinkedList in java is doubly linked list. Both Singly and Doubly linked lists can be circular.



**Stacks**

Stacks are complex data structures that can help us solving many complex problems.

* Implement undo feature
* Build Compilers (Eg. Syntax check)
* Evaluate arithmetic expression
* Build Navigation (Forward/ backward button on browser)

Stacks in real example are a Stack of books and it is Last in First Out (LIFO). If we want to take last book then we need to take out all the other books. Practical example is undo feature in text editors. When we do some typing in text editor and the click on undo (ctrl + Z) the the last change gets undone first and the sequence LIFO continuous. Internally Stacks use linkedlist or array to store. Stack is kind of a wrap on top of them.

Operations:

All the operations are run in O(1)

push(item): to add an item to the top

pop() : to remove an item from the top

peek(): returns the item on the top without removing from the stack

isEmpty(): Checks if stack is empty or not.

There are no lookups in Stack. Stacks are not really meant for storing list of object and looking up form them later.

Reverse a string:

We can easily solve problems involving going back or doing something in reverse order, stacks are the friend for us. That’s whey back button in browser and undo features are built using stacks.

**public** String stringReverser(String input) {

**if**(input==**null**) **throw** **new** IllegalArgumentException();

Stack<Character> stack = **new** Stack<>();

**for**(**char** ch: input.toCharArray()) {

stack.push(ch);

}

StringBuilder reversed=**new** StringBuilder();

**while**(!stack.empty()) {

reversed= reversed.append(stack.pop());

}

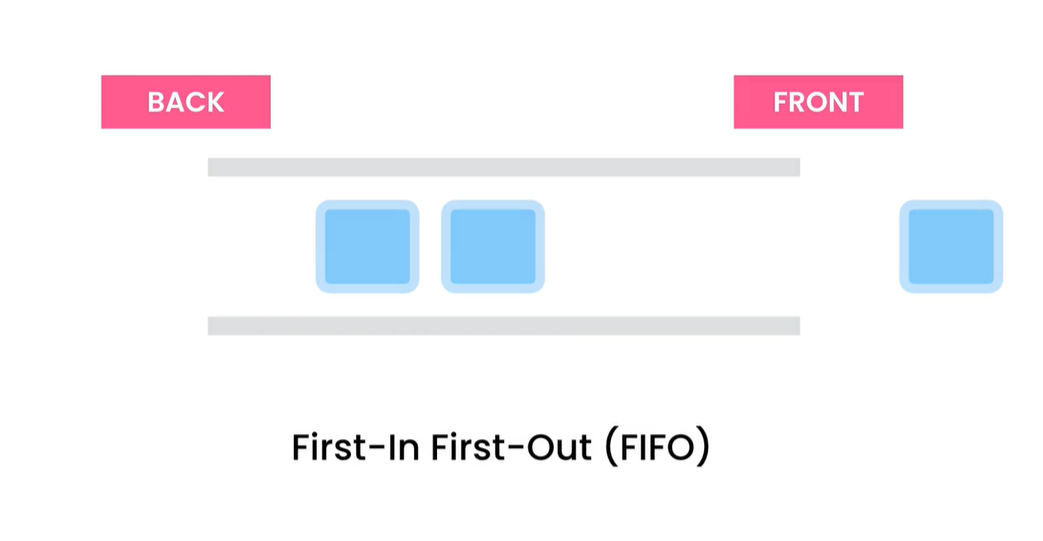
**return** reversed.toString();

}

Queues

What are queues?

Queue is a data structure similar to Stack except that in queue we have FIFO (First in first out). The first that enters will the item that gets out first.



Real world examples:

* Printers use queues to manage jobs. They jobs in the order of their submission.
* Operating systems use queues for processes.
* Webservers for incoming requests
* Building live support systems. So the resource will service consumer as per who came first.

