Data Structures - Arrays, LinkedLists, and Dynamic Arrays

Before we understand what, each data structure is, we should understand how they are used and what operations they support. (what are we really trying to solve here)

1. What is the difference between an Interface and a Data Structure

Interface	Data Structure
What you want to do	How do you do it
Specifications	Representation
What data you can store	How you can store it
What the operations do/support and mean	Algorithms (how to support those operations)

You should be able to differentiate/separate what you want to do, and how you want to do it, before selecting which tool (data structure) to you want to solve it with.

2. Data can be stored in either a ***fixed size data structure***, or a ***dynamic size (not fixed).***

Static Sequence Interface (Arrays In Java)

These operations hold the fundamentals of all data structures

1. **Items**: $x_0, x_1, x_2 \ldots x_n$
   There is no change in the number of items (the size is fixed)

2. **Build (x)** : make a new D.S. for the items.
   Java Ex : int array = new int[n]

   Where, a data structure 'array' has been initialized to support the items, with a fixed size of 'n'

   Ex:

3	*6*	*1*	*13*	*7*

3. **len()** : return n
   Ans: 5

4. **iter_seq():** output x_0, x_1, x_2 … x_{n-1} in a sequential order

   3 6 1 13 7

5. **get_at(i):** return x_i (index i)

   ex: array[3] = 13

6. **set_at(i,x):** set x at i

   ex: array[1] = 2

3	2	1	13	7

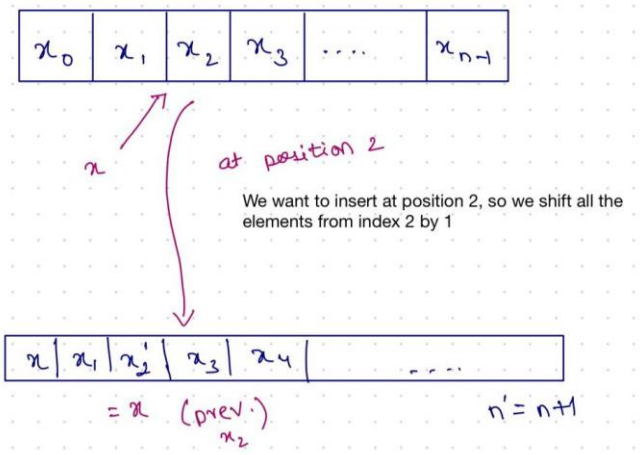
Dynamic Sequences

Here, we have two additional operations.

1. **insert_at(i,x):** insert in the middle of the sequence

Make x the new x_i , we insert x into the ith index.

So that means, ***Shifting*** x_i -> x_{i+1} -> x_{i+2}….. -> x_{n-1} -> x_n



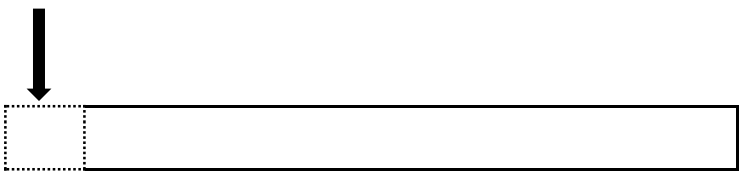
2. **delete_at(i,x):** delete in the middle of the sequence

Now x_i -> x_{i-1} after the index x. (We shift backwards)

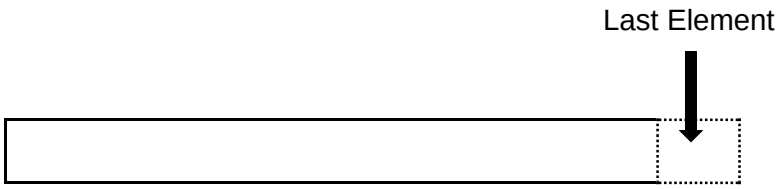
get(i) -> now calls at the new indexing

Now lets look at a case where we insert/delete - first and last(x)

First Element



Changes the indies by incrementing +1

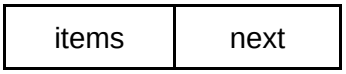


Doesn't change the indices of any.

Therefore, the get and set of the first/last indices are special cases.
Sometimes by analyzing the special case for these, it may be more efficient than the plain insert.

Linked Lists

- This data structure stores items in a bunch of nodes.
- Each node has an item in it, and a next field, which points to the next item in the list.



2 class variables

- These are assembled into a structure where
  Item -> value
  Next -> provides us an order
  & we keep track of head, length

- Linked Lists are linear data structures, which can be traversed one by one.

- Linked Lists have the advantages over stacks and queues where elements can be

Some possible implementations are,

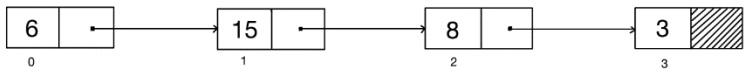
1) The array Interface (index-based operations)
2) Using an internal cursor and removed in any order, thus more versatile.
3) Iterator (OOP Design Pattern)

Native Array

6	15	8	3

In the form of Linked List:

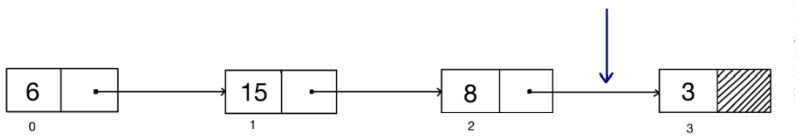
Front



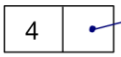
Operations in a Linked List

1. Insert an element 'item' at index i

Ex:

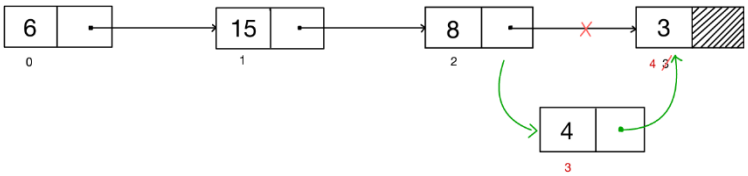


Adding an item 4 in index 3

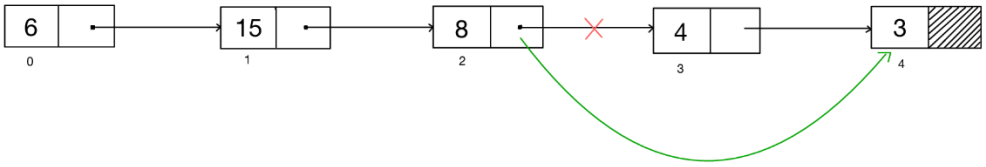
1) Create node 

2) Find the node at index 3

3) Add the node by:
    a) Change the pointer of its (index-1) to point to our new node
    b) Point the new pointer of our node to the element at index (prev index, now index+1, after updating the pointer)

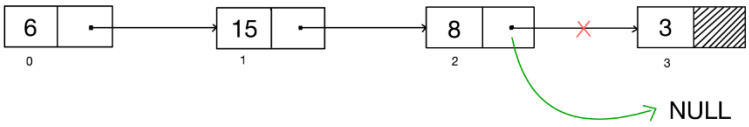


4) For delete, the last step is:
    a) Remove the pointer from index-1 and point it to index+1

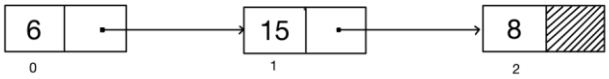


2. Insert/Delete at the end of the list

a)



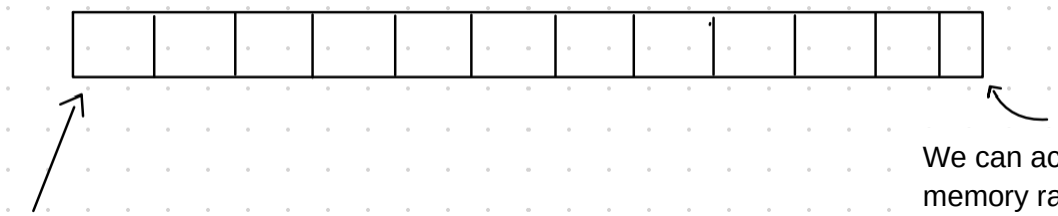
b)



Behind the scenes:

- In the word – RAM model of computation,
  The memory = array of w-bit words

- RAM can be represented as,



We can access this memory randomly(RAM)

For the w-bits, we can access them equally quickly.

- Lets consider that
  array = consecutive chunk of memory

ex: array for size 4

- that means, the i^{th} index of the array, can be found in the RAM

  array[i] = RAM Memory[address(array)+i] → offset arithmetic

  And we know, array access time = O(1)

That would mean,

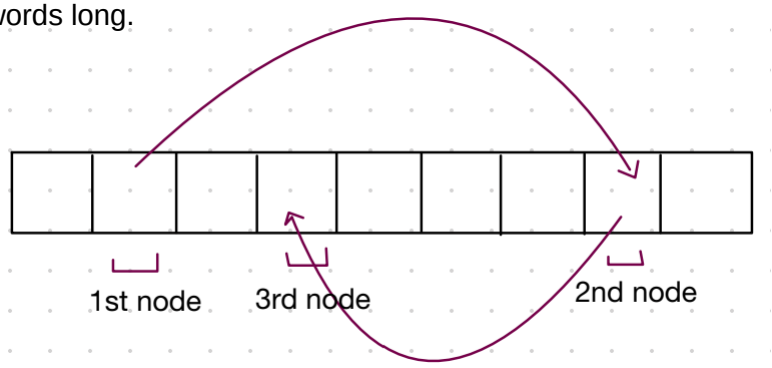
O(1) is for the '*get and set'* values.

- Build = O(n)
- Iterate = O(n)

- In Linked Lists, pointers can be stored in a single word, which means we can de-reference them (we can see what is on the other side of the pointer) in constant – time in our word – RAM model.

- In reality, each of these nodes is stored somewhere in the array of the computer (RAM), in an arbitrary order.

- Using this, we can allocate an array of size n (in linear time), where each data structure in 2 words long.



The pointers are just indices, in a giant memory array.

Takeaways for LinkedLists:

1. Insert & Delete() -> first element = O(1) time

2. Everything else is slow,
   - put in the i^{th} position
   - get(i)
   - find(i)
   - insert/delete at the end of the list

Conclusion:

- Arrays are great at random access

- Linked Lists = bad at random access, but great if you are working on the ends

Note: It may be asked in the interview, how to access the linked list's last element in constant time.

Ans: Start a point at the end of the list – tail pointer. This is also known as D.S. augmentation.

Data Structure	Static	Dynamic		
	Get_at(i)	Insert_first(x)	Insert_last(x)	Insert_at(x)
	Set_at(i)	Delete_first(x)	Delete_last(x)	Delete_at(x)
Array	O(1)	O(n)	O(n)	O(n)
Linked List	O(n)	O(1)	O(n)	O(n)
Dynamic Array (List in Python)	O(1)	O(n)	O(1)	O(n)