**Data Structure and Algorithm**

Contents

[Big O : 1](#_Toc81404962)

[Array: 1](#_Toc81404963)

[Arrays in Memory: (as a data structure) 1](#_Toc81404964)

[Sort Algorithm: 2](#_Toc81404965)

[Bobble Sort Algorithm: 2](#_Toc81404966)

[Insertion Sort: for(j end --)for (i->j) 3](#_Toc81404967)

[Shell Sort Algorithm: 3](#_Toc81404968)

[Recursion: 4](#_Toc81404969)

[Merge sort: 4](#_Toc81404970)

[Quick Sort: 5](#_Toc81404971)

[Counting Sort: 6](#_Toc81404972)

[Stable Counting Sort: 6](#_Toc81404973)

[Radix Sort: 6](#_Toc81404974)

[List: 7](#_Toc81404975)

[*ArrayList*: 7](#_Toc81404976)

[Stack 8](#_Toc81404977)

[Queue: 9](#_Toc81404978)

[Hash Table: 9](#_Toc81404979)

[Tree: 9](#_Toc81404980)

[Heap: 11](#_Toc81404981)

[Search Algorithm: 12](#_Toc81404982)

[Linear Search 12](#_Toc81404983)

[Binary Search: 12](#_Toc81404984)

Abstract Data Type (ADT) vs Data Structure (DS):

|  |  |
| --- | --- |
| **ADT** | **DS** |
| Logical level and a description | Concrete and Implementation Level |
| Logical picture of data and operations to manipulate the component elements of the data | Actual representation of the data during the implementation  Algorithms to manipulate the data elements |
| Represent particular set of behavior (ex. Stack is LIFO) | Technique and strategy for implementation(ex. Using linked list or array to implement stack)-**Using Libraries**-java.util.Stack … |
| Stack, queue, priority queue, dictionary, sequence, set | Array, linked list, hash table, trees |
|  |  |
|  |  |

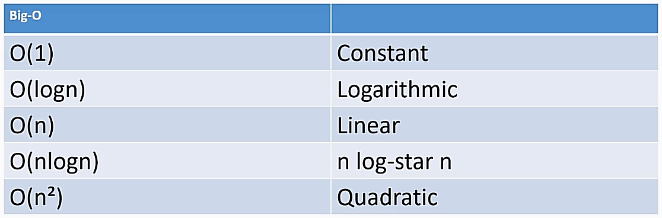
**data structure:**  to organize and store data.

**Algorithm**: steps you have to perform

***Arrays***: is a kind of data structure that each value has an index and order data sequentially

***Trees***: is hierarchical data structure or abstract data type like parents and children notion

Big O : to measure the time complexity of an algorithm. The smaller the Big O, the better algorithm you have.



## Array:

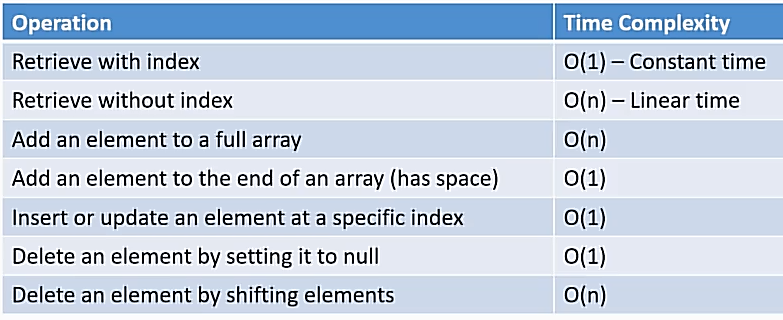
It is not a dynamic data structure because:

1- You have to set a size when you declare it

2- You cannot change its size

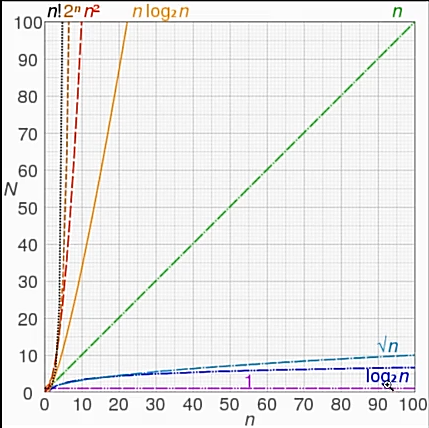
## Arrays in Memory: (as a data structure)

* **Contiguous block in Memory:** Arrays occupy a block of memory (huge block) and each *item* have equal amount of size which is *4 byte* (strings and objects reference address are stored in the array’s element)- that is why it cannot be resized
* If you have the address of first element of array in the memory you can calculate other elements address and reach the item easily by multiplying the index and the size of elements (that’s why arrays starts from 0, because the first item is multiplied by 0 and added by first memory address)



So if it requires to use loops the time complexity is O(n), and if it does not require the time complexity is O(1)

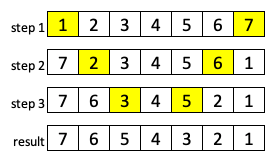
For counting the Big O notation or time complexity you should consider the worst case of the algorithm.



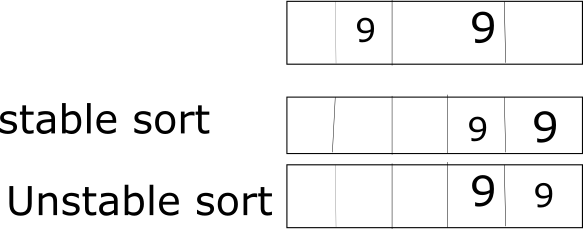
## Sort Algorithm:

Bobble Sort Algorithm: to sort an array of numbers, by comparing one element with the next element over and over. In each swipe you will find the last or the first element and you need to continue it over and over.

It is an *in-place algorithm* because you do not need to use another array to store data but a temporary variable. O(n2) time complexity.



Stable and Unstable Sort Algorithm: When you have same elements in your data, for example arrays, the location of same elements might change. If the location of the same elements changes by sorting algorithm, it is unstable and if it doesn’t change the order it is stable.



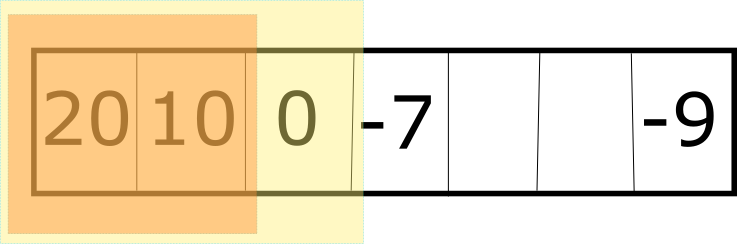
#### Selection Sort:

It is an In-Place algorithm (it means that does not need memory) and we compare each element and store the **maximum element indexes**, then at the end of that loop we will assign the element of that indexes to the last available indexes. Each time we move downward like bubble sort. Time complexity is O (n2). It is not a stable sort.

### Insertion Sort: for(j end --)for (i->j)

It sorts an array by sorting each partition, when sweep through each element. Each partition will sort again when reach new partition with a new element. This sort is start from the beginning.

It is an In-Place Algorithm and Stable, O (n2), and quadratic.



public class InsertionDemo {  
 public static void main(String[] args) {  
 int[] sArray = {20, 10, -5, 0, -15, 55, 7 };  
 for (int unsortedElement = 1; unsortedElement < sArray.length ; unsortedElement++) {  
 int newElement = sArray[unsortedElement];  
 int i;  
 for (i = unsortedElement; i >0 && sArray[i-1] > newElement ; i--) {  
 sArray[i]=sArray[i-1];  
 }  
 sArray[i]=newElement;  
 }  
 }  
}

### Shell Sort Algorithm:

Is a variation of insertion sort and it says if the insertion sort is partly sorted, the time complexity of sorting will become linear instead of quadratic.

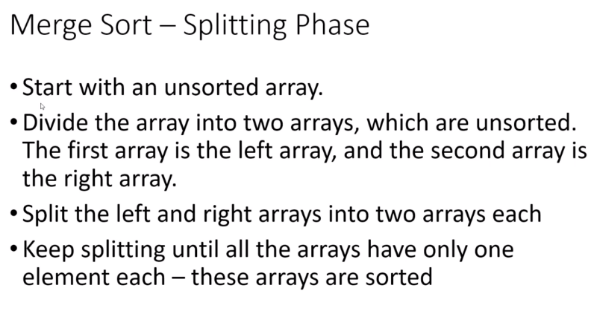
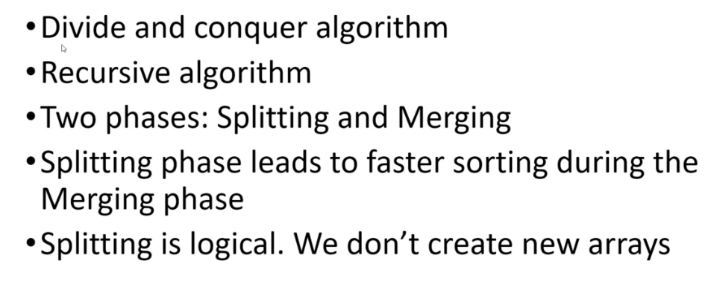
Shell started out sort insertion with higher gap and then reduced the gap to reach 1(normal insertion sort). So to start with how big gap is the thing that impact on time complexity (you can check on Wikipedia) Knuth is propose that the best way is to start with the gap of (3k-1)/2 which is smaller than number of elements of array.

public class MyShellSort {  
 public static void main(String[] args) {  
 int[] elements = {50,7,-1,9-26,0,55,2,87,-100,23};  
 for (int gap = elements.length/2; gap>0 ; gap/=2) {  
  
 for (int i = gap; i < elements.length ; i++) {  
 int newElement = elements[i];  
 int j = i;  
 while(j>=gap && elements[j-gap]>newElement){  
 elements[j]= elements[j-gap];  
 j-=gap;  
 }  
 newElement = elements[j];  
 }  
 }  
 }  
}

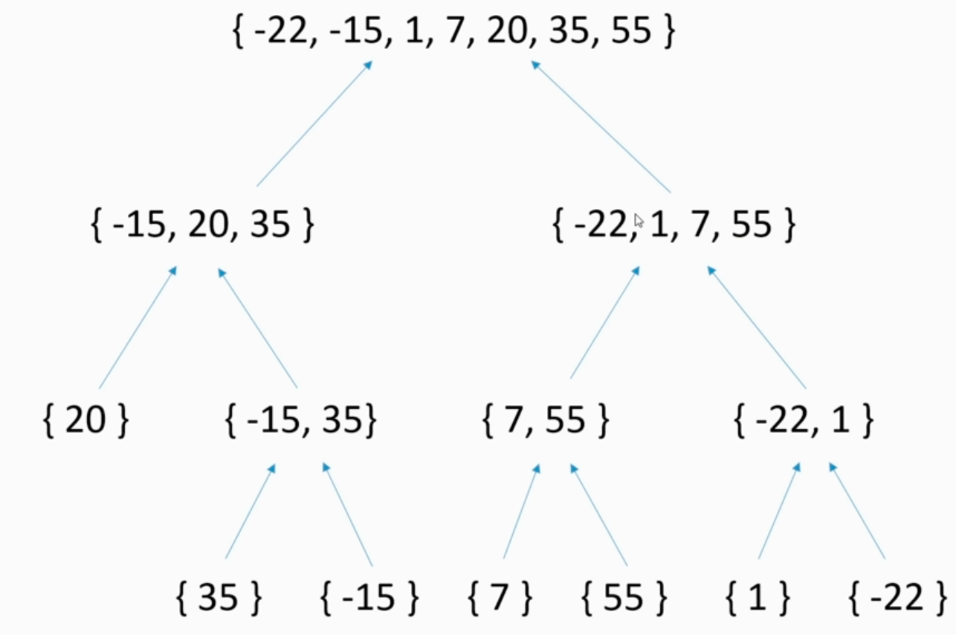
## Recursion:

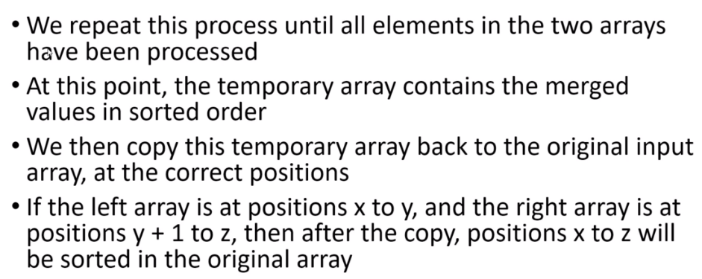
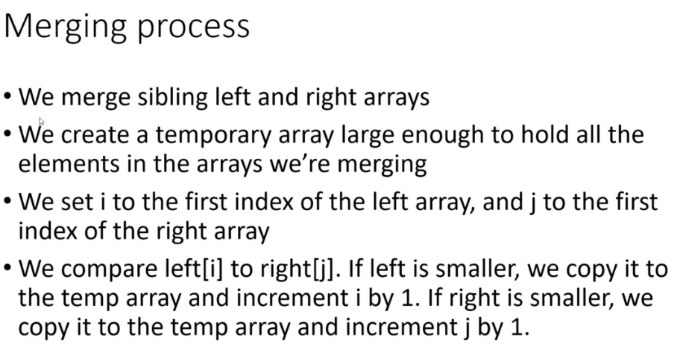
Means calling a method inside itself. Most of the time the iterative models are faster than recursive ones but as they are easier to read and understand, developers use recursion.

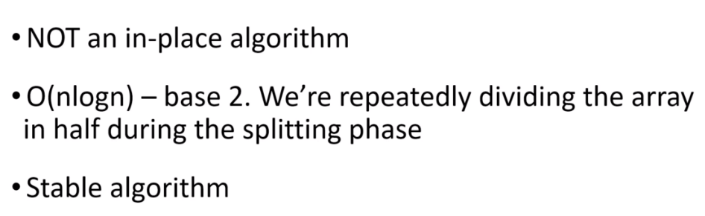
### Merge sort:



There are two parts here, firstly dividing to single element, by 2 recursively. Secondly, merging them in order. O(nLogn)







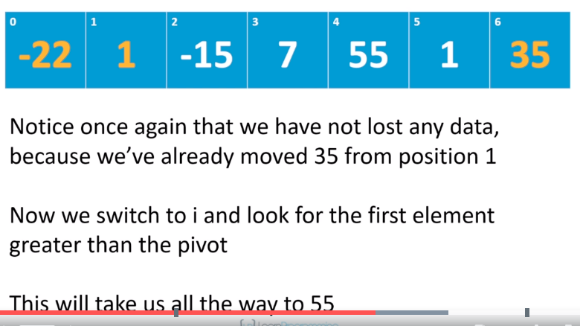
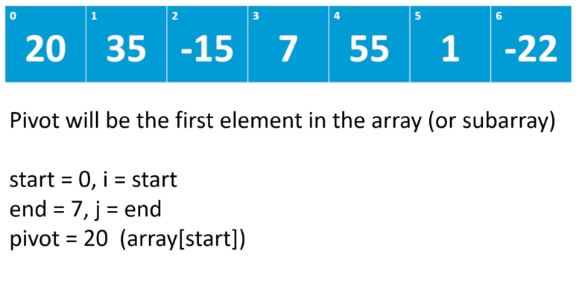
public static void split(int[] arr, int start , int end){  
  
 if( end - start < 2 ){  
 return;  
 }  
  
 int mid = (end + start)/2 ;  
 *split*(arr, start, mid );  
 *split*(arr, mid, end);  
 *merge*(arr, start, mid, end);   
}  
  
//20, 35, -15, / 7, 55, 1, -22  
private static void merge(int[] arr, int start, int mid, int end) {  
  
 //mid-1 is last element of left and mid is first element of right  
 //stops us from further action  
 if(arr[mid-1] <= arr[mid]){  
 return;  
 }  
  
 int i = start;  
 int j = mid;  
 int tempIndex = 0;  
  
 int[] temp = new int[end - start];  
  
 //33 ,35 34, 36  
 while(i < mid && j < end ){  
 temp[tempIndex++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];  
 }  
  
 //33 , 36 34 , 35 -> 36 should be added  
 System.*arraycopy*(arr, i, arr, start + tempIndex, mid-i);  
 System.*arraycopy*(temp, 0, arr, start, tempIndex);  
}

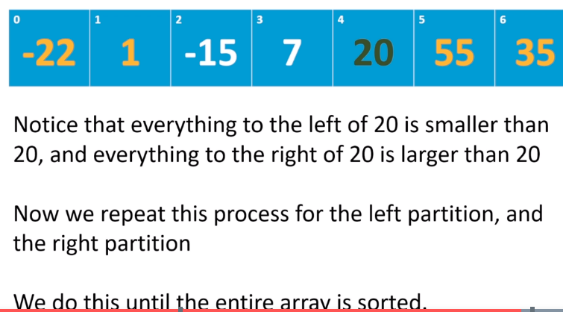
### Quick Sort:

Can be written recursively. Choose pivot element and divide the array to two half. Left lower than pivot element, and right half, elements bigger than pivot element. (Partitioning step) we do it recursively for the right and left sides.

In place/ no-need so much memory (advantage over merge sort) - most of the time O(nLogn) in worst case it can be O(n2) - unstable

First element would be pivot (here is 20), start = 0, and end = 7. We alternatively check elements from end and beginning and add to i and remove from j until i and j reach together. Now, we do this recursively from left and right part of the j element.





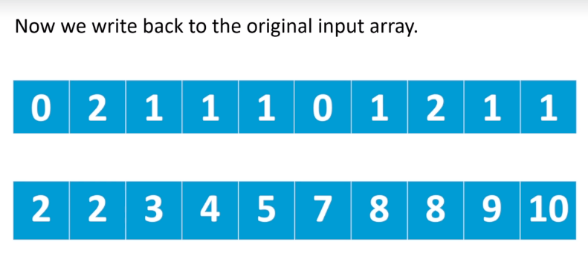
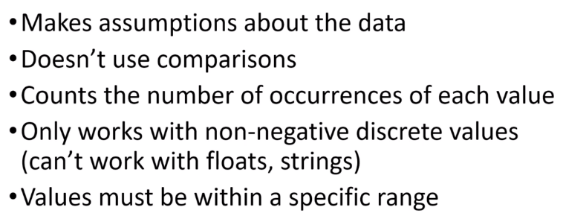
private static void quickSort(int[] array, int start, int end) {  
  
 if(end - start < 2 ){  
 return;  
 }  
  
 int pivotIndex = *partition*(array, start, end);  
 *quickSort*(array, start, pivotIndex);  
 *quickSort*(array, pivotIndex + 1, end);  
}  
  
private static int partition(int[] array, int start, int end) {  
 int pivot = array[start];  
 int i = start;  
 int j = end;  
  
 while( i < j ){  
  
 while( i < j && array[--j] >= pivot ); //empty while, just to decrement j  
 if( i < j ){  
 array[i] = array[j];  
 }  
 while (i < j && array[++i] <= pivot);//empty while  
 if(( i < j)){  
 array[j] = array[i];  
 }  
  
 }  
 array[j] = pivot;  
 return j;  
}

### Counting Sort:

ASSUMPTION: all the data are discrete and within specific range. No string, No negative, not huge

Not in-place, best when values in the range of certain values. O(n)-> because of assumption

Ideal for sorting arrays which the range of numbers and number of items that are close



public static void countingSort(int[] array, int min, int max){  
 int[] counter = new int[max - min + 1];  
  
 for (int i = 0; i < array.length ; i++) {  
 counter[ array[i] - min ]++;  
 }  
 int j = 0;  
  
 for( int i = min; i <= max; i++){  
 while(counter[i-min] > 0){  
 array[j++] = i;  
 counter[i-min]--;  
 }  
 }  
}

### Stable Counting Sort:

The way to change the conting sort to a stable counting sort is: after finishing the counting array, we should declare the location of each item. For example, in the above example, {0,2,1,1,1,0,1,2,1,1} 🡪{0,2,3,4,5,5,6,8,9,10}

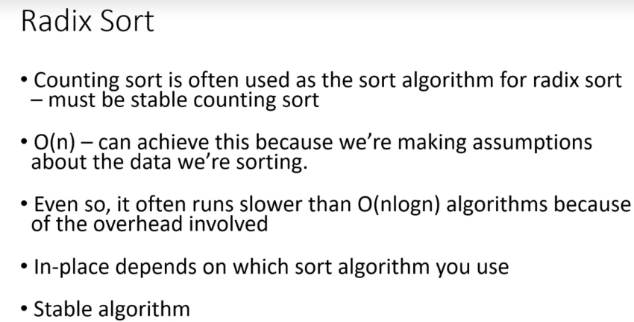
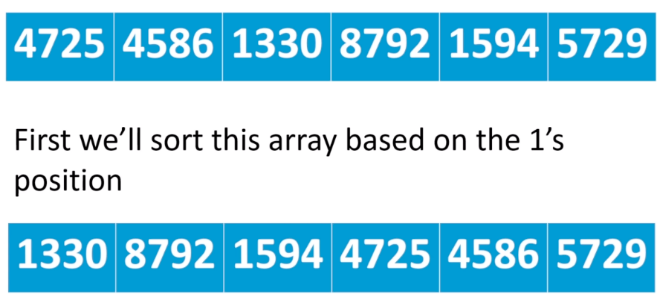
So by this shifting, we can declare each item’s exact location.

Radix Sort: (data with same radix and width)

We cannot use this sort for numbers with decimal point( just integer and String)

The key element here is that, in each step, we have to use stable sort algorithm .

ASSUMPTION: Data must have same radix and width (1234, radix and width of 4, 10 is 2, 0 and 1)



All have 4 digits, so we can use radix algorithm here. We sort this list of numbers in 4 steps. First phase, we sort numbers by 1 position, then 10’s, then 100’s, and 1000’s.

**There are 5 steps:**

1. Get the digit of each position (exp. 1🡪9)
2. Locate its stand an add to the counting array (exp.: {{0,2,1,1,1,0,1,2,1,1}})
3. Readjust the array (exp.: {0,2,3,4,5,5,6,8,9,10})
4. Add the items to a temporary array based on the location defined above

**NOTE: you have to add the items in reverse form to keep it stable**

1. Replace temp. array to the array

May achieve linear time, because it has assumption. Might in-place or not. stable.

## List:

### ArrayList:

When you instantiate it, as default, it creates a list of 10 elements (you can add more if you want). Add or remove in arrayLists are slow, so it is better to add the capacity if you have large items to be added. Capacity is the number of elements that can be added to an array, and size() is the number of items being added to the list.

For instantiating ArrayList<> it is better to use List to implement ArrayList, because it is easier to shift to another form of list later.

Equals (): we can override it to compare two objects in a LinkedList.

**LinkedList is not good in**: Add, remove and accessing items w/o having the index

#### Vector:

Vector is synchronized but arraylist is not, so vector is thread-safe. Arraylist is published after vector to be used where we don’t need multithreading and as a lighter form of vector.

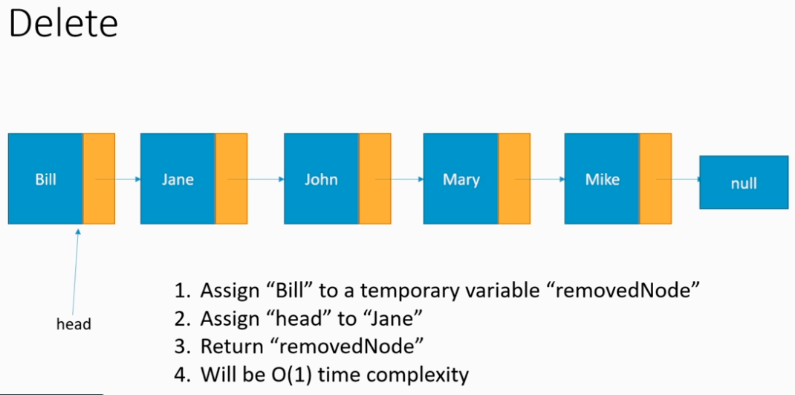
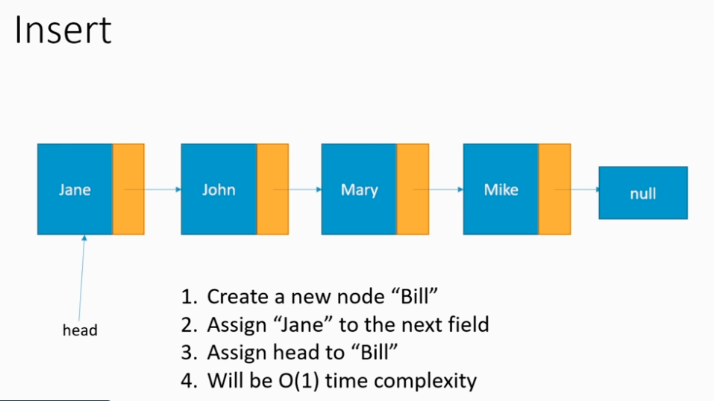
#### LinkedList:

##### Singley:

Each item aware of another item in the list because each item which is known as node contains a link of another item. So we have a node, which first element of it is data, and second one is the link to the next element.

So it is a class containing a node with two field. First field is data that stores in the node, and second one is another node. We have a head that points to a node.

Time complexity of inserting item to the head of the linkedList is O(1) because it does not depend on the size of data.



For deleting the first item, we just shift the header link to the next node. Time complexity here is also O(1). That’s because for inserting item to the list, we don’t shift elements.

We have singly linked list and doubly linked list.

**Disadvantage**: more data that should be stored.

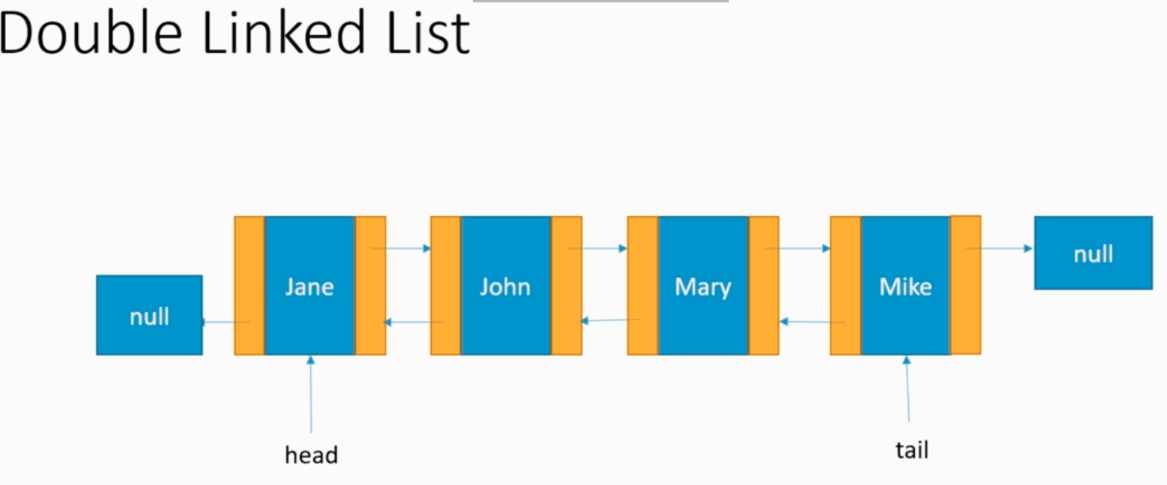
**To create**: we need Object, Node to store object and its node pointer (node), and LinkedList to assign head to an object.

**NOTE**: if you want to use linked list as an stack, you just add item to the head, while if you want to used it as a Queue you should add item to the tail .

##### Double:

It is a node containing a data instance, head and tail node to point to previous and next node. This helps to get the first and last item of the list with constant order. But for reaching an item inside the list, you need to traverse through the list like singly linkedList.

In order to add a field in between two fields or remove, firstly we should find the location which has O(n) time complexity.



Note: if you want to use linkedList in multithread, you need to call synchronized the class by yourself.

Circular Linked List: the only difference is that the tail node address to the head node

Stack:

* An abstract data type that dictates what operation we can perform on a set of data and does not tell us how the data should be stored.
  + LIFO (top element) useful for call,
  + Push (add to the top of the list)
  + Pop (get the top and remove it)
  + Peek (get the top)

Best way use linked list. With LinkedList its order is O (1) in Array is O (n). If you know the maximum index of stack, you can use array also.

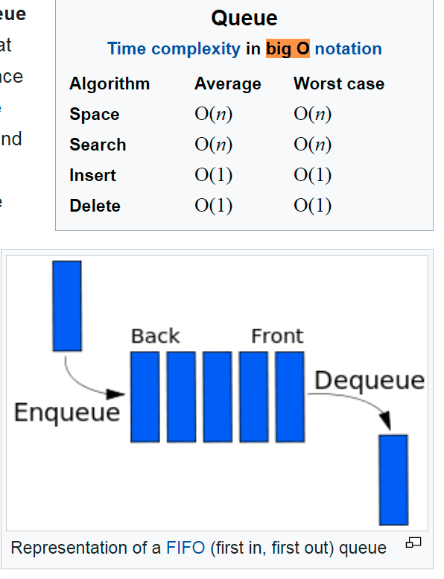
There is a Stack built-in class in java, which is not recommended to be used. It mentions that it would be better to use Deque instead. But, as linkedList is implemented Deque interface, the best way to apply Stack in java is by using ***LinkedList*** and benefit its push (), pop (), and peek () methods.

پس بهترین راه برای ساختن یک stack اینه که یک کلاس جدا ایجاد کنی و متدهای مربوط به stack را با linkedList پیاده کنی.

## Queue:

Is an array of data, which stores data by FIFO principle. There are **back** and **front** pointer in it which you can add() item and remove() item from the Queue. Added item will add to the end of the array(back) and remove method, removes the older object in the front index.

You should write the code in a way that update and double the array as less as possible.



### Hash Table:

Is an **Abstract data type**: it does not dictate how to store data- exist for speedy retrieval data

Store data based on **Key** and **value**. Maps, Dictionaries, Look up tables and associative arrays are Hash Table.

Associative array is one type of Hash Table (in php).

You can use any keys, but those keys are converted to integer. We use Hashing to convert any data type to int. Any object as hashCode() method. It happens when two integer assigns to a key, we call it **Collision**.

In hashing we have product number (**Key**) and product (**Value**)

**Load Factor**; how full an array is. Load factor = # of items / capacity = size/capacity

Is used to decide when to resize the array backing the hash table.

we take a key -> map it to an integer by hash function -> and use that integer to retrieve the item

to store data we use array.

# Tree:

Is a Hierarchical data structure. Ideal when things contain other, or in inheritance (in java classes) Useful when items are dependent on another item.

**Node**: each element of a tree (each node has only one **parent** EXCEPT root node doesn’t have any parent) *parent-children relationship*

**Leaf nodes**: Node with no children

**Sub-tree**: Each tree has one or several sub-trees. All the nodes downward

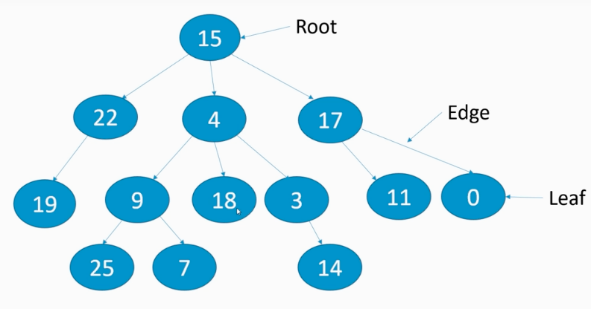
**Edge**: arrows from parent to children (only p->c)

**Path**: a sequence of moving from one node to another. (**There is no cyclic path in Tree**).

**Root-Path**: A path from a node upward to its parent’s node. Number of edges in this way, is called depth of the path **depth of path (upward)**. Depth of root node is 0. **Height of a node (downward)** is the longest path from the node to a leaf. Height of 4 is 2. A leaf node height is 0. Tree height is the root node height.

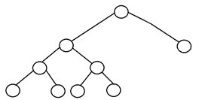
**Ex**.: - Like classes in java, because each class can only extends one class, so it has only one parent.

* Like drive, folders and files

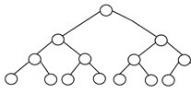
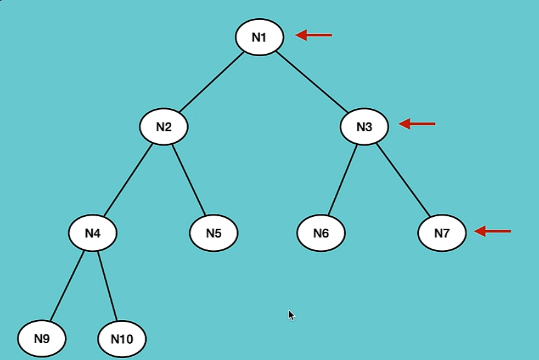


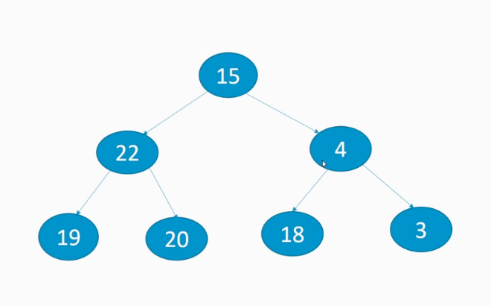
#### Binary Tree:

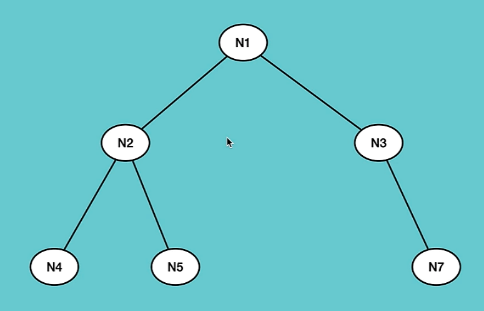
Nodes have 0, 1, or 2 children

**Full binary tree** : each node has 0 or 2 children 

**Complete binary tree** means every node has 2 children except last node(it can have a node with one child, but it should be at the most left side of the leaf ), or leaf. And it should be at the most left side of the tree.

**Perfect Binary tree:** each node has **2 children** except the leaf nodes and all **at the same level** 

**Balanced Binary tree:** all the nodes at the same level should have the same height . 

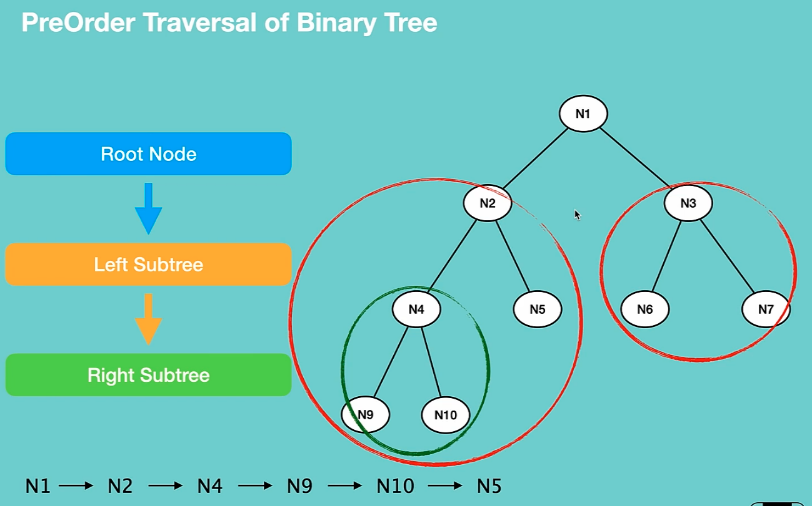
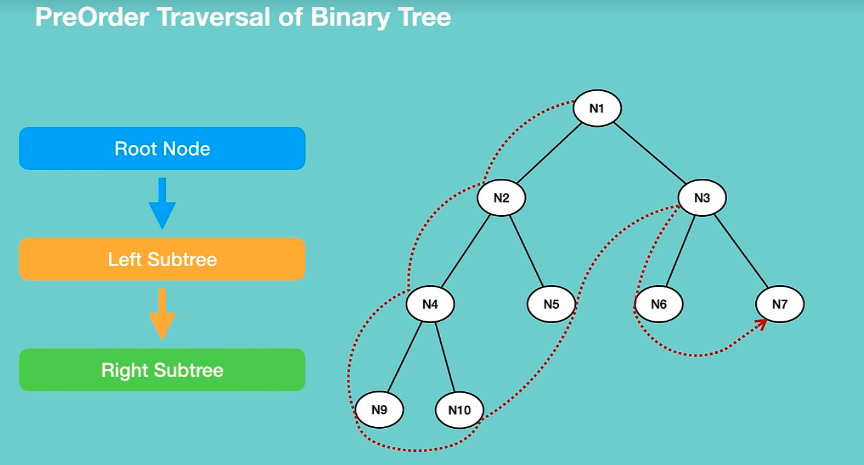
**How to create a tree:**

**We can do it by 2 ways:** 1- by using list, 1- by using linkedList (each node stores data and consists of left and right element )

We have 4 types of traversing over trees’ items:

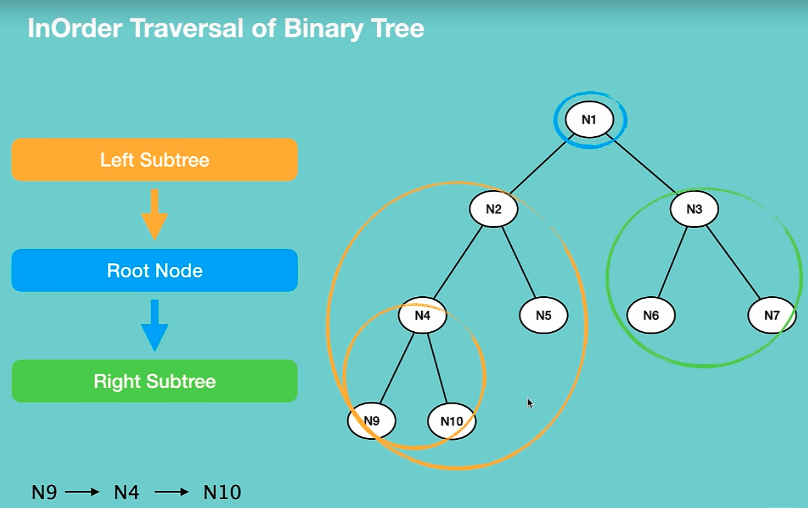
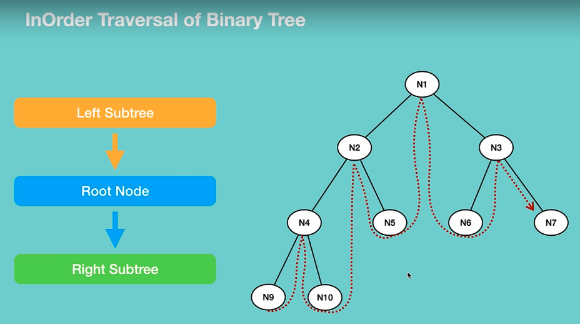
**Pre-order,** **in-order**, **Post-Order(Depth-first search),** and **Level-Order(breadth-first search):**

**For pre-order tree:** *root-> left subtree -> right subtree (recursion) : O(n)*

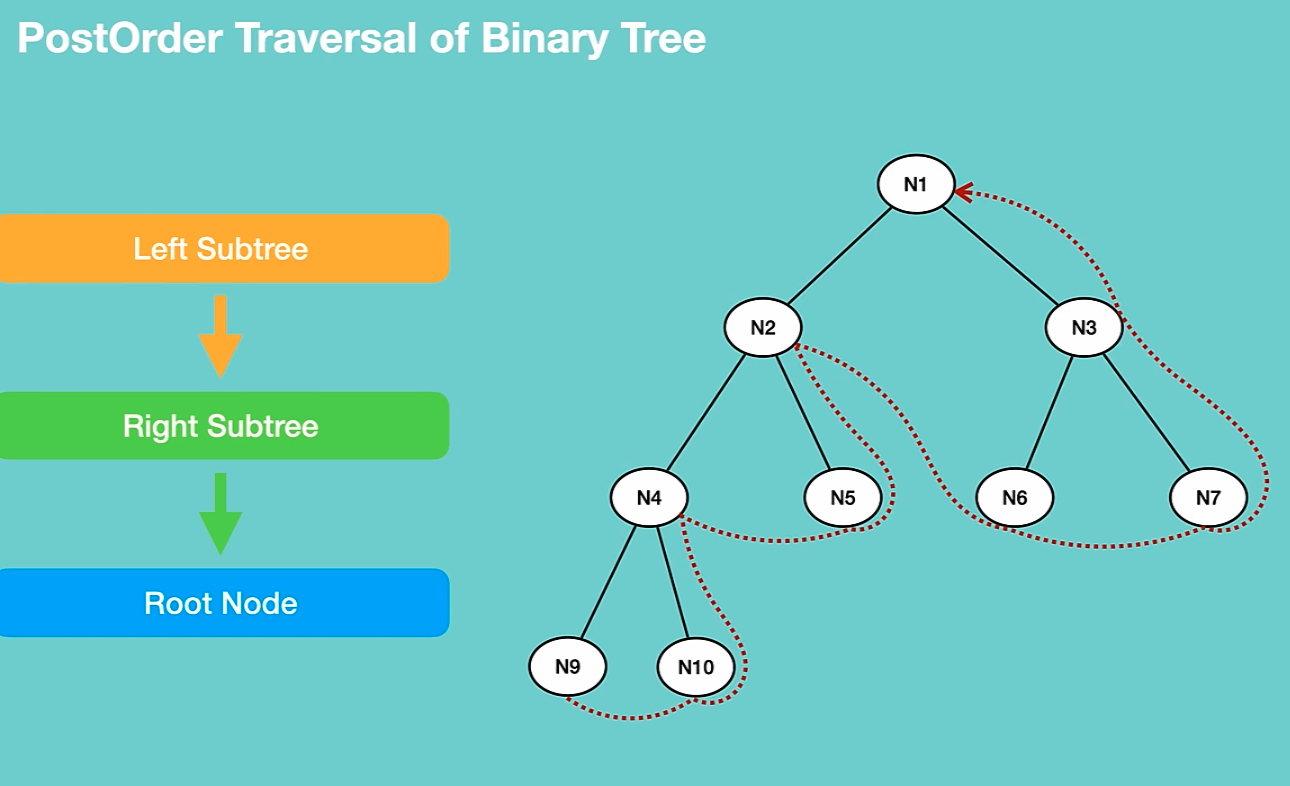
def preorderTraversal(newRoot):  
 if not newRoot:  
 return  
 print(newRoot.value)  
 preorderTraversal(newRoot.left)  
 preorderTraversal(newRoot.right)

**For in-order tree:** *left subtree->root->right subtree (recursion): O(n)*

def inOrderTraversal(rootNode):  
 if not rootNode:  
 return  
 inOrderTraversal(rootNode.left)  
 print(rootNode.value)  
 inOrderTraversal(rootNode.right)

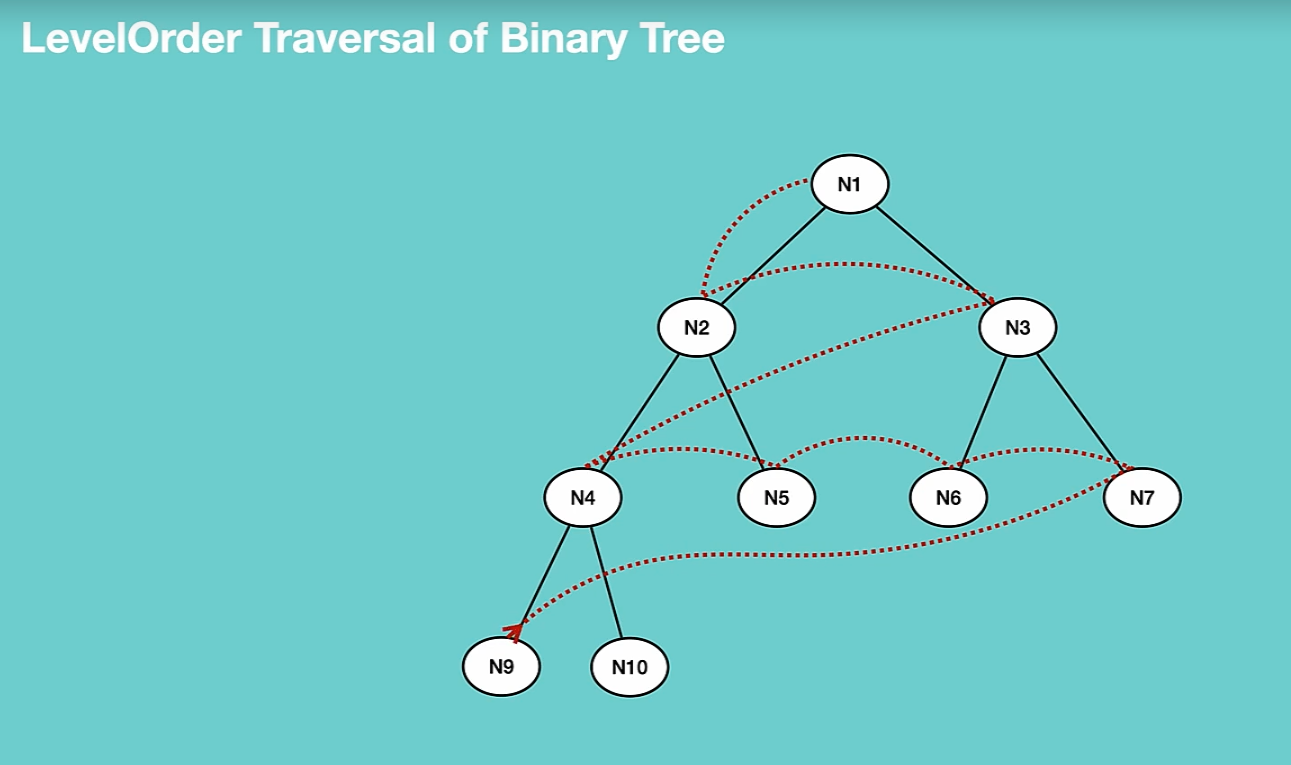
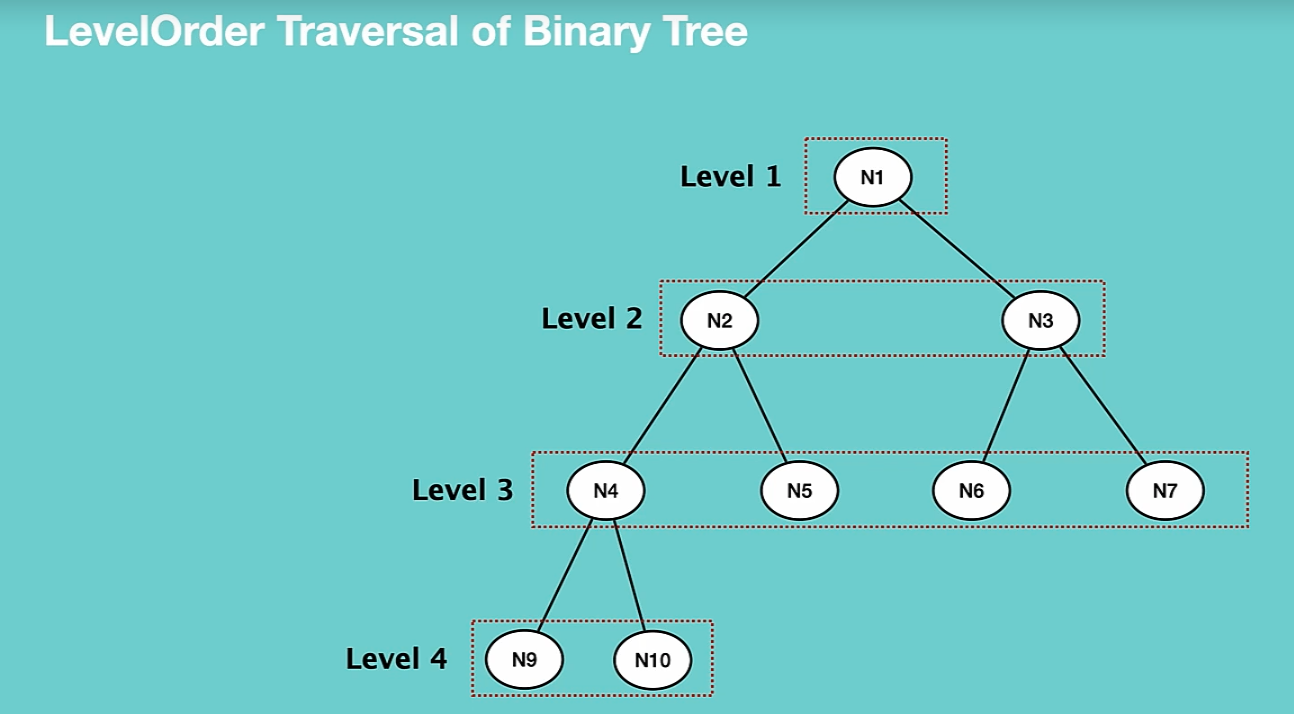
**Post-Order traversal:** *left subtree -> right subtree -> root (recursion) : O(n)*



def postOrderTraversal(treeNode : TreeNode):  
 if not treeNode:  
 return  
 postOrderTraversal(treeNode.left)  
 postOrderTraversal(treeNode.right)  
 print(treeNode.value)

**Level order Traversal Model: with the benefit of queue O(n)**

This is the best traversal model for finding a node. Because it uses queue that is much efficient than stack. Also for adding an item to tree we use this travers method. -+



Result: N1->N2->N3->N4->N5->N6->N7->N8->N9->N10

def levelOrderTravers(treeNode: TreeNode):  
 if not treeNode:  
 return  
 queueStorage = queue.Queue()  
 queueStorage.enqueue(treeNode)  
  
 while not queueStorage.isEmpty():  
 root = queueStorage.dequeue()  
 print(root.value.data)  
  
 if root.value.left is not None:  
 queueStorage.enqueue(root.value.left)  
  
 if root.value.right is not None:  
 queueStorage.enqueue(root.value.right)

for a Tree with LinkedList base and using queue for traversing, these methods can be implemented by using level order traverse:

insert, delete a node(find a node by value, find the last node, delete the last node, delete a node), delete the Tree(see Tree directory)

We can create the Tree by python list too: we start from index 1 for easier numerical operation.

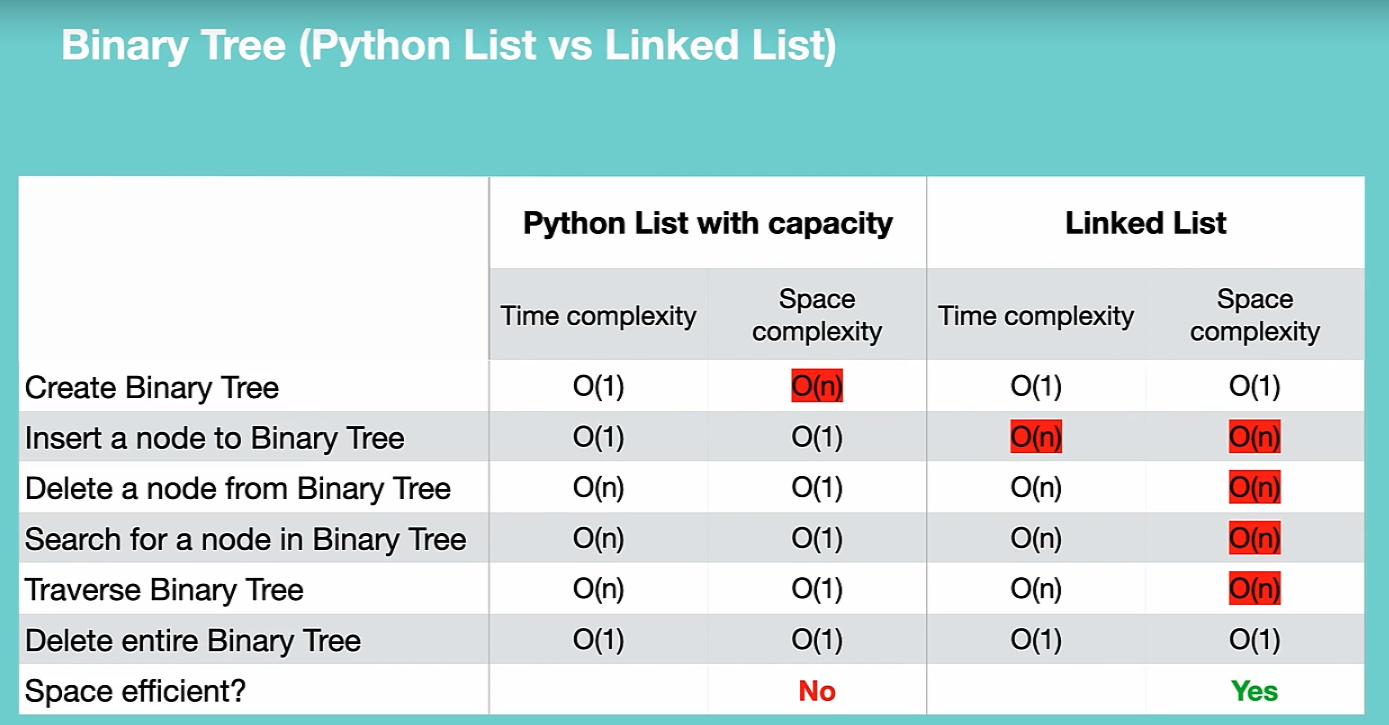
Each node (n) has left and right node as: 2n, 2n+1

Writing traversal functions with list is easier and the same as queue.

**Tree with LinkedList vs with python list**:

Use Linked List If: you have issue with space and you don’t know how many elements you have

Use List if : You know the number of elements



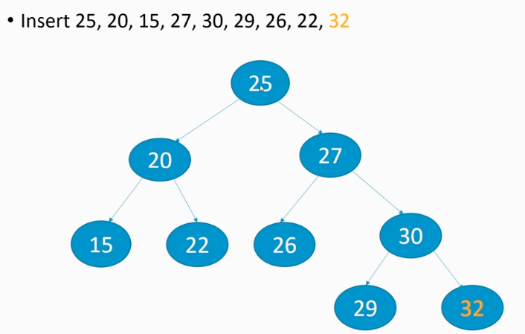
##### Binary Search Tree (BST):

In practice we cannot use tree, but BST. It can performs insertions, deletions, and retrievals in O (log n).

Left child has smaller value than root, and right child has bigger value (**Rule**). Because of this, we can do a binary search.

Faster search time than unsorted list.

The order of inserting numbers impact on the appearance of a tree.



Minimum value: if we go all the way down to the left (15)

Maximum value: if we go all the way down to the right (32)

In implementing the Tree: we have a class containing Left and Right TreeNode in TreeNode class. Another class is Tree that has a root node. To insert() to the tree, we need an insert method in TreeNode to add to left or right and an insert method in Tree class to add root for the tree if it is null.

**Traversal: similar to the binary tree**

* **Level**: visit nodes on each level -> **(25, 20, 27, 15, 22, 26, 30, 29, 32)** UP TO DOWN
* **Pre-order**: visit the root of every subtree first -> **(25, 20, 15, 22, 27, 26, 30, 29, 32)** entire sub-left trees and entire sub-right trees then root
* **Post-order**: visit the root of every subtree last -> (**15, 22, 20, 26, 29, 32, 30, 27, 25**) Left Right Root
* **In-order**: visit left child, then root, then right child. (**data is sorted and faster than SOME OF the sort algorithms**) -> **(15, 20, 22, 25, 26, 27, 29, 30, 32)** starts from LEFT to ROOT and RIGHT

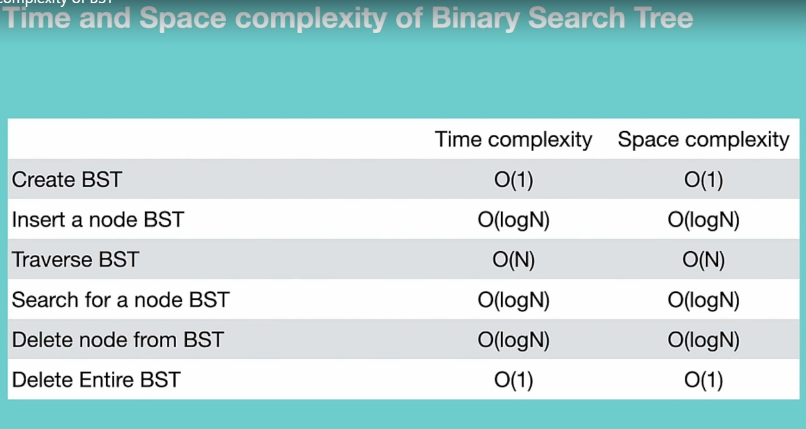
**For deleting a node:**

If the node that is supposed to be deleted is a leaf: easily delete it

If it has only one left or right node: replace it with right or left,

But if it has both right and left: we have to find the minimum right value of node.right subtree. Then substitute its value with the node and then delete the node with minimum value:

*# deletion of Node*def minvalue(rootNode : BSTree):  
 currNode = rootNode  
 while (currNode.left is not None):  
 currNode= currNode.left  
 return currNode  
  
  
def deleteNode(rootNode: BSTree, value):  
 if rootNode is None:  
 return rootNode  
  
 if rootNode.data > value :  
 rootNode.left = deleteNode(rootNode.left, value)  
  
 elif rootNode.data < value:  
 rootNode.right = deleteNode(rootNode.right, value)  
  
 else:  
 if rootNode.left is None:  
 temp = rootNode.right  
 rootNode= None  
 return temp  
  
 if rootNode.right is None:  
 temp = rootNode.left  
 rootNode = None  
 return temp  
 temp = minvalue(rootNode.right)  
 rootNode.data = temp  
 rootNode.right = deleteNode(rootNode.right, temp.data)  
 return rootNode



## Heap:

(Complete binary, max or min heap)

A special type of binary tree. A **complete** (all full except deaf level) **binary** tree that satisfy the heap property (Max or Min). A full, mean every children should be full, if not, it must be as left as possible.

**Max** heap: every parent is **greater** than or equal to its children (root is max)

**Min** heap: every parent is **less** than or equal to its children (root is min)

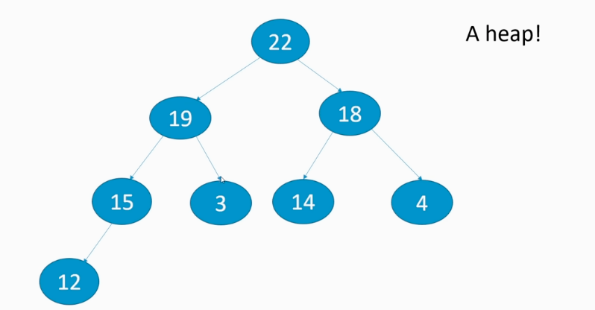
Usually implemented as arrays. The **max or min values are always at the root** of tree.

**Heapify**: process of converting a binary tree to a heap.

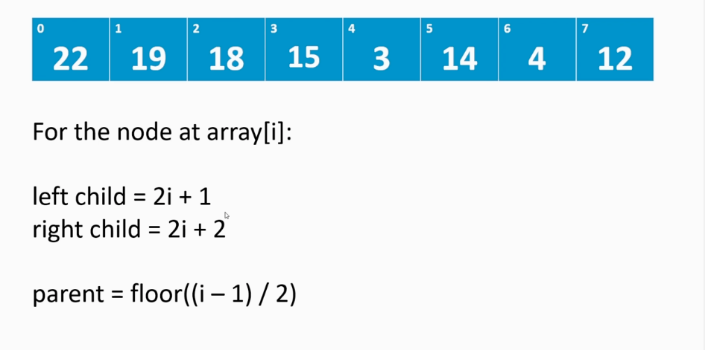
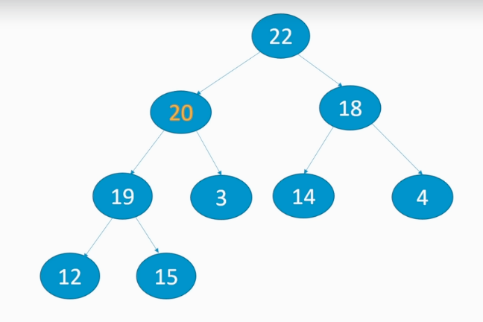
The point of using heap is to access max or min value in constant time. O(1)

The only relatives in heap is **parents** and **children**. It is not essential to have ordered items in a level.

In a heap, when you travel from top to bottom or vice versa, it should follow the descending or ascending order.



We can store heap as arrays. To find each parent left and right child we can follow the following order.

**Heapify:** the process of converting a tree to a heap.

To create a heap, we should write an arrays, and add items to it by following the above order and rules.

When we want to delete an item, first substitute the last item in array with the item we want to delet. Then we should heapify the heap based on min or max heap. Lets say, we want to delete,20, 15 is substituted with 20 and start heapify that. Move 19 upward to 15 nd it is done.

When we use heap, we generally working with the root.

**Insert**(O(logn), **because** of fixing the above items),

**Delete**(O(nlogn)), **peek**

Priority Queue:

This is the complete binary tree or heap in java. It is min heap and if you want max heap you need to provide comparator to the class. Highest priority is always in root. Limit numbers of priority can be stored

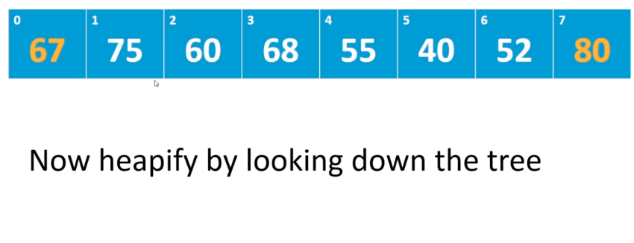
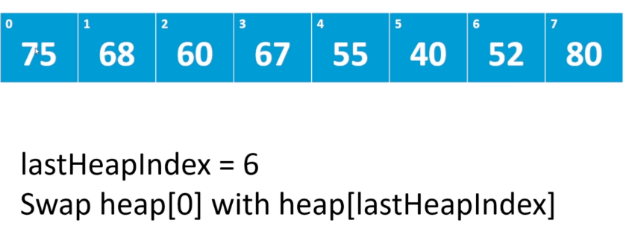
Not synchronized.

Peek and poll. Based on max or min heap it changes.

Poll and remove are the same in priorityqueue.

**Heap Sort:**

Max heap, root is larger. The way we do here is that, as the root is max, we swap the root with the last element of the array, heapify the remaining array and do this job again until cover all element of the array.

In place, O(nlogn)

## Search Algorithm:

Linear Search: iterate over array items with a for loop and return the index of the element that maches. O(n)

### Binary Search:

Requires a sorted data. First should sort the data. then compare the element with the item in the middle of array. If it is greater, then we compare with the right hand side of the element, else we check right elements. O(logn)

There are two ways of implementing binary search. Iterative model and recursive. Almost the same, but iterative is much better.

private static int binary(int[] input, int value) {  
 int start = 0;  
 int end = input.length;  
  
 while(start < end){  
 int mid = (end + start)/2;  
 if(value == input[mid]){  
 return mid;  
 }else if(input[mid] <value){  
 start = mid + 1;  
 }else{  
 end = mid;  
 }  
 }  
 return -1;  
}  
  
public static int binaryRecursive(int[] input ,int start, int end , int value){  
 int mid = (end + start )/2;  
 if(end - start < 1){  
 return -1;  
 }  
  
 if(input[mid] == value){  
 return mid;  
 }  
 if(input[mid]< value){  
 return *binaryRecursive*(input, mid+1 , end , value);  
 }else{  
 return *binaryRecursive*(input, start, mid, value);  
 }  
}