**Data Structure**

**Data Structure and Algorithm**

1. [Basic](#ds_basic)
   1. [Introduction](#ds_introduction)
   2. [Recursion](#ds_recursion)
   3. [Algo Run Time Analysis](#ds_run_time_algo)
2. [Physical Data Structure](#physical_ds)
   1. [Array](#array_ds)
      1. [Single Dimension Array](#oned_array)
      2. [Double Dimension Array](#twod_array)
   2. [Linked List](#linked_list)
      1. [Single Linked List](#single_ll)
      2. [Circular Single Linked List](#circular_single_ll)
      3. [Double Linked List](#double_ll)
      4. [Circular Double Linked List](#circular_double_ll)
3. [Logical Data Structure](#logical_ds)
   1. [Stack](#lds_stack)
   2. [Queue](#lds_queue)
   3. [Tree](#lds_tree)
      1. [Binary Tree](#lds_binary_tree)
      2. [Binary Search Tree](#lds_binary_search_tree)
      3. [AVL Tree](#lds_avl_tree)
      4. [Binary Heap](#lds_binary_heap)
      5. [Trie](#lds_trie)
   4. [Hashing](#lds_hashing)
   5. [Graph](#lds_graph)
4. Miscellaneous Data Structure
   1. Sorting
5. Algo Techniques

5.1. Introduction to problem solving Approach

5.2. Introduction to problem solving Patterns

5.2.1. Magic Framework

5.2.2. Greedy Algorithm

5.2.3. Divide and Conquer

5.2.4. Dynamic Programming

5.2.5. Frequency Counter

5.2.6. Multiple Pointer Patterns

5.2.7. Sliding window Patterns

**1. Basic**

**1.1. Introduction**

Data Structure is way to organize data in a way that enables it to be processed in an efficient time.

Algorithm is a set of rules to be followed to solve a problem.

Type of DS

Primitive DS Non-Primitives DS

Integer

Float

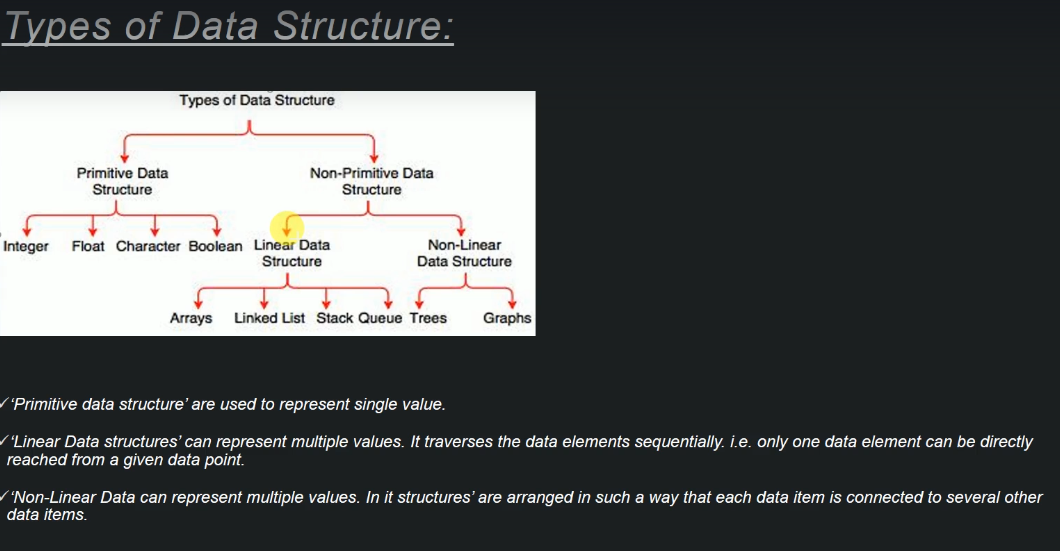
Character Physical DS Logical DS

Boolean Array Stack

Linked List Queue

Tree

Graph



**1.2. Recursion**

* A process / function that call itself.
* Same operation is performed multiple time with different inputs.
* In every step we try to make problem smaller
* We mandatory need to have a base condition when tells system when to stop the recursion
* Format of a recursive method

Recursive Case: Cases where the function needs to recall itself.

Base Case: Cases where the function doesn’t recall.

sampleRecursion(parameter){  
 if(baseCase is satisfied)  
 return same base case value  
 else  
 return sampleRecursion(modified parameter)  
}

* How recursion work internally

main(){ ----1  
 bar();  
 sop("main"); ----14  
}  
bar(){ ----3  
 doWork();  
 sop("bar"); ----12  
}  
doWork(){ ----5  
 doMore();  
 sop("doWork"); ----10  
}  
doMore(){ ----7  
 sop("doMore"); ----8  
}

STACK  
-----  
  
| |  
| |  
| doWork() | ----6, 9  
| bar() | ----4, 11  
| main() | ----2, 13  
|-----------|

main(){  
 foo(3);  
}  
foo(n){  
 if(n<1)  
 return   
 else  
 foo(n-1)  
 print "Hello n"  
}

| |  
|foo(1) |  
|foo(2) |  
|foo(3) |  
|main() |  
|--------|

* Program and examples

PrintNumber:

FactorialSeries:

FibonacciSeries:

NumberPalindrome:

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s1_c2_recursion>

**1.3. Algo Run Time Analysis**

It is a study of given algorithm run time. In layman language we can say “How much time will the given algorithm will take to run”.

Notification for algo run time analysis.

3 notations used to calculate run time analysis.

1. **Omega (Ω)**

* Min Time
* Given lower bound of a given algorithm.
* For any given input running time of a given algorithm will not be less than a given time.

e.g. sort 1000 number

Omega (10): Never take less than 10 sec. it may take 11,12…sec

1. **Big-O (O)**

* Max Time
* Given upper bound of a given time
* For any given input, running time of a given algorithm will not be more than given time
* We say that an algorithm if O(f(n)) if the number of simple operation the computer has to do in eventually in less than a constant time(f(n)) if n increase.
  + f(n) could be liner f(n) = n
  + f(n) could be quadratic f(n) = n2
  + f(n) could be constant f(n) = 1
  + f(n) could be something entirely different

E.g. sort 1000 input

Big-o=O (100): Time taken to solve is less than 100 sec. it may take 99, 98… sec

Rule of Thumb:

* O(2n) -> O(n)
* O(500) -> O(1)
* O(13n2) -> O(n2)
* O(n+10) -> O(n)
* O(100n+50) -> O(n)
* O(n2+5n+8) -> O(n2)

1. Arithmetic Operations are constant
2. Variable assignment is constant
3. Accessing elements in an array(by index) is constant
4. In a loop complexity is the length of loop times the complexity of whatever happens inside the loop.

1. **Theta**

* Decide whether upper bound and lower bound of a given algorithm are same or not.
* For given input running time of a given algorithm will on an average be equal to given time.

Example

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 13 | 3 | … | 55 | 41 | 23 | … | 19 | 1 | 10 | 17 |

Find a given number in that array

We search in each index. It may be in 1st index or it may be last index.so time take is n\*1.

So, Omega (1): it may be in 1st cell i.e. not less than minimum time

Big-O(n): it may be in last cell i.e. max time

Theta: (n/2): average time.

**Time Complexity Name Example**

O (1) constant Adding an element at front of a limited list

O (log n) logarithm Find an element on sorted array

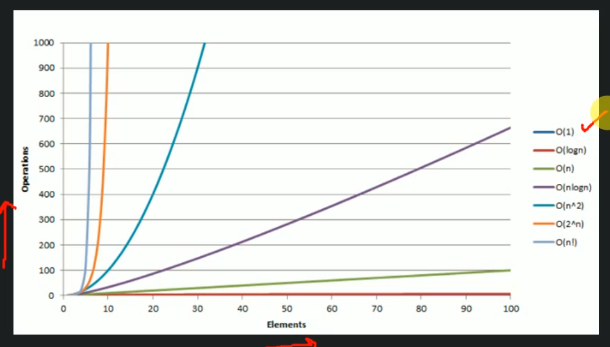
O (n) liner find an element on un sorted array

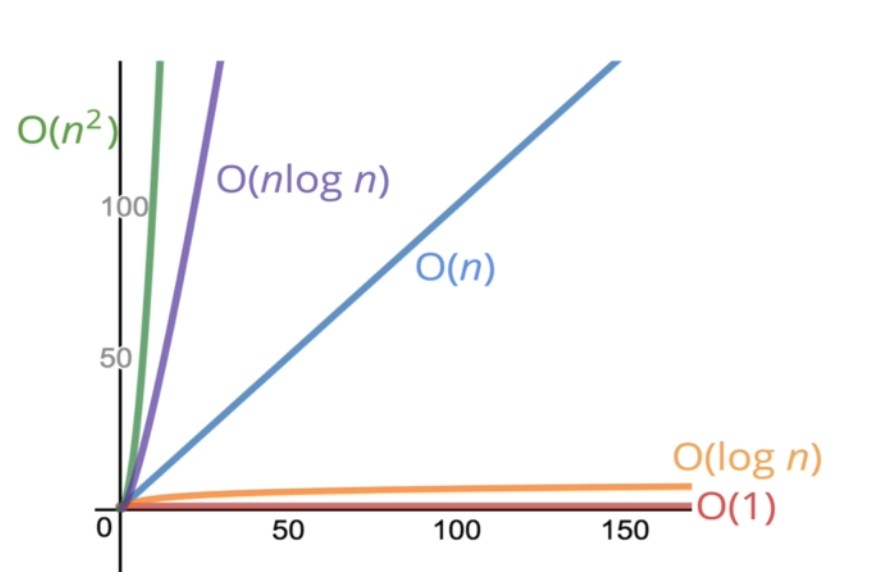
O (n log n) liner logarithm Marge sort

O (n2) Quadratic shortest path between 2 nodes in a graph

O (n3) Cubic Matrix multiplication

O (2n) Exponential Tower of Hanoi Tower





**Time Complexity:**

**Time Complexity of recursive algorithm #1**

How to calculate ‘Algorithm time complexity’

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 13 | 3 | … | 55 | 41 | 23 | … | 19 | 1 | 10 | 17 |

findBiggestNumber(int arr[])  
 biggestNumber = arr[0] ----------------O(1)  
 loop: i = 1 to length(arr) - 1 --------O(n) } -------O(n)  
 if arr[i] > biggestNumber }-------O(1) }  
 biggestNumber = arr[i] }  
 return biggestNumber ----------------O(1)  
  
 Time Complexity : O(1) + O(n) + O(1) = O(n)

**Time Complexity of recursive algorithm #2**

public long addUpToN(final long n) {  
 long sum = 0;------------------- 1 assignment  
 for (int i = 1; i <= n; i++) {  
 //---1 assignment(int i = 1), n comparison(i <= n), n addition and assignment(i++)  
 sum = sum + i; -- n assignment and n addition  
 }  
 return sum;  
}  
so totally 5n operations and and 2 assignment  
so roughly n operation O(n)

[011\_data\_structure/AddUpToN1.java at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/blob/master/src/s1_c3_bigO/AddUpToN1.java)

Reduced time complexity

public long addUpToN(final long n) {  
 return (n \* (n + 1)) / 2;  
 //-- 1 multiplication, 1 addition, 1 division  
 //so total 3 operations O(1)   
}

[011\_data\_structure/AddUpToN2.java at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/blob/master/src/s1_c3_bigO/AddUpToN2.java)

**Time Complexity of recursive algorithm #2**

public void countUpAndDown(int n){  
 sop("Going Up");  
 for(int i = 0; i < n ; i++){ }  
 sop(i); }---O(n)  
 } }  
 sop("At top going Down");  
 for(int j = n-1; j >= 0 ; j--){ }  
 sop(j); }---O(n)  
 } }  
 sop("Back down ! Bye");  
}  
//so total time complexity is O(n)

**Time Complexity of recursive algorithm #3**

public void printAllPairs(int n){  
 for(int i = 0; i < n ; i++){ }  
 for(int j = 0; i < n ; j++){ } }  
 sop(i, j); } ---O(n) }---O(n)  
 } } }  
 } }  
}  
//O(n) operation is inside another O(n)  
So time complexity is O(n2)

**Time Complexity of recursive algorithm #4**

public void logAtLeast5(int n){  
 for(int i = 0; i < Math.max(5, n) ; i++){  
 //O(n) i.e. (5, n) for n  
 sop(i)  
 }  
}  
  
public void logAtMost5(int n){  
 for(int i = 0; i < Math.min(5, n) ; i++){  
 //O(1) i.e. (5, n) for 5  
 sop(i)  
 }  
}

**Time Complexity of recursive algorithm #5**

findBiggestNumber(arr[], size) -----T(n)  
 static highest = Integer.MIN -----O(1)  
 if size equals-1 -----O(1)  
 return highest -----O(1)  
 else -----O(1)  
 if arr[size] > highest -----O(1)  
 update highest -----O(1)  
 return findBiggestNumber(arr, size-1) -----T(n-1)  
  
  
 T(n) = O(1)+T(n-1) -----equation 1  
 T(-1) = O(1) -----Base condition  
 T(n-1) = O(1) + (T(n-1)-1) -----equation 2  
 T(n-2) = O(1) + (T(n-2)-1) -----equation 3  
  
 T(n) = 1+T(n-1)  
 = 1+1+(T(n-1)-1) -----from equation 2  
 = 2+T(n-2)  
 = 2+1+T(n-2)-1) -----from equation 3  
 = 3+T(n-3)  
 = K+T(n-K) -----K as constant  
 = (n+1)+T(n-(n+1)) -----replace k with n+1  
 = n+1+T(-1) -----Base condition  
 = n+1+1  
 = O(n)

**Time Complexity of recursive algorithm #6**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 |

Given sorted array of 11. find time complexity for search no 110

BinarySearch(int findNumber, int arr[], start, end) -----T(n)  
 if(start equals end) -----O(1)  
 if(arr[start] equals findNumber) -----O(1)  
 return start -----O(1)  
 else -----O(1)  
 return error number not present -----O(1)  
  
 mid = findMid(arr[],start,end) -----O(1)  
 if mid > findNumber -----O(1)  
 BinarySearch(int findNumber, int arr[], start, mid) -----T(n/2)  
 else if mid < findNumber -----O(1)  
 BinarySearch(int findNumber, int arr[], mid, end) -----T(n/2)  
 else if mid equals findNumber -----O(1)  
 return mid -----O(1)  
  
 T(n) = T(n/2)+1 -----equation 1  
 T(1) = 1 -----base condition  
 T(n/2) = T(n/4) + 1 -----equation 2  
 T(n/4) = T(n/8) + 1 -----equation 3  
  
  
 T(n) = T(n/2)+1  
 = (T(n/4)+1)+1)  
 = T(n/4) + 2  
 = (T(n/8)+1)+2)  
 = T(n/8) + 3  
 = T(n/2k) + k  
 = T(1) + log n replace k with log n  
 = O(log n)

**Space Complexity:**

1. Most primitives (boolean, numbers, null) are constant space
2. String require O(n) space (where n is the String length)
3. References types are generally O(n), where n is the length (for arrays).

**Space Complexity algorithm #1**

public long sum(int[] arr){  
 long total = 0; ------1 assignment   
 for(int i = 0; i < arr.length; i++){ ------1 assignment  
 total = total + arr[i];  
 }  
 return total;  
}  
//so space complexity is O(1)

**Space Complexity algorithm #2**

public int[] doubleArray(int[] arr){  
 int[] newArr = new int[arr.length] ------n assignment   
 for(int i = 0; i < arr.length; i++){ ------1 assignment  
 newArr.push(2\*arr[i];  
 }  
 return total;  
}  
//so space complexity is O(n)

**Logarithms:**

log2(8) = 3 i.e. 23=8

log2(value) = exponent i.e. 2exponent=value

we will omit 2

log == log2

8 25

÷2 ÷2

4 12.5

÷2 ÷2

2 6.25

÷2 ÷2

1 3.125

÷2

1.5625

÷2

0.78125

log(8) = 3 log(25) = 4.65

The algorithm of a number roughly measures the number of times you can divide that number by 2before you get a value that’s less than or equals to 1.

**2. Physical Data Structure**

**2.1. Array Data Structure**

* Array can store data of specified data type
* It has contagious memory location
* Every cell of an array has unique index
* Index start with 0
* Size of array need to specified mandatory and cannot modified

**One-dimensional Array**

Having one row

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 |

**Two-dimensional Array**

m\*n array i.e. m rows and n column. 2\*5 example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 |
| 100 | 200 | 300 | 400 | 500 |

**Three-dimensional Array**

m\*n\*l i.e. depth \* row \* column e.g. rube cube.

**Q. How array represent in memory**

A. arr[5] = {10,20,30,40,50};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 |

arr[2][3] = {{10,20,30},{40,50,60}};

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 |

arr[2][3][3] = {{{10,20,30},{40,50,60},{70,80,90}},

{{100,110,120},{130,140,150},{160,170,180}}};

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 | 160 | 170 | 180 |

**2.1.1. One Dimensional Array**

**Time complexity: Declaring, Instantiating, Initializing 1D array.**

Declare  
 dataType []arr; -----O(1)  
 e.g. int []arr;  
  
Instantiation  
 arr = new dataType[size]; -----O(1)  
 e.g. arr = new int[5];  
  
Initialization  
 arr[0] = 10; -----O(1)  
 arr[1] = 20; -----O(1)  
 arr[2] = 30; -----O(1)  
 arr[3] = 40; -----O(1)  
 arr[4] = 50; -----O(1) i.e. O(n)  
  
Declare, Instantiation and Initialization  
int arr[] = {10,20,30,40,50}; -----O(1)

**Time complexity: Inserting a value in 1D array**

insert(arr, valueToBeInserted, location)  
 if arr[location] occupied -----O(1)  
 return error; -----O(1)  
 else -----O(1)  
 arr[location] = valueToBeInserted -----O(1) i.e. O(1)  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Traversing in 1D array**

traverseArray(arr)  
 loop i = 0 to arr size -----O(n)  
 print arr[i] -----O(1)  
  
Total time complexity: O(n)  
Space complexity: O(1)

**Time complexity: Accessing given cell 1D array**

accessingCell(arr, cellNo){  
 if(cellNo > arr.size) ----O(1)  
 return error cell no is higher than array size ----O(1)  
 else ----O(1)  
 print arr[cellNo] ----O(1)  
}  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Search a given value in an 1D array**

searchInAnArray(arr, value){  
 loop i=0 to arr size -----O(n)  
 if arr[i] equals value -----O(1)  
 return i; -----O(1)  
 else -----O(1)  
 return error value not present -----O(1)  
}  
  
Total time complexity: O(n)  
Space complexity: O(1)

**Time complexity: Deletion a value in an 1D array**

deletion(arr, location){  
 if arr[location] is occupied -----O(1)  
 arr[location] = Integer.MIN -----O(1)  
 else -----O(1)  
 return location is already empty -----O(1)  
}  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time Complexity of 1D array**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Time Complexity** | **Space Complexity** |
| Create an empty array | O(1) | O(n) |
| Inserting a value in an array | O(1) | O(1) |
| Traversing in a given array | O(n) | O(1) |
| Accessing a given cell | O(1) | O(1) |
| Searching a given value | O(n) | O(1) |
| Delete a given cell value | O(1) | O(1) |
| Delete a given value | O(n) | O(1) |

**Example**

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c1_array>

**2.1.2. Two Dimensional Array**

**Time complexity: Declaring, Instantiating, Initializing 2D array.**

Declare  
 dataType [][]arr; -----O(1)  
 e.g. int [][]arr;  
  
Instantiation  
 arr = new dataType[row][column]; -----O(1)  
 e.g. arr = new int[2][3];  
  
Initialization  
 arr[0][0] = 10; -----O(1)  
 arr[0][1] = 20; -----O(1)  
 arr[0][2] = 30; -----O(1)  
 arr[1][0] = 40; -----O(1)  
 arr[1][1] = 50; -----O(1)  
 arr[1][2] = 60; -----O(1) i.e. O(m\*n)  
  
Declare, Instantiation and Initialization  
int arr[][] = {{10,20,30},{40,50,60}} -----O(1)

**Time complexity: Inserting a value in 2D array**

insert(arr, valueToBeInserted, row, column)  
 if arr[row][column] occupied -----O(1)  
 return error location is already occupied; -----O(1)  
 else -----O(1)  
 arr[row][column] = valueToBeInserted -----O(1) i.e. O(1)  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Traversing in 2D array**

traverseArray(arr, row, column)  
 loop i = 0 to row -----O(m)  
 loop j = 0 to column -----O(n)  
 print arr[i][j] -----O(1)  
  
Total time complexity: O(mn)  
Space complexity: O(1)

**Time complexity: Accessing given cell 2D array**

accessingCell(arr, rowNo, columnNO){  
 return arr[row][column] -----O(1)  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Search a given value in an 2D array**

searchInAnArray(arr, value){  
 loop i=0 to row -----O(m)  
 loop j=0 to column -----O(n)  
 if arr[i][j] equals value -----O(1)  
 print(i, j); -----O(1)  
 else -----O(1)  
 return error value not present -----O(1)  
}  
  
Total time complexity: O(mn)  
Space complexity: O(1)

**Time complexity: Deletion a value in an 2D array**

deletion(arr, row, column){  
 if arr[row][column] is occupied -----O(1)  
 arr[row][column]= Integer.MIN -----O(1)  
 else -----O(1)  
 return location is already empty -----O(1)  
}  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time Complexity of 2D array**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Time Complexity** | **Space Complexity** |
| Create an empty array | O(1) | O(mn) |
| Inserting a value in an array | O(1) | O(1) |
| Traversing in a given array | O(mn) | O(1) |
| Accessing a given cell | O(1) | O(1) |
| Searching a given value | O(mn) | O(1) |
| Delete a given cell value | O(1) | O(1) |
| Delete a given value | O(mn) | O(1) |

**Example**

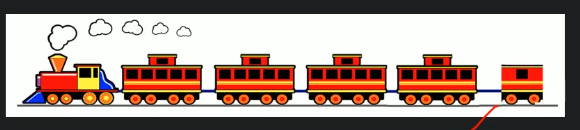
<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c1_array>

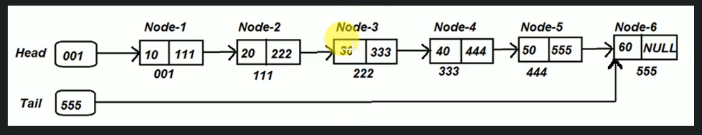
**2.2. Linked List**

A linked list is a liner data structure where each element is a separate object. Each element is also called as node. Node comprises two thing one is data other is reference of next node.

Linked list is variable in size

e.g Train





* There must be a head and tail present in Linked list
* Node contain data and next node reference
* It is liner means it travers sequentially i.e. if you want to travers from note 1 to 5 then you can direct go to node 5. You need to go via node 2,3,4.

Difference between Linked List and Array

* Separate Object vs One object
* Delete an Object vs Delete a value
* Variable size vs Fixed in size
* Traversing access (traverse node by node) vs Random access (direct access any cell)
* Connected via node with next pointer vs Index order
* Insertion and deletion is quick vs can be expensive

Components of Linked List

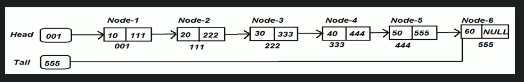
* Node: Contains data and reference of next node
* Head: Reference to first node of list
* Tail: Reference to last node of list

Types of Linked List

* Single Linked List

In Single linked list each node contains the data and reference of next node.

It is most basic LL which gives the flexibility to add/remove nodes at run time.

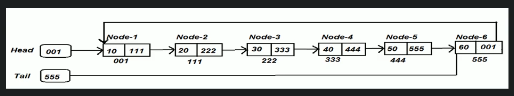


* Circular Single Linked List

In Circular linked list same as SLL. In case CSLL last node contain the address of node 1. Where in SLL last node contain the address null.

When we want to loop through the list indefinitely until the list exist.

e.g. LUDO game. Here 4 players is present. 1st p1 turn the p2 then p3 then p4. Again from p4 , p1 turn will come. IN SLL it is not possible.

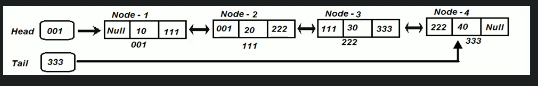


* Double Linked List

In DLL each node contains two reference, i.e. previous and next node reference.

When we want to in both directions depending upon our requirement.

e.g. Music player. Here we can able to access both next song and previous song.

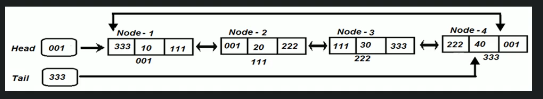


* Circular Double Linked List

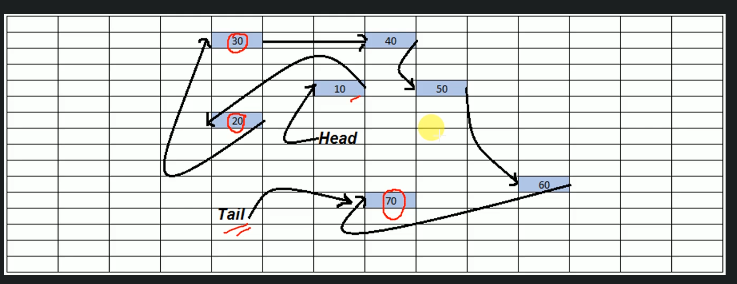
In CDLL same as DLL, the only change is that end node contains the reference of first node and vice versa.

Want to loop in through list in indefinitely in both forward and backward direction.

e.g. Alt+Tab button in windows OS.



How linked list store in memory



Common Operation of linked list

Creation of linked list

Insertion of linked list

Traverse of linked list

Searching in a linked list

Deletion of a node in linked list

Deletion of a linked list

**2.2.1 Single Linked List:**

**Time Complexity: Single Linked List: Creation**

CreateSingleLinkedList(nodeValue)  
 create a head and tail pointer and initialize with ZERO ---- O(1)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 node.next=null ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Single Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equlas 0)   
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 head=node -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = null -----O(1)  
 last.next = node -----O(1)  
 last = node -----O(1)  
 //to keep track of last node   
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 tempNode.next=node -----O(1)  
   
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Single Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print currentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Single Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = start to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

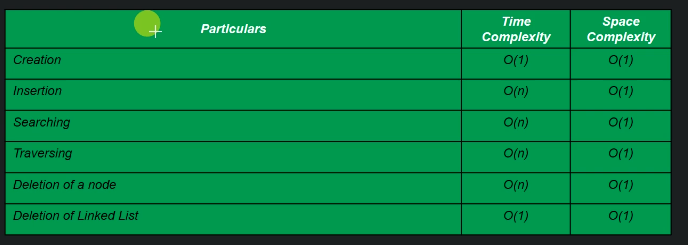
**Time Complexity: Single Linked List: Deletion of a node**

deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 if this is the only element   
 in list then update tail = null; ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head=tail=null; return ----O(1)  
 loop till 2nd last node(tmpNode) ----O(1)  
 tail = tmpNode; ----O(1)  
 tmpNode.next = null; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Single Linked List: Deletion a linked list**

deleteLikedList(head, tail)  
 head = null ----O(1)  
 tail = null ----O(1)  
  
Time complexity -----O(1)  
Space complexity -----O(1)

**Time Complexity of Single Linked List array**

****

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**2.2.2. Circular Single Linked List:**

**Time Complexity: Circular Single Linked List: Creation**

createCircularSingleLinkedList(nodeValue)  
 create a head and tail pointer and initialize with ZERO ---- O(1)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 node.next=node ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Circular Single Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equals 0)  
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 head = node -----O(1)  
 tail.next=head -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = head -----O(1)  
 tail.next = node -----O(1)  
 last = node -----O(1)  
 //to keep track of last node  
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 tempNode.next=node -----O(1)  
  
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Circular Single Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print currentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Circular Single Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = start to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

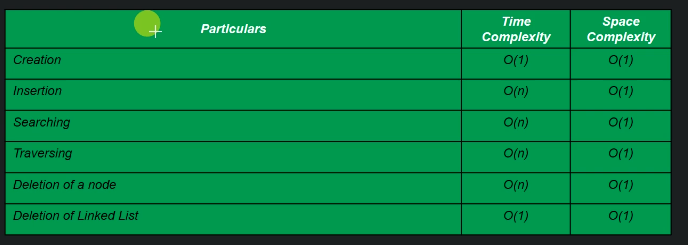
**Time Complexity: Circular Single Linked List: Deletion of a node**

deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 tail.next = head ----O(1)  
 if this is the only element in list then update tail = null;  
 ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head=tail=null ----O(1)  
 loop till 2nd last node(tmpNode) ----O(1)  
 tail = tmpNode; ----O(1)  
 tmpNode.next = head; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Circular Single Linked List: Deletion a linked list**

deleteLikedList(head, tail)  
 head = null ----O(1)  
 tail.next = null -----O(1)  
 tail = null ----O(1)  
  
Time complexity -----O(1)  
Space complexity -----O(1)

**Time Complexity of Circular Single Linked List array**

****

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**2.2.3. Double Linked List:**

**Time Complexity: Double Linked List: Creation**

CreateDoubleLinkedList(nodeValue)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
 node.next=node.previous=null ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Double Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equlas 0)  
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 node.previous = null -----O(1)  
 head.previous = node -----O(1)  
 head = node -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = null -----O(1)  
 node.previous = last -----O(1)  
 last.node = next -----O(1)  
 last = node -----O(1)  
 //to keep track of last node  
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 node.previous = tempNode -----O(1)  
 tempNode.next=node -----O(1)  
 node.next.previous = node ----O(1)  
  
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Double Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Reverse Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: tail to head -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = head to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

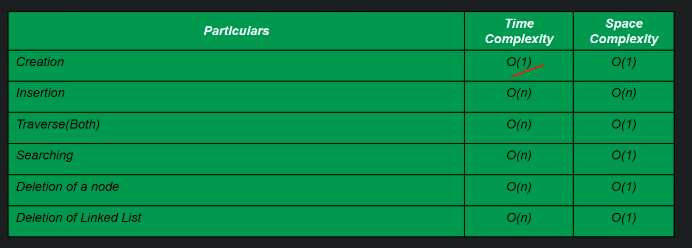
**Time Complexity: Double Linked List: Deletion of a node**

deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 head.previous = null; ----O(1)  
 if this is the only element in list  
 then update head=tail=null; ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head=tail=null ----O(1)  
 tail = tail.previous; ----O(1)  
 tail.next = null; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
 tmpNode.next.previous = tempNode ----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion a linked list**

deleteLikedList(head, tail)  
 loop: temp head to tail ---- O(n)  
 temp.previous = null; ---- O(1)  
 head = tail = null; ---- O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity of Circular Single Linked List array**



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**2.2.4. Circular Double Linked List:**

**Time Complexity: Circular Double Linked List: Creation**

CreateDoubleLinkedList(nodeValue)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
 node.next=node.previous=node ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Circular Double Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equlas 0)  
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 node.previous = tail -----O(1)   
 head.previous = node -----O(1)  
 head = node -----O(1)  
 tail.next = node -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = head -----O(1)  
 node.previous = tail -----O(1)  
 tail.next = node -----O(1)  
 tail = node -----O(1)  
 //to keep track of last node  
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 node.previous = tempNode -----O(1)  
 tempNode.next=node -----O(1)  
 node.next.previous = node ----O(1)  
  
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Circular Double Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Circular Double Linked List: Reverse Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: tail to head -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = start to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion of a node**

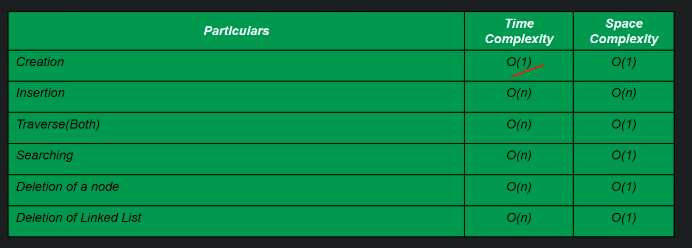
deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 head.previous = null; ----O(1)  
 tail.next = head; ----O(1)  
 if this is the only element in list  
 then update head.next=head.previous=head=tail=null; ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head.next=head.previous head=tail=null ----O(1)  
 tail = tail.previous; ----O(1)  
 tail.next = head; ----O(1)  
 head.previous=tail; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
 tmpNode.next.previous = tempNode ----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion a linked list**

deleteLikedList(head, tail)

tail.next = null ---- O(n)  
 loop: temp head to tail ---- O(n)  
 temp.previous = null; ---- O(1)  
 head = tail = null; ---- O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

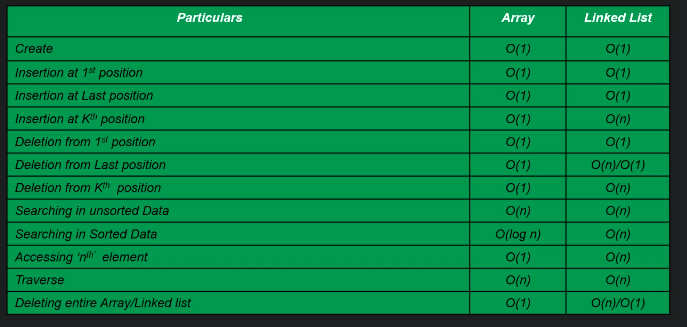
**Time Complexity of Circular Single Linked List array**



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**Different between Array and Linked List**



**Practical Use of linked list**

Alt tab button in windows and windows photo viewer

**3. Logical Data Structure**

**3.1. Stack**

* Insertion/deletion follows LIFO (Last In First Out)
* When we need to create last incoming data first algo
* E.g. browser back button.
* Common operations in stack are
  + createStack()
  + push()
  + pop()
  + peek()
  + isEmpty()
  + isFull()
  + deleteStack()
* We can implement stack in two ways using array and using linked list

**Implement Stack using Array**

Props: Easy to implement

Cons: Fixed in size

**Time Complexity: createStack()**

createStack(int size)  
 create black array od 'size' ----O(1)  
 Initialize variable 'topStack' to -1 ----O(1)  
   
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: push()**

pushOperation(value)  
 if stack is full ----O(1)  
 return error; ----O(1)  
 else ----O(1)  
 insert value at top of the array ----O(1)  
 update topStack++; ----O(1)  
   
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: pop()**

popOperation(value)  
 if stack is empty ----O(1)  
 return error; ----O(1)  
 else ----O(1)  
 print top of the stack ----O(1)  
 update topStack--; ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: peek()**

peekOperation(value)  
 if stack is empty ----O(1)  
 return error; ----O(1)  
 else ----O(1)  
 print top of the stack ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isEmpty()**

isEmptyOperation()  
 if topStack = -1 ----O(1)  
 return true; ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isFull()**

isFullOperation()  
 if topStack equals array.size ----O(1)  
 return true; ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: delete()**

deleteOperation()  
 arr = null ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Implement Stack using Linked List**

Props: Variable in size

Cons: Moderate in implementation

**Time Complexity: createStack()**

createStack()  
 create an object of SingleLinkedList class ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: push()**

pushOperation(value)  
 create a node ----O(1)  
 node.value = value; ----O(1)  
 node.next = head; ----O(1)  
 head = node; ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: pop()**

popOperation()  
 if isEmpty() ----O(1)  
 return error; ----O(1)  
 else  
 tmpNode = head ----O(1)  
 head = head.next ----O(1)  
 return tmpNOde.value ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: peek()**

peekOperation()  
 return header.value ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: delete()**

deleteOperation()  
 head = null ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**When to use and when to avoid**

LIFO principle, cannot easily corrupted, random access be not possible

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c1_stack>

**3.2. Queue**

* New addition member will join at end.
* Follows FIFO (First In First Out) principle.
* E.g. Billing counter
* Common operation in Queue
  + createQueue()
  + enQueue()
  + deQueue()
  + peekInQueue()
  + isEmpty()
  + isFull()
  + deleteQueue()
* We can implement queue in two ways
  + Array
    - Linear Queue
    - Circular Queue
  + Linked List
    - Linear Queue

**Implement Linear Queue using Array**

**Time Complexity: createQueue()**

createQueue(size)  
 create a blank array of 'size' ----O(1)  
 initialize topOfQueue, beginningOfQueue ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(n)

**Time Complexity: enQueue()**

enQueue(value)  
 if queue is full ----O(1)  
 return error message ----O(1)  
 else ----O(1)  
 arr[topOfQueue+1] = value ----O(1)  
 topOfQueue++ ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: deQueue()**

deQueue()  
 if queue is empty ----O(1)  
 return error message ----O(1)  
 else ----O(1)  
 print arr[beginningOfQueue] ----O(1)  
 beginningOfQueue++ ----O(1)  
 if beginningOfQueue > endOfQueue ----O(1)  
 beginningOfQueue=-1  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: peek()**

peek()  
 if queue is empty ----O(1)  
 return error message ----O(1)  
 else ----O(1)  
 print arr[beginningOfQueue] ----O(1)  
   
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isEmptyQueue()**

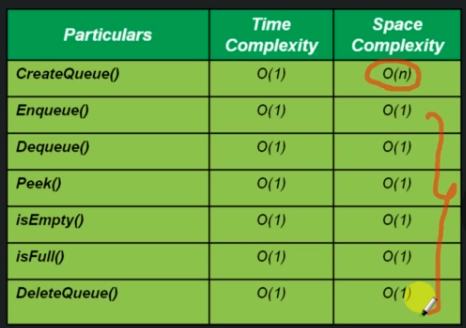
isEmptyQueue()  
 if beginningOfQueue == -1 ----O(1)  
 return true ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isFullQueue()**

isFullQueue()  
 if topOfQueue == arr.length-1 ----O(1)  
 return true ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: deleteQueue()**

deleteQueue()  
 arr = null ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c2_queue>

**Implement Circular Queue using Array**

**Why learn Circular Queue?**

deQueue cause some blank cell in linear queue (Array implementation)

**Time Complexity: createQueue()**

createQueue(size)  
 create a blank array of 'size' ----O(1)  
 initialize topOfQueue, beginningOfQueue ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(n)

**Time Complexity: enQueue()**

enQueue(value)   
 if (isQueueFull) ----O(1)  
 return error message ----O(1)  
 else ----O(1)  
 if(topOfQueue+1 == size) ----O(1)  
 topOfQueue = 0; ----O(1)  
 else ----O(1)  
 topOfQueue ++; ----O(1)  
 arr[topOfQueue] = value; ----O(1)  
   
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: deQueue()**

deQueue()  
 if (isQueueEmpty) ----O(1)  
 return error message ----O(1)  
 else ----O(1)  
 print(arr[beginningOfQueue]) ----O(1)  
 if(beginningOfQueue == topOfQueue) ----O(1)  
 beginningOfQueue = topOfQueue = -1 ----O(1)  
 else ----O(1)  
 beginningOfQueue + 1 = size ----O(1)  
 beginningOfQueue = 0; ----O(1)  
 else ----O(1)  
 beginningOfQueue++ ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: peek()**

peek()  
 if queue is empty ----O(1)  
 return error message ----O(1)  
 else ----O(1)  
 print arr[beginningOfQueue] ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isEmptyQueue()**

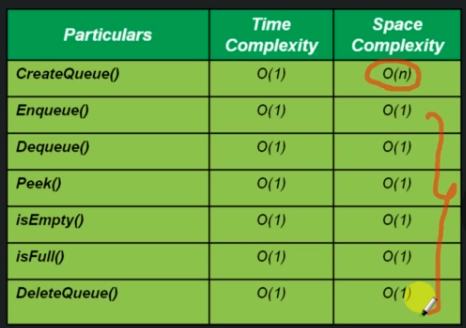
isEmptyQueue()  
 if topOfQueue == -1 ----O(1)  
 return true ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isFullQueue()**

isFullQueue()  
 if topOfQueue+1 == beginningOfQueue ----O(1)  
 return true ----O(1)  
 else if (beginningOfQueue == 0 && topOfQueue+1 == size) ----O(1)  
 return true ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: deleteQueue()**

deleteQueue()  
 arr = null ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c2_queue>

**Implement Linear Queue using Linked List**

**Time Complexity: createQueue()**

createQueue()  
 create a blank SingleLinkedList ----O(1)  
   
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: enQueue()**

enQueue(value)  
 createNode ----O(1)  
 node.value=value ----O(1)  
 node.next=null ----O(1)  
 if tail = null ----O(1)  
 tail = node ----O(1)  
 else ----O(1)  
 tail.next=node; ----O(1)  
 tail = node; ----O(1)  
   
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: deQueue()**

deQueue(value)  
 if head = null ----O(1)  
 error queue is empty ----O(1)  
 else ----O(1)  
 tmpNode = header ----O(1)  
 header = tmpNode.next ----O(1)  
 return tmpNode.value ----O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: peek()**

peek()  
 if head = null ----O(1)  
 error queue is empty ----O(1)  
 else ----O(1)  
 head.value ----O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: isEmpty()**

isEmpty()  
 if head = null ----O(1)  
 return true ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: delete()**

isEmpty()  
 if head = tail = null ----O(1)  
   
Time Complexity: O(1)  
Space Complexity: O(1)



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c2_queue>

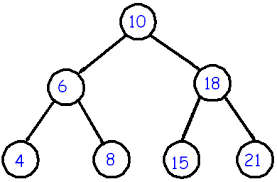
**Time and space complexity between Array and Linked List**

****

**When to use and when to avoid**

FIFO principle, not corrupted, random access not possible

**3.3. Tree**

****

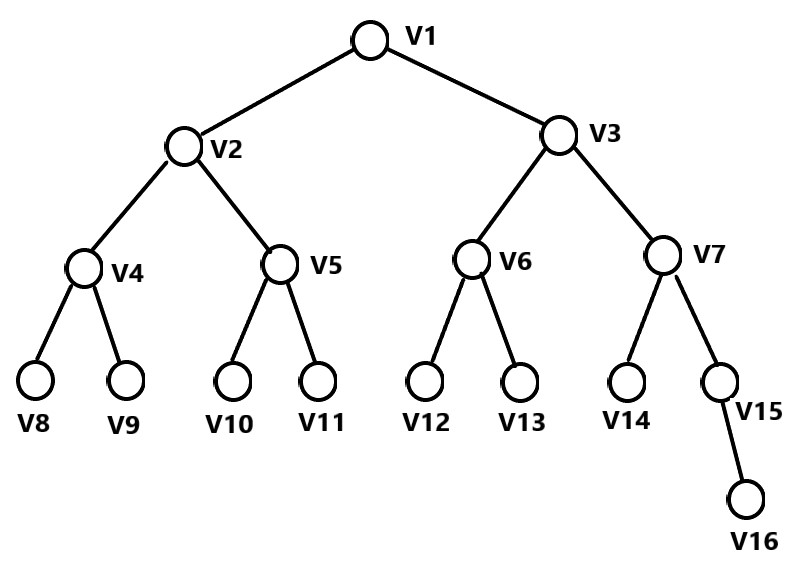
**Properties of Tree**

* Used to represent data in hierarchy form
* Every Node (Ideally) having two node (Data and Reference)
* It has root node and 2 disjoint tree called left subtree and right subtree

**Why we learn Tree**

Try to archive time complexity for insertion, deletion and searching compare to Array and List.

**Tree terminology**



Root: Node with no parents [V1]

Edge: Link between parents and child [V1 🡨🡪 V2, V5 🡨🡪 V11, …]

leaf: Mode with no child [V8, V9, V16, …]

Sibling: Child of same parent [V2 and V3, V4 and V5, …]

Ancestor: Means parent, grand-parent, great-grand-parent, and so one for a given node

[Ancestor of V9 is V4(parent), V2(grand-parent), V1(great-grand-parent),

Ancestor of V6 is V3(parent), V1(grand-parent)]

Depth of Node: Length of the path from root to that note. [Depth of V12 is 3 i.e. V1 🡨🡪 V3, V3 🡨🡪 V6 and V6 🡨🡪 V12]

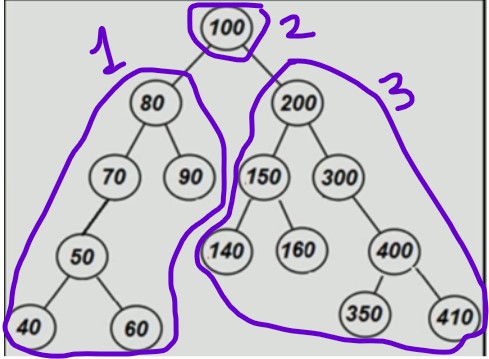
Height of Node: Length of the path from that node to deepest node [Height of V3 is 3 i.e. V3 to V16(as V16 is deepest node]

Height of tree: Height of root node is 4 i.e. V1 to V16

Depth of tree: Depth of root node i.e. 0. It is always 0.

Predecessor: Predecessor of a node is the immediate previous node in inorder traversal of the Binary tree.

Successor: Successor of a node is the immediate next node in inorder traversal of Binary tree.



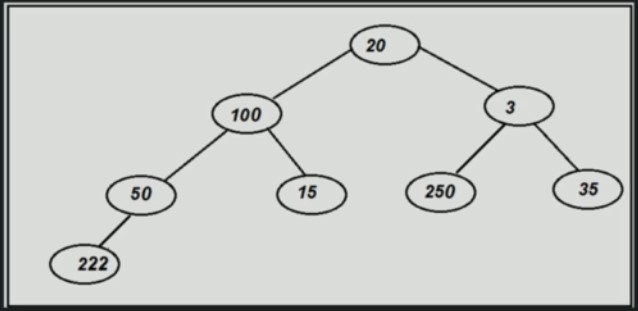
Inorder traversal : 40, 50, 60, 70, 80, 90, 100, 140, 150,160, 200, 300, 350, 400, 410 1 2 3

Predecessor of 100 is 90, and of 160 is 150 and so on

Successor of 100 is 140, and of 160 is 200 and so on

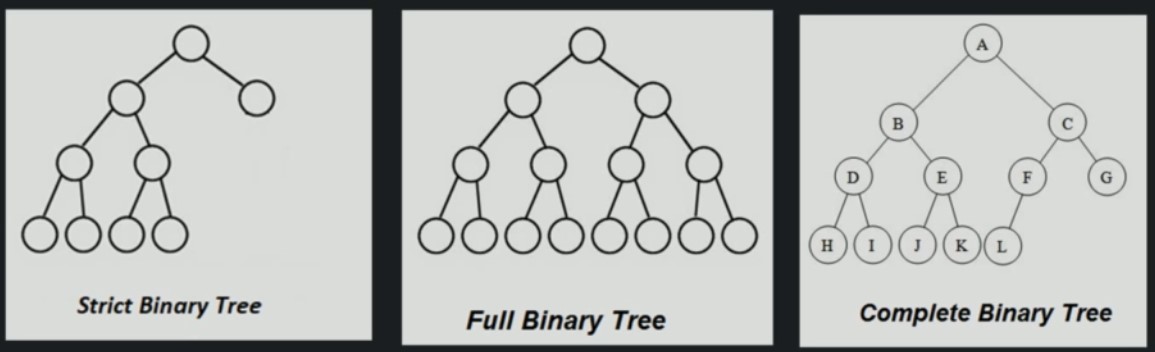
**3.3.1. Binary Tree**

* A tree is called as binary tree if each node has zero, one or two chield.
* It is a family of data structure (BST, Heap tree, AVL, Red-Black, Syntax Tree, Huffman Coding Tree, etc …)



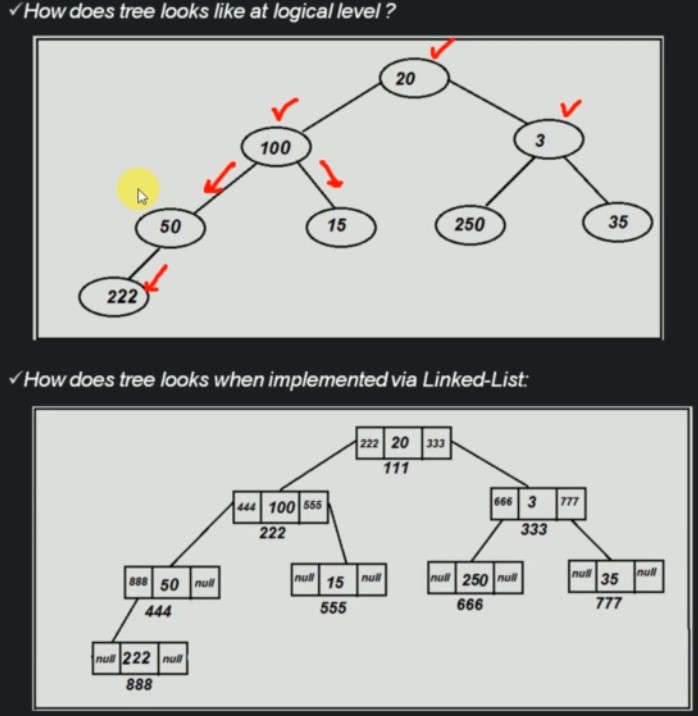
Types of Binary Tree

* Strick Binary Tree : If each node has either two children or non.
* Full Binary Tree : If each non leaf node has 2 children and all leaf nodes are at same level.
* Complete Binary Tree : If all levels are completely filled except possibly the last level and the last level has all keys as left as possible.

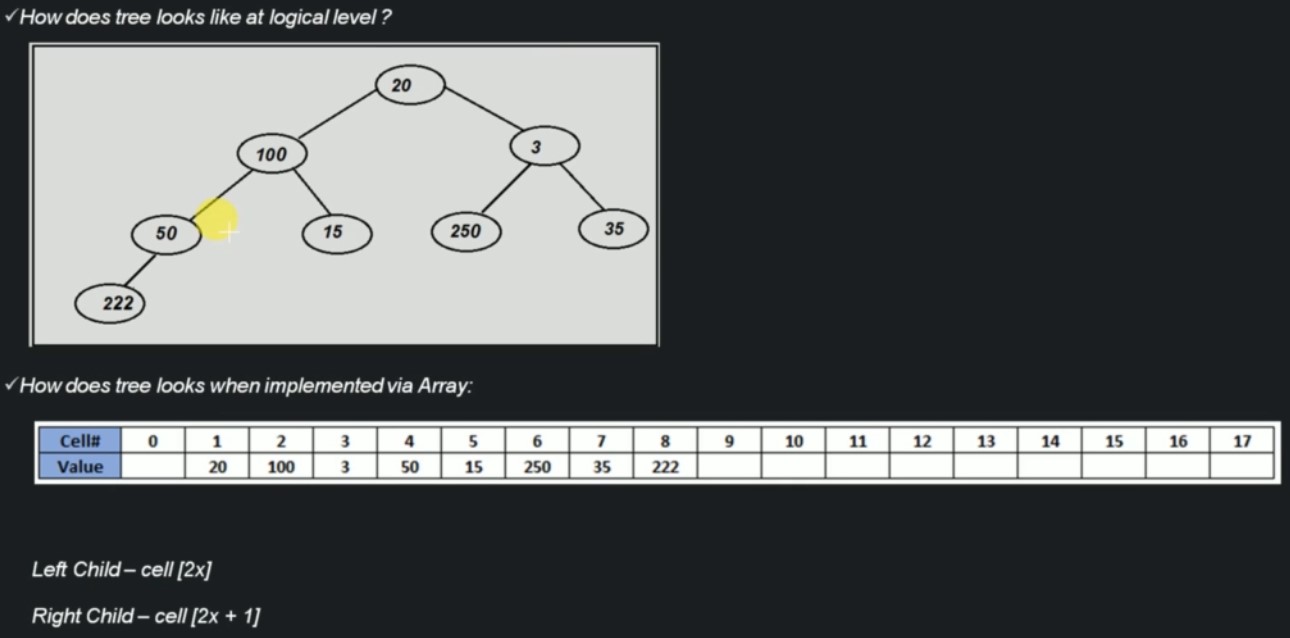


Tree Representation:

Using Linked List:



Using Array



**Binary Tree Using Linked List**

**Time Complexity: Creation of Binary Tree**

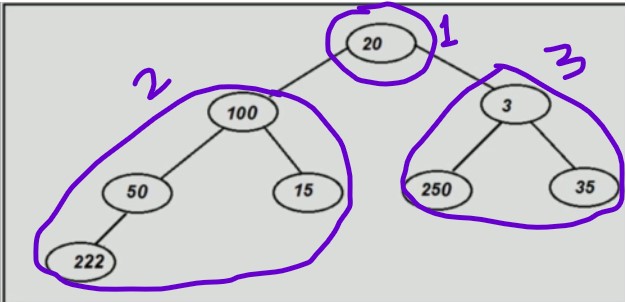
createBinaryTree  
 create an object of Binary tree class ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Traversing all node of Binary Tree**

* Depth First Search
  + PreOrder Traversal (root-left subtree-right subtree)
  + InOrder Traversal (left subtree-root-right subtree)
  + PostOrder Traversal (left subtree-right subtree-root)
* Breadth First Search
  + LevelOrder Traversal (Level order traversal)

**PreOrder Traversal:**

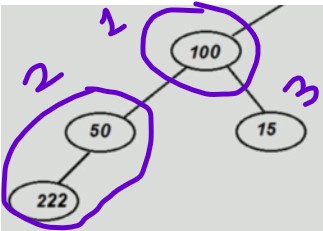
root left\_subtree right\_subtree



20 left\_subtree right\_subtree

20 pre\_order right\_subtree

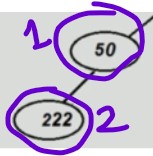
20 (r-ls-rs) right\_subtree



20 (100 left\_subtree right\_subtree) right\_subtree

20 (100 pre\_order right\_subtree) right\_subtree

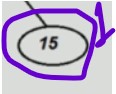
20 (100 (r-ls-rs) right\_subtree) right\_subtree



20 (100 (50 222) right\_subtree) right\_subtree

20 100 50 222 pre\_order) right\_subtree

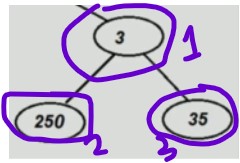
20 100 50 222 (r-ls-rs)) right\_subtree



20 100 50 222 15 right\_subtree

20 100 50 222 15 pre\_order

20 100 50 222 15 (r-ls-rs)



20 100 50 222 15 (3 ls rs)



20 100 50 222 15 (3 250 rs)



20 100 50 222 15 (3 250 35)

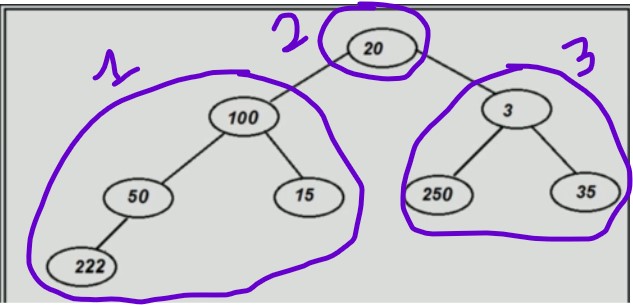
Final: 20 100 50 222 15 3 250 35

**Time Complexity: PreOrder Traversal**

preOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 print root ---O(1)  
 preOrderTraversal(root.left) ---T(n/2)  
 preOrderTraversal(root.right) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**InOrder Traversal:**

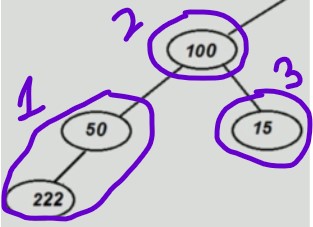
left\_subtree root right\_subtree



left\_subtree 20 right\_subtree

InOrder 20 right\_subtree

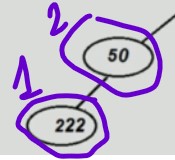
ls-r-rs 20 right\_subtree



ls-100-rs 20 right\_subtree

InOrder-100-rs 20 right\_subtree

(ls-r-rs)-100-rs 20 right\_subtree



222 50 100 rs 20 right\_subtree

222 50 100 InOrder 20 right\_subtree

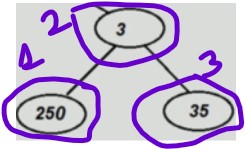
222 50 100 (ls-r-rs) 20 right\_subtree



222 50 100 15 20 right\_subtree

222 50 100 15 20 InOrder

222 50 100 15 20 ls-r-rs



222 50 100 15 20 ls-3-rs

222 50 100 15 20 InOrder-3-rs



222 50 100 15 20 250-3-rs

222 50 100 15 20 250-3-InOrder



222 50 100 15 20 250-3-35

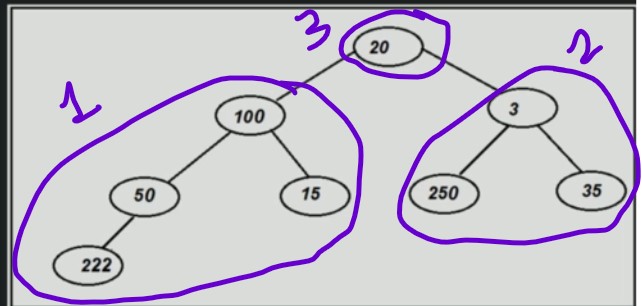
Final: 222 50 100 15 20 250 3 35

**Time Complexity: InOrder Traversal**

inOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 inOrderTraversal(root.left) ---T(n/2)  
 print root ---O(1)  
 inOrderTraversal(root.right) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**PostOrder Traversal:**

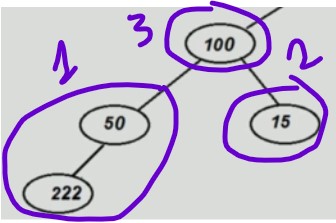
left\_subtree right\_subtree root



left\_subtree right\_subtree 20

postOrder right\_subtree 20

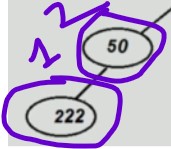
ls-rs-r right\_subtree 20



ls-rs-100 right\_subtree 20

postOrder-rs-100 right\_subtree 20

(ls-rs-r)-rs-100 right\_subtree 20



(222 50)-rs-100 right\_subtree 20

222 50-postOrder-100 right\_subtree 20

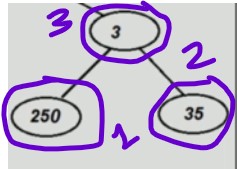
222 50-(ls-rs-r)-100 right\_subtree 20



222 50 15 100 right\_subtree 20

222 50 15 100 postOrder 20

222 50 15 100 ls-rs-r 20



222 50 15 100 ls-rs-3 20

222 50 15 100 postOrder-rs-3 20

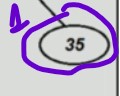
222 50 15 100 (ls-rs-r)-rs-3 20



222 50 15 100 (250)-rs-3 20

222 50 15 100 250-postOrder-3 20

222 50 15 100 250-(ls-rs-r)-3 20



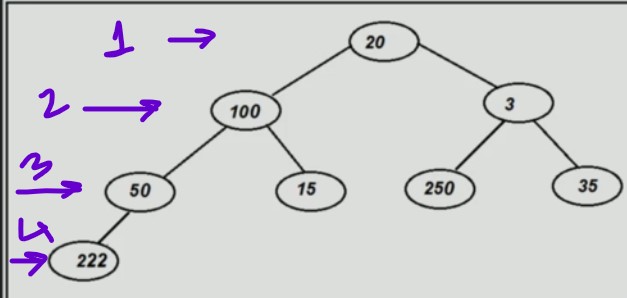
222 50 15 100 250-(35)-3 20

Final: 222 50 15 100 250 35 3 20

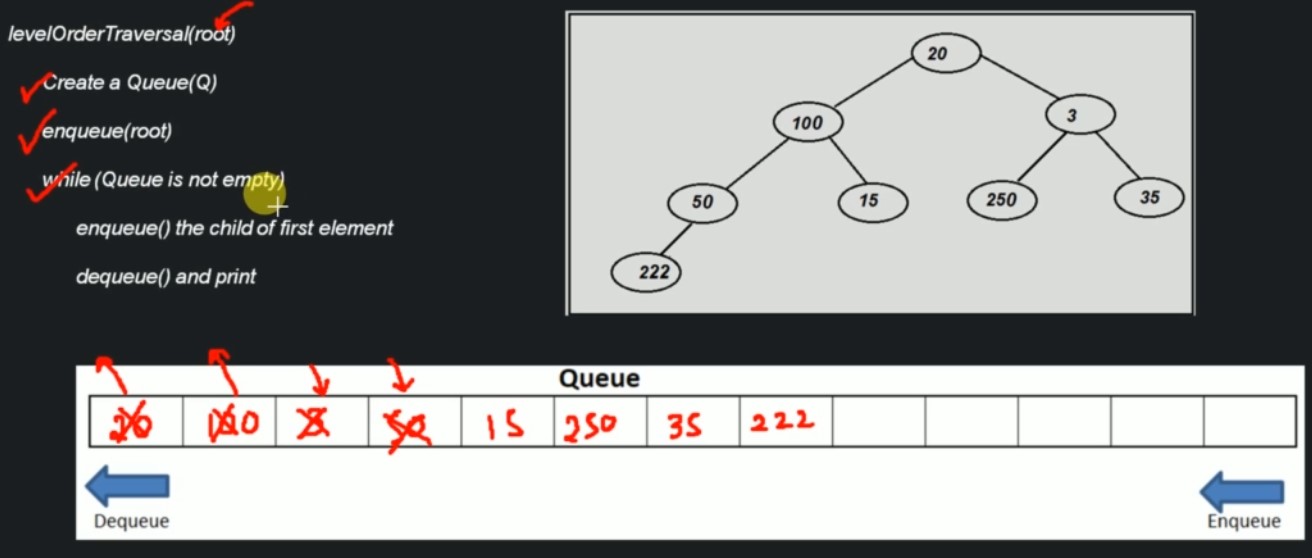
**Time Complexity: PostOrder Traversal**

postOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 inOrderTraversal(root.left) ---T(n/2)  
 inOrderTraversal(root.right) ---T(n/2)  
 print root ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**LevelOrder Traversal:**



Final: 20 100 3 50 15 250 35 222



**Time Complexity: LevelOrder Traversal**

levelOrderTraversal(root)   
 create a Queue(Q) ---O(1)  
 enQueue(root) ---O(1)  
 while(Queue is empty) ---O(n)  
 enQueue() the child of first element---O(1)  
 deQueue() and print ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: Searching a node in Binary tree**

searchForGivenValue(value)   
 if(root == null) ---O(1)  
 return error message ---O(1)  
 else ---O(1)  
 do a levelOrderTraversal ---O(n)  
 if value found ---O(1)  
 return success message ---O(1)  
 else ---O(1)  
 return unsuccessful message ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: Inserting Node in Binary tree**

insertNodeBinaryTree()  
 if root is blank ---O(1)  
 insert node t root ---O(1)  
 else ---O(1)  
 do a level order traversal and find first blank space ---O(n)  
 insert in that blank space ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: Delete a value in Binary tree**

deleteValueFromBinaryTree()  
 search for the node to be deleted ---O(n)  
 find deepest node in the binary tree ---O(n)  
 copy deepest node's data in current node---O(1)  
 delete deepest node ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: Delete a Binary tree**

deleteBinaryTree()  
 root = null ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)



[011\_data\_structure/src/s3\_c3\_tree/binarytree at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c3_tree/binarytree)

**Binary Tree Using Array**

**Time Complexity: creation of a Binary tree**

createBinaryTree()  
 create a blank array of 'size' ---O(1)  
 update last index to 0 ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(n)

**Time Complexity: Insert a node in Binary tree**

insertValueInBinaryTree()  
 if tree is full ---O(1)  
 return error message ---O(1)  
 else ---O(1)  
 insert value in first unused call of array ---O(1)  
 updateLastUsedIndex ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: Search a node in Binary tree**

searchValueInBinaryTree()  
 traverse the entire array from 1 to last usedIndex ---O(n)  
 if value was found ---O(1)  
 return success message ---O(1)  
 return error message ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(1)

**Traversing all node of Binary Tree**

* Depth First Search
  + PreOrder Traversal (root-left subtree-right subtree)
  + InOrder Traversal (left subtree-root-right subtree)
  + PostOrder Traversal (left subtree-right subtree-root)
* Breadth First Search
  + LevelOrder Traversal (Level order traversal)

**Time Complexity: PreOrder Traversal**

preOrderTraversal(index) ---T(n)  
 if index>lastUsedIndex ---O(1)  
 return ---O(1)  
 else ---O(1)  
 print current index.vale ---T(1)  
 preOrderTraversal(index\*2) ---O(n/2)  
 preOrderTraversal(index\*2+1) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(1)

**Time Complexity: InOrder Traversal**

inOrderTraversal(index) ---T(n)  
 if index>lastUsedIndex ---O(1)  
 return ---O(1)  
 else ---O(1)  
 inOrderTraversal(index\*2) ---T(n/2)  
 print current index.vale ---O(1)  
 inOrderTraversal(index\*2+1) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(1)

**Time Complexity: PostOrder Traversal**

postOrderTraversal(index) ---T(n)  
 if index>lastUsedIndex ---O(1)  
 return ---O(1)  
 else ---O(1)  
 postOrderTraversal(index\*2) ---O(n/2)  
 postOrderTraversal(index\*2+1) ---T(n/2)  
 print current index.vale ---T(1)  
  
Time Complexity: O(n)  
Space Complexity: O(1)

**Time Complexity: LevelOrder Traversal**

levelOrderTraversal(index)  
 loop: 1 to lastUsedIndex ---O(n)  
 print current index.value ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(1)

**Time Complexity: Delete a value from Binary Tree**

deleteNodeFromBinaryTree()  
 search for desired value in Binary tree ---O(n)  
 if value found ---O(1)  
 replace the call with last cell and update lastUsedIndex---O(1)  
 return error message ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(1)

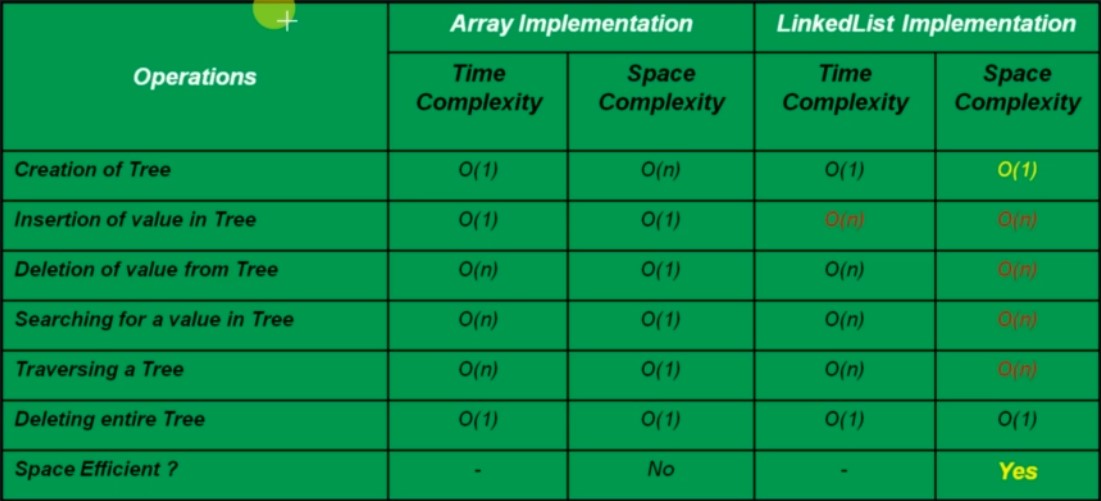
**Time Complexity: Delete a Binary Tree**

deleteBinaryTree()  
 set array as null ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)



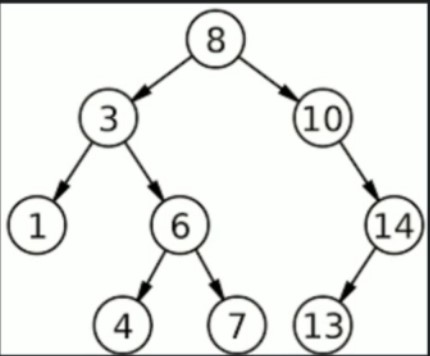
[011\_data\_structure/src/s3\_c3\_tree/binarytree at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c3_tree/binarytree)

Binary tree using Linked List vs Array



**3.3.2. Binary Search Tree**

* The left sub-tree of a node has a key less than or equal to its parent node’s key.
* The right sub-tree of a node has a key greater than to its parent node’s key.



Ex:

Take 8 as parent node. So all the left-sub tree values i.e. 1,3,4,6,7 has less than 8 or equal to 8. Same all the right-sub tree value i.e. 10, 14,13 has grater than 8.

**Time Complexity: creation of a Binary Search tree**

createBinarySearchTree  
 Initialize root with null ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: Search a node in Binary Search tree**

searchBinarySearchTree(root, value)  
 if (root is null) ---O(1)  
 return null ---O(1)  
 else if ( root == value) ---O(1)  
 return root ---O(1)  
 else if ( value < root) ---O(1)  
 searchBinarySearchTree(root.left, value) ---T(n/2)  
 else if ( value > root) ---O(1)  
 searchBinarySearchTree(root.right, value) ---T(n/2)  
  
Time Complexity: O(log n)  
Space Complexity: O(log n) because of recursive

**Traversing all node of Binary Search Tree**

* Depth First Search
  + PreOrder Traversal (root-left subtree-right subtree)
  + InOrder Traversal (left subtree-root-right subtree)
  + PostOrder Traversal (left subtree-right subtree-root)
* Breadth First Search
  + LevelOrder Traversal (Level order traversal)

**Time Complexity: PreOrder Traversal**

preOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 print root ---O(1)  
 preOrderTraversal(root.left) ---T(n/2)  
 preOrderTraversal(root.right) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: InOrder Traversal**

inOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 inOrderTraversal(root.left) ---T(n/2)  
 print root ---O(1)  
 inOrderTraversal(root.right) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: PostOrder Traversal**

postOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 inOrderTraversal(root.left) ---T(n/2)  
 inOrderTraversal(root.right) ---T(n/2)  
 print root ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: LevelOrder Traversal**

levelOrderTraversal(root)   
 create a Queue(Q) ---O(1)  
 enQueue(root) ---O(1)  
 while(Queue is empty) ---O(n)

deQueue() and print ---O(1)  
 enQueue() the child of dequed element---O(1)  
   
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: Insert a node in Binary Search tree**

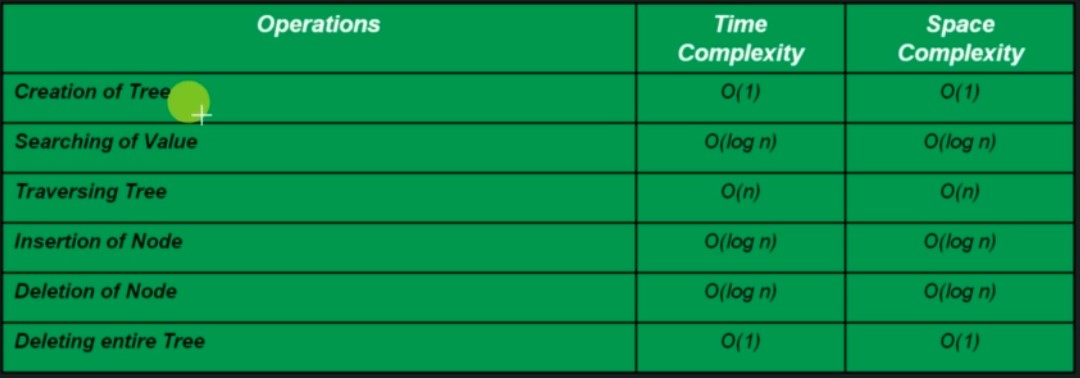
insertInBinarySearchTree(cNode, value) //cNode as currentNode ---T(n)  
 if (cNode is null) ---O(1)  
 create a node, insert value in it ---O(1)  
 else if (value < cNode's value) ---O(1)  
 cNode.left=insertInBinarySearchTree(cNode.left, value) ---T(n/2)  
 else ---O(1)  
 cNode.right=insertInBinarySearchTree(cNode.right, value)---T(n/2)  
 return cNode; ---O(1)  
  
Time Complexity: O(log n)  
Space Complexity: O(log n)

**Time Complexity: Delete a node in Binary Search tree**

deleteNodeBinarySearchTree(root, value) ---T(n)  
 if(root=null) ---O(1)  
 return null ---O(1)  
 else if (value < root.value) ---O(1)  
 then root.left=deleteNodeBinarySearchTree(root.left, value)---T(n/2)  
 else if (value > root.value) ---O(1)  
 then root.right=deleteNodeBinarySearchTree(root.right, value)---T(n/2)  
 else //if current node is to be deleted ---O(1)  
 if root have both children, then find minimum element from right sub-tree---O(log n)  
 replace current node with minimum node from right subtree and delete minimum node from right---O(1)  
 else if nodeToBeDeleted has only left child ---O(1)  
 then root = root.left ---O(1)  
 else if nodeToBeDeleted has only right child ---O(1)  
 then root=root.right ---O(1)  
 else //if nodeToBeDeleted do not have child ---O(1)  
 root=null ---O(1)  
 return root ---O(1)  
  
Time Complexity: O(log n)  
Space Complexity: O(log n)

**Time Complexity: Delete a Binary Search tree**

deleteBinarySearchTree()  
 root=null ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)



[011\_data\_structure/src/s3\_c3\_tree/binarysearchtree at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c3_tree/binarysearchtree)

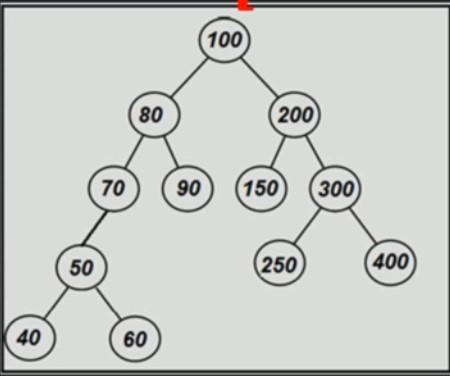
**3.3.3. AVL Tree**

**Why AVL tree?**

* Depending on incoming data, A Binary Search Tree can get skewed and hence its performance starts going down. Instead of O(log n) for insertion/searching/deleting it can go up to O(n).
* AVL tree attempts to solve this problem of ‘skewing’ by introducing concept called ‘Rotation’.

**What is AVL Tree?**

* AVL tree is a balanced ‘Binary Search Tree’ where the height of immediate subtrees of any node differs by at most one (also called balanced factor).
* If at any time hights differ by more than one, rebalancing is done to restore this property (called rotation).
* Empty height always considered -1

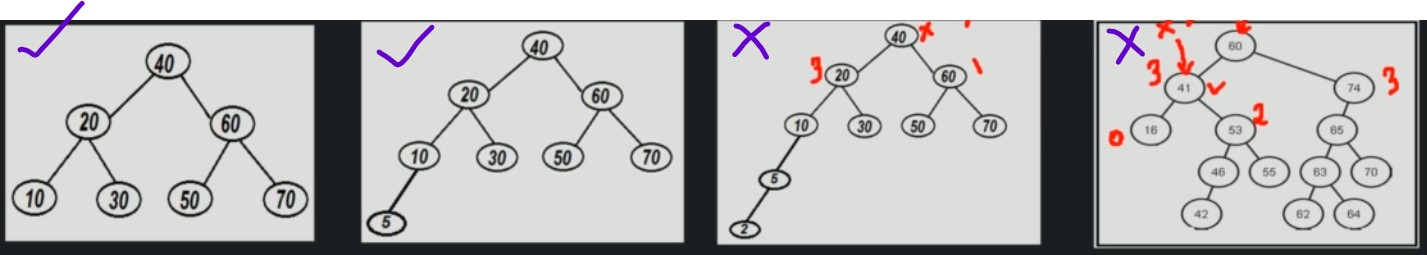


At node 100: Height of left subtree is 3 and Height of right subtree is 2 so the height difference is 3-2=1(at most 1: true)

At node 80: Height of left subtree is 2 and Height of right subtree is 0 so the height difference is 2-0=2(at most 1: false)

At node 70: Height of left subtree is 1 and Height of right subtree is -1(empty height always considered -1) so the height difference is 1-(-1)=2(at most 1: false)

So at some node the hight difference is more then 1. So, the above tree is not an AVL tree.



**Time Complexity: creation of a AVL tree**

createAVLTree  
 Initialize root with null ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: Search a node AVL tree**

searchAVLTree(root, value)  
 if (root is null) ---O(1)  
 return null ---O(1)  
 else if ( root == value) ---O(1)  
 return root ---O(1)  
 else if ( value < root) ---O(1)  
 searchAVLTree (root.left, value) ---T(n/2)  
 else if ( value > root) ---O(1)  
 searchAVLTree (root.right, value) ---T(n/2)

Time Complexity: O(log n)  
Space Complexity: O(log n) because of recursive

**Traversing all node of AVL Tree**

* Depth First Search
  + PreOrder Traversal (root-left subtree-right subtree)
  + InOrder Traversal (left subtree-root-right subtree)
  + PostOrder Traversal (left subtree-right subtree-root)
* Breadth First Search
  + LevelOrder Traversal (Level order traversal)

**Time Complexity: PreOrder Traversal**

preOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 print root ---O(1)  
 preOrderTraversal(root.left) ---T(n/2)  
 preOrderTraversal(root.right) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: InOrder Traversal**

inOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 inOrderTraversal(root.left) ---T(n/2)  
 print root ---O(1)  
 inOrderTraversal(root.right) ---T(n/2)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: PostOrder Traversal**

postOrderTraversal(root) ---T(n)  
 if(root is null) ---O(1)  
 return error message ---O(1)  
 else  
 inOrderTraversal(root.left) ---T(n/2)  
 inOrderTraversal(root.right) ---T(n/2)  
 print root ---O(1)  
  
Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: LevelOrder Traversal**

levelOrderTraversal(root)   
 create a Queue(Q) ---O(1)  
 enQueue(root) ---O(1)  
 while(Queue is empty) ---O(n)

deQueue() and print ---O(1)  
 enQueue() the child of dequed element---O(1)

Time Complexity: O(n)  
Space Complexity: O(n)

**Time Complexity: Insert a node in AVL tree**

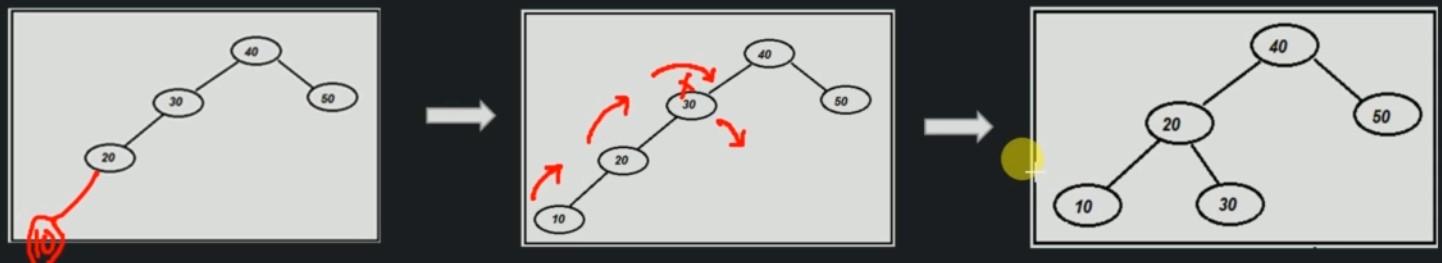
Case 1: When rotation not required

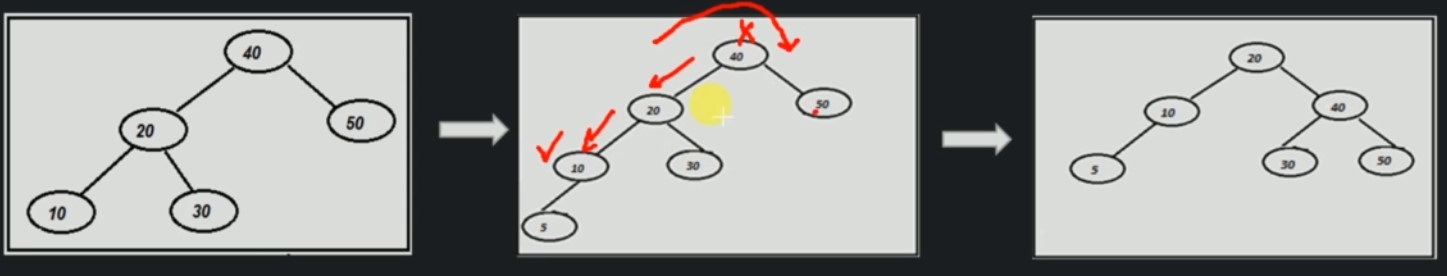
Case 2: When rotation required (LL, LR, RR, RL)

Left-Left Condition:

What is Left-Left condition?

* Left-Left node from currentNode causing disbalance
* In this case do a Right Rotation





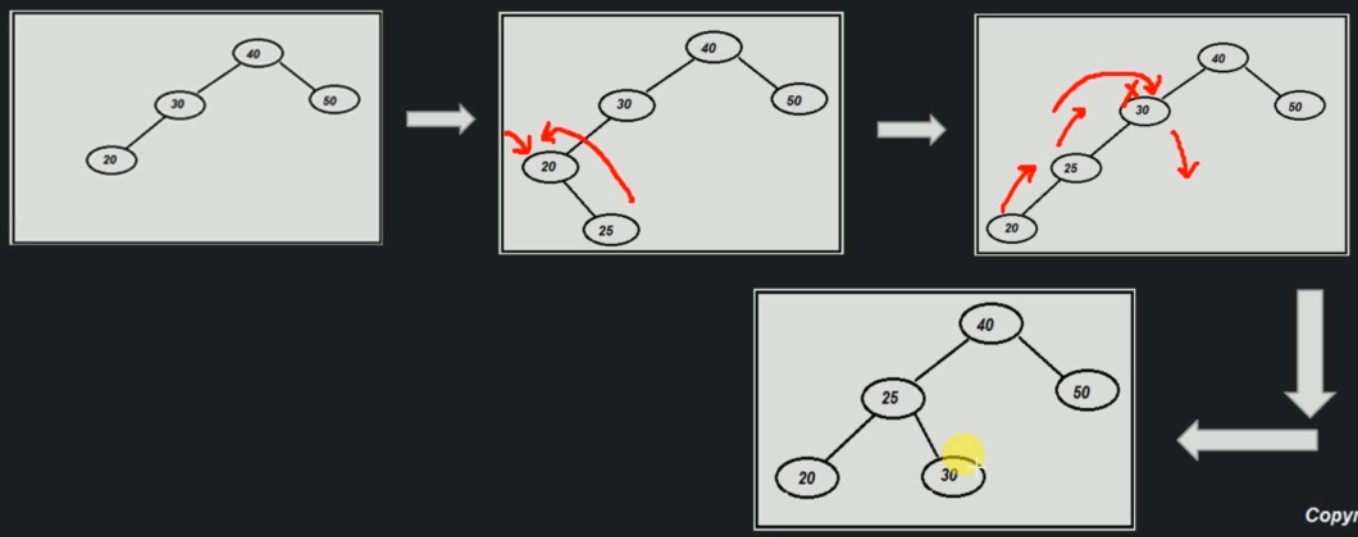
**Time Complexity: Left-Left condition**

rightRotate(cDisbandedNode) cDisbandedNode as currentDisbandedNode  
 newRoot=cDisbandedNode.Left ---O(1)  
 cDisbandedNode.Left=cDisbandedNode.Left.Right ---O(1)  
 newRoot.Right=cDisbandedNode ---O(1)  
 cDisbandedNode.Height=calculatedHeight(cDisbandedNode) ---O(1)  
 newRoot.Height=calculatedHeight(newRoot) ---O(1)  
 return newRoot ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

Left-Right Condition:

What is Left-Right condition?

* Left-Right node from currentNode causing disbalance
* In this case do a Left Rotation followed by Right Rotation



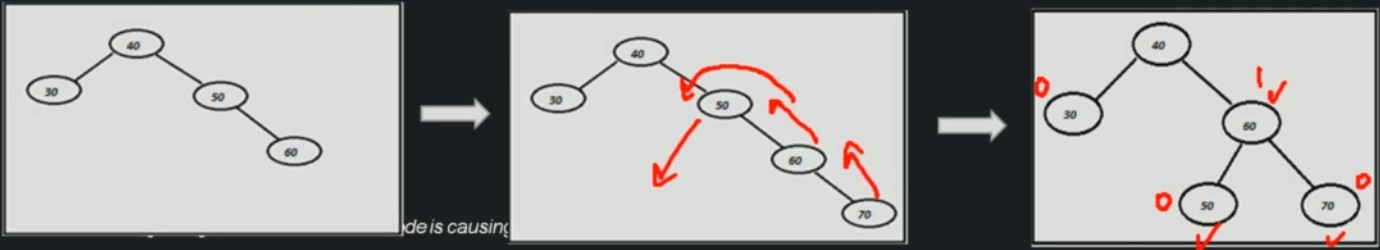
**Time Complexity: Left-Right condition**

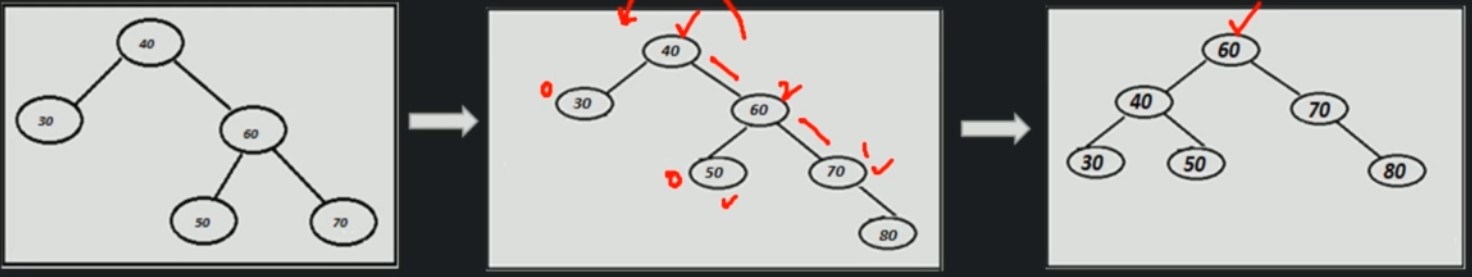
leftRotate(cdNode) cdNode as currentDisbandedNode  
 newRoot=cdNode'sLeftChild.Right ---O(1)  
 cdNode'sLeftChild.Right=cdNode'sLeftChild.Right.Left ---O(1)  
 newRoot.Left=cdNode'sLeftChild ---O(1)  
 cdNode'sLeftChild.Height=calculatedHeight(cdNode'sLeftChild)---O(1)  
 newRoot.Height=calculatedHeight(newRoot) ---O(1)  
 return newRoot ---O(1)  
  
rightRotate(currentNode) ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

Right-Right Condition:

What is Right-Right condition?

* Right-Right node from currentNode causing disbalance
* In this case do a Left Rotation





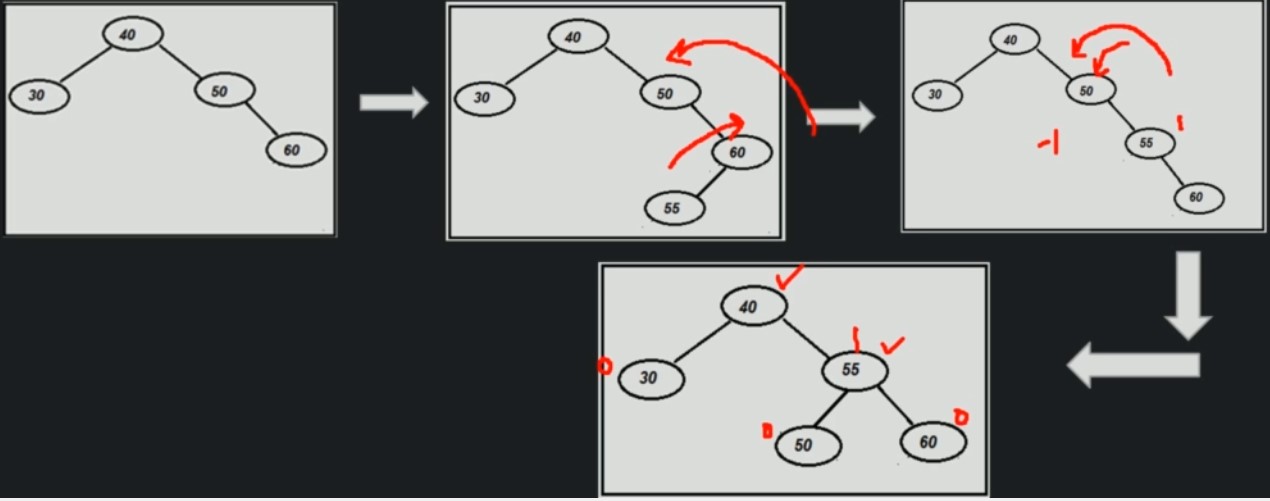
**Time Complexity: Right-Right condition**

leftRotate(cdNode) cdNode as currentDisbandedNode  
 newRoot=cdNode.Right ---O(1)  
 cdNode.Right=cdNode.Right.Left ---O(1)  
 newRoot.Left=cdNode ---O(1)  
 cdNode.Height=calculatedHeight(cdNode) ---O(1)  
 newRoot.Height=calculatedHeight(newRoot) ---O(1)  
 return newRoot ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

Right-Left Condition:

What is Right-Left condition?

* Right-Left node from currentNode causing disbalance
* In this case do a Right Rotation followed by Left Rotation



**Time Complexity: Right-Right condition**

rightRotate(cdNode'sRightChild) cdNode as currentDisbandedNode  
 newRoot=cdNode'sRightChild.Left ---O(1)  
 cdNode'sRightChild=cdNode'sRightChild.Left.Right ---O(1)  
 newRoot.Right=cdNode'sRightChild ---O(1)  
 cdNode'sRightChild.Height=calculatedHeight(cdNode'sRightChild)---O(1)  
 newRoot.Height=calculatedHeight(newRoot) ---O(1)  
 return newRoot ---O(1)  
  
leftRotate(cdNode) cdNode as currentDisbandedNode  
 newRoot=cdNode.Right ---O(1)  
 cdNode.Right=cdNode.Right.Left ---O(1)  
 newRoot.Left=cdNode ---O(1)  
 cdNode.Height=calculatedHeight(cdNode) ---O(1)  
 newRoot.Height=calculatedHeight(newRoot) ---O(1)  
 return newRoot ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: Insertion in AVL tree (End to End)**

insert(root,data)  
 if(root==null) ---O(1)  
 return new Node(data)//BST condition ---O(1)  
 else if(data <= root.data) ---O(1)  
 root.left=insert(root.left, data)//BST condition ---O(log n)  
 else ---O(1)  
 root.right=insert(root.right, data)//BST condition ---O(log n)  
 int balance=height(root.left)-height(root.left) ---O(1)  
 if(balance>1)//If left subtree is overloaded ---O(1)  
 if(height(root.left.left)>=height(root.left.right)) ---O(1)  
 RightRotation(root)//LL condition ---O(1)  
 else ---O(1)  
 LeftRotation(root.left);RightRotation(root)//LR condition---O(1)  
 else if(balance < -1)//If right subtree is overloaded ---O(1)  
 if(height(root.right.right)>=height(root.right.left))---O(1)  
 LeftRotation(root)//RR condition ---O(1)  
 else ---O(1)  
 return RightRotation(root.right);LeftRotation(root)//RL condition---O(1)  
 root.height=max(root.left, root.right)+1 ---O(1)  
 return root ---O(1)  
  
Time Complexity: O(lon n)  
Space Complexity: O(log n)

Deletion of node from AVL Tree

Case 1: When tree does not exist.

Case 2: When rotation is not required

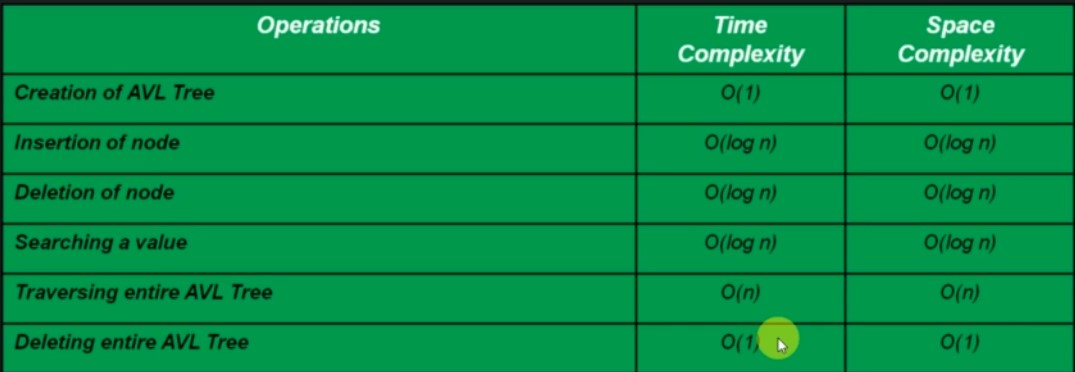
Case 3: When rotation is required (LL, LR, RR RL)

**Time Complexity: Deletion of node from AVL Tree**

deleteNodeOfAVL(currentNode, value)  
 if(currentNode==null)  
 return null;  
 if(value<currentNode.value)  
 currentNode.left=deleteNodeOfAVL(currentNode.left, value)---O(log n)  
 else if(value>currentNode.value)  
 currentNode.right=deleteNodeOfAVL(currentNode.right, value)---O(log n)  
 else//current node is the node tobe deleted  
 if currentNode have both children, then find maximum element from right source(case #3)  
 replace currentNode with minimum node from right subtree and delete minimum node from right  
 else if node toBeDeleted has only left child(case #2)  
 then currentNode=currentNode.left  
 else if node toBeDeleted has only right child(case #2)  
 then currentNode=currentNode.right  
 else//node toBedeleted do not have child  
 currentNode=null  
 int balance=checkBalance(currentNode.left, currentNode.right)  
 if(balance>1)  
 if(checkBalance(currentNode.left.left, currentNode.left.right)>0)  
 currentNode=RightRotate(currentNode)//LL condition  
 else  
 currentNode.left=LeftRotate(currentNode.left)//LR Condition  
 currentNode=RightRotate(currentNode)  
 else if(balance<1)  
 if(checkBalance(currentNode.right.right, currentNode.right.left)>0)  
 currentNode=LeftRotate(currentNode)//RR condition  
 else  
 currentNode.right=RightRotate(currentNode.Right)//RL Condition  
 currentNode=LeftRotate(currentNode)  
 if(currentNode.left != null)  
 currentNode.left.setHeight(calculatedHeight(currentNode.left))  
 if(currentNode.right != null)  
 currentNode.right.setHeight(calculatedHeight(currentNode.right))  
 currentNode.setHeight(calculatedHeight(currentNode))  
 return currentNode;  
  
Time Complexity: O(lon n)  
Space Complexity: O(log n)

**Time Complexity: Deletion entire AVL Tree**

deleteAVLTree()  
 root=null ---O(1)  
   
Time Complexity: O(1)  
Space Complexity: O(1)



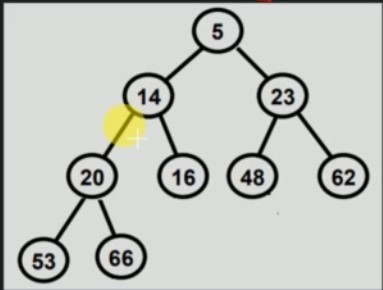
**BinarySearchTree vs AVL**



[011\_data\_structure/src/s3\_c3\_tree/avltree at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c3_tree/avltree)

**3.3.4. Binary Heap**

* Binary Heap is a Binary Tree with some special properties
* Heap Property
  + Value of any given node must be <= value of children (Min-Heap)
  + Value of any given node must be >= value of children (Min-Heap)
* Complete Tree
  + All levels are completely filled except possibly the last level and the last level has all keys as left as possible.
  + This makes Binary Heap ideal candidate for Array Implementation



Why should we learn Binary Heap?

There are cases when we want to find ‘min/max’ number among set of numbers in log(n) time. Also, we want to make sure that inserting additional numbers does not take more than O(log n) time.

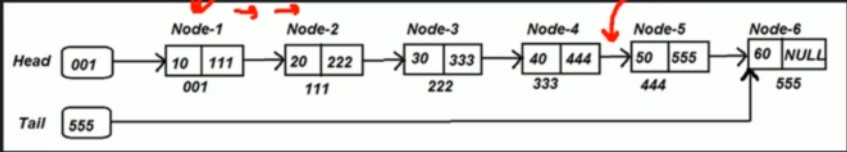
Possible solutions

1. Stores the numbers in sorted array



Issue here is that once we insert/delete a new number, our array need to be adjusted again to keep it sorted which will take O(n) time.

1. Stores the numbers in Linked list in sorted manner



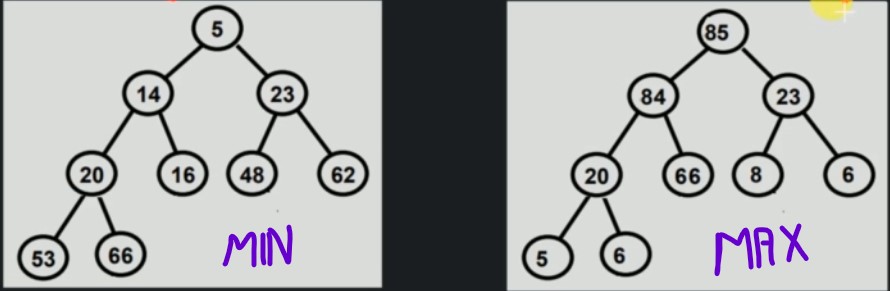
Practical Use

* Prime’s Algorithm
* Heap Sort
* Priority Queue

Types of Binary Heap

Min Heap: If the value of each node is less than or equal to value of both of its children

Max Heap: If the value of each node is more than or equal to value of both of its children



Common operations:

createHeap : Create a blank array to be used for sorting heap

peekTopOfHeap : return min/max from Heap

extractMin : extracts Min from heap. We can extract only this node

extractMax : extracts Max from heap.

sizeOfHeap : returns the size of the Heap

insertValueInHeap : Insert value in Heap

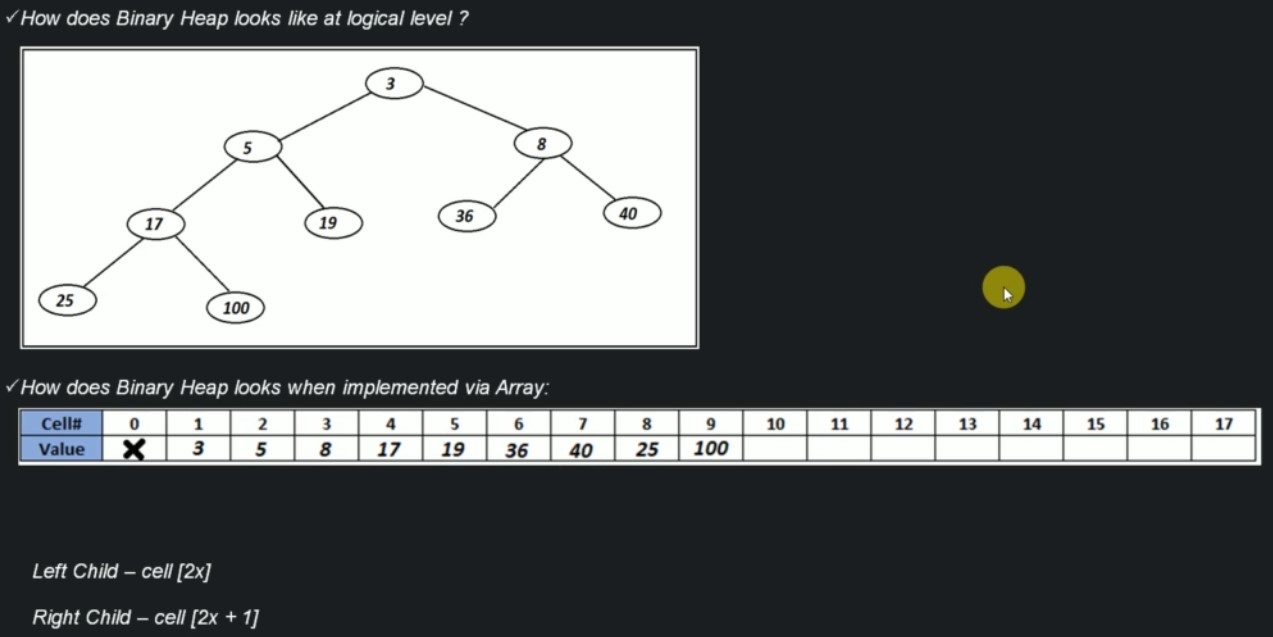
deleteHeap : delete the entire Heap

Implementation Options:

Array based Implementation

Reference based implementation

Binary Heap: Array based Implementation



**Time Complexity: Creation of Binary Heap**

createBinaryHeap(size)  
 create a blank array of size(size+1) ---O(1)  
 initialize sizeOfHeap with 0 ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: peek of Binary Heap**

peekOfHeap()  
 if tree does not exist ---O(1)  
 return error message ---O(1)  
 else ---O(1)  
 return 1st cell of array ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: Size of Binary Heap**

sizeOfHeap()  
 return sizeOfHeap ---O(1)  
  
Time Complexity: O(1)  
Space Complexity: O(1)

**Time Complexity: Insertion in Binary Heap**

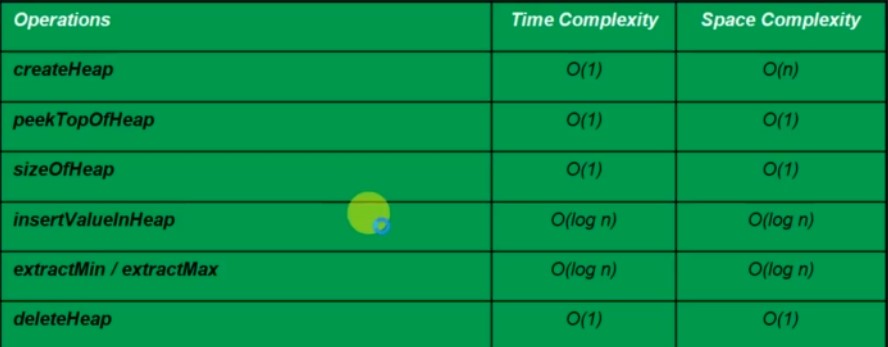
InsertInHeap(value)  
 if tree does not exist ---O(1)  
 return error message ---O(1)  
 else ---O(1)  
 insert value in first unused cell of array ---O(1)  
 sizeOfHeap++ ---O(1)  
 heapifyBottomToTpo(sizeOfHeap) ---O(log n)  
  
Time Complexity: O(log n)  
Space Complexity: O(log n)

**Time Complexity: ExtractMin Heap from Binary Heap**

extractMin(value)  
 if tree does not exist ---O(1)  
 return error message ---O(1)  
 else ---O(1)  
 extract 1st cell of array ---O(1)  
 promote last element to first ---O(1)  
 sizeOfHeap-- ---O(1)  
 heapifyTopToBottom ---O(log n)  
  
Time Complexity: O(log n)  
Space Complexity: O(log n)

**Time Complexity: Delete Binary Heap**

deleteHeap()  
 set array as null ---O(1)  
   
Time Complexity: O(1)  
Space Complexity: O(1)

****

Binary Heap: References based Implementation

Why to avoid References based Implementation?

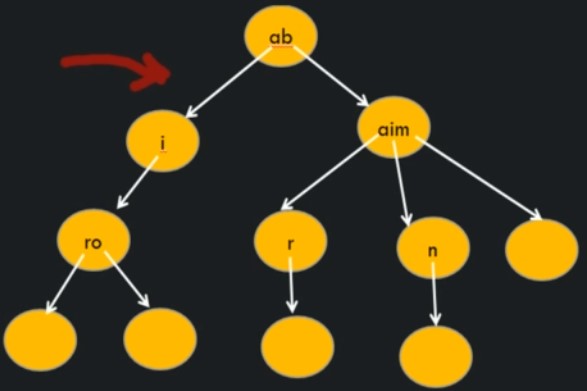
There is no way to find out the last element of the Binary Heap.

[011\_data\_structure/src/s3\_c3\_tree/binaryheap at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c3_tree/binaryheap)

**3.3.5. Trie**

What is Trie?

* It is a search tree, which is typically used to store/search strings in space/time efficient way.
* In it, any node can store non repetitive multiple characters
* Also, every node store ‘link’ of next character of the String
* Also, every node keeps a track of ‘end of String’



Why Learn Trie?

Used to solve many standard problems in efficient way

* Spelling checker
* Auto complete String

**Time Complexity: Creating a Trie**

createTrie()  
 create a blank root node

Time Complexity: O(1)  
Space Complexity: O(1)

Inserting a String in Trie

Case 1: Trie is blank

Case 2: New String’s prefix is common with another String’s prefix (aio)

Case 3: New String’s prefix is already present as complete String (airk)

Case 4: String to be inserted in already presented Trie

Searching a String in Trie

Case 1: String does not exist in Trie

Case 2: String exist in Trie

Case 3: Current String is a prefix of another String. But the String does not exist in Trie

Deleting a String in Trie

Case 1: Some others word’s prefix is same as Prefix of this word

Case 2: The word is a prefix of some other words

Case 3: Some other word is prefix of this word

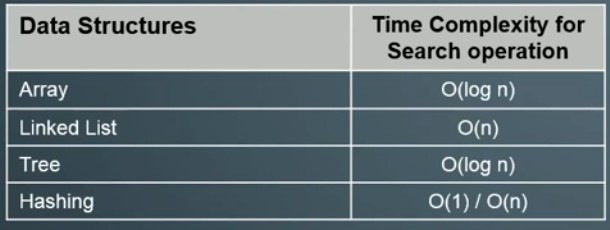
Case 4: No one is dependent on this word

[011\_data\_structure/src/s3\_c3\_tree/trie at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c3_tree/trie)

**3.4. Hashing**

Hashing is a method of sorting and indexing data. The data behind hashing is to allow large amounts of data to be indexed using keys commonly created by formulas.

Why needs Hashing?



Some Terminologies

**Hash Function:** A Hash function is any function that can be used to map data of arbitrary size to data of fixed size.

**Key:** Input data given by user

**Hash Value:** The values returned by a hash function are called hash values, hash codes, digests or simply hashes.

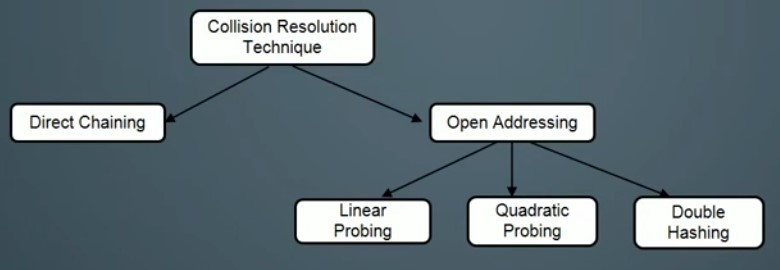
**Hash table**: It is a data structure which implements an associative array abstract data type, a structure that can map keys to values.

**Collision:** A collision occurs when two different key to a hash function produce the same output called hash value.

Characteristics of Good Hash Function:

* It distributes hash values uniformly across the hash table
* The Hash function uses all the input data.

Collision Resolution Technique:



**Direct Chaining:**

Implements the bucket as linked lists. Colleding elemets are stored in these list.

**Open Addressing:**

Colliding elemets are stored in other vacant buckets. During storage and lookup, these are found through so-called ‘probing’

Linear Probing: It is a stratagy for resolving collisions, by placing the new key into the closeest following empty cell

Quadritic Probing: Quadratic probing operaties by taking the original hash index and adding successive values of an arbitary quadratic polynomial until an open slot is found.

Double Hashing: Interval probes is computed another hash function.

Pros and Cons of Collision Resolution Technique:

Direct Chaining

* No fear of exhausting Hash Table buckets
* Fear of big Linked List (can impact performance big time)

Open Addressing

* Easy implementation
* Fear of exhausting Hash table bucket

If input size is unknow then always use ‘Open Addressing’, else can use any of two.

If deletion is very high then we should always go for Direct Chaining.

Pros and Cons of Hashing:

Pros: On an average Insertion/Deletion/Search operations takes O(1) time.

Cons: In the worst case Insertion/Deletion/Search moght take O(n) time (when hash function is not good enough).

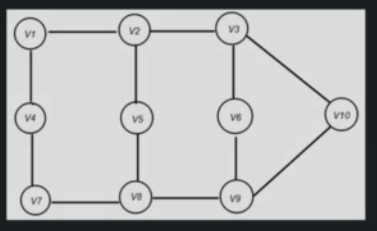


[011\_data\_structure/src/s3\_c4\_hashing at master · bibhusprasad/011\_data\_structure (github.com)](https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c4_hashing)

**3.5. Graph**

What is Graph?

Graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices



V: {V1, V2, V3, V4, V5, V6, V7, V8, V9, V10}

E: {V1-V2, V1-V4, V2-V3, V2-V5, V3-V6, V3-V10, V4-V7, V5-V8, V6-V9, V7-V8, V8-V9, V9-V10}

Common Graph Terminology?

Vertices: Vertices are the nodes of the graph

Edges: Edges are the arcs that connect pairs of vertices

Unweighted graph: A graph not having a weight associate with any edge

Weighted graph: A graph having a weight associate with each edge

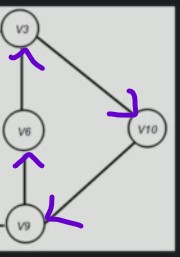


Undirected graph: It is a graph that is set of vertices connected by edges, where the edges don’t have a direction associated with them.

Directed graph: It is a graph that is a set of vertices connected by edges, where the edges have a direction associate with them.

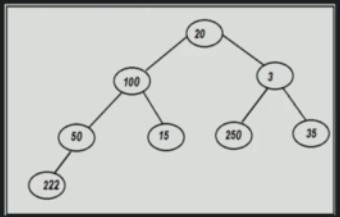


Cyclic graph: A cyclic graph is a graph having at least one loop.

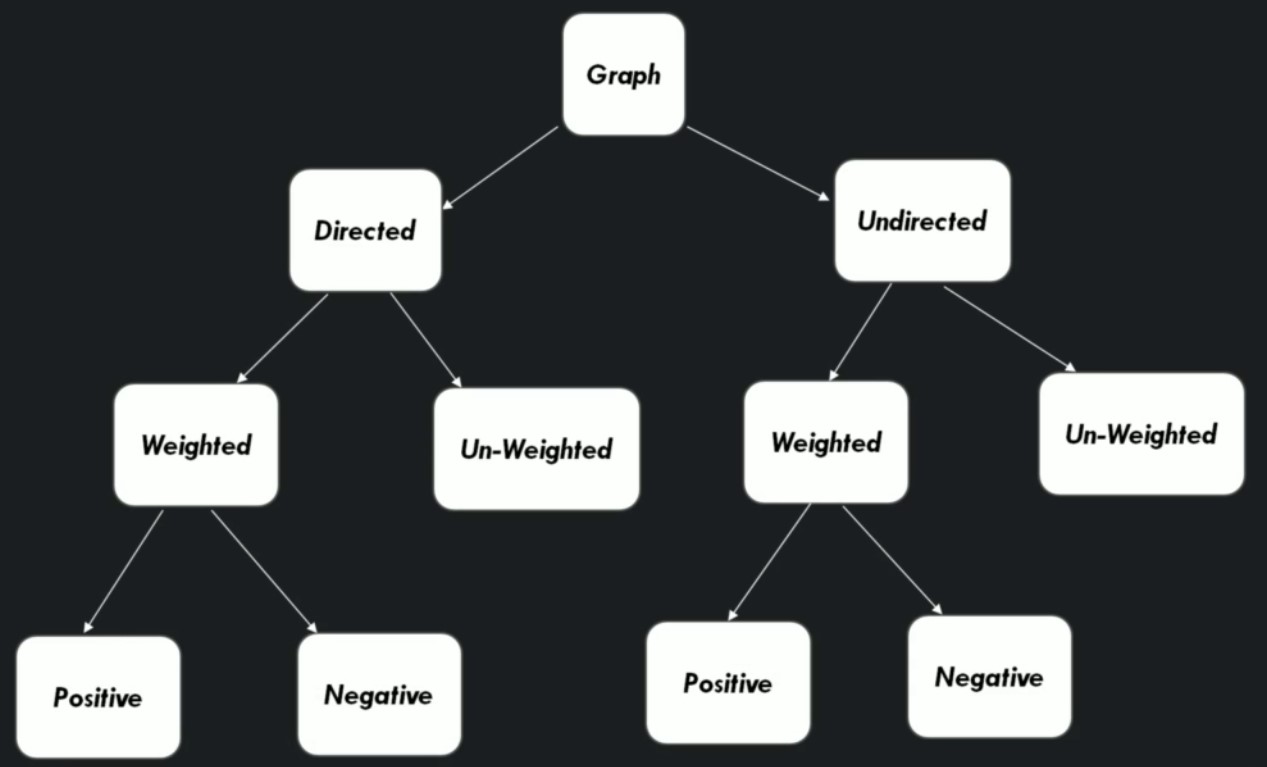


Acyclic graph: An acyclic graph is a graph having no loop

Tree: Tree is a special class of Directed Acyclic graph.

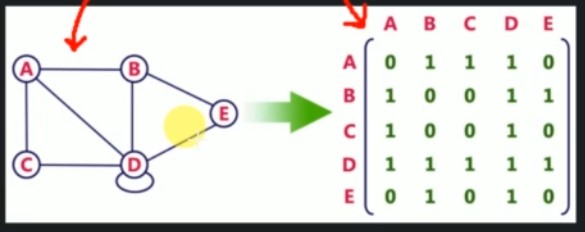


Types of Graph:

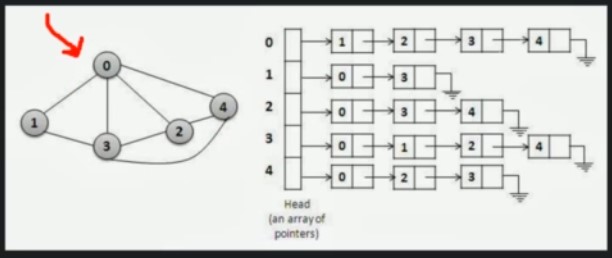


How Graph is represented:

1. Adjacency Matrix: In Graph theory, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.



1. Adjacency List: In a Graph theory, an adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the sets of neighbours of vertex in the Graph.



When to use which representation:

* If the Graph is complete or near to complete, then we should use adjacency matrix
* If the numbers of edges are few, then we should go for adjacency list