① A data structure is a storage that is used to store & organize data. It is a way of arranging data on a computer so that it can be accessed & updated efficiently.

Data structure
- Linear
  - Static
    - Array
  - dynamic
    - Queue
    - Stack
    - Link list
- Non-linear
  - Tree
  - Graph

② An array is a collection of or group of similar datatypes or data items or elements. It has contiguous memory locat-ions. Here static memory allocation happens.

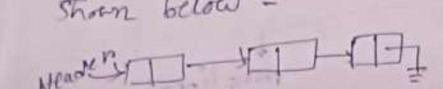
0	1	2	3

→ array

③ A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays is generally stored in row major order in the memory. The general form of declaring N-dimensional arrays is shown below -

data_type array_name[size1][size2] ... [sizeN];

(4) L.L. is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a l.l. are linked using pointers as shown below -



(5) Advantages of l.l. over array -

1) Here dynamic memory allocation happens that means we can allocate memory during runtime.

2) Here the size of list isn't fixed, i.e we can allocate as much no. of nodes that are required at any time.

3) L.L can store info. anywhere in the memory.

4) It is more efficient than array.

5) Here insertion & deletion operation is faster.

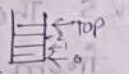
6) A d.l.l is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list, & also the data.

Advantages of d.l.l. over s.l.l. - In copy

(7)

1) Implementing stacks
2) queues using linkedlist
3) Implementation of graphs
4) Implementing hash tables
5) Portraying a polynomial with a l.l.
6) large no. arithmetic
7) linked allocation of files
8) memory management with l.l.

(8) A stack is a linear data structure that follows the LIFO (last in first out) principle. A stack can be defined as a container in which insertion & deletion can be done from the one end known as the top of the stack.



Applications of stack -

- Execution of recursive programme
- Evaluation to arithmetic oper expressions
- Binding rules (static bindin & dynamic binding)

Basic operations performed on stack -

1) push
2) pop
3) isEmpty
4) isFull
5) top (Display the topmost element of the stack)

(10) A Queue is defined as a linear data structure that is open at both ends & the operations are performed in first In First Out (FIFO) order. We define a queue

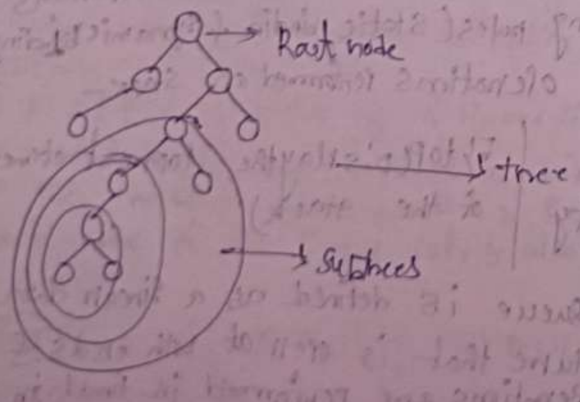
→ Insert the node in increasing order.

to be a list in which all additions to the list are made at one end, & all deletions from the list are made at the other end.

11) Basic operations on queue –

1) enqueue
2) dequeue
3) peek
4) isFull
5) isEmpty

12) A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node & has subtrees connected to the root.



13) Basic terminology in tree –

1) Root →
2) node
3) Parent
4) sibling
5) Edge
6) Child
7) General tree (tree)
8) ~~Left child~~ Leaf node
9) Subtree
10) Binary search tree
11) Height
12) Leaf
13) Degree
14) Depth
15) Level
16) AVL tree
17) Binary tree
18) Path
19) Spanning tree
20) Heap
21) Descendent (Successor node)
22) Ancestor (Predicessor)

14) Binary trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree & right subtree. Based on the no. of children, binary trees are divided into 3 types – full binary tree, complete binary tree & perfect binary tree.

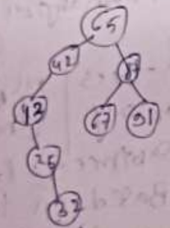
15) Tree traversal is a process to visit all the nodes of a tree & may print their values too.

Tree data structure can be traversed in following ways (types):

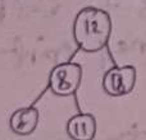
1) Depth First Search or DFS –
 1) Inorder traversal
 2) Preorder traversal
 3) Postorder traversal
2) Level order traversal or Breadth First Search or BFS.
3) Boundary Traversal
4) Diagonal traversal

16) A BST BT is knowned as BST. If each node 'N' satisfies the following properties
i) The value at N is greater than every value in the left subtree of N & is less than every value in the right subtree of N.

62, 42, 13, 87, 91, 67, 25, 32



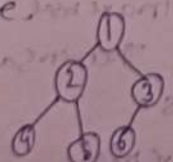
(17) Full binary tree – A full binary tree is a binary tree with either zero or 2 child nodes for each node. (or)



→ full binary tree.

A tree called bin[ary] tree if there [are] max. no of nodes possible in all levels. i.e. it con[tains] just 1 ...

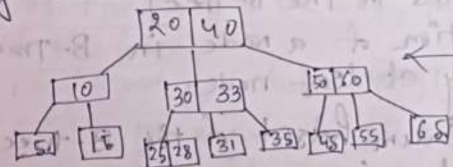
Complete binary tree – A BT is called CBT if it has max no. of nodes in all level except possibly that last level & the filling of last level will start from as left as possible.



→ CBT

(18) B-Trees – B-tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node & large key values by keeping the height of the tree relatively small.

B-tree is a special type of self-balancing search tree in which each node can contain more than one key & can have more than 2 children. It is a generalized form of binary search tree. It is also known as a height-balanced m-way tree.



← B-Tree

Properties – i) All leaves (leaf) are at the same level.

ii) B-Tree is defined by the term minimum degree 't'. The value of 't' depends upon disk block size.

iii) Every node except the root must contain at least t-1 keys. The roots may contain a minimum of 1 key.

→ Insert the node in increasing order.

iv) All nodes (including root) may contain at most (2^*t-1) keys.
v) No. of children of node is equal to the number of keys in it plus 1.
vi) All keys of a node are sorted in increasing order. The child between 2 keys k1 & k2 contains all keys in the range from k1 & k2.
vii) B-Tree grows & shrinks from the root which is unlike Binary search tree. Binary search trees grow downward & also shrink from downward.
viii) the tc of to search, insert & delete is $O(\log n)$ [like other BSTs, n = total no. of elements in the B-Tree]
ix) Insertion of a node in B-Tree happens only at leaf node.

→ The minimum height of the B-tree that can exist with n numbers of nodes & m is the maximum no. of children of a children of a node can have is:

$$\boxed{h_{min} = \lceil \log_m(n+1) \rceil - 1}$$

→ The maximum height of the B-Tree that can exist with n number of node's & t is the minimum number of children that a non-root node can have is:

$$\boxed{t = \lceil \tfrac{m}{2} \rceil}$$ &

Operations on a B-Tree

1) Traversal - Traversal is also similar to Inorder traversal of B.T. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children & keys. In the end, recursively print the rightmost child.

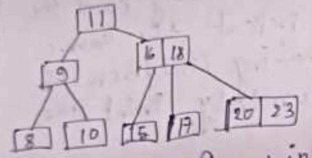
2) Searching an element in a B-Tree

Searching for an element in a B-Tree is the generalized form of searching an element in a Binary search Tree. The following steps are followed -

i) Starting from the root node, compare K with the first key of the node.
If $\boxed{K = \text{The first key of the node}}$, return the node & the index.
ii) If $\boxed{K.\text{leaf} = \text{true}}$, return $\boxed{\text{NULL}}$ (i.e. not found).
iii) If $\boxed{K < \text{the 1st key of the root node}}$, search the left child of this key recursively.
iv) If there is more than one key in the current node & $\boxed{K > \text{the 1st key}}$, compare K with the next key in the node.
If $\boxed{K < \text{next key}}$, search the left child of this key (ie. K lies in between the 1st & the 2nd keys).
Else, search the right ~~the~~ child of the key.
5) Repeat steps 1 to 4 until the leaf is reached.

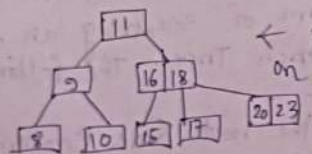
Searching Example

i) Let us search key $\boxed{K=17}$ in the tree below of degree 3.



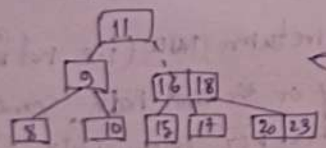
← B Tree

ii) $\boxed{K}$ is not found in the root so, compare it with the root key.



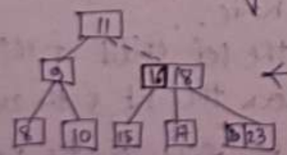
← K isn't found on the root node

iii) Since $\boxed{K>11}$, go to the right child of the root node.



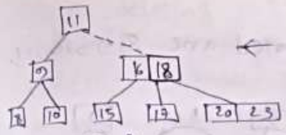
← go to the right subtree

iv) Compare K with 16. Since $\boxed{K>16}$, Compare K with the next key 18.



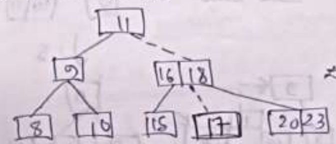
← Compare with the keys from left to right

v) Since $\boxed{K<18}$, K lies between 16 & 18. Search in the right child of 18.



← K lies in between 16 & 18

vi) K is found.



← K is found

Insertion on B-tree

Inserting an element on a B-tree consists of 2 events: searching the appropriate node to insert the element & splitting the node if required. Insertion operation always takes place in the bottom-up approach.

Insertion operation

i) If the tree is empty, allocate a root node & insert the key.

ii) Update the allowed number of keys in the node.

iii) Search the appropriate node for insertion.

iv) If the node is full, follow the steps below

v) Insert the elements in increasing order.

vi) Now, there are elements greater than its limit. So, split at the median.

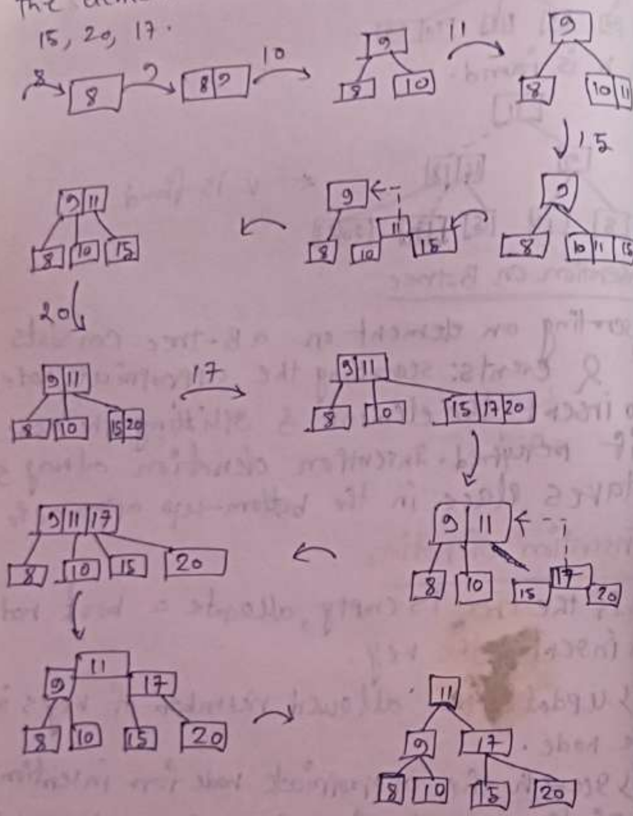
vii) Push the median key upwards & make the left keys as a left child & the right keys as right child.

viii) If the node is not full, follow the steps below.

ix) Insert the node in increasing order.

Insertion Example

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.



Inserting elements into a B-tree

Deletion from a B-Tree

Deletion from a B-Tree – Deleting an element on a B tree consists of 3 main events: searching the node where the key to be deleted exists, deleting the key & balancing the tree if required.

While deleting a tree, a condition called underflow may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.

Deletion operation's term

i) Inorder predecessor – The largest key on the left child of a node is called its inorder predecessor.

ii) Inorder successor – The smallest key on the right child of a node is called its inorder successor.

Deletion operation – Before going through the steps below, one must know these facts about a B tree of degree m.

i) A node can have a max of m children (i.e 3)

ii) A node can contain a maximum of $\lceil m-1 \rceil$ keys (i.e 2)

iii) A node should have a min of $\lceil m/2 \rceil$ children (i.e 2)

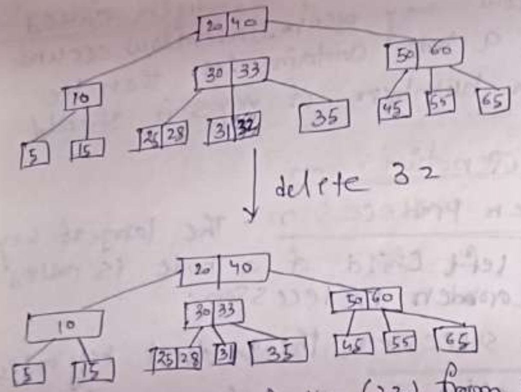
iv) A node (except root node) should contain a minimum of $\lceil m/2 \rceil - 1$ keys (i.e 1)

There are 3 main cases for deletion operation in a B tree.

Case I – The key to be deleted lies in the leaf. There are 2 cases for it.

i) The deletion of the key doesn't violate the property of the minimum number of keys a node should hold.

In the tree below, deleting 32 does not violate the above properties.



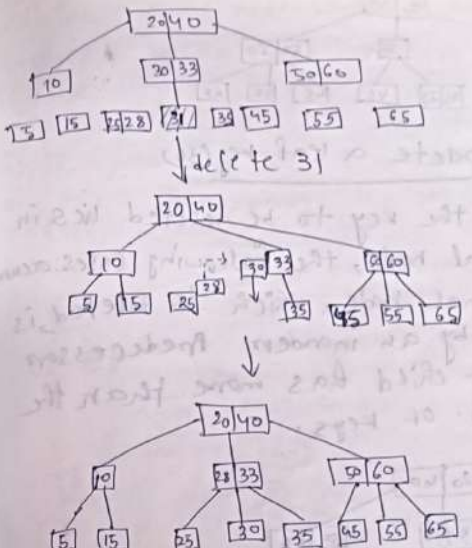
deleting a leaf key (32) from B-tree

(ii) The deletion of the key violates the property of the minimum no. of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

First, visit the immediate left sibling. If the left sibling node has more than a minimum no. of keys then borrow a key from this node.

Else check to borrow from the immediate right sibling node.

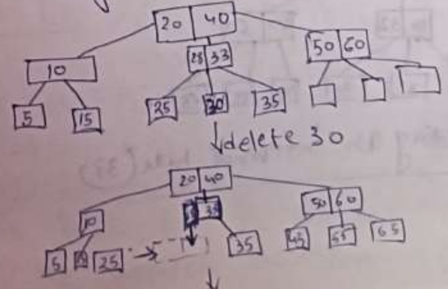
In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling node.

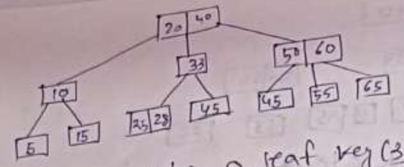


Deleting a leaf key (31)

If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.

Deleting 30 results in the above case

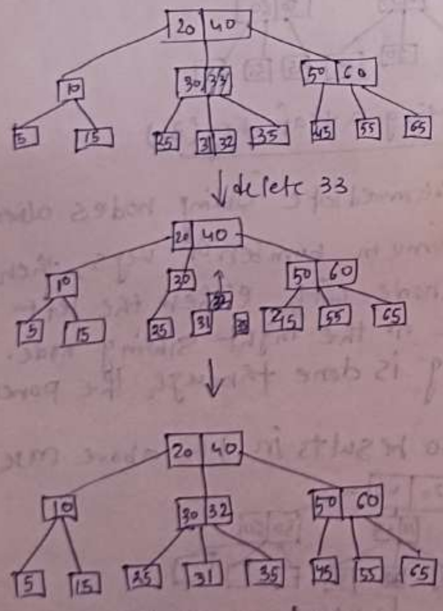




Delete a leaf key (30)

Case II - If the key to be deleted lies in the internal node, the following cases occur.

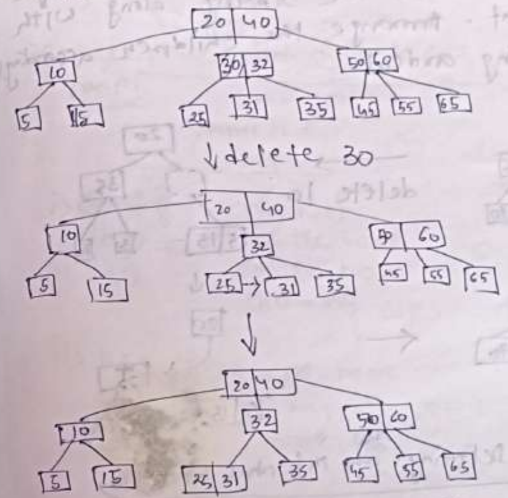
i) The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum no. of keys.



Deleting an internal node (33)

ii) The internal node, which is deleted, is replaced by an ~~inde~~ inorder successor if the right child has more than the min no. of keys.

iii) If either child has exactly a min no. of keys then, merge the left and right children.



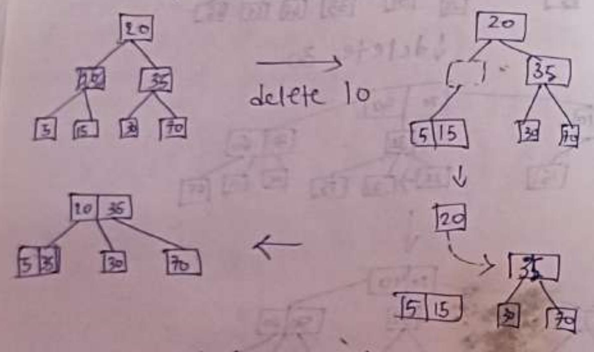
Deleting an internal node (30)

After merging if the parent node has less than the min. no. of keys then, look for the siblings as in case I.

Case III - In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer no. of keys in the node (i.e less than the min required) then look for the inorder predecessor & the inorder successor. If both the childn

-en contain a min. no. of keys then, borrow-ing cannot take place. This leads to case II (3) i.e merging the children.

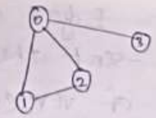
Again look for the sibling to borrow a key. But, if the sibling also has only a min no. of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).



Deleting an internal node (10)

19) Graph data structure – A graph ds is a collection of nodes that have data & are connected to other nodes. Every relationship is an edge from one node to another. More precisely, a graph is a data structure (V,E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



← vertices & edges

In the graph,

$V = \{0,1,2,3\}$

$E = \{(0,1),(0,2),(0,3),(1,2)\}$

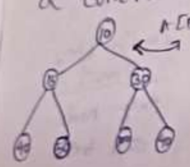
$G = \{V,E\}$

20) Difference b/w directed & undirected graph

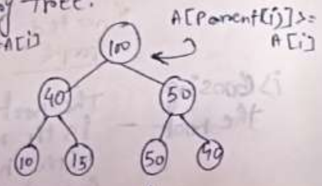
	Directed graph	Undirected graph
i) Choosing the root –	The root is the node with no incoming edges	Any node can be chosen as the root
ii) DFS check –	No node must be visited more than once	No node must be visited more than once. Also, the parent shouldn't be considered as a child
iii) Connectivity check –	Check that all node are visited.	Check that all nodes are visited.
iv) TC –	O(V+E)	O(V+E) //
v) Defi –	A type of graph that contains ordered pairs of vertices	A type of graph that contains unordered pairs of vertices.

ii) Edges represent the direction of vertexes	Edges donot represent the direction of vertexes.
iii) An arrow represents the edges	* Undirected arcs represent the edges.

2) Heap - A heap is a special Tree-based data structure in which the tree is a complete binary tree.



min heap



max heap

Operation of Heap Data Structure

i) Heapify: A Process of creating a heap from an array.

ii) Insertion: Process to insert an element in existing heap time complexity O(log n)

iii) Deletion: Deleting the top element of the heap or the highest priority element, and then organizing the heap & returning the element with time complexity O(log n)

iv) Peek: to check or find the first (or can say the top) element of the heap.

Types of Heap Data Structure

Generally, Heaps can be of two types:

i) max-Heap: In a max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

ii) Min-Heap: In a min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

for max Heap - The total number of Comparisons required in the max heap is according to the height of the tree. The height of the complete b.t. is always log n; therefore, the time complexity would also be O(log n).

for min-heap - The total no. of Comparisons required in the min heap is according to the height of the tree. The height of the complete binary tree is always log n; therefore, the time complexity would also be O(log n).

Properties of Heap

i) CBT – A heap tree is a complete b.t, meaning all levels of the tree are fully filled except possibly the last level, which is filled from left to right. This property ensures that the tree is efficiently represented using an array.

ii) Heap Property – This property ensures that the minimum (or maximum) element is always at the root of the tree according to the heap type.

iii) Parent-child Relationship – The relationship between a parent node at index 'i' & its children is given by the formulas: left child at index $2i+1$ & right child at index $2i+2$ for 0-based indexing of node numbers.

iv) Efficient Insertion & Removal – Insertion & removal operations in heap trees are efficient. New elements are inserted at the next available position in the bottom-rightmost level, and the heap property is restored by comparing the element with its parent & swapping if necessary. Removal of the root element involves replacing it with the last element & heapifying down.

v) Efficient Access to External – The minimum or maximum element is always at the root of the heap, allowing constant-time access.

Heap Operations –

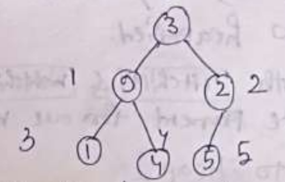
i) Heapify – It is the process of creating a heap data structure from a binary tree. It is used to create a min-Heap or a max-Heap.

i) Let the i/p array be –

3	9	2	1	4	5
0	1	2	3	4	5

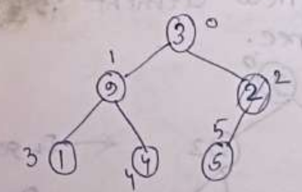
← Initial Array

ii) Create a complete binary tree from the array



← Complete binary tree

iii) Start from the 1st index of non-leaf node whose index is given by $\boxed{n/2-1}$.



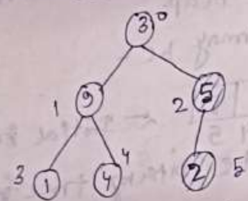
← Start from the 1st non-leaf node.

iv) Set current element $\boxed{i}$ as $\boxed{\text{largest}}$.

v) The index of left child is given by $\boxed{2i+1}$ & the right child is given by $\boxed{2i+2}$.

If $\boxed{\text{leftchild}}$ is greater than $\boxed{\text{current Element}}$ (i.e element at $\boxed{i\text{th}}$ index) set $\boxed{\text{leftChildIndex}}$ as largest. If $\boxed{\text{rightChild}}$ is greater than element in $\boxed{\text{largest}}$, set $\boxed{\text{rightChildIndex}}$ as $\boxed{\text{largest}}$.

vii) Swap $\boxed{\text{largest}}$ with $\boxed{\text{currentElement}}$



← swap if necessary

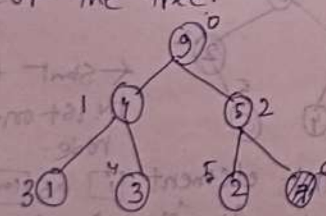
viii) Repeat steps iii) – vii) until the sub-trees are also heapified.

For min-Heap, both $\boxed{\text{leftChild}}$ & $\boxed{\text{rightChild}}$ must be larger than the parent for all nodes.

Insert Element into Heap –

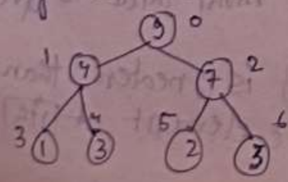
For max-Heap

i) Insert the new element at the end of the tree.



← Insert at the end

ii) Heapify the tree.



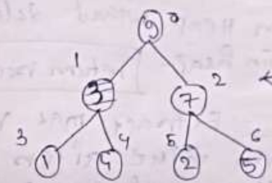
← Heapify the array

For min-Heap, the above algo. is modified so that $\boxed{\text{parentNode}}$ is always smaller than $\boxed{\text{newNode}}$.

Delete Element from Heap

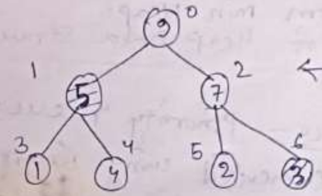
For max-Heap

i) Selected the element to be deleted.



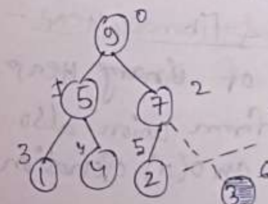
← selected the element to be deleted

ii) Swap it with the last element.



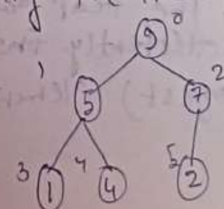
← Swap with the last element

iii) Remove the last element.



← Remove the last element

iv) Heapify the tree.



← Heapify the array

For min-Heap, above algo. is modified so that both $\boxed{\text{child Nodes}}$ are ~~greater~~ smaller than $\boxed{\text{current Node}}$.

Peek (Find max/min) – Peek operation returns the maximum element from max Heap or minimum element from min Heap without deleting the element.
For both max & min heap – $\boxed{\text{return rootnode}}$

Extract-max/min – Extract max returns the node with maximum value after removing it from a max-Heap whereas Extract min returns the node with minimum after removing it from min Heap.

Implementation of Heap Data Structure

Applications
i) Priority queues – Priority queues can be efficiently implemented using Binary Heap because it supports insert() delete() & extractmax(), decreasekey() operation in O(log N) time.
ii) Binomial Heap & Fibonacci Heap – They are the variations of Binary Heap. These variations perform union also in O(log N) time which is an O(N) operation in Binary Heap.
iii) Order Statistics – The Heap data structure can be used to efficiently find the Kth smallest (or largest) element in an array.

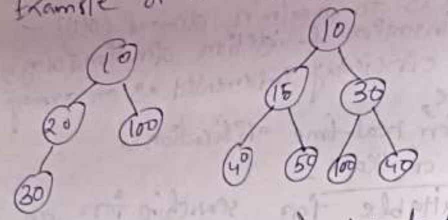
Advantages of Heaps –
i) Fast access to max/min element (O(1))
ii) Efficient insertion & deletion operations (O(log n))
iii) Can be efficiently implemented as an array.
iv) Flexible size.
v) Suitable for real-time applications.

Disadvantages of Heaps –
i) Not suitable for searching for an element other than max/min (O(n) in worst case)
ii) Extra memory overhead to maintain heap structure.
iii) Slower than other data structures like arrays & linked lists for non-Priority queue operations.

Applications of Heaps (others)
i) Heap sort – It uses Binary Heap to sort an array in O(n log n) time.
ii) ~~Priority~~ Graph algorithms – The Priority queues are specially used in graph algorithms like Dijkstra's shortest path & Prim's minimum spanning Tree

Binary heap – A Binary Heap is a Complete B.T. which is used to store data efficiently to get the max or min element based on its structure. It is represented basically as an array.
A binary heap is either min Heap or max Heap. In a min Binary heap, the key at the root must be minimum among all keys present in Binary Heap. The same Property must be recursively true for all nodes in Binary Tree. max Binary Heap is similar to min Heap.

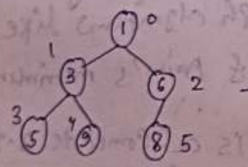
Example of min Heap,



- The root element will be at Arr[0].
- The below table shows indices of others nodes for the ith node, i.e., Arr[i]:

Parent node — Arr[(i-1)/2]
L child " —— Arr[(2*i)+1]
R child " ⟶ Arr[2*i+2]

The traversal method use to achieve Array representation is **Level order Traversal** (for Binary heap)



1	3	6	5	9	8
0	1	2	3	4	5

Heap	Tree
1) It is a kind of Tree itself.	1) The tree isn't a kind of heap
2) Usually, Heap is of 2 types, max & min heap.	2) Tree can be of various types, Eg (AVL, BT, BST)
3) It is ordered	3) BT isn't ordered but BST is ordered.
4) Insert & remove operation takes time of O(log(N)).	4) Insert & remove operation will take time of O(N)
3) It can also be refferred to as priority queue.	5) A tree can also be refferred to as connected undirected graph with no cycle.
6) Finding max/min value in Heap is O(1) in the respective min/max heap.	6) Finding min/max value in BST is O(log(N)) & Binary tree is O(N)
7) It can be build in linear tc.	7) BST O(N + log(N)) & Binary tree O(N).

Is the structure of a heap unique?

No, because there can be multiple valid heap structures for a given set of values. However, the property of being a min-heap or max-heap is unique for a given set of values. Both of these structures are valid heap structures, but one is a min heap & the other is a max heap.

Some other types of Heaps—

1) Binomial Heap – The main application of Binary heap is as implement a

Priority queue. Binomial Heap is an extention of Binary heap that provides faster union or merge operation with other operations provided by Binary Heap.
A binomial heap is a collection of binomial trees.
A binomial trees of order 0 has 1 node. A binomial tree of order k can be constructed by taking 2 binomial trees of order k-1 & making one the leftmost child of the other.

2) <u>Fibonacci heap</u> - It is a ds for Priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other Priority queue data structures including the binary heap & binomial heap.

③ Leftist Heap, ④ K-ary Heap

<u>Greedy algorithm</u> - It is an algo. paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious & immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for greedy.