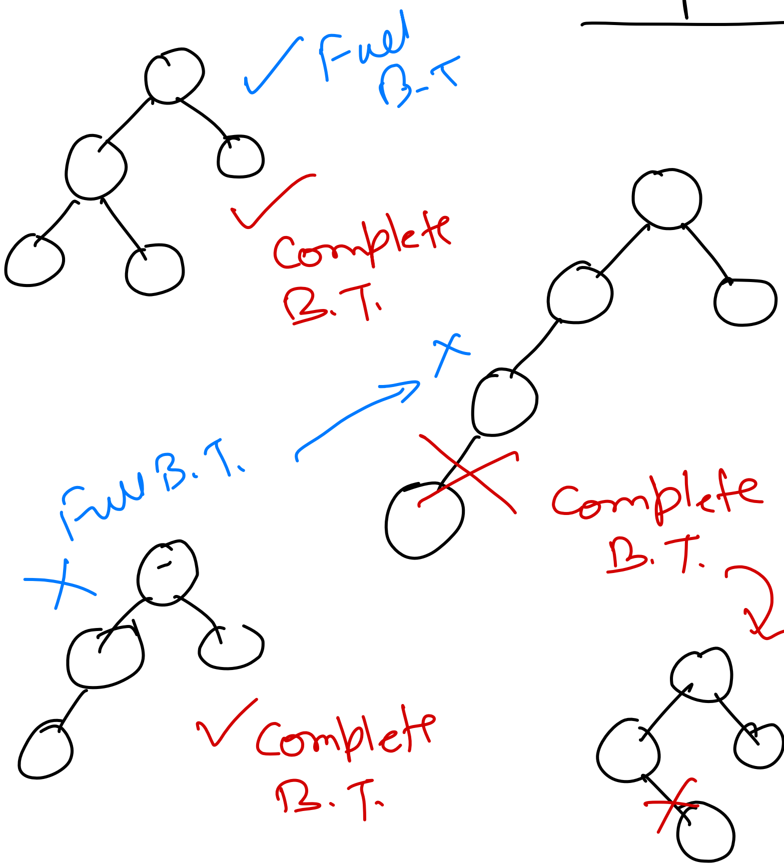
Heap Sort



Binary Tree

Complete Binary Tree

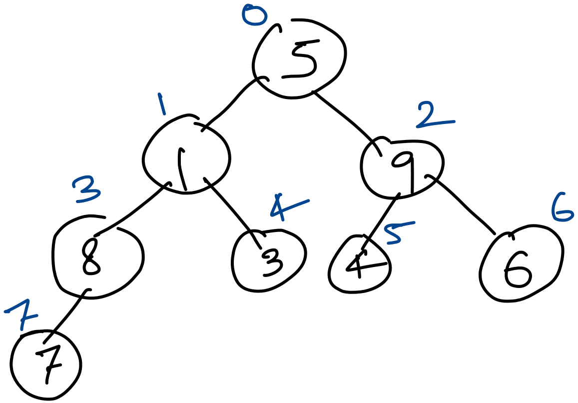
↓

1. Node on one level will have all its child before nodes at next level can have child.
2. Child will exist from left to right.

Full Binary Tree

↓

Each node will have either 0 or both children.



0	1	2	3	4	5	6	7
5	1	9	8	3	4	6	7

$n=8$

i^{th} element

left: $2i+1$

right: $2i+2$

$$n-1 = 2i+2$$

$$2i = n-1-2$$

$$i = \frac{n-3}{2}$$

$$= \frac{n}{2} - \frac{3}{2}$$

$$= \frac{n}{2} - 1$$

Last child node in tree $= (n-1)$

Last parent node in tree $= \left(\frac{n}{2} - 1\right)$

Heap Sort uses Heap data structure

Ascending order

Descending order

↳ Is a binary tree.

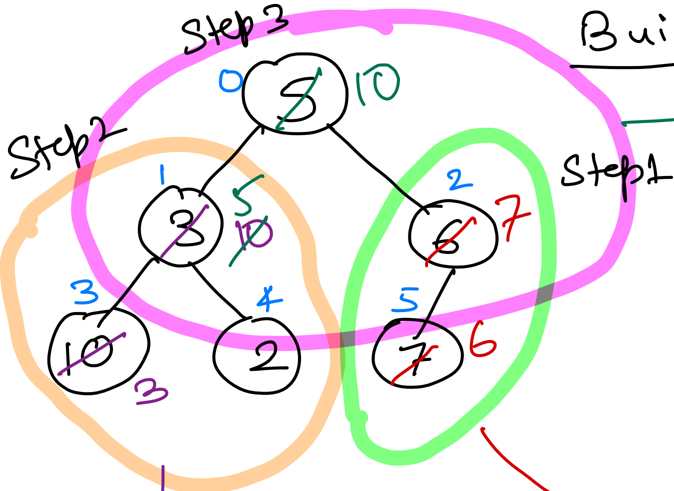
Each node satisfies Heap property.

Max Heap ⇓ Largest value is present in root node.

Min Heap ⇓ Smallest value is present in root node.

Max Heap
Each parent node value > value of its child nodes.

Min Heap
Each parent node value < value of its child nodes



Build Max Heap

Do not satisfy max heap prop

0	1	2	3	4	5
~~5~~ 10	~~3~~ ~~10~~ 5	~~6~~ 7	~~10~~ 3	2	~~7~~ 6

→ Start with last parent.

Do not satisfy max heap property.

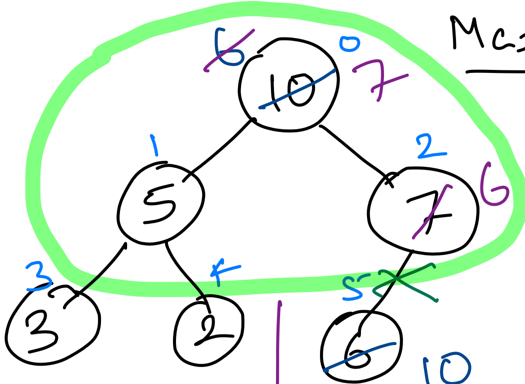
⇓

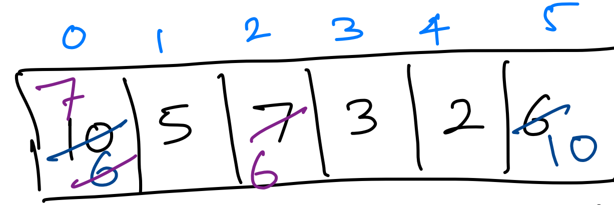
Swap parent's value with its child having largest value.

Do not satisfy max heap propert

⇓

Max Heap





element To Be Sorted = ~~6~~ 5

Not a max heap

⇓

Swap parent and max child.

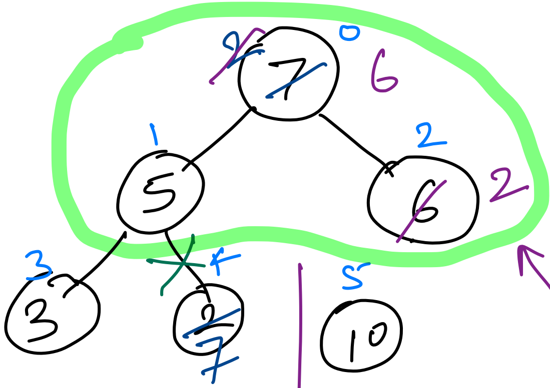
① Swap value of root and last child.

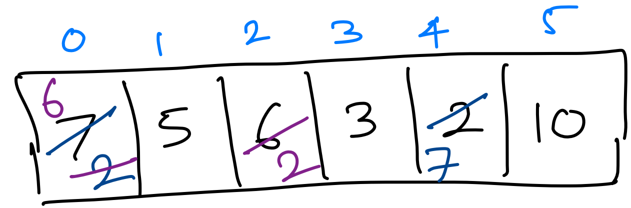
② Remove last child from heap

③ Make sure root satisfies max heap property.

⇓

Max Heap





element To Be Sorted = ~~5~~ 4

Not a max heap

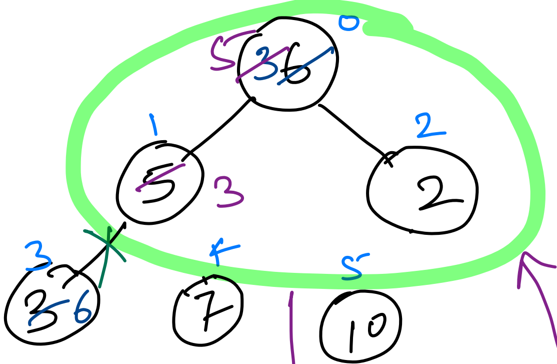
⇓

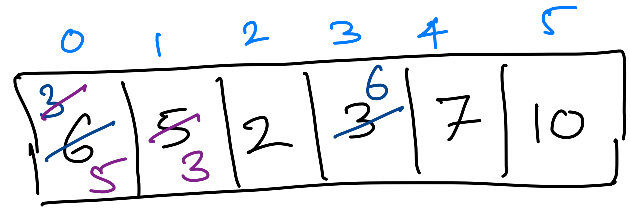
Swap root and max child value

1. Swap value of root and last child.
2. Remove last child from heap
3. Make sure root satisfies max heap property.

⇓

Max Heap





elementToBeSorted = ~~4~~ 3

Not a max head

⇓

Swap root and max child values

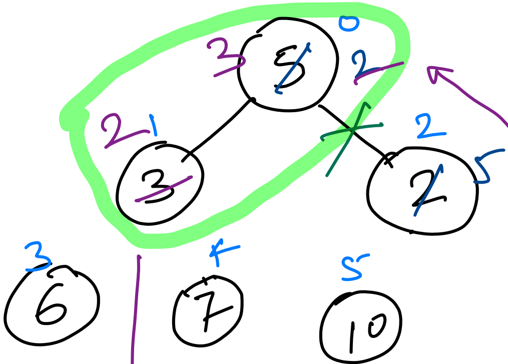
① Swap value of root and last child.

② Remove last child from heap

③ Make sure root satisfies max heap property.

⇓

Max Heap



0	1	2	3	4	5
~~2~~ ~~5~~ 3	2 ~~3~~	5 ~~2~~	6	7	10

elementToBeSorted = ~~3~~ 2

Not a max heap
⇓
Swap root and max child

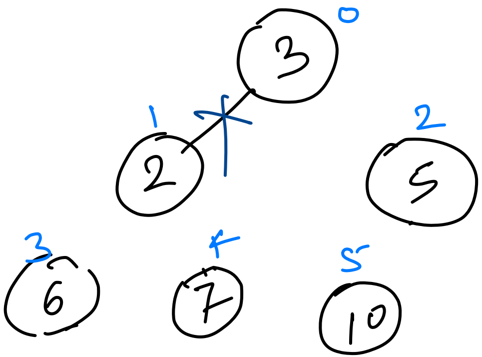
① Swap value of root and last child.

② Remove last child from heap

③ Make sure root satisfies max heap property.

⇓

Max Heap



0	1	2	3	4	5
~~3~~ 2	~~2~~ 3	5	6	7	10

elementToBeSorted = ~~2~~ 1

1. Swap value of root and last child.

2. Remove last child from heap

3. Make sure root satisfies max heap property.

Heap Sort (arr)

1. Convert input into max heap.
2. while (elementsToBeSorted > 1) do

    1. Swap value of root and last child.

    2. Remove last child from heap ⟹ Reduce elementsToBeSorted by 1

    3. Make sure root satisfies max heap property.

HeapSort(arr)

- ConvertToMaxHeap(arr, n) $\rightarrow$ $n \log_2 n$
- Set lastChildPos to n - 1
- while (lastChildPos > 0) $\rightarrow$ $n-1$ times.
  - Swap root(0) and lastChildPos values
  - if (lastChildPos > 1)
    - MakeMaxHeap(arr, 0, lastChildPos) $\rightarrow$ $\log_2 n$
  - Decrement lastChildPos by 1
- Stop

$= n \log n + (n-1) \times \log_2 n$

$= n \log n + n \log n - \log n$

$2 \cancel{2} n \log n - \cancel{\log n}$

Space $= O(1)$

Time $O(n \log n)$ $\Leftarrow$

ConvertToMaxHeap(arr, n)

- Set lastParent to n / 2 - 1
- while (lastParent >= 0) $\rightarrow$ runs $\frac{n}{2}$ times.
  - MakeMaxHeap(arr, lastParent, n) $\rightarrow$ $\log_2 n$
  - Decrement lastParent by 1
- Stop

$= \frac{n}{2} \times \log_2 n = \cancel{\frac{1}{2}} n \log_2 n$

Space $= O(1)$

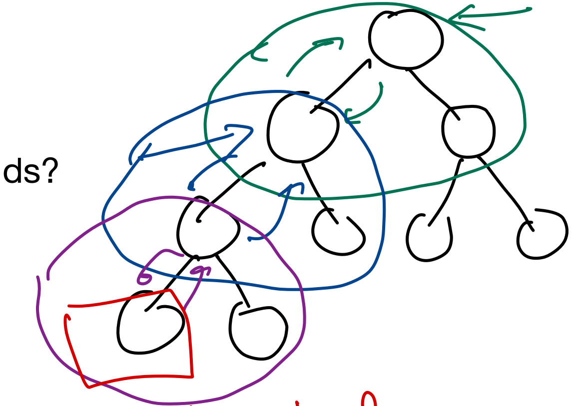
<u>Time</u> $O(n \log_2 n)$

```
MakeMaxHeap(arr, parent, n)
// Find which child has largest value
- Set maxChildPos to 2 * parent + 1
- Set rightChildPos to 2 * parent + 2
// If right child exist, is it the one with value larger of two childs?
- if (rightChildPos < n)
  - if (arr[rightChildPos] > arr[maxChildPos])
    - maxChildPos = rightChildPos

// Check if parent has value larger than the largest child
- if (arr[parent] > arr[maxChildPos])
  - Stop

// As child has larger value, swap it with parent
- Swap values at parent and maxChildPos

// Child value has changed, if its a parent node then it should still be a max heap
- if (maxChildPos is a leaf node)
  - Stop
- MakeMaxHeap(arr, maxChildPos, n)
- Stop
```



leaf -

height of tree $\parallel$ $\log_2 n$

Time = $O(h)$

$\Rightarrow$ $\underbrace{(2 * maxChildPos + 1) < n}_{}$

$\downarrow$

maxChildPos has atleast left child.

tail recursion

Space = $O(1)$ ← once we remove tail recursion

Hash Table → Efficient Searching.

Sorted Array → Binary Search ⇒ $O(\log n)$

Linked List → Linear Search ⇒ $O(n)$

Insert ⇒ $O(n)$

BST ⇒ $O(\log n)$

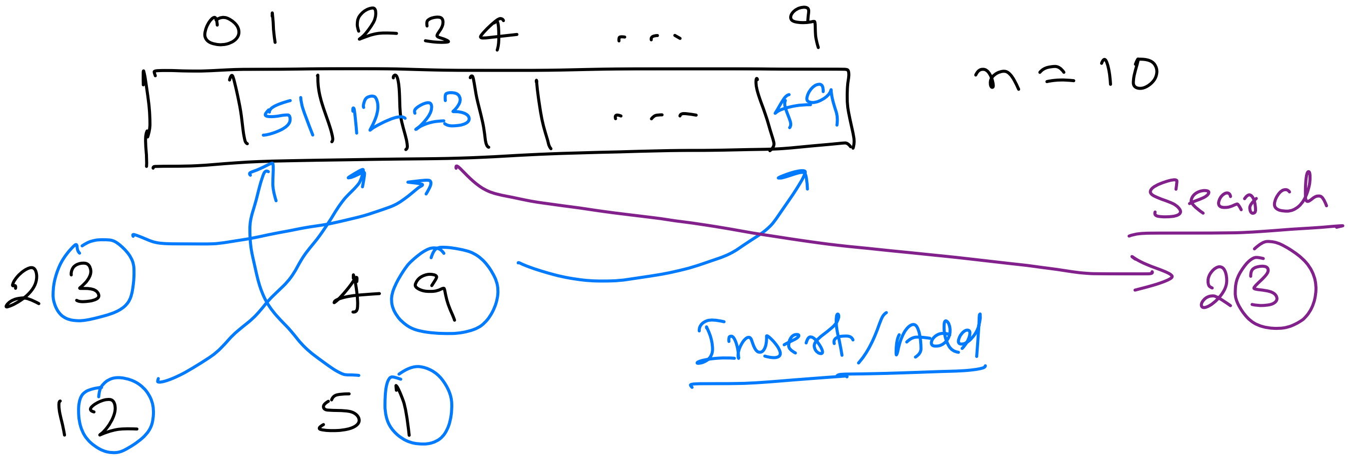
Insert ⇒ $O(\log n)$

Hash Table

→ Search $O(1)$

↳ Insert $O(1)$

Ideal Scenario



Hash Table → Is a collection of buckets.

Bucket → Place in hash table where key/value is stored.

Hash Function Key → Hash Function → bucket id

Hashing.

Hash Table using Array

Hash Function => MOD N

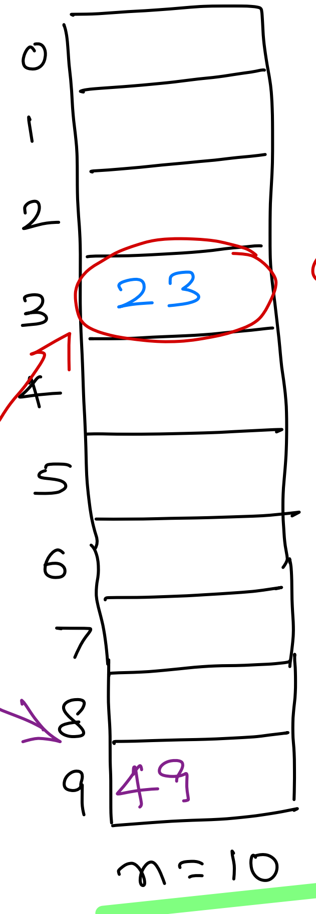
23 → $\boxed{HF}$ → 3

49 → $\boxed{HF}$ → 9

Search (23)

23 → $\boxed{HF}$ → 3

53 → $\boxed{HF}$ → 3



Collision

⇓

when multiple keys are mapped to the same bucket

Collision → Prevent, Handle

Prevent: How to avoid?

→ By using a better hash function.

1. MOD N
2. Folding ⇒ Break key into multiple parts and fold them.

$\underbrace{95}\ \underbrace{18}\ \underbrace{32} \Rightarrow$

$$\begin{array}{r} 95 \\ 18 \\ +\ 32 \\ \hline 145 \end{array}$$

③ Mid Square ⇒ Square of key and pick few digits from middle.

$$12 \xrightarrow[\text{Square}]{} 1\boxed{4}4 \to 4$$

<u>String to number</u> → add ASCII values of each character in string.

<u>JAVA</u> each class hashCode()

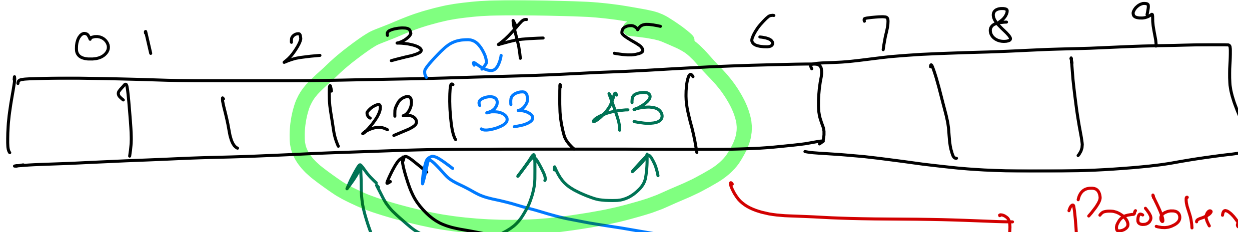
Hash Table

- Static Hash Table.
  - Symbol Table
  - Table of Keywords.
- Dynamic Hash Table
  - JAVA Util class.

Handle Collision

- Chaining ⟹ allow multiple keys to be stored in a bucket
  - Array
  - Linked list
  - Search Tree.
- Probing.
  - Linear
  - Quadratic

Linear Probing



insert(23)

23 → $\boxed{\text{HF}}$ → 3

insert(33)

33 → $\boxed{\text{HF}}$ → 3

insert(43)

43 → $\boxed{\text{HF}}$ → 3

Problem with linear Probing

⇓

Clustering (Primary)

In case of collision, we linearly scan hash table for next empty bucket and store new key there.

id + i

⇓

0, 1, 2 ....

we do probing n times, and if no bucket found ⟹ resizing A hash table.

↳ re hashing of existing keys.

<u>Load Factor</u> → How full/empty hash table can be before we ~~need~~ need to resize it.

<u>Quadratic Probing</u> → Fixes primary clustering.

$$id + \underline{c}x^2 + \underline{b}x + \underline{d}$$

0, 1, 2, ...

some constants

$c = 1 \quad b = 0 \quad d = 0$

$i'd + x^2$