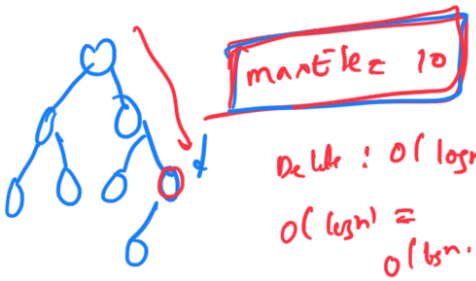
HEAPS -2

	Insertion	Deletion Max()	getMax()
Heap	O(logn)	O(logn)	O(1)
B.B.S.T	O(logn)	O(logn)	O(logn)/O(1)

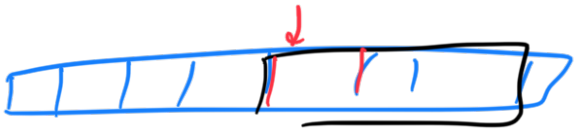
B.B.S.T → H = O(logn)



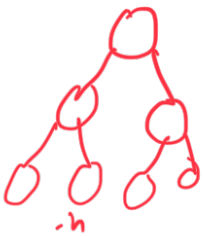
Delete : O(logn)
O(logn) = O(h).

Advantages of Heaps

1) Heap is implemented using arrays.
Better locality of reference and operations
are cache friendly



2

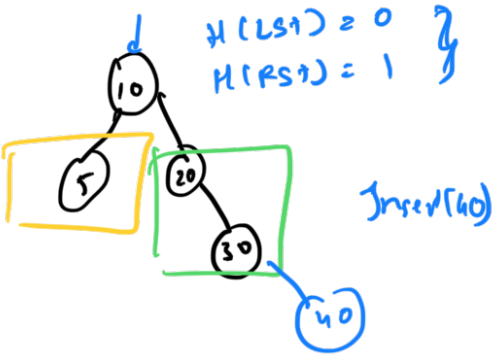


2) B.B.S.T have a higher constant 'h'
the Time Complexities

T.C: k.logn

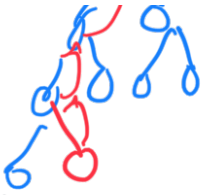
Balancing the Tree

O(logn)



3) T.C of building Heap: O(n)

" " B.B.ST = O(nlogn)

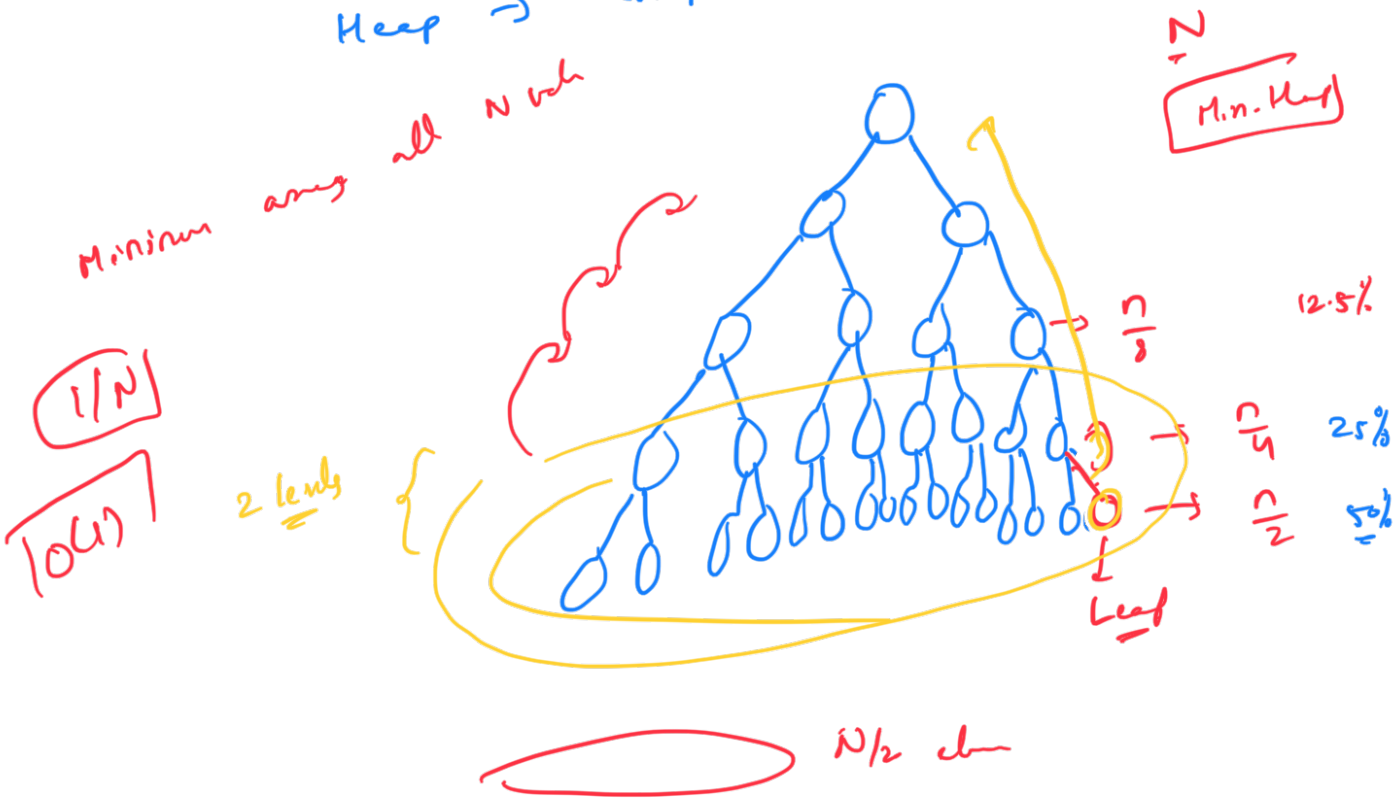


4) Heap does not require extra space for storing pointers.

5) Average insertion time is O(1) in heap!

Heap → Complete BT

Minimum array all N values



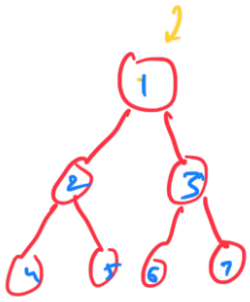
Disadvantage

1) Searching : O(log n) / BBST

2) BBST is an ordered D.S ↙ Floor, Ceiling
[Print In sorted order, Inorder Successor]

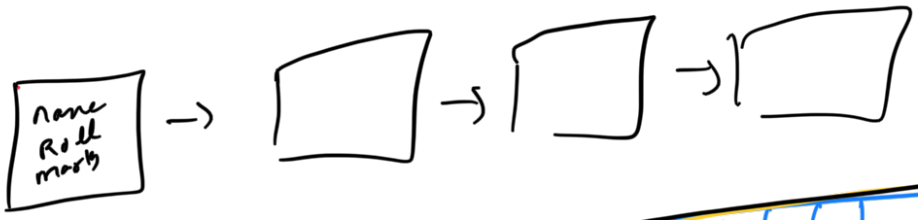
Priority_Queue }

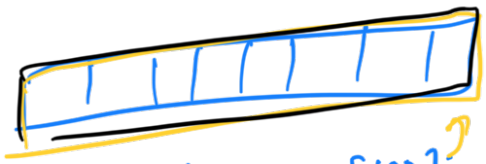
Heap P }



Duplicate

1) push(x) → add()
2) pop() → remove() / poll()
3) top() → peek
4) empty()
5) size()





Student array[100]

class student () {
string Name;
int Roll;
int marks;
char gender;
}

Max Heap :
priority_queue <int> PQ;

Min Heap :
priority_queue < int, vector<int>, greater<int> >
↓
Comparator.

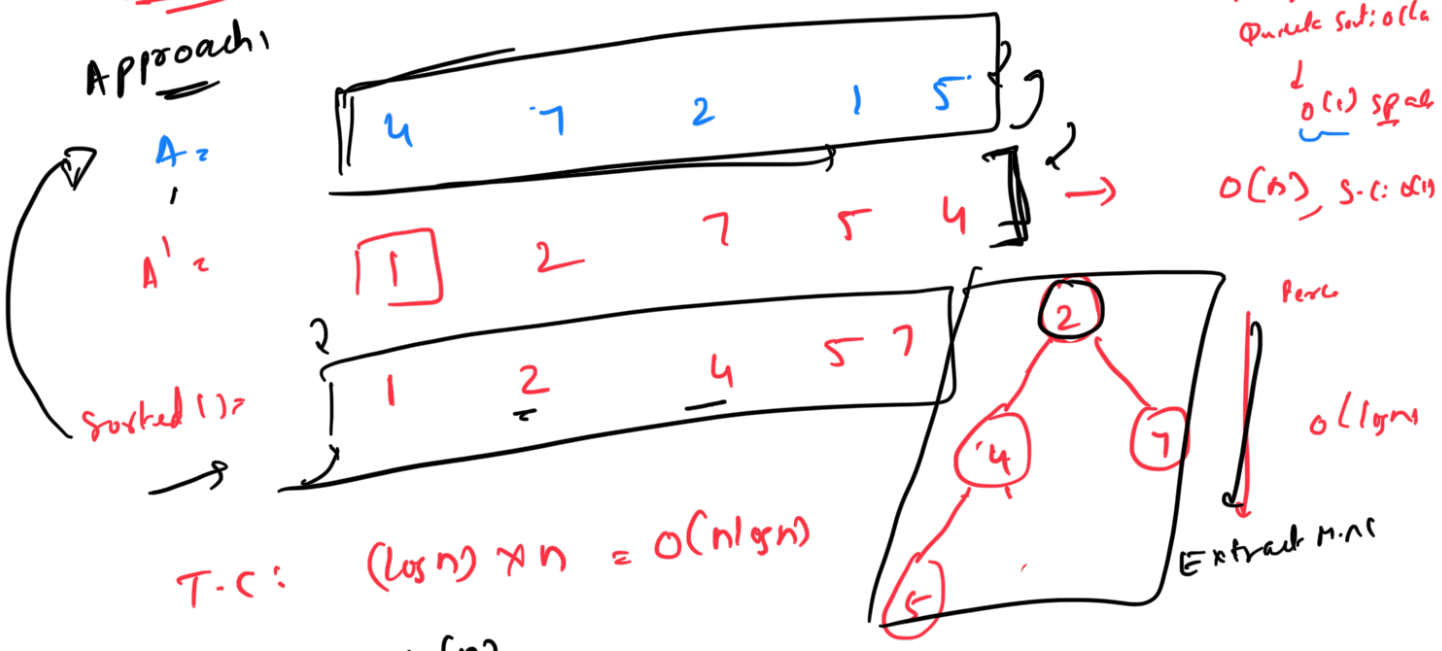
```
typedef struct node{
  int key;
  int val;
}Node;

struct compare{
    bool operator()(Node const& n1, Node const& n2){
        return n1.val < n2.val;
    }
};
priority_queue<Node, vector<Node>, compare> pq;
```

Heap sort

Question:

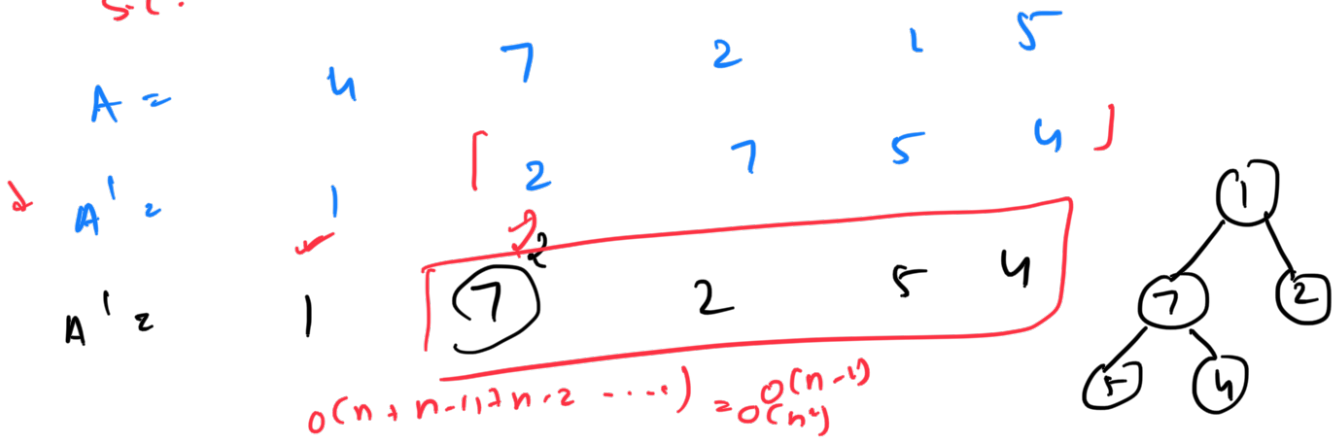
Approach:



T.C: $(\log n) \times n = O(n \log n)$

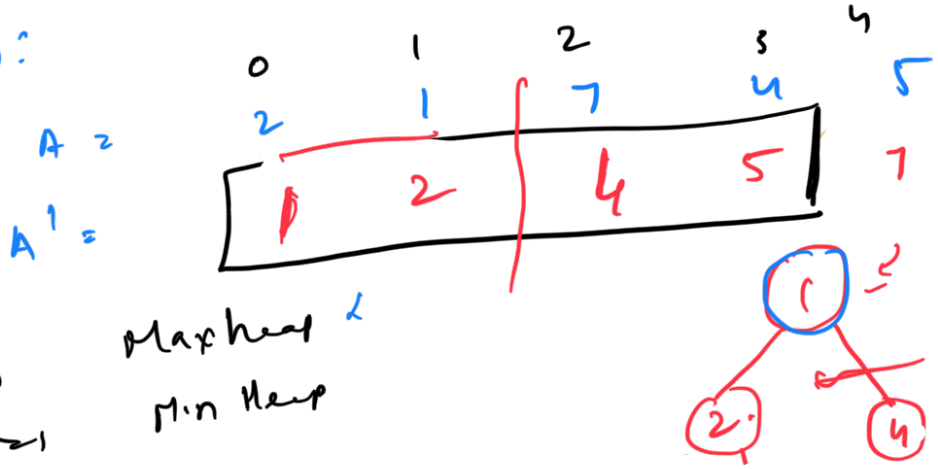
S.C: O(n)

O(nlogn)
S.C: O(1)
Merge Sort: O(log
Quick Sort: O(lo
O(1) space
O(n) S.C: O(n)



$O(n + n-1 + n-2 + \dots) = O(n^2)$

Approach:



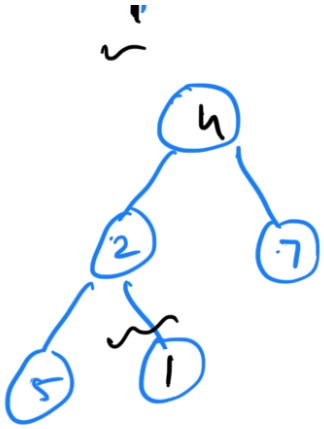
Increasing => Min Heap

Decreasing => Max Heap

A' = 4 2 7

T.C: O(nlogn)

S.C: O(1)



Question: k^{th} Smallest element in a stream of numbers

$N = 10^5, k = 10^4$

K=3	0	1	2	3	4	5	6	7	8	9	10
A =	10	12	8	14	4	20	16	7	2	9	13
	-1	-1	12	12	10	10	10	8	7	7	7

↓8 ↓10 ↓12

Brute force:

N [-> whenever you get a new element, sort it -> O(nlogn)

-> A[k-1]: O(1)

T.C: $O(n^2 \log n)$

Approach 2:

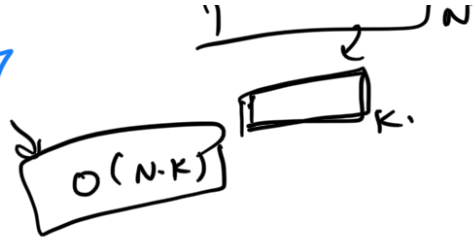
8 10 13 17

-> Always maintain a sorted array

O(n) [-> When a new element comes, insert in the sorted position

N operations

T.C: $O(N^2)$



Approaches :

k = 5

A = 3 5 1 6 9 → Max elmt

{ → Always maintain a set of K elements
→ Return the max of this

Min elmt / Max / K^{th} → Heaps

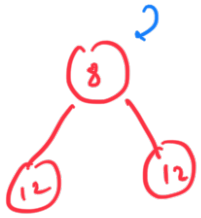
→ Min Heap / Max Heap?

→ Min Heap

temp = 12

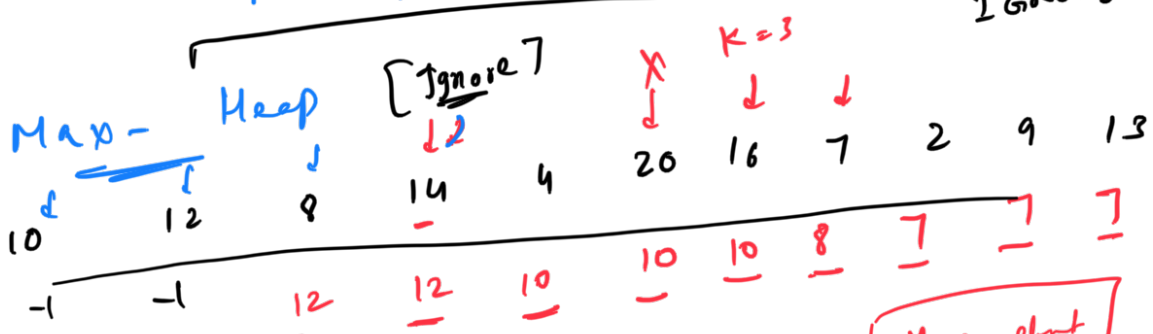
k=3

A = 10 12 8 14 4 20 26



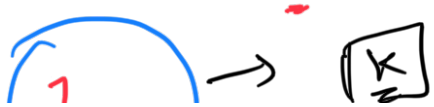
Min Heap

→ Max-Heap [Ignore]



Max-elmt

O(log n)



→ Delete
→ Insert New
↓
O(log n)

Max-Heap
k elements

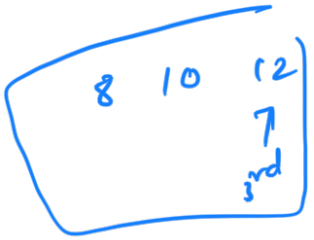
T.C: O(n x log k)

S.C: O(k)

log k + log k.

Heap[0]

12 → 3rd smallest
↓
4th smallest

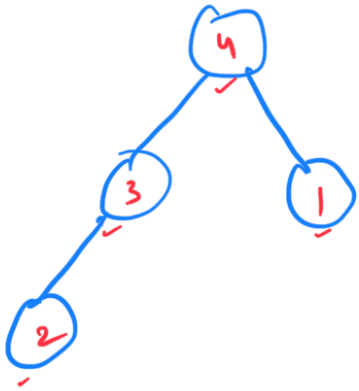


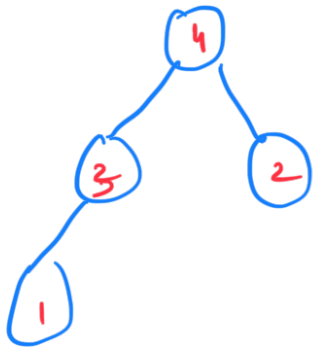
3rd smallest
3rd element in the sorted / ascend

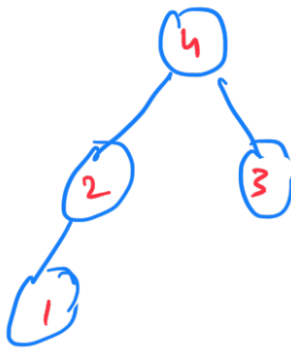
Question: No. of ways to build a max Heap
→ unique Element

N = 4 [1, 2, 3, 4]

Ans = 3 (4-1)

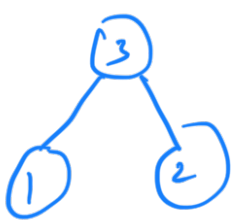






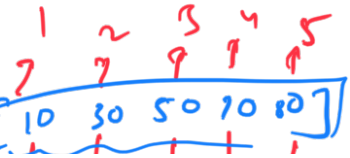
N = 3 → [1, 2, 3]

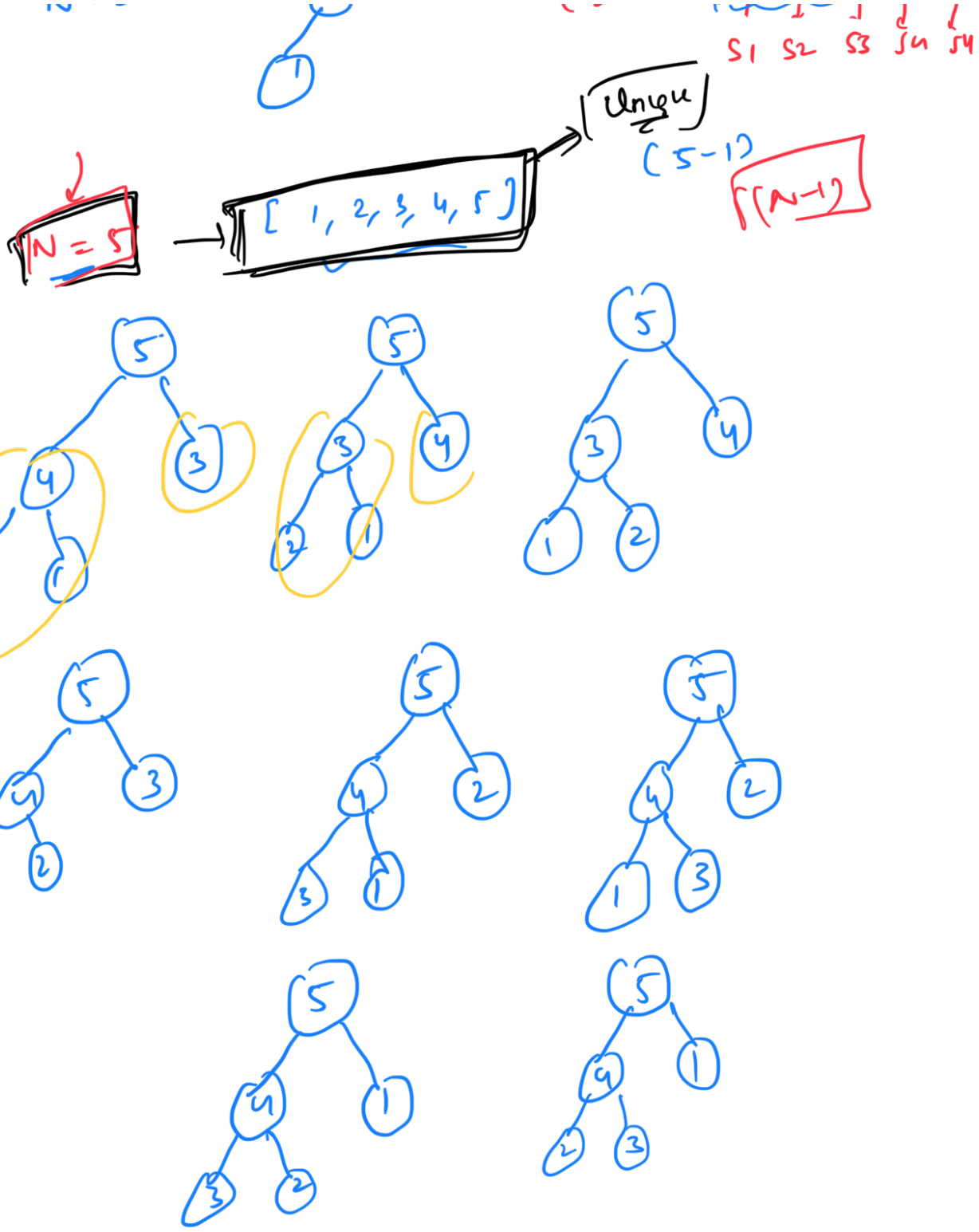
(3-1)



N = 2 [1, 2]

(2-1)





OBSERVATIONS:

1) Structure of the heap would remain same

2) No. of max heaps will not depend on value but rather only n.

3) there is only 1 choice. Elements of

q) Any relationship between LST & RST?

s) L + R = N-1 ✓

Assumption: Let ways(N) be the No of max heaps we can form using N elements.

N-1 → L, R

ways(N) = $^{N-1}C_L$ × ways(L) × ways(R) ✓

$^{N-1}C_R$

N-1 elements

L + R = N-1

Root, LST (L), RST (R)

$^{N}C_R = {}^{N}C_{N-R}$

$^{N-1}C_L = {}^{N-1}C_{N-1-L} = {}^{N-1}C_{N-L-1}$

L + R = N-1

R = N-1-L

$^{N-1}C_L$

1 Combination

N = 7

LST: [1,2,3]

T1 or T2 | 7 | T3 or T4

RST: [4,5,6]

[1,2,3,4,5,6,7]

N-1 = 6

L + R = 6

L = 3

R = 3

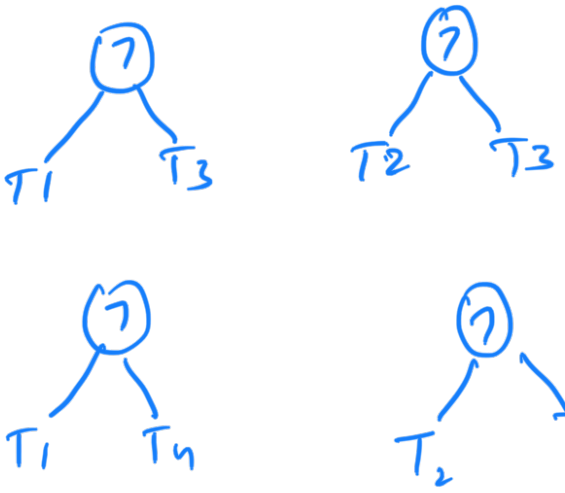
$^{6}C_3$ = 6×5×4 / 3×2×1 = 20

2 × 2 = 4

3, 3, 6, 6

T3 T4

N=5



h

L and R?

$H = \text{floor}(\log n)$

$\log_2 4 = 2$

$\log_2 3 \; [1, \ldots]$

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

$\left(\frac{2^{h-1}-1}{2}\right)$

N = 3 ⇒ 1

N = 4 ⇒ 2

N = 5 ⇒ 2

N = 6 ⇒ 2

N = 7 ⇒ 2

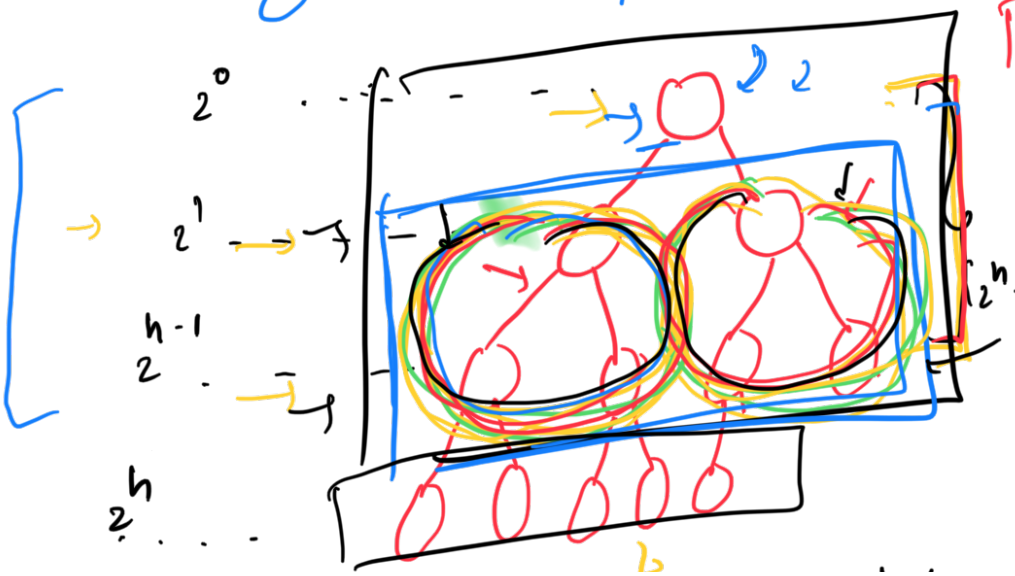
N ≥ 1, 3

$\left(\frac{2^h - 1 - 1}{2}\right)$

N

$= \frac{2^h - 2}{2}$

$= [2^{h-1} - 1]$



$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$

$2^h - 1 - 1$

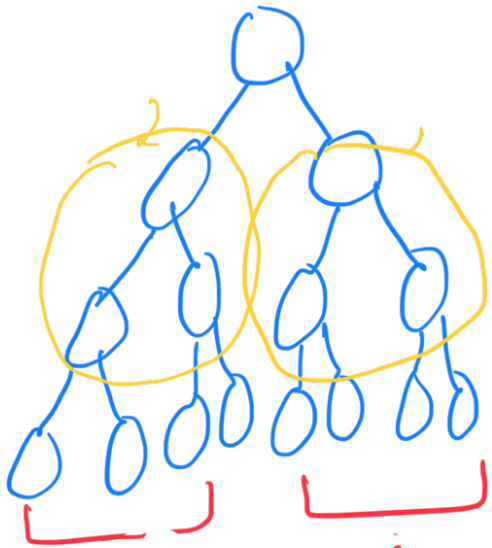
$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = \frac{1(2^h - 1)}{2-1} = 2^h - 1$

#lastLevel = $N - (2^h - 1)$

$\frac{2^h - 2}{2} = 2^{h-1} - 1$

#Nodes_LST = $2^{h-1} - 1 + x$

#Nodes_RST = $2^{h-1} - 1 + y$



2^{h-2} — — —

2^{h-1} — — —

2^h — — —

h

$2^h = 8$

$\frac{2^h}{2}$

6

5

4

h

h

2

1

4

4

4

if $\left(\frac{2^h}{2}\right)$

#Last Level = $N - (2^h - 1)$

if($N - (2^h - 1) \leq 2^h / 2$)

$x = N - (2^h - 1)$;

$y = 0$

else {

$x = 2^h / 2$.

$y = [N - (2^h - 1)] - x$

[i] = -1;

dp

-1 -1 -1 -1 -1 -1

};

dp[N] = {-1};

int ways (N) {

if (N == 0) return 0;

if (N == 1) return 1;

if (dp[N] != -1) ;

h = floor(log n);

L = $2^{h-1} - 1$;

R = $2^{h-1} - 1$;

if (N - $(2^h - 1)$ $\leq$ $2^h/2$) {

L = L + (N - $2^h - 1$);

}

else {

L = L + $2^h/2$;

R = R + N - $(2^h - 1)$ - $2^h/2$;

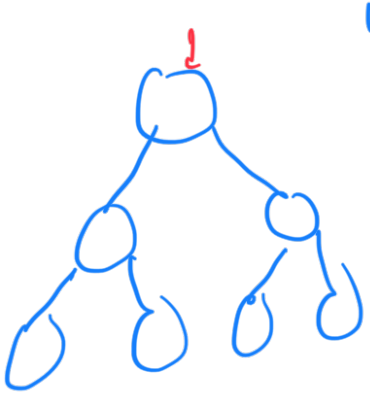
}.

dp[N] = N_C^1 x ways(L) x ways(R);

return dp[N];

}

Dynamic Programming



N = 7

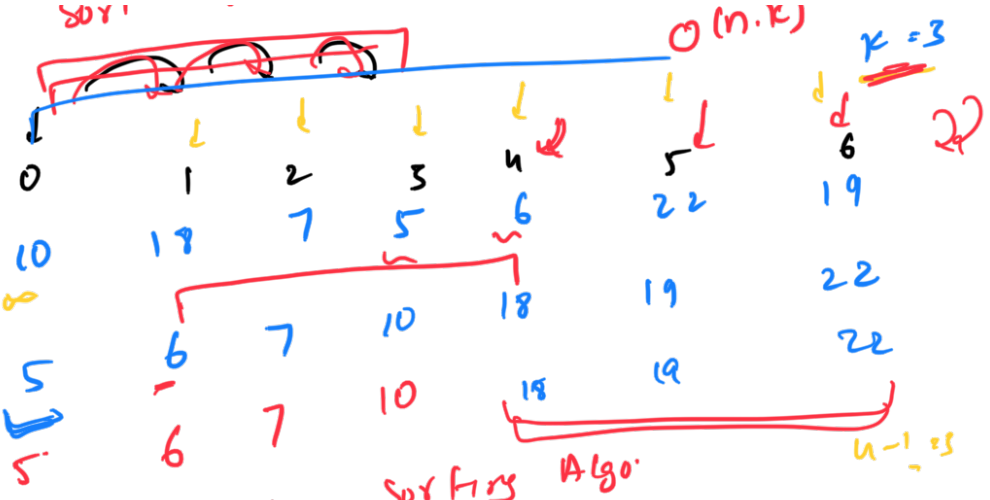
L = 3

R = 3

ways (7) = $^6C_3$ x ways(3) x ways(3).

K-sorted Array

Question:



Approach1: Use any sorting Algo.

T.C: O(nlogn)

Approach2:

{-k, -k+1 ... 0, 1, 2, ... k}

Min Heap

Size = (k+1)

-> 1 iteration: -> Deleting Min Element → logk

-> Insertion → logk.

n times

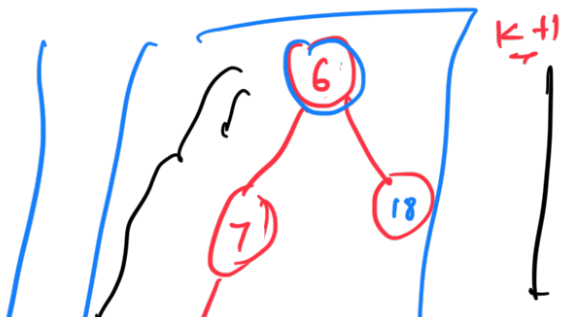
T.C: O(nlogk)

k == n

nlogn

A = 10 18 7 5 6 22 19

Sort(A) = 5

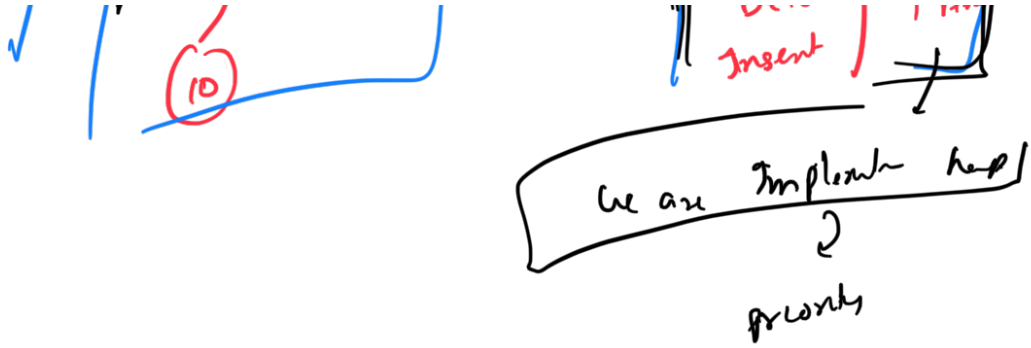


K=3

Min-Heap

Percolate Down

Delete




```
typedef struct node{
  int key;
  int val;
}Node;

struct compare{
    bool operator()(Node const& n1, Node const& n2){
        return n1.val < n2.val;
    }
};
priority_queue<Node, vector<Node>, compare> pq;
```

Kth Smallest Element

```
priority_queue<int> pq;
vector<int> ans;
for(int i = 0; i < k; i ++){
     pq.push(a[i]);

     if(i == k-1) ans.push_back(pq.top());
     else  ans.push_back(-1);
}
for(int i = k; i < n; i++){
     if(a[i] < pq.top()){
          pq.pop();
          pq.push(a[i]);
     }
     ans.push_back(pq.top());
}
return ans;
```

No. of max heaps using N elements

```
dp[n] = {-1};
int numMaxHeaps(int n){
      if(n == 0) return 0;
      if(n == 1) return 1;

      if(dp[n] != -1) return dp[n];

      h = floor(logn);
      num_lastlevel = n - (2^h - 1);
      L = 2^(h-1) - 1;
      R = 2^(h-1) - 1;
      if(num_lastlevel <= 2^(h-1))
            L += num_lastlevel;
      else{
            L += 2^(h - 1);
            R += num_lastlevel - 2^(h - 1);
      }
      dp[n] = n-1CL * numMaxHeaps(L) * numMaxHeaps(R);
      return dp[n];
}
```

Sort a nearly sorted array

```
vector<int> kPlaces(vector<int> a, int k){
      n = a.size();
      // Initialize a min heap
      priority_queue<int, vector<int>, greater<int>> pq;

      // Push first k+1 elements
      for(int i = 0; i <= min(k, n-1); i++){
            pq.push(a[i]);
      }
      for(i = k+1; i<n; i++){
            ans.push_back(pq.top());
            pq.pop();
            pq.push(a[i]);
      }

      while(!pq.empty()){
            ans.push_back(pq.top());
            pq.pop();
      }
      return ans;
}
```