Heap Basics-

Raj
DATE / /
PAGE

greater<pair<int, char>>

(max heap)

① • Making Priority Queue with storing a pair

priority-queue <pair<int, char>, vector<pair<int, char>>> ?;

container

↳ * gives the largest element at top (first)

• In leetcode

ListNode temp = object

ListNode* temp = pointer

* Sorting is done on basis of first number

Heaps

Raj

DATE / /

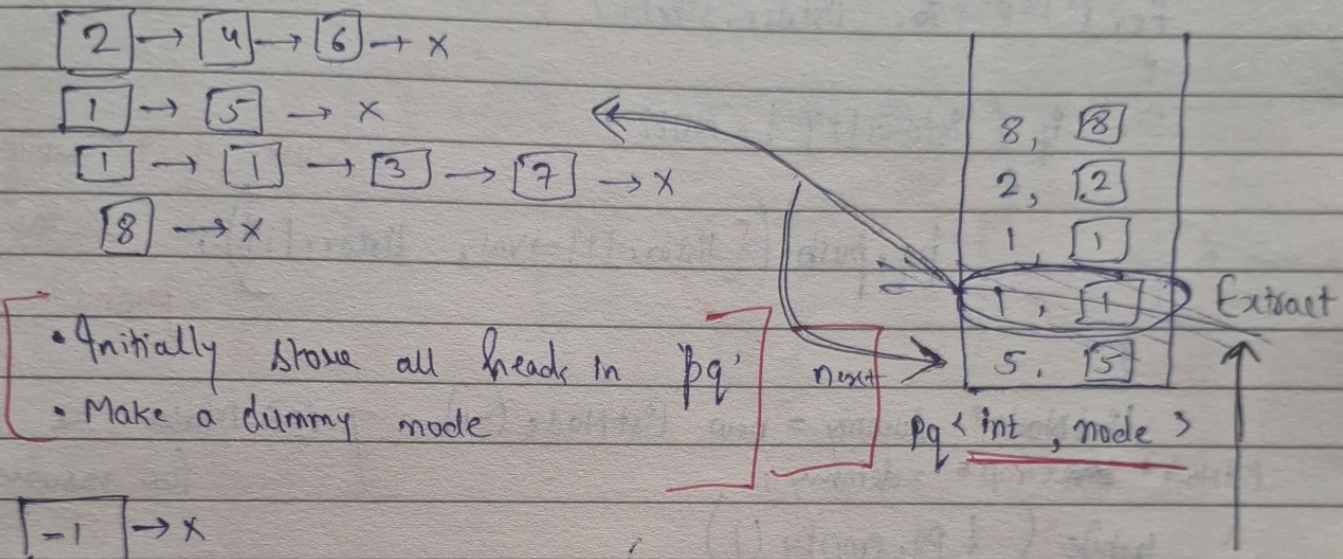
PAGE

Q Merge K sorted linked list

Input : [1,4,5] [1,3,4] [2,6]

output: [1,1,2,3,4,4,5,6]

Priority queue (min)



- Initially store all heads in 'pq'
- Make a dummy node

dummy Node → -1 → X

↑ temp

(1) • Extract the minimum element (FIRST ELEMENT) from min heap

-1 → 1

↑ temp

(2) and ~~add~~ add the next node of recently added node to priority Queue.

Code ***

all be same

priority_queue < pair < int, ListNode* >, vector < pair < int, ListNode* >>, greater < pair < int, ListNode* >>>

① Add all the head, head node to min heap

```
for ( i=0 to listarr.size() )
{
    if ( listarr[i] != NULL )
    {
        pq.push ( { listarr[i]->val, listarr[i] } );
    }
}
```

→ use { } while pushing 2 things by .push() fxn

② Make a LL for answer

```
ListNode* dummy = new ListNode(-1)
ListNode* temp = dummy;
while ( !pq.empty() )
{
    pair < int, ListNode* > p = pq.top(); // take smallest element (top)
    pq.pop();                             // remove it
    ListNode* x = p.second;

    temp -> next = x;        Join it
    temp = temp -> next;     increment temp   } traversal

    if ( x -> next != NULL ) pq.push ( { x -> next -> val, x -> next } )
}
```

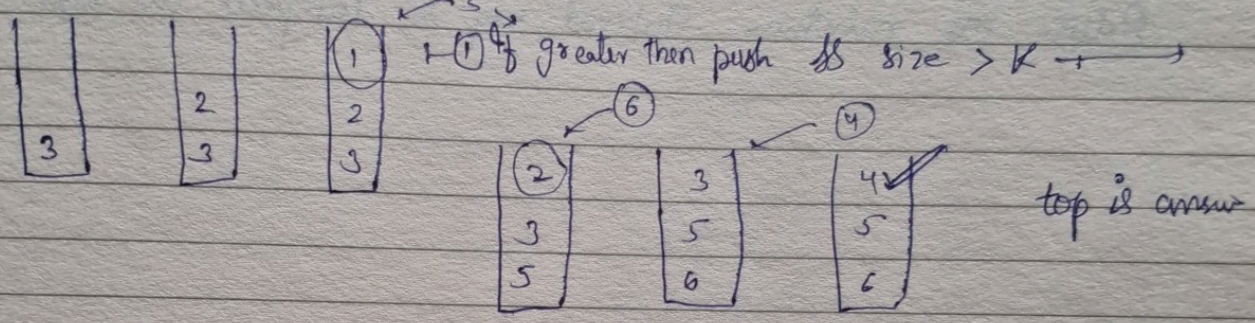
push next node of recently added node

return dummy -> next;

K^{th} largest element

- Make a min heap of size k
- K^{th} element is the K^{th} largest

eg: [3, 2, 1, 5, 6, 4] k = 3 output : 4



- When heap size >= k
  - if upcoming element is more than top
    - pop top
    - push this element

```
// Min heap
priority_queue <int, vector<int>, greater<int>> pq;
for (i=0 to n)
{
    if (pq.size() < k)      pq.push(arr[i])
    else if (pq.top() < nums[i])
    {
        pq.pop()
        pq.push(num[i]);
    }
}
return pq.top
```

Keep it of size "k"

HEAPS

Raj
Page:
Date: / /

#TYPES

Max Heap

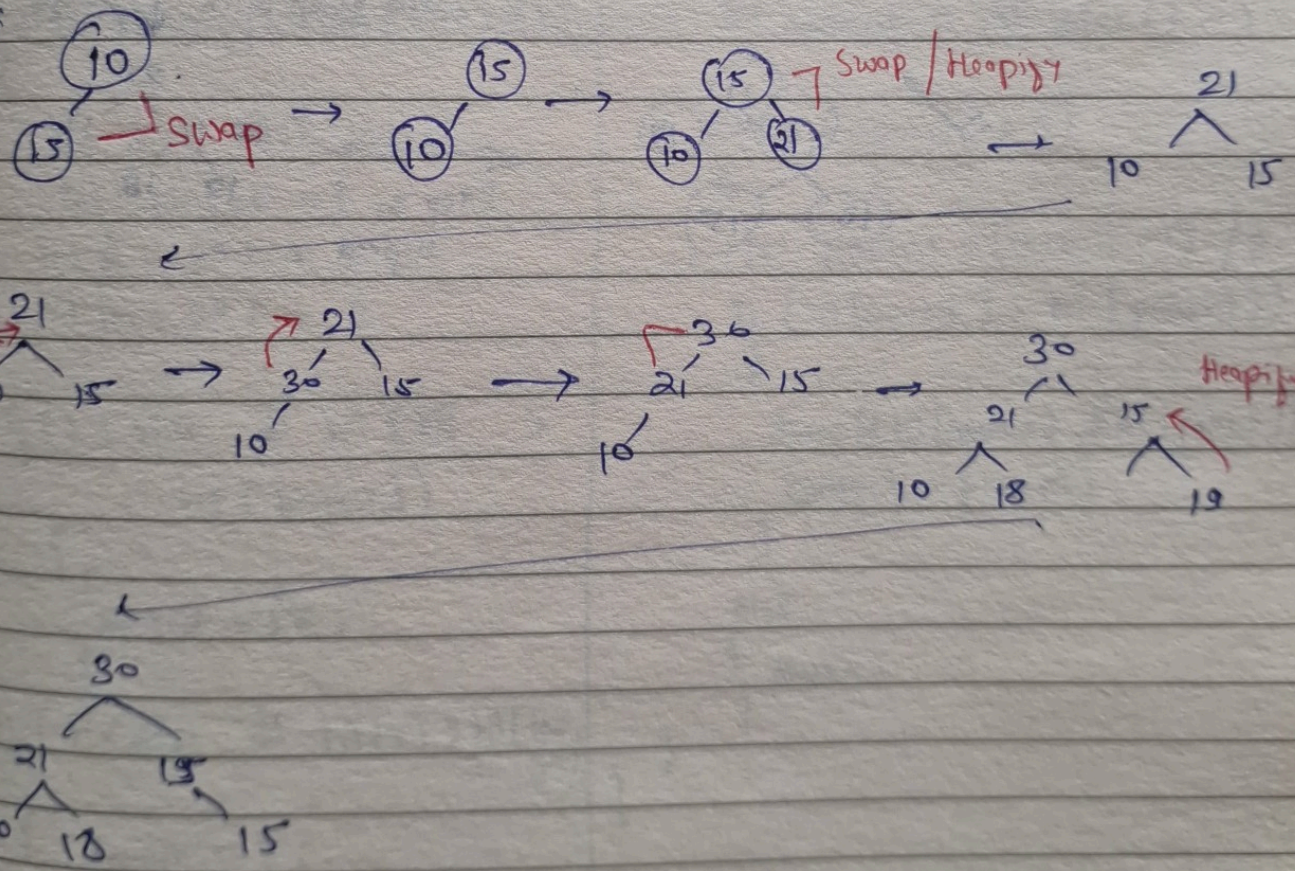
Parent is larger than all elements of it's subtree

Min Heaps

" Smaller ...

Array to MAX Heaps (conversion)

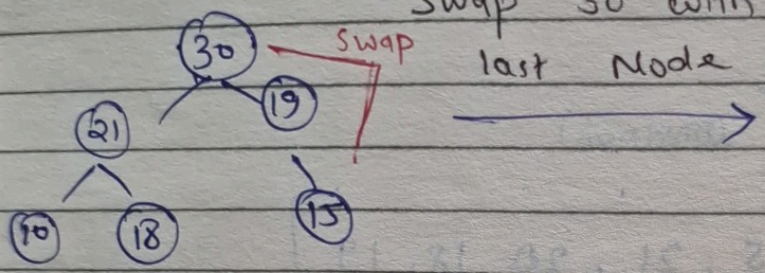
array: [10, 15, 21, 30, 18, 19]



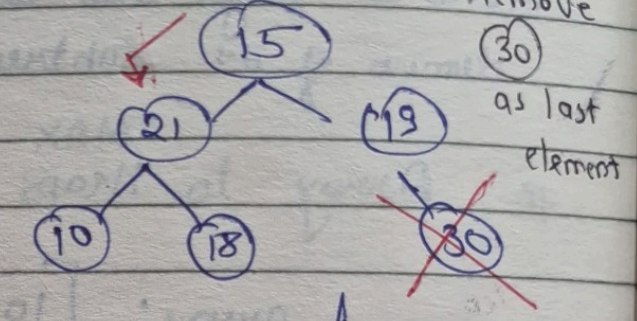
Raj
Page:
Date: / /

Popping elements

eg: Pop : 30

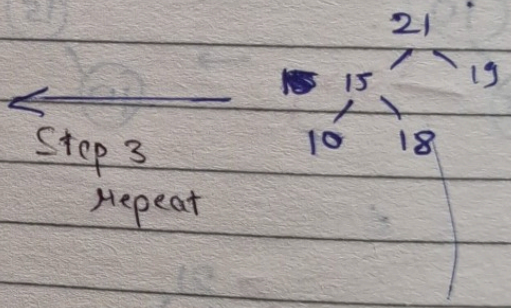


1. Swap 30 with last Node

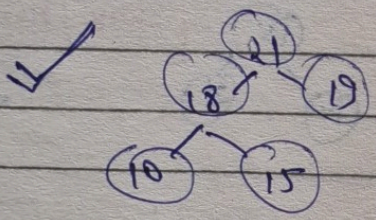


2. Remove 30 as last element

3. compare 15 with its child and swap with Bigger one



Step 3 Repeat



Raj
Page:
Date: / /

Heap: Insertion & Pop & Heapify.

```
void Swap ( int *a, int *b)
{
    int temp = *a;
     *a = *b
     *b = temp
}
```

```
void Heapify(vector<int>& hT, int i)
{
  int size = hT.size();
  int largest = i;
  int l = 2*i + 1
  int r = 2*i + 2

if(l < size && hT[l] > hT[largest])
  {
    largest = l;
  }
if(r < size && hT[r] > hT[largest])
  { largest = r;
  }
if(largest != i)
{
  swap(&hT[i], &hT[largest]);
} heapify(hT, largest);
```

→ for remaining array

```
void insert(vector<int>& hT, int newnum)
{
  hT.push_back(newnum);
  int current = hT.size() - 1

  while (current > 0)
  {
    int parent = (current - 1) / 2
    if (hT[current] > hT[parent])
    {
      Swap(&hT[current], &hT[parent]);
      current = parent;
    }
    else
    { Break;
    }
  }
}
```

Page: Raj
Date: / /

```
void delete (vector<int>& hT, int num)
{
    int size = hT.size();
    int i;

    for (i=0; i<size(); i++)
    {
        if (num == hT[i])        break;
    }

    swap (&hT[i], &hT[size-1]);
    hT.pop_back();

// Update size after popping
    size = hT.size();

// heapify from current index to adjust rest of heap
if (i<size)
    { heapify (hT, i);
    }
```

Raj
Page:
Date: / /

Concept of Insertion | deletion | Heapify | Heapsort

① Insertion

1. Insert at end
2. Compare to parent until no parent exist

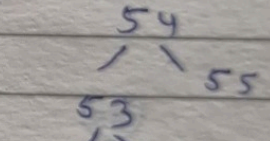
3

Node: i^{th} index
left child: $2*i$ index
right child: $(2*i+1)$ index
Parent: $(i/2)$

② Heapify

void

input: [0 | x | 54 | 53 | 55 | 52 | 50]



Leaf nodes are $\left(\frac{n}{2}+1\right)$ to (n) index

Page: Raj
Date: / /

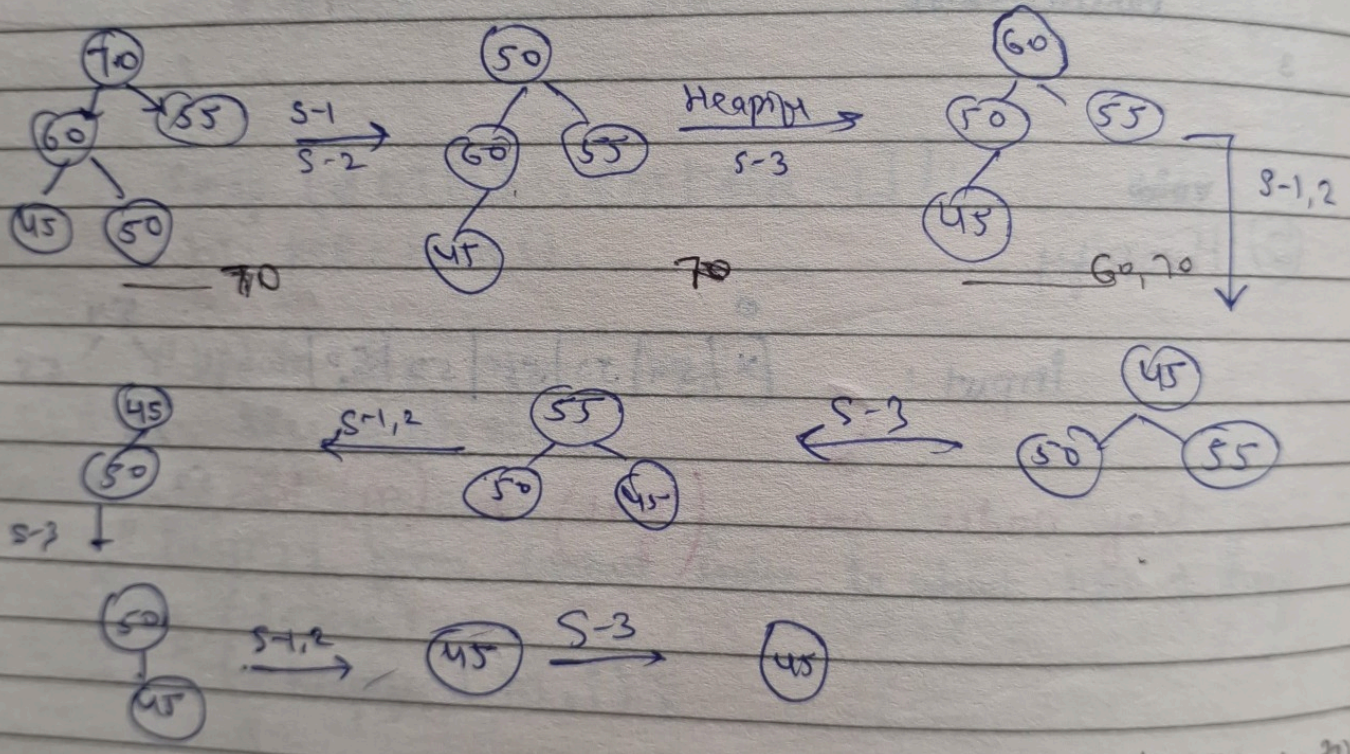
Heap Sort

eg:

X	70	60	55	45	50

Size = 5

Algo :- 1) Swap first and last, 2) size--
3) Root at correct posn = Heapify



∴ 45, 50, 55, 60, 70

$O(n \log n)$

Raj
Page:
Date: / /

```
void heapSort (int arr[], int n)
{
    int t = n
    while (t > 1)
    {
        swap( arr[t], arr[1]);
        size--;

        Heapify (arr, size, 1);
    }
}
```

Priority Queue

→ You can make Max / Min Heap

Page: Raj
Date: / /

Q Numbers are getting inputed and we have output median after each input

Sol) odd terms = middle term
even terms = avg. of middle terms

data should be SORTED

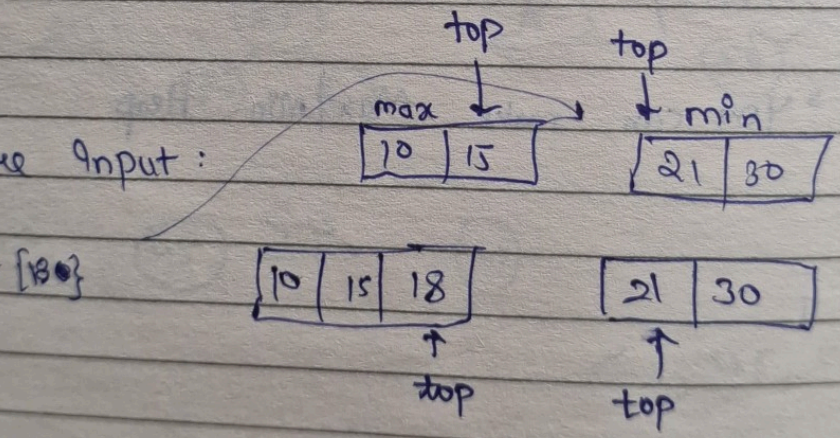
- Now instead of swapping at each input

↓

HEAPS :-

- Keep one Max Heap and Min Heap
- Partn array in 2 parts before input
- Insert such that the difference in size of min/Max heap is 1

eg:



(case-1) If size of Max Heap & Min Heap is not equal = Top of Larger size Heap.

(case-2) Size equal then median is avg of Both Tops

Raj
Page:
Date: / /

How Max / Min heaps are inputed

1) If size equal for Both. - Input acc. to top of heaps

2) Unequal size - [10|15|18] [21|30] ⇒ [10|15|17] [18|21|30]
↓17

code.

min heap

priority-queue <int, vector<int>, greater<int>> pqmin
priority-queue <int, vector<int>> pqmax

max heap

Unequal size for Both Heaps

```
Void insert (int x)
{
if(pqmin.size() = pqmax.size())
 {
  if(pq.max.size()==0)
   {
    pqmax.push(x);
    return;
   }
  if(x<pqmax.top())
   { pqmax.push(x);
   }
 else
  { pqmin.push(x);
  }
}
```

```
else {
  if(pqmax.size() > pqmin.size())
   {
    if(x>=pqmax.top())
     {
      pqmin.push(x);
     }
    else
     {
      int temp = pqmax.top();
      pqmax.pop();
      pqmax.push(x);
      pqmin.push(temp);
     }
   }
  else {//same for pqmin
  }
```

Page: Raj
Date: / /

```
double findMedian ()
{
    if (pqmax.size() == pqmin.size())
    {
        return (pqmax.top() + pqmin.top())/2;
    }
    else if (pqmax.size() > pqmin.size())
    {
        return pqmax.size();
    }
    else
        return pqmin.size();
```

Merge 'K' Sorted Arrays. ✗ optimal

1) Create min heap of pairs
   Pairs (value, array no.)
2) Insert {first element, array no.} of all sorted array into min heaps
3) We will pop element from minheap and store in answer array
   Insert the next element of sorted array into minheap

Pair (array number , current index of array of min heap)

Raj
Page:
Date: / /

4) We also need to track indices of element from which array the comes from

eg:

Array 1: [1 | 4 | 7]

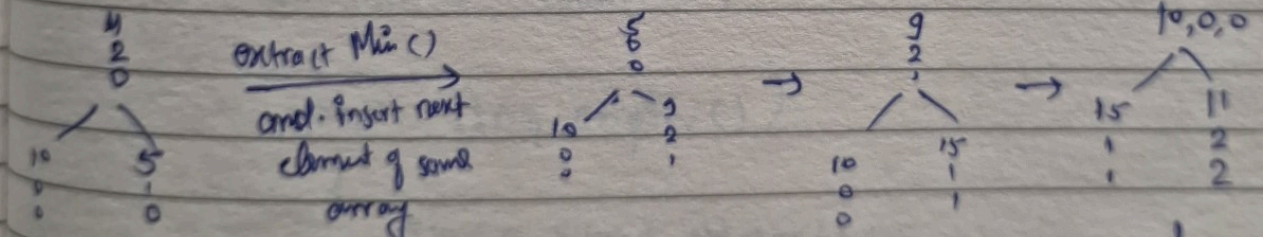
Array 2: [3 | 5]

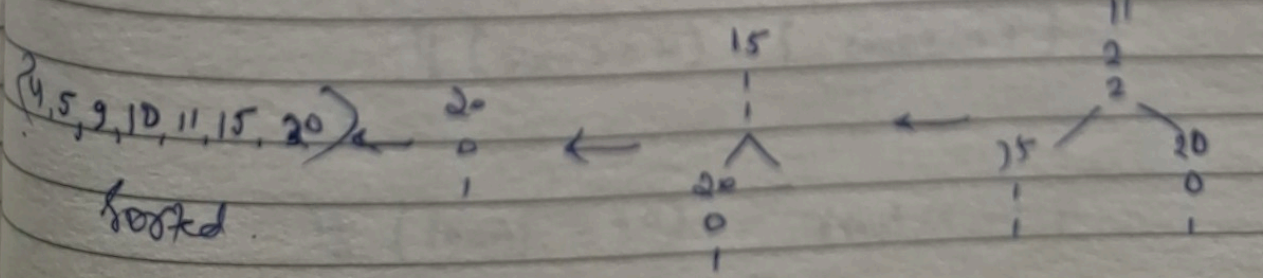
Array 3: [2 | 6 | 7]

{1,1}, {2,3}, {3,2} {4,1}

ex. arr[][] = { 0 <10, 20, 30>, → {30, 0, 2}
<5, 15>,
<4, 9, 11> }

{ number: array number, index in orig. arr } Store all this for each element

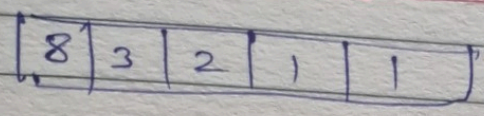




Raj
Page:
Date: / /

Smallest Subsequence with Sum K

1) Keep a maxheap and insert all elements (get a array in dec" order)



2) Keep popping the elements and keep adding them to variable 'sum' and maintain 'count'

3) When sum >= k , Break and answer = count

```
priority_queue <int, vector<int>> pq

for(int i=0; i<n ; i++) { pq.push(arr[i]; }

int sum=0, count=0;

while (!pq.empty())
{
    sum = sum + pq.top();
    pq.pop();

    if (sum >= k) { count ++ }
}

if (count == 0)  cout << -1;
else cout << count;
```