Queues

} uses parameters front & end

* considering ~~state~~ array implementation
  front = end = -1 (finite size)

* Insertion / Enqueue

  In case of Queues, insertion takes place always at end

  Basic idea – end++; Q[end] = new_val;

  Check for Overflow / is it first insert() (front will be updated then)

* Deletion / Dequeue

  Deletions take place at beginning of queue

  Basic Idea } temp = Q[front]
  front++; return temp;

Check for underflow / is it the single item that's going to get deleted

8	7	4	13	12	11

front = 0 1 2 3 4 5
end = 0 1 2 3 4 5

Lets discuss linked list implementation of Queues

```
struct node {  int val;
               struct node *next;
            }

struct Queue {
               struct node *front, *end;
               int count;
             }

void main ( ) {

struct Queue *Q;
```

~~// struct Queue Q2; Q = &Q2;~~
~~// space allocated in heap stack corr- to function call~~

```
Q = (struct Queue*) malloc(sizeof(struct Queue));
Q -> front = NULL;
Q -> end = NULL;
Q -> count = 0;
```

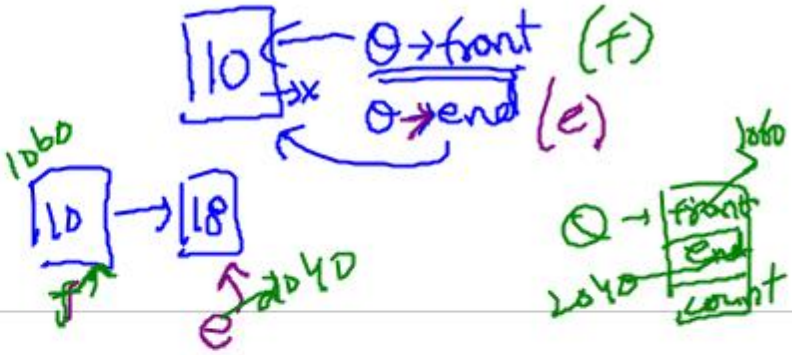
// Empty Queue

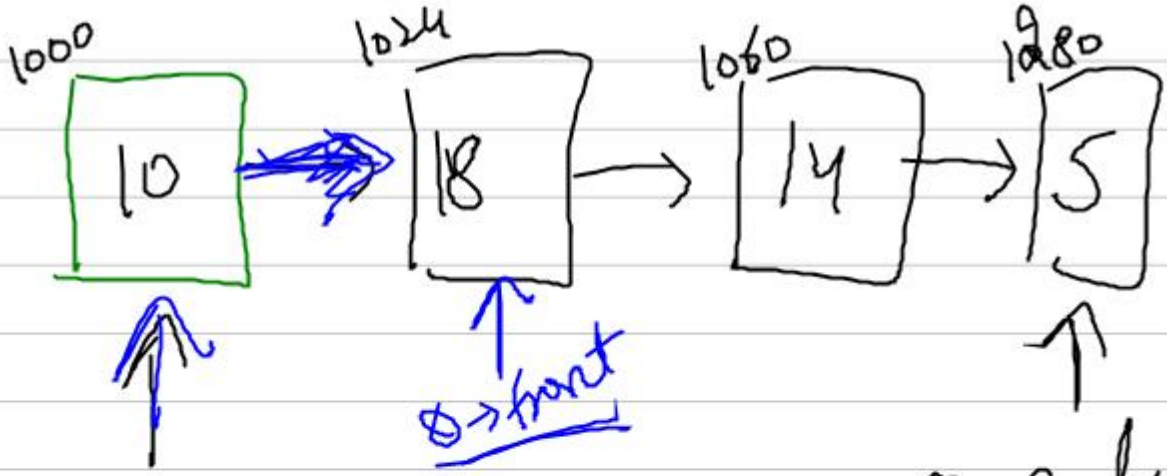
Insert(Q, 10);

Insert(Q, 18);

Insert(Q, 14);

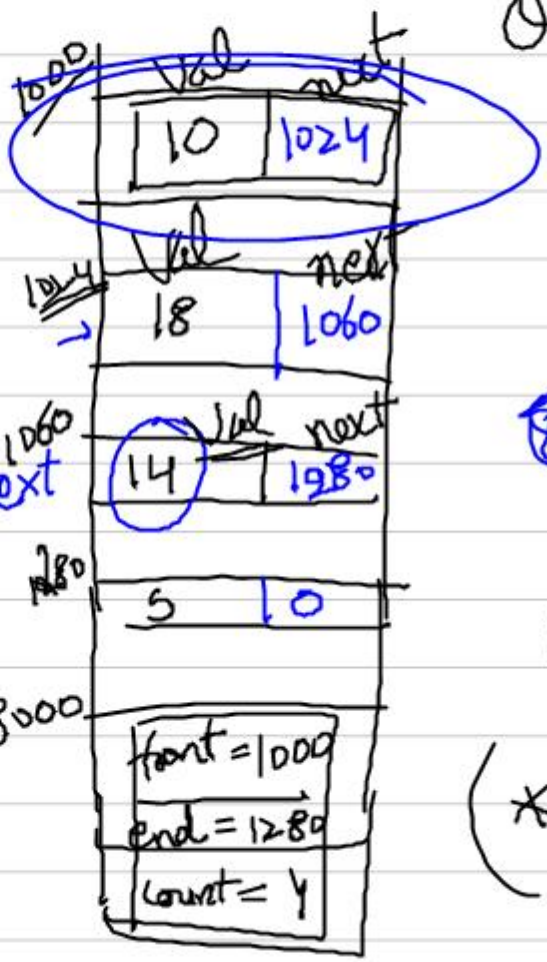
Insert(Q, 5);





Q->front

8000



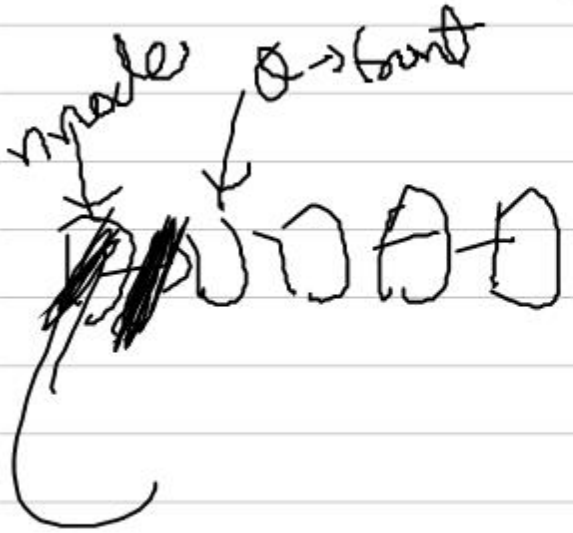
(Q->front)->next->next->val = 16

Q->count

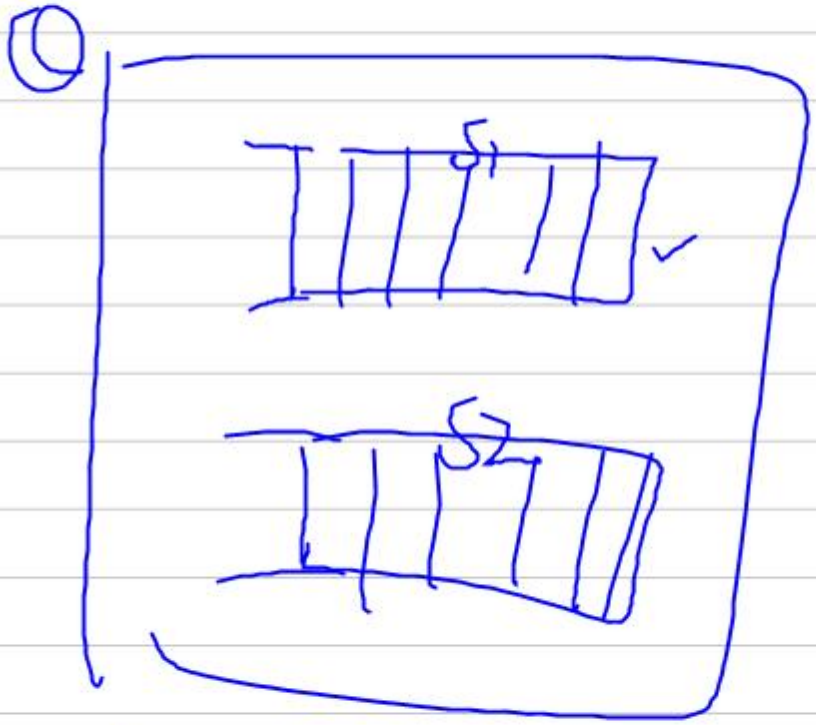
(*Q).count

```
insert (struct Queue *Q, int ~~value~~ data)
{

if (front == NULL)

else {

}
```

```
int delete (struct Queue *Q){
    struct node *mnode;
    if (Q==NULL ||
        Q->front == NULL) {
            { printf("—————")
              return -1;
            }
    else{ int temp = (Q->front)->val;
          if (Q->front != Q->end)
            { mnode = Q->front
              Q->front = Q->front->next;
              Q->count --;
            } mnode->next = NULL;
              free(mnode);
          else { mnode = Q->front;
                 Q->front = NULL;
                 Q->end = NULL;
               (Q->count)--;
               free(mnode);
          }
          return temp;
    } //end of else.
```



Implement Queue using 2 Stacks:



N ? = 10000

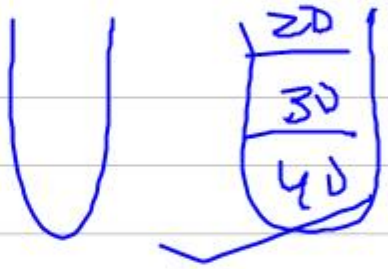
```
struct stack {
    int A[N];
    int top;
}
```

~~struct Que~~

push(S)
pop(S)
initialize( )
isempty( )
is full( )

```
struct Queue {
        struct Stack *S1, *S2;
        _ front
        _ end - -
}

main(    )

    struct Queue Q;
    Q-front = Q-end = -1;
    ~~Q.S1.top = -1; Q.S2.top = -1;~~
    initstage(Q.S1);
    Q-count = 0;

~~insert~~
enqueue(_____Q, int data)
{
        push() /pop(  ) ____ top

}
```

10 20 30 40 delete

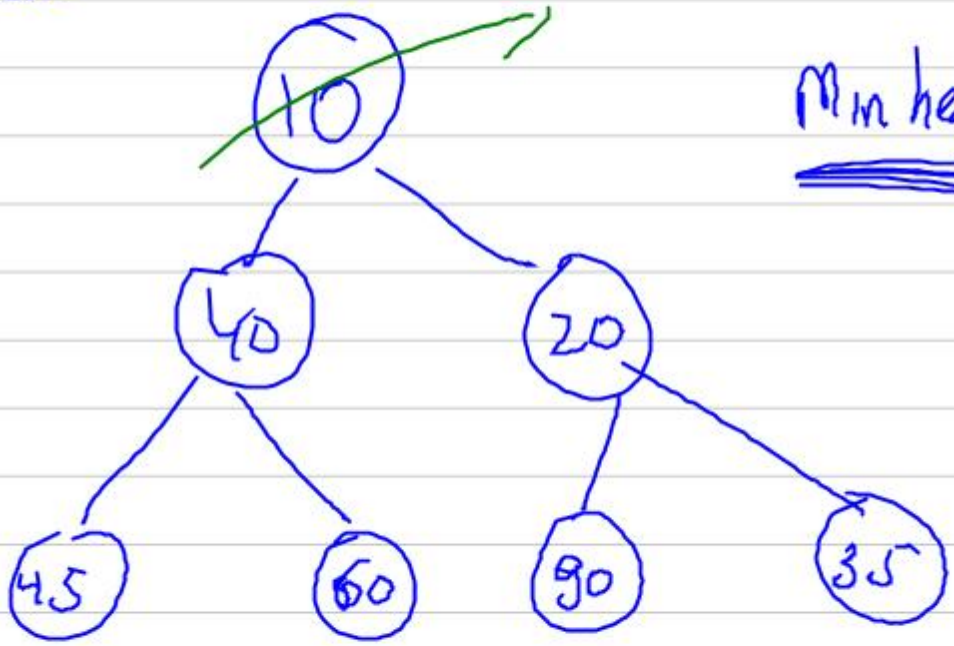


Amortized Analysis

Heaps

→ Use for implementation of priority Queues

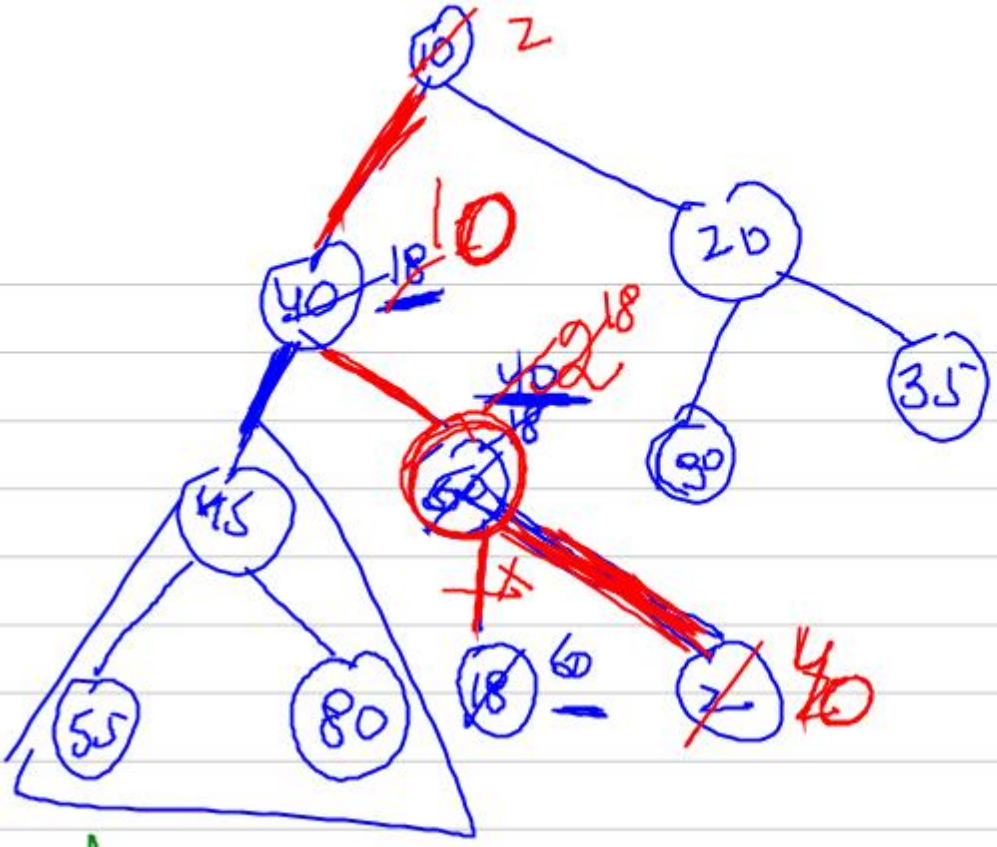
⇒



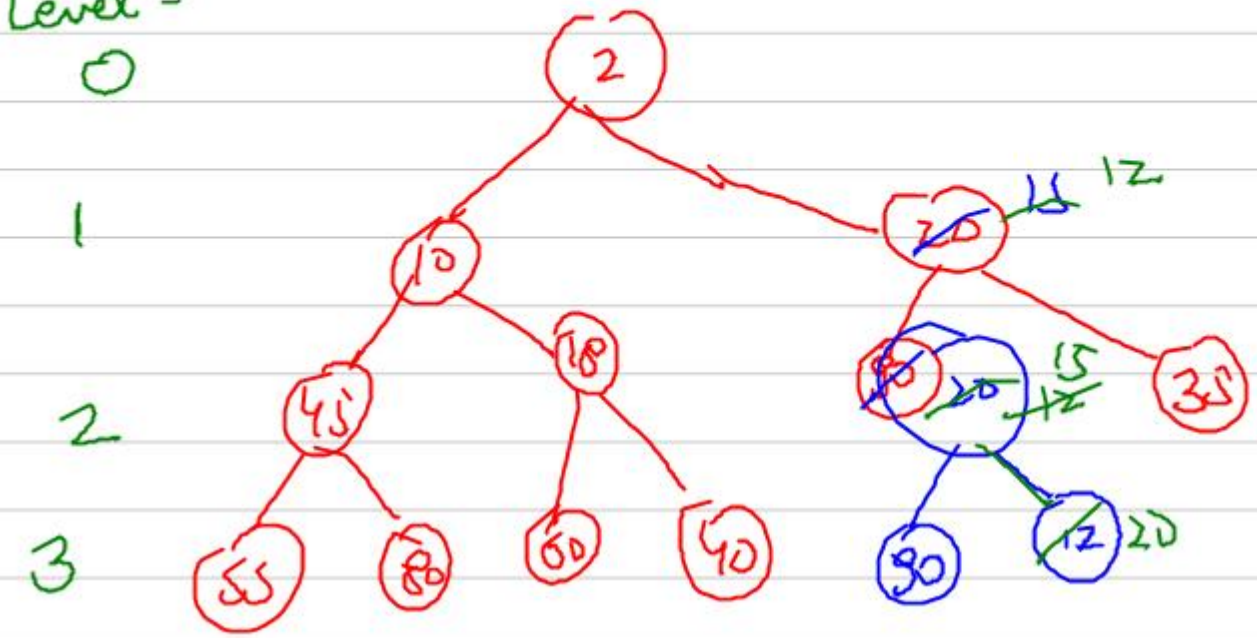
Min heap

For min heap, this relation that value of parent node is always less than equal to the value of its children nodes is preserved at every node. For a single node / leaf node, this rel. is trivially true.

// Another condition → binary tree



Level -



	Level →		Total # of nodes (Max)
2^0	0	$2^k \le$	$1 \quad 2^{0+1}-1$
2^1	1		$3 \quad 2^{1+1}-1 \quad 2^{k+1}-1$
2^2	2	$\le 2^{k+1}-1$	$7 \quad 2^{2+1}-1$
2^3	3		$15 \quad 2^{3+1}-1$
2^k	k	$2^{(k-1)+1}-1+1$	$2^{k+1}-1$

Height of tree if n nodes

$= O(\log n)$

$n = 2^k$

$\log_2 n = k$

Insert $(\log n)$

Delete $\rightarrow O(\text{height}) = O(\log n)$

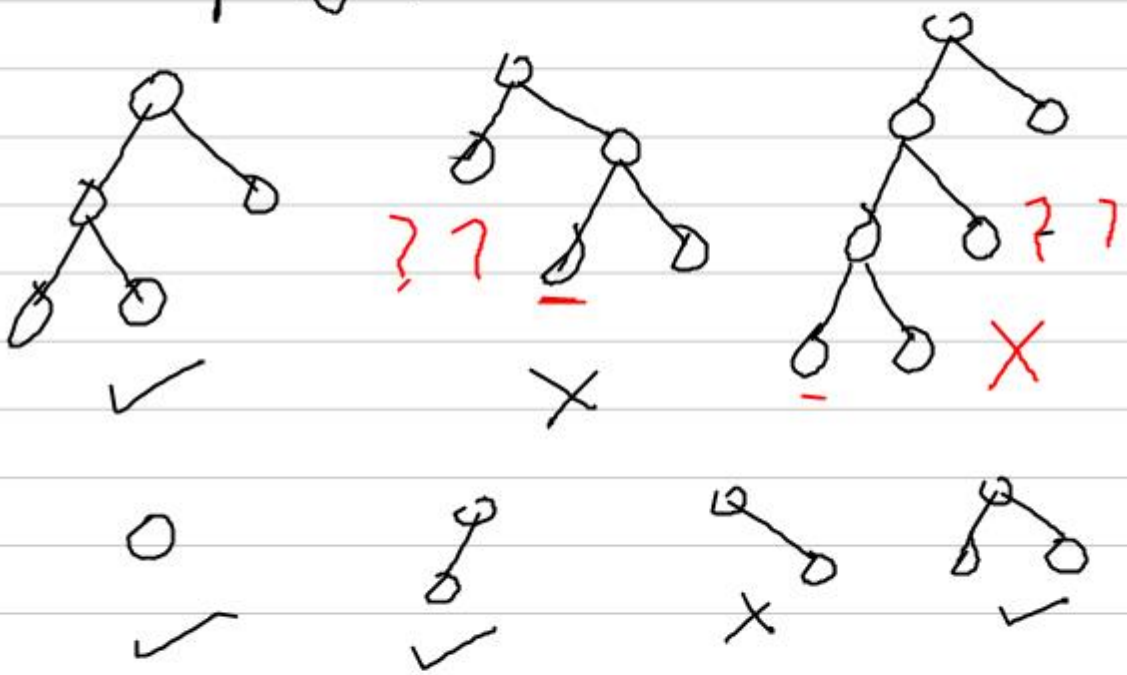
↳ Swap root node with last inserted ~~value~~ node
Delete last & then & percolate down
the root node value till you
reach the leaf node or you get
the order property satisfied

Heaps

Min-heap Max heap

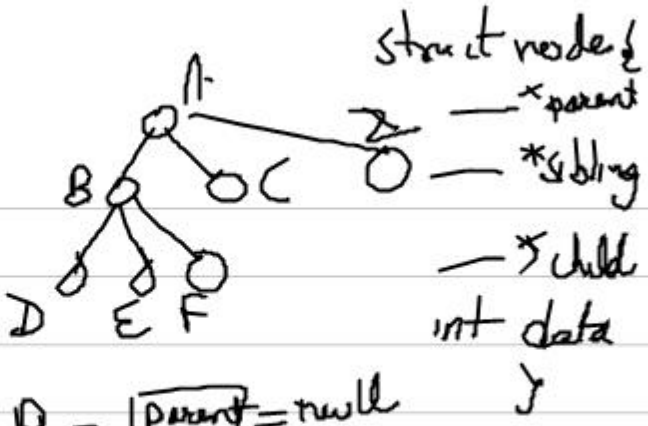
Complete Binary Tree

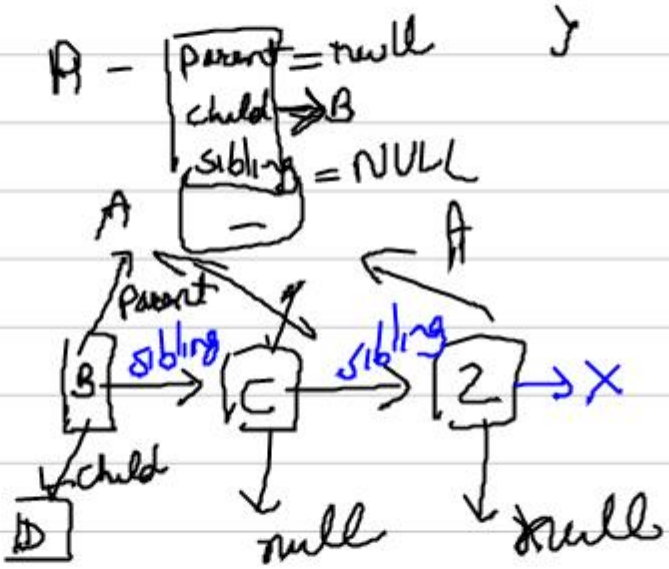
Binary Tree where we fill the nodes from top to bottom & ~~then~~ within a ~~level~~ level - left to right. i.e. to say we don't go to next level till previous level is completely filled



In general trees are implemented using

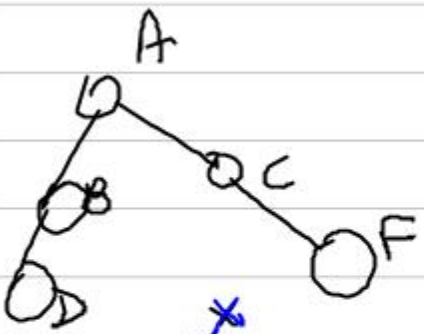
① Pointers



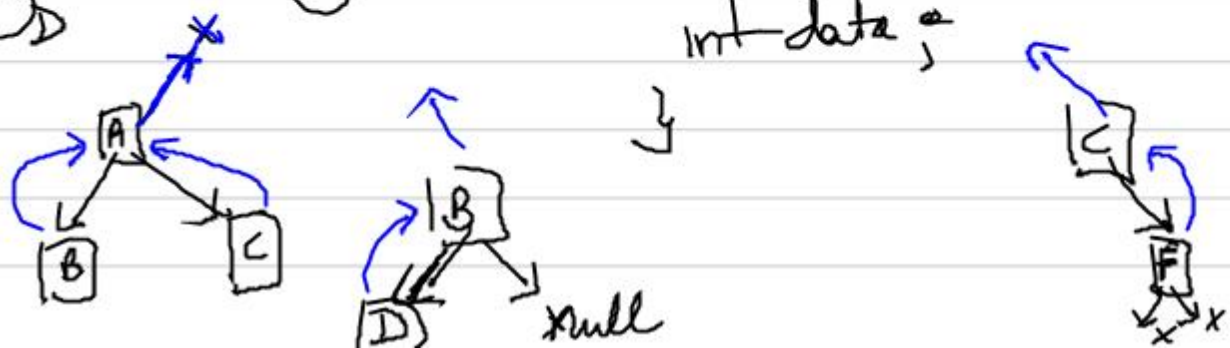


If # of children is finite fixed value known in advance, we can also have node structure in somewhat different way

E.g Binary tree - At most 2 children



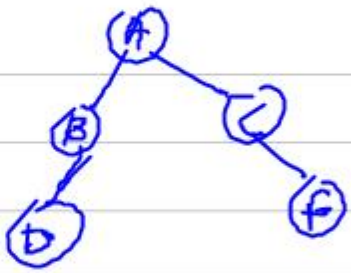

```
struct node {
    struct node *parent;
    ____________ *lchild;
    ____________ *rchild;
    int data;
}
```



i \ j	0	1	2	3	4
0	0	1	2	0	0
1	0	0	0	1	0
2	0	0	0	0	2
3	0	0	0	0	0
4	0	0	0	0	0

n x n

A	B	C	D	E
0	1	2	3	4



Edge[n][n]

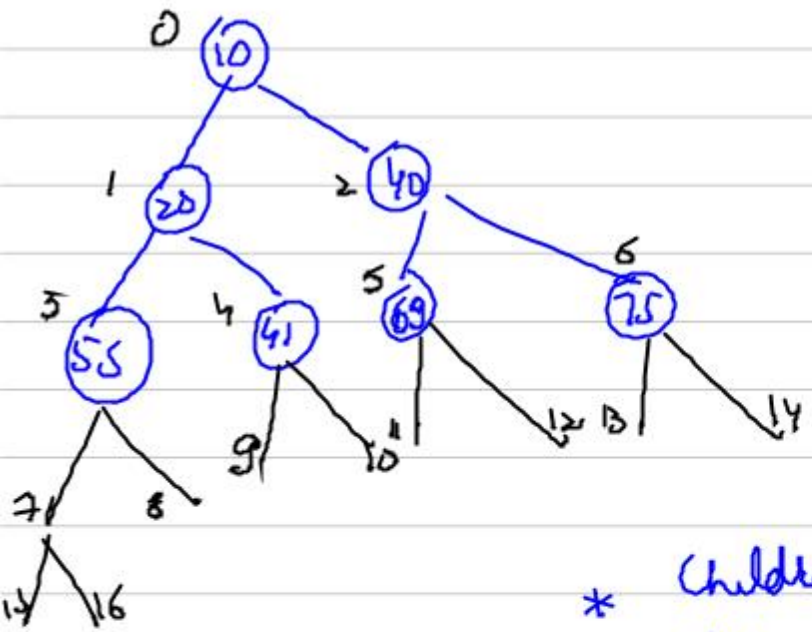
Edge[i][j] == 1 if i is parent of j & j is left child

= 2 if j is right child of i

10	20	40	55	41	69	75	
0	1	2	3	4	5	6	7

n_nodes = 6

(last index of valid entries)



* Children of node i are at indices 2i+1 & 2i+2

* Parent of node i is at index floor((i-1)/2)

Heap / Binary heaps

→ Binary tree that follows 2 properties

① Structure property — It is a complete binary tree

② Order property —

For min-heap, value at each node is less than or equal to value of any of its children nodes, if any.

For max-heap _____

_____ more than _____

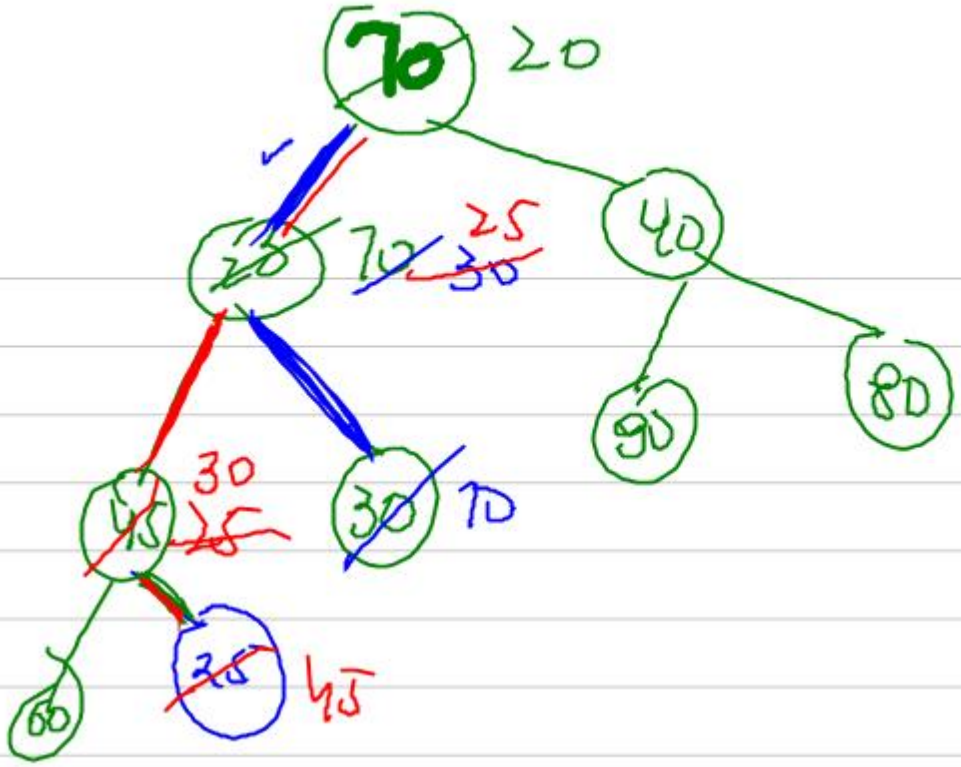
_____

_____

Deleting a node from n-nodes min-heap

- $O(\log n)$ operation

* Delete the root node
* Remove last node & keep it as root.
- Then percolate it down to maintain order property



Insertion = n_node ++;
Elem[n_node] = new_value

// Repeatedly do the following -
- check if parent node value is less than this
If yes, break;
else swap

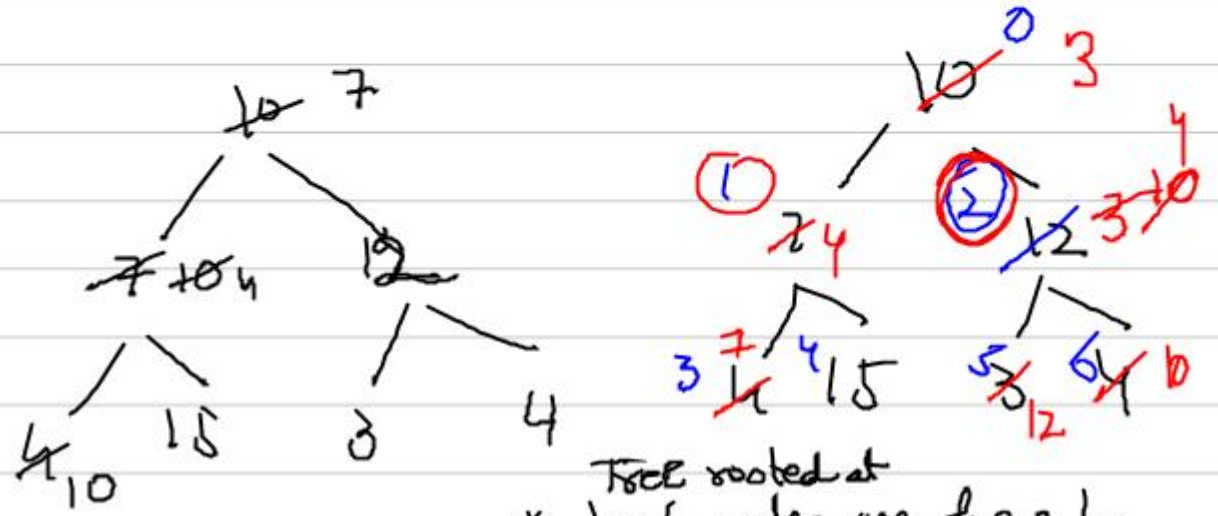
until you reach the root node

Insertion of n nodes take how much time to build a heap?

height = $O(\log n)$

$$\log 1 + \log 2 + \log 3 + \log 4 + \cdots + \log n$$

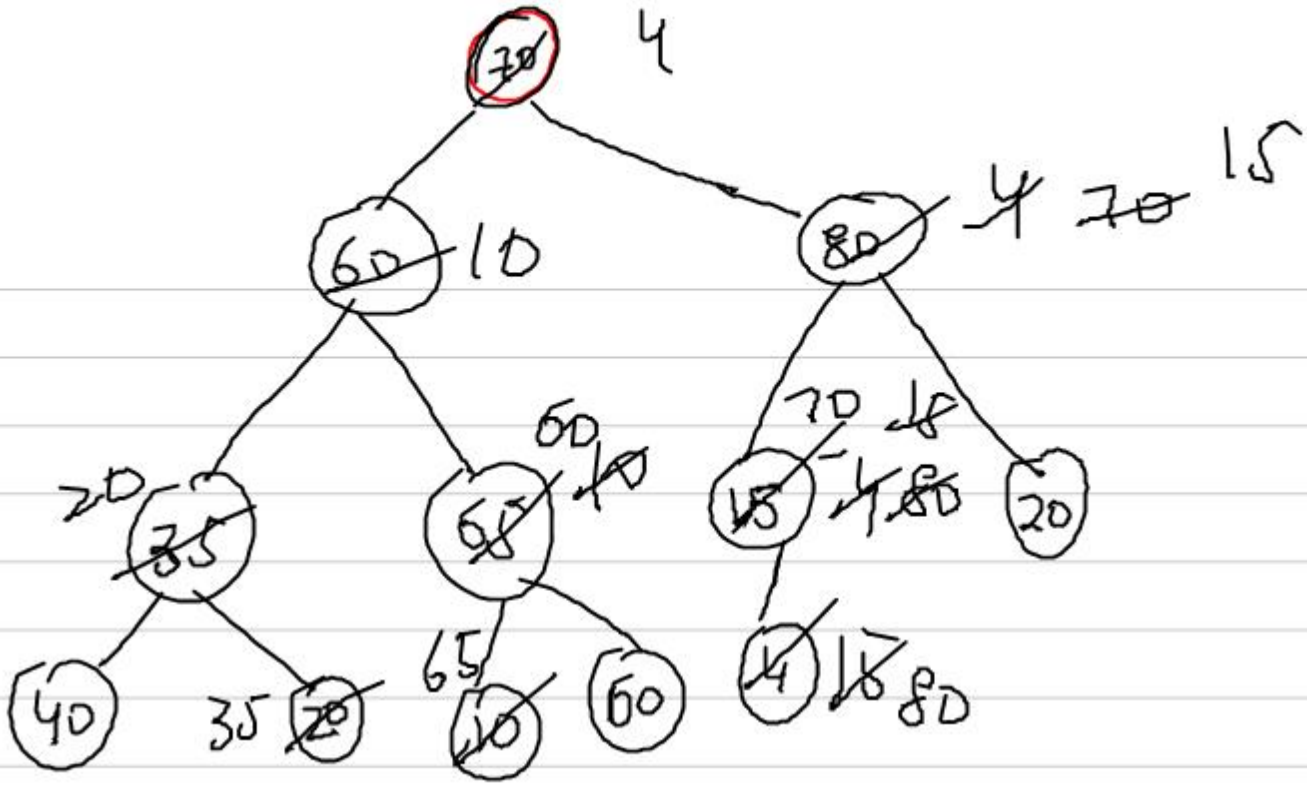
$$= \log(n!) = \underline{\underline{O(n \log n)}}$$



Tree rooted at

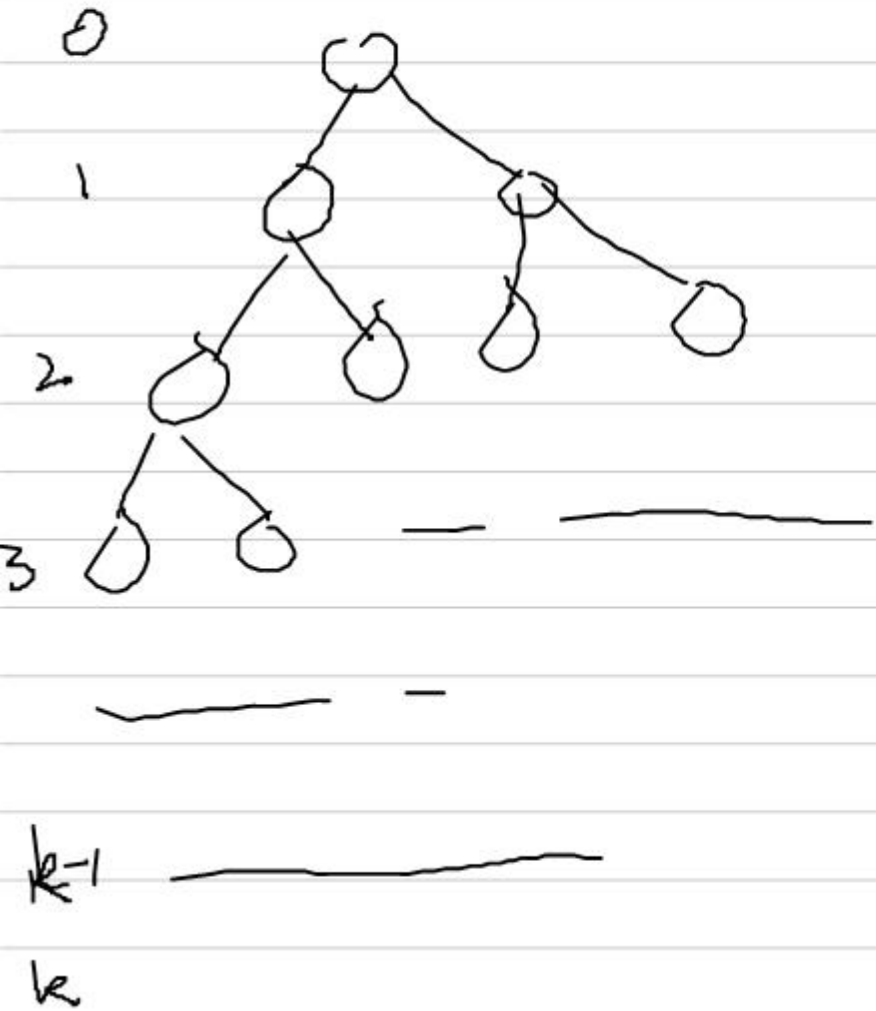
* Leaf nodes are heap by default as they are individual nodes

* Now we will go <u>up</u> level by level till we reach root node level beginning from leaf nodes level
* And at level, we will heapify the tree rooted at nodes of that level one by one



4, 10, 15, 20, 60, 70, 20,

40, 35, 65, 60, 80



$$n = 2^{k+1} - 1$$

$$= 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^k$$

$$T = 1 \cdot 2^{k-1} + 2 \cdot 2^{k-2} + 3 \cdot 2^{k-3} + \dots + (k-1) 2^1 + k\, 2^{k-k}$$

$$T = 1 \cdot 2^{k-1} + 2 \cdot 2^{k-2} + 3 \cdot 2^{k-3} + 4 \cdot 2^{k-4} + \cdots (k-1)2^1 + k \cdot 2^0$$

$$2T = 2^k + 2 \cdot 2^{k-1} + 3 \cdot 2^{k-2} + \cdots + k \cdot 2^1$$

$$2T - T = 2^k + 2^{k-1} + 2^{k-2} + \cdots 2^1 - k$$

$$= \left(\frac{2^{k+1} - 1}{2-1}\right) - 1 - k$$

$$T = O(n)$$

build_heap (A[1 — n])

id = $\lfloor \frac{n}{2} \rfloor + 1$

for (i = id ; i ≥ 0 ; i--)

heapify (A[ ], i)

HW Prog. Assignment

* WAP for insertion | deletion | display → end
  in SLL
* ———————— in DLL
* ———————— for heap

Due ~~Tuesday~~ Wed 18th September evening

* Implement Queue using 2 stacks]

puneet goyal

SLL_pgoyal.c
dll_puneet_goyal.c
heap_puneet_goyal.c

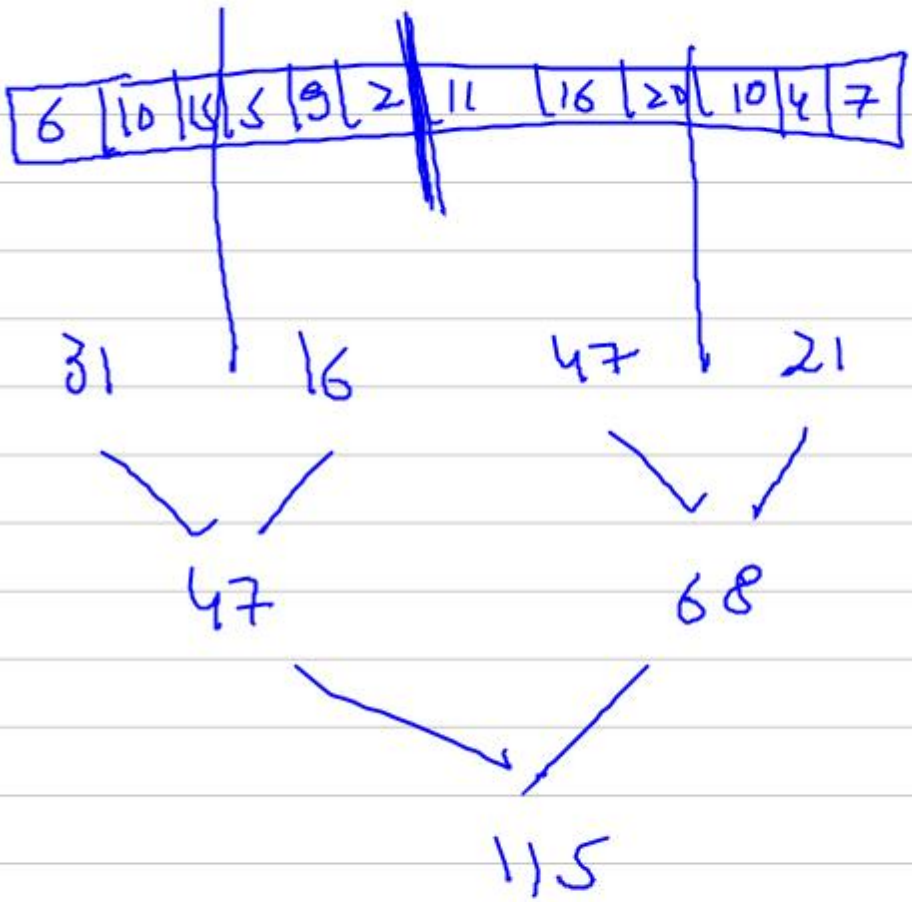
↳ zip it .zip (No .rar)

Subj → Prog. Assignment 2 (Adv. Algo)

Srand(1);

Rand( )

Divide & Conquer Approach



$$T(n) = a\,T(n/b) + f(n)$$

e.g. Binary search

Problem: Given a sorted array of size n, and an element key, find if key is present in given array

find 15 in

5	7	10	15	30	41	65	78	90
		2	3		5		7	

A 0 n-1

```
int binarysearch( int A[ ], int beg, int end,
                  int key)
{ if (beg>end) return -1
  int mid = floor( (beg + end)/2 );
  if (A[mid] == key) return mid;
  if (A[mid] < key)
        return binarysearch(A, mid+1, end, key)
  else
        return b___ (A, beg, mid-1, key)
}
```

bsearch(A, 0, 7, 15) bsearch(A, 0, 7, 70)

	5	7	10	15	30	41	65	78	90
Index	0	1	2	3	4	5	6	7	8

← A

```
bsearch(A, 0, 7, 15)
{   is beg > end  X
    mid = (0+7)/2 = 3
    A[mid] = 15 == 15 Yes
        return 3;
}
```

```
bsearch(A, 0, 7, 70)
{   0 > 7  X
    mid = 3
    15 < 70  True ✓
        bsearch(A, 4, 7, 70)
        {   mid = 5
            41 < 70 ✓
            bsearch(A, 6, 7, 70)
            {   mid = 6
                A[mid] = 65 < 70
                bsearch(A, 7, 7, 70)
```

bsearch(A, 7, 7, 70)

{ is beg > end ×

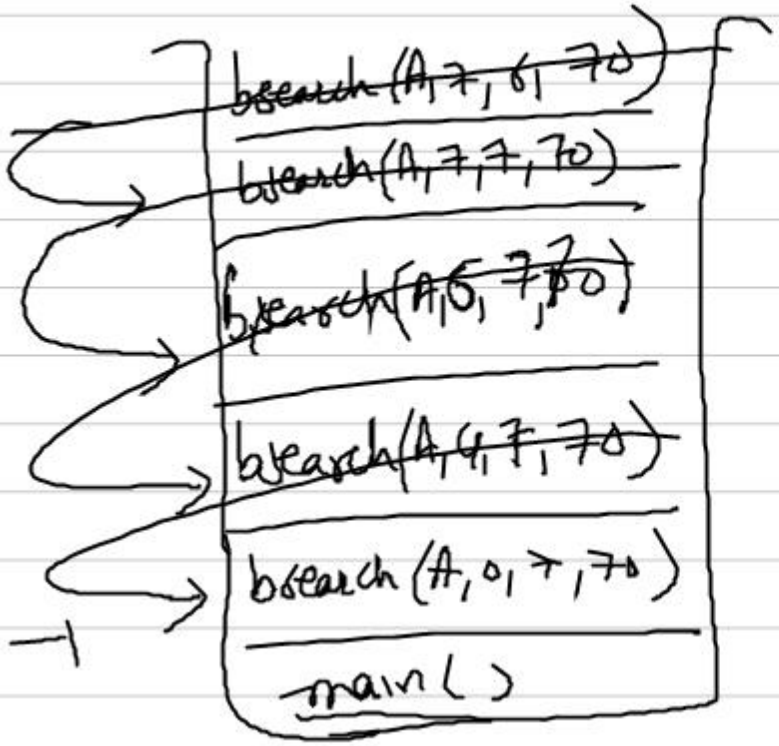
mid = 7

A[7] = 78 < 70 no

bsearch(A, 7, 6, 70)

{ 7 > 6 True return -1

}



function calling stack

$$T(n) = \Omega(1)$$

$$= O(\log n)$$

Merge Sort is one of the important sorting algorithms based on Divide & Conquer approach

→ We partition the given array of random elements into two halves - left half + right half. (i.e. partitioning acc. to elements position)

→ — ——

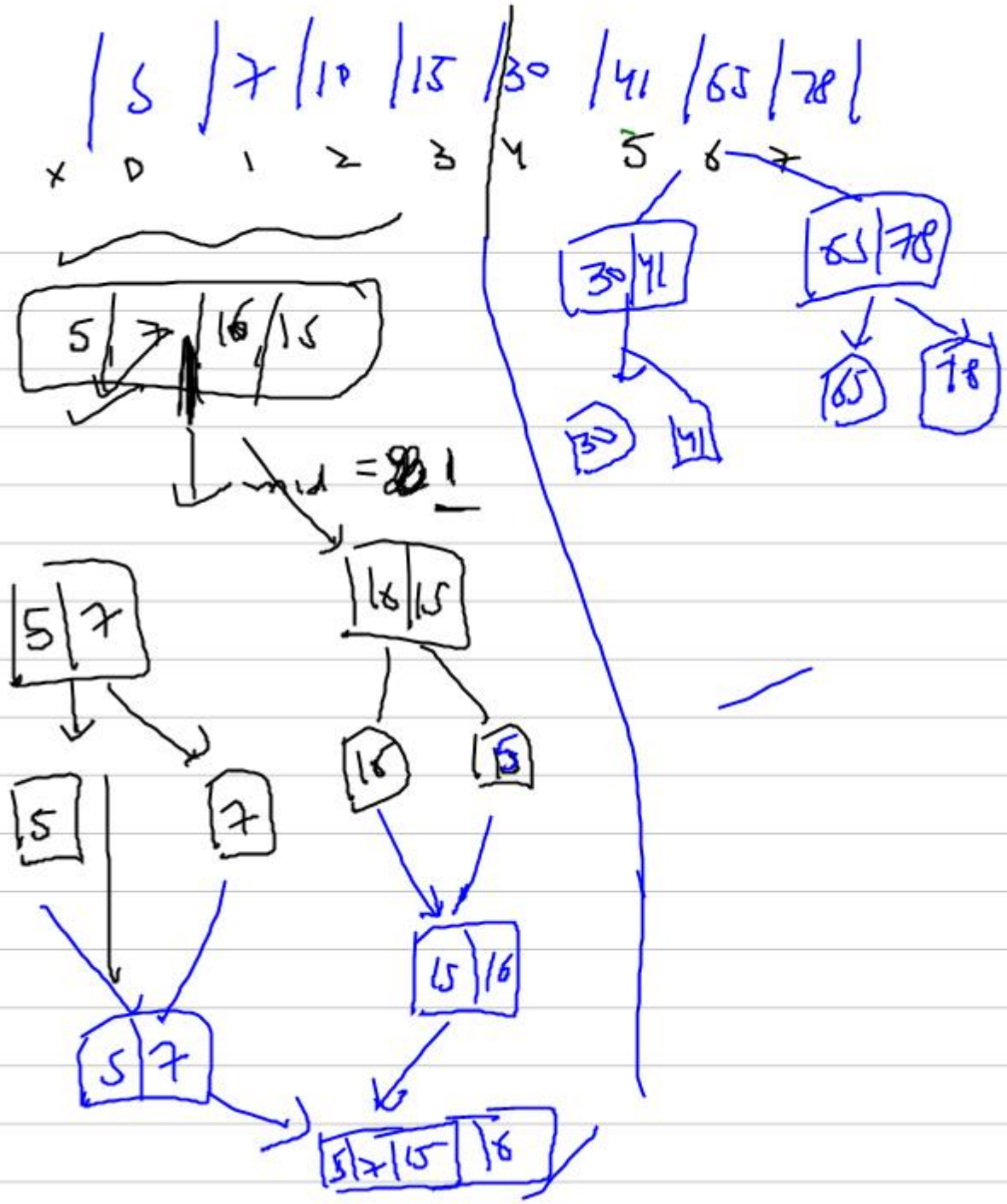
→ Merge ⇝ $O(m+n)$

→ Time complexity of merge sort algo is $\Theta(n \log n)$

$$T(n) = 2T(n/2) + \Theta(n)$$

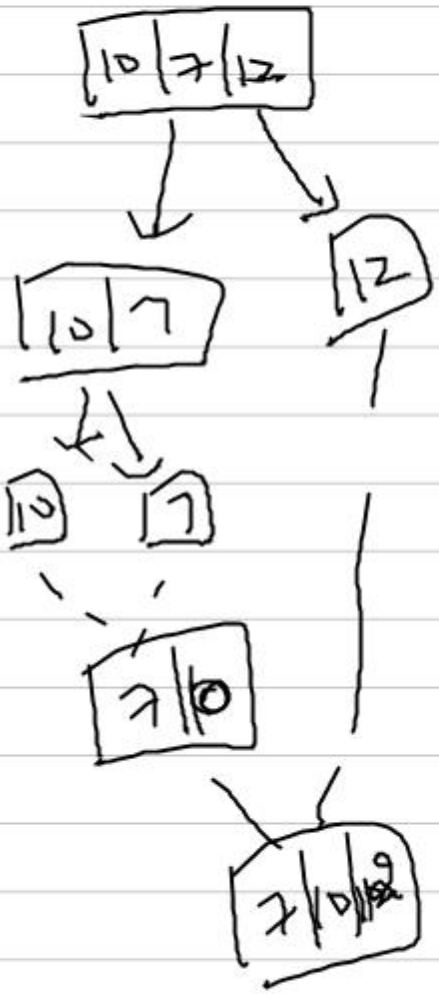
$$\Rightarrow T(n) = n \log n$$

Case II of Master's Theorem


5 7 10 15 30 41 65 78
x 0 1 2 3 4 5 6 7
5 7 16 15
mid = 1
5 7
16 15
5
7
16
5
5 7
15 16
5 7 15 16
30 41
65 78
30
41
65
78

10	7	12	15	6
0	1	2	3	4

Idx



```
void msort(int A[], int beg, int end)
{ if (beg < end){
        mid = (beg+end)/2 ;
        msort(A, beg, mid) ;
        msort(A, mid+1, end) ;
        merge(A, beg, mid, end);
}
}
```

HW merge(

msort(A, 0, 10)

5	9	12	4	2	1	20	14	20	10	11

mid=5

msort(A, 0, 5)

5	9	12	4	2	1

mid = 2

msort(A, 0, 2)

5	9	12

mid=1

msort(A, 0, 1)

mid=0

5	9

mid=0

msort(A, 0, 0) msort(A, 1, 1)

5	9

merge

5 9

~~merge(A, 0, 1)~~
~~merge(A, 0, 1, 2)~~
~~msort(A, 1, 1)~~
~~msort(A, 0, 2)~~ bag=2 end=2 mid=1
~~msort(A, 0, 5)~~
msort(A, 0, 10)
main( )

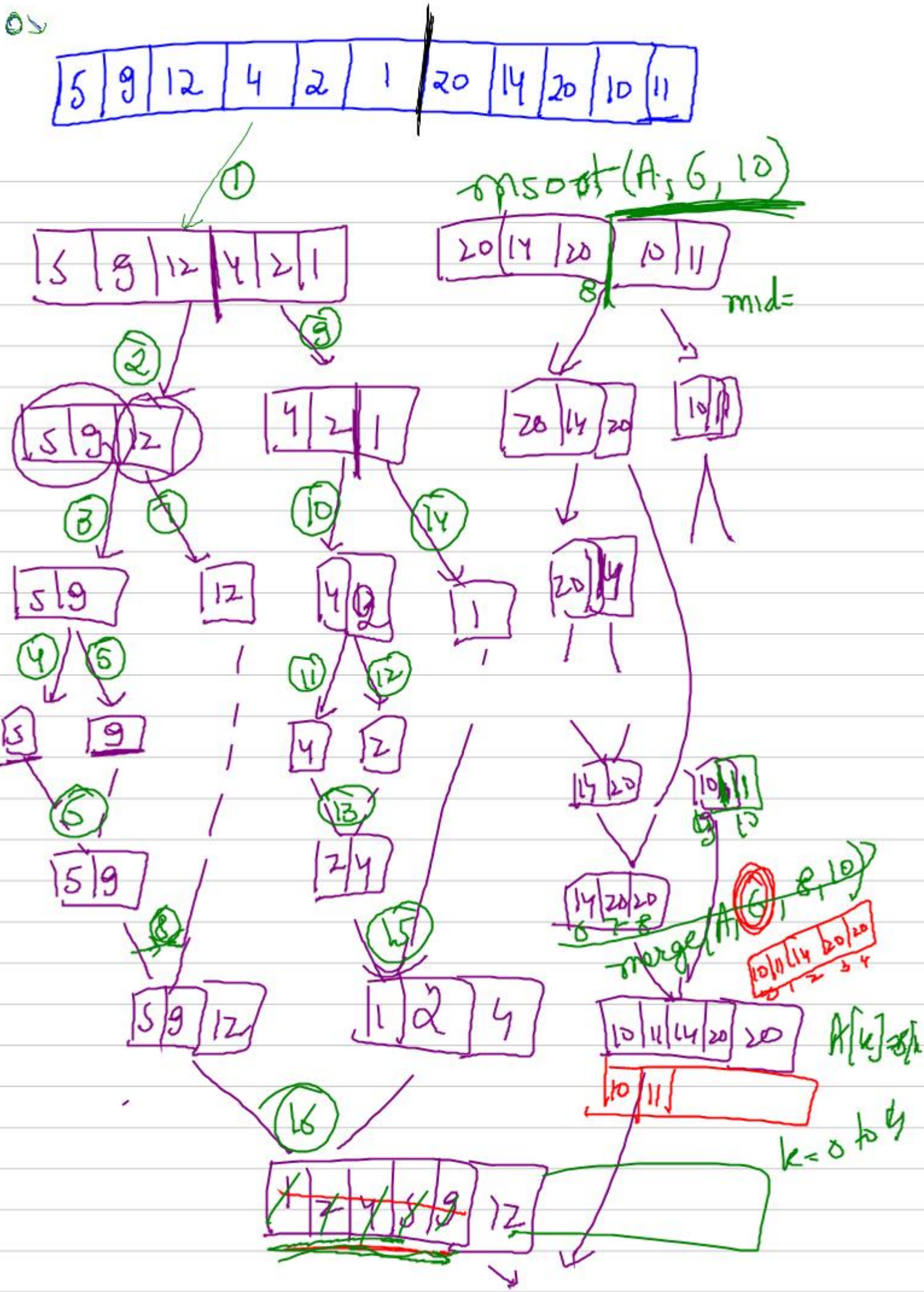
main( ) {

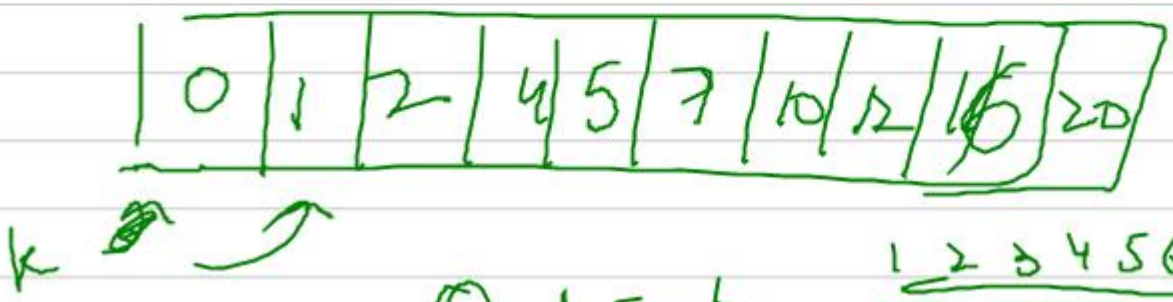
msort(A, 0, 10) {

msort(A, 0, 5) {

~~msort(A, 0, 2)~~ {

~~msort(A, 0, 1)~~ {


5 9 12 4 2 1 20 14 20 10 11
msort(A, 6, 10)
mid=
merge(A, 6, 8, 10)
A[k]
k=0 to 4



Quicksort

5 3 1 9 8 2 4 7