# Binary Search

| 2 | 5 | 10 | 23 | 32 | 50 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

$l$ ↑ $m$ $r$

if ($l > r$) return -1.

if ($x == arr[m]$) return arr[m]

else if $x > a[m]$ $l = m+1$ ; bin_s($l$, $r$)

else if $x < a[m]$ $r = m-1$ ; bin_s($l$, $r$)

---

# Abstract Data Type

- Abstraction of D.S, a logical view only
- Provides an interface to how a data structure must adhere.
- ~~Functions and operations that~~ How the data is stored and operated
- No implementation

D.S. A way to store and organize data so that it can be used and accessed effeciently.

Study D.S.

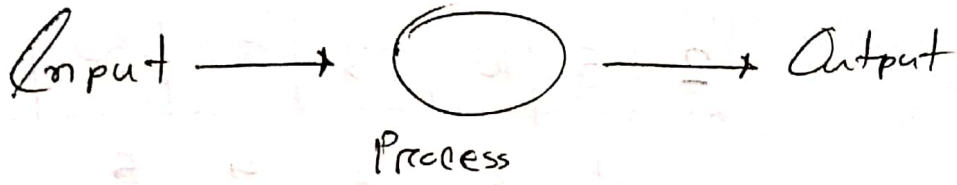
- ~~ADT~~ Logical View
- Operations
- Cost of Operations

ADT

- Implementation

# Data Structures

13-01-19




To effectively utilize two resources → time and memory used we need algorithms and data structures.

NP-complete → till date no efficient solution has been found.

## Complexity

```
for(int i=1; i<=n; i++)   → c1 × (n+1)
    printf("...");       → c2 × n
```

$c_1 n + c_1 + c_2 n$

$= n(c_1 + c_2) + c_1$

$= nC + D$

Talking about complexity and growth we discard constant terms and lower-eff.

Growth of this function is linear.

Big O — $O(n)$ → Worst case complexity

Big Omega — $\Omega(n)$ → Best case "

Big theta — $\Theta$ → Idk what this is, LIU.

Asymptotic Notation ~~Relation~~

$O(1)$ → means O is independent of the n.

it means constant complexity.

~~Here,~~ $f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

Here, exist the constant $c_1$ such that,

$0 \leq f(n) \leq c\,g(n)$, $\forall\ n > n_0$

then $f(n) = O(g(n))$

Similiarly, $f(n) \geq c \cdot g(n)$, $\forall\ n \geq 0$

then $f(n) = \Omega(g(n))$

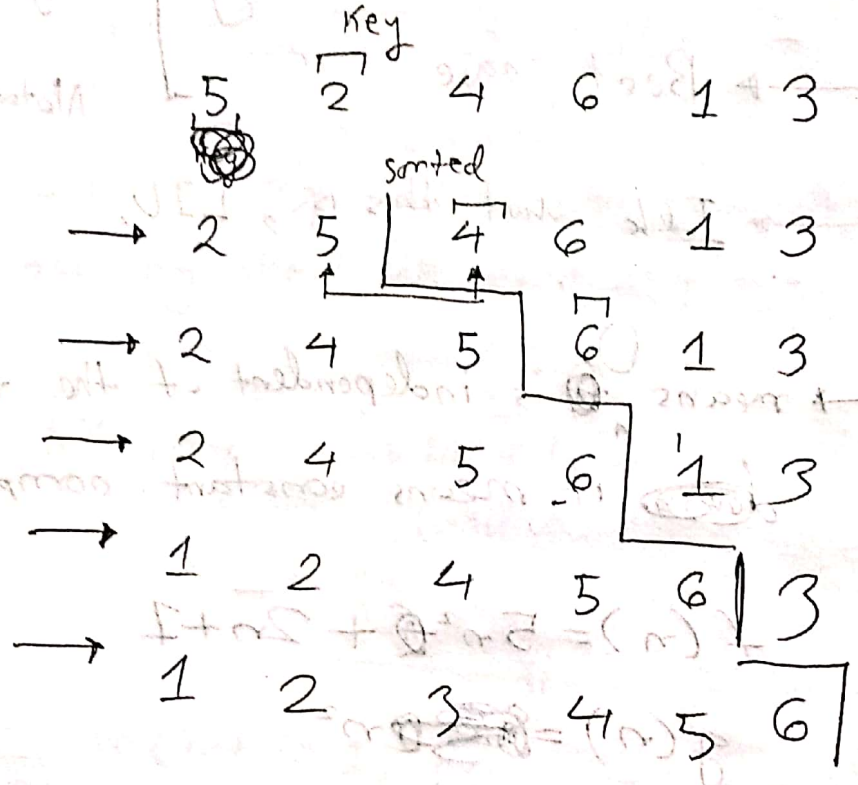
For some constant,

$c_2 g(n) \leq f(n) \leq c_1 g(n)$

then $f(n) = \Theta(g(n))$

# Chapter-02

Insertion sort ⟶ $O(n^2)$




```
for j = 2 to ~~array.length~~ A.length
    Key = ~~A[y]~~ A[j]
    i = j - 1
    while (A[i] > key && i > 0)
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

| cost | times |
|---|---|
| $c_1$ | $n$ |
| $c_2$ | $n-1$ |
| $c_3$ | $0$ |
| $c_4$ | $n-1$ |
| $c_5$ | $\sum_{j=2}^{n} t_j$ |
| $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| $c_7$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| $c_8$ | $n-1$ |

$$T(n) = c_1(n) + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} t_{j-1} + c_7 \sum_{j=2}^{n} t_{j-1} + c_8(n-1)$$

Best case scenario: if the array is already sorted,

Best case:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_6 \times 0 + c_7 \times 0 + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_6)n$$

$$- (c_2 + c_4 + c_5 + c_6)$$

$$T(n) = an + b$$

$$\therefore \Theta(n)$$

Worst case: $c_1(n) + c_2(n-1) + c_4(n-1) + c_5\left\{\frac{n(n+1)}{2} - 1\right\}$

$$+ c_6 \frac{(n-1)n}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$$

$$T(n) = an^2 + bn + c$$

$$\therefore O(n^2)$$

Space-complexity $O(1)$ [Memory complexity]

Because it doesn't grow with the size of input.

Proving correctness of an algorithm

→ Loop invariant

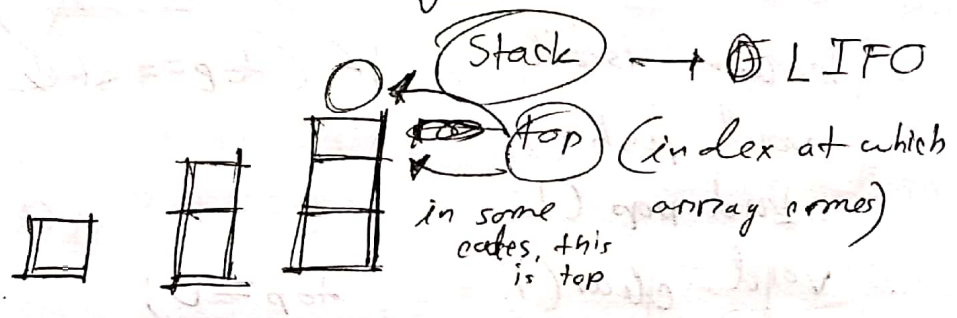
- initialization → before exe
- maintainance → true ~~before/after~~ during execution
- termination → after exe

#3

# Stack

## ADT

### Abstract Data Type




insert → push() (at the end)

remove → pop() (from the end)

```
void push (int s[], int &top, int item)
{
    if (top < stack_max)
      {
          s[top+1] = item;
      }
    else { error ( );
      }
int pop (int s[], int &top)
   { if top == 0 (error) return -1
     else {
   }  }
```

functions

boolean isemnty(); if (top == 0)

bolean isfull(); if (top == stack_max)

void push();

void pop();

void clear() top = 0;

void size(); size = top

top; top();

Lab task: #include <stack> for STL

(3-4)*5+7 infix notations

↓

postfix notations

Reverse Polish Notation

(3+4) x5-6

3,4+5x6-

when they find (+), it discards 3, 4

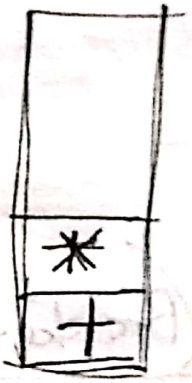
infix → post fix

A + B * C − D + E

when we find number we put it in post fix

A B C * + D E +

we find stack we put in stack.




(ii)

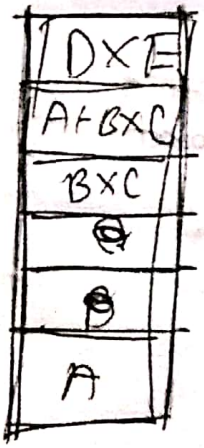
when we find one with lower precedence, we pop all the elements

(i)

when we find an operators with higher prec. we push

push/pop in stack

post-fix → infix



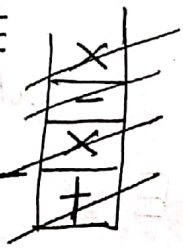

we push numbers in stack

Scanned with CamScanner

# infix → postfix

A+BXC-DXE

ABCX+DEX-

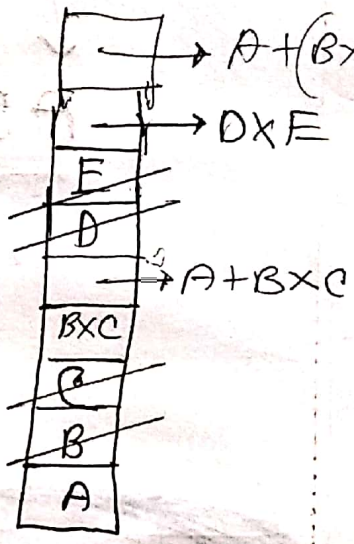


pop → lower priority operator
→ end of string

infix

push → if number
→ if operator, \

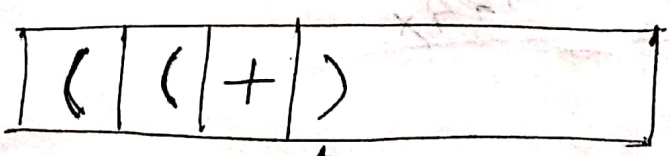
pop → end of array




## Infix (Brackets)

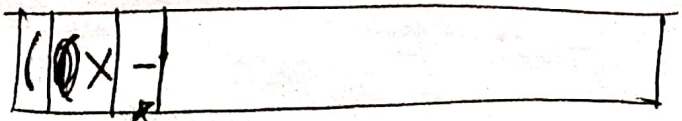
pop

((A+B)XC-D)XE




↑ pop until opening bracket

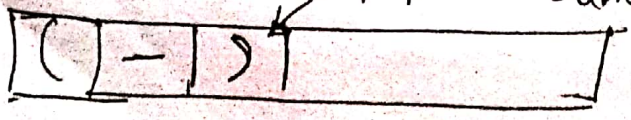
AB+



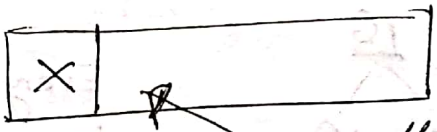

lower prity

AB+CX

pop all elements until opening bracket



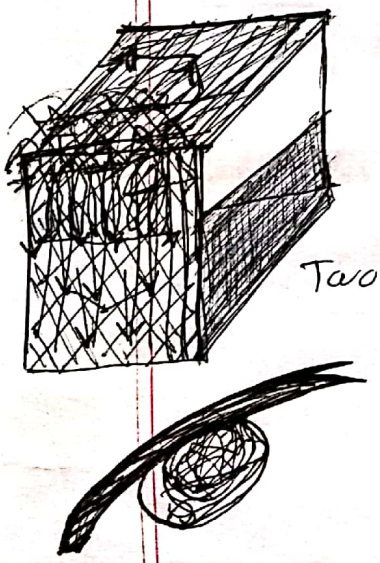

AB+CXD-

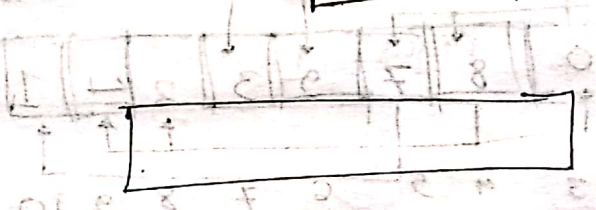



pop all till end of string.

AB+CXD-EX

## Queue FIFO





front rare

Two pointer approach

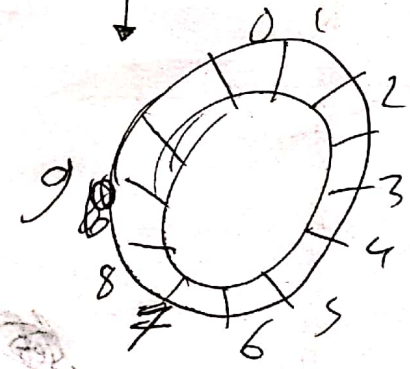
front rare

→ queue from front to rare.

front++ to delete

rare++ to put in

The code of circular queue is after priority queue




queue_max

← Use modulus

8 % 10 = 8

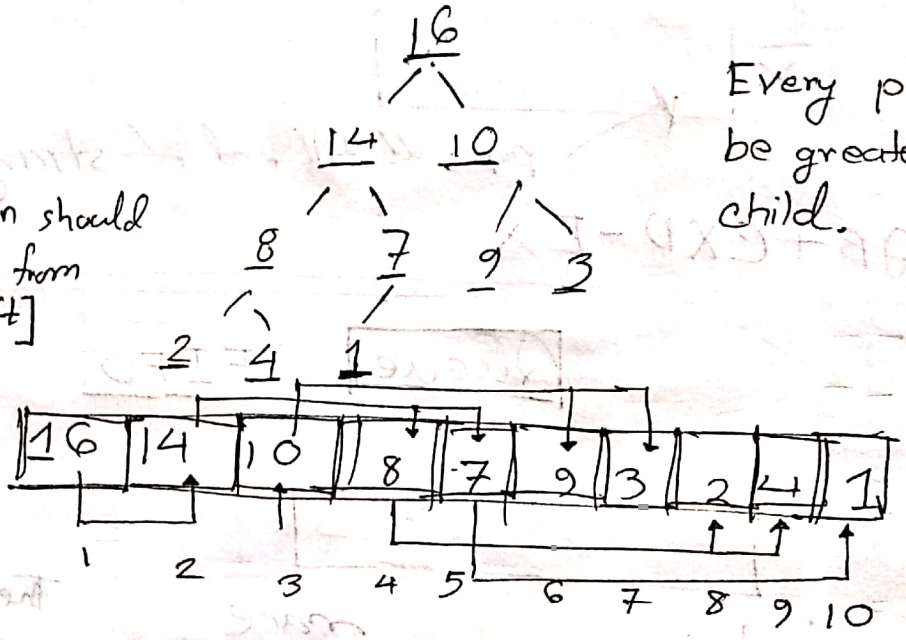
9 % 10 = 9

10 % 10 = 0

[circular queue to ~~max return~~ make memory more effecient]

# Heap

- nearly complete binary tree




[Children should be filled from the left]

Every parent must be greater than his child.

1 → 2 3

2 → 4 5

3 → 6 7

i → (2i) (2i+1)

where i = index number

child_left (i) {

return 2*i;

child=right (i) {

return 2*i+1;

parent (i),

return $\lfloor \frac{i}{2} \rfloor$

① Max $A[parent(i)] \geq A(i)$ [For every i, other than the root]

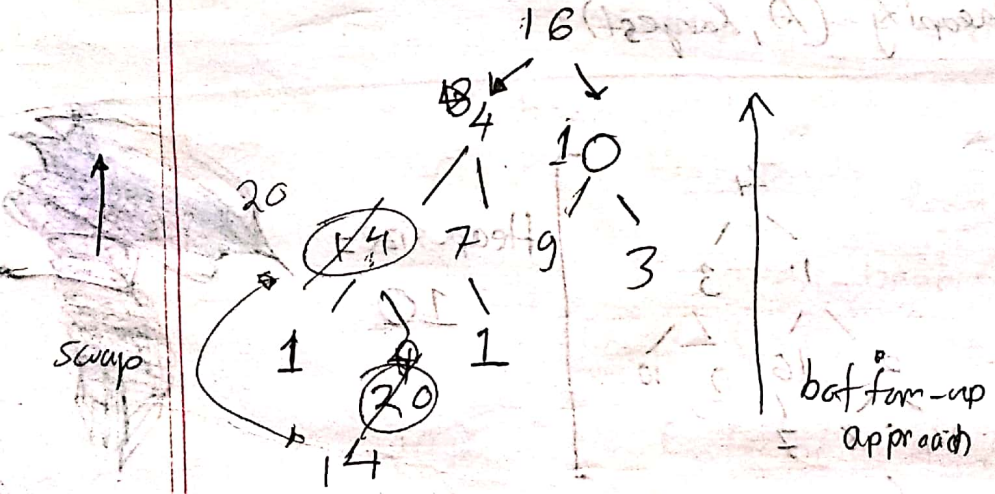
$A[parent(10)]$

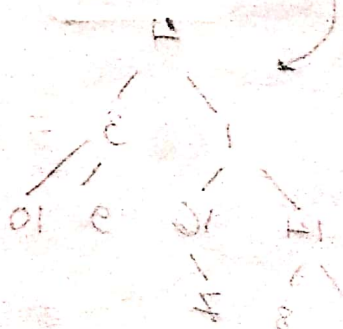
$\forall(i)$, other than root

② Min $A[parent(i)] \leq A(i)$ $\forall$ i other than root

Leaf nodes (with no child are nodes after $\frac{n}{2}$)

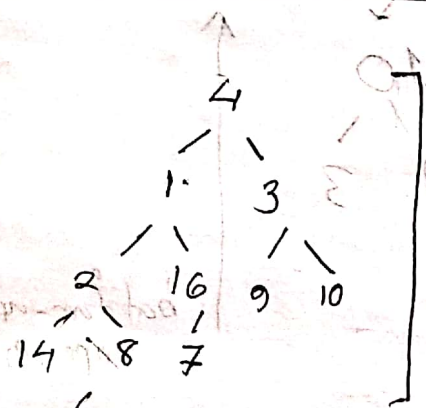
max_Heapify → converts random array to heaps



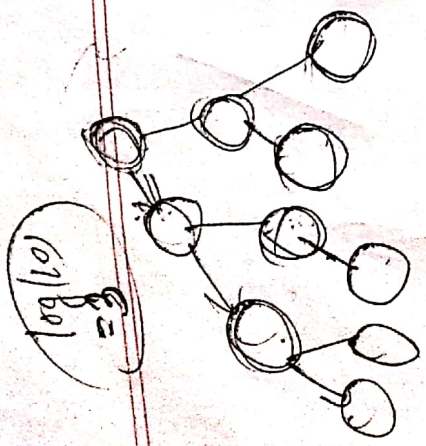



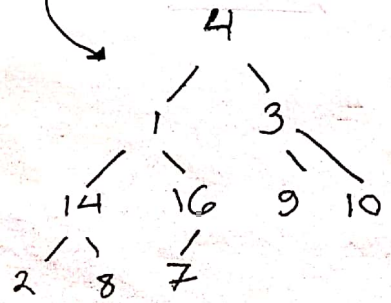
→ O(log n)

```
max_heapify(A,i)
    l = left(i)
    r = right(i)
if (l <= heap_size and A[l] > A[i]) largest = l;
 else largest = i
if (r <= heap_size and A[r] > A[largest]) largest = r
if largest ≠ i
    swap A[i], A[largest];
    max_heapify (A, largest);
```




Heap-size = 10



Heap_size → No. of elements in a heap

build max_heap (A)

for all the leaf nodes

for (i = heap_length/2 down to 1)

up each

max_heapify (A, i);


Complexity of Heap

Build_max_heap → O(n log n)

i. $\frac{n}{2} \rightarrow 1$

max_heap → O(log n)

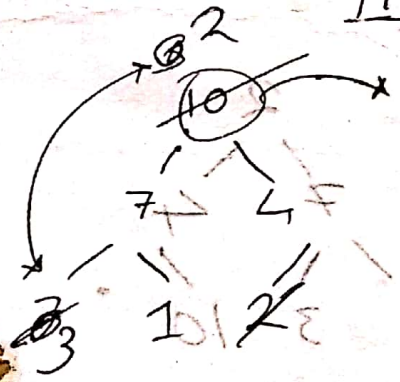
Check this

build_max_heap (A) A.heap_size = A.heap_length

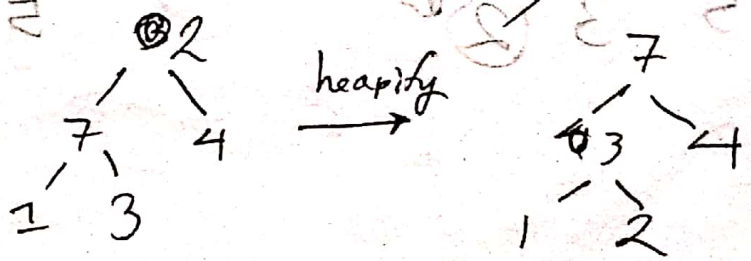
for (i = heap_length/2 down to 1)

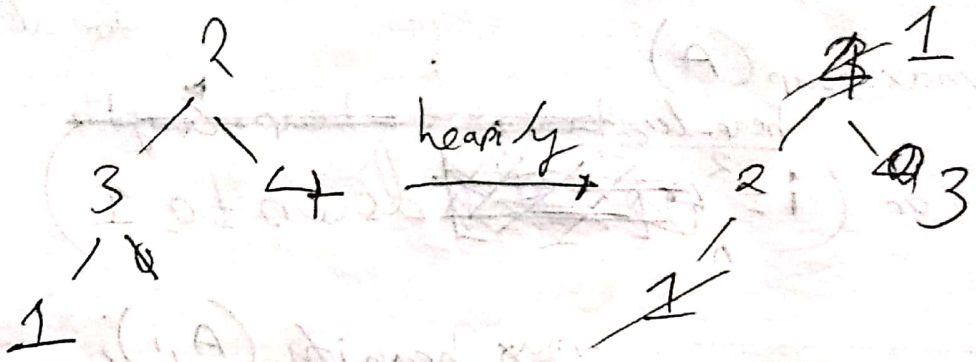
max_heapify (A, i)

Heap Sort

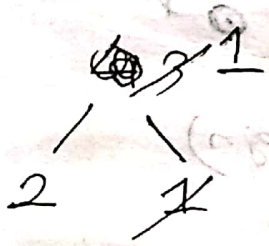



Extract the parent (take it from the heap) and replace it with a leaf node (last one)

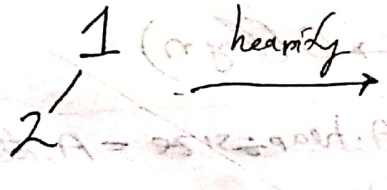





arr | 10 | 7 | 4 | | | |

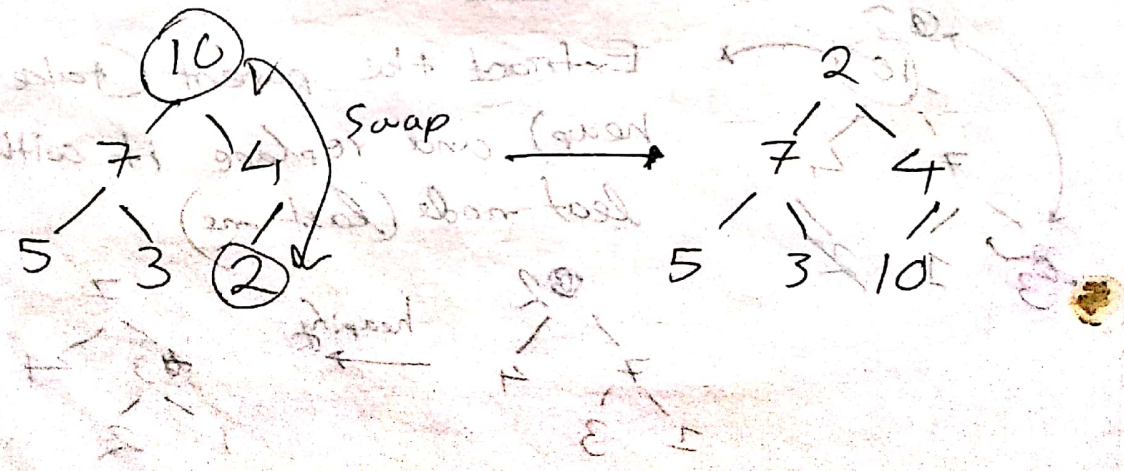



arr | 10 | 7 | 4 | 3 | | |



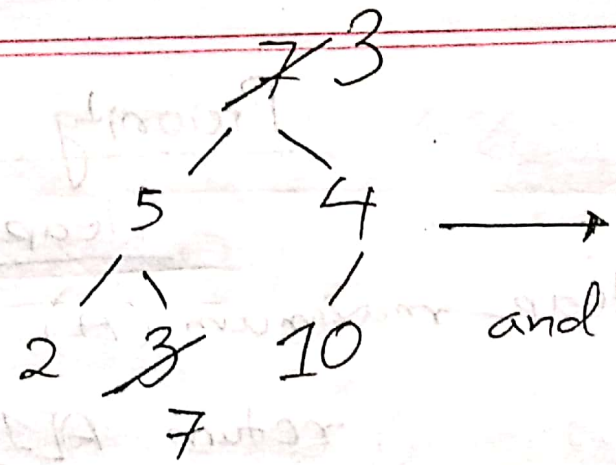

| 10 | 7 | 4 | 3 | 2 | 1 |

Instead of putting it on other array we can just swap with the last node.



heapify (size of heap, starting node)

~~(level or~~ ~~parent)~~




and so on -----

Heap_sort(A)

build_max_heap(A) ——→ O(n log

for (i = A.length down to 2) ——— O(n)

{

swap A[1], A[i]

A.heap-size --;

max_heapify(A, 1) ——→ O(log

}

#6

# Priority Queue using Heap

Max Value in Prio Queue

```
Heap_maximum(A)
    return A[1];
```

Extract (Like POP) [log n]

```
Heap_extract_max(A)
    if (heap_size >= 1)
    { max = A[1];
      A[1] = A[heap_size];   // swap(1, heap_size);
      heap_size--;
      max_heapify(A,1);
      return max;
    }
```

(Change value) O(log n)

position, changes value at that position.

```
Heap_increase_key(A, i, key)
  if A[i] > key
  else
      while(i > 1 and A[parent(i)] < A[i])
      {
          swap(A[i], A[parent(i)]);
          i = parent(i);
      }
```

```
max_heap_insert (A, key)
{
    A.heapsize++;
    A.heapsize = -∞;
    heap_increase_key (A, heap_size, key)
}
```

Circular

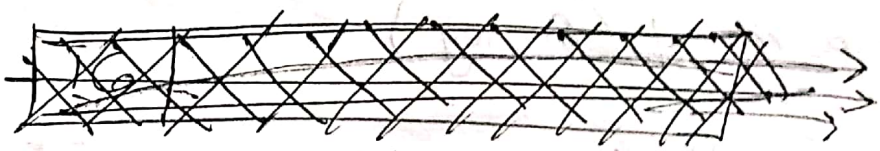
~~Double Ended~~ Queue

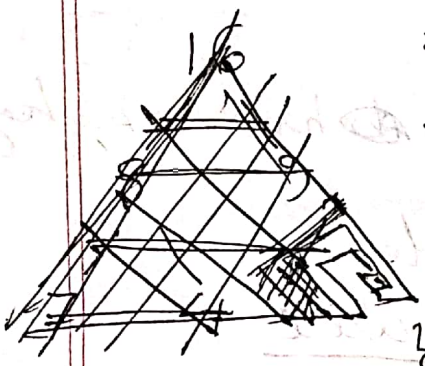
```
void enque(item)
    if (q_size > queue_max)
        error "Max size exceeded"
    else q[rear] = item
        rear = (rear + 1) % queue_max
        q_size++

dequeue (void)
    if (q_size == 0)
        error "Queue is empty"
    else
        item = q[front]
        front = (front + 1) % queue_max
        q_size--
    return item
```

#6

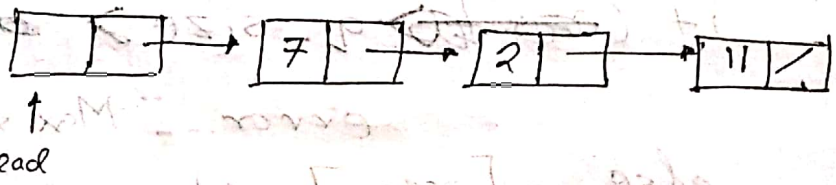
# Linked List

Monday





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




Removing element from linked list

```
Node* remove_head (node *head)
    head = head->next
    return head
```

~~Inserting element into linked list~~ if ( ) remove_head

```
node* remove_value (node *head, int key)
    if (head==NULL) //error
    else
        node *temp = head
```

```
node *prev = new_node()
while (temp! = NULL and temp->data! = key)
        prev = temp
        temp = temp->next
if (temp! = NULL)   prev->next = temp -> next
```

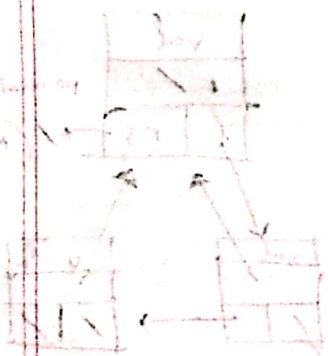
---

reverse display

```
void display (Node *node)
    if (Node != NULL)
    else {
        display (node->next)
    } cout << node -> data
```

---

```
reverse (Node *head)
    Node * prev = NULL,
    Node *current = head
    while (current != NULL)
        Node * temp = current -> next;
        current -> next = prev
        prev = current
        current = temp
```



10.4 CLRS

# Tree

BST

```
struct Node {
    int value;
    Node *left_child;
    Node *right_child;
    Node *parent;
};
```

---

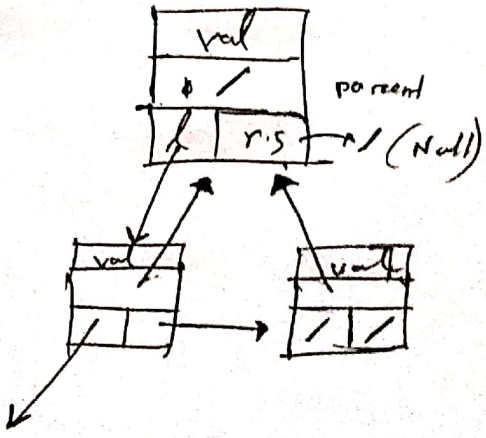
Tree

```
struct Node {
    int value;
    Node *child1;
    .
    .
    .
    Node *childn;
    Node *parent;
};
```

N-children

---

```
struct Node{
    int value;
    Node *left_child;
    Node *right_sibling;
    Node *parent;
};
```



# Complexity Comparism

| | array | Linked list | heap | BST |
|---|---|---|---|---|
| Insert | $O(n)$ | $O(1)$ | $O(\log n)$ | $O(h)$ |
| remove | $O(n)$ | $O(1)$ | $O(\log n)$ | |
| search/access | $O(\log n)$ | $O(n)$ | $O(n)$ | |

array: insert, remove — with index

search/access: $O(\log n)$ ↑ binary search

BST: $O(h)$ ↑ Height of the tree

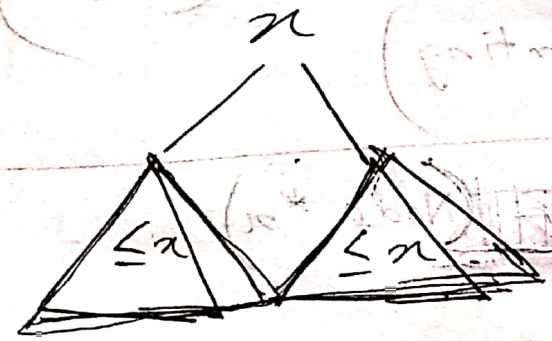
AVL

For ~~BST~~, height of tree is $\log n$.

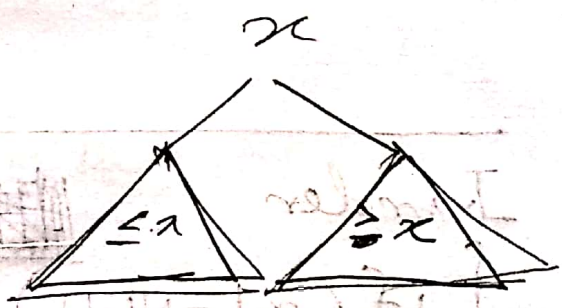
↓

A type of BST

## Max-heap



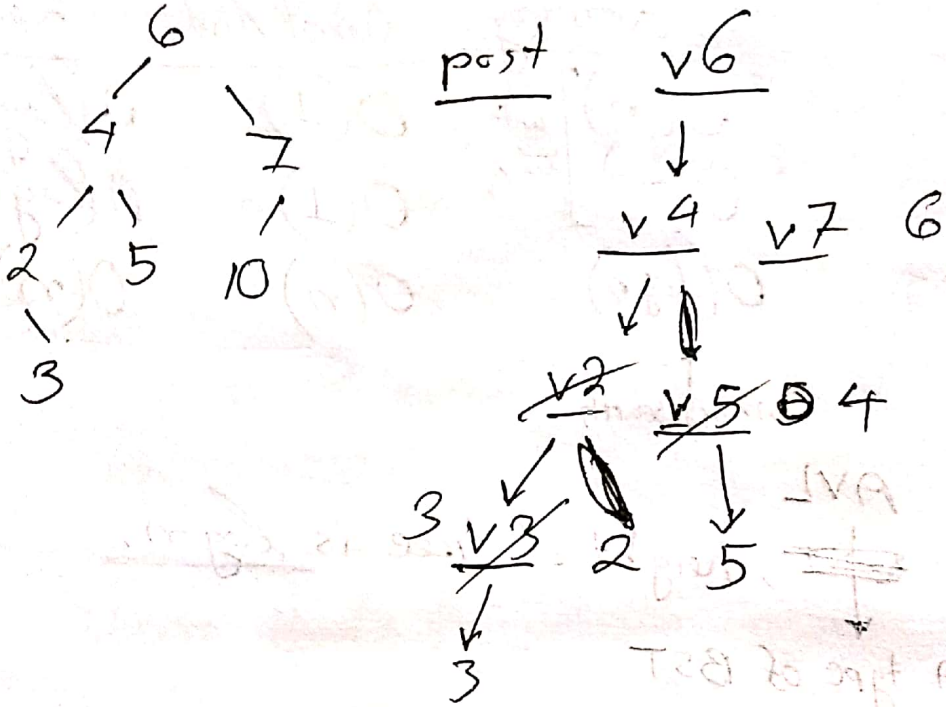

## BST




| | |
|---|---|
| Pre-order | PLR |
| Post | LRP |
| In-order | LPR |

Binary Search Tree condition is different from Binary Tree

# B.S.T




In-order traversal in BST is sorted array

BST ⟶ In-order traversal (O(n))
(sorting)

Inorder (Node *x)
{ if (x != NULL)
    inorder (x->left);
    cout << x->data;
    inorder (x->right);
}

# BST

14-02-19

BST operations

```
struct Node {
    int value;
    Node *left
    Node *right
    Node *parent
}
```

```
Node* tree_search (Node *x, int key) { if(x== NULL || x->value == key)
    return x
else if (x -> value > key)
        tree_search (x -> left, key)
else if (x -> value < key)
        tree_search (x -> right, key)
}
```

Complexity O(h)

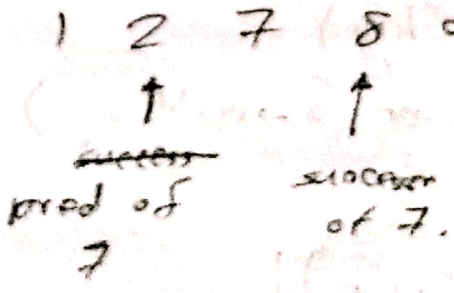
Iterative :

```
Node * tree_search (Node *x, int key)
    while (x != NULL || x -> value != key)
        if (x -> value > key)  x = x -> left
        else  x = x -> right
    return x;
```

```
Node * tree_minimum(node *x) {
    if (x->left != NULL and x != NULL) {
        tree_minimum(x->left) }
else return x;
    }
```

Successor: Smallest value greater than a certain number.

Predecessor: Largest value smaller than a certain number.




Ancestors, Descendants

Successor → (i) go to right child

(ii) Start looking for minimum till NULL

(iii)

```
Node* tree_successor (node *x)
        if (x -> right != NULL)
                return Tree_minimum (x -> right)
        p = x -> parent
        while (p -> value < x -> value and p != NULL)
                p = p -> parent
        return p;
```

```
insert_tree (node *root, int x)
{
    node *temp = create_node(x);
    node *current = root;
    while (current != NULL)
    {
    if (current -> data ≥ x) current = current -> right
```

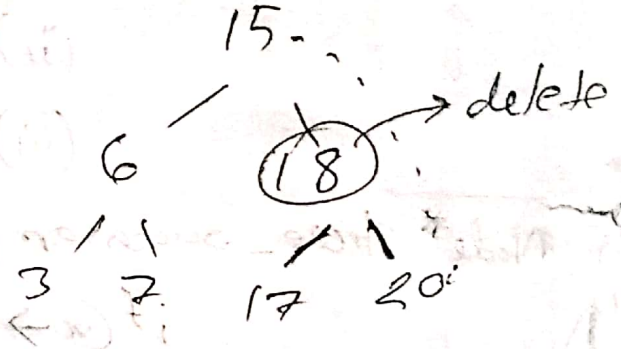
(concept (go left/right till you reach NULL)
(insert in the NULL place)

(*) make sure you appoint the parent pointer.

similiar to linked list deletion

void delete_tree node

{




}

Case 1: No child node deletion.

→ just ~~delete the parent~~ change the child of the parent to NULL

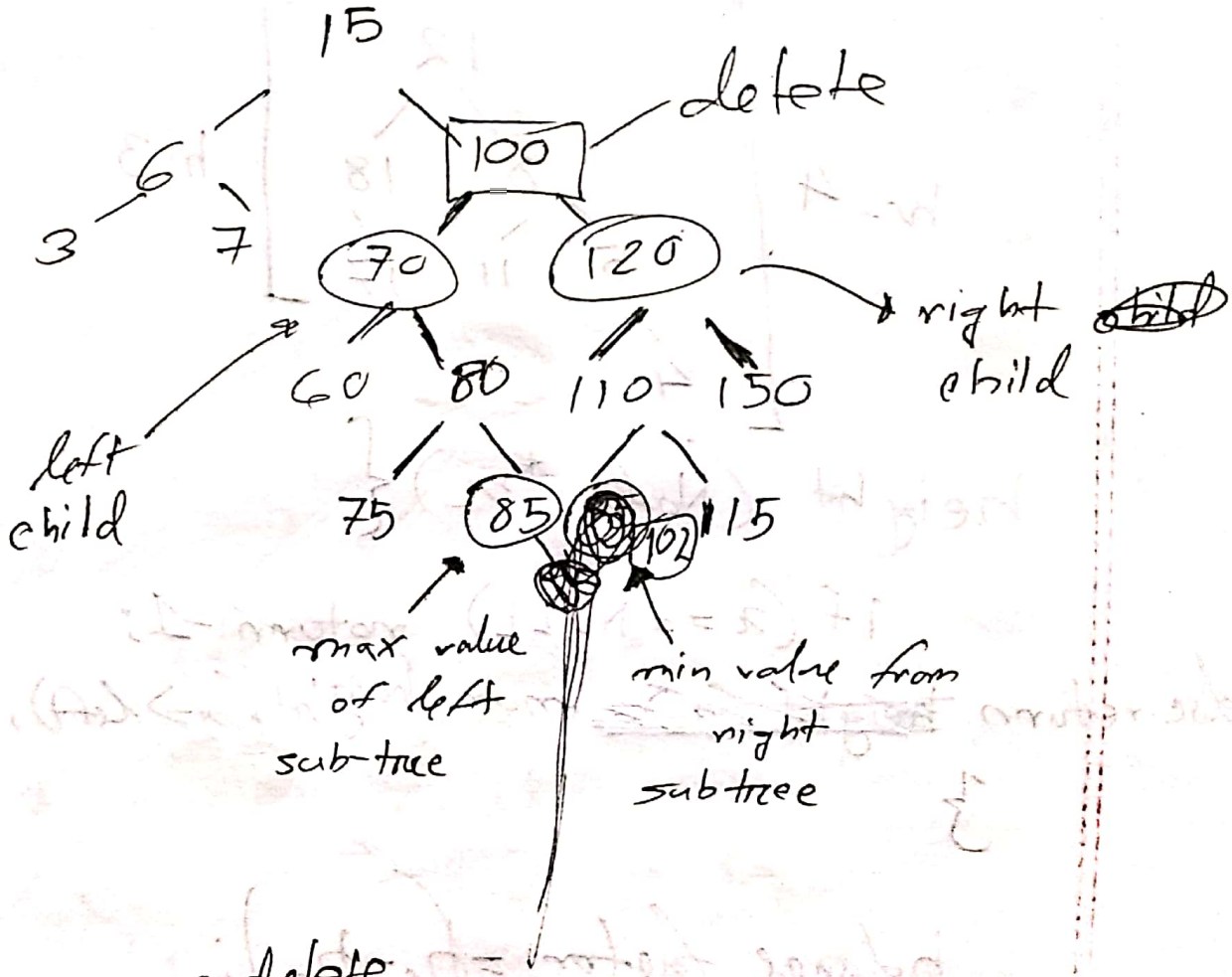
Case 2: Single child

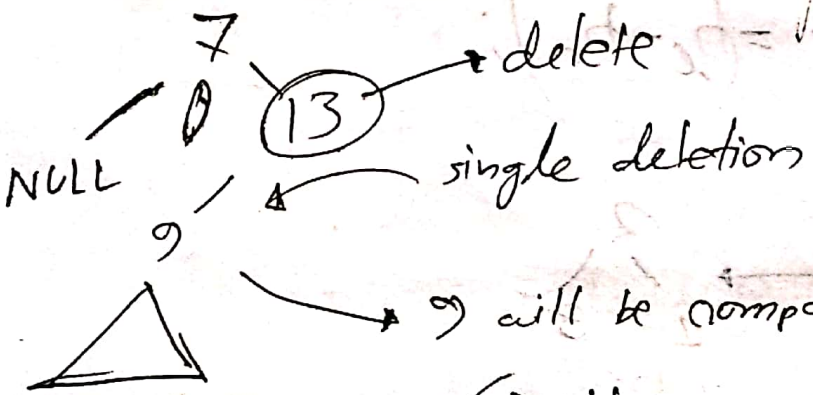
→ just like linked list

Case 3: Two child deletion.

replace with → max value from left sub-tree

→ min value from right sub-tree.



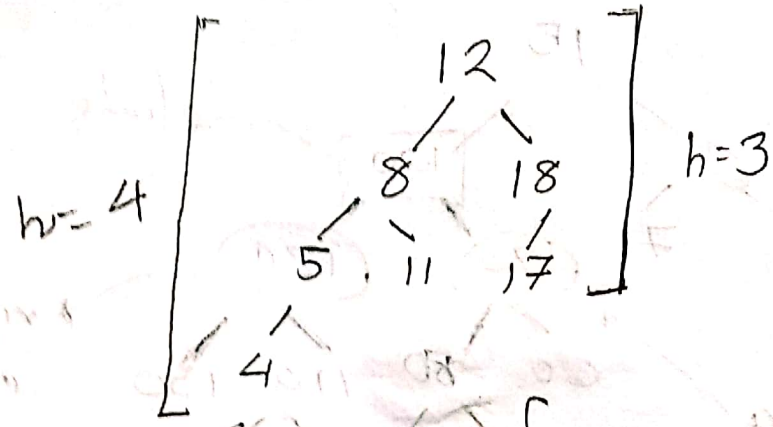




9 will be compared with 7.

(> then go right)

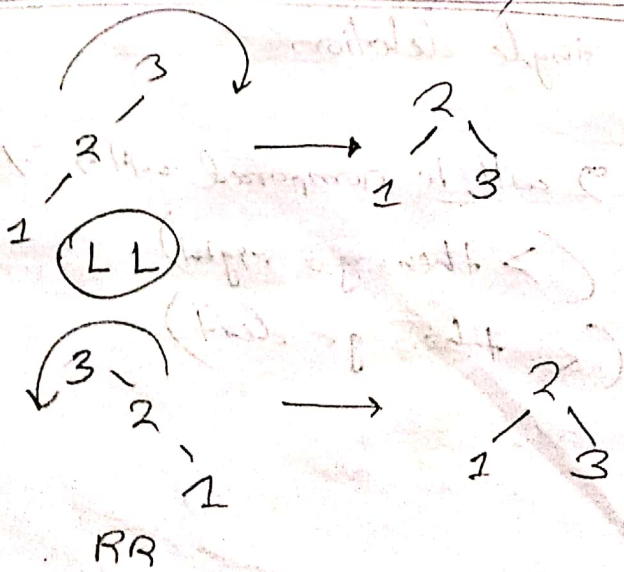
(< then go left)

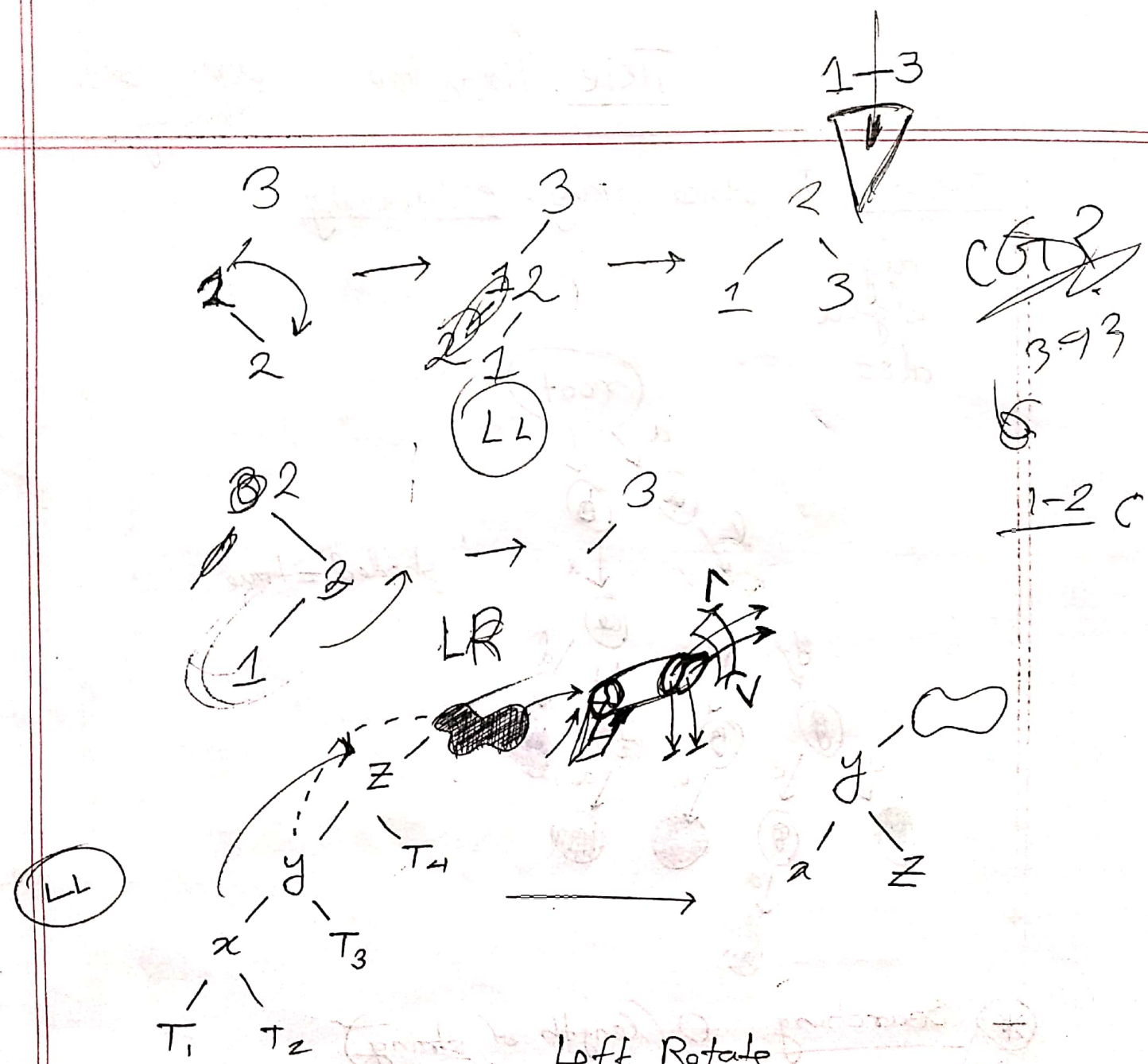
# AVL Tree




```
height (Node *x){
    if (x == NULL) return -1;
else return max(height(x->left), height(x->rib)) +1;
}
```

balance factor = $(h_l - h_r)$;






## Left Rotate

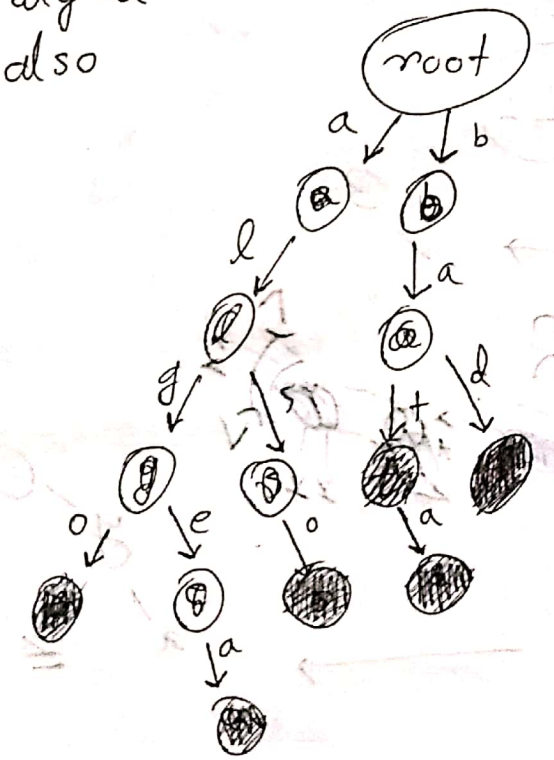
2 line (i) Parent of y should be parent of z and vice-versa.
2 line (ii) ~~Left~~ Right child of y is z and $T_4$ parent is y.
2 line (iii)

# Trie / Prefix Tree

09/03/2020
Monday

**Purpose:** To store strings effeciently

algo
algea
also




Searching O(length of string)

(*) Customization

Valid words

↳ Add new property to Nodes

Bool flag

↳ true means valid words
↳ false by default means invalid.

Implementation:

```
struct Node{
    bool endmark
    node *next[26]
}
```

```
Node* create_node (void)
{
    Node * temp = new Node();
    temp.endmark = false;
    for (int i=0; i<26; i++)
        temp->next[i] = NULL
    return temp;
}
```

```
void insert(char *str, int len), node *root)
    node *current = root
    for (int i=0; i < length; i++)
    {       id = str[i] - 'a'
            if (current->next[id] == NULL)
            {       current->next[id] = create_Node();
            }
```

```
        current = current->next[id];
    }
    current->endmark = True;
}
```

---

```
bool Search (char *str, int len)
    node *current=root
    for (i=0 ____ len-1)
        id = ____
        if (current->next[id] == NULL
            return false;
        current = current->next[id]
    return true;
    if (current.endmark == True)
        return True;
    else return false;
```

---

```
int main()
{
    node *root= create_Node(void)
}
```

void delete

---

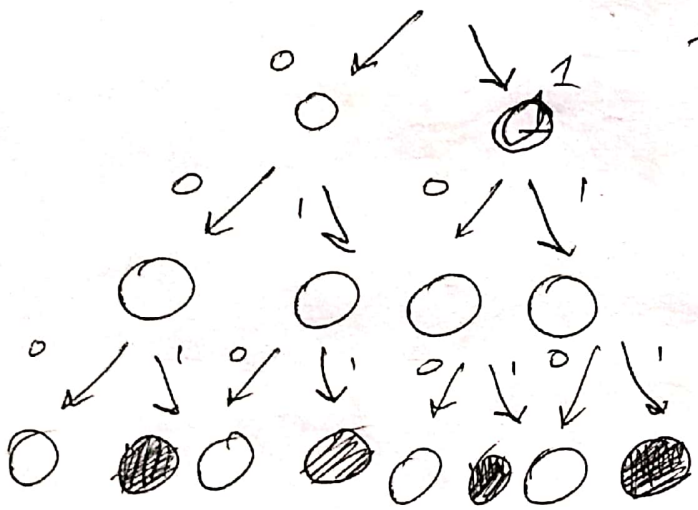
prefix counter

| | bat | after inserting bata | after inserting bad |
|---|---|---|---|
| b | 1 | 2 | 3 |
| a | 1 | 2 | 3 |
| t | 1 | 2 | d = 1 |
| a | | 1 | |

* longest common prefix

→ the ~~counter~~ characters upto which the counter value is same.

Using triad to store binary values




max value for XOR operation

3 → 0 1 1
5 → 1 0 1
7 → 1 1 1
1 → 0 0 1

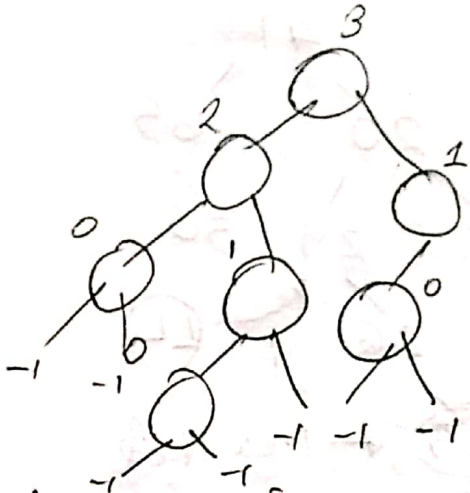
For 0 0 1

looks for 1 1 0

goes to 1, then 1, there is no zero. So, goes to 1.

Ans: 111

# AVL Tree



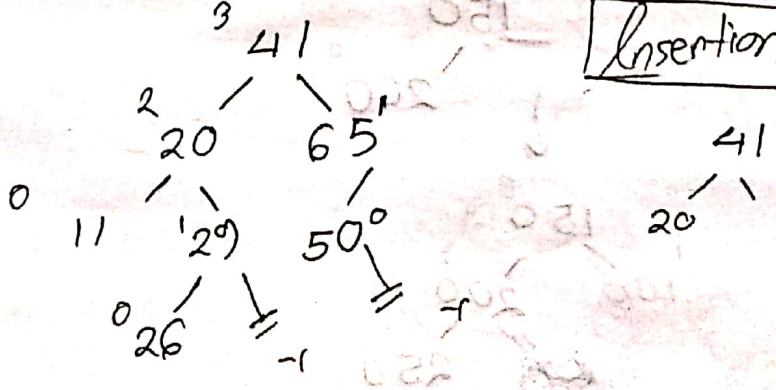

height_of_node = max{ height_of_node(left), height_of_node(right) } + 1.

~~For leaf nodes, height = 0.~~

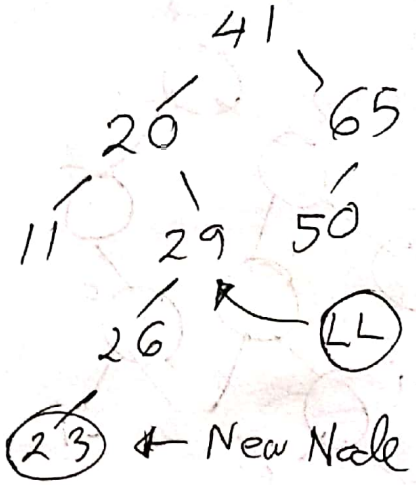
If (node == NULL) height = (-1).

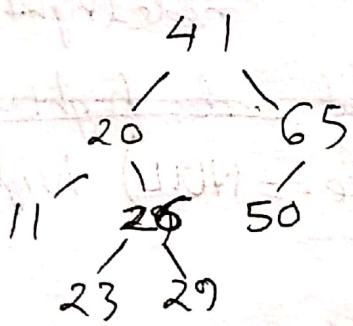
## AVL tree

Require heights of left and right children of every node to differ by at most 1. (±)

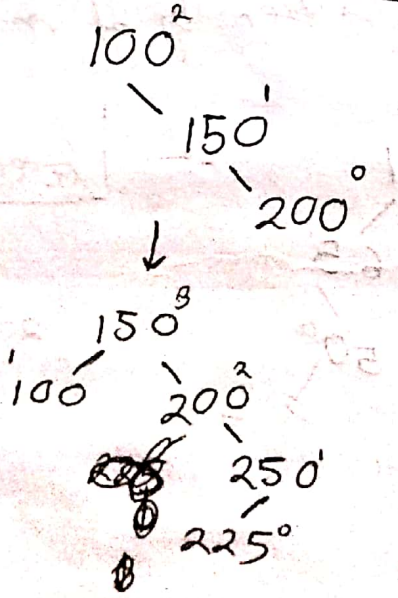


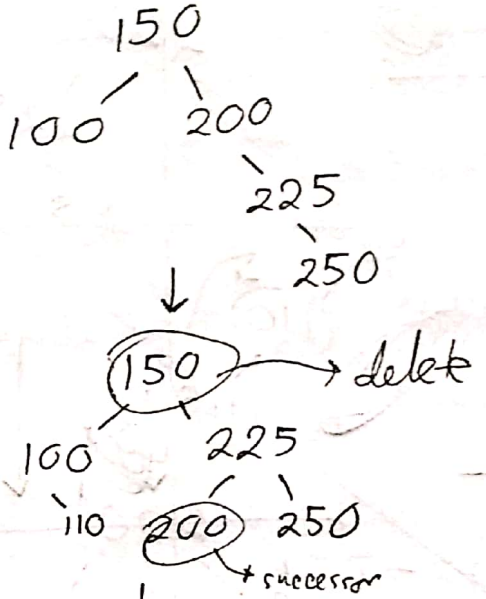

## Insertion:



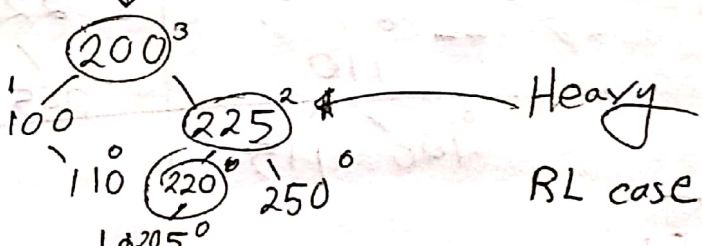



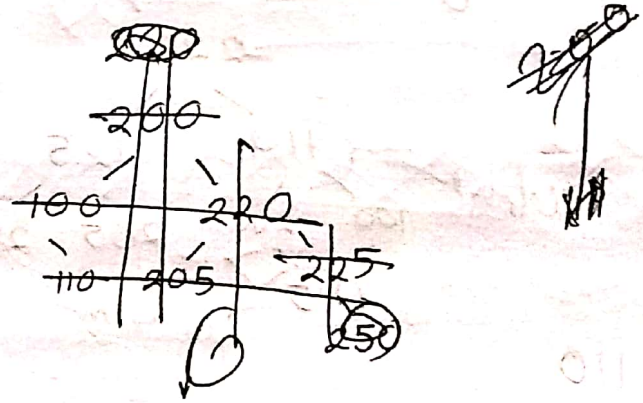

## AVL Tree (Mid)

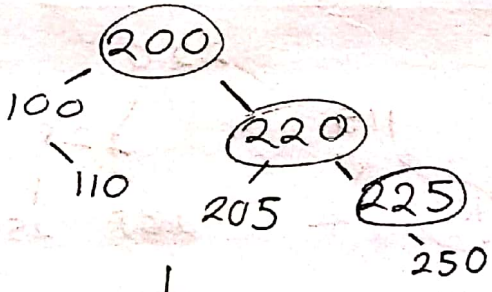


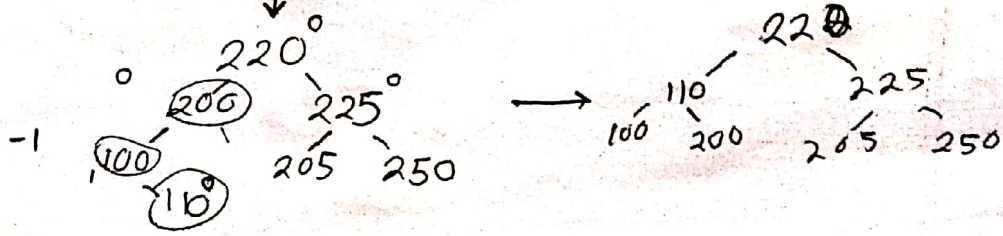



To see which case is being violated, just find the longest path or heavy path.



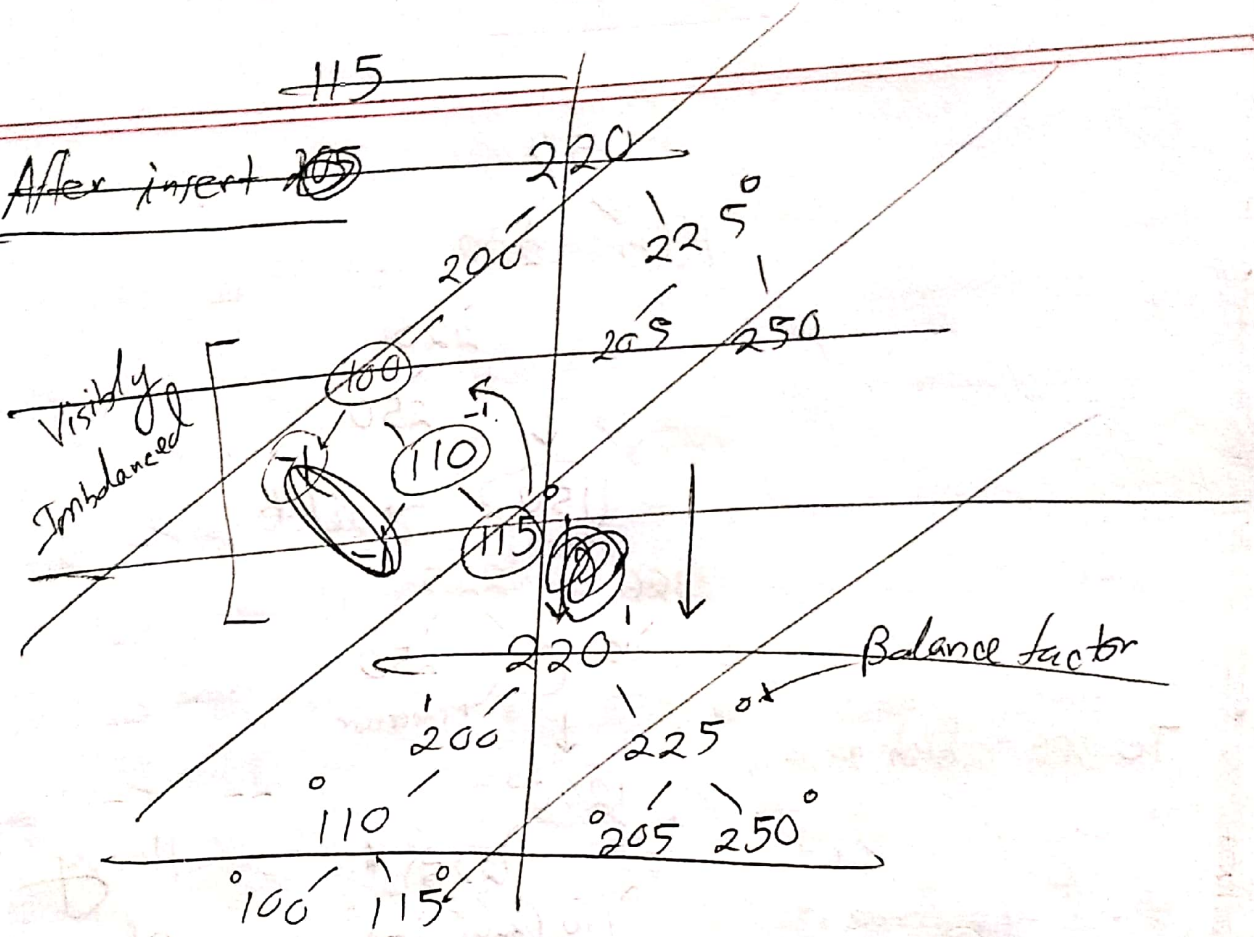



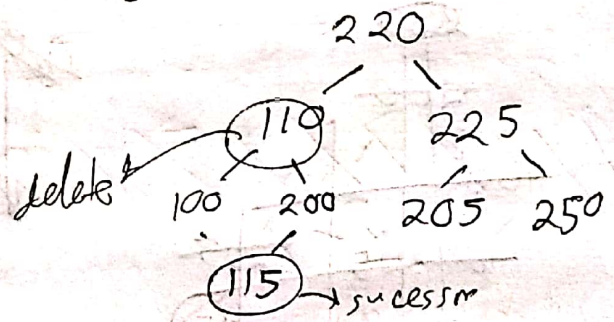



115

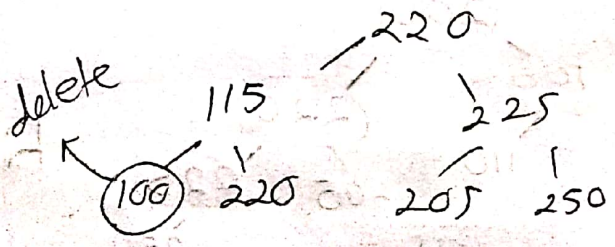
After insert ~~105~~




After inserting 115




Delete 110



Delete is like BST