DATA STRUCTURE

"Not too Detailed Not too Short" Notes

INDEX

By Quantum City

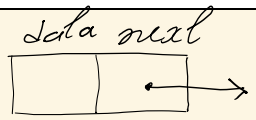
1. Linked list, Complexity, Stack and Queue

//Lecture 1

1.1) Linked list :

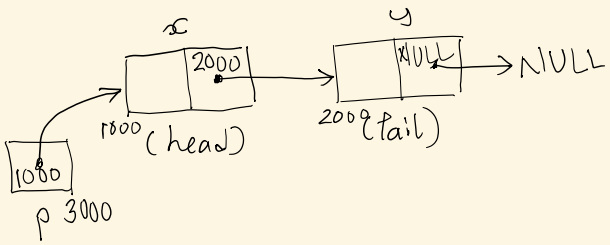
Linked list is a structure which consists of data and pointer.

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



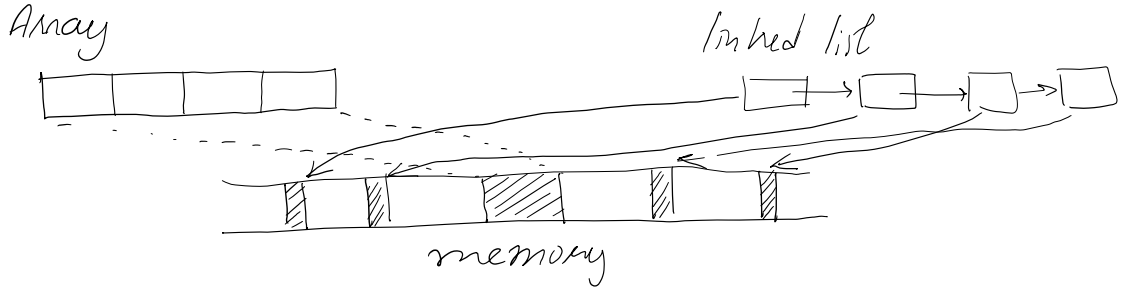
What happens in memory when we write : Struct node x; ? – it will assign memory block according to the size of struct that you have learned in c programming. That memory block will be consisting of data and next pointer.

```
struct node{
    int data;
    struct node *next;
}x, y, *p;
...
p = &x;
x.next = y;
y.next = NULL;
```



but why we need linked list ? – LL is used in implementing playlist in music app. In browser pages to get back and forward. And in operating system, we navigate from one tab to another tab or process.

1.1.1) Array vs Linked list :



In case of array memory assigned is continuous and in case of linked list it is not. So, when we access any element of array operating system will fetch few continuous blocks from RAM to cache so first access will be slow and then every access will be fast because OP have taken data into cache which is superfast but in case of linked list as memory allocated is not continuous every access will take time.

But we can increase the size of linked list while size of array is fixed at the time of compilation. This is main advantage of linked list over array. But linked list uses too much memory. And random access is not possible in linked list as we have traverse whole list just to get the last element.

	Array	Linked list
Cache locality	☑	✗
No. of elements dynamic	✗	☑
Memory	✗	☑

Random access	☑	✗

1.1.2) Operations on linked list :

1) **Finding length of linked list** : count until head is null

```
int length(struct node *head){
    int count = 0;
    while(head != NULL){
        count ++;
        head = head->next;
    }
    return count;
}
```

2) **Printing linked list** : print until head is null

```
int length(struct node *head){
    while(head != NULL){
        printf("%d", head->data);
        head = head->next;
    }
}
```

3) **Insertion at beginning** :

```
int insertatbegin(struct node *head){
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = 13;
    newNode->next = head;
    head = newNode;
    return 0;
}
```

Not required but good practice

4) **Insertion at end** :

```
int insertatend(struct node *head){
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = 13;
    newNode->next = NULL;
    while(head->next != NULL) head = head->next;
    head->next = newNode;
    return 0;
}
```

5) **Insertion at the middle** : Suppose middle element has data 23 then

```
int insertatmid(struct node *head){
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = 13;
    newNode->next = NULL;
```

```
    while(head->next->data != 23) head = head->next;
    newNode->next = head->next;
    head->next = newNode;
    return 0;
}
```

6) **Deleting first node** : We can simply move head to head->next but in this way memory leak problem will be there.

```
int deleteatbegin(struct node *head){
    if(head == NULL)return 0;
    if(head->next == NULL){
        free(head);
        head = NULL;
    }else{
        struct node *temp = head;
        head = head->next;
        free(temp);
    }
    return 0;
}
```

7) **Deleting last node** :

```
int deleteatend(struct node *head){
    if(head == NULL)return 0;
    if(head->next == NULL){
        free(head);
        head = NULL;
    }else{
        while(head->next->next != NULL) head = head->next;
        free(head->next);
        head->next = NULL;
    }
    return 0;
}
```

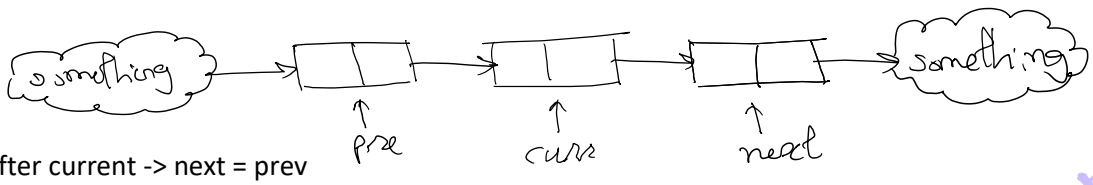
8) **Deleting intermediate node** :

First find that node and point to before that node then curr->next = curr->next->next; and then delete curr->next.

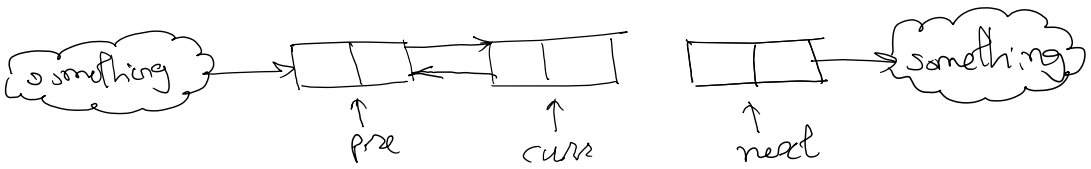
You can also apply recursion on these functions instead of while loop.

//Lecture 4a

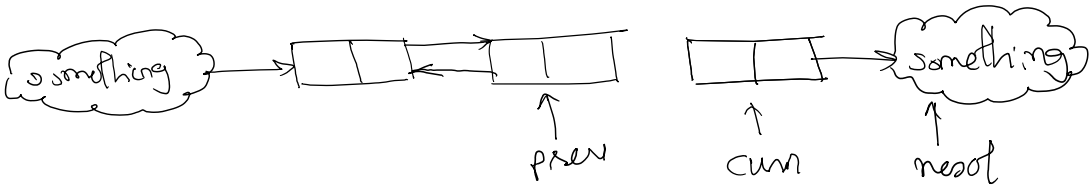
1.1.3) Reverse a linked list :



After current -> next = prev



After, prev = curr; curr = next; next = next->next;

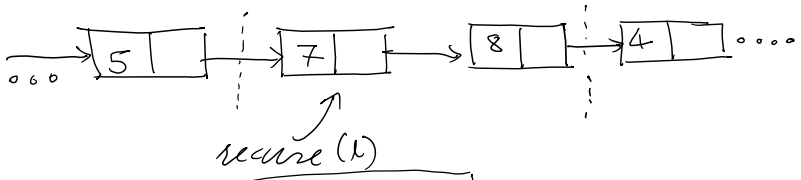


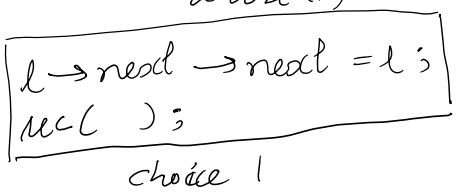
We do this until curr == NULL

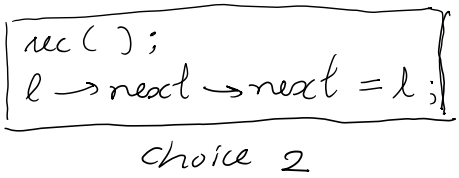
```
Node *reverseIterative(Node *head){
    if(head == NULL || head->next == NULL) return head;
    Node *prep = NULL;
    Node *currp = head;
    Node *nextp = head->next;
    while(currp){
        currp->next = prep;
        prep = currp;
        currp = nextp;
        if(nextp) nextp = nextp->next;
    }
    return prep;
}
```

//Lecture 4b

Reverse a linked list using recursion :







When we follow choice 1 templet in recursion then we will loss out connection with 8 node. So somehow, we need to go at the last node and do recursion of l->next->next = l; So, that is why second choice is best.

```
void *reverserecursive(Node *head){
    if(head == NULL || head->next == NULL)return;
    reverserecursive(head->next);
    head->next->next = head;
}
```

But after executing this function our head is pointing to last node because we have done reverse so now our first node is last and our last node is not pointing to NULL. Thus, in main function we carry one pointer which points to last element of our original linked list.

```
int main(){
    ...
    Node *last = head;
    while(last->next)last = last->next;
    reverserecursive(head);
    head->next = NULL;
    head = last;
    ...
}
```

Another method without last node and head->next = NULL in main function is to return head pointer in reverserecursive function call. And after last recursive call head->next should points to NULL. So final recursive program will be

```
head *reverserecursive(Node *head){
    if(head == NULL || head->next == NULL)return head;
    Node *n = reverserecursive(head->next);
    head->next->next = head;
    head->next = NULL;
    return n;
}
```

Here in above example, time complexity is O(n) as it traverses whole linked list. But space complexity is not O(1) it is O(n) because each recursive call takes O(1) time and there are n such. Consider one code where only while loop runs O(n) times then space complexity is O(1) as only that program is getting pushed into stack but in case of recursive calls n activation records are being pushed into stack.

//Lecture 4c

1.1.4) Type of linked list :

- **Circular linked lists** : Circular list is the simply linked list with last node points to head of the linked list.
- **Doubly linked list** : We can navigate in both direction

//Lecture 5a

1.2) Asymptotic Analysis :

Till now, we have seen linked list and operation on it. So, it is logical to ask that which operation takes more time ? And what if operations are too large to fit in memory ?

why is my program so slow?

why does it run out of memory?

Analyzing running time :

- Can be run on different computers
- Different clock speed
- Different instruction set
- Different memory access speed

} A posteriori analysis → Platform dep.

The goal of asymptotic analysis is to simplify analysis of running time by getting rid of "details". Like rounding : 1,000,0001 = 1,000,000 and $3n^2 = n^2$.

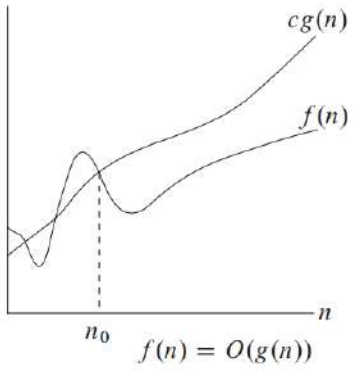
//Lecture 5b

1.2.1) Big oh and big omega asymptotic notations :

Now, we are going to represent time complexity and space complexity with a funtion of input size.

1) Big-oh O notation :

$T(n)$ is $O(g(n))$ if there exist constant $c > 0$ and $n_o \geq 0$ so that for all $n \geq n_o$, $T(n) \leq cg(n)$. Provides asymptotic upper bound.



$f(n) = n^2$ & $g(n) = 2^n$ we say $f(n) = O(g(n))$

	n^2		2^n
1	1	<	2
2	4	=	4
3	9	>	8
n_0 → 4	16	=	16
5	25	<	32
6	36	<	64
7	49	<	128

2) Big Omega Ω Notation :

$T(n)$ is $\Omega(g(n))$ if there exist constant $c > 0$ and $n_o \geq 0$ so that for all $n \geq n_o$, $T(n) \geq cg(n)$. Provides asymptotic lower bound.

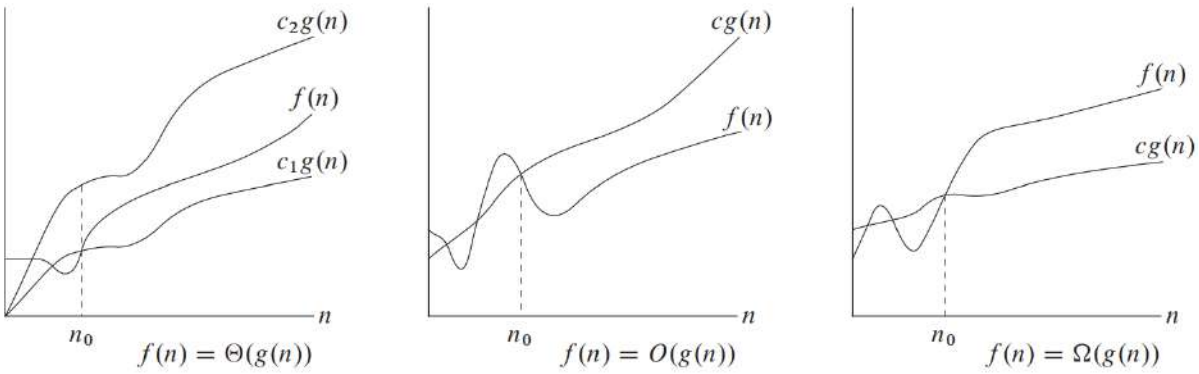
We can say that $T(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(T(n))$.

//Lecture 5c

3) θ Theta notation :

For a given function $g(n)$ we denote by $\theta(g(n))$ the set of functions:

$$\theta(g(n)) = \{f(n): there\ exist\ positve\ constants\ c_1, c_2, and\ n_o such\ that\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ for\ all\ n \ge n_o\}$$



//Lecture 5d

What does it mean to say Aysmptotically larger ? – It means we are ignoring constants and looking at significant terms. Example,

$n^2 + 1000^{1000^{1000}} \propto n^3$ ... after ignoring constant

Clearly, $2n^2 > n^2$ but asymptotically they are equal. but $n^{5+n} \neq n^n$ because $n^{5+n} = n^5 . n^n \neq n^n$.

Q : Prove that n^ϵ is asymptotically larger than $(\log n)^k$ where $\epsilon < 0$ and $k > 0$. –

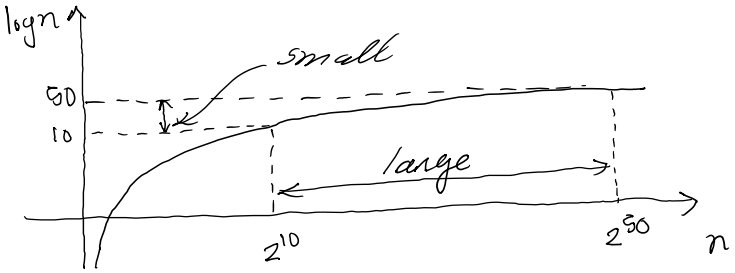
$n^\epsilon \ ? (\log n)^k \longrightarrow \epsilon \log n \ ? \ k \log\log n \longrightarrow \log n \ ? \ \log\log n$

Take $n = 2^m$ where $m > 0$,

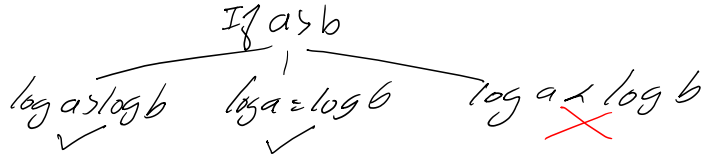
$m \ ? \ \log m$ clearly $m > \log m$ Therefore $n^\epsilon > (\log n)^k$

//Lecture 5e

But, we know that $n^3 > n^2$ but still after taking log we get log n ? log n. Clearly both are equal but we know that it is wrong. So, when to take log and when not to ?



If you see carefully after taking log the value become so small that we cannot say if they are asymptotically not equal. which means <u>if after taking log things are far then before taking log they would be very very far.</u> Thus, if we encounter situations when a > b then we cannot say that log a > log b but <u>if log a > log b then we can definatly say that a > b but we can't say a = b</u>. We can say,



Some operation which always works :

- Cancel out common terms (multiply or divide)
- Ignore constant only when c f(n) form otherwise never ignore.
- And for log we have already discussed.
- Sometimes we have to assume n(ex. 2^{128}, 2^{256}, 2^{1024},…)

//Lecture 5f

4) o little-oh :

Defintion : $T(n)$ is $o(g(n))$ for any constant $c > 0$ there is $n_o > 0$ so that for all $n \geq n_o$

$$T(n) < cg(n)$$

See the definition of big O in which there exists c is written wherein little-oh has any constant means every it should be true for every c.

That is $n^2 \neq o(2n^2)$ because for c = ½. Both are equal but we want strickly greater.

5) ω little-omega :

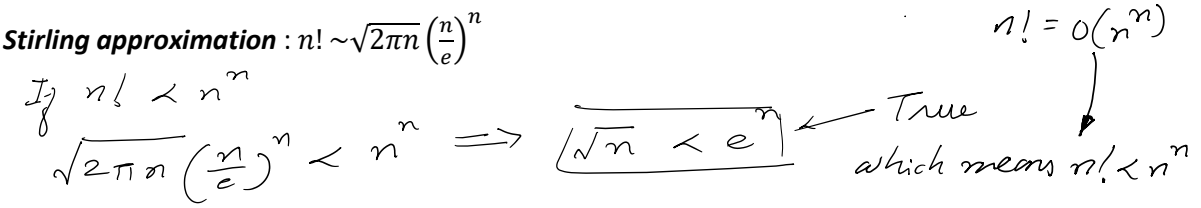
Definition : $T(n)$ is $\omega(g(n))$ for any constant c>0 there is $n_o > 0$ so that for all $n \geq n_o, T(n) > cg(n)$

In this definition also for any constant is written.

And we can observe one thing that $T(n) = \omega\big(g(n)\big)\ if\ and\ only\ if\ g(n) = o(T(n))$

Imcomparability : $n^{1+\sin(n)}$ is incomparable to n since sin(n) oscillates between -1 and 1.

Stirling approximation : $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$



But if we take log then, lg (n!) = lg(n) + lg(n-1) + …. + lg(1)

Which means lg(n!) <= lg(n) + lg(n) + … n times ➔ lg(n!) <= nlgn

And now, lg(n!) = lg(n/2) + lg(n/2) + … till n/2 terms and then + lg(n/2 -1) + … + lg(1)

This will surely be less than nlgn. ➔ lg(n!) >= nlgn

Combining these two we get $\boldsymbol{\lg(n!) = \theta(nlgn)}$

NOTE :

1) **If we say $n^2 = O(n^3)$ formally it is $n^2 \in O(n^3)$ because if you see at definition also $O(n^3)$ is set of function.** ← set ← element of $O(n^3)$
2) **All the asymptotic notions are sets. But we use = ambiguously**.

Q : For any two non-negative function f(n) and g(n), both of which tends to infinity, we must have either f(n) = O(g(n)) or g(n) = O(f(n)) ? – This seems true but you can have f(n) = n^2 and g(n) = n for even input and n^3 for odd inputs. Here g(n) goes up and down and is making spiral shape with f(n) so it is sometimes O(f(n)) and sometimes Ω(f(n)).

//Lecture 6c

1.2.2) Analyzing the loop time complexity :

```
main(){
    for(int i = 0;i<N;i++)
        for(int j = i+1;j<N;j++){
            statement;
        }
}
```

$i=0 \quad j=1 \to N \Rightarrow N$
$i=1 \quad j=2 \to N \Rightarrow N-1$
$i=2 \quad j=3 \to N \Rightarrow N-2$
$\vdots$
$i=N-1 \quad j=N-N \Rightarrow 1$

$\frac{n^2+n}{2} = O(n^2)$

In above case we do loop unrolling.

```
for(int i = 1; i<N; i+=2){
    sum += i;
}
```

$i = 1, 1+2, 1+2+2, 1+2+2+2, \ldots$
at k^{th} iteration $i = 1 + 2k$

In this example we say after k step i becomes equal to N and then loop gets over. N = 1 + 2k meaning k = (N-1)/2 and we know that complexity is O(k) (why?) = O(N/2) = O(N)

```
for(int i = N; i<=1; i-=2){
    sum += i;
}
```

In first example, $i = 1 \xrightarrow{+2} 3 \xrightarrow{+2} 5 \xrightarrow{+2} 9 \xrightarrow{+2} 11 \ldots N$ $\quad O(N/2)$

In second example, we are doing reverse

Meaning time complexity will still remains same i.e. O(N/2) = O(N)

Thus, to find time complexity first assume O(k) which means after k step condition will false and then find relation between k and n.

```
for(int i = 2; i<=n; i=i*i){
    sum += i;
}
```

$i = 2, 2^2, (2^2)^2, (2^4)^2, (2^8)^2, \ldots, 2^{2^k}$
$2^4 \quad 2^8 \quad 2^{16}$

$2^{2^k} = N, \quad k = \log\log n, \quad O(k) = O(\log\log n)$

```
for(int i = 1; i<=N; i*=2){
    for(int j = 1; j<=i*i; j++){
        sum++;
    }
}
```

$i = 1 \to 1$
$i = 2 \to 2^2$
$i = 4 \to 4^2 = 2^{2\cdot 2}$
$\vdots$
$i = 2^k \to (2^k)^2 = 2^{2\cdot k}$

thus i loop runs

We again assume that i loop terminates after k iteration at the end of k iteration the value of $2^k = N$.

$Total = 1 + 2^2 + 2^{2\cdot 2} + 2^{2\cdot 3} + \ldots + 2^{2\cdot k} = 2^{2\cdot k} = O(N^2)$

```
for(int i = 0;i<n; i*=2) sum += i;
```

first look goes initial condition

In above case, as the value of i will remain zero because of initial condition so this is infinite loop.

In short, we can say that

***Increment* : O(n) if (i = i + c)**

Doubling **: O(log n) if (i = 2 * i)**

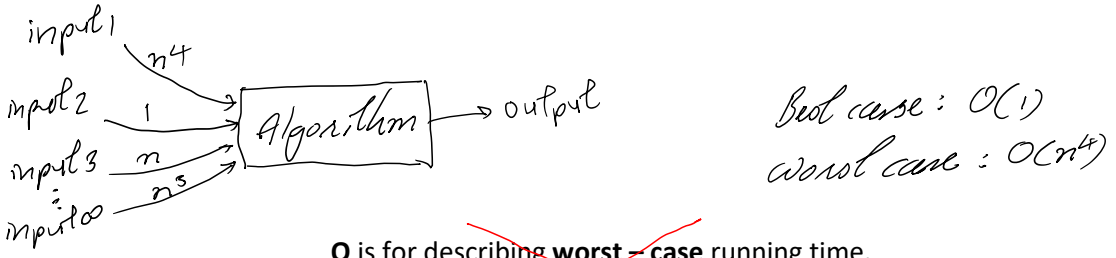
Exponentiation **: O(lg lg n) if (i = i * i)**

//Lecture 7a

1.2.3) Best case, Worst case :

Best case : Whatever be the minimum time for any possible input. Even on 1 input you are taking O(1) and all other inputs you are taking $O(n^4)$ then best case time complexity is O(1).

Worst case : Whatever be the maximum time for any possible input. Even on one input you are taking $O(n^3)$ and all other inputs you are taking O(n) worst case time complexity $O(n^3)$.



O is for describing **worst – case** running time.

Ω is for describing **best – case** running time.

//Lecture 7c

Q : Consider an algorithm A which takes $\theta(n)$ in best case and $\theta(n^2)$ in worst case. Then which of the following is/are true ?

1) Algorithm time complexity is $O(n^2)$ – true
2) Algorithm time complexity is $\Omega(n)$ – true
3) Algorithm time complexity is $O(n^3)$ – true
4) Algorithm time complexity is $\Omega(n^2)$ – False because it means for all cases time complexity >= n^2 but worst case is n^2 so not possible.
5) Algorithm time complexity is $\Omega(1)$ – true
6) Algorithm best case time complexity is $\theta(n)$ – true
7) Algorithm worst case time complexity is $\theta(n^2)$ – true
8) Algorithm best case time complexity is $O(n)$ – true
9) Algorithm worst case time complexity is $\Omega(n^2)$ – true
10) Algorithm best case time complexity is $O(n^2)$ – true

//Lecture 8a

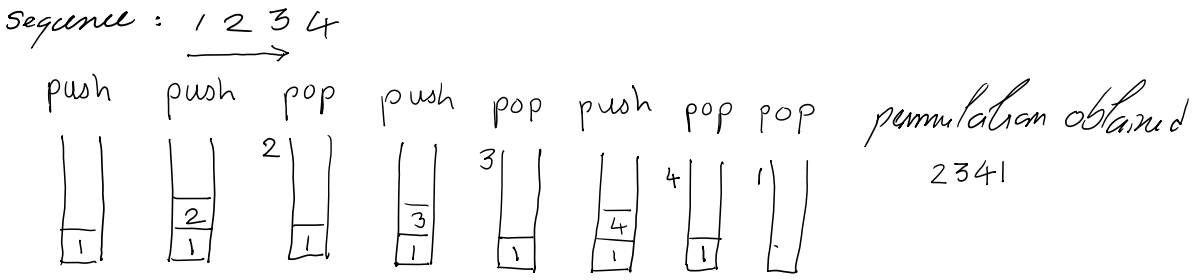
1.3) Stack and Queue :

Abstract data types : An abstract data type (ADT) specifies the operation that can be performed on the collection. It's abstract because it doesn't specify how the ADT will be implemented. A given ADT can have multiple implementations. For example, linked list, stack, queue.

1.3.1) Stack :

Why stack ? – some of the application includes reverse a word, undo mechanism in text editors, function calls, expression evaluation, balancing parenthesis.

Stack permutation of a sequence : It is a permutation obtained after pushing and popping every alphabet of sequence. Here pushing and popping can be done at any time meaning between alphabet.



Number of stack permutations : also known as Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$

//Lecture 3b

Implementing stack :

We can implement stack using array. we take array of some size and a point to index which initially have value -1.

```
push(k){
    top++;
    if(stack.size() == N) "Overflow", return;   → O(1)
    a[top] = k;
}
pop(){
    if(top == -1) "Underflow", return;          → O(1)
    return a[top--];
}
```

Using linked list, we can also implement stack.

//Lecture 9a

1.3.2) Introduction to queue :

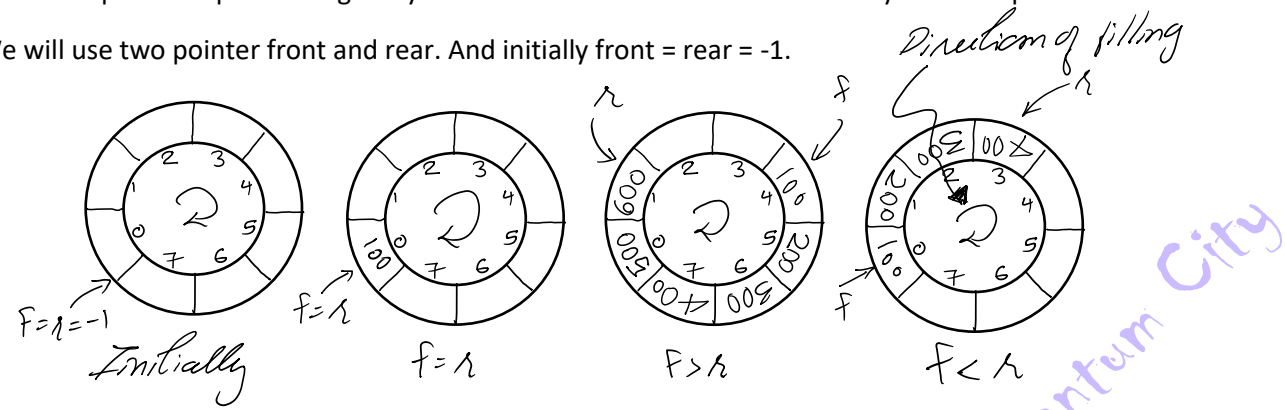
Why Queue ? – Sometimes it is also desirable to access the element which got inserted first. For example, in networking, call center phone systems, operating system.

//Lecture 9b

1) **Implementing queue using array** :

We can implement queue using array but it is not as effective as circular array-based implementation.

We will use two pointer front and rear. And initially front = rear = -1.



How to check if

- *Queue is full* – when (r+1) mod n = f
- *Queue is empty* – When f = -1 or r = -1 or both.

```
Enqueue(data){
    if((rear + 1) % N == front){
        printf("Queue is full");
        return;
    }
    front = (front == -1) ? 0 : front;
    rear = (rear+1) % N;
    array[rear] = data;
}

Dequeue(){
    if(front == -1){
        printf("Queue is empty");
        return;
    }
    data = array[front];
    if(front == rear) front = rear = -1;
    front = (front+1) % N;
    return data;
}
```

Before incrementing we check

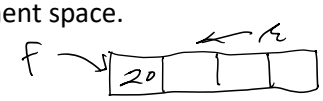
There is also operation of getsize() which returns no. of elements in queue.

```
Getsize(){
    if(front == -1 && rear == -1) return 0;
    return (front>rear) ? N-front+rear+1 : rear-front+1;
}
```

for front ≤ rear

//Lecture 9c

There is also one implementation where initially front = rear = 0 instead of -1. But in this we have wastage of one element space.



In such implementation, how to check if

- *Queue is full* – front == (rear + 1) % N
- *Queue is empty* – Front == rear == 0

//Lecture 9d

2) Implementing queue using linked list :

Enqueue( ) : Inserting an element at end of list

Dequeue( ) : Deleting an element from beginning of list

//Lecture 9e

3) Queue using two stacks :

We use one stack for insertion (S1) and one for deletion of element (S2).

For *Enqueue ( )* : we will simply push into S1

For *Dequeue ( )* : if S2 is not empty then pop(S2) and if S2 is empty we will transfer all the elements from S1 to S2 and then pop(S2).

//Lecture 10a

Stack using two queues :

For *push()* : we will simply push into non-empty queue.

For *pop()* : We dequeue n-1 element (meaning all except one) and enqueue in second queue. One element is remaining which will be top element in case of stack.

//Lecture 10b

1.3.3) Few applications of stack :

1) **Balancing parentheses** :

```
Create a stack.
while(input is finished){
    if(character is an opening delimiter like (, {, [)
        PUSH it into the stack;
    if(character is a closing symbol like ), }, ]){
        POP the stack;
        if(stack is empty) report error;
        if(symbol POP-ed is not the corresponding delimiter) report error;
    }
}
//At the end of the input
if(stack is not empty) report error;
```

2) **Two stacks in one array** :

To implement two stacks, we manage two pointer points to top elements of both stacks. After pushing elements, if we encounter situation where Top1 = Top2. We say stack overflow. Remember in this implementation we first increment top1 or decrement top2 and then store data.

3) **Infix, prefix and postfix and conversion to each other** :

Ways of writing expression :

1. *Infix notation* A * B + C / D
2. *Prefix notation* (also known as “Polish Notation”) + * A B / C D
3. *Postfix notation* (also known as “Reverse Polish Notation”) A B * C D / +

Convert infix to prefix and postfix :

Step 1 : apply appropriate brackets based on precedence and associativity.

Step 2 : Convert each inner bracket to postfix separately.

Step 3 : Combine and repeat step 2.

Example : $a+b*c-d/e*f \longrightarrow a+(b*c)-((d/e)*f)$

$(a)+(bc*)-(de/f*) \longrightarrow abc*+de/f*-$ ← postfix

$(a+b*c-d)/(e*f) \longrightarrow (a+(b*c)-d)/(e*f)$

$(-+a*bcd)/(*ef) \longrightarrow /-+a*bcd*ef$ ← prefix

Now, we will see how computer do these operations. It uses stack.

- ***Infix to postfix*** **using stack** :

```
input is -
    if('(') push in stack;
    if(')') pop until left parenthesis is popped;
    if(operator){
        lower priority is in input then pop all;
        Higher priority is in input then push;
        same priority pop except ↑ ;
    }
```

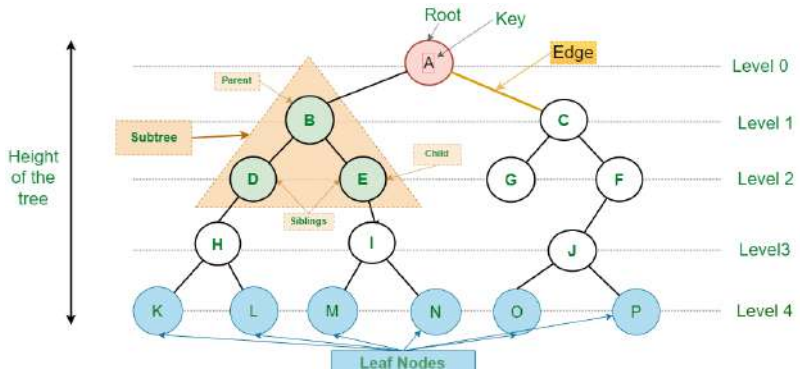
We can calculate result value from postfix operation by simply taking one empty stack we push number from left to right whenever we encounter binary operations, we pop top two elements and perform operation and then push the result again into the stack.

2. Binary, AVL tree

//Lecture 11a

2.1) Binary trees :

Binary tree is tree having at most 2 children per node.



Depth (or level) of a node : The depth of a node is the number of edges from the node to the root. Root node will have depth (or level) of 0.

There are few types of binary tree based on different conditions :

Full binary tree : In a full binary tree all nodes have either 0 or 2 children.

Complete tree : All levels except last are full. Last level is left-filled.

Perfect binary tree : Complete binary tree when last level is also full.

Remark : some authors considers complete tree as perfect binary tree but that is just matter of notion. If some strange term appears in exam they will specify.

No. of leaves = (Number of nodes with degree 2) + 1

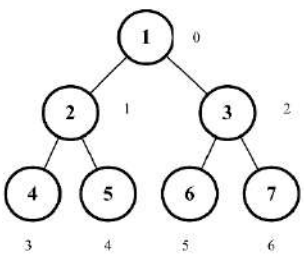
In full binary tree all internal nodes have degree 2 so here *no. of leaves = Internal nodes + 1*.

In general, in *m-ary* tree if n is total number of nodes and i is internal nodes then **n = mi + 1** and **(m-1)i + 1** leaves.

Q : The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is ? – Maximum node happen in perfect binary tree where all parents have two children. So total number of nodes in perfect binary tree at height h is 2^h and total number of nodes in binary tree is $2^{h+1} – 1$.

2.1.1) representation of tree :

1) **Array representation** :



Here indexing is from 0

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

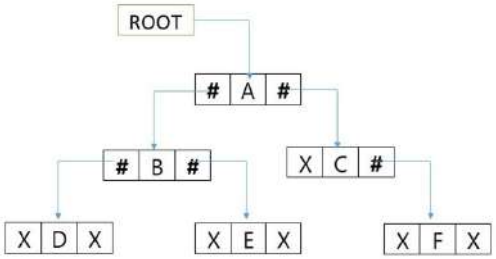
Let index of node = i

parents = $\lceil \frac{i}{2} \rceil - 1$

Left child = $2i + 1$

Right child = $2i + 2$

2) Linked list representation :



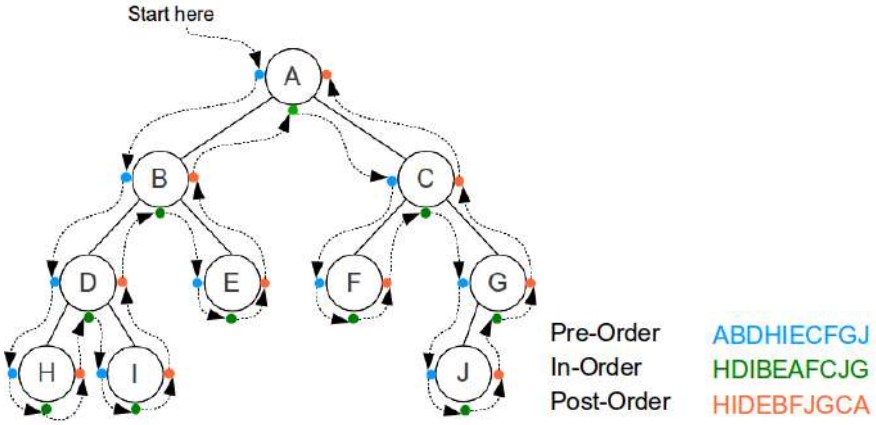

```
struct node{
    int data;           //element
    struct node *left;  //pointer to l child
    struct node *right; //pointer to r child
};
struct node *root;
root = (struct node*)malloc(sizeof(struct node));

root->data = 3;
root->left = NULL;
root->right = NULL;
```

//Lecture 12a

2.1.2) Binary tree traversals :

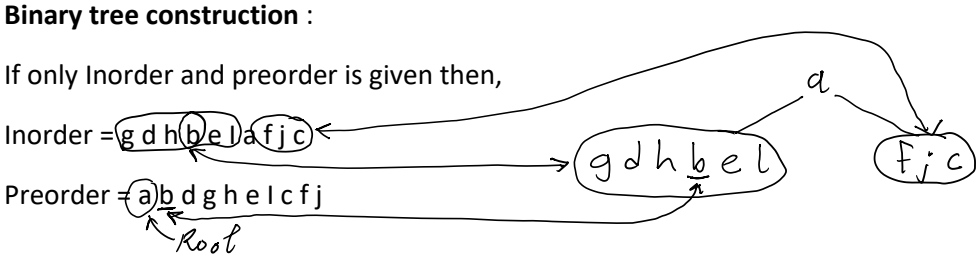
- Pre-order (root left right)
- In-order (left root right)
- Post-order (left right root)



//Lecture 12b

Binary tree construction :

If only Inorder and preorder is given then,



You know that preorder contains all root in sequence. Meaning first element of preorder is root of the tree. And inorder contains root at the middle and all left nodes on left side of that root and right nodes on right-hand side.

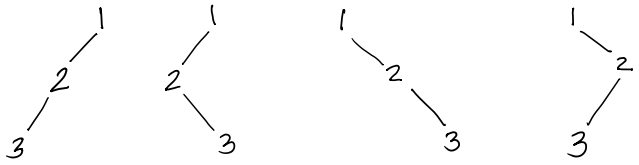
If inorder and postorder is given then we follow same procedure but instead of going left to right for selecting root we go right to left as main root of tree appears at the last in postorder.

But if only postorder or only preorder is given then we can't uniquely construct binary tree.

Q : but how many binary trees are possible with given only postorder traversal or given only preorder traversal ? – Catalan number = number of stack permutation.

//Lecture 12c

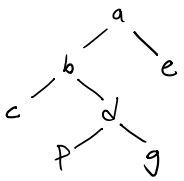
Q : But what if preorder and postorder is given is it possible to construct ? – with preorder = 1 2 3, postorder = 3 2 1, no. of possible binary tree…



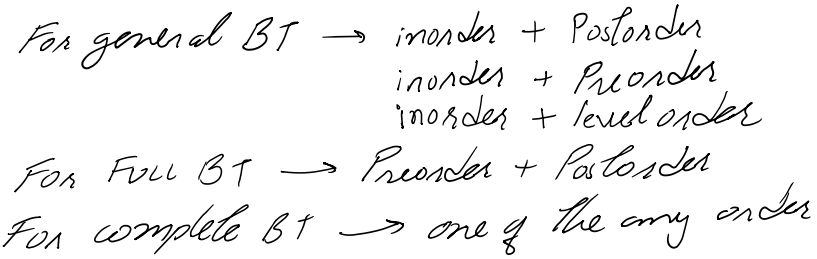
But if question was construct FULL binary tree using

Preorder = a b c d f g e

Postorder = c f g d b e a



Here *a* is root as it appears at the start of pre and end of post. And then go with instinct. If complete binary tree was given then things will be easy. First see if elements are power of 2 – 1. If not then you have to make one more node to satisfy left fill property. In short, we can construct unique BT iff

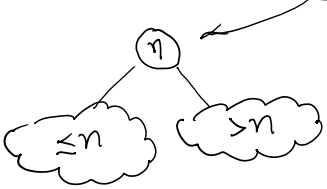


From above we can conclude that we can construct unique any BT iff we have inorder + any order.

//Lecture 13a

2.2) Binary search tree :

Binary search tree is binary tree such that all the nodes at the RHS of the root node of subtree is greater than value stored at root node and all the nodes at the LHS of the root node of subtree is less or equal to value stored at root node. For example, for any node



Now, we know that inorder traversal of any tree can be thought of as projecting values of each node from left to right on number line. And From property of BST we know that minimum number occurs

at the LHS of tree and maximum element occurs at the RHS of tree. And if we project on number line then we have sorted sequence of numbers meaning *inorder traversal of BST always gives sorted number in increasing order*.

2.2.1) Operation on BST :

1) **Search in BST** : Let's say you want to search k. We first visit root node

   Step 1 : if value is less then root value then visits left node of root and go to step 1

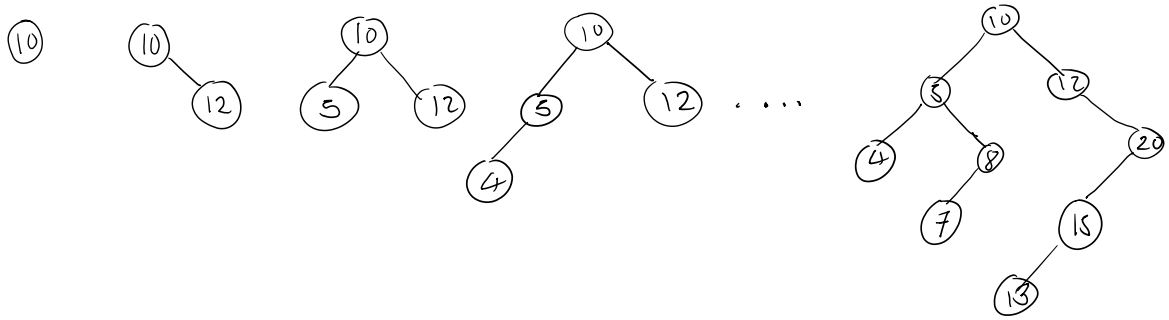
   Step 2 : if value is less then root value then visits right node of root and go to step 1

   Step 3 : If value is equal then return else return error.

   Step 4 : If you reach leaf node (and k !=) then return k is not found.

2) **Insertion in BST** : same as searching we first search for the position if we reach leaf then check if value is less than or equal to k if it is then k will be left child if not then k will be right child.

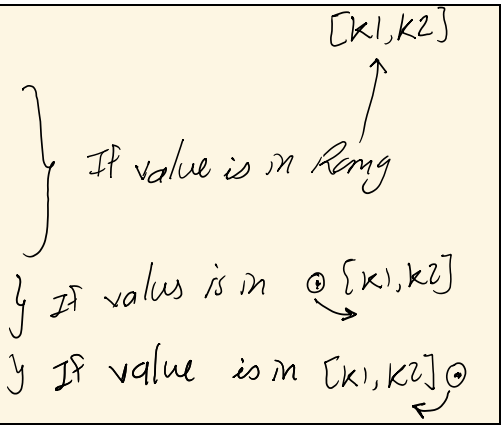
Q : Build BST from given numbers. 10, 12, 5, 4, 20, 8, 7, 15 and 13 –



Kth maximum or Minimum element in BST : first we find inorder traversal of BST then we find kth element from left to get kth minimum or you can find kth element from right to get kth maximum.

3) **Range search in BST** :

```
void RangePrinter(node *root, int k1, int k2){
    if(root == NULL) return;
    if(root->data >= K1 && root->data <= k2){
        printf("%d", root->data);
        RangePrinter(root->left, k1, k2);
        RangePrinter(root->right, k1, k2);
    }
    else if(root->data < k1)
        RangePrinter(root->right, k1, k2);
    else RangePrinter(root->left, k1, k2);
}
```



Complexity of range searches : You will defiantly encounter nodes between k1 and k2 let's say there are m nodes between range. So, m + something. This something is some extra nodes traversed. At worst we can search in longest chain in BST which has length of h (height) so total time complexity is O(h + m). where h = lg n. and if n >> m then time complexity will become O(lg n).

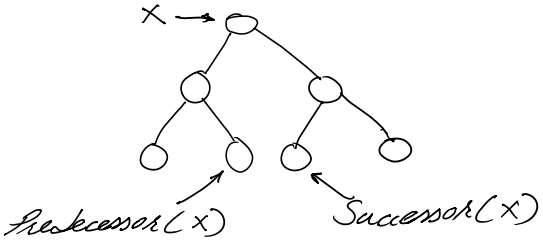
//Lecture 13c

4) **Deletion in BST** :

But before that we need to understand few concepts clearly.

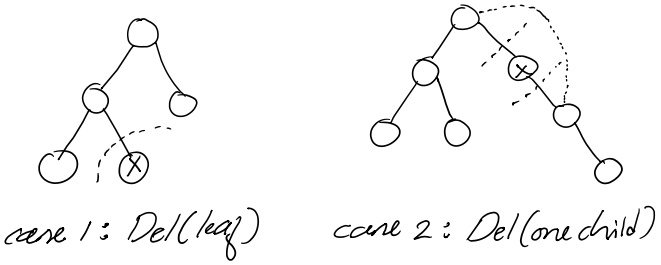
In-order predecessor : Maximum value in left-subtree

In-order successor : Minimum value in right subtree

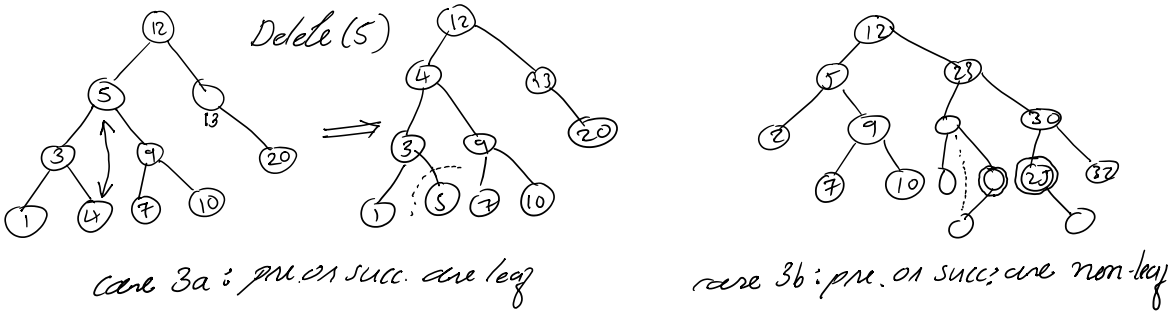


Now we can delete node from BST,

Three cases in delete :



- *Case 1* : Leaf node – just delete it
- *Case 2* : One child – Delete it, connect child to parent
- *Case 3* : Two children – you can either replace the which you want to delete with in-order successor or predecessor. In case 3 we have two subcases,



//Lecture 14a

2.2.2) Possible probe sequence :

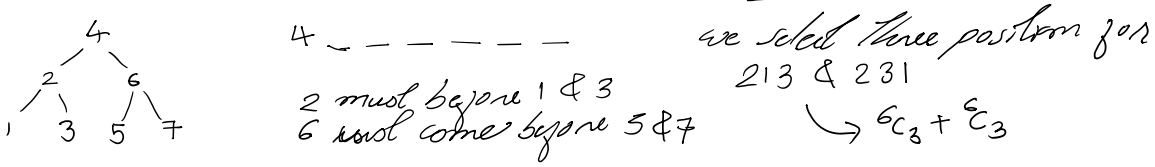
Let's say we want to find legal sequence of BST search we can encounter while searching key 10. 1, 2, 5, 20, 25 these are the nodes encounter (not in order) how many sequences of these nodes are possible ? – A legal sequence has values greater than 10 in decreasing order and values less then 10 in increasing order. And these lesser and greater values can occur randomly but in order. For example,

1, 20, 2, 5, 25 is not legal sequence because values greater than 10 are not in order if it would be 1, 25, 2, 5, 20 then it is legal sequence. So, number of such legal order is 5! / (3! X 2!). because we form two groups {1, 2, 5} and {25, 20} now these may occur randomly but it should be in order.

So, whenever you are given sequence asking which one is valid do not waste time in making tree. Just collect values less and greater than key then see if less values are in increasing order and greater values are in decreasing order. This method is only valid when there is successful search for key is given if unsuccessful search for key is given then this method gives wrong results. For example,

Suppose the BST has been unsuccessfully searched for key 273. And sequence is given 550, 149, 507, 395, 463, 402, 270. We see that all the values less than 273 are in increasing order and values more than 273 are in decreasing order but still it is false because after 395 it should have taken left child or less value because 273 is less than 395 but it has traverse right child which means it is false sequence.

Number of permutations of inserting :



In remaining position, we can always place 6 before 5 and 7 but 5 and 7 can be in any order so, total 2! For 6 5 7. Total number of permutations of inserting = 2*C(6, 3)*2.

2.2.3) BST time complexity :

Three cases are possible. i.e. best, average, worst case

Operation	Best Case	Average Case	Worst Case
Search	O(1) – in first node is key	O(lg n) – height of tree	O(n) – chain like structure
Deletion	O(1) – first node is deleted	O(lg n) – height of tree	O(n) – Chain like structure
Insertion	O(1) – insertion on empty tree	O(lg n) – height of tree	O(n) – Chain like structure

//Lecture 15

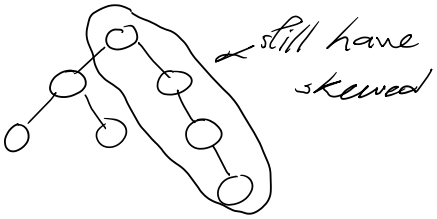
2.3) AVL tree :

There is one problem you can observed with binary search tree that in worst case it takes O(n) time complexity because of possibility of chain like structure we say “BST can be skewed”.

Solution : requires a *balance condition* that ensures height is always O(lg n)

Any suggestion for balance condition ? –

Suggestion 1 : right and left subtree of root have equal number of nodes. But can result in such structure,



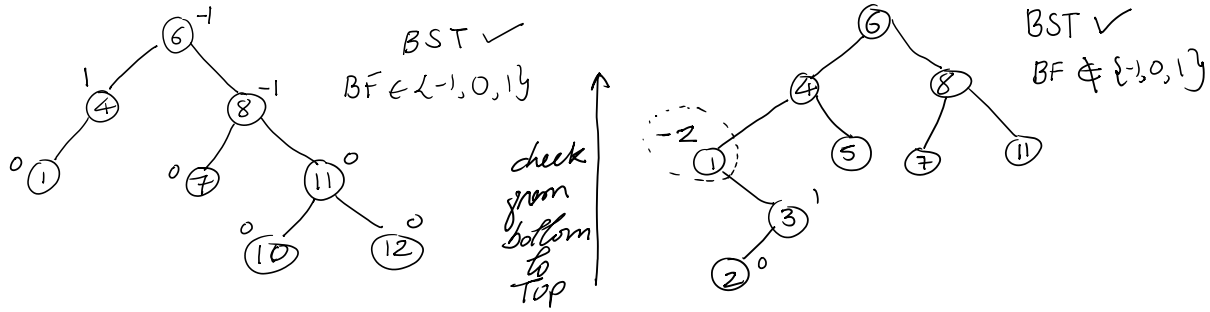
Suggestion 2 : right and left subtree of root have equal number of nodes an <u>equal height</u>. But still we can have such structure

Suggestion 3 : right and left subtree of every node have equal number of nodes. This suggestion will always make sure that height is lg n. but this condition is too strong we can only make perfect trees.

<u>*Final suggestion*</u> : balance of every node is between -1 and 1 i.e. balance(node) ∈ {-1, 0, +1}

Where **balance(node) = height(node->left) – height(node->right)**

Tree formed by applying such conditions or restriction are called **AVL tree**.

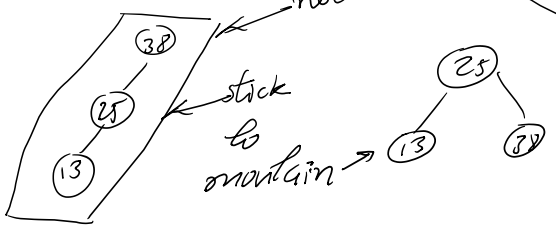


2.3.1) Operations on AVL tree :

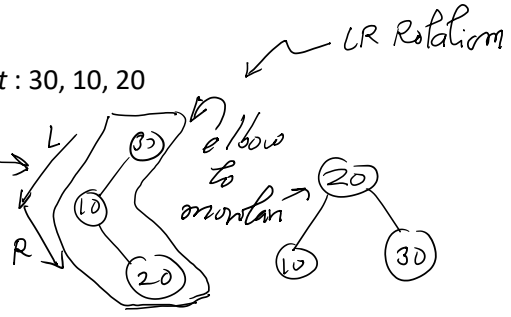
```c
//Structure of Binary tree node
struct BinaryTreeNode{
    struct BinaryTreeNode *left;
    struct BinaryTreeNode *right;
    int data;
}
//Structure of AVL tree node
struct AVLTreeNode{
    struct AVLTreeNode *left;
    struct AVLTreeNode *right;
    int data;
    int height;  //for balance factor
}
```

1) **AVL search** : Same as usual BST search because AVL is BST only
2) **AVL insert** : Same as BST insertion + balancing

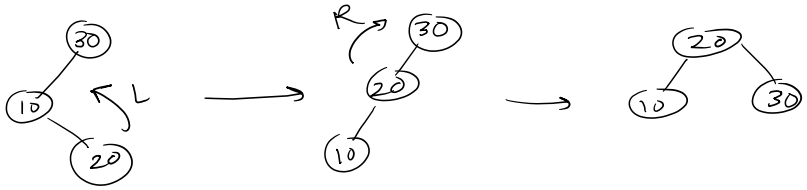
Insert : 38 25 13



Insert : 30, 10, 20

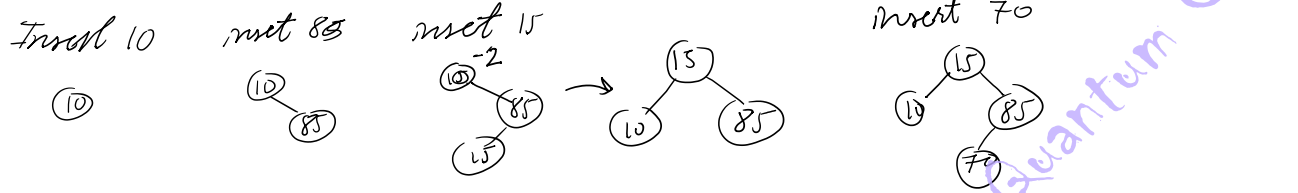


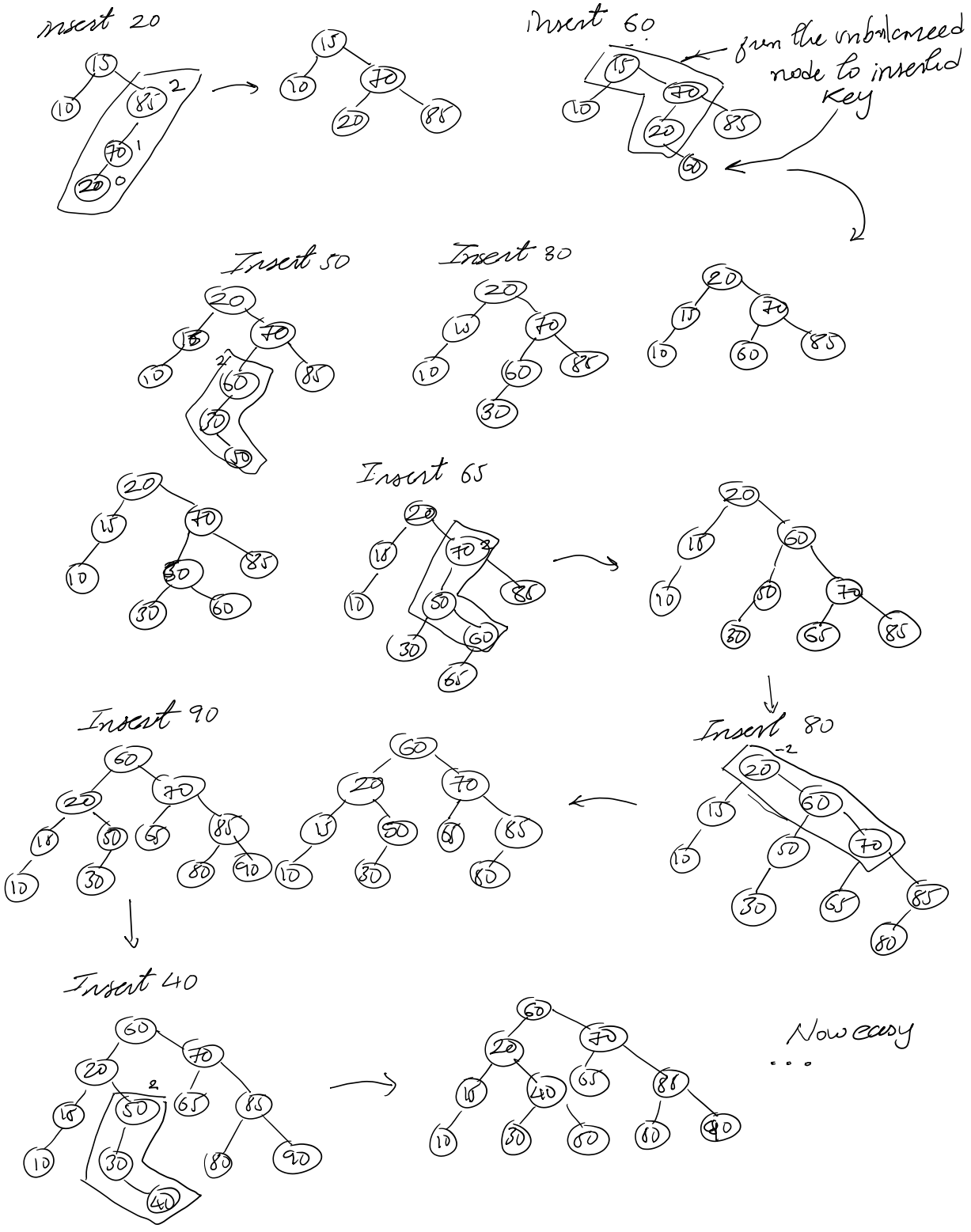
To convert elbow to mountain we first convert it into stick and then we convert it into mountain.



This means if stick => Single rotation, if elbow => Double rotation

Insert in order : 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



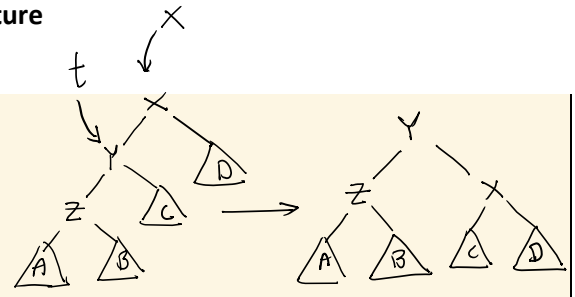


One thing you have noticed that in <u>case of insertion we only have at most 2 rotation => O(1) rotations</u>. Because one rotation takes constant time -> we just need to manipulate few pointers.

LL Rotation :

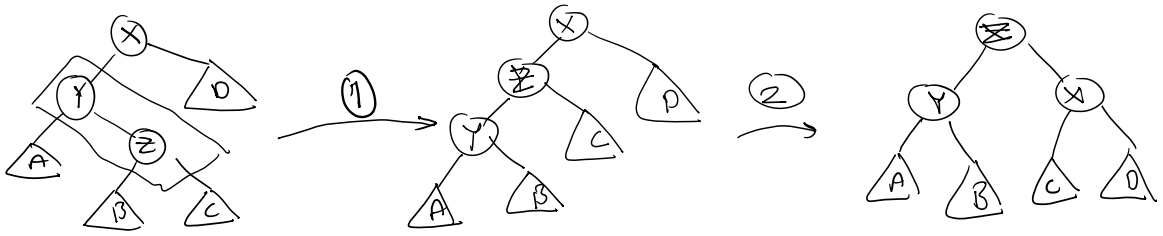
```
LL_Rotation(X){
    struct AVLTreeNode *t = X->left;
```

```
    X->left = t->right;
    t->right = X;
    //height change
    X->height = max(X->left->height,
                    X->right->height)+1
    t->height = max(t->left->height,
                    t->right->height)+1
    return t;
}
```



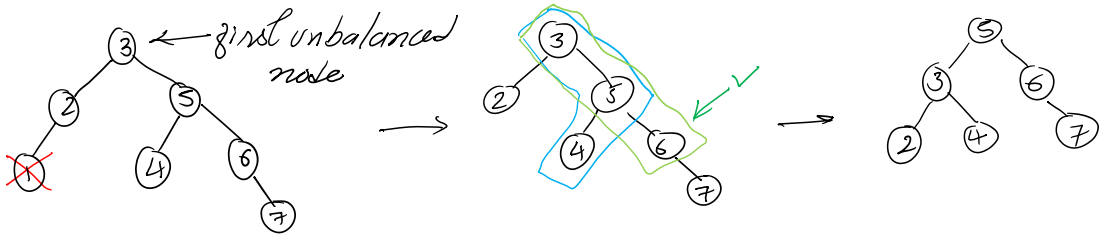
Double LR rotation :

```
Double_Rotation_LR(X){
    X->left = RR_Rotation(X->left); ..(1)
    X = LL_Rotation(X); ..(2)
    return X;
}
```



3) Deletion in AVL :

After deletion select first unbalanced node and to decide whether to take elbow or stick, take path which has highest height from unbalanced node. For example, delete(1)



But what if 4 also have one child then both heights will become same so in that case always choose stick.

Step 1 : First delete node

Step 2 : Go till root (this means do balancing for each child till root) and find first imbalance node. Identify stick and elbow.

Step 3 : Go for shape for which higher height child. If heights of children are same then select stick.

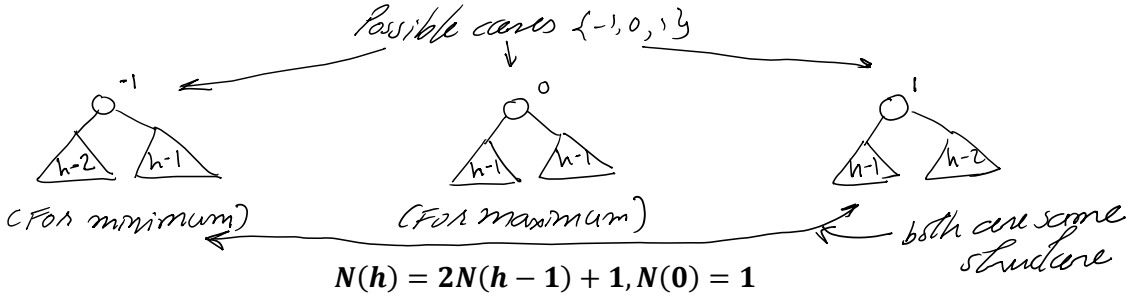
<u>As step 2 said we do this process till root so we may have to apply rotation O(lg n) times</u>.

2.3.2) Minimum and maximum number of nodes in AVL tree :

In this we are given height h and we have to find minimum and maximum number of nodes possible in AVL tree.

1) Maximum number of nodes possible in AVL tree of height h :

For maximum number of nodes, we can fill every level. We will have perfect BST. = $2^{h+1} - 1$.



$$N(h) = 2N(h-1) + 1, N(0) = 1$$

After solving we get N(0) = 1

N(1) = 3 … This is telling you that this is the maximum number of nodes height 1 can have

N(2) = 7

We saw height is given => Maximum number of nodes.

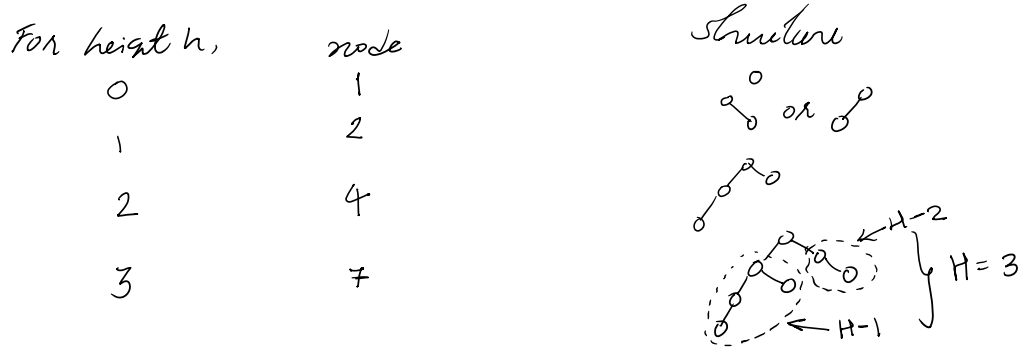
Number nodes are given => minimum possible height ?

$$height \propto \frac{1}{number\ of\ nodes} \qquad \ldots(1)$$

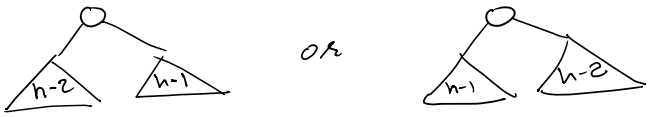
As we want minimum height it can only be obtained by filling all levels by max possible nodes from n. we know that maximum nodes = $2^{h+1} - 1$ = n then we find h by taking ceil. Meaning $h = \lceil \lg(n+1) - 1 \rceil$.

2) Minimum number of nodes possible in AVL tree of height h :

For minimum we have following structure,



Which means for minimum number of nodes in AVL of height h is



$$N(h) = N(h-1) + N(h-2) + 1, N(0) = 1, N(1) = 2$$

N(1) = 2

N(2) = 4 … this is telling you that you are allowed to add 3 (7-4) more nodes before it's height gets +1.

N(3) = 7

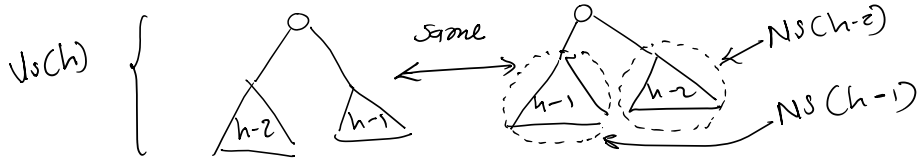
We saw height given => Minimum number of nodes

What if number of nodes are given => maximum height possible ? for example, 12 nodes => find maximum possible height of AVL tree ? – in this case, we know from (1) that if max height then min nodes so we have to find out minimum number of nodes for which maximum height is h. so we use N(h) formula and we see what value of N(h) just smaller than n. we return h at that point.

Now, we answer what if h is given => total number of distinct AVL tree ? – question asks for total number of AVL tree structure for height h. This is tough problem that is why we ask another question Let NS(h) be the number of different shapes of a minimal AVL tree of height h.

NS(h) = number of minimal AVL trees of height h.

As for minimal AVL tree of height h two structure are possible and both structures are symmetric. So, we can just find of one of the structures and then multiply it by 2.



Number of minimal AVL trees of height h is $\boldsymbol{NS(h) = 2 \times NS(h-1) \times NS(h-2), NS(0) = 1, NS(1) = 2}$

We can create variety of recurrence relation based on condition given for example, Minimum number of nodes in AVL tree of height h such that the number of nodes in left subtree should be within the factor of 2 of the number of nodes in right subtree. – Within the factor of 2 means number of elements in left subtree should be at least half of the number of elements in right subtree.

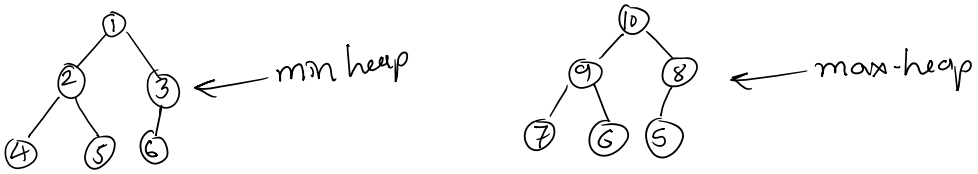
So, $N(h) = N(h-1) + \frac{1}{2}N(h-1) + 1$ with $N(0) = 0, N(1) = 1$

3. HEAP

//Lecture 16a

Heap is a tree structure with property that all levels except last are full. Last level is left-filled. This is complete binary tree. In addition to that it also has following property :

- Parents <= child (min heap)
- Parents >= child (max heap)



Heap = complete binary tree + parents ? child $(? = \leq \boldsymbol{if}\ \mathbf{min}\ \boldsymbol{and} = \geq \boldsymbol{if}\ \mathbf{max})$

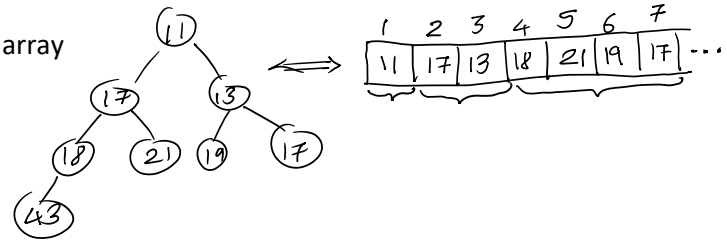
3.1) Introduction :

As heap is binary tree, we can store it in array

Parent (i) : return i/2

Left (j) : return 2j

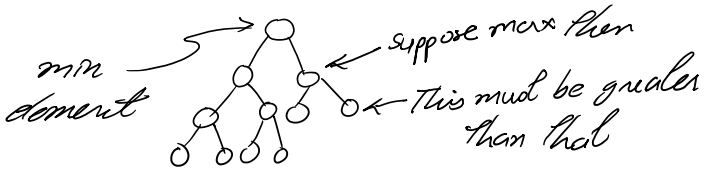
Right (k) : return 2k + 1



3.1.1) Min and max element in heap :

If min heap => min element is always root and max element is always one of the leaves.

As in min heap all elements below parents is greater which means lowest value must occur at root and talking about max element so, let's say max element is internal node then,



There can be some duplicate of min element so it can be in subtree of root but root will still be min.

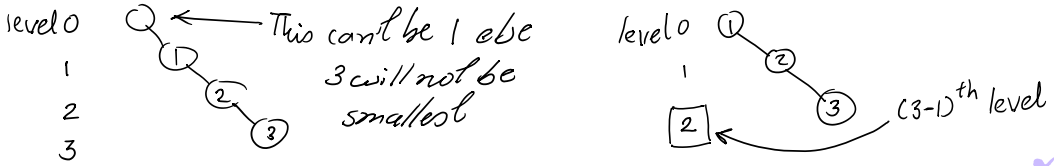
Similarly,

If max heap => max element is always root and min element is always one of the leaves.

Let's generalize above question to kth minimum in min heap,

Definition : Kth smallest element is the minimum possible n such that there are at least k elements in the array <= n

At worst kth minimum can be at (k-1)th level if root is at 0^{th} level.



So, you are not sure where is kth minimum but you are 100% sure where it can never be.

Kth minimum in min heap is always at most in (k-1)th level

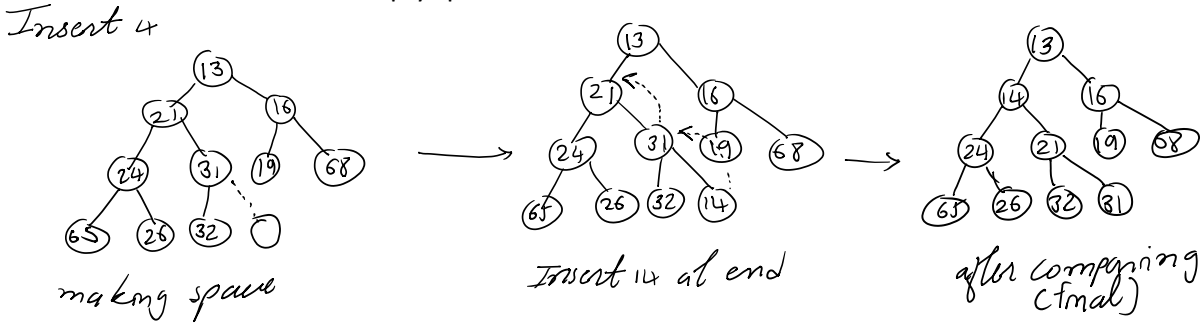
For min heap		
Where is min element	Always on root	If duplicate then also in subtree
Where is max element	Always on leaf	If duplicate then also in internal nodes
Where is kth min element	Always in 0th to (k-1)th level	If duplicate then also in kth level or above

For max heap		
Where is min element	Always on leaf	If duplicate then also in internal nodes
Where is max element	Always on root	If duplicate then also in subtree
Where is kth max element	Always in 0^{th} to (k-1)th level	If duplicate then also in kth or above level

//Lecture 16c

3.1.2) Insertion in heap :

To Insert element in heap first we create hole or allocate space at the end of heap. And then insert that element into that hole or empty space.



After inserting 14 at the end we compare it with parents 31>14 so we swap then 21>14 so again we swap then we stop because 13<14.

Meaning at worst we have to traverse whole height so; TC of insertion is O(lg n).

//Lecture 18c

Average time complexity in heap insertion :

$$E[X] = \sum_i probability\ that\ node\ belong\ to\ level\ i \times cost\ of\ leveling$$

$$= \frac{1}{2} \times 0 + \frac{1}{2^2} \times 1 + \frac{1}{2^3} \times 2 + \cdots = O(n)$$

Prob. that inserted node is in last level — 0 cost — we have to move upward by 1 cost — Prob. that inserted node is in second last level

//Lecture 16c

3.1.3) Top down build heap :

In this we discuss time complexity of building whole heap given sequence of elements.

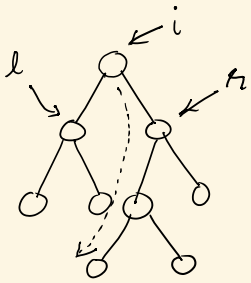
We know that at worst each insertion can take lgn time but we are over estimating it because root takes only O(1) time then left and right child of root takes O(1) time similarly but I can say for sure that leaves of heap at worst can take lgn time each. And leaves contribute half of the node so total time complexity for leaves insertion is nlgn so whole insertion is obviously > nlgn. Can we reduce time complexity ? – Yes, using bottom up approach.

//Lecture 17a

But before that we need to learn some operations :

1) **Heapify method** :

```
Max-heapify(int *A, int i){
    int l = left(i); //2i
    int r = right(i); //2i+1
    if(l<=A.heap-size and A[l]>A[i])
        largest = l;
    else largest = i;
    if(r<=A.heap-size and A[r]>A[largest])
        largest = r;
    if(largest != i) {
        swap(A[i], A[largest]);
        Max-heapify(A, largest);
    }
    return;
}
```



If largest = i then no issue because max-heap following property but if not then we may have to go downwards

Similarly, we can have heapify for min-heap. This was **heapify-down** procedure.

	Heapify-up	**Heapify-down**
Comparisons	logn	2*logn
Swap	logn	Logn

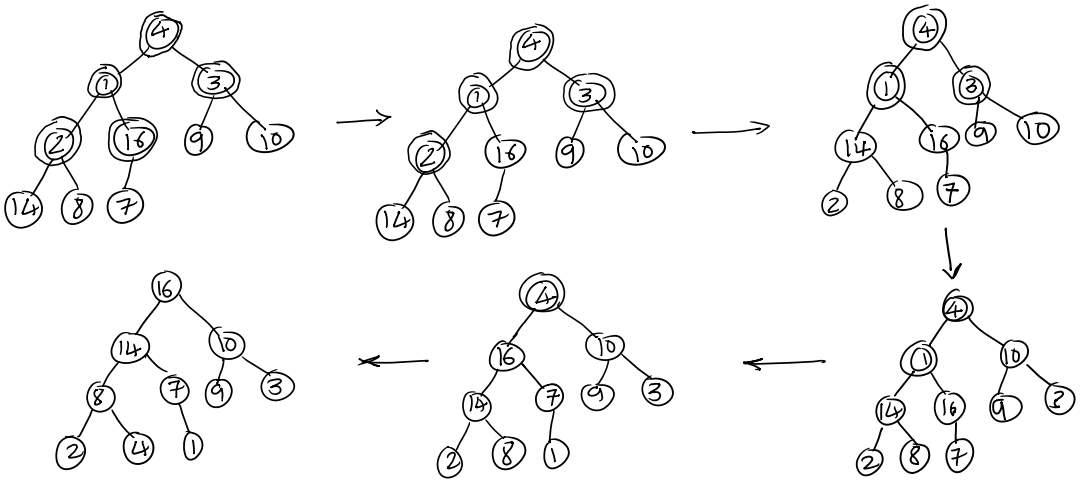
//Lecture 17b

2) **Build heap procedure** :

We first fill heap with random insertion then we apply heapify procedure in all internal nodes. Heapify method checks heap property in downward direction.

```
for(int i = n/2; i<0; i--) Max-heapify(i);
```

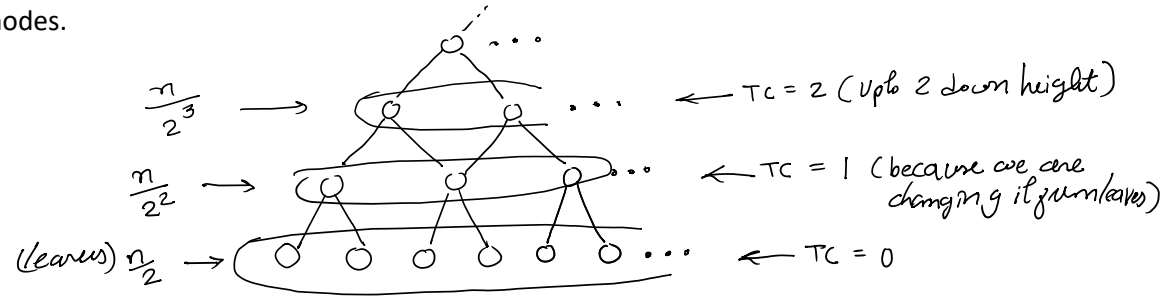
Example, Let's say A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7} we convert this into max-heap,



//Lecture 17c

Time complexity of build heap (bottom-up) :

As you may have noticed we are spending 0 time on leaves as we are starting heapify from internal nodes.



Total time complexity :

$$\frac{n}{2} \times 0 + \frac{n}{2^2} \times 1 + \frac{n}{2^3} \times 2 + \cdots + 1 \times \log n = \boldsymbol{\frac{n}{2} \times \sum_{h=0}^{\log n} (no.of\ nodes\ at\ height\ h) \times h}$$

If n is too large then summation becomes,

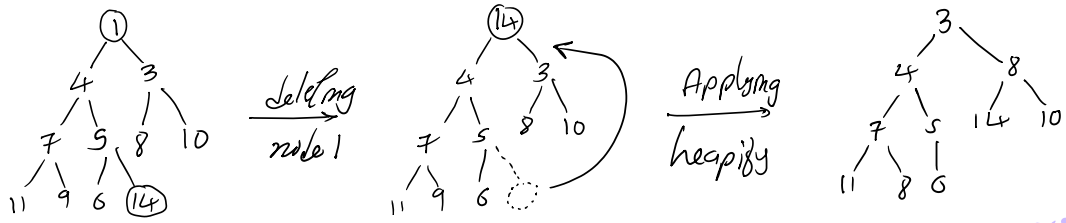
$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$ therefore, $O\left(\frac{n}{2} \times 2\right) = \boldsymbol{O(n)}$

Top down Heap build	Bottom up heap build
Using repeated insertion	Using heapify on internal nodes
O(n log n)	O(n)
Spends O(n lgn) time on leaves	Spends 0 time on the leaves

//Lecture 17d

3.1.4) Delete min in min heap :

After deleting node we'll replace it with last element and apply heapify on root node.



```
DeleteMin(int *A){
    int min = A[0];
    A[0]=A[N-1];
    N--;
    heapify(A[0]);
    reutrn min;
}
```

TC = O(lgn)

In a heap, elements can either move upwards or downwards.

- *Upwards* : If an element is inserted, it may move upwards towards the root. In such case we need one comparison
- *Downwards* : If the minimum element is deleted, it is first replaced with rightmost leaf, and then root element may move downward.

//Lecture 19a

Min-heap operation	Best case	Worst case	Average case
Insertion	O(1)	O(log n)	O(1)
Deletion	O(1)	O(log n)	O(log n)
Find-min	O(1)	O(1)	O(1)
Heapify	O(1)	O(log n)	O(log n)

Avg. TC of both deletion and heapify is O(log n) because they follow top down approach while insertion follows bottom up approach.

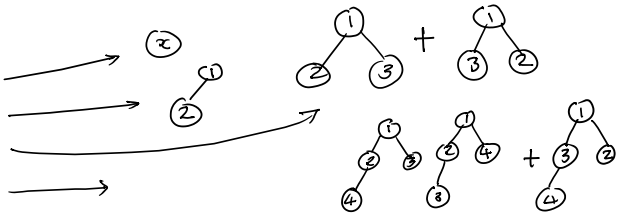
//Lecture 18a

3.2) Combinatorics, heap and binary tree :

3.2.1) Number of min heaps :

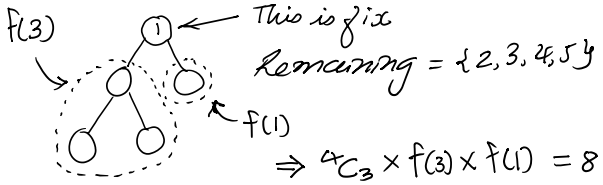
To find number of min heap or max heap, one thing to note that structure remains same as heap is complete binary tree it means we can't have skewed tree like structure or something similar.

Number of nodes f(n)	Number of heaps
1	1
2	1
3	2
4	3
5	?



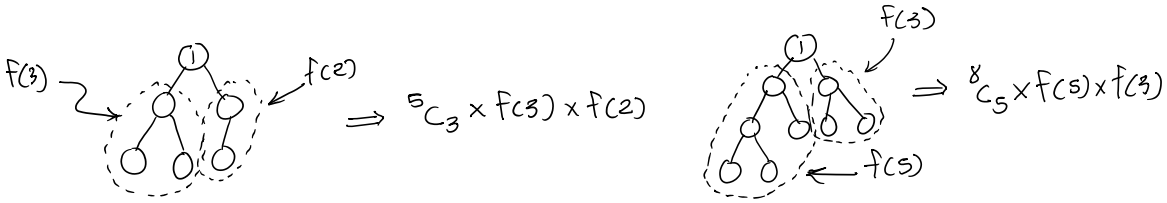
Here we take min heap, as max heap is also similar to min heap just one difference. But how to find for bigger number like 5 or more than 5 ?

Trick is to divide the tree into subtree whose number of heaps has already been calculated. For example, take 5 nodes 1, 2, 3, 4, 5. Then number of heaps would be



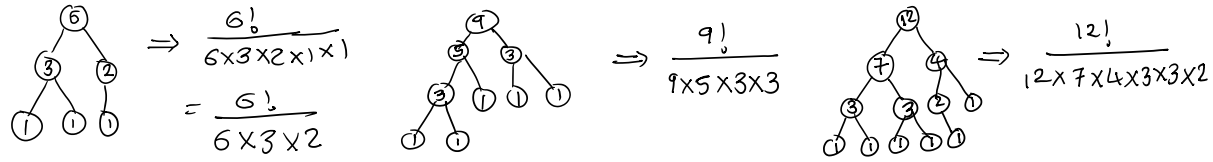
we choose 3 from there for left subtree and remaining goes to right automatically

Similarly, for 6 nodes and 9 nodes we have

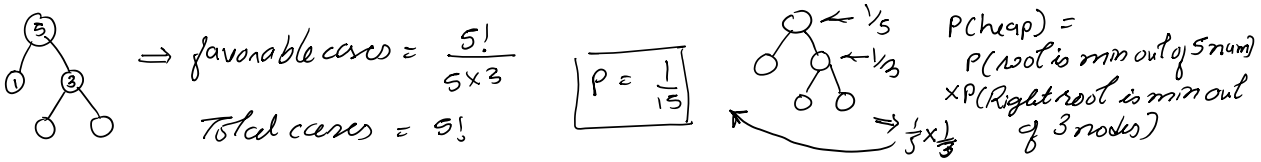


Shortcut to find no. of min heaps : shortcut method is actually logical it is simplified version of previous results.

In shortcut method we first draw structure then we count number of nodes for each sub-root then we do follow :



Q : What is the probability that given tree structure is heap ? – This is one of the applications of shortcut method, question asks us $\frac{favorable\ cases}{Total\ cases}$. Suppose tree structure is given like



There is also second method to find probability (given on right hand side)

//lecture 18b

3.2.2) Number of binary trees :

Number of shapes of binary tree possible (given n nodes) is given by Catalan number $\frac{C(2n,n)}{n+1}$

Number of labelled binary tree possible (given n nodes) is given by $\frac{C(2n,n)}{n+1} \times n!$

//Lecture 19b

3.3) Heap sort and Heap vs BST :

3.3.1) heap sort :

BuildHeap takes O(n) time.

Heapify takes O(log n) time.

Heapsort makes one call to buildHeap and n calls to Heapify because of delete min operation.

This, the complexity of heapsort is O(n + nlogn) = O(nlogn)

//Lecture 19c

3.3.2) Heap vs BST :

Heap is good at findMin/findMax O(1) while balanced BST is good at all finds O(log n).

1) **Advantage of binary heap over a balanced BST** : Average time insertion into a binary heap is O(1), for balanced BST is O(log n)

2) **Advantage of balanced BST over binary heap** : Search for arbitrary elements in balanced BST is O(log n). For heap, it is O(n) in general, except for the largest element which is O(1).

Merge two heaps : just put the two arrays together and create a new heap out of them which takes O(n).

3.4) Priority queues :

A *priority queue* is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.

A priority queue is a data structure that supports two basic operations: inserting a new item and removing element with the largest (or smallest) key.

Priority queues operations : insert (i.e., enqueue) and delete min (i.e., dequeue), decrease key

Which data structure is best two implement priority queues operations :

	Insert	**deleteMin**	**decreaseKey**
Unsorted linked list	O(1)	O(n)	O(n)
Sorted linked list	O(n)	O(1)	O(n)
Unsorted array	O(1)	O(n)	O(n)
Sorted array	O(n)	O(1)	O(n)
Balanced BST	O(log n)	O(log n)	O(log n)
Heap	O(log n)	O(log n)	O(log n)

4. Hashing, RC major in Array

//Lecture 20a

Superfast method for searching, inserting, deleting.

When to prefer hashing :

- When order of data does not matter
- The relationship between data does not matter

Application :

- Google search (page rank algorithm)
- File system – c/user/desktop… to sector on disk
- Compiler – symbol table
- Digital signatures
- Machine learning (locality sensitive hashing)

Our goal is to do insertion, deletion, search in O(1) on avg time complexity.

Direct addressing table : a fancy name for "array"… it is same as array where elements are stored at index same as their array index. It's limitations :

1) Key must be non-negative integer. In case string or other datatype is given we first map it to int and then store
2) Range of keys must be small
3) Keys must be dense i.e. not many gapes in the key values.

How to avoid these limitations ?

- Map non-negative keys to integers
- *Map large integers to smaller integers*

//Lecture 20b

4.1) Hash table :

With *direct addressing*, an element with key k is stored in slot k. With *hashing*, this element is stored in slot h(k); that is, we use a **hash function** h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a **hash table** T[0…m-1] :

$$h : U \rightarrow \{0, 1, 2, 3, \ldots, m-1\}$$

Division method (mod operator) : map into a hash table of m slots. $hash(k) = k\%m$

Q : how to pick m (table size) ? – if m is power of two, say 2^n, then (key mod m) is the same as extracting the last n bits of the key. This is not good idea as it only considers last n bits two number can have some common last bits in that case collision happens. What if m is 10^n, then the hash value is the last n digit of the key. This is also not good idea because 4 and 34 maps to same location.

Rule of thumb : pick a *prime number*, close to a power of two, to be m.

We want h(k) depends on *every bit* of k, so that the differences between different k's are fully considered.

Load factor : $\alpha = \frac{n}{m}$ ← No. of elements ← size of table

4.1.1) Simple uniform hashing : is when any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to

$$\textbf{\textit{Uniform}}: \underset{h \in U}{\text{Prob.}}[h(x) = i] = \frac{1}{m} \; for\ all\ x\ and\ all\ i$$

4.1.2) Collisions and its resolution techniques : Since |U|> m, there must be 2 keys that have the same hash value. We need a mechanism for handling collisions.

Collision resolution techniques :

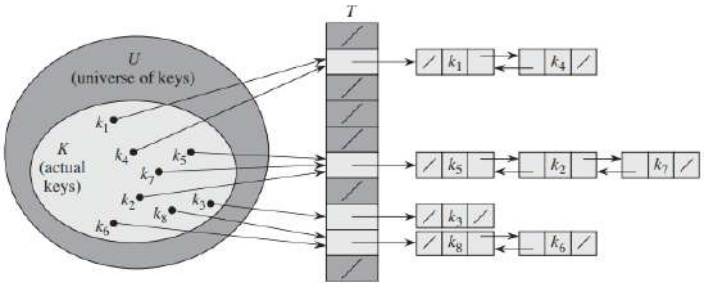
- *Separate chaining* – also called close addressing or open hashing. Close addressing because Im not allowed to search for an element besides that index. It's called open hashing because Im allowed to use external data structure like linked list etc.
- *Linear probing*
- *Quadratic probing*
- *Double hashing*

} open addressing or closed hashing

Index

//Lecture 20c

1) **Separate chaining** : all keys that map to the same hash value are kept in a list.



Worst case performance :

- Search (k) : worst case = O(length of chain), Worst length of chain : O(n) all maps to same.
- Insert (k) : Need to check whether key already exists, still takes O(length of chain)
- Delete (k) : need searching so O(length of chain)

However, in practice, hash tables work really well, that is because the worst case almost never happens. And average case performance is really good.

//Lecture 20d

Theorem : *In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time O(1+α), under the assumption of simple uniform hashing*.

Proof. We know that here α represents load factor which in chaining represents average length of chain. So, consider we search for key which is not present in hash table we will first mapped to some entry in hash table (which takes O(1)) then in average case we go through whole chain and we don't get that element so O(length of chain) = O(α). Total of O(1+α).

Q : What if we ask above theorem but in successful search ? – then also answer remain same.

Time to successful search for ith item in table having n items. = time to insert ith item when there were i-1 items in hash table = unsuccessful search with i-1 item in the table.

//Lecture 21a

From now on we will talk about open address hash tables or closed hashing where all elements are stored in the has table i.e. n<=m and there is no chain. To avoid collision, we use **probing**.

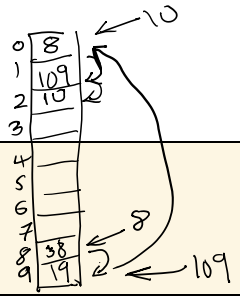
Q : How to probe ? – we want to design a function h, with the property that for all k ∈ U :

$$h: U \times \{0, 1, 2, \dots, m-1\} \rightarrow \{0, 1, 2, \dots, m-1\}$$

Universe of keys Trial count slot in table

2) Linear probing :

Idea : use empty space in the table.




```
if h(key) is already full,
    try (h(key) + 1)% TableSize. if full,
    try (h(key) + 2)% TableSize. if full,
    try (h(key) + 3)% TableSize. if full,...
```

Example : insert 38, 19, 8, 109, 10

Here we have used ith probe was (h(key) + i) % TableSize

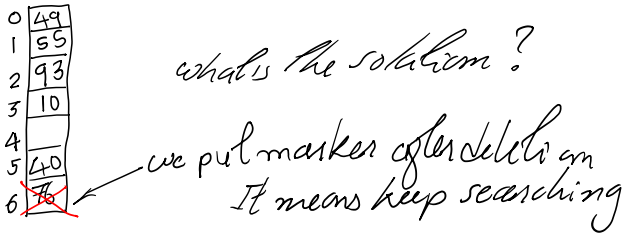
In general, we have some f and use $\big(h(key) + f(i)\big)\ \%\ TableSize$. In this f(i) can be any function of i

We have named few probing strategies depending upon f(i)

- Linear probing when f(i) = i
- Quadratic probing when f(i) = $c_1 i + c_2 i^2$ or i^2
- Double hashing when f(i) is another hash function i x h_1(key)

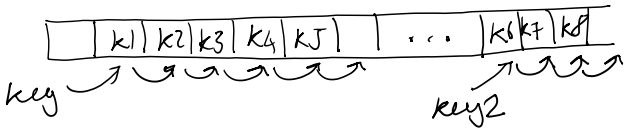
Search in linear probing : Continue looking at successive locations till we find key or encounter an empty location.

Deletion in linear probing : We can't just delete element because it may create empty space. Because of this empty space upcoming search may affect and gives us wrong results. For example, here we delete 76 and then search for 55 then we will get "Not found" although it is present.



Problem with linear probing :

Primary cluster



//Lecture 21b

3) Quadratic hashing :

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) mod\ m$$

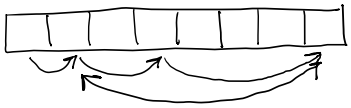
where c2 != 0 if it were then h(k, i) degrades to linear probing. But this is general formula we basically take c1 = 0 in most cases.

Note that we apply all of these things whenever collision happens.

Problem with quadratic hashing : if two keys have the same initial probe position, then their probe sequences are the same.

Since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This is called **secondary clustering**.

Cycles in quadratic probing :



But this cycle can be eliminated with careful selectin of c1, c2 and h(k).

4) **Double hashing** :

$$h(key) = \big(h(key) + ih_1(key)\big) mod\ m$$

here h and h1 are different hashing functions. But how it solves secondary clustering problem ? – since secondary clustering satisfy $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$ but double hashing do not have these problem if $h(k_1, 0) = h(k_2, 0)$ is true then new hash function for k1 would be $h(k_1, 1) + h_1(k_1, 1)$.

NOTE : The main advantage of Chaining over open addressing is that in Open Addressing sometimes though element is present we can't delete it if Empty Bucket comes in between while searching for that element; Such Limitation is not there in Chaining.

//Lecture 21c

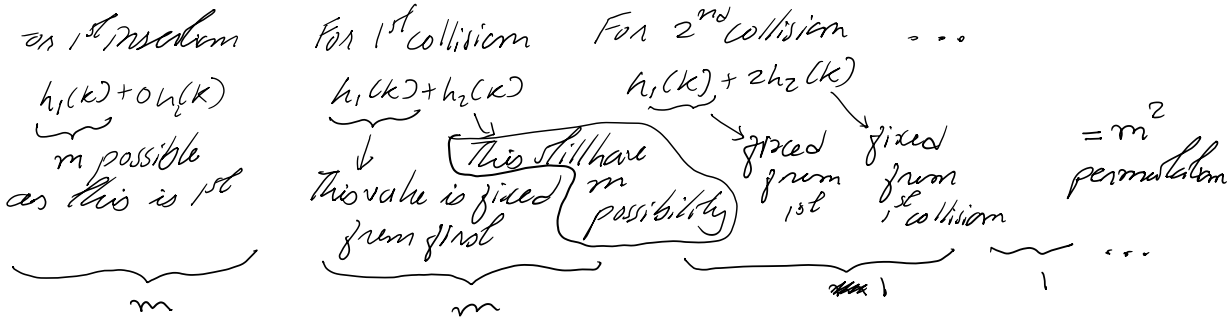
4.1.3) Possible number of probes :

Possible no. of probes in linear probing = m because only for first position we have to decide position.

For quadratic probing = m because only for first position we have to decide position.

In uniform hashing we can have m! permutation of probes as each position is equally likely to choose even after some insertion.

In double hashing, we have m^2 permutation how ?



//Lecture 22a

4.2) Analysis of open addressing :

In this section we are going to use uniform hashing so there is no point of collisions. We are only concerned about element in cell. So, there are m! permutation is possible.

Load factor α in open addressing : In open addressing, the hash table can "fill up" so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

4.2.1) Search time open addressing :

1) **Unsuccessful search time** : Given an open-address hash table with load factor α the expected number of probes in an unsuccessful search is **at most** $\frac{1}{1-\alpha}$, assuming uniform hashing.

Proof. We have an open-address hash table with m slots, load factor α, and uniform hashing, where 0 < α < 1. This means that there is n = α * m elements stored in the hash table.

In an unsuccessful search, we're looking for an element that is not in the hash table. We start by hashing the key and checking the slot. If it's empty, we're done. If it's occupied, we need to probe further. Probability that slot is occupied is α and probability that slot is empty is $1 - \alpha$. Now, let X represents number of probes required to find empty slot. Then,

$$E[X] = (1-\alpha) + 2\alpha(1-\alpha) + 3\alpha^2(1-\alpha) + \cdots$$

$$E[X] = (1-\alpha)(1 + 2\alpha + 3\alpha^2 + \cdots) = 1 + \frac{\alpha}{1-\alpha} = \boldsymbol{\frac{1}{1-\alpha}}$$

Unsuccessful search time is same as number of probes required to insert an element into an open hash table having n element because in unsuccessful case also we stop as soon as we encounter empty slot and in insertion also same.

2) **Successful search time** : Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$, assuming uniform hashing that each key in the table is equally likely to be searched for.

A successful search for a key k reproduces the same probe sequence as when the element with key k was inserted. If K was the (i+1)th key inserted into the hash table, then we know that there are i key already inserted so expected number of probes made in a search for k is at most 1/(1-i/m) = m/(m-i). Taking average of all n keys in the hash table :

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{dx}{x} = \boldsymbol{\frac{1}{\alpha}\ln\frac{1}{1-\alpha}}$$

No. of comparisons made during unsuccessful search : First key got mapped to a particular location then we do our first comparison, then we follow search according to type of probing (for example, in linear probing we do linear search, in quadratic probing we do quadratic search) till we hit empty space. We again have to check whether that space is empty. So, in total we do cluster comparison + 1. 1 is due to empty space check.

No. of comparisons made during successful search : At worst we have to cover whole cluster and last element of cluster is our key because successful search is given. So, no. of comparisons = cluster comparisons only.

//Lecture 23b

4.2.2) Expectation in hashing :

Here Bernoulli's random variable is begin used.

1) **Expected items per slot** : we know that it is α but still prove it.

Let X : Number of items in a particular slot then Xi : ith item maps to particular slot

X = X1 + X2 + X3 + … + Xn (why upto Xn because we have n items at the end), E[X] = ?

E[Xi] = 1/m as there are m slot and ith item can map to any of the m slots in uniform hashing. Therefore, E[X] = n/m

2) **Expected number of empty locations** :

Let X : number of empty locations then Xi : ith slot is empty

X = X1 + X2 + X3 + … Xm (why upto m because there are m slots), E[X] = ?

E[X1] is same as all the n keys mapped to other location apart from location 1. Means $\left(1-\frac{1}{m}\right)^n$.

Therefore, E[X] = $m \times \left(1-\frac{1}{m}\right)^n$

3) **Expected number of collisions** :

X : number of collisions, Xi = collisions at ith insertion.

E[X1] = 0 because at first insertion whole hash table is empty so there will be no collisions.

E[X2] = 1/m. because after first insertion element must be in any of the slot and we again have to map 2nd element to that slot only to result in collision.

E[X3] = 2/m. because after two insertion 2 slot must be filled and 3rd element must be mapped to one of those 2 filled slots. Similarly, for E[4] …

E[X] = E[X1] + E[X2] + … + E[Xn] = 0 + 1/m + 2/m + … + (n-1)/m = $\frac{n(n-1)}{2m}$

NOTE : when open addressing is given without specifying probing strategy consider random probing.

//Lecture 24a

4.3) Row major and column major in arrays :

When we have array then we know that in memory we cannot directly store it in array formate because memory is linear and that is why we got two choices i.e. either to put it in row wise or column wise.

Now, in memory we do not have memory 1, 2, 3, if we want to get element, we want base address of that array and then we can use index to get desired element.

In Row major : Suppose a is m x n array. Address of a[i][j] = base + (ni + j) ← This is called 99 not

In column major : Suppose a is m x n array. Address of a[i][j] = base + mj + i

//Lecture 24b

4.3.1) RM and CM in higher dimensional array :

Consider 3x3x3 array

1 2 3 10 11 12 19 20 21

4 5 6 13 14 15 22 23 24

7 8 9 16 17 18 25 26 27

Memory sequence in RM : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 …

Memory sequence in CM : 1 4 7 2 5 8 3 6 9 10 13 16 11 14 …

Some terminology which we are going to use

Ni = size of ith dimension

ni = index of an element in dimension i (this is index of particular element)

For row major :

2D = $n_1N_2 + n_2$

3D = $n_1N_2N_3 + n_2N_3 + n_3$

For Column major :

2D = $n_1 + n_2N_1$

3D = $n_1 + n_2N_1 + n_3N_1N_2$

Don't visualize it just remember the pattern…

First write n1 + n2 + n3 = offset. Now if row major is asked then start from left to right and if column major is given then start from right to left.

Some variation may possible. Our all formula is valid if indexing starts from 0 but What if indexing starts from random value let's say we have array M : [5….15][4….20] then we will subtract i by 5 and j by 4. Offset of M[i][j] in row major order will be

$N_1 = 15 - 5 + 1 = 11$

$N_2 = 20 - 4 + 1 = 17$

$offset = (i-5) + (j-4)$

$= (i-5)N_2 + (j-4) = 17i - 5.17 + j - 4$