Day-20

Module -vi :- Intro to graphs graph types, ~~term~~ terminologies, Representation of ~~graph~~ graphs, Graph traversal :- BFS, DFS, Spanning tree, Minimum Spanning tree, shortest path algorithm, Def hashing, Functions, Load factor and collison, (open addressing (linear probing). Chaining method.

Module - v :- Binary trees :- Intro, types, terminology, code, Representation.

Bst :- Intro, representation, Code Sequential representation B-in tree B tree, B+ tree

Module - iii :- Comparison linear binary, Big O notation -

Selection, Bubble :- Define, how it works.

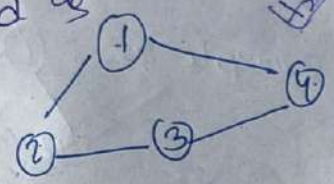
Module 4 :-

Intro to graph :- Graph is a non linear data structure which is made of vertices and edges. It's used to represent relationship between two objects and it has many real life implementation like :- Website backlinks, telephone network, social media network.

Graph is represented as

$G = (V, E)$

where G is the graph which consists of |V| vertices and |E| edges



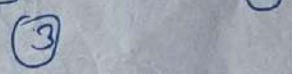
$|V| = \{1, 2, 3, 4\}$

$e = \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\}$

Graph types :- There are several types of graph :-

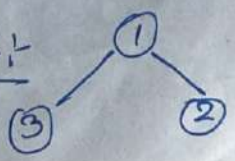
- Vertex / vertices :- It's a entity which holds multiple data / information in it.

example :-



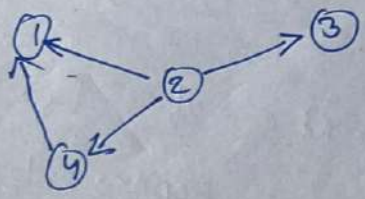
- Edge :- Edges is a connecting point between two vertices and nodes

example :-



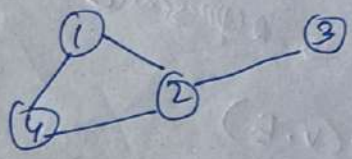
- Direct edges :- An edge which has direction known as direct edge.

ex:-



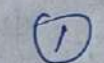
- Undirect edge :- An edge ~~that~~ which doesn't have any direction or it's bidirected.

example :-



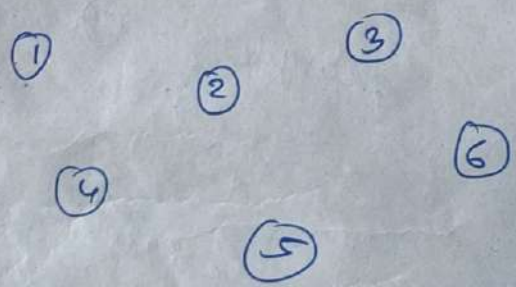
- Initial graph :- A graph containing no edges and one vertex

example :-



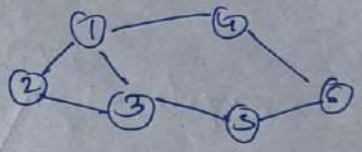
- NULL Graph :- It's like a trivial graph if several vertices and no edges.

example :-



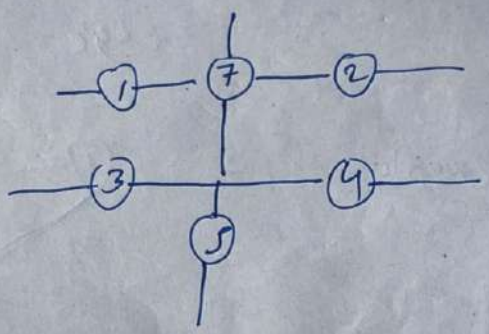
• Finite Graph :- A graph's nodes and edges are limited in a number.

example :-



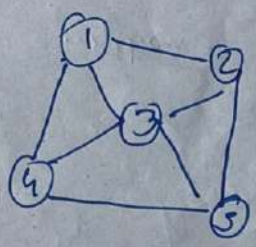
• Infinite graph :- A graph's vertices or edge infinite.

example :-



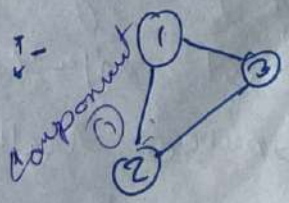
• Connected graph :- If ~~an edge~~ there's always an edge between to vertices for travel it's known as connected graph.

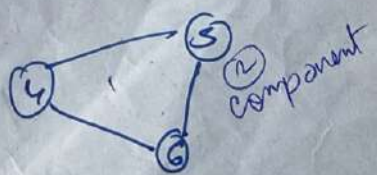
example :-



• Disconnected graph :- When there's no edge b/w two vertices in a gr

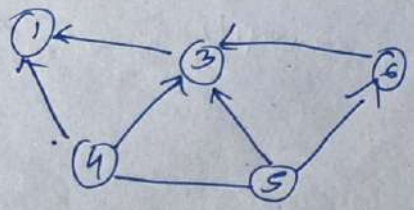
example :-





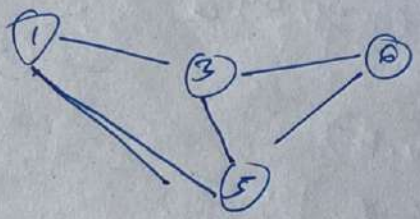
- **Direct graph** :- A graph's all the edges are direct.

example:-



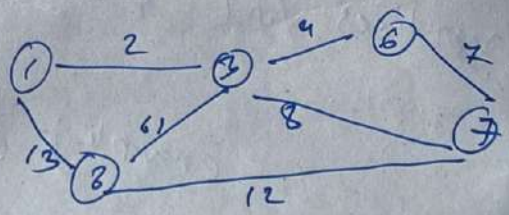
- **Undirect graph**:- A graph's all the edge are undirect

example:-



- **Weighted graph**:- If there's a cost value assigned to edge's of a graph.

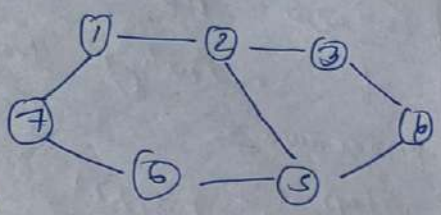
example.



Representation of a graph:- Graph can be represented in several ways.

1) Adjacency List
2) Adjacency Matrix
3) Cost adjacency list
4) Sparse Matrix
5) cost adjacency Matrix

1) Adjacency list



1	→ {2, 7}
2	→ {1, 3, 5}
3	→ {2, 4}
4	→ {3, 5}
5	→ ~~{2, 3}~~ {2, 4, 6}
6	→ ~~{5, 7}~~ {5, 7}
7	→ {1, 6}

Adjacency matrix

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	1
2	1	0	1	0	1	0	0
3	0	1	0	1	0	0	0
4	0	0	1	0	01	0	0
5	0	1	0	1	0	1	0
6	0	0	0	0	1	0	1
7	1	0	0	0	0	1	0

[if a[i][j] then's an edge b/w then 1 otherwise 0]

• Graph traversal :- It's a method to traverse one vertex to another vertex through an connected edge between them for a particular ~~purpose~~ reason.

There are two types of graph traversal

BFS, DFS :-

1) BFS :- It's a well known traversal technique for to ~~do~~ traversing a graph

Queue ADT is implemented by the time of executing this traversal technique.

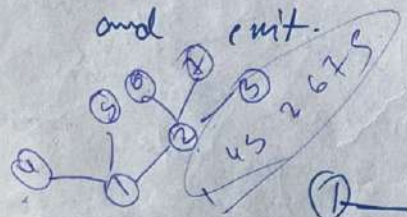
procedure :-

Step 1 :- Start with any vertex.

step 2 :- Make a visited list and enqueue the source vertex and mark it visited afterwards enqueue all the adjacent vertices of that source vertex and ~~dequeue~~ mark them visited.

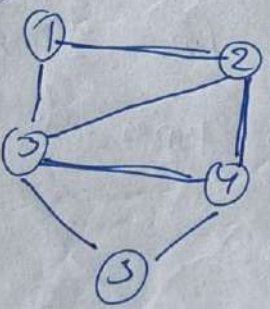
Step 3 :- Make this process Repeat unless all the vertices are visited.

Step 4 :- Return the BFS to main function and exit.

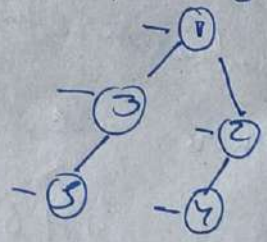


enqueue :- 1 2 3 4 5

visited :- 1 2 3 4 5



[BFS spanning tree Lot]



Lot :- 1, 3, 2, 5, 4

- BFS traversal is not unique!

DFS:- It's a graph traversal technique. Stack ADT in initialised while executing this traversal.

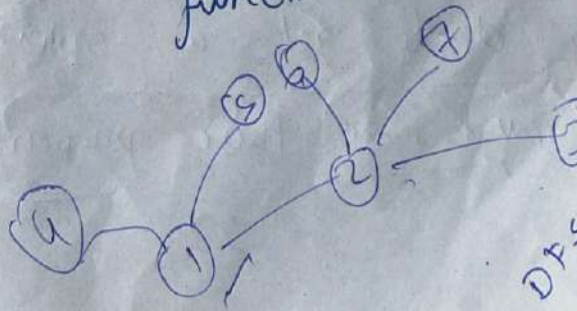
Procedure :-

Step 1 :- Start with any vertex.

Step 2 :- Initialise a stack and travel a source vertex and push it onto the stack and pop it and add the source vertex into the visited list and push any one of adjacent vertex of that source vertex and try to go deep on that vertex and if all the adjacent vertices of the source vertex is traversed then back track and travel the remaining one's.

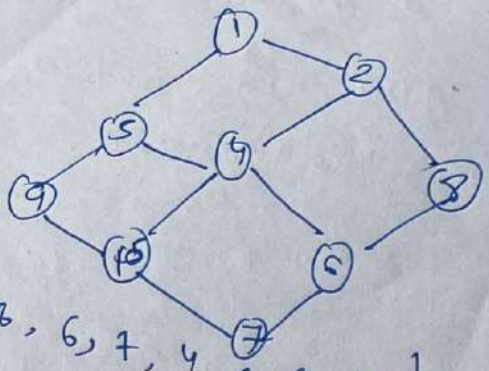
Step 3 :- Repeat Step 2 unless and until all the vertices are visited and receive the dfs result.

Step 4 :- Return the dfs into the main function and exit.



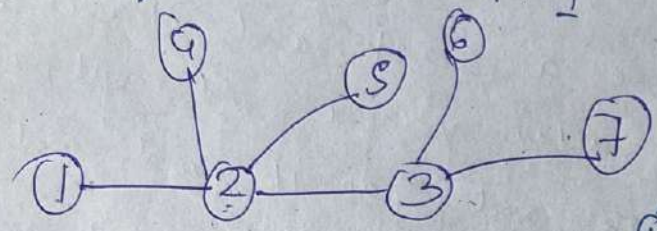
DFS:- 1 2 3 4 5 6 7

BFS:- 1 2 3 4 5 6 7



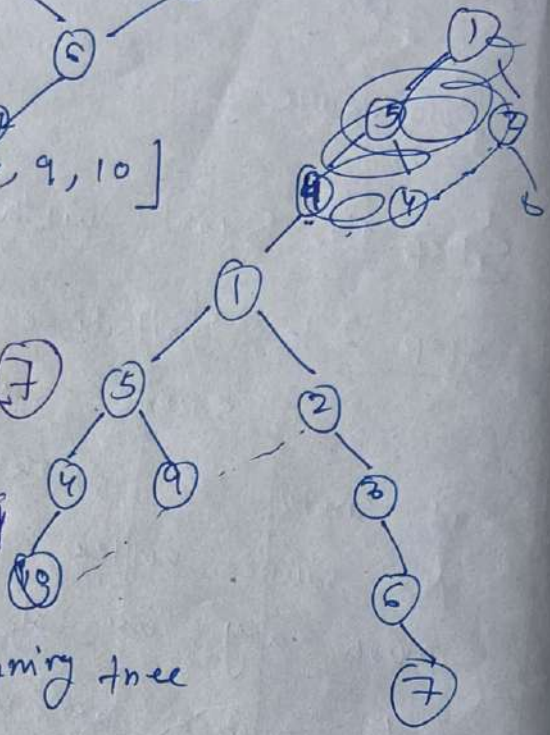
Stack [1, 2, 8, 6, 7, 4, 5, 9, 10]

Result [1 2 8 6 7 4 5 9 10]



DFS 1 2 ~~4~~ 3 3 7 6 5 ~~4~~ 4

Result:- 1 2 3 7 6 5 4



DFS Spanning tree

~~Inorder~~

Preorder :-

Result:- [10, 4, 5, 9, 1, 2, 8, 6, 7]

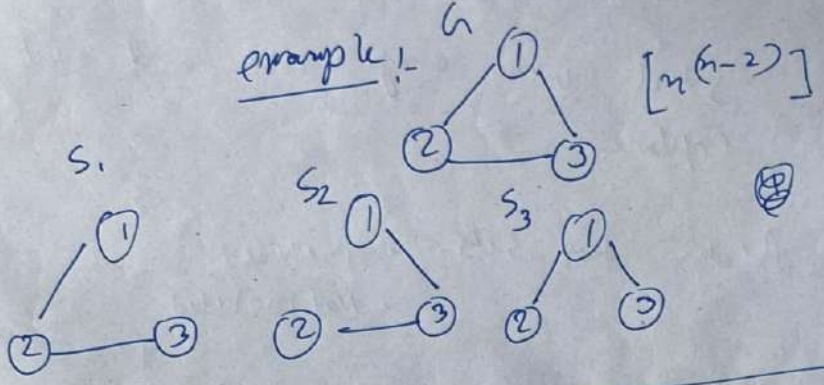
- Spanning tree :- A subgraph of a graph "G" is known as spanning tree. for a "n" no of "G" graph $n^{(n-2)}$ Spanning trees are possible. $S \subseteq G$ where $|V|$ vertices and $|V-1|$ edge must be present as the "S" subgraph of "G". In order to get a spanning tree a graph must have these properties.

- Connected :- There's always an edge b/w two vertices

- Complete :- An undirect graph.

- Acylic Graph :- There's no cycle in a graph -

example :-



- Minimum Cost Spanning trees :- A minimum cost spanning tree only can be constructed for weighted, undirect, connected graph: where the some of cost values are given on a graph's edge the cost value denote storages :- time, or cost, space anything. Since there are $n(n-2)$ Spanning trees a graph can have searching for minimum cost spanning trees are time taken. So we use two greedy methods to find mst of a graph.

- Prim's
- Kruskal

Prims Algo :- Prim's is well known algorithm to finding mst of a graph in an efficient way. the t.c of this algorithms is $O(v^2)$

procedure:-

Step 1:- Start with any vertex and start explore

Step 2:- Make two sets
- included
- Not included

add the source vertex into included mst and all the remaining ones in not included mst.

Step 3:- Start explore and comparing from the source vertex to the adjacent vertices, if any adjacent edges' edge's cost is minimum then simply add it to the included mst set. Repeat this unless and until all the vertices are explored and not included set contains $\emptyset$ value.

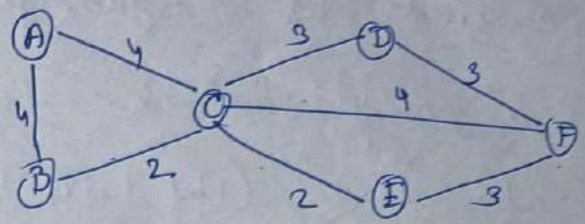
Step 4 :- Sum all the cost & end exit.

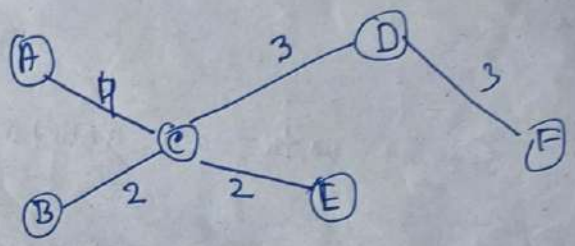
Try not to create a cycle

E → C = 2
E → F 3

C → A = 4
C → B = 2
C → D = 3
C → F = 4
E → F = 3

B → A = 4
C → A = 4
C → D = 3
C → F = 4
E → F = 3





MST :- 4+2+2+3+3
= 14 (Any)

V = { A, B, C, D, F }
V = { A, B, D, F }
V = { A, D, F }
V = { A, F }
V = { A }
V = { ∅ }

A = { E }
A = { C, E }
A = { B, C, E }
A = { B, C, D, E }
A = { B, C, D, E, F }
A = { A, B, C, D, E, F }

Dijkstra Algorithm :- This algorithm is used for finding Single Source Shortest path of a graph. It was conceived by Edsger. W. Dijkstra in 1959. It's a greedy method. Time Complexity of this algo is $O(|E| + |V| \log |V|)$

Procedure :-

Step 1:- Start with an arbitary vertex

Step 2:- Make a table where in left hand side source vertex and Destination vertex in right hand side.

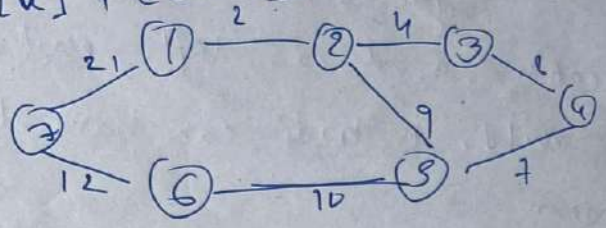
Step 3:- Set the value of current vertex to "0" and remaining one's to "∞".

Step 4:- Every time compare if a adjacent ~~vert~~ edge is smallest ~~and~~ pick it and try to reach destination as minimum cost as can be, ~~Determination~~ :-

Step 5:- Make these process repeat unless and untille all the ~~vertex~~ the reaching point is found.

Step 6:- Return the shortest path into main function and exit.

Relaxation if $(d[u] + c(u,v) < d[v]$ then $d[v] = d[u] + c(u,v))$



$5 \to 1 = 9+2 = 11$

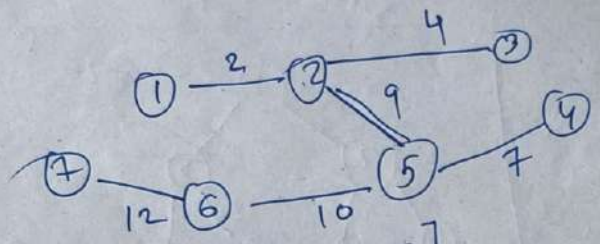
$5 \to 3 = 9+4 = 13$

$5 \to 4 = 7$

$5 \to 6 = 10$

$5 \to 6 \to 7 = 10+12 = 22$

$5 \to 2 \to 1 = 9+2+21 = 23$



~~[5 → 2 → 3 → 4 → 6 → 1 → 7]~~

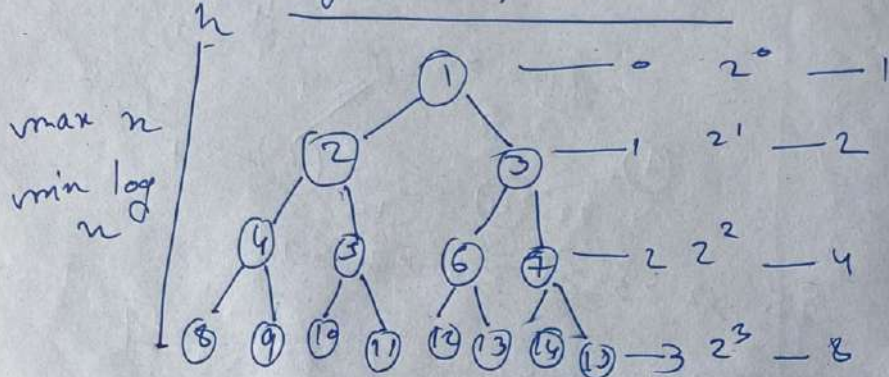
[5 → 2 → 4 → 6 → 7 → 1 → 3]

Source vertex	Destination vertex 1, 2, 3, 4, 7, 6
5	∞ ∞ ∞ ∞ ∞ ∞
5, 2	∞ 9 ∞ ∞ ∞ ∞
5, 2, 4	∞ 9 ∞ 7 ∞ ∞
5, 2, 4, 6	∞ 9 ∞ 7 ∞ 10
5, 2, 4, 6, 1	11 9 ∞ 7 ∞ 10
5, 2, 3, 4, 6, 1	11 9 13 7 ∞ 10
5, 2, 3, 4, 6, 1, 7	11 9 13 7 22

Module 4 - tree VI

Binary trees: It's a non linear hierarchical data structure. Which have a data section and a left child and a right child. A node can have at most two children

Logical Representation



A tree can have maximum node = 2^i

and maximum child $2^{(h+1)}-1$

min child ~~$2^{(h+1)}$~~ $h+1$

Terminology:

1. Node
2. Edge
3. Root
4. child
5. Leaf
6. height
7. Parents
8. siblings
9. Internal
10. Non internal

[no of possible binary trees with n nodes

$(2^n - n)$

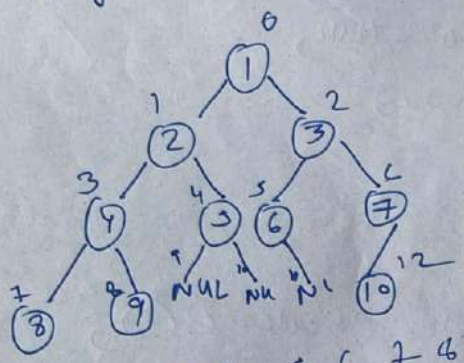
and 4 nodes

$14 \left(\frac{(2n)!}{(n)!*(n+1)!}\right)$

• Representation :-

① Array Representation / Sequential :-

Sequential representation is an array representation of a binary tree. where the root node is in "0" th idx and left node child $2*i+1$ and right $2*i+2$ if the array's index starts with "0"



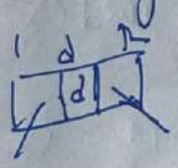
arr_tree [0 1 2 3 4 5 6 7 8 9 10 11 12
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]

root → $\frac{i}{2}$

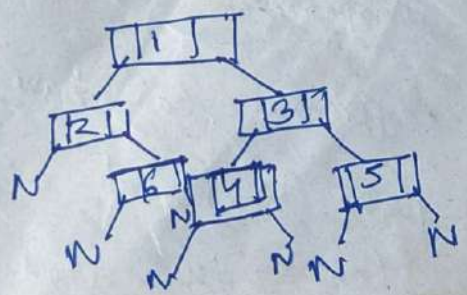
left → $2*i+1$

Right → $2*i+2$

② Linked Representation :- A common representation and implementation of a tree using linked list. We use doubly linked list

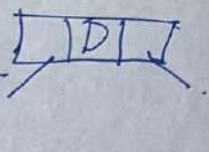


example



code :-

```
struct node {
    int data;
    struct node * left;
    struct node * right;
};
```




```
struct node * createtree (int data)
{
    struct node * newnode;
newnode = malloc (sizeof (struct node *));
    newnode -> data = data;
    newnode -> left = NULL;
    newnode -> right = NULL;
    Return newnode;
}
main () {
struct node * n = createtree (1);
n -> left = createtree (2);
n -> right = createtree (3);
free (n);
return 0;
}
```

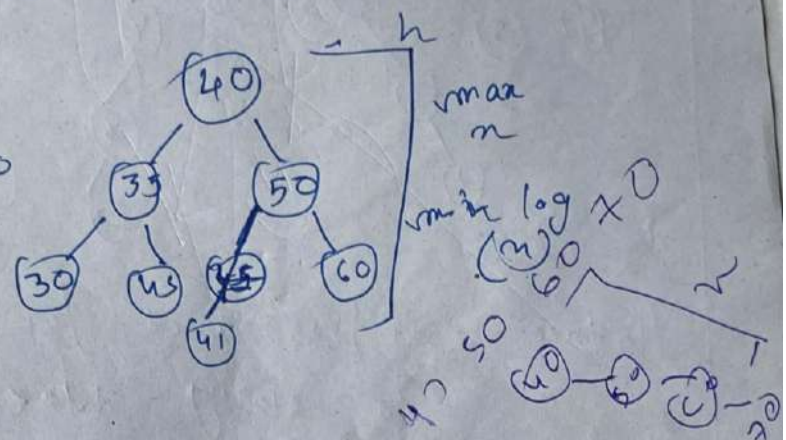
- **Bst:-** It's special kind of tree structure used to searching purpose. Searching can be done in log (n) time complexity.

Properties:-

- A node can have at max two child
- For "n" no of left ~~node~~ child is less than the root node & right. child is

is greater than the root node.

- Searching time depend on the height of the Bst.
- Max height of a Bst can have "n" time and min can have log(n) time.
- From "n" no of ~~Bst~~ trees n! Bst can be generated. [Duplicates are not allowed].



key :- 41
41 > 40
50 > 41
41 = 41

① ② ③ [Result]

Algo_bin (root, key)
begin :=
if (root == NULL)
return NULL;

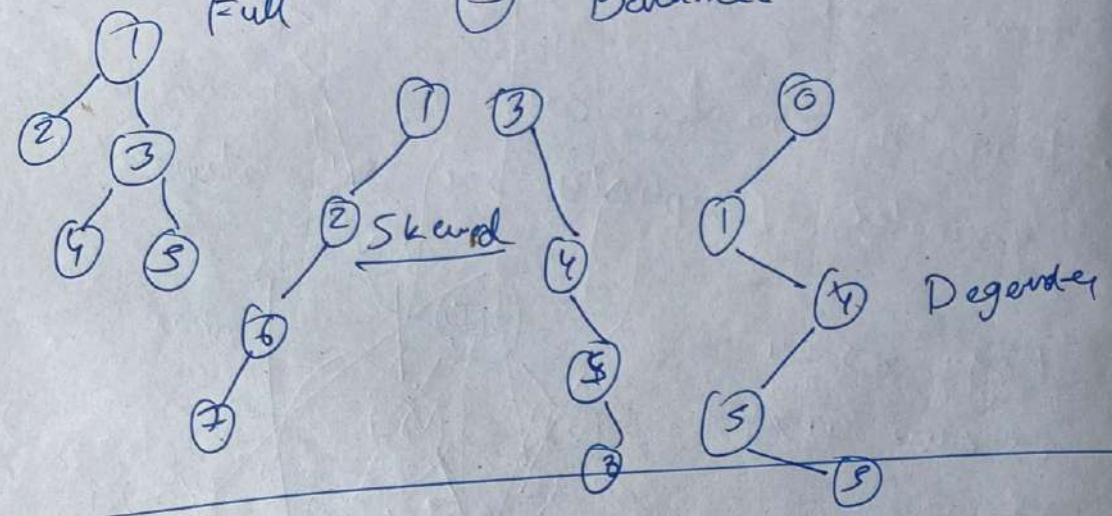
if (root == (d) root == key)
return Root;

else if (d) root > key)
return (root->left, key

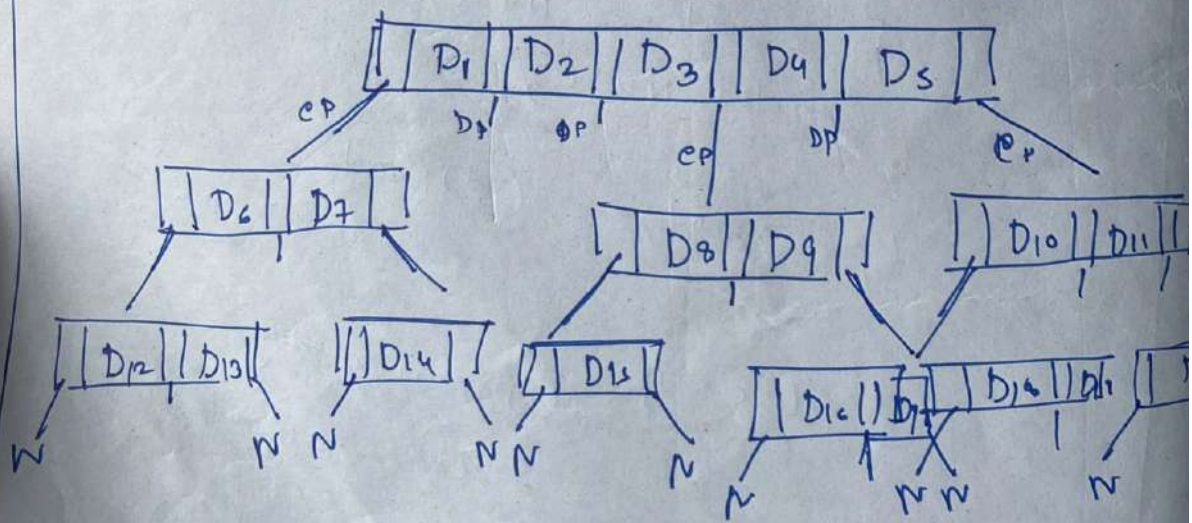
else return (root->right, key);

end :=

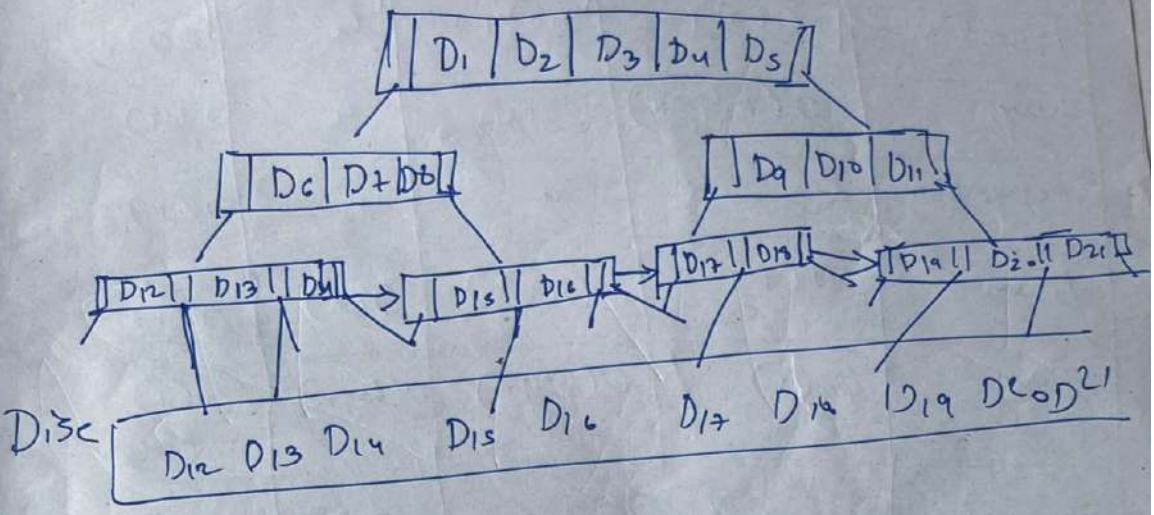
- Types of trees
  1. Full binary
  2. Perfect "
  3. Skewed "
  4. Degenerate "
  5. Balanced "



B tree :- It is used for multilevel indexing to storing data efficiently. Balanced tree because all the leaf are at the same level and non leaf, internal node also can have DP



B + tree :- Same as B tree But only leaf nodes can have DP which points to a disc sector in a secondary memory and all the leaf nodes are connected with a next node pointer like a linked list



B tree	B+ tree
1) Pointers can be found at any node	1) only at leaf node
2) Since pointers can be found at any node search time maximizes	2) only in leaf node minimum time.
3) Insertion is difficult	3) Insertion is easy
4) Deletion is time taken	4) Deletion is easy
5) B tree can have maximum time 'n'	5) minimum time $\log(n)$
6) No Linked list	6) Linked list

Schedule

Time Complexity

Space complexity

	best	ave	worst	S.C
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble	"	"	"	$O(1)$
Insertion	$O(n^2)$	"	"	$O(1)$
heap	$O(N \log N)$	"	"	$O(1)$
merge Sort	"	"	"	$O(N)$
Linear	$O(1)$	$O(n^2)$	$O(n)$	$O(1)$
binary	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Dijkstra	$O(\lvert V\rvert + \lvert E\rvert (\log \lvert V\rvert))$			
Prim's	$O(V \log V + E \log \lvert V\rvert) = O(E \log V)$ // $O((V+E) \log V)$			
Kruskal	$O(E \log E)$			

Linear

1) It's a sequential search technique

2) $O(n)$

Binary

1) Every time it eliminates the half of the array at each iteration and compare the mid element with the key

2) $O(\log n)$

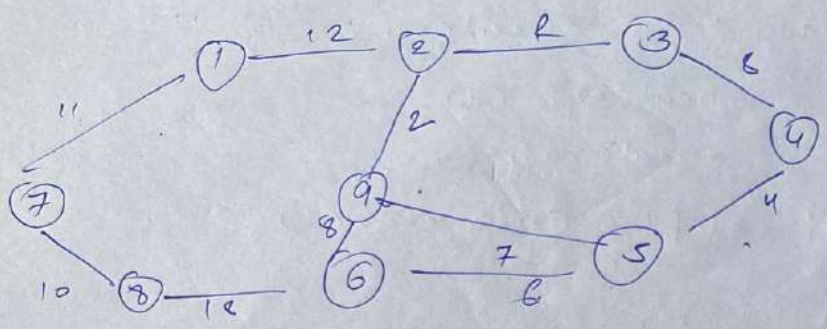
>> Works for small data set	>> Big dataset
>> Deals with sorted and unsorted both data set	>> Only sorted

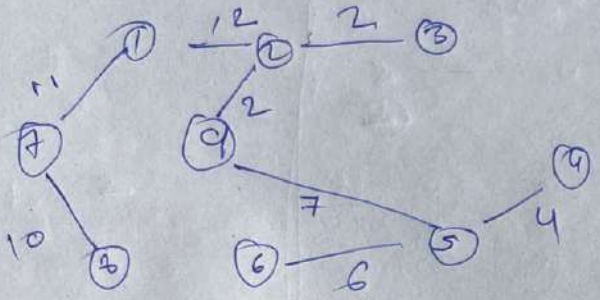
>> for (i = 1 to n)
if (a[i] == x)
return i;
else -1;
	>> mid = (i+j)/2
	if (i == j)
	return -1;
	if (a[mid] == x)
	return mid;
	else if (a[mid] > x)
	j = mid+1;
	fun call ();
	else
	i = mid+1;
	fn call ();

Jntu, SK

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
    int data;
    struct node * left;
    struct node * right;
};
struct node * root = NULL;
```





MST = 6 + 4 + 7 + 2 + 2 + 12 + 11 = 44

V = {1, 2, 3, 4, 5, 9, 8, 7}

V = {1, 2, 3, 4, 9, 8, 7}

V = {1, 2, 3, 9, 8, 7}

V = {1, 2, 3, 8, 7}

V = {1, 3, 8, 7}

V = {1, 8, 7}

V = {8, 7}

V = {8}

V = {}

A = {6}

A = {5, 6}

A = {4, 5, 6}

A = {4, 5, 6, 9}

A = {4, 6, 5, 9, 2}

A = {2, 3, 4, 5, 6, 9}

A = {1, 2, 3, 4, 5, 6, 9}

A = {1, 2, 3, 4, 5, 6, 7, 9}

A = {1, 2, 3, 4, 5, 6, 7, 9, 8}

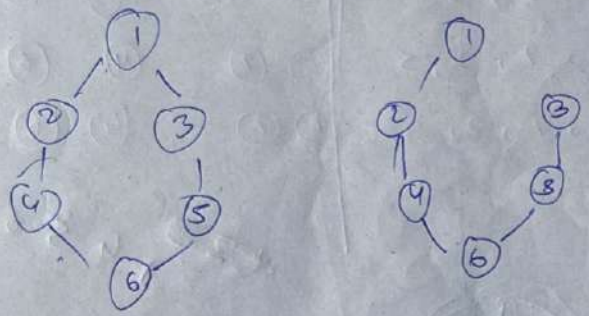
6 → 7 = 4
6 → 8 = 18
5 → 4 = 4
6 → 9 = 8
6 → 8 = 7
4 → 3 = 8
5 → 9 = 7
6 → 9 = 8
6 → 8 = 18
4 → 3 = 8
6 → 8 = 18
9 → 2 = 2
6 → 8 = 18
2 → 1 = 12
2 → 3 = 2
2 → 1 = 12
6 → 8 = 18
7 → 7 = 11
6 → 8 = 18
7 → 8 = 10

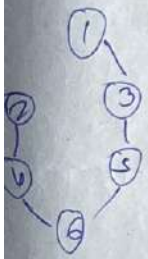
Spanning tree :- Spanning tree is a Subgraph of a graph

$S \subseteq G$

S is a subgraph of "G" which |V| vertices must be present in "S" and |V|-1 edges, inorder to make a spanning tree a graph must ensure, - Connected
- Completed
- Acyclic
- |V|
- |V| - 1

$n^{(n-2)}$





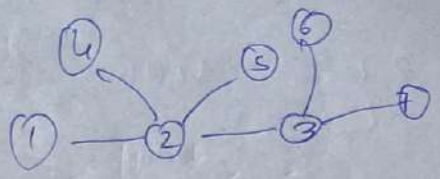
7^{5} $n^{(n-2)}$

Dijkstra :- It's a method for finding single source shortest path for a graph "G". It was conceived by the dutch computer Scientist Edsger W. Dijkstra in 1956. The complexity of dijkstra is $O(V)^2$ $(|V| \times |V|)$

it can be improved to $O((V+E) \log |V|)$

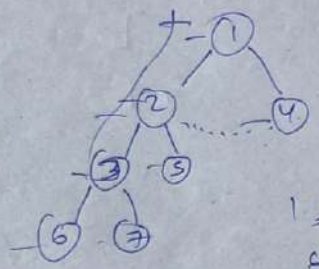
$O(|V| + |E| \cdot \log |V|)$

BFS :- Breadth first search



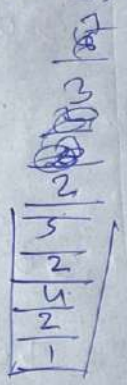
Q [1 2 3 4 5 6 7]

Result 1 2 3 4 5 6 7



1, 2 3
4, 7, 5,
4

DFS :- Stk

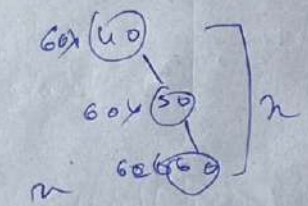


Result :- 1 4, 5, 3,
6 2, 6, 7, 3

Bst

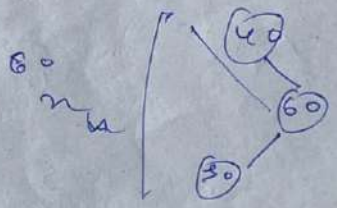
40 50 60

key = 60

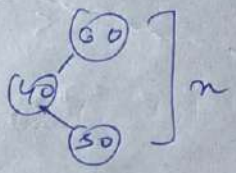


3 = 3!

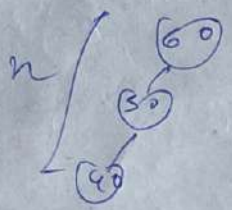
40 60 50



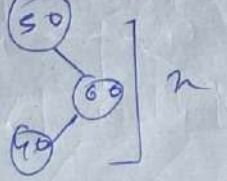
60 40 50



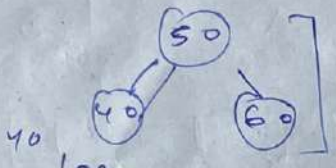
60 50 40



50 60 40



50 40 60



Selection

22 40 14 30 66

14 40 22 30 66

14 22 40 30 66

Bubbl

22 40 14 30 66

22 14 40 30 66

22 14 30 40 66

5

4 4 3 2 1

4 5 3 2 1

4 3 5 2 1

4 3 2 5 1

3 2 1 5 P-1

3 4 2

3 1 5

3 4

2 1 5

1 4 5 P-2

2 3

2 1 4 5

3 4 5 P-3

Binary Search

key = 4

0	1	2	3	4
5	15	10	18	60

1) mid = 2 == 4 ✗
2) mid = 3 == 4 ✗
3) mid > 4 ✗
   mid < 4 ✓
4) mid = = 4 ✓

0 + 4
= 4/2 = 2

3
i

3+4/2 = 7/2
3

T.C

Selection	O(n)	O(n)	O(n)	O(1)
Bubble	O(n)	O(n)	O(n)	O(1)
Insertion	O(n)	O(n)	O(n)	O(1)
Linear	O(1)	O(n)	O(n)	O(1)
Binary	O(1)	O(log n)	O(log n)	O(1)
Heap	O(n log n)	n(log n)	O(n log n)	O(n log n)
Quicksort	4	4	4	O(n)
Dijkstra	O((V+E) + log(V))		O(V²)	
Prims	O((V+E) + log(V))		O(V²)	

1) Algo sum_two matrices ——— T.C

2) " Multiplication ——— "

3) " nested loop — $O(n^2)$

4) " Linear loop — $O(n)$

5) (n+1) — for (i=0; i<n; i++){

(n+1)×n — for (j=0; j<i; j++){

n×n — n — stmt;

$O(n^2)$ } }

6) for (i=1; i<n; i=i*2){ — logn

stmt; ——— (logn)

}

or $2^x = n$

$2^x = n$

$x = (\log_2 n)$

$O(\log_2 n)$

i	[value of i]
1	i = i*2
2	2 = 2*2
2^2	$2^2 = 2^2 \times 2$
2^3	$2^3 = 2^3 \times 2$
2^4	$2^4 = 2^4 \times 2$
.	.
.	.
2^x	?

Algo sum (int A[], int n){ — 1

7) int sum = 0 — 1

for (i=0; i<=n-1; i++){ — n+1

sum = sum + A[i]; — n

}

return; — 1

int main (){ — 1

int A[] = {n}; — 1

n = sizeof (A)/sizeof (A[0]); — 1

cout << sum (A, n); — 1

}

— 1

$O(n)$ $O(2n)$ $O(2n+1)$

S
A — n
n — 1
i — 1
$O(n)$

T.C

$O(n)$

$O(n)$

multi (A[], B[], n) {

c[]

for (i=0; i<=n-1; i++) {

n

n+1 — for (i=0; i<=n-1; i++) {

(n+1)*n — c[i] for (j=0; j<n; j++) {

c[i][j] = a[i][k] * b[k][j];

for (k=0; k<=n-1; k++) {

c[i][j] = a[i][k] + b[k][j];

}

}

}

T.C

$2n^3 + 3n^2 + 2n + 1$

$O(2n^3)$

$O(n^3)$

S.C

i — 1

j — 1

A — n^2

C — n^2

B — n^2

n — 1

$O(n^2)$

for (i=1 to n) {

for (j=1 to i) {

stmt;

}

}

i	j	en
1	1	1<10
2	1	1<2
	2	2<20
3	1	
	2	
	3	

$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} = \frac{n^2+1}{2} = O(n^2)$

Time Complexity:

1) for (i=0; i<n; i++) — n+1

Multiplication

for (j=0; j<=n-1; j++) — n*(n+1)

for (k=0; k<n; k++) — n*n*(n+1)

Stmt; — n*n*n

$t(n) = 2n^3 + 3n^2 + 1$

$t(n) = O(n^3)$

t.c = $O(n^3)$

s.c = $O(n^2)$

2)

for (i=0; i<n; i++) — (n+1)

for (j=0; j<n; j++) — n*(n+1)

C[i][j] = A[i][j] + B[i][j]; — n*n

$t(n) = 2n^2 + 2n + 1$

$t(n) = O(n^2)$

t.c = $O(n^2)$

s.c = $O(n^2)$

s.c	
A	n^2
B	n^2
C	n^2
n	1
i	1
j	1
k	1
	$3(n^2)+4$
	$O(n^2)$

s.c	
A	n^2
B	n^2
C	n^2
n	1
i	1
j	1
	$3n^2+3$
	$O(n^2)$

3) for (i=0; i<n; i++) — (n+1)

for (j=0; j<i; j++)

Stmt;

$O(n^2)$

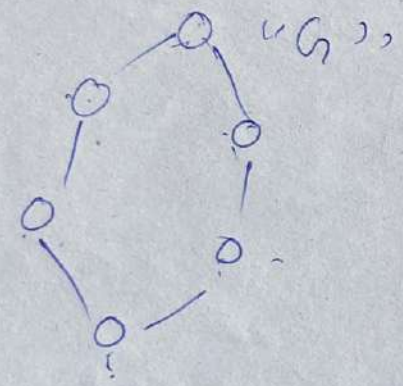
s.c	
i	1
n	1
j	$1+2+3+4+5+\dots+n$
	$O(n^2)$

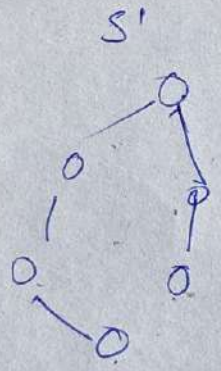
$\frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$

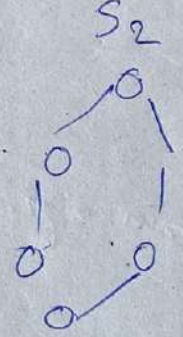
s.c = O(1)

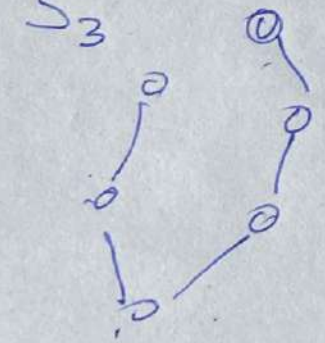
T.c = $O(n^2)$.

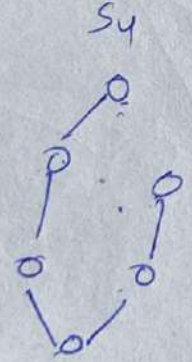
i	j	No. of times executed
0	0	0<0
1	0	0<1
2	0	0<2
	1	1<2
3	0	0<3
	1	1<3
	2	2<3
4	0	0<4
	1	1<4
	2	2<4
	3	3<4
n		

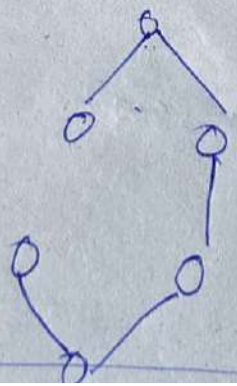


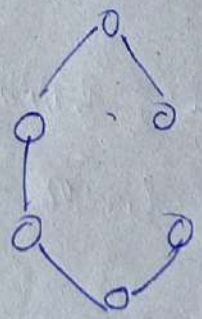












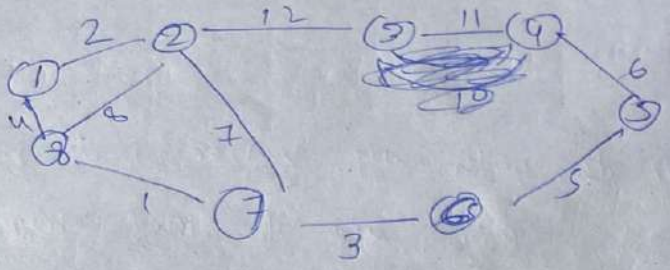
Prims:- It is a gr
Cost spa
efficient and

Procedure:-

Step 1:- Star
minim

Step 2:- Now

Step 6:- Return the MST and main function and exit.

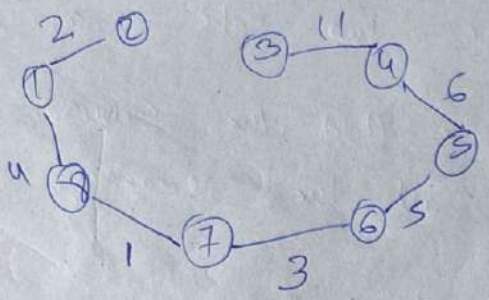


7→2=7
7→6=3
7→4=1

7→2=7
7→6=3
4→1=4
8→2=8

6→5=5
7→2=7
8→1=4
8→2=8

1→2=2
6→5=5
7→2=7
8→2=8



1+3+4+2+5+6+11
= 32 (Ans)

2→3=12
6→5=5

V = {1, 2, 3, 4, 5, 6, 8}
A = {7}

V = {1, 2, 3, 4, 5, 6}
A = {7, 8}

2→3=12
5→4=6

V = {1, 2, 3, 4, 5}
A = {6, 7, 8}

2→3=12
4→3=11

V = {2, 3, 4, 5}
A = {1, 6, 7, 8}

V = {4, 3, 5}
A = {1, 2, 6, 7, 8}

V = {4, 3}
A = {1, 2, 5, 6, 7, 8}

V = {3}
A = {1, 2, 4, 5, 6, 7, 8}

V = {∅}
A = {1, 2, 3, 4, 5, 6, 7, 8}

Graph terminologies

(1) $G = (V, E)$

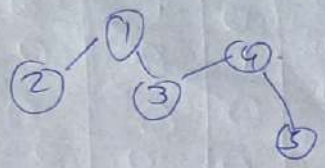
(2) vertex :-

(3) Edge :-

(4) Direct edge.

(5) weighted

(6) undirect

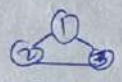


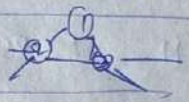
$V = \{1, 2, 3, 4, 5\}$

$E = \{(1,2), (1,3), (2,1), (3,1), (3,4), (4,3), (4,5), (5,4)\}$

3

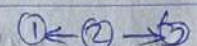
Graph type :-

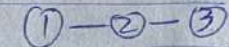
(1) Finite graph 

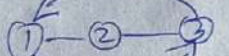
(2) Infinite graph 

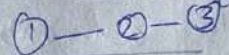
(3) NULL graph (1)

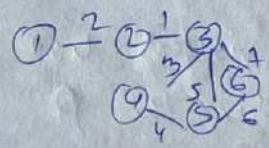
(4) Trivial graph (1) (2) (3) (4)

(5) Direct 

(6) undirect 

(7) connected 

(8) Disconnected 

(9) Weighted :- 

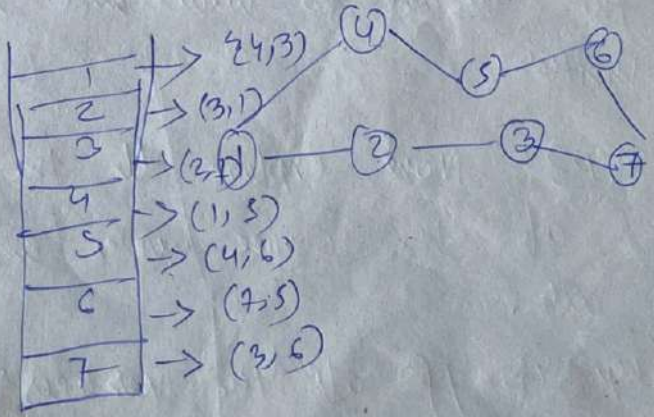
Graph Representation

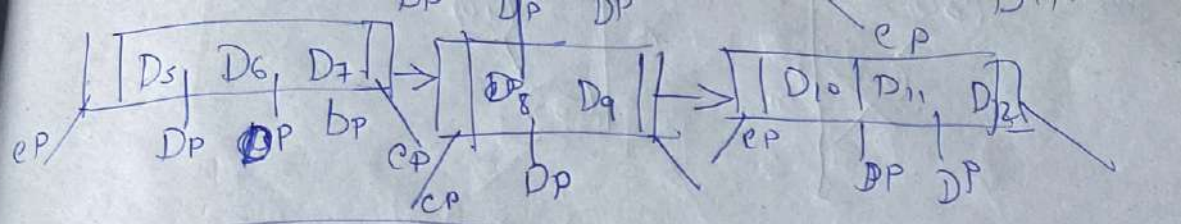
(1) Adj list

(2) " matrix

(3) cost adj "

(4) " " matrix





Bst:- It's special kind of Binary tree which is mainly used for searching purpose

Searching can be done in an efficient and minimum time $O(\log n)$.

Properties :-

1) A Bst can have at max two child.

2) N no of left child is less ~~that~~ than the root node and right child is greater than the root node.

3) Searching time depends on the height of tree

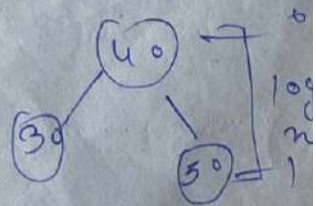
4) A tree can have max n and minimum ($\log n$)

5) For n of tree n! Bst's can be generated

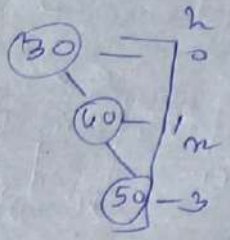
example:- 40 50 30

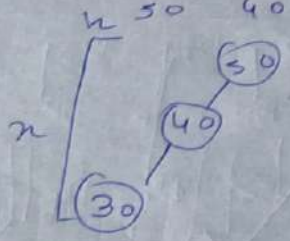
2^0
2^1
2^2

40 30 50



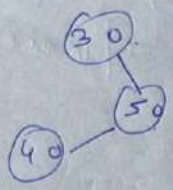
30 40 50



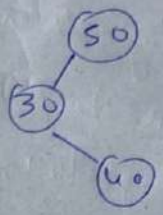


(c)

30 50 40

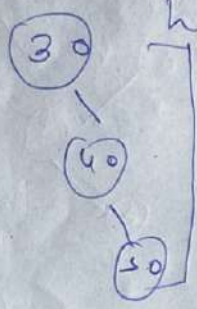


50 30 40



For different set of key we can generate different size of Binary Search tree

30 40 50



key = 50

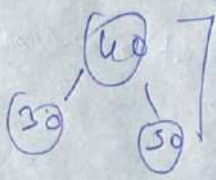
50 > 30 ✗

40 < 50

50 = 50

3 comparison

(d) 40 30 50

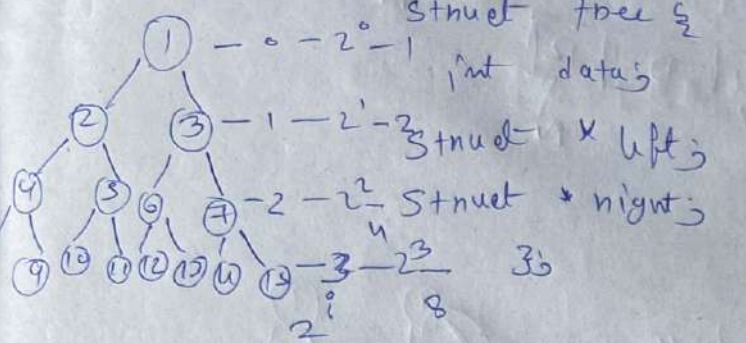


key = 50

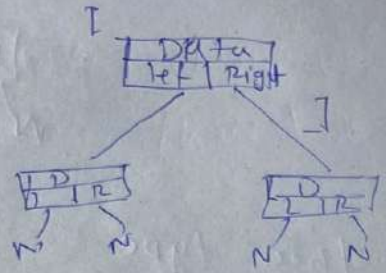
50 > 40 2 comparison

50 = 50

- ~~At max 2 child~~
- A tree structure made of node's and edges
- In C lang we define it in a structure




```
struct tree {
    int data;
    struct tree * left;
    struct tree * right;
};
```



- Non ~~lear~~ linear hierarchial data structure

max node: 2^i

max child $= 2^{(n+1)} - 1$

min " $= 2^{n+1}$

Terminologies

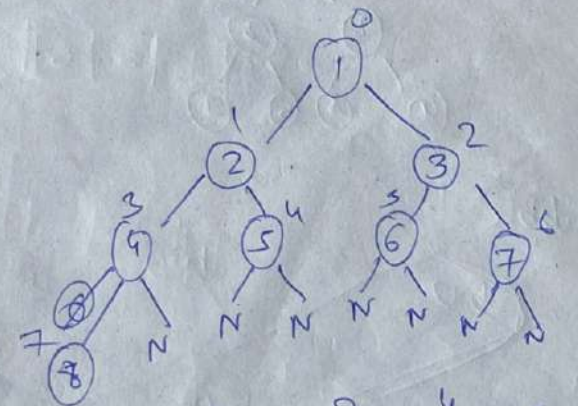
1. Root :-
2. Child :-
3. Parent,
4. Sibling
5. Ancestor
6. Descendants
7. Leaf
8. Edge
9. node
10. Height
11. Level

Sequential :- Array representation of a tree

root = 0th

L- child :- $2*i+1$

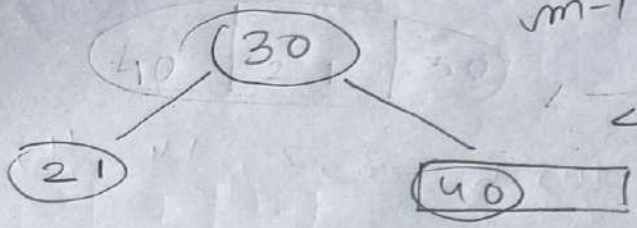
R- " :- $2*i+2$

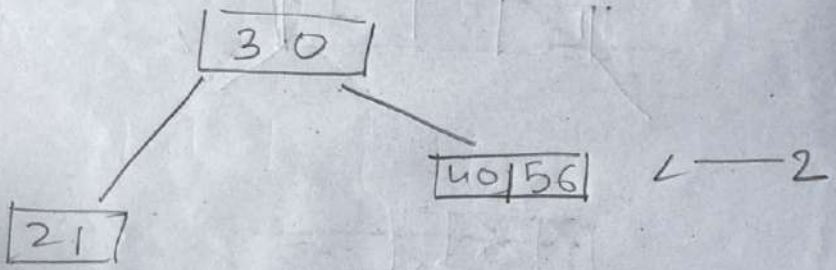


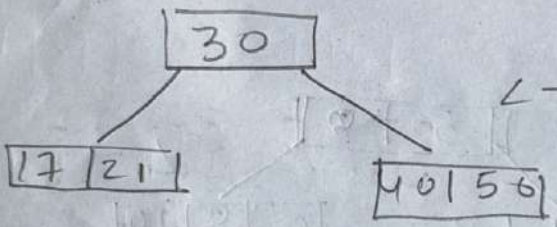
	0	1	2	3	4	5	6	7
tree	1	2	3	4	5	6	7	

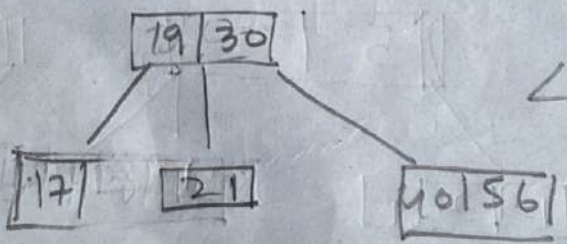
40, 21, 30, 56, 17, 19, 48, 29, 24

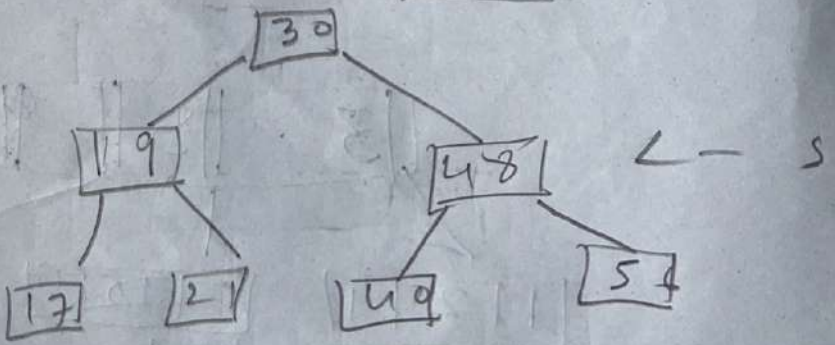
m = 3
m-1 = 2

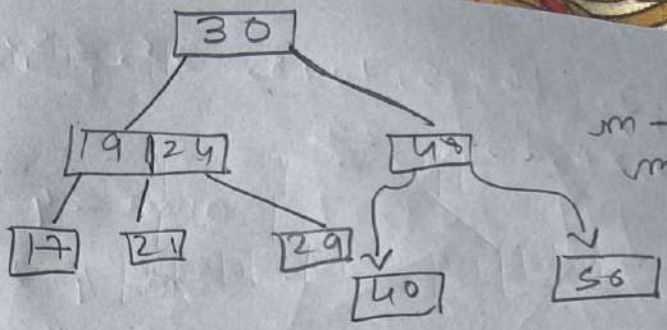






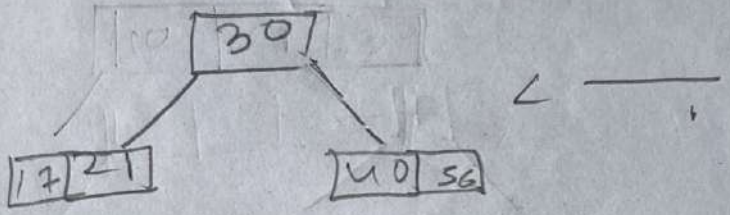


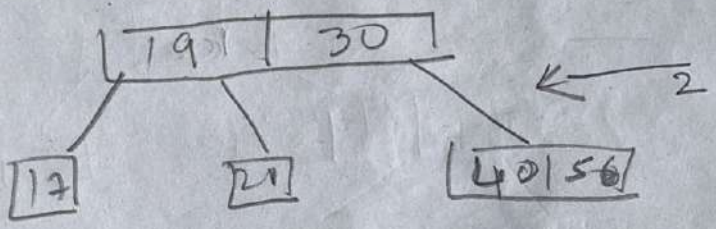


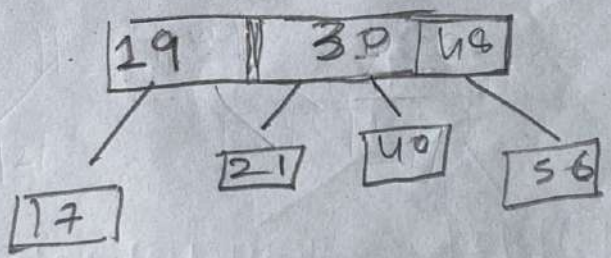


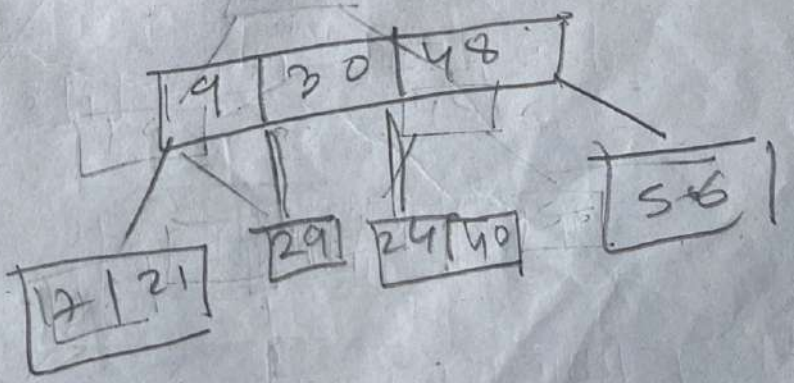
$m - 1 = 2$
$m = 3$

Order -4









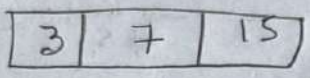
7, 15, 3, 16, 5, 1, 18, 35, 28, 39

17, 14, 20, 22, 25, 27

order = 4

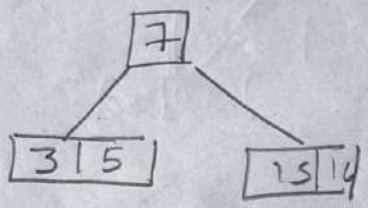
m = 4

max key : 3

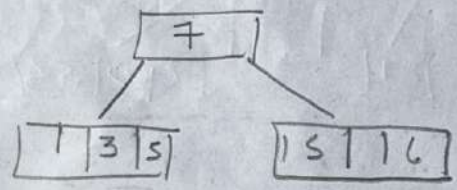


5 →

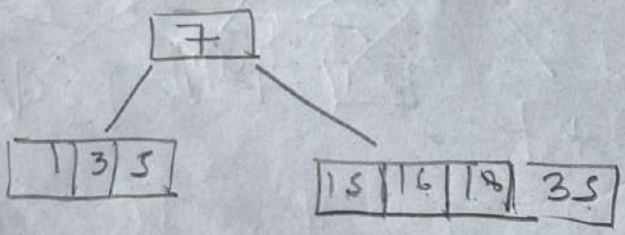
16 →



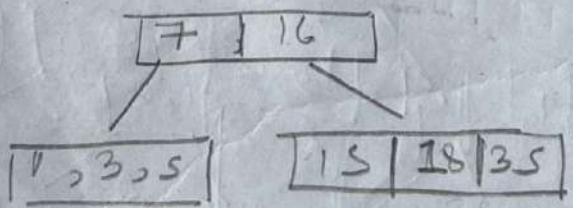
1 →



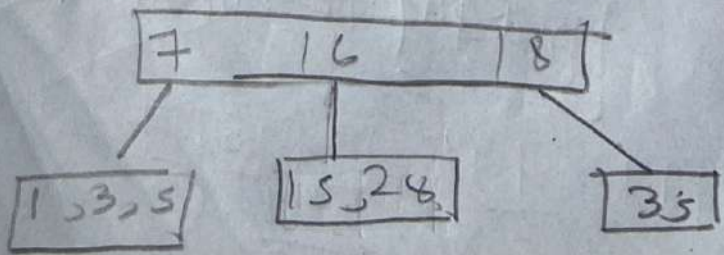
18



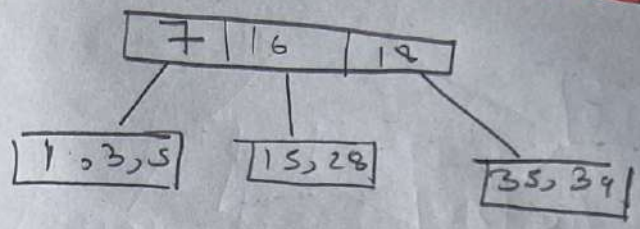
35



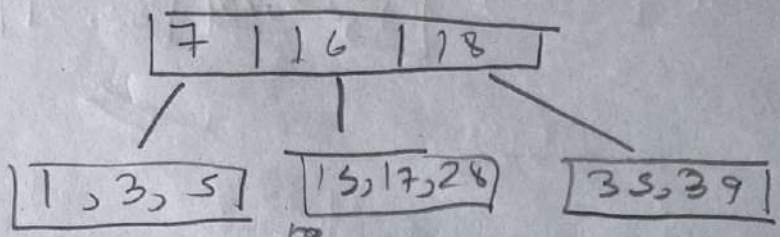
28



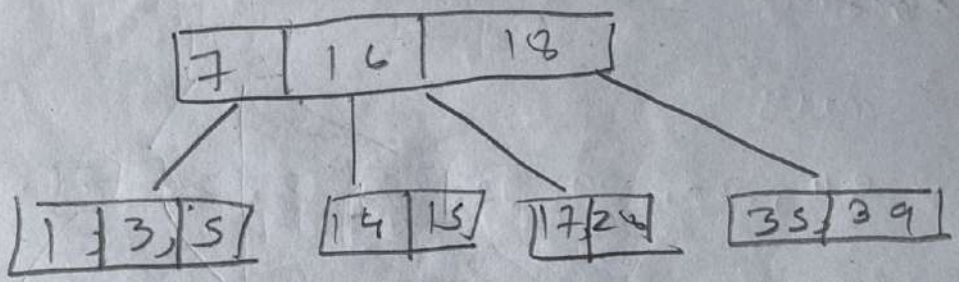
39


7 | 16 | 18
1, 3, 5
15, 28
35, 39

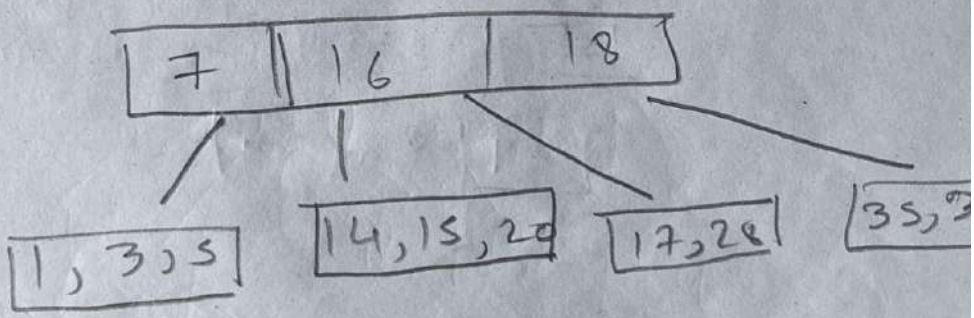
17


7 | 16 | 18
1, 3, 5
15, 17, 28
35, 39

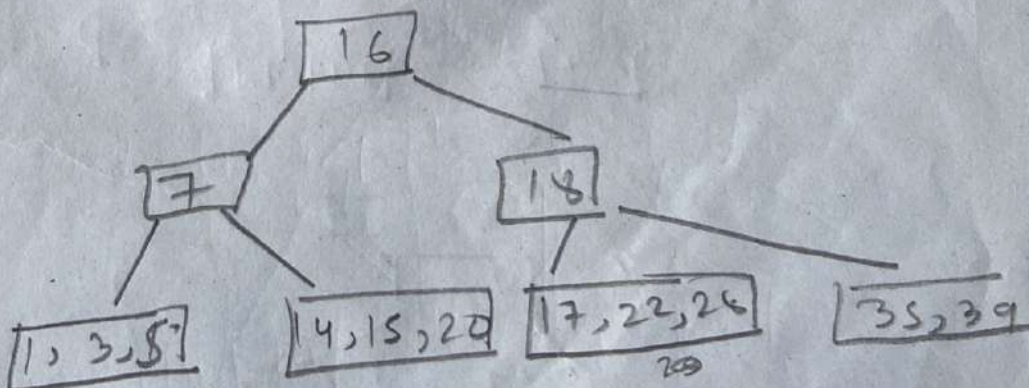
14

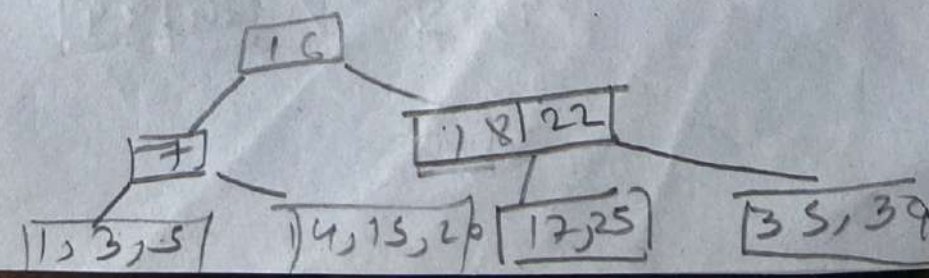

7 | 16 | 18
1 | 3 | 5
14 | 15
17 | 24
35 | 39

20

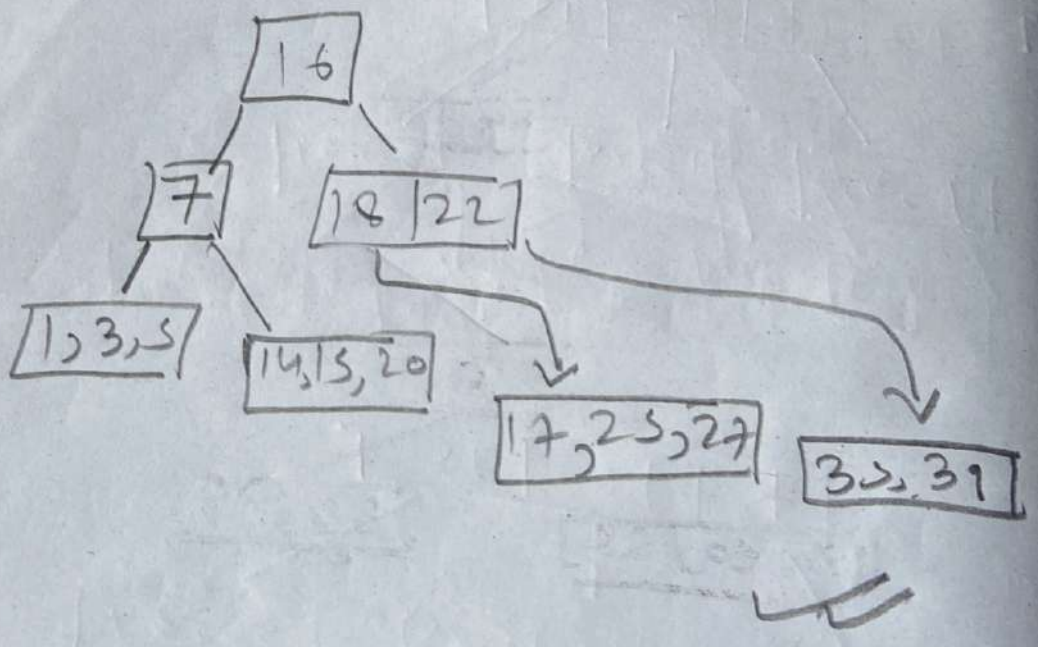

7 | 16 | 18
1, 3, 5
14, 15, 20
17, 28

22


16
7
18
1, 3, 5
14, 15, 20
17, 22, 26
35, 39


16
7
18 | 22
1, 3, 5
14, 15, 20
17, 25
35, 39

27

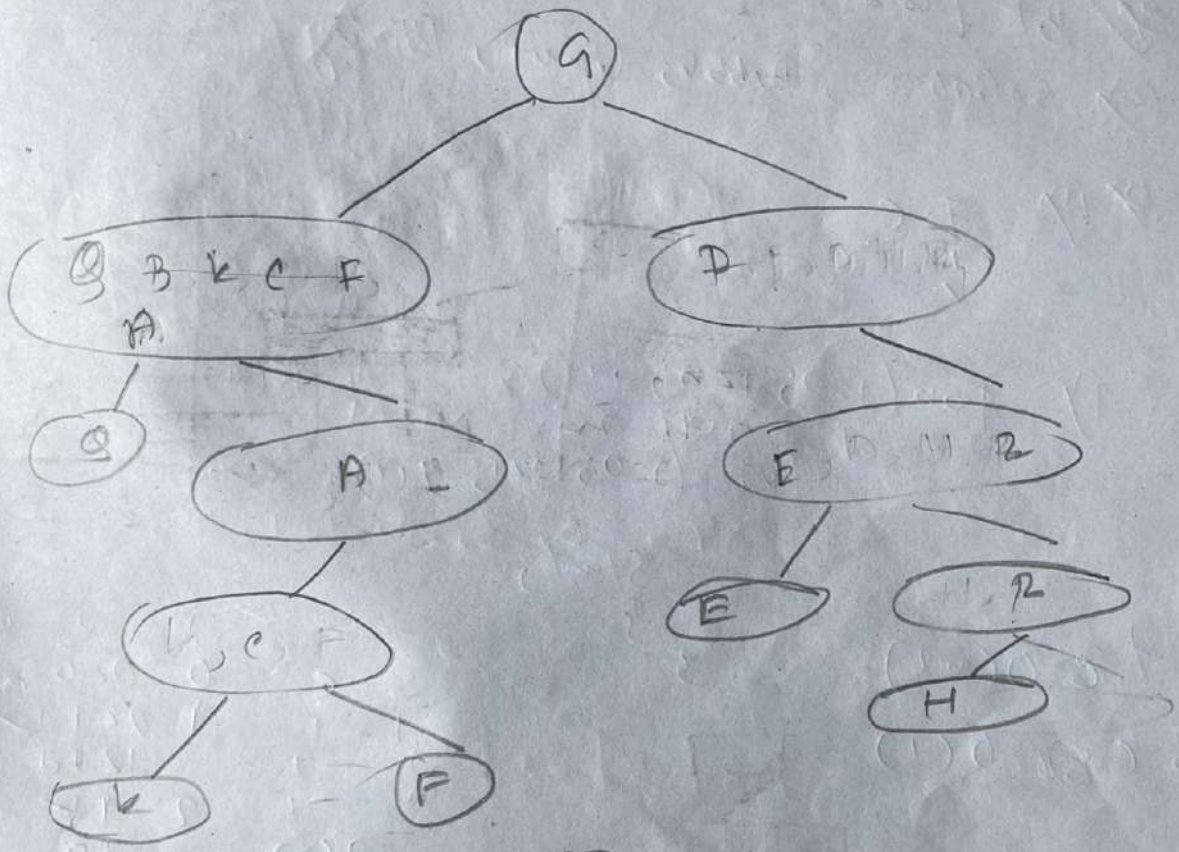


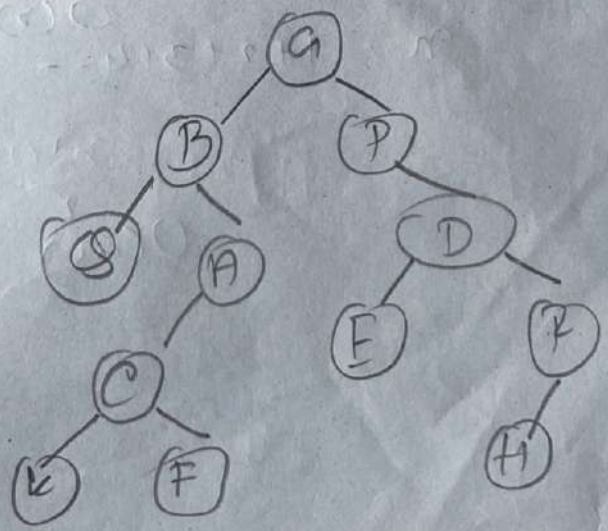
Construct a B tree of order 4 usi

eg Consider the following sequence of binary tree traversal

Inorder :- G, B, K, C, F, A, G, P, E, D, H, R

Preorder :- G, B, G, A, C, K, F, P, D, E, R, H





T.C

```
for(i=0; i<n; i++)      n+1
    for(j=0; j<i; j++)
        stmt;
```

t.c $O(n^2)$

s.c $O(1)$

	i	j	after
	0	0<0	0
	1	0<1	1
		1<1	
	2	1<2	
		2<2	2
	0		
	1		3
	2		
	3		
	0		
	1		4
	2		
	3		

s.c

i — 1
j — 1
n — 1
$O(n)$

$n = 1+2+3+4 \cdots \frac{n(n+1)}{2}$

$= \frac{n^2+1}{2}$

$O(n^2)$

for (i=0; i<n; i=i*3) { — $\log_3 n$

Stmt;

}

i
0
$1 = 1*2 = 2$
$3 = 3*2 = 2^2$
$3^2 = 3^2*2 = 2^3$
$3^3 = 3^3*2 = 2^4$
3^k

$3^k = ?$

Assume the upper mentioned loop will execute as long as $i < n$ it will terminate as long as

$i >= n$

Since $i = 2^k$

$3^k >= n$

$3^k = n$

$k = \log_3 n$

T.C

$O(\log_3 n)$

S.C

i	n
1	1

$O(1)$ $O(1)$

$O(2)$

---

for (i=0; i<n; i=i*2) { — $O(\frac{n}{2^{100}})$

Stmt;

}

$n+1$ $\frac{n}{2^{100}}$

$O(\frac{n}{2^{100}})$

Dominating $O(n)$ ✓

int P=0

for (i=0; i<n; i=i*2) { — $\log n$

for (j=0; j<P; j=j*2) { — $\log P$

Stmt

}

}

$P = \log n$

$P = \log \log n$

$\log n \times \log$

```
for (i=0; i<bound; i++) {                    —n+
    for (j=0; j<n; j+=2) {                   —n×(n+1)/2
        sum+=j;                              —n×(n
    }
    for (j=1; j<n; j*=2) {                   —n×(n)×log n
        sum+=j;                              —n×n×log n
    }                                        —n×n×log n
```

$3n\log n + 3n^2 + 2n + 1$

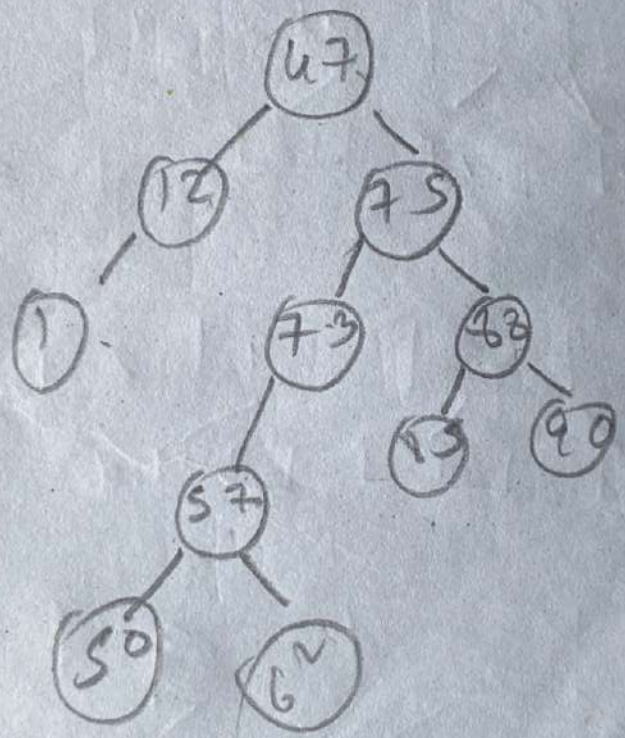
$= 3n\log n$

$= O(n\log_2 n)$

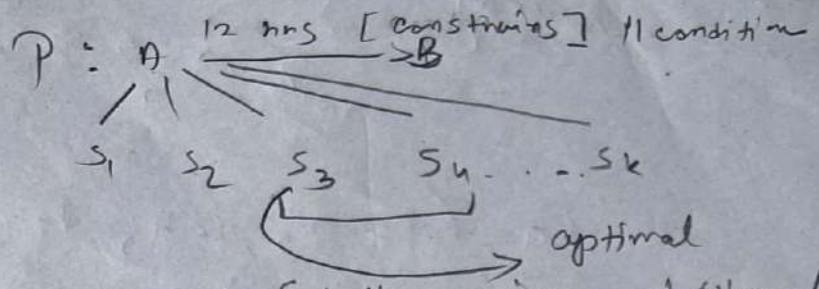
---

of array 4:- 5, 3, 21, 9, 1, 13, 2, 7, 10,

Bst 47, 12, 75, 88, 96, 73, 57, 1, 85, 50, 62

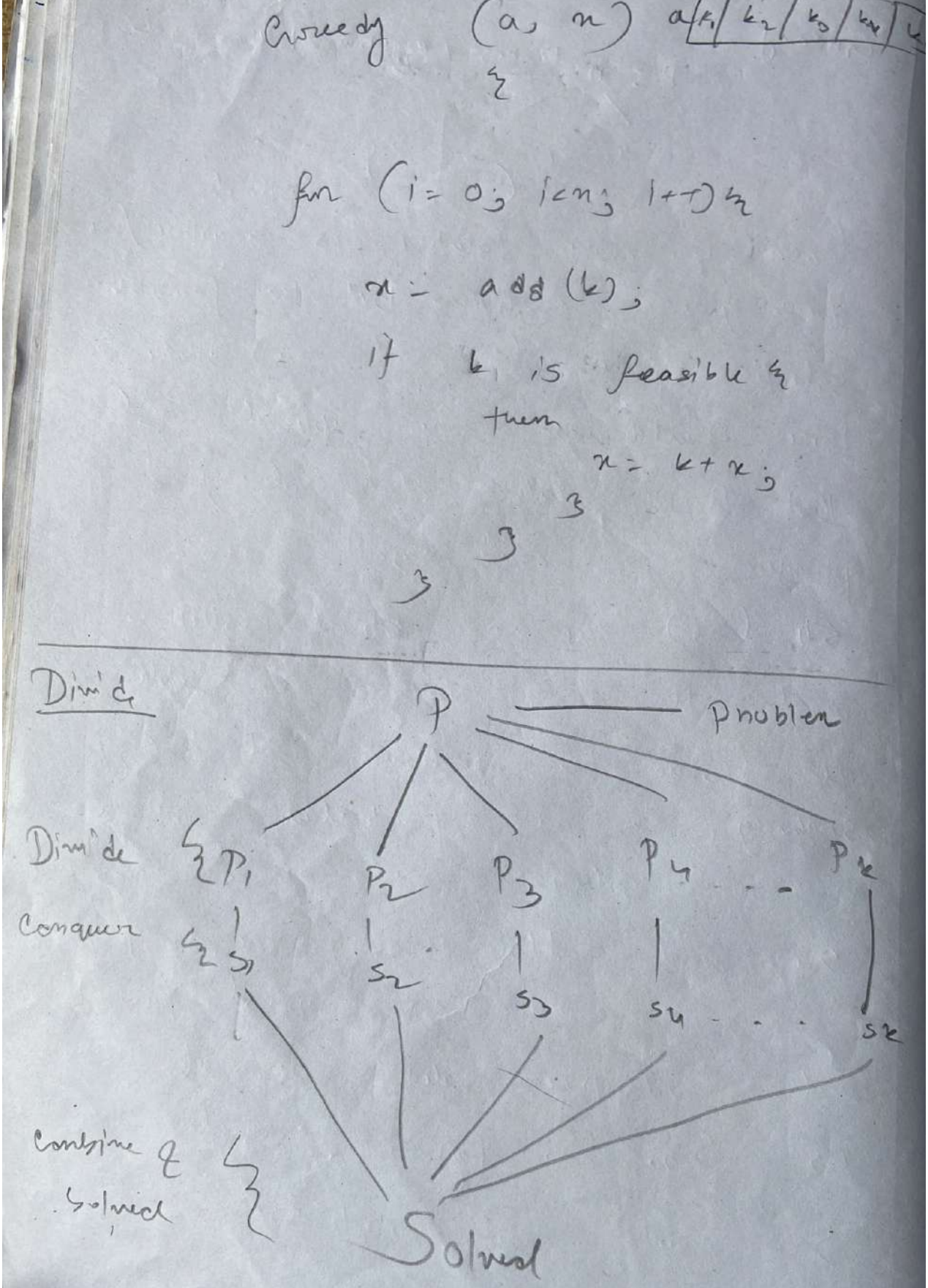


Greedy



Given a every solution is adaptive for the given condition But the feasible solution's are (S_3 and S_4) since there could be one optimal and ~~Best~~ Best solution for solving a problem in this case of these ~~S_4~~ (S_3) as an optimal solution and all the solutions can able to solve the problem but being sticking on a particular condition as two solutions are chosen S_3 and S_4 and after choosing these of these as the minimum cost optimal feasible and adaptive solution out of them is S_3. So for this particular problem S_3 should .the solution.

```
Greedy (a, n)
{
for (i = 0; i<n; i++) {
    x = add (k);
    if k is feasible {
        then
            x = k + x;
    }
}
}
```



```
algo DAC (D) {
    if (D (small)) {
        S (D);
    }
    else {
        Divide P1, P2, P3, P4, Pk;
        Apply DAC (P1, P2, P3, P4 ... Pk);
        combine (DAC (P1) + (P2) + (P3) + (Pk));
    }
}
```

Analysis: If a problem P is given & the problem is big cannot solved it directly. Break this problem into smaller & subproblems and find the solution for each subproblem & and recursively solved & if then add all the subproblems "k" and combine them and make big problem solved. Algorithm is recursive & every subproblem must be same nature if the problem is to sort

emty.

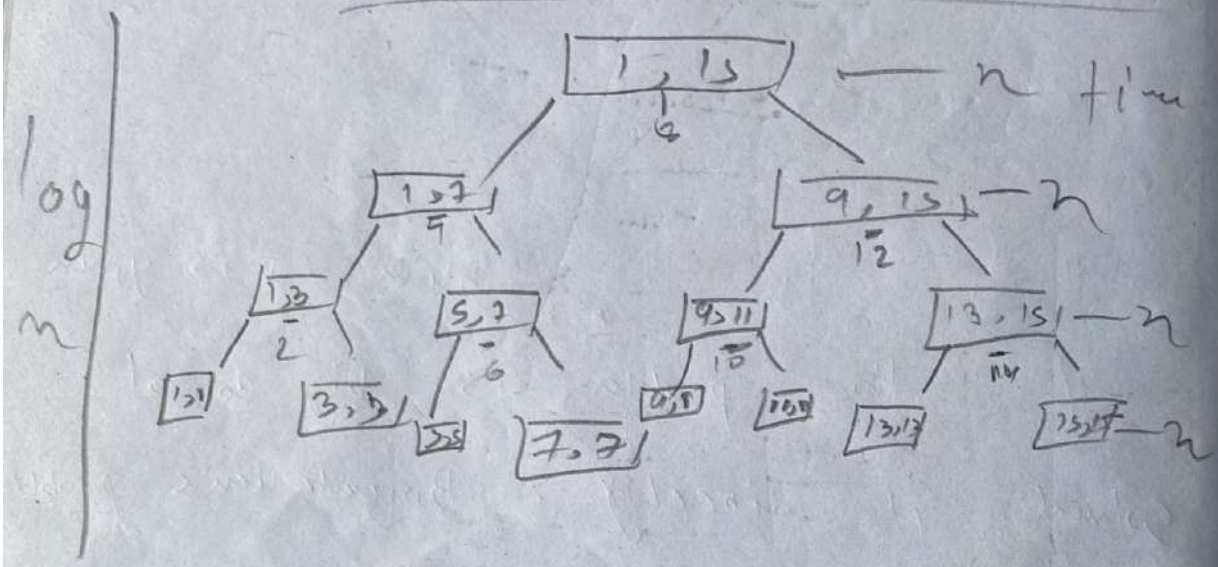
$\frac{1+15}{2} = \frac{16}{2} = 8$

Quick Sort

$\frac{1+15}{2} = \frac{16}{8} = 8$

[1 . . . . . 15]

Suppose the ~~mid~~ element will be partition way come at



The time complexity is depend on this particular constrain is [n times] height logn

Θ nxlogn

$O(n\log n)$

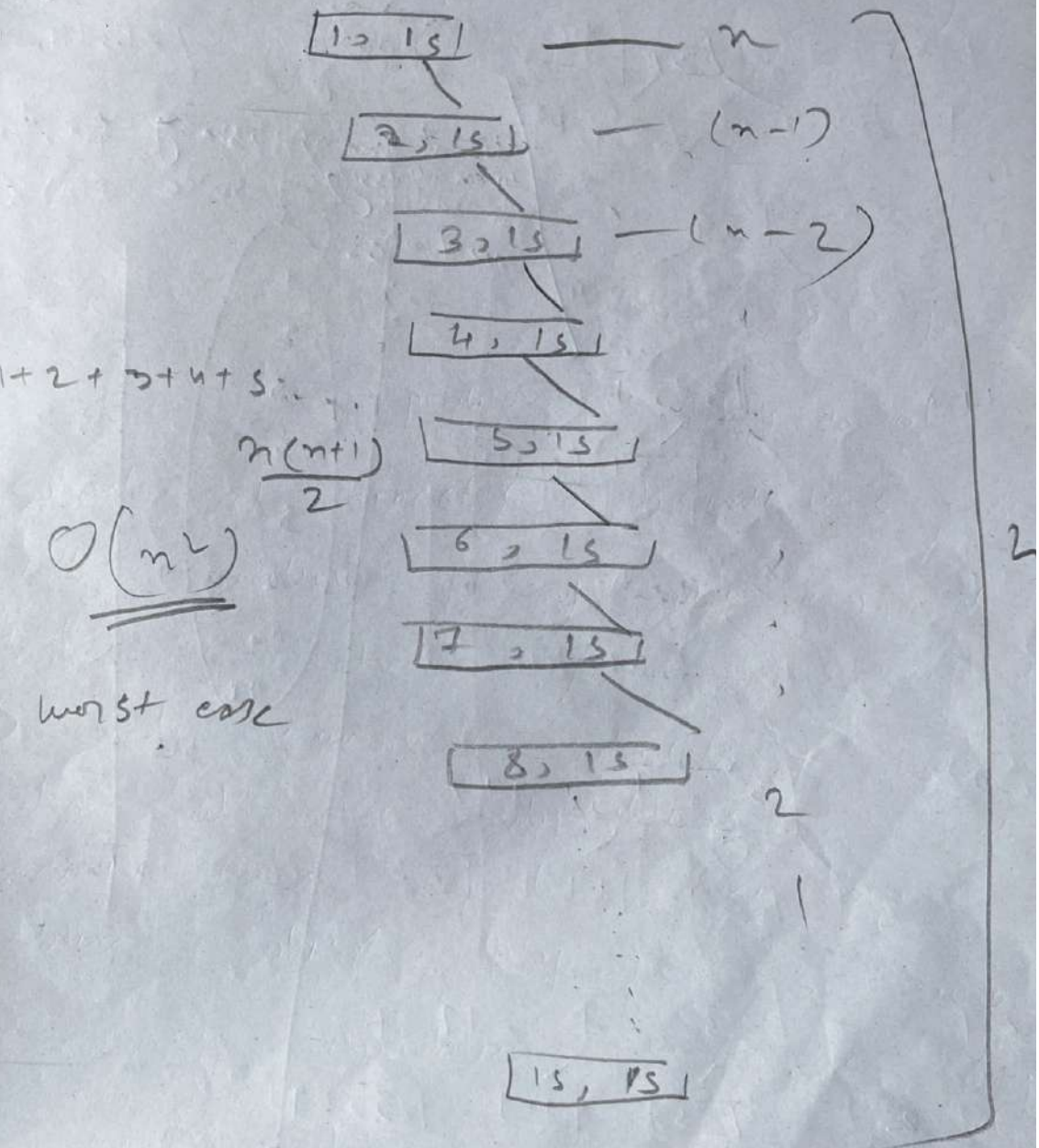
[best case average case]

Worst

Suppose for partition

log done at beginning

[1 - - - - - - - - - - 15]

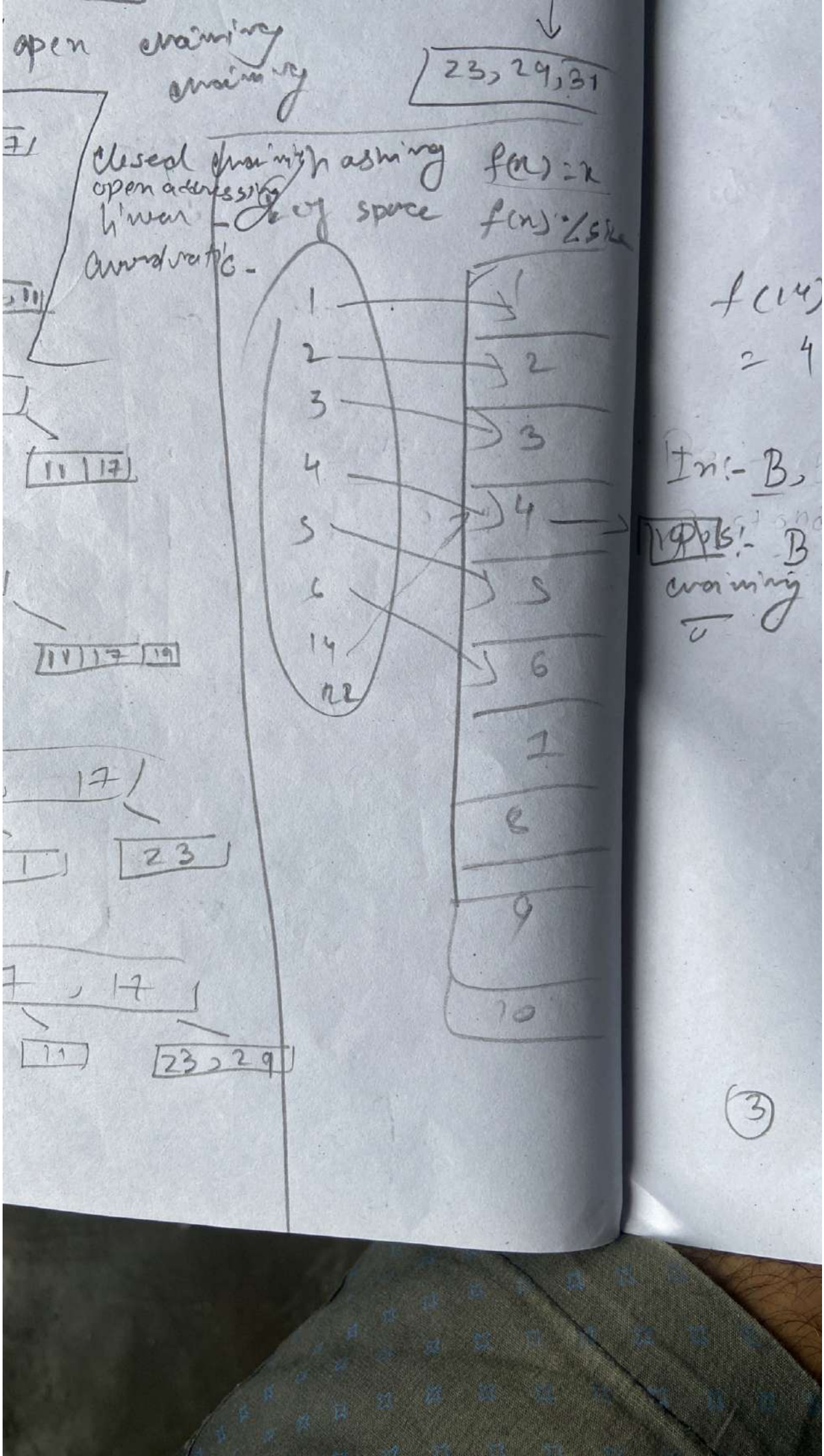


1+2+3+4+5 ... n

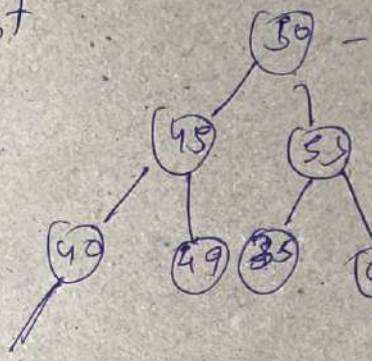
$\frac{n(n+1)}{2}$

$O(n^2)$

worst case


open chaining
chaining
23, 29, 31
Closed hashing
open addressing
linear
Overview -
f(x) = x
f(x) = x%5
1
2
3
4
5
6
14
22
11 17
11 17 19
17
23
11
23, 29
f(14)
= 4
In:- B,
B
chaining
C
(3)

Bst



2^0

2^1

$2^2 = 2^i$

Code:- struct node * search (struct node * root, int key)

{

if (root == NULL) {

return NULL;

}

if (root->d == key) {

return root;

}

else if (root->d < key) {

return search (root->left, key);

}

else {

search (root->right, key);

}

}