3110 098 229

Data Structure And Algorithms

ALGORITHM →

- Is a procedure having well defined steps for solving a particular problem.
- Has finite set of logic / instructions, written in order for accomplishing the predefined task.
- Not complete soln of code, its just soln (logic) of a problem.
- Can be represented by informal description using a flowchart or Pseudo code.

CATEGORIES OF ALGORITHM —

- SORT – Algorithm developed for sorting items in certain order.
- SEARCH – Algorithm developed for searching the items inside a data structure.
- INSERT – Algorithm developed for inserting an item inside a data structure.

Page : / /
Date :

- DELETE – Algorithm developed for deleting the existing element from the data structure.

- UPDATE – Algorithm developed for updating the existing element inside a data structure.

CHARACTERISTICS OF ALGORITHM (Design)

- INPUT – Must have 0 or well defined inputs

- OUTPUT – Must have 1 or well defined outputs and should match with desired input.

- FEASIBILITY – Must be terminated after finite no. of steps.

- INDEPENDENT – Must have step by step directions which is independent of any programming code.

- UNAMBIGUOS – Must be clear. Each of steps and I/O must be clear and lead to only one meaning.

CHARACTERISTICS →

Time complexity: It represents the amount of time required by program or code to the completion.

SPACE COMPLEXITY – It is the amount of memory space required by algorithm, during its course of its execution.

* Space complexity is required in situations when limited memory is available and for the multiuser system.

EACH ALGORITM MUST HAVE →

- Specifications –
  Description of the computational procedure.
- Pre-conditions-
  The condition (s) on input.
- Body of an Algorithm –
  A sequence of clear and unambigous instructions.
- Post conditions –
  The conditions on output.

Examples →.

Algorithm to multiply two nos x and y and display result in z.

STEP 1: START
STEP 2: declare three integers x, y & z
STEP 3: define values of x and y

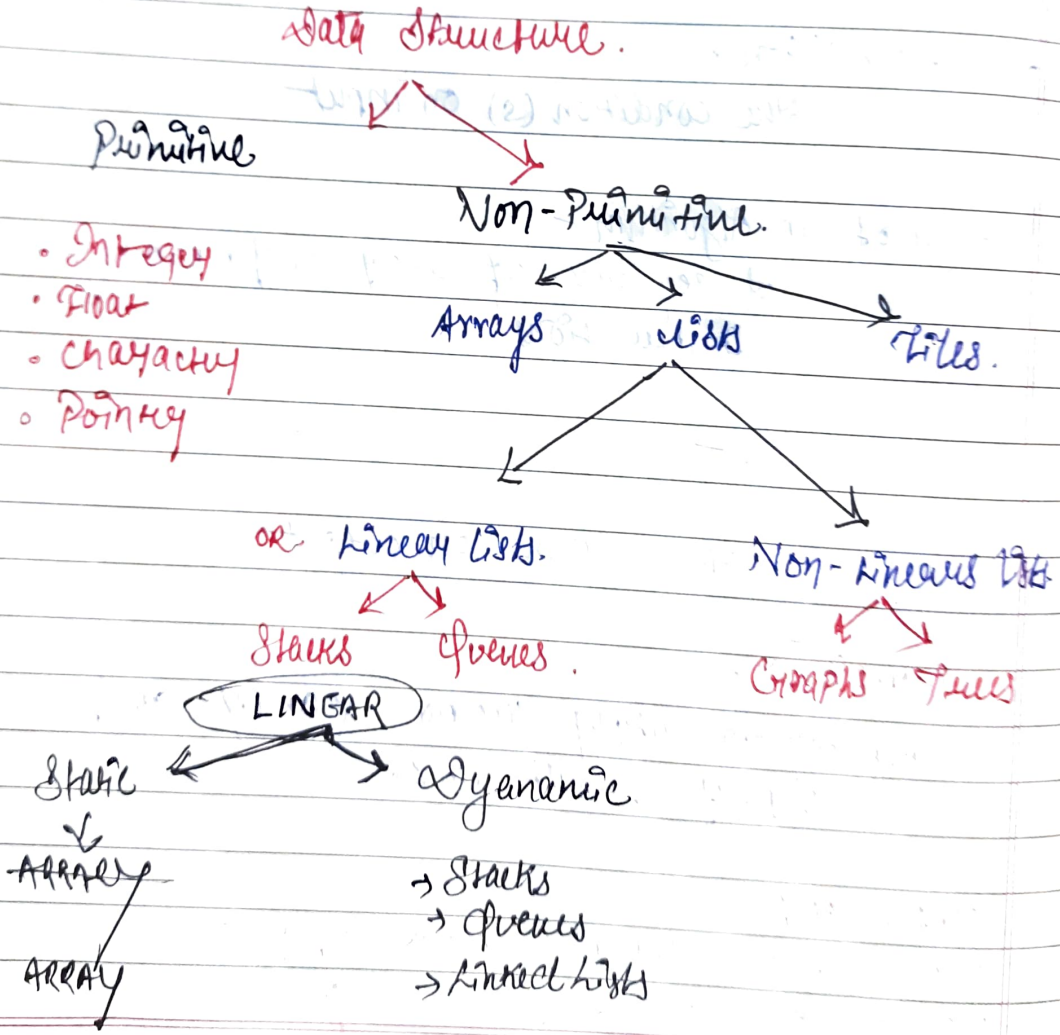
Date : / /
Page :

STEP 4: Multiply values of x & y
STEP 5: Store the o/p of step 4 in Z
STEP 6: print Z
STEP 7: STOP

Alternative

Step 1 : START MULTIPLY
Step 2 : get values of x & y
Step 3 : Z ← x*y
Step 4 : display Z
Step 5 : STOP

CLASSIFICATION OF DATA STRUCTURE.



Primitive Data Structures →

These are basic data structures and are directly operated upon by machine instructions.

These generally have different representations on different computers.
Integers, floating point numbers, character constants, string constants, pointers etc.

Non-Primitive Data Structures →

Language provides a way of defining our own data type.

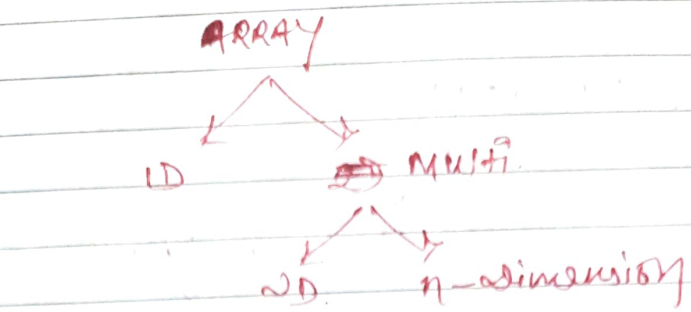
Non primitive data are data types defined by the user.

More Sophisticated.
Derived from primitive data structure.
Emphasize the structuring of group of homogenous or heterogenous data items.

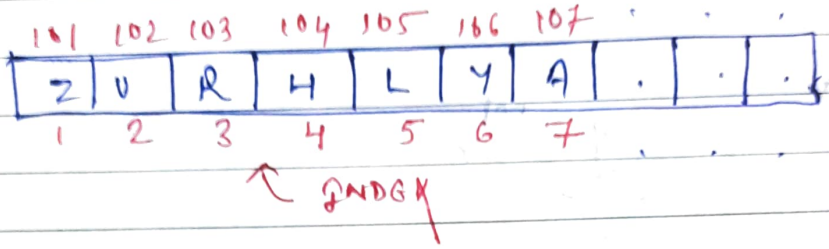
Date : / /
Page :

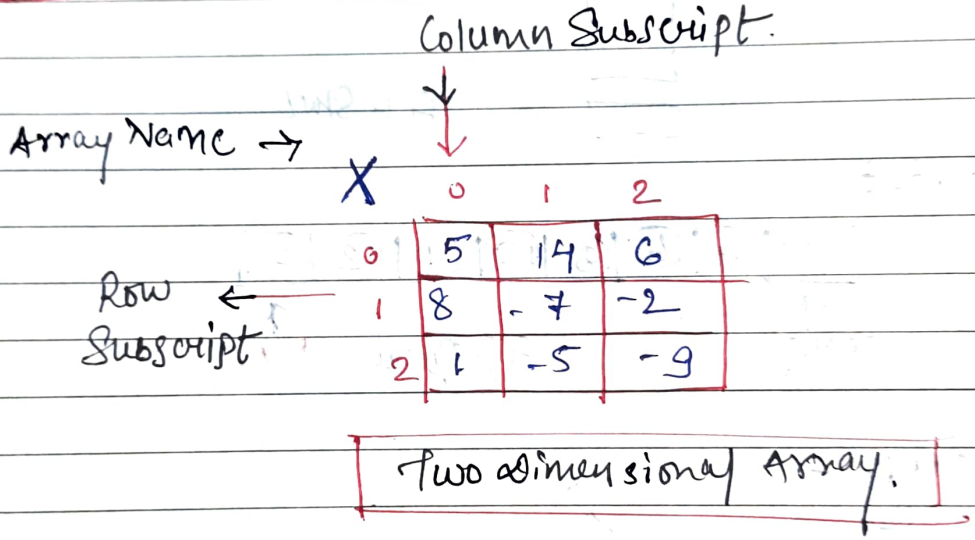
Comparision chart for linear and non-linear data structure

	LINEAR	NON LINEAR
	• data items are arranged in ordered manner • elements are attached adjacently	• Arranges data in sorted order, and there exist a relationship b/w the data elements
Traversing	• Data elements can be accessed in one time (single run)	• Traversing of data elements in one go isn't possible.
Implement	• Simpler Implementation	• Complex Implementation.
Level	• Single level involved.	• Multiple level involved
Examples	• Array, queue, stack, linked list etc.	• Tree and graph
Memory	• Ineffective memory utilization	• Effective memory utilization

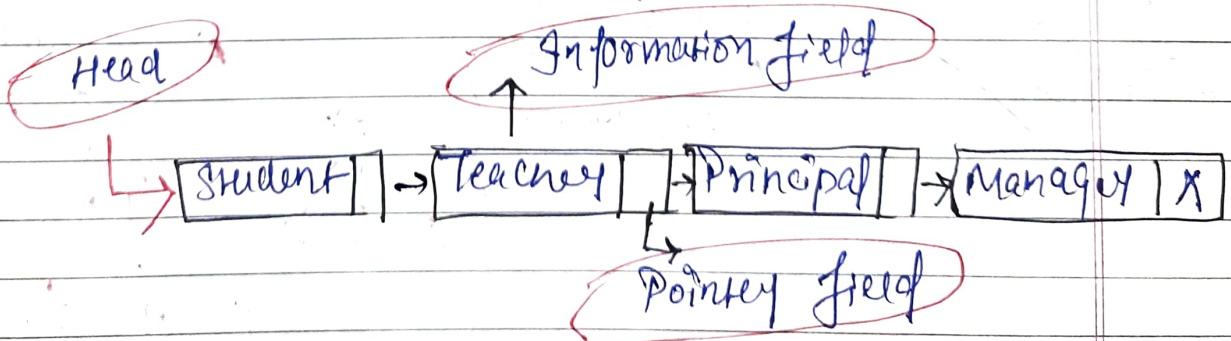


MEMORY LOCATION





LINEAR LINKED LIST –



Date : / /
Page :

Welcome.txt

primary name

dot or Seperator.

Secondary Name or Extension

STACK -

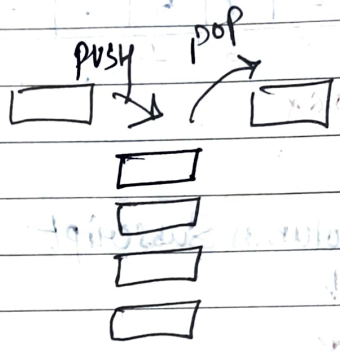


Fig. Stack

QUEUE

FRONT

Fig. Queue

REAR

GRAPH -

Real time Example.

Abstract Data type.

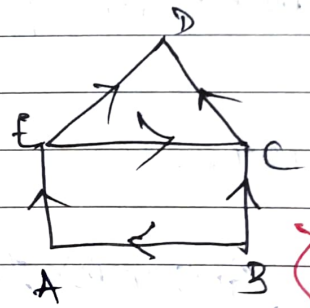


Fig. Graph.

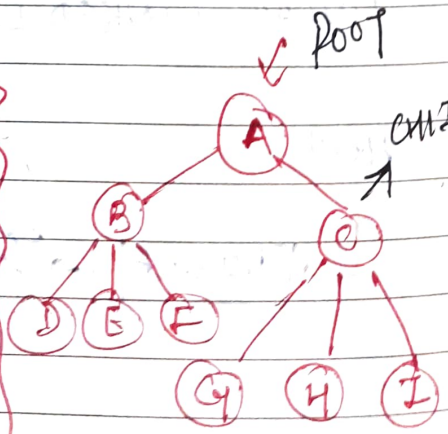


Fig. Tree

Date : / /

Page : _____

Self Notes.

Algorithm for binary search →.

```
while ( low ≤ high)
    mid ← (low/high)/2
        if searchkey = A[mid]
          return mid
        if searchkey > A[mid]
          low ← mid + 1
          else high ← mid - 1
        return NIL
```

10	14	19	26	31	42	44
0	1	2	3	4	5	6

Search 42

mid = $\frac{0+6}{2}$ = 3.

~~mid ≠ 3~~. searchkey ≠ a[mid]

searchkey > a[mid]

low = mid + 1.

low = 4.

mid = $\frac{4+6}{2}$ = 5

a[mid] = search (42) at 5.

Algorithm for Linear search –

Sequential.

Step1 START
step2 i=1
Step3 if i>n go to step 7
step4 if A[i]==x go to Step6
step5 i=i+1
Step6 Go to step 2
Step7 Print Element x found at i
Step8 Print Element not found
Step9. STOP

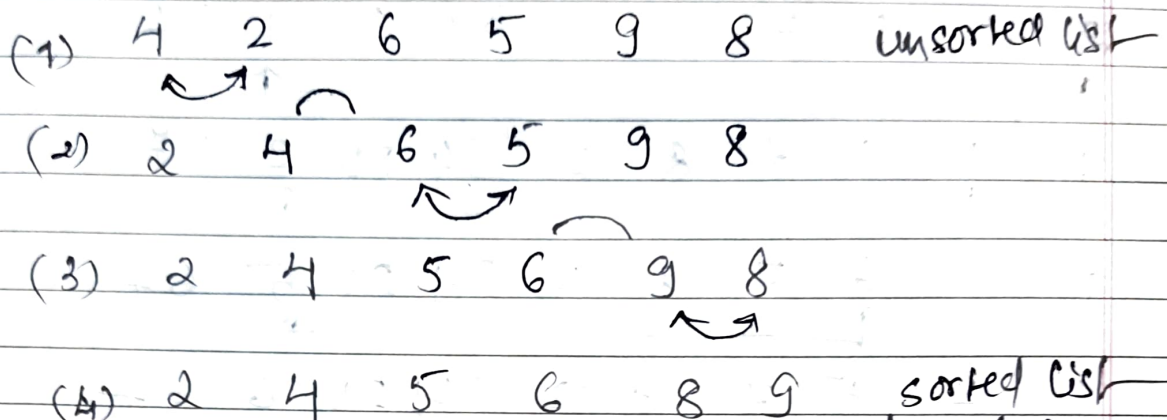
Step1: START
Step2: i=1
Step3: if i>n go to step 8
Step4: if A[i]==x go to step 7
Step5: i=i+1
Step6: go to step 3
Step7: Print Element (x) found at i
Step8: print Element not found.
Step9: STOP.

Date : / /
Page :

Sorting Algorithms.

Bubble Sort — pair of adjacent element is compared and swapped if they are not in order.

Worst case complexity $O(n^2)$



least (n)
avg case (n^2)
Worst case $O(n^2)$

Algorithm for Bubble Sort.

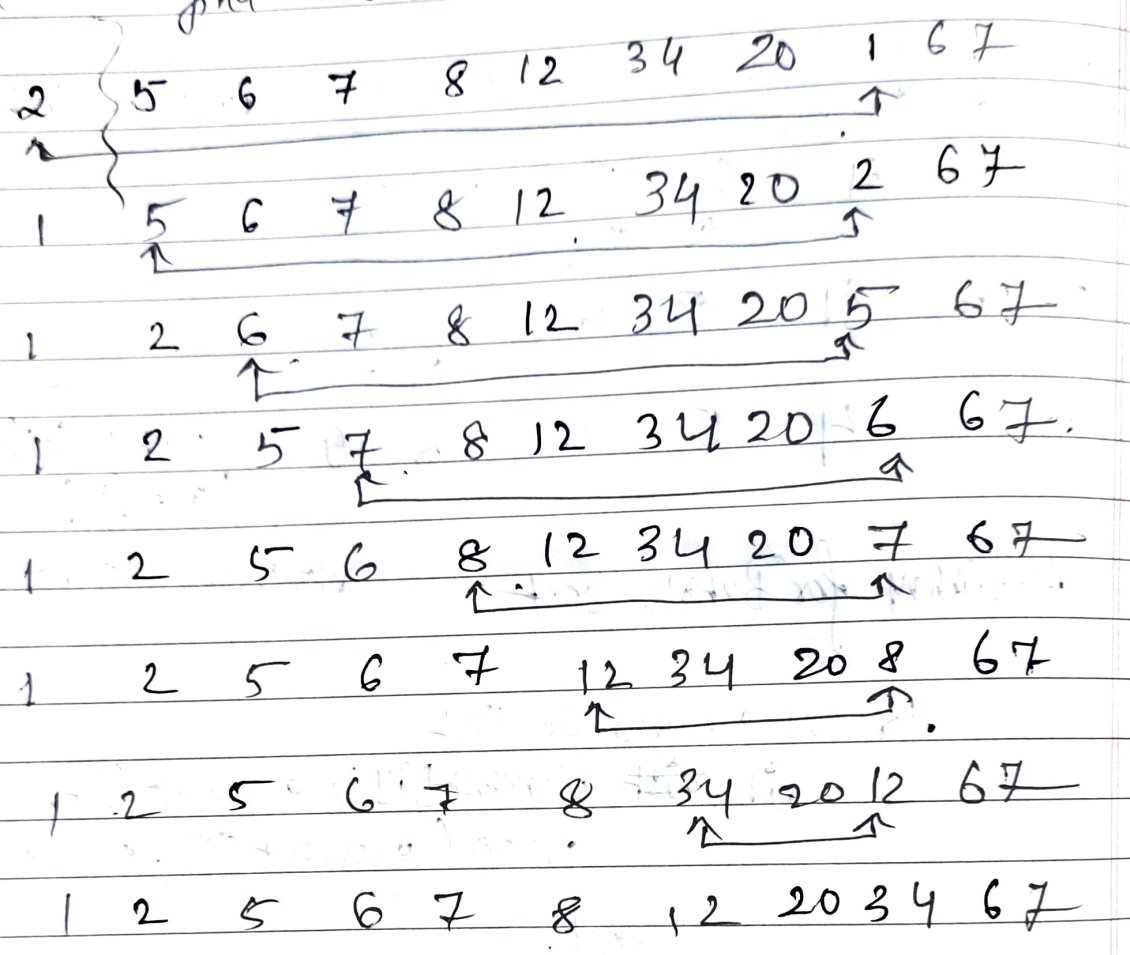
```
for all element in list
    if element[i] > element[i+1]
        swap ( element[i], element[i+1] ).
end.
```

Selection sort

Divide and conquer

Sorted part — left end — Unsorted part — Right end

find min and swap with left element



Algorithm →

Set min to location 0
search min element in list
swap the value with location min
increment min to point to next element
repeat until list is sorted.

Complexity —

Best case (n^2)
avg case (n^2)
worst case (n^2)

Insertion Sort.

Sorted Array is maintained.
not suitable for large data sets.
Avg / worst case $O(n^2)$
$(n-1)$ pass req to sort n elements.

In each pass we insert current element at appropiate place so that the elements in current range are in order.

each pass has k comparisions where k is pass no.

6 5 4 2 3

$n = 5$
pass $= n-1$
$= 4$.

pass 1: 5 6 4 2 3.

pass 2: 4 5 6 2 3.

pass 3: 2 4 5 6 3.

pass 4: 2 3 4 5 6

Algorithm for Insertion sort.

```
int i, key, j;
for (i=1; i<n; i++)
{ key = arr[i];
  j = i-1
  while (j>=0 && arr[j] > key)
  { arr[j+1] = arr[j];
    j = j-1;
  }
  arr[j+1] = key;
}
```

Assignment 2.1

SHRIDIP
19BCE10582

95, 80, 16, 54, 5, 70, 15, 25, 90, 5

Insertion Sort →

5, 95, 80, 16, 54, 5, 70, 15, 25, 90

5, 95, 80, 16, 54, 70, 15, 25, 90, 5

5, 5, 95, 80, 16, 54, 70, 15, 25, 90

5, 5, 15, 95, 80, 16, 54, 70, 25, 90

5, 5, 15, 16, 95, 80, 54, 70, 25, 90

5, 5, 15, 16, 25, 95, 80, 54, 70, 90

5, 5, 15, 16, 25, 54, 95, 80, 70, 90

5, 5, 15, 16, 25, 54, 70, 95, 80, 90

5, 5, 15, 16, 25, 54, 70, 80, 95, 90

5, 5, 15, 16, 25, 54, 70, 80, 90, 95

No. of Iterations = 10.

Selection Sort

95, 80, 16, 54, 5, 70, 15, 25, 90, 5

5, 80, 16, 54, 5, 70, 15, 25, 90, 95

5, 5, 16, 54, 80, 70, 15, 25, 90, 95

5, 5, 15, 54, 80, 70, 16, 25, 90, 95

5, 5, 15, 16, 80, 70, 54, 25, 90, 95

5, 5, 15, 16, 25, 70, 54, 80, 90, 95

Sorted. 5, 5, 15, 16, 25, 54, 70, 80, 90, 95

~~5, 5, 15, 16, 25, 5~~

No. of Iterations = 6.

Bubble Sort

95, 80, 16, 54, 5, 70, 15, 25, 90, 5

80, 95, 16, 54, 5, 70, 15, 25, 90, 5

80, 16, 95, 54, 5, 70, 15, 25, 90, 5

80, 16, 54, 95, 5, 70, 15, 25, 90, 5

80, 16, 54, 5, 95, 70, 15, 25, 90, 5

80, 16, 54, 5, 70, 95, 15, 25, 90, 5

80, 16, 54, 5, 70, 15, 95, 25, 90, 5

80, 16, 54, 5, 70, 15, 25, 95, 90, 5

80, 16, 54, 5, 70, 15, 25, 90, 95, 5

(1) 80, 16, 54, 5, 70, 15, 25, 90, 5, 95

16, 80, 54, 5, 70, 15, 25, 90, 5, 95

16, 54, 80, 5, 70, 15, 25, 90, 5, 95

16, 54, 5, 80, 70, 15, 25, 90, 5, 95

16, 54, 5, 70, 80, 15, 25, 90, 5, 95

16, 54, 5, 70, 15, 80, 25, 90, 5, 95

16, 54, 5, 70, 15, 25, 80, 90, 5, 95

Insert operation in BST. (Algorithm)

If root is NULL
then create root node
return

If root exists then
compare the data with node.data.

while until insertion position is located.

If data is greater than node.data
goto right subtree

else
goto left subtree.

end while

Insert data

end If.

Implementation of insert function:-

```
void insert (int data) {
    struct node * tempNode = (struct node *)
        malloc(size of (struct node
    struct node *current
    struct node *parent

    tempNode -> data = data;
    tempNode -> leftchild = NULL
    tempNode -> rightchild = NULL

// if tree is empty create root node
    if (root = NULL) {
        root = tempNode;
    }
    else {
        current = root
        parent = null

        while (1) {
            parent = current

            // go to left of tree
            if (data < parent -> data)
                { current = current -> left
                    child.
                // insert to the left
                if (current == NULL)
                { parent -> left child = temp
                    return; }
            }
```

Date : / /
Page :

// go to right of tree

```
else {
    current = current -> right child;
    // insert to the right.
    if (current == NULL)
    {
        parent -> right child = tempNode
        return;
    }
}
}
}
}
```

Search operation (BST) Algorithm

If root.data is equal to search.data
    return root
else
    while data not found.
        If data is greater than node.data
            go to right subtree
        else
            go to left subtree.
        If data found return
            return node
    endwhile
    return data not found.
end if

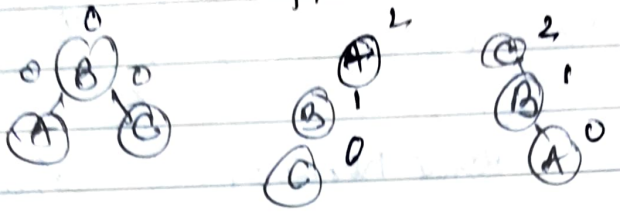
Implementation of algorithm

```
struct node* search (int data)

    struct node* current = root
    printf ("visiting elements");
    if (current != NULL)
        printf ("%d", current->data);

// go to left tree.
    if (current->data > data) {
        current = current -> leftchild
    }

// else go to right tree.
    else {
        current = current -> rightchild;
    }

// not found.
    if (current == NULL) {
        return NULL;
    }
    return current;
}
```

AVL TREES →

- Binary tree in a sorted manner
- Height balancing BST.
- checks height of left and right sub trees assures that difference is not more than 1.

this difference is called balance factor.



if diff in height of left & right sub trees is more than 1

the tree is balanced using some rotation techniques.

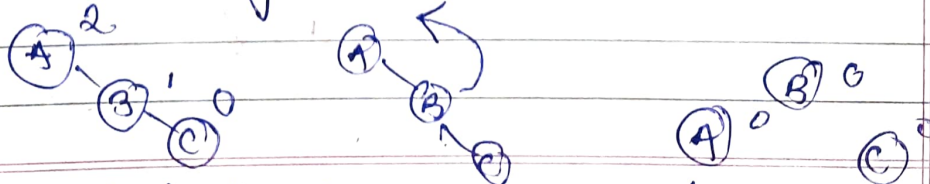
AVL ROTATIONS →

- left } single rotations
- Right }
- Left Right } double rotations
- Right left }

to have unbalance tree we need a tree of height 2

LEFT ROTATION —

If tree becomes unbalanced when node is inserted to right subtree then we perform a single left rotation.



node A becomes unbalanced as a node is inserted in the right subtree of its right subtree.

Page : / /
Date : / /

RIGHT ROTATION —

AVL tree may become unbalanced if a node is inserted in the left subtree, the tree needs right rotation.

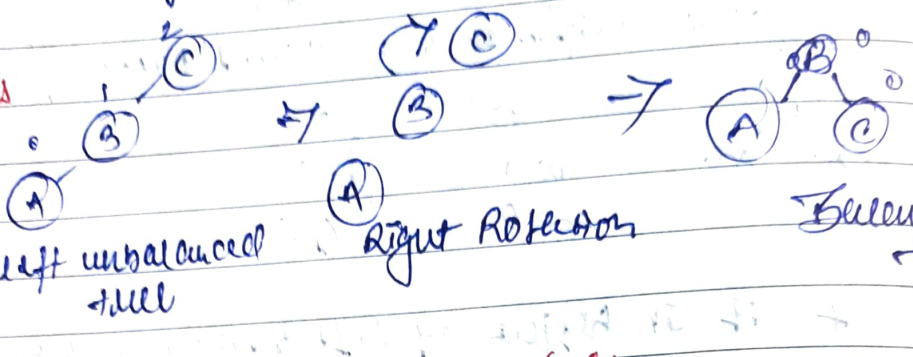
n^{n-2}

n = nodes

4^{4-2}

4^2

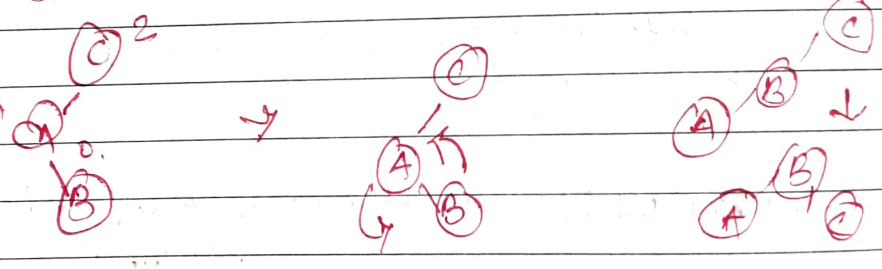
16



LEFT-RIGHT ROTATIONS → (C)

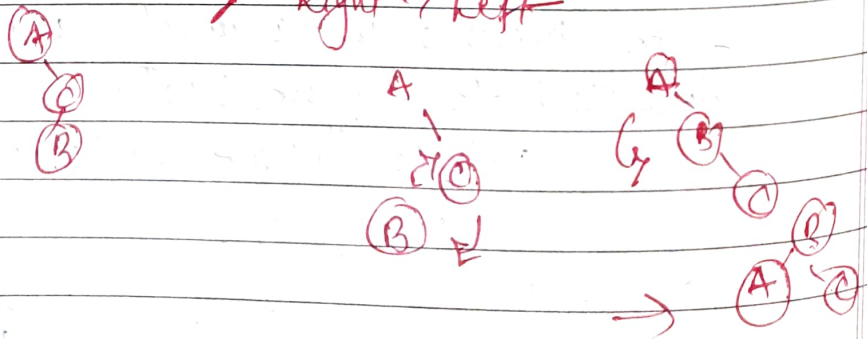
→ complex combination of left rotation followed by right rotation

State.



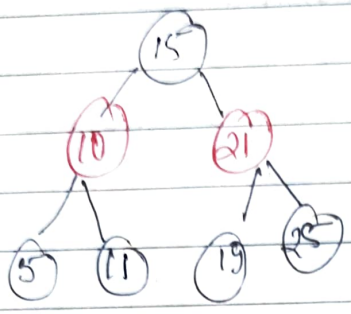
RIGHT LEFT ROTATION —

→ Right → Left



Red Black Tree.

- Root node is black.
- Red node can't have red children
- Nill nodes are black.
- No. of times black node appearing till nil is same for every path



Every BlackRed Tree a AVL Tree
But all AVL Tree are not BlackRed.

Stacks and queues →.

Stack –

linear data structure in which elements can be inserted and deleted from one side of list, called the top of stack.

follow LIFO (Last In First Out) Rule

Diagramic Representation of Stack →.

3	2 ← top.
2	9
1	5
0	0

operations
push()
pop()

we keep track of the last element present in the list with a pointer called top

Date : / /
Page :

Queue:

Linear data structure in which elements can be inserted from one side of list called rear.

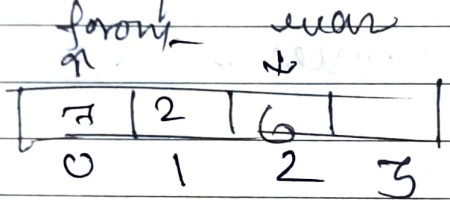
and the elements can only be deleted from other side called front.

follows FIFO (First In First Out)

operations of stack

enqueue()

dequeue()



Linked lists

linear data structure where each element is a separate object.

not stored at contigous location.

elements are stored using pointers.

Each node made up of 2 things.

data & reference to next node

Prep DSA TGC   Neso Academy

- Static Memory (Compile Time)
- Dynamic Memory (Execution Time) (Pointers)
  (allocated)

Structures and functions →

call by value.
call by reference

Linked Lists →

~~Maintain list in memory~~
array / linked list

types.
- Single (Navigation forward only →)
- Doubly (Nav for/back)
- Circular (Last element linked to 1st element)

Single linked list:
data. (actually data)
link (address of next node)

| data | link |

(Node)

(First Node access with pointer which is Head)

(Representation Array / linked list).

23	54	78	90
1000	1004	1008	1012

→ Sequential Representation

int (4) bytes.

struct node {
int data
struct node *link.
};

(multiple data).
single link)

creating node in C → Linked list

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

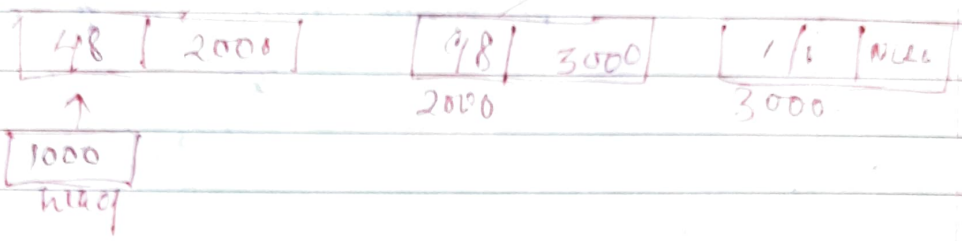
struct node {
int data;
struct node *head = NULL;
head = (struct node *) malloc (sizeof (struct node));

(allocate memory space)

Date : / /
Page :

```
head -> data = 45;
head -> link = NULL;
printf("%d", head -> data);
return 0;
}
```

Creating linked list in C.




```
struct node *current = malloc(sizeof(struct node));
current -> data = 98
current -> link = ~~3000~~ NULL
head -> link = current;
return 0;
```

Binary Tree.

left element < root

root > right element

AVL Trees →

① It is BST. (duplicate elements not allowed)

② height of subtree {1, 0, 1}

(height {left - right = {-1, 0, 1}

for each node.

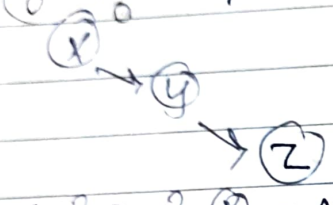
↳ difference is known as balance factor.

If height balance is 2, -2 we have to balance (AVL tree)

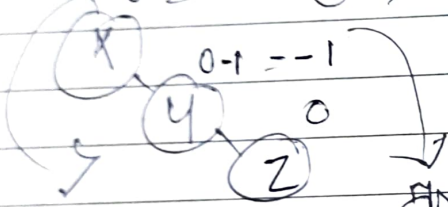
Example

x, y, z

(0 - 1) = -1.

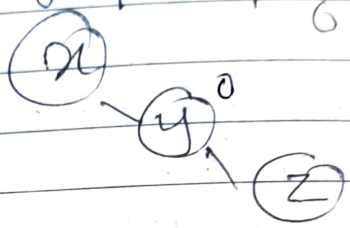


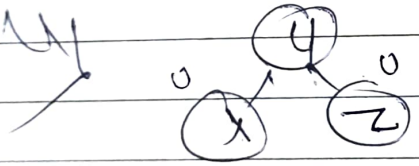
0 - 2 = -2 (X) → Not in set {0, -1, 1}.



Tree need to be balanced.

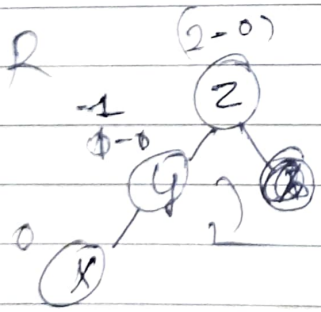
0 - 1 = -1 Okay!





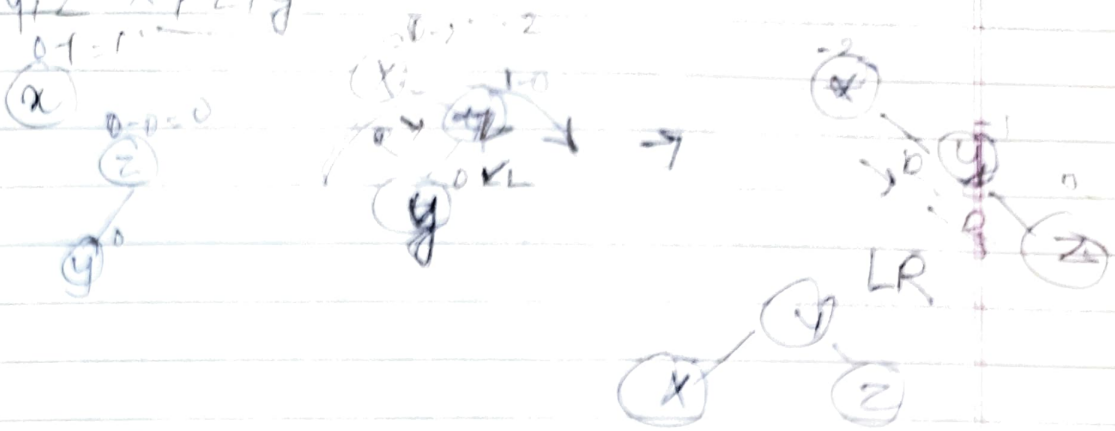
L Rotation

R Rotation



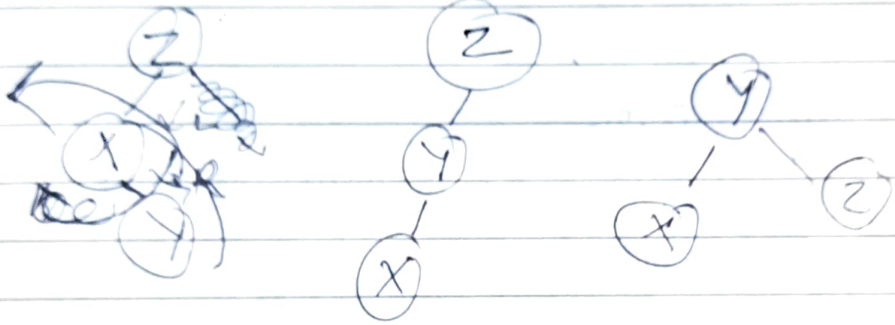
> Right < left

(3) ~~x, y, z~~ x, z, y



> RL
< LR

(4) z, x, y



left rotation.

R R (ins).

Right rotation

(ins).

Right left rotation

left Right rotation

Hashing →

- Searching technique:
- Time Complexity O(1) → constant
- Store & Retrieve

linear search worst case O(n)
binary " " O(log n)

- We use Hash functions / Tables / Address → store key/data
- COLLISION.

Ex key = 6, 26.
we can store 10 values in table.

3 method to calculate Hash Function
- division
- folding
- modular mult
- mid square

→ For division
m = size = 10 of Hash Table.

$$[h(k_i) = k_i \% m \quad m = 10.]$$

6 % 10
= 6

26 % m = 6
26 % 10 = 6.
already 6 (key) exists

when col. hash fun() for 2 keys having same index.
//

Date : / /
Page :

- To resolve collision we have 2 techniques

1. Open Hashing (closed addressing)
2. closed Hashing (Open Addressing)

** You can't Remove Collision you can minimize.

→ Linear, Quadratic, Double Hashing → Probing.

→ chaining

Example Question

A = 3, 2, 9, 6, 11, 13, 7, 12

size of HT [m = 10]

3+10=13

h(k) = 2k+3. # Hash function

~~division method~~ and closed addressing ~~to store values~~

(open hashing)

h(k) = 2k+3

Key	Location
3	(2×3+3)%10 = 9 [k%m]
2	(2×2+3)%10 = 7
9	(2×9+3)%10 = 1
6	(6×9+3)%10 = 7
11	(11×9+3)%10 = 2
13	(13×9+3)%10 = 2
7	(7×9+3)%10 = 6
12	(12×9+3)%10 = 1

0	
1	9
2	13
3	
4	
5	
6	7
7	2
8	
9	3

Key	location (i%10=m)
3	(2×3+3)%10 = 9
2	(2×2+3)%10 = 7
9	(2×9+3)%10 = 1
6	(2×6+3)%10 = 5
11	(2×11+3)%10 = 5
13	(2×13+3)%10 = 9
7	(2×7+3)%10 = 7
12	(2×12+3)%10 = 7

Prob 1

0		
1	9	
2		
3		
4		
5	6	→ [11]
6		
7	2	→ [7] → [12]
8		
9	3	→ [13]

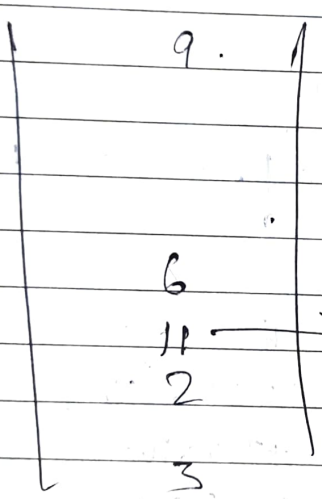
Chain Method

↓

Separate linked list will be created

Same { (Closed Hashing) / (Open addressing)
↳ Linear Probing (default)

Method



→ Linear Probing
- just below element if space free

Probing - Searching Kaise Karna

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3

free

	Location	Probes
3	9	1
2	7	1
9	1	1
6	5	1
11	5	2
13	9	2
7	7	2
12	7	5
	(4)	
Key	location	Probes

~~Linear Probing → Jumping~~

~~Order of Elements after insertion~~

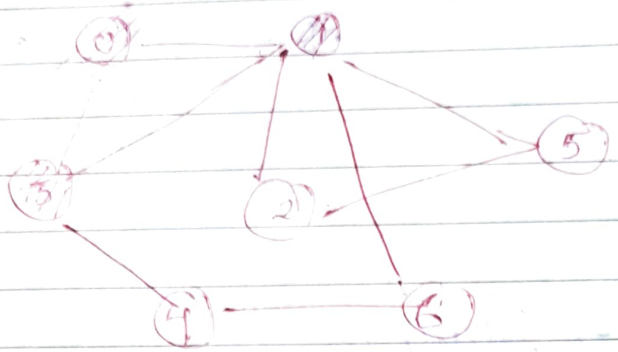
Order of elements after insertion → (13, 9, 12, —, —, 6, 11, 2, 7, 3)

~~Insert ki at first free location from (u+i) % m where~~
i varies range 0 to m-1
u → location

Graph Traversals →

(BFS) Breadth first search.

Queue. FIFO

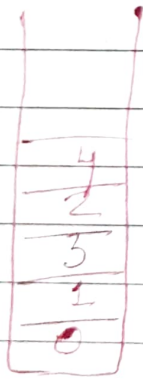


Queue | 0 | 3 | 1 | 5 | 6 | 2 |

Result 0 3 1 5 6.

DFS

Stack → LIFO



Result. 0, 1, 3, 2

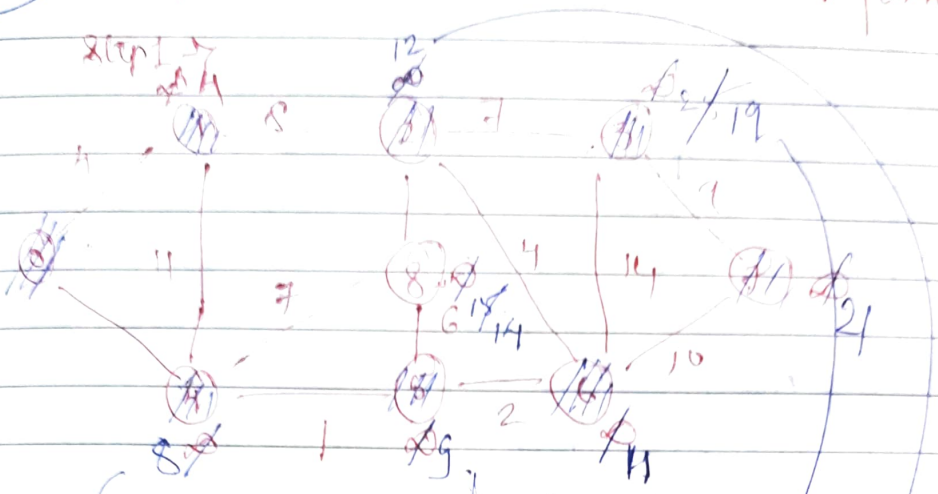
$(d(u) + c(u,v) < d[v])$

Dijkstra's algorithm

$d[v] = d[u] + c(u,v)$

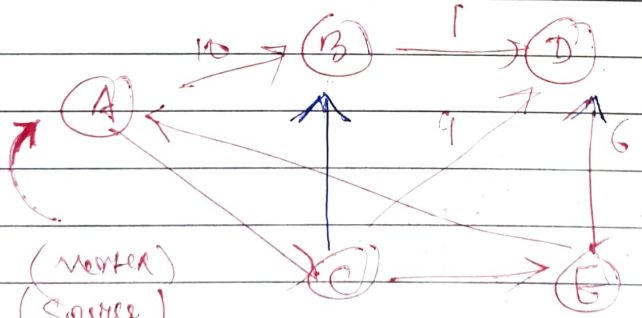
formula

(single source)



Selected vertex → 0, 1, 4, 5, 6, 2, 8, 3, 7

once vertex is visited dist won't be update.



Selected (vertex)	A (Source)	B	C	D	E	Path
A	0	∞	∞	∞	∞	
C		10	5	∞	∞	
E		8	14	7		
B		8	13			
		9	← Pointer.			

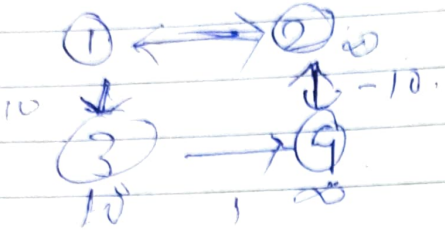
pointers store back ↑ point out do changes

Path A to D. DBCA = 9. distance

Date : / /
Page :

Limitation:

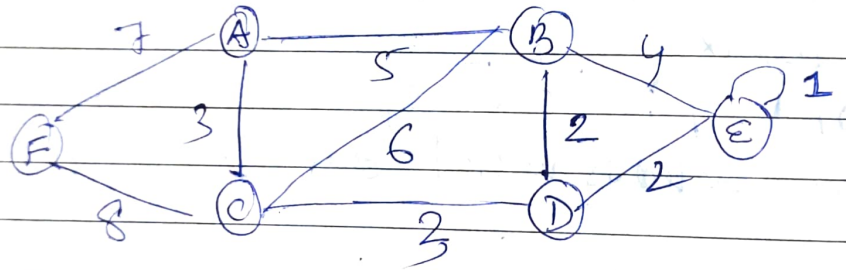
may / may not work with -ve edge weight



Surely Not work on cyclic graph having negative weight

Incorrect result

Kruskal's Algorithm



Remove

- SELF LOOPS AND
- PARALLEL EDGE (more wgt) delete keep low (wt)

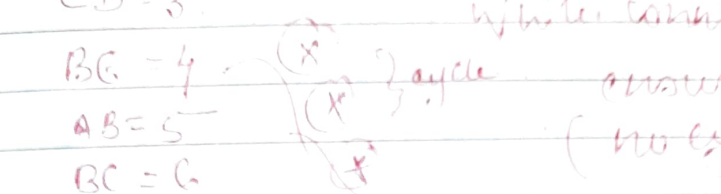
Next Step

Arrange all edge weight in incr order.

(2nd Step).

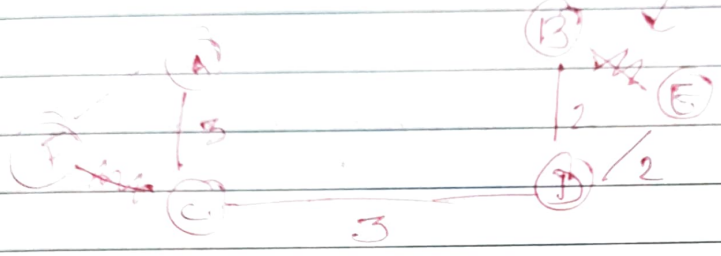
BD = 2 ✓
DE = 2 ✓
AC = 3 ✓
CD = 3 ✓
BE = 4 ✓
AB = 5 ✓
BC = 6 ✓
AF = 7 ✓
FC = 8 ✓

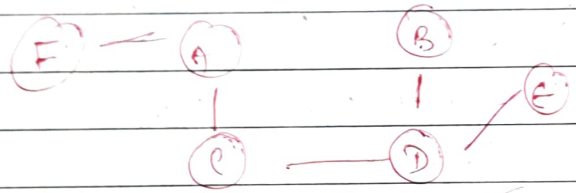
3rd Step.
Choose edges with the low edge with while connecting all vertices (no cycles)



cycle

can't connect since it will give cycle





~~MST doesn't contain any cycle~~

11/9/17.

Minimum Spanning tree

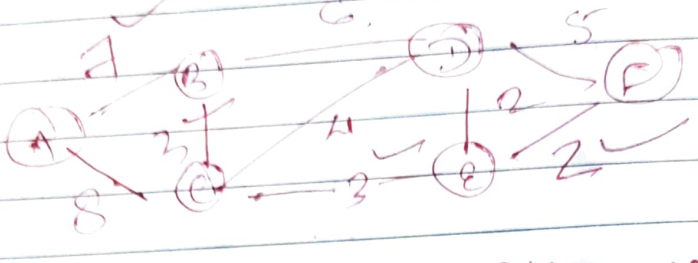
vertices.

(n-1 edges.

Prims Algorithm

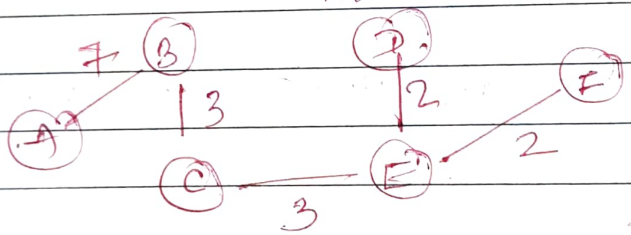
Step 1 → Remove self loops
→ parallel edges.

remove → (keep edge which has min cost/wt)



Step 2 → Choose any arbitrary vertex

A) check outgoing edges from vertex
choose min edge (wt) first
incident



(MST) → Same no. of vertices
(n-1) of edge

G(V, E).
G' is (V', E')

V' = V

E' ⊆ E
↓
subset