V V COLLEGE OF ENGINEERING

VVNAGAR

TISAYANVILAI

COURSE FILE REGULATION 2021

Name of the staff : T. KARTHIJA

Designation:AP/CSE

Semester/Year: 03 / II

Branch:CSE

Subject Code/ Name : CS3301 - DATA STRUCTURES

Unit I

Date of submission	Signature of staff Incharge	Verification by HOD	Verification by Principal
21.9.22	[illegible]	[illegible]	[illegible] 21/9/22

Unit II

Date of submission	Signature of staff Incharge	Verification by HOD	Verification by Principal
21.9.22	[illegible]	[illegible]	[illegible] 21/9/22

Unit III

Date of submission	Signature of staff Incharge	Verification by HOD	Verification by Principal
31.10.22	[illegible]	[illegible] 31/10/22	[illegible] 01/11/22

Unit IV

Date of submission	Signature of staff Incharge	Verification by HOD	Verification by Principal
31.10.22	[illegible]	[illegible] 31/10/22	[illegible] 01/11/22

UnitV

Date of submission	Signature of staff Incharge	Verification by HOD	Verification by Principal
31/10/22	[illegible]	[illegible] 1/11/22	[illegible] 01/11/22

V V COLLEGE OF ENGINEERING

(Approved By AICTE, New Delhi and Affiliated to Anna University, Chennai)
V V NAGAR, ARASOOR - 628656

Department of Computer Science and Engineering

INSTITUTION VISION:

Emerge as a premier technical institution of global standards, producing enterprising, knowledgeable engineers and entrepreneurs.

INSTITUTION MISSION:

- Impart quality and contemporary technical education for rural students.
- Have the state of the art infrastructure and equipment for quality learning.
- Enable knowledge with ethics, values and social responsibilities.
- Inculcate innovation and creativity among students for contribution to society.

DEPARTMENT VISION:

Produce competent and intellectual Computer Science graduates by empowering them to compete globally towards professional excellence.

DEPARTMENT MISSION:

- Provide resources, environment and continuing learning processes for better exposure in latest and contemporary technologies in Computer Science and Engineering.
- Encourage creativity and innovation and the development of self-employment through knowledge and skills, for contribution to society
- Provide quality education in Computer Science and Engineering by creating a platform to enable coding, problem solving, design, development, testing and implementation of solutions for the benefit of society.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

The graduates of Computer Science and Engineering shall possess

- Have a successful career in computer software and hardware allied industries or shall pursue higher education or research or emerge as entrepreneurs.
- Have expertise in the areas of design and development of software solutions, real-time applications, web based solutions, etc.
- Contribute towards technological development through academic research and industrial practices and adapt to evolving technologies through life-long learning
- Practice their profession with good communication, leadership, ethics and social responsibility.

PROGRAM SPECIFIC OUTCOMES (PSOS)

PSO1	To involve students in development of projects using emerging Information and Communication technologies.
PSO2	To get succeed in competitive examinations for successful higher studies and employment.

V V COLLEGE OF ENGINEERING
(Approved By AICTE, New Delhi and Affiliated to Anna University, Chennai)
V V NAGAR, ARASOOR - 628656

Department of Computer Science and Engineering

PROGRAM OUTCOMES (POs)

PO1	Engineering knowledge	Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/ development of solutions	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	Conduct investigations of complex problems	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	Modern tool usage	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	The engineer and society	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Individual and team work	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communication	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

CS3301 DATA STRUCTURES

L	T	P	C
3	0	0	3

UNIT I LISTS 9

Abstract Data Types (ADTs) – List ADT – Array-based implementation – Linked list implementation –Singly linked lists – Circularly linked lists – Doubly-linked lists – Applications of lists – Polynomial ADT – Radix Sort – Multilists.

UNIT II STACKS AND QUEUES 9

Stack ADT – Operations – Applications – Balancing Symbols – Evaluating arithmetic expressions- Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Circular Queue – DeQueue – Applications of Queues.

UNIT III TREES 9

Tree ADT – Tree Traversals - Binary Tree ADT – Expression trees – Binary Search Tree ADT – AVL Trees – Priority Queue (Heaps) – Binary Heap.

UNIT IV MULTIWAY SEARCH TREES AND GRAPHS 9

B-Tree – B+ Tree – Graph Definition – Representation of Graphs – Types of Graph – Breadth-first traversal — Depth-first traversal — Bi-connectivity – Euler circuits – Topological Sort – Dijkstra's algorithm – Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES 9

Searching – Linear Search – Binary Search. Sorting – Bubble sort – Selection sort – Insertion sort – Shell sort –. Merge Sort – Hashing – Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

TEXT BOOKS

1. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd Edition, Pearson Education, 2005.
2. Kamthane, Introduction to Data Structures in C, 1st Edition, Pearson Education, 2007

REFERENCES

1. Langsam, Augenstein and Tanenbaum, Data Structures Using C and C++, 2nd Edition, Pearson Education, 2015.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein, Introduction to Algorithms", Fourth Edition, Mcgraw Hill/ MIT Press, 2022.
3. Alfred V. Aho, Jeffrey D. Ullman,John E. Hopcroft ,Data Structures and Algorithms, 1st edition, Pearson, 2002.
4. Kruse, Data Structures and Program Design in C, 2nd Edition, Pearson Education, 2006.

UNIT - I
LISTS

ABSTRACT DATA TYPES (ADTS)

Basic rule in programming is that no routine should exceed a page. To accomplish this, pgm is broken down into modules. Each module is a logical unit & does a specific job.

Adv.

- easy to debug.
- easy for several people to work on modular pgm simultaneously
- easy to change.

An ADT is a set of operations. ADT are mathematical abstractions. Objects such as lists, sets, graphs along with their operations can be viewed as ADTs, just as integer, reals, Booleans.

LIST ADT.

- List is of the form $A_1, A_2, A_3 \ldots A_N$.
- whose size is N.
- size of empty list is 0.
- A_{i+1} follows A_i and A_{i-1} preceeds A_i

Operations on List :

- PrintList
- MakeEmpty
- Find
- Insert
- Delete
- FindKth

Array implementation of List.

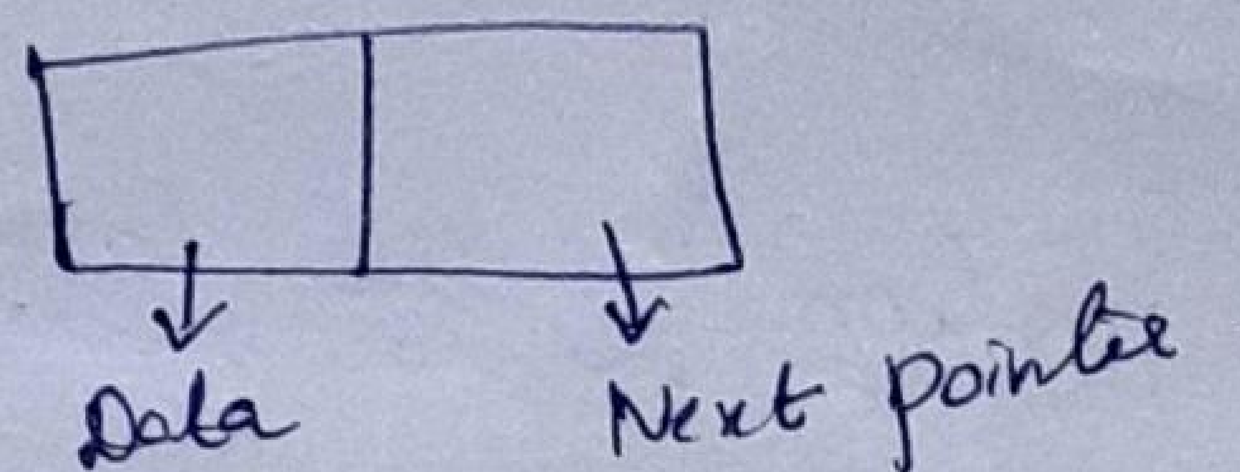
- Size of the list is required.
- If dynamically allocated, then an estimate of the maximum size of the list is required.

- Insertion in 0th position need to create a new space at the first and then insert the new element.
- Deleting the 0th position need to delete the first element and move the rest of the element one step.
- So both these operations will take more time and expensive.

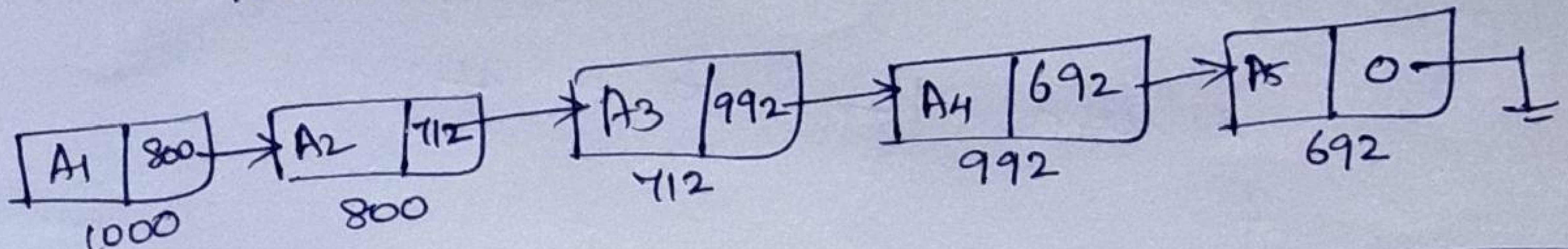
Linked Lists.

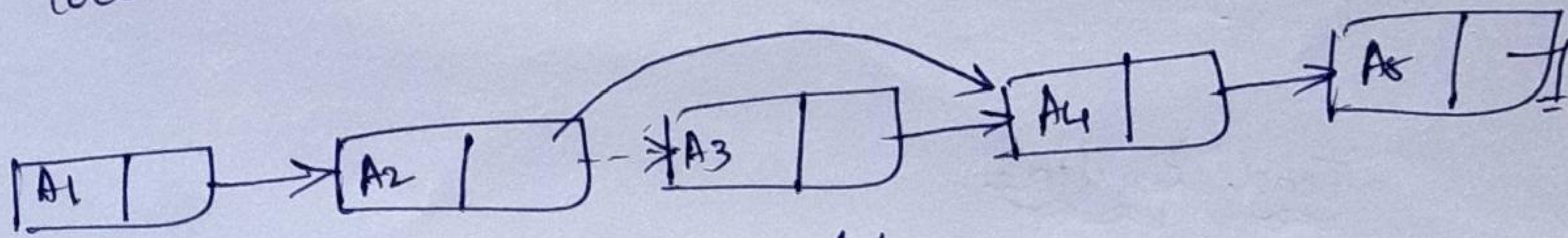
- To avoid linear cost of insertion & deletion linked list is used.
- List is not stored continuously.

- Consists of series of structures, not in adjacent mry.

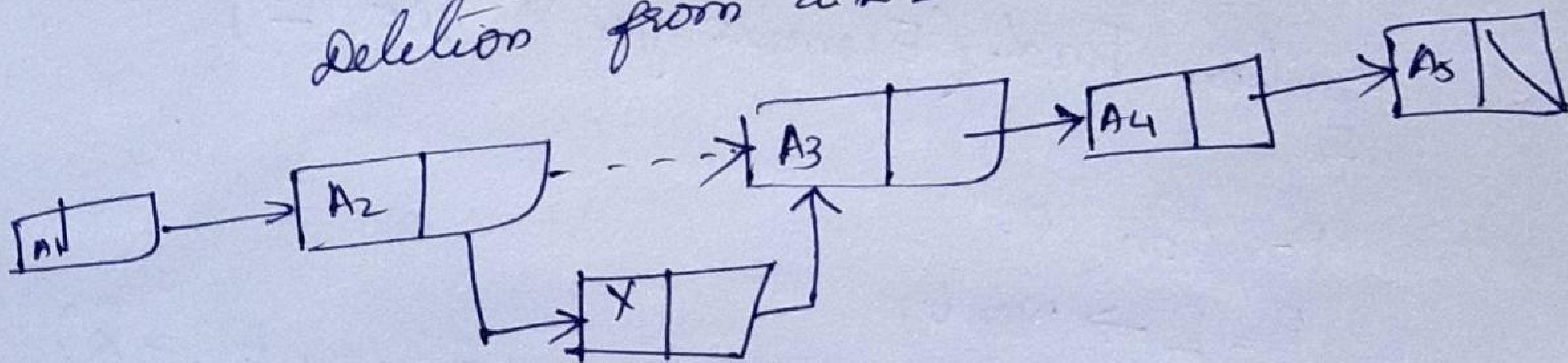


- Last cells next pointer points to null.





Deletion from a LL.



Insertion into a LL

Type declaration of LL

```
struct Node
{
 ElementType Element;
 Position Next;
};
```

Check LL is empty.

```
int IsEmpty ( List L)
{
return L->Next == NULL;)
}
```

check current position is the last in LL

```
int IsLast (Position P, List L)
{
return P->Next == NULL;
}
```

Find routine.

```
Position Find (ElementType X, List L)
{
 Position P;
 P = L->Next;
 while (P != NULL && P->Element != X)
   P = P->Next;
   return P;
 }
```

Deletion.

```
Void Delete (ElementType x, List L)
{
  Position P, TmpCell;
  P = FindPrevious (X, L);
  if (! IsLast (P, L))
  {
    TmpCell = P->Next;
    P->Next = TmpCell->Next;
    free (TmpCell);
  }
}
```

FindPrevious.

```
Position FindPrevious ( ElementType x, List L)
{
  Position P;
  P = L;
  while ( P->Next != NULL && P->Next->Element != X)
      P = P->Next;
  return P;
}
```

Insertion

```
Void Insert(ElementType x, List L, Position P)
{
 Position TmpCell;
 TmpCell = malloc (Sizeof (Struct Node));
  if (TmpCell == NULL)
     FatalError ("out of space!");

  TmpCell -> Element = x;
  Tmpcell -> Next = P -> Next;
   P -> Next = TmpCell;

 }
```

To Delete a list.

```
Void DeleteList (List L)
 {
  Position P, Tmp;
   P = L -> Next;   /header/
   L -> Next = NULL
   while (P != NULL)
    {
     Tmp = P -> Next;
      free (P);
      P = Tmp;
     }
  }
```

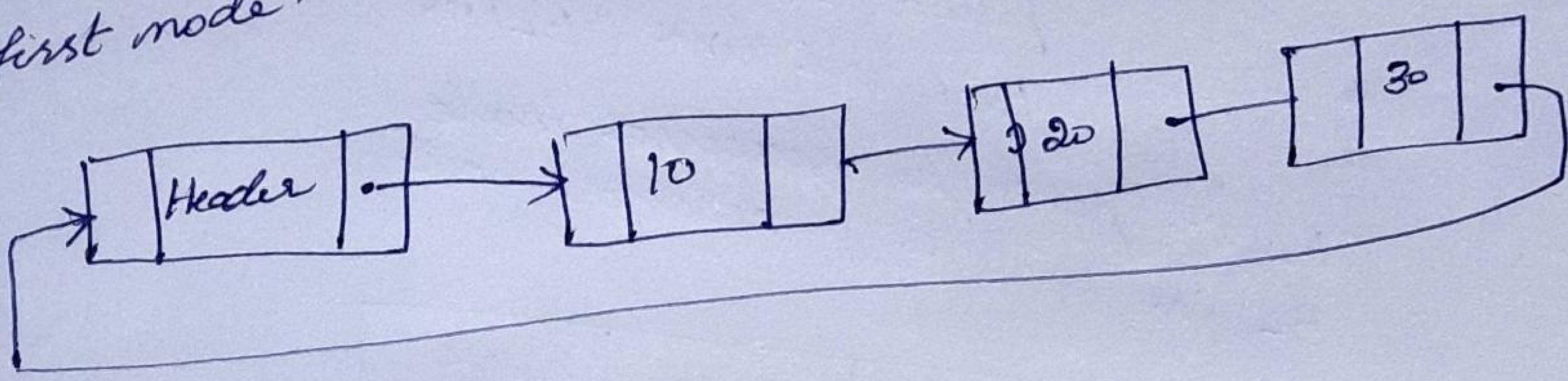
Scanned by TapScanner

Circular Linked List.

In circular linked list the pointer of the last node points to the first node. Circular LL can be implemented as Singly LL and Doubly LL with or without headers.

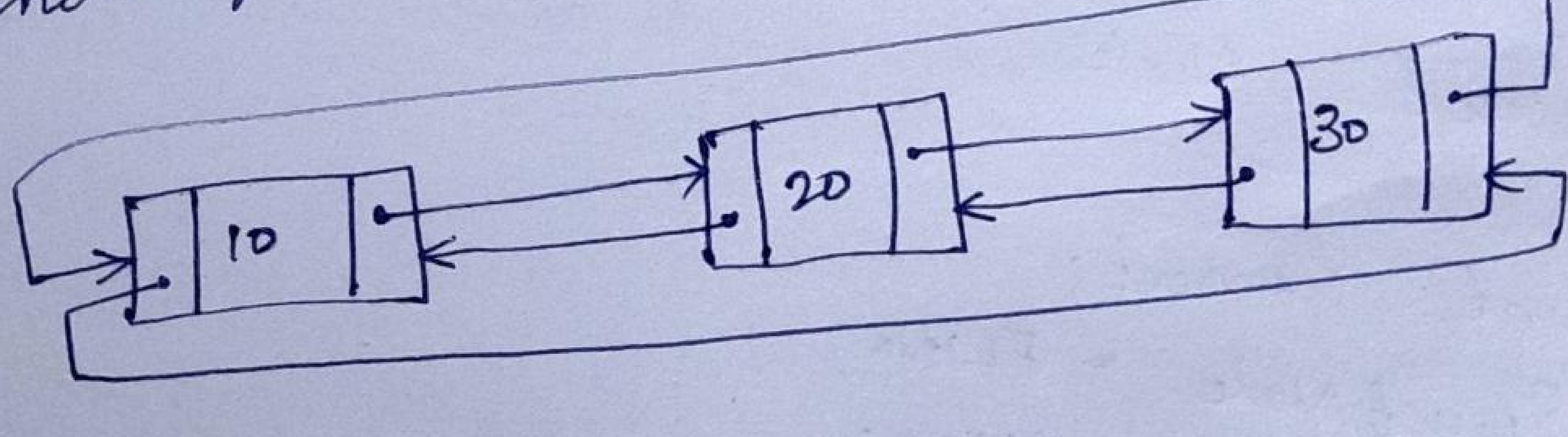
Singly Linked Circular List.

A singly Linked circular list is a linked list in which the last node of the list points to the first node.



Doubly Linked Circular list.

- is a doubly LL in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.

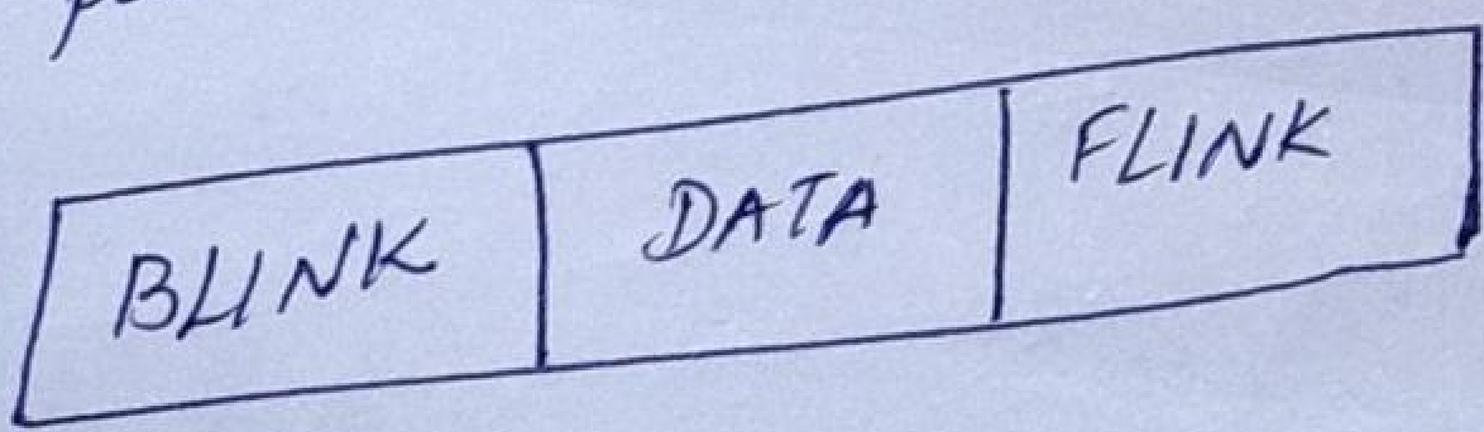


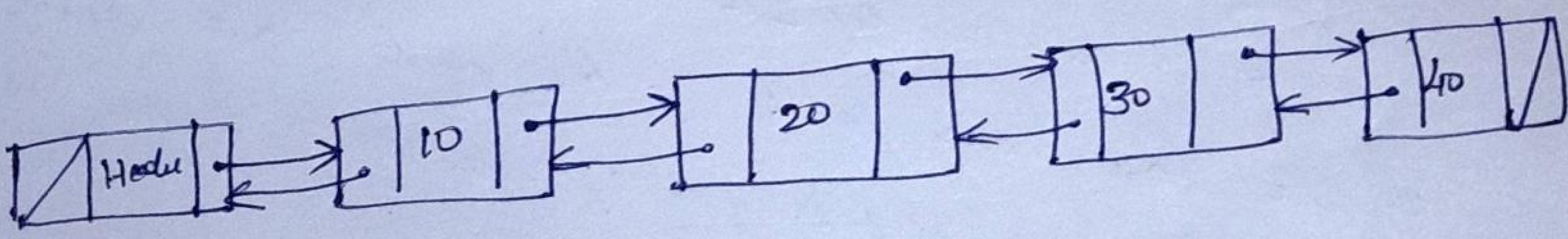
Advantages of circular LL.
- traverse the list starting at any point.
- quick access to the first and last records.
- Circularly doubly LL allows to traverse the list in either direction.

DOUBLY LINKED LIST.

- Each node has 3 fields
  - data field
  - forward link (FLINK)
  - Backward link (BLINK)

FLINK points to successor node.
BLINK points to predecessor node.



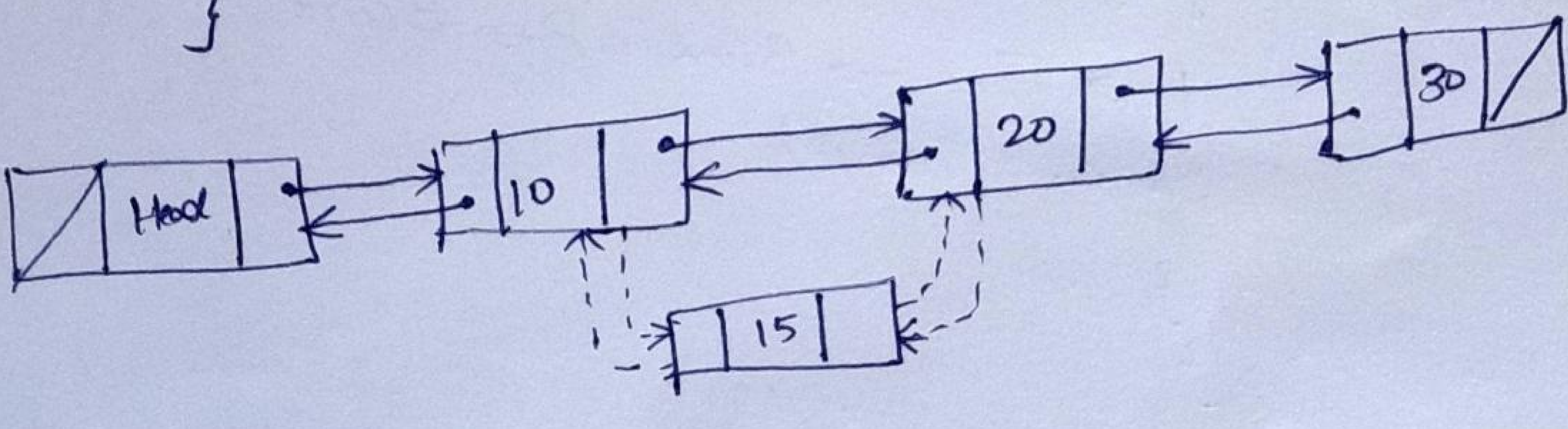


Structure declaration.

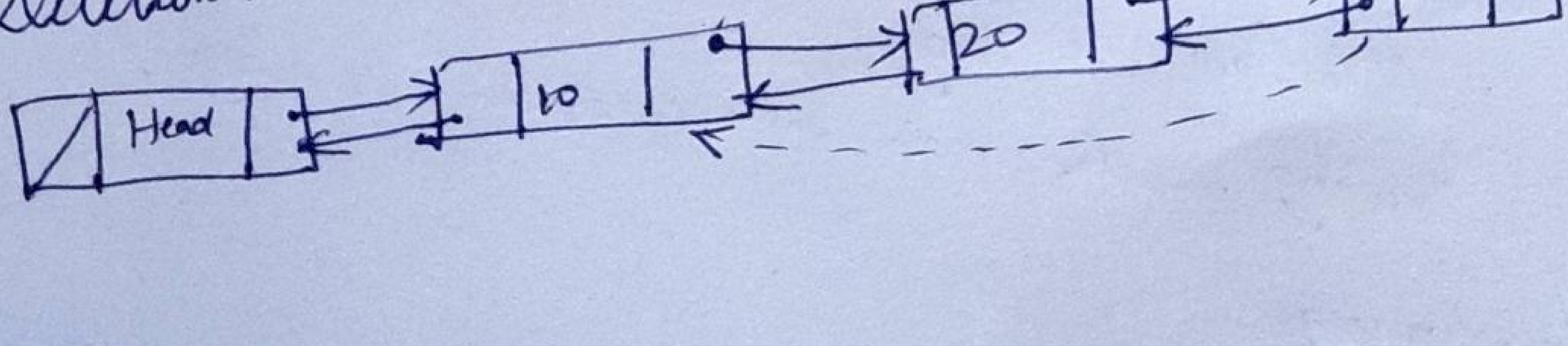
```
Struct Node
{
 int Element;
 Struct Node * FLINK;
 Struct Node * BLINK;
};
```

Insertion:

```
Void Insert (int X, List L, position P)
{
Struct Node * Newnode;
Newnode = malloc (Sizeof (Struct Node));
 if (Newnode != NULL)
  {
   Newnode → Element = X;
   Newnode → Flink = P → Flink;
   P → Flink → Blink = Newnode;
   P → Flink = Newnode;
   Newnode → Blink = P;

  }
 }
```



Deletion:



Delete routine:

```
Void Delete ( int X, List L)
{
position P;
P= Find ( X,L);
if (IsLast (P,L))
 {
  Temp = P;
  P→ Blink → Flink = NULL
  free (Temp);
  }
 else
  {
   Temp = P;
   P→ Blink → Flink = P→ Flink;
   P→ Flink → Blink = P→ Blink;
   free (.Temp);
   }
  }
```

Advantage:

- Deletion is easy.
- Finding predecessor & successor is easy.

Disadvantage:

- More mry is needed since it has 2 pointers.

APPLICATIONS OF LINKED LIST

1. Polynomial ADT
2. Radix Sort
3. Multilist.

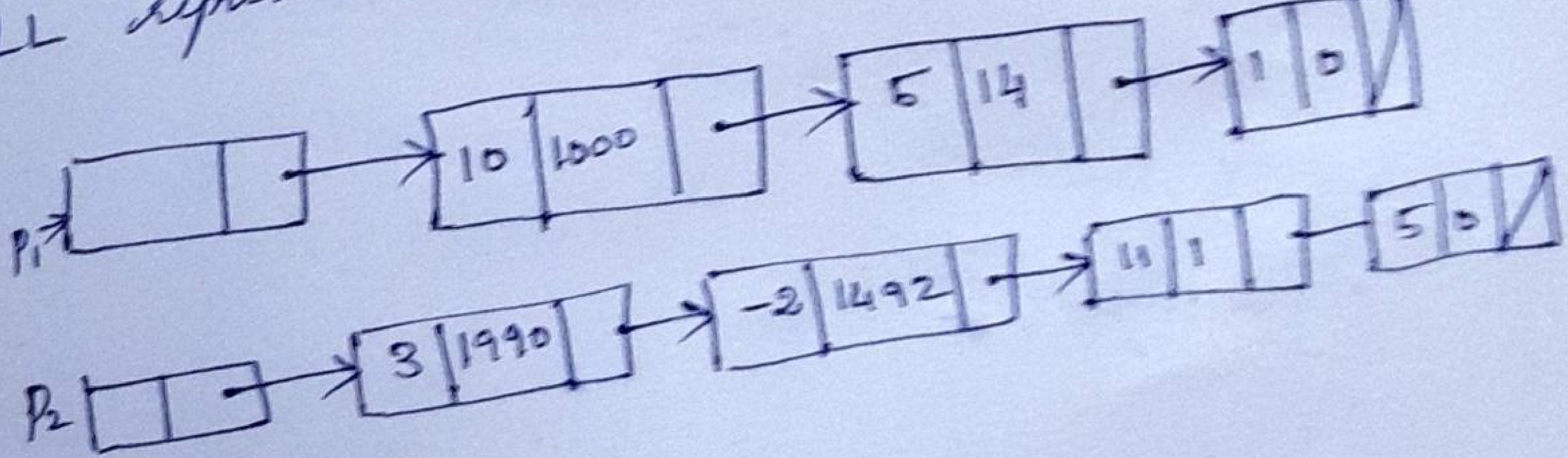
POLYNOMIAL ADT

- We can perform the polynomial manipulations such as addition, subtraction, differentiation etc.
- Polynomial can be expressed in a ADT structure.
- If all the exponent values are present then it can be represented in an array.
- If any exponent value is missing, it is easy to represent in LL.

Eg. $P_1(x) = 10x^{1000} + 5x^{14} + 1$

$P_2(x) = 3x^{1990} - 2x^{1492} + 11x + 5$

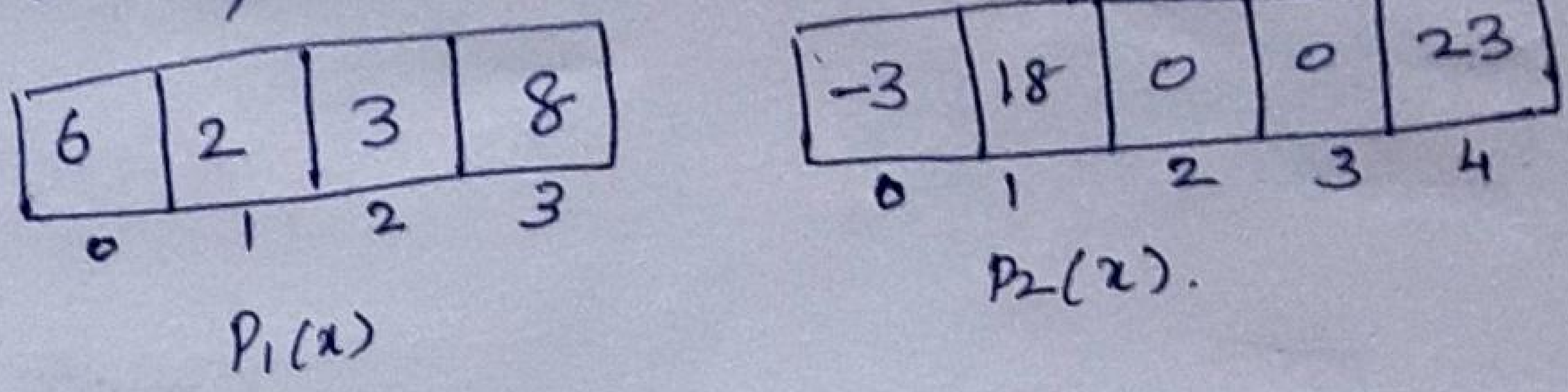
LL representation.



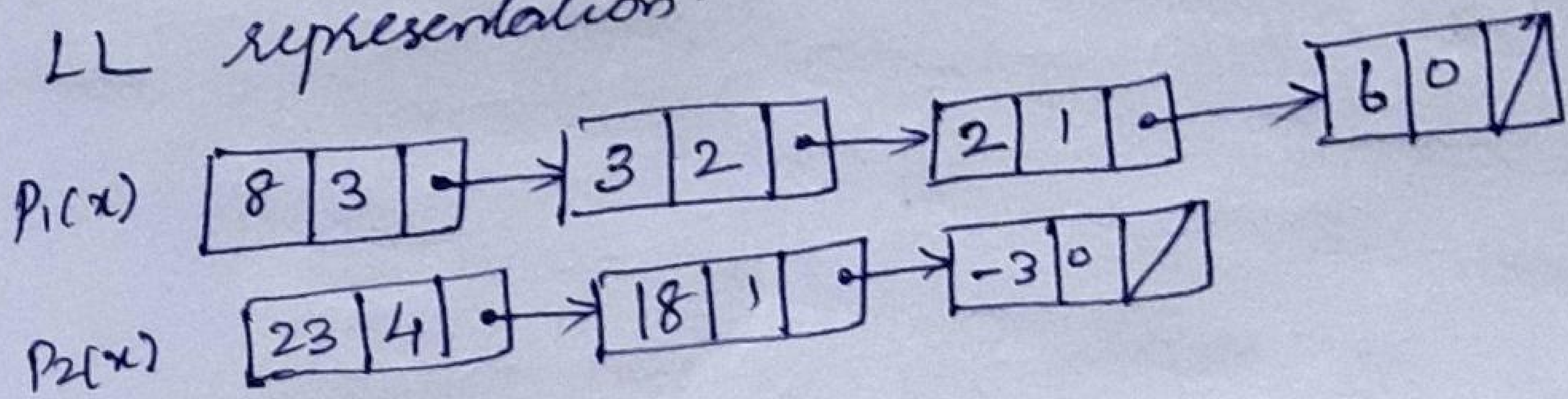
$P_1(x) = 8x^3 + 3x^2 + 2x + 6$

$P_2(x) = 23x^4 + 18x - 3$

can be represented in an array as,



LL representation.



Adv. of array implementation:

- Easy to represent dense polynomial

Type declaration of array implementation of P-ADT.

```
typedef Struct
{
int CoeffArray [MaxDegree + 1];
int HighPower;
}* polynomial;
```

Initialize a polynomial to zero.

```
Void ZeroPolynomial( Polynomial Poly)
{
 int i;
 for (i=0; i<= MaxDegree; i++)
     poly -> CoeffArray [i]=0;
   Poly -> HighPower=0;
   }
```

Add two polynomial:

```
Void AddPolynomial (Polynomial poly1, polynomial poly2,
                                  Polynomial polySum)

{
 int i;
 ZeroPolynomial (PolySum);
 PolySum -> HighPower = Max (Poly1 -> HighPower,
                              poly2-> HighPower);

  for (i= PolySum -> HighPower; i>=0; i--)
     polySum -> CoeffArray [i]= Poly1 -> CoeffArray [i] +
                                  Poly2 -> CoeffArray [i];

    }
```

Multiply 2 Polynomial.

```
Void MultPolynomial ( Polynomial Poly1, Polynomial Poly2,
                      Polynomial PolyProd )

{
  int i, j;
  ZeroPolynomial ( PolyProd );
  PolyProd -> HighPower = Poly1 -> HighPower + Poly2 -> HighPower;
  if ( PolyProd -> HighPower > MaxDegree )
      Error ( "Exceeded array Size" );
  else
   for ( i=0 ; i <= Poly1 -> HighPower ; i++ )
     for ( j=0; j <= poly2 -> HighPower ; j++ )
        polyProd -> CoeffArray [i+j] +=
                 Poly1 -> CoeffArray [i] *
                 poly2 -> CoeffArray [j];

}
```

Declaration of LL (Poly).

```
typedef Struct Node * ptrToNode;
Struct Node
{
  int Coefficient;
  int Exponent;
  PtrToNode Next;
}; typedef PtrToNode Polynomial;
```

Scanned by TapScanner

RADIX SORT.

Radix Sort is also known as card sort.

eg.

64, 8, 216, 512, 27, 729, 0, 1, 343, 125

Step ①. Bucket sorts by the least significant digit

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

The buckets are again sorted by the next least significant digit.

		729				64			
8	216	27		343					
1	512	125							
0									
0	1	2	3	4	5	6	7	8	9

This is again sorted by the next least significant digit.

64									
27									
8									
1							729		
0	125	216	343		512				
0	1	2	3	4	5	6	7	8	9

0 1 8 27 64 125 216 343 512 729

This is the sorted list.

The running time for the algorithm is

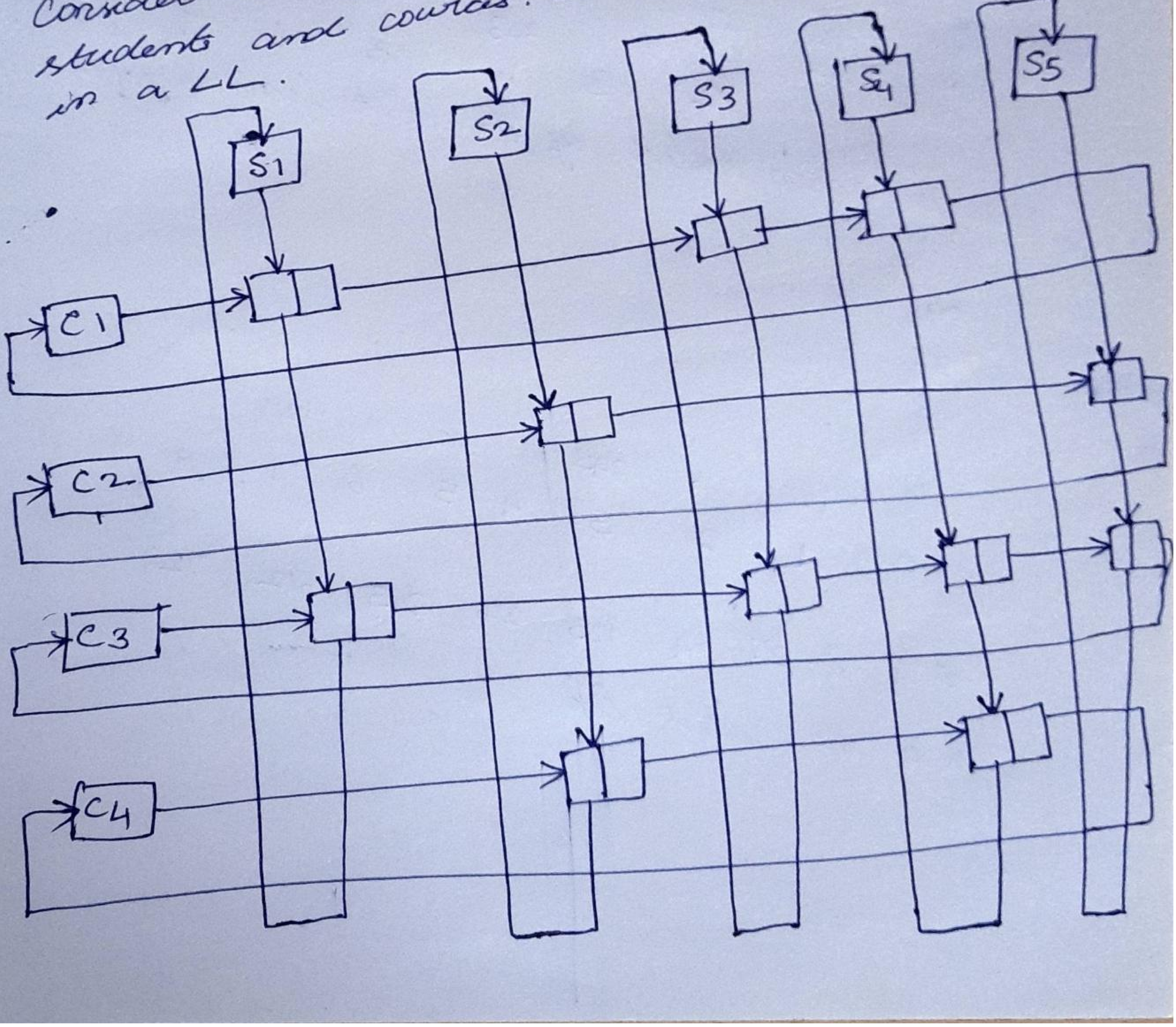
$O(P(N+B))$ where P → no. of passes.

N → no. of element

B → no. of bucket.

MULTI LISTS

A multilist is a combination of more than one list. All the lists are combined to one list. All the lists use a header & are circular. Consider an eg. of University with large no of students and courses. Each class can be represented in a LL.



To list all the students in C3. start at C3 and traverse its list. The first cell belongs to S1, then S3 & then S4. The traversal is carried out until the header is reached.

Circular list reduces the usage of space but at the expense of time.

Cursor base LL: (Beyond Syllabus)

To implement LL in languages like BASIC and FORTRAN, cursor implementation is used. because these languages does not support pointers.

Two features of pointers are

① The data are stored in a collection of structure. Each structure contains data & a pointer to the next structures.

② A new structure can be created by calling malloc () and deleted by calling free();

These 2 features should be simulated by the cursor implementation.

To satisfy ① a global array of structure is used. To satisfy ② a list is maintained.

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Initial cursor space.

Slot	Element	Next
0	–	6
1	b	9
2	f	0
3	header	7
4	–	0
5	header	10
6	–	4
7	c	8
8	d	2
9	e	0
10	a	1

Cursor implementation of LL.

The value 0 for next is the equivalent of a NULL pointer.

If L is 5 then L is, a b e.

M is 3 then M is c d f.

Declaration for cursor. LL.

```
Struct Node
{
ElementType Element;
Position Next;
};
```

CursorAlloc & CursorFree.

```
Static Position CursorAlloc (void)
{
  Position P;
   P = CursorSpace [0]. Next;
   CursorSpace [0]. Next = CursorSpace [P]. Next;
   return P;
 }
```

```
CursorFree (Position P)
{
  CursorSpace [P]. Next = CursorSpace [0]. Next;
  CursorSpace [0]. Next = P;
 }
```

To check is Empty.

```
int IsEmpty (List L)
 {
  return CursorSpace [L]. Next == 0;
 }
```

To check is Last Element.

```
int IsLast (Position P, List L)
 {
  return CursorSpace [P]. Next == 0;
 }
```

Find.

```
Position Find ( ElementType X, List L)
{
Position P;
P = CursorSpace [L]. Next;
While (P && CursorSpace [P]. Element != X)
    P = CursorSpace [P]. Next;
return P;
}
```

Deletion.

```
Void Delete ( ElementType X, List L)
{
 Position P, TmpCell;
 P = FindPrevious (X, L);
 if (! IsLast (P, L))
  {
   TmpCell = CursorSpace [P]. Next;
   CursorSpace [P]. Next = CursorSpace [TmpCell]. Next;
   CursorFree (TmpCell);
  }
}
```

Insertion.

```
Void Insert (ElementType X, List L, Position P)
{
 Position TmpCell;
 TmpCell = CursorAlloc ();
  if (TmpCell == 0)
    FatalError ("out of space");
 CursorSpace[TmpCell].Element = X;
 CursorSpace [TmpCell].Next = CursorSpace [P].Next;
 CursorSpace [P].Next = TmpCell;
}
```

Scanned by TapScanner

UNIT - II

STACKS & QUEUES.

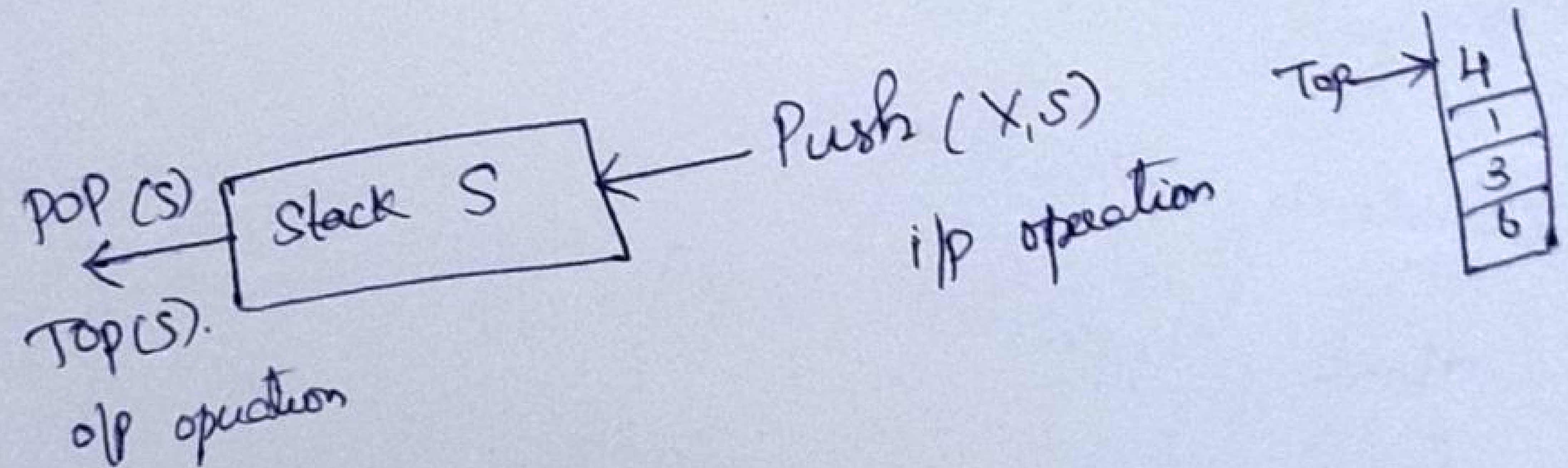
STACK ADT.

A stack is a list with the restriction that insertion and deletion can be performed ~~or~~ in only one position called end of the list called top.

operations on stack are

1) PUSH - insertion
2) POP - deletion.

Stack are known as LIFO (Last in First Out) lists.



Implementation of Stack.

- Array implementation
- Linked List "

Linked List implementation.

1) Declaration of a stack.

```
Struct Node
{
ElementType Element;
ptrToNode   Next;
};
```

2. To check if stack is empty.

```
int IsEmpty (Stack S)
{
return S-> Next ==NULL;
}
```

3. Create a new stack.

```
Stack CreateStack (Void)
{
 Stack S;
 S= malloc (Sizeof (Struct Node));
 if (S==NULL)
    error
 MakeEmpty (S);
 return S;
}
```

4. MakeEmpty.

```
Void MakeEmpty (Stack S)
{
 if (S == NULL)
    error ("Must create a stack first");
 else
  while ( !IsEmpty (S)).
     Pop (S);
}
```

5. Push.

```
Void Push ( ElementType X, Stack S)
{
 PtrToNode TempCell;
 TempCell = malloc (Sizeof (Struct Node));
  if ( TempCell == NULL)
     error
  else
  {
  TempCell -> Element = X;
  TempCell -> Next = S -> Next;
   S -> Next = TempCell;
   }
}
```

[S is the header node]

6. Return the top element in the stack.

```
ElementType Top (Stack S)
{
  if (! IsEmpty (S))
     return (S->Next-> Element);
  Error ("Empty stack");
  return 0;
}
```

7. Pop.

```
Void Pop (Stack S)
{
  ptrNode FirstCell;
  if (IsEmpty (S))
     Error ("Empty Stack");
  else
  {
    FirstCell = S-> Next;
    S->Next = S-> Next ->Next;
    free (FirstCell);
  }
}
```

Scanned by TapScanner

Array Implementation of Stack.

This avoids the usage of pointer. The array size should be declared in advance. Parameter considered here is "TopOfStack".

For an empty stack TOS value is -1.

To push an element, increase the TOS value by 1 and then set Stack[TOS] = X.

To pop an element, set the return value to Stack[TOS] and decrease the value of TOS.

2 errors that may occur here are,

- Push on a full stack called stack overflow.
- Pop on an empty stack called Stack underflow.

To avoid the overflow condition, size of the array can be fixed with the maximum size.

1) Stack declaration.

```
Struct StackRecord
{
int capacity;
int Tos;
ElementType *Array;
};
```

2) Stack Creation.

```
Stack CreateStack (int MaxElements)
{
Stack S;
if (MaxElements < MinStackSize)
    Error ("Size is toosmall");

S = malloc (sizeof (struct StackRecord));
if (S == NULL)
    Error ("out of space");

S-> Array = malloc (Sizeof (ElementType) * MaxElements);
if (S->Array == NULL)
    Error ("out of space");
S-> Capacity = MaxElements;
MakeEmpty (S);
return S;
}
```

3. Freeing Stack.

```
void DisposeStack (Stack S)
{
 if (S != NULL)
 {
  free (S->Array);
  free (S);
 }
}
```

4. To check if Empty:

```
int IsEmpty (Stack S)
{
return S→Tos == EmptyTos;
}
```

5. To make a stack empty:

```
Void MakeEmpty (Stack S)
{
S→ Tos= EmptyTos;
}
```

6. Push.

```
Void Push (ElementType X, Stack S)
{
if (Isfull (S))
   Error ("Full");
else
 S→Array [++ S→ Tos] = X;
}
```

7. Return TOS:

```
ElementType Top (Stack S)
{
if (! IsEmpty (S))
   return S→Array [S→Tos];
Error ("Empty stack");
return 0;
}
```

8. Pop.

```
Void  Pop (Stack S)
{
  if (IsEmpty (S))
     Error ( "Empty Stack");
  else
   S->Tos --;
}
```

9. Give Top element & pop a stack.

```
ElementType TopAndPop (Stack S)
{
 if (!IsEmpty (S))
   return S->Array[S->TOS --];
 Error ("Empty stack");
 return 0;
}
```

APPLICATIONS OF STACK.

1) Balancing symbols
2) Postfix expression evaluation.
3) Infix to postfix conversion
4) Function call.

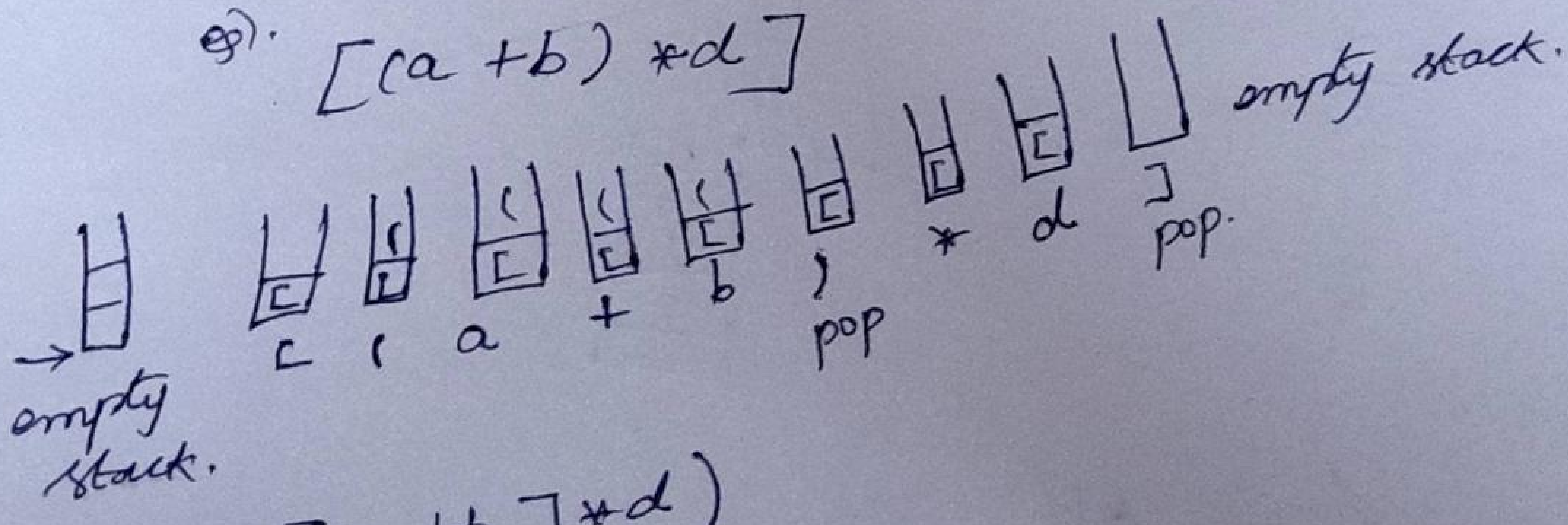
1. Balancing symbols.

i) Make an empty stack.

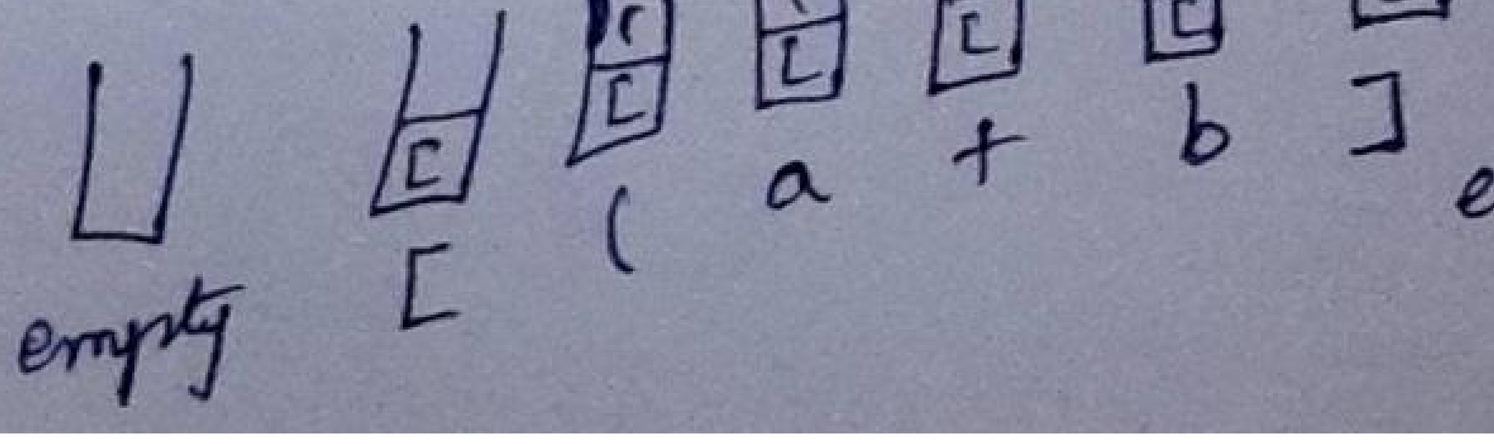
ii) Read Characters until end of file. If the character is opening symbol push it onto the stack. If it is a closing symbol, check the stack, if it is empty report an error or pop the stack. If the symbol popped out is not the corresponding opening symbol then report error.

iii) At the end of file if the stack is empty, then it is balanced, if not report error.

eg). [(a+b)*d]



[(a+b]*d)



error because this does not match

2. Postfix expression Evaluation

Read the postfix expression one character at a time until it encounters # (eof).

i) If the character is an operand, push its associated value onto the stack.

ii) If it is an operator, pop two values from the stack, apply the operator to them and push the result to the stack.

Q. AB * C DE / - + . Assume values of A B C D E as 1, 2, 3, 4, 2 respectively.

Input		Stack (top to bottom)
A		1
B		2, 1
*	2 * 1	2
C		3, 2
D		4, 3, 2
E		2, 4, 3, 2
/	4/2	2, 3, 2
-	3 - 2	1, 2
+	2+1	3

Result = 3

3. Conversion of infix to postfix expression.

Evaluation order in which operations are executed.

① Brackets ② Exponentiation ③ * or / ④ + or −

Operators with same priority are evaluated from left to right.

Algorithm:

Read the infix expression one character at a time until it encounters the delimiter '#'

i) If the character is an operand, place it onto the output.

ii) If the character is an operator, push it onto the stack. If the stack operator has a high or equal priority than input operator then pop that operator from the stack and place it onto output.

iii) If the character is a left parenthesis push it onto the stack.

iv) If the character is a right paranthesis pop all the operators from the stack till it encounters left paranthesis, discard both the parenthesis in the output.

a) $a+b*c+(d*e+f)*g$

I/P	Stack (top → bottom)		O/P
a	(empty)		a
+	+		a
b	+		ab
*	*, +	* > +	ab
c	*, +		abc
+	+	+ < * + = +	abc*+
(	(, +		abc*+
d	(, +		abc*+d
*	*, (, +		abc*+d
e	*, (, +		abc*+de
+	+, (, +	+ < *	abc*+de*

Symbol	Stack	Postfix
f	+ (+	abc*+de*f
)	+	abc*+de*f+
*	* +	abc*+de*f+
g	* +	abc*+de*f+g
No i/p. so pop the stack.	(empty)	abc *+de * f+g * +

Post fix expression of

a+b*c+(d*e+f)*g is,

abc*+de*f+g*+.

Assignment.

① A+(B*C-(D/E↑F)*G)*H).

ABC* DEF↑/G*-H*+

② A$B*C-D+E/F/(G+G)

4. Function call.

The following informations are needed to be saved during a function call.

1) Register value : The local variable names in the calling routine.

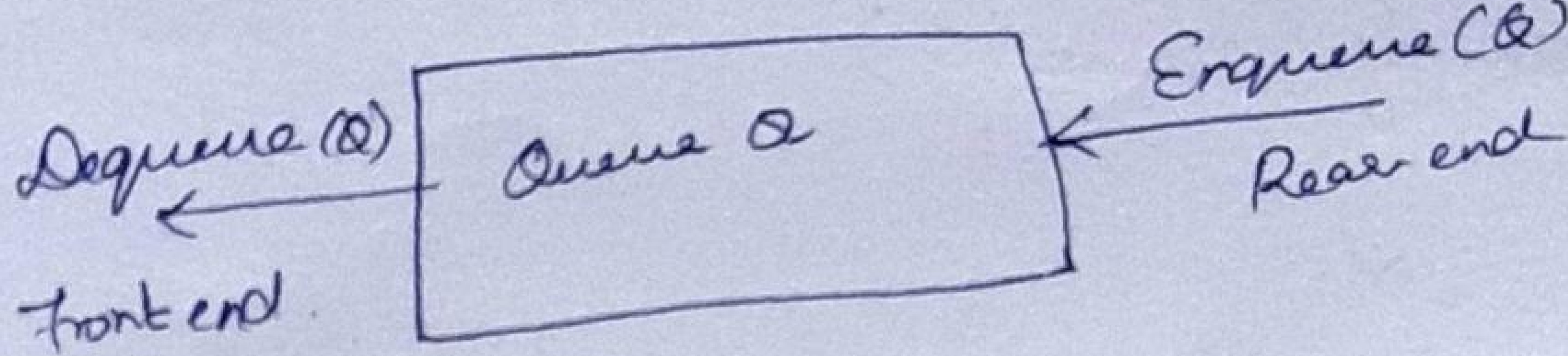
2) Return address : is the current location in the calling routine.

When a function is called, the called function should be executed and after that the next instruction to be executed is the instruction next to the calling function. So that the address should be saved in a stack. The address will be present in the top of stack and easily it can be loaded.

Queue ADT

Like stacks, queues are lists. With queue, insertion is done at one end and deletion is performed at the other end.

Array implementation of Queues:



- Insertion is refered to as Enqueue and done at Rear end
- deletion is called as Dequeue. & done at Front end.

To Enqueue X, increment size and Rear, then set Queue[Rear] = X.

To Dequeue, set the return value to Queue [front], decrement size and then increment front.
The size of the array is fixed, but the size of the queue keeps on changing as the insertion and deletion are carried out.

Declaration:

```
Struct queue
{
int size, front, rear, capacity;
elementtype * data;
};
```

Is Empty.

```
int IsEmpty (Queue Q)
{
return Q->size ==0;
}
```

To make an empty queue.

```
Void MakeEmpty (Queue Q)
{
Q -> Size = 0;
Q -> Front = 1;
Q -> Rear = 0;
}
```

Insertion:

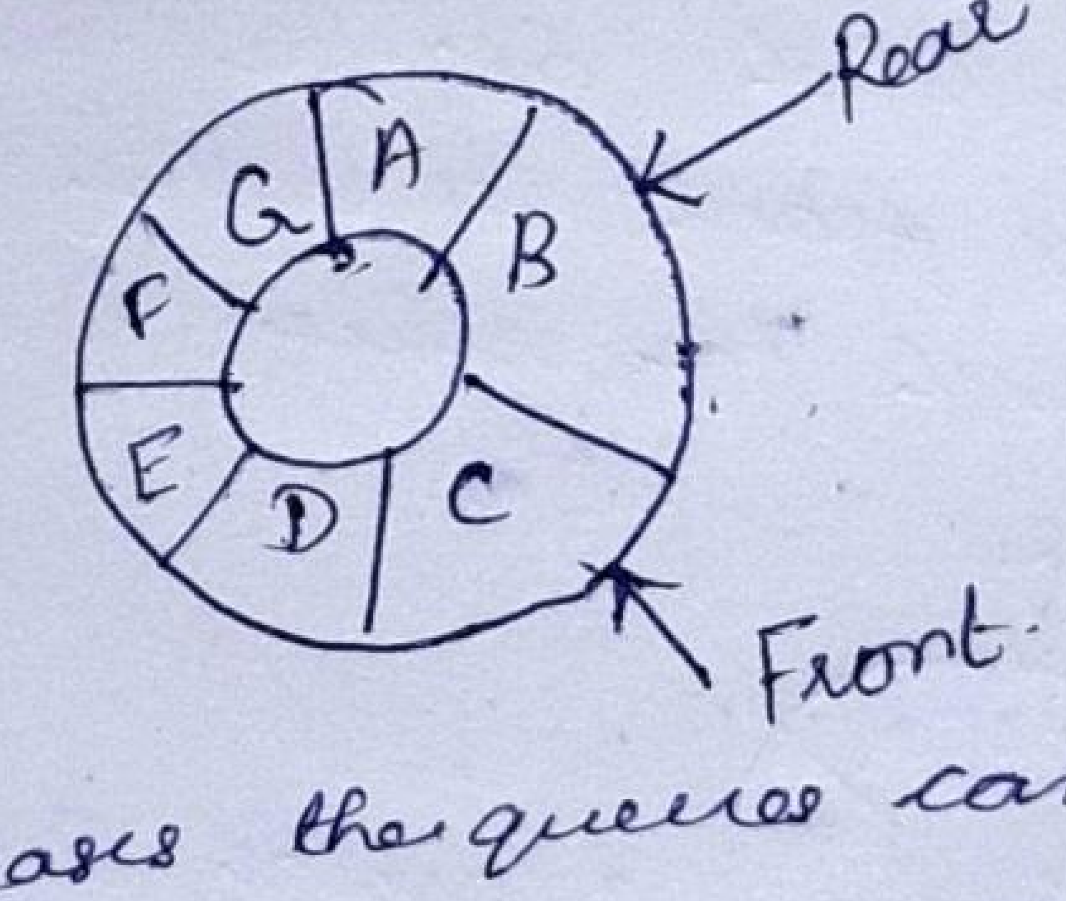
```
int Succ (int Value, Queue Q)
{
if (++ Value == Q-> Capacity)
    Value =0;
  return Value;
}
Void Enqueue (ElementType X, Queue Q)
 {
  if (IsFull (Q))
     Error
  else
  {
  Q -> size ++;
  Q -> Rear = Succ (Q -> Rear, Q);
  Q -> Array [Q -> Rear] = X;
  }
}
```

Dequeue.

```
Delete (Queue Q)
{
data = Q[front];
front = front +1;
}
```

Circular Queues:

Circular queues are the queues implemented in circular form rather than a straight line. This is used to overcome the problem of unutilized space in linear queue implemented as an array.



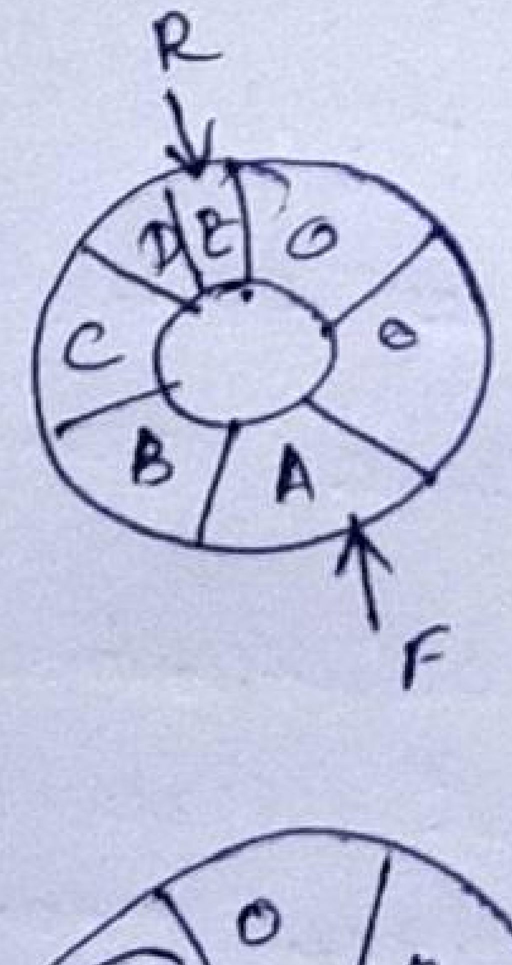
In some cases the queues can be reported as full even though slots of the queue are empty. Suppose an array x of n elements is used to implement a circular queue, the total size of the circular queue will be x [n-1]. When any one of the cell in the array is empty, insertion can be done directly. Here no rear & front end is taken into account. This is the main advantage of using circular queue.

Scanned by TapScanner

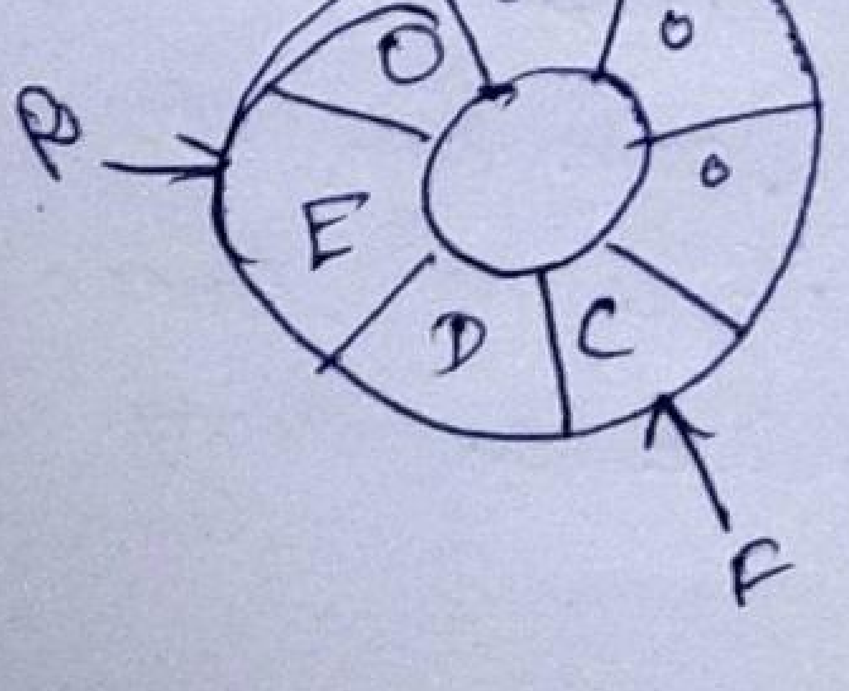
Insertion:

Initially the value of front and rear end is -1.

- If the front & rear are in adjacent location it means that the queue is full.
- If the value of front is -1 it denotes that the queue is empty and then the first element should be inserted. The values of front & rear are set to 0. and the new element is placed at 0^{th} position.
- Some of the position at the front end of the array might be empty. This happens if some element from the queue is deleted, when the value of rear is Max-1 and the value of front is greater than 0. In such case value of rear is set to 0 and the element to be added to this position.
- The element is added at the rear position in case the value of front is either equal to or greater than 0 and the value of rear is less than Max-1.



circular Q after adding 5 elements.



After deleting 2 elements.

Priority Queue:

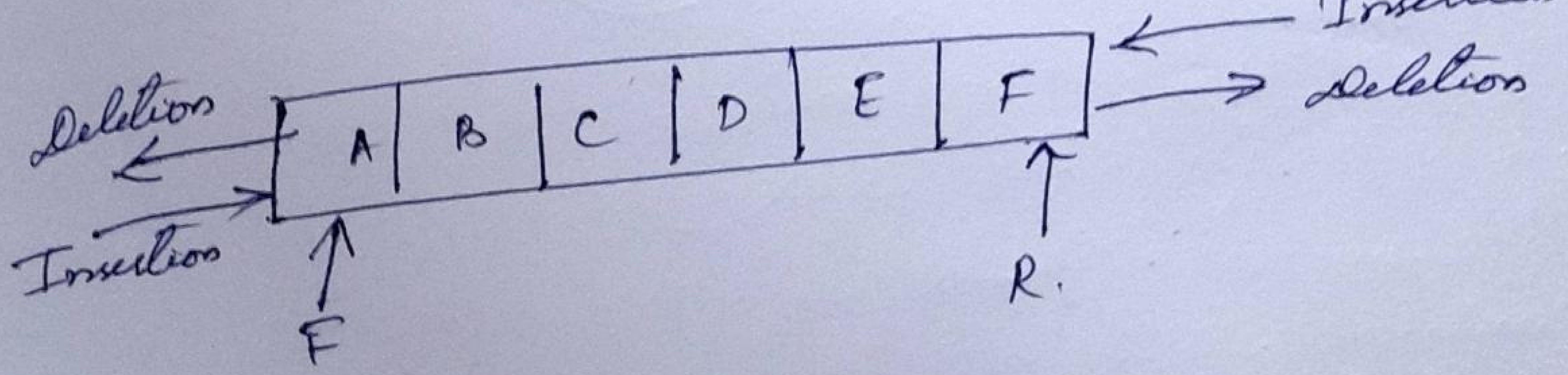
A priority Queue is a collection of elements where each element is assigned a priority and the order in which elements are deleted and processed is determined from the following rules.

- An element of higher priority is processed before any element of lower priority.
- Two elements with same priority are processed according to the order in which they are added to the queue.

eg. Time sharing system.

Double Ended Queue or Deque.

A dequeue is a linear list in which elements can be added or removed at either end but not at middle.



• Types of Dequeue

i) Input restricted dequeue

Insertion is allowed at only one end but deletion at both end.

2) Output restricted dequeue.

Deletion is allowed at only one end but deletion at both end.

• Application of Queue

• In a printer, the jobs sent are placed on a queue.

• There are many network setups of personal computers in which the disk is attached to one machine known as the file server. Users on other machines are given access to files on a first-come first-served basis on a queue.

UNIT - III

TREES

TREE ADT

Basic Terminologies

Root:

Root is a unique node in the tree to which further sub-trees are attached.

Parent node:

Node having further sub-branches is called parent node.

Child node:

The sub branches of the node called child node.

Leaves:

The node having no further sub-branches is called leaf node. These are the terminal nodes of the tree.

Degree of a node:

The total no. of sub-trees attached to a node is called degree of the node.

Degree of a tree:

The maximum degree in the tree is degree of tree.

Level of a node.

The level of a tree is the length of the path from the root to a node. The root node is always at level zero. The adjacent nodes of root are supposed to be at level 1 and so on.

Height

The maximum level is the height of the tree.

Predecessor.

While displaying the trees, nodes occur previous to some other node are called predecessor of a node.

Successor:

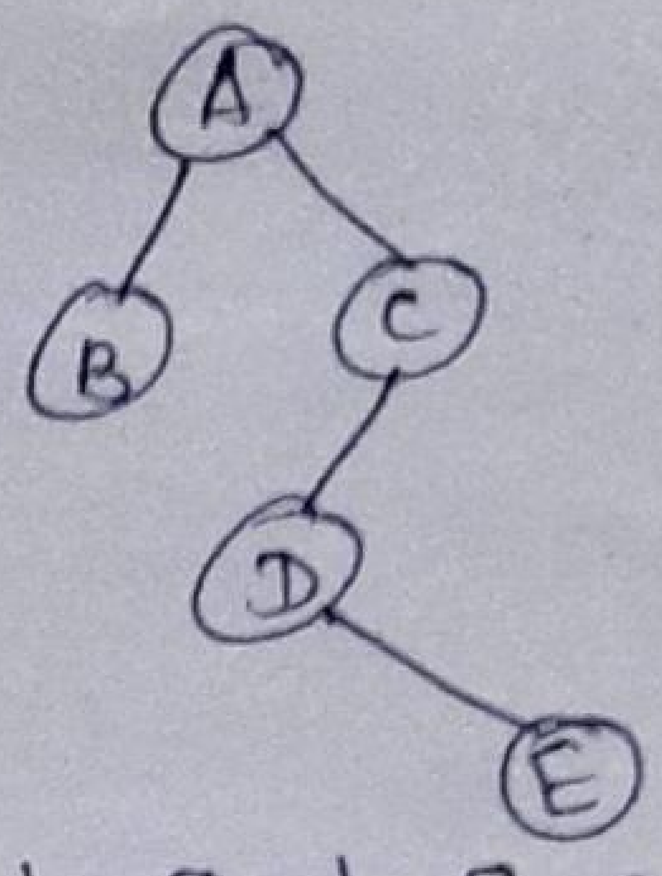
While displaying the tree, nodes occur next to some other node are called successor of a node.

Internal & external node.

Non-leaf node is called internal node. Leaf node is called external node.

Sibling.

The nodes with common parent are called siblings.



Root : A

Parent : A, C, D

Child : B, C, D, E

Leaves : B, E

Degree of nodes : A - 2 , B - 0 , C - 1 , D - 1 , E - 0

Degree of tree : 2

Level of tree : A-0, B-1, C-1, D-2, E-3.

Height of tree : 3

Predecessor : Inorder traversal BADEC
Predecessor of A is B , E is D.

Successor of A is D , E is C.

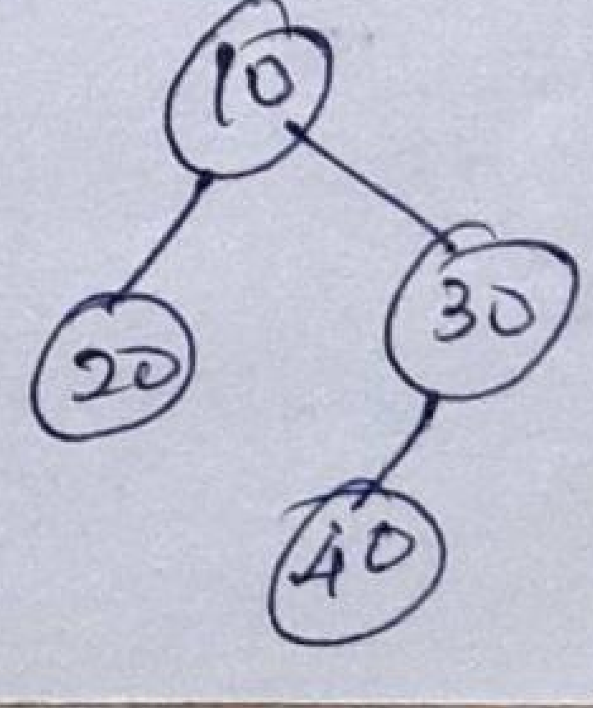
Internal nodes : C, D

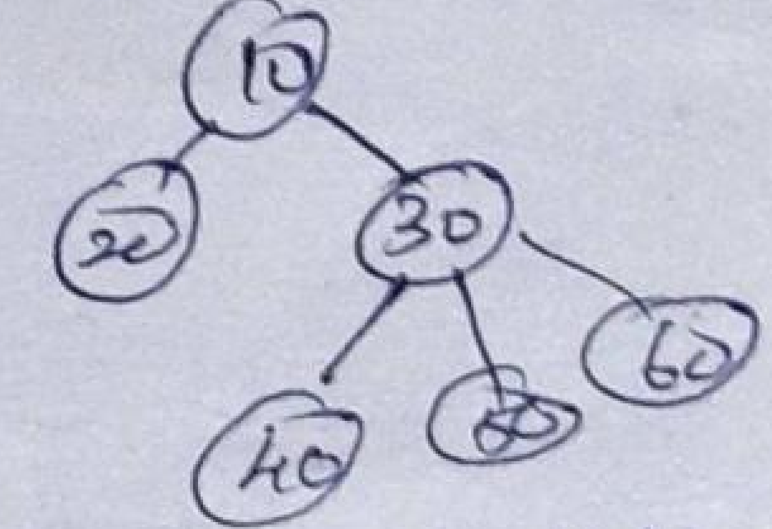
External Nodes : B, E

Sibling : B, C

__Binary tree__ :

It is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called left sub tree and right sub tree.

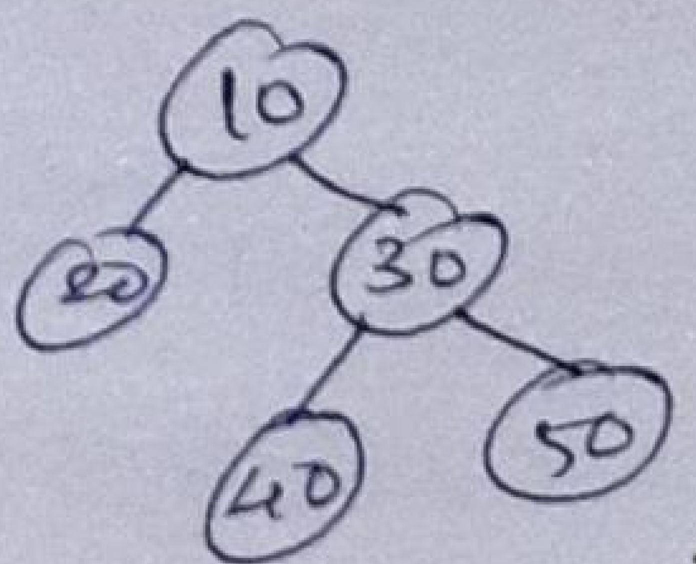


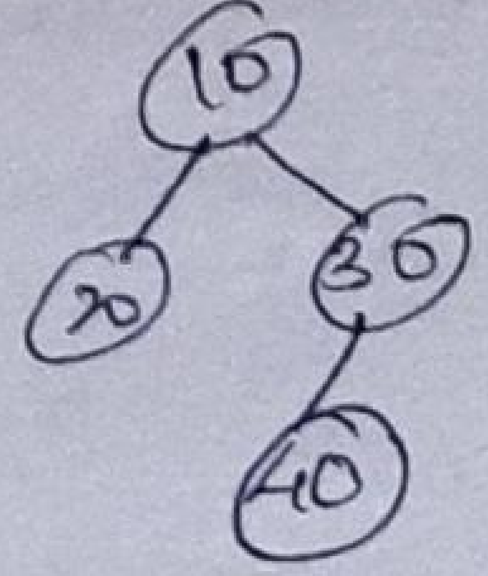


Not a binary t

Full binary tree:

A full binary tree is a tree in which every node has zero or 2 children. ($2^{h+1}-1$)





Not

Complete binary tree:

A complete binary tree is a full binary tree in which all leaves are at the same level.

The total no. of nodes can be found by the formula,

$2^{h+1}-1$, h - height of tree.

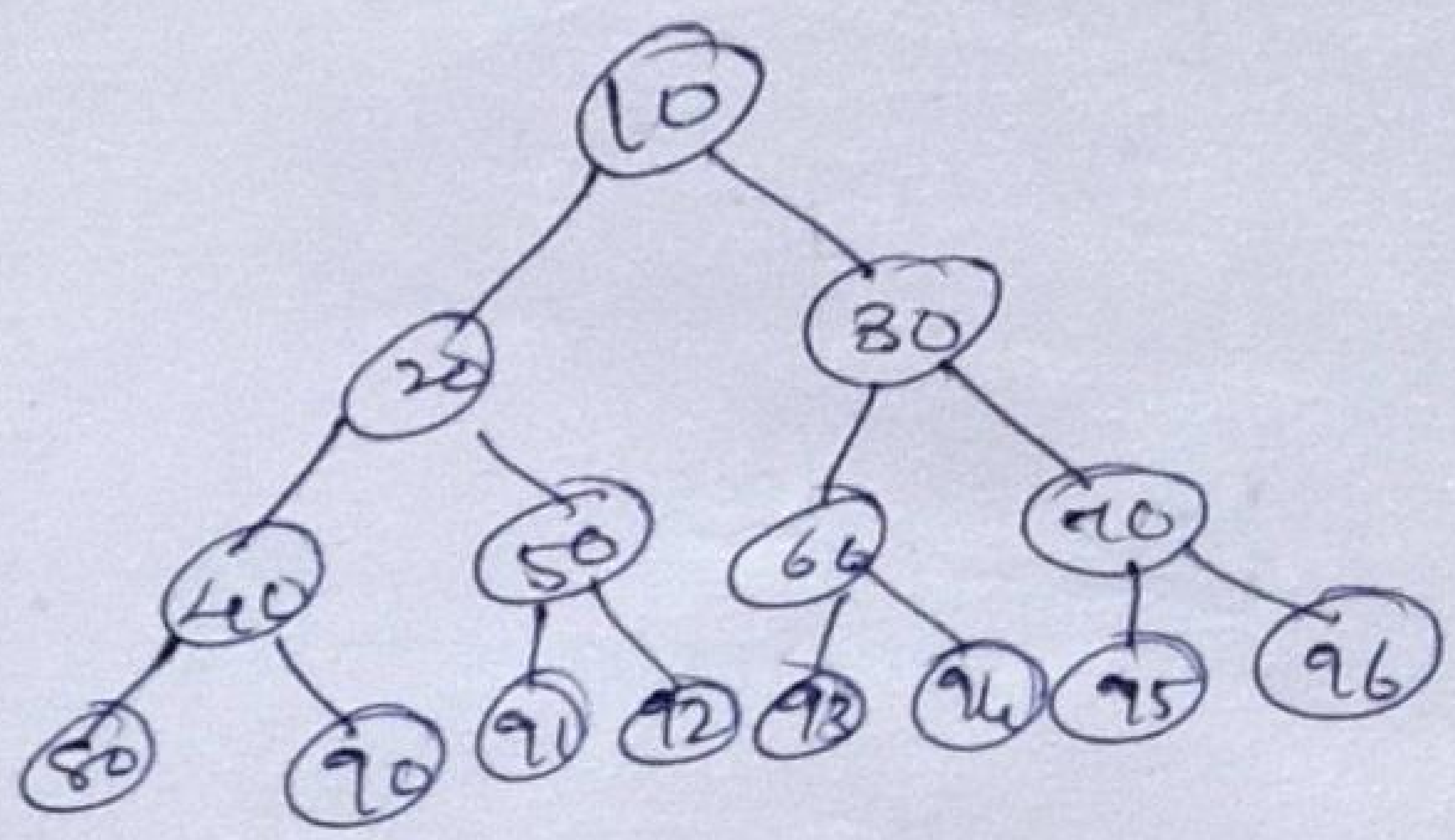
The No. of nodes in a level can be found by.

2^h. h is level.

If h = 3

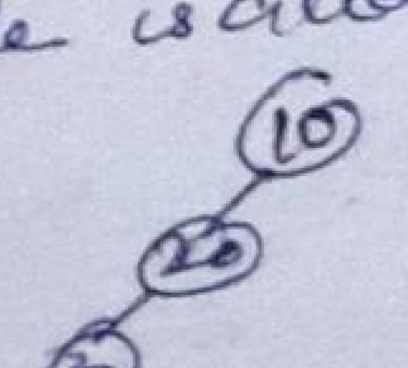
Total no. of nodes : 15

No. of nodes in level 3 : 8



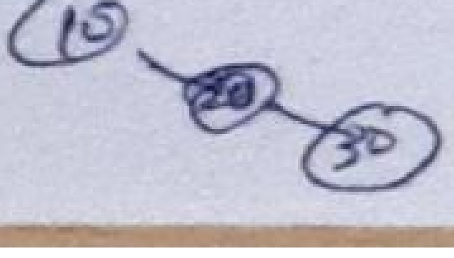
Left Skewed tree

Tree in which each node is attached as a left child of parent node.



Right Skewed tree.

Tree in which each node is attached as a right child of parent node.



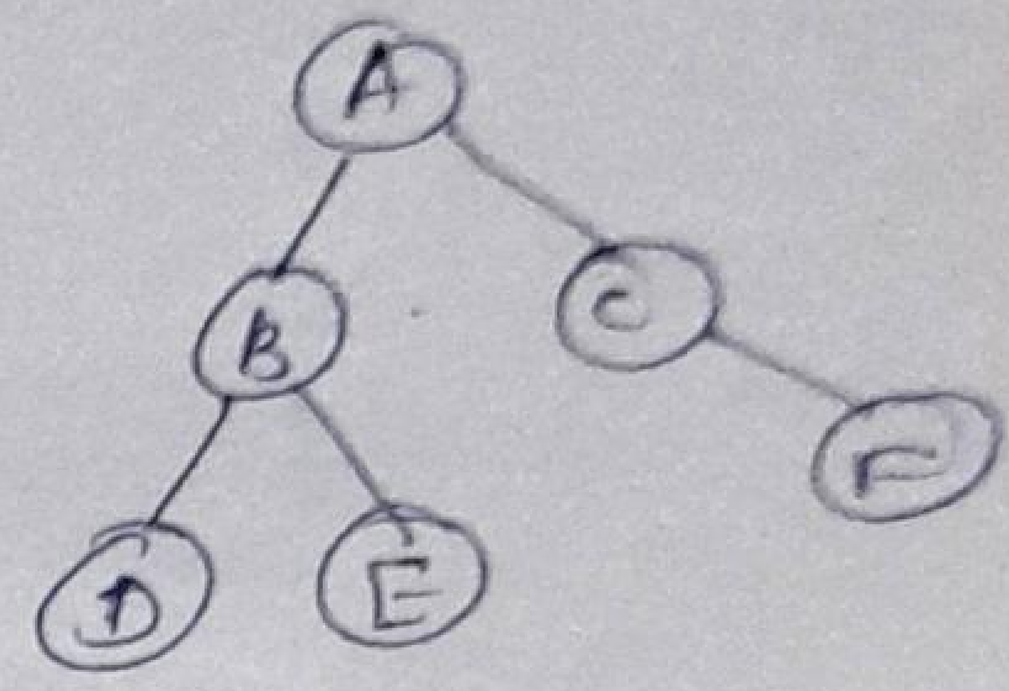
Scanned by TapScanner

Tree Traversals:

Traversing is the process of visiting every node in the tree exactly once. A complete traversal of a binary tree visits the node of the tree in some linear sequence.

Types.

- Preorder or depth first order
- Inorder or symmetric order
- Post order.



Inorder Traversal.

1. Traverse the left sub tree inorder
2. Visit the root
3. Traverse the right sub tree inorder.

Inorder traversal of the binary tree, DBEACF

Routine:

```
Void inorder (Tree T)
{
if (T != NULL)
  {
  inorder (T -> left);
  Printf ("%d", T -> data);
  inorder (T -> right);
  }
}
```

Preorder traversal.

1. Visit the root
2. Traverse the left sub tree in preorder
3. Traverse the right subtree preorder.

ABDECF

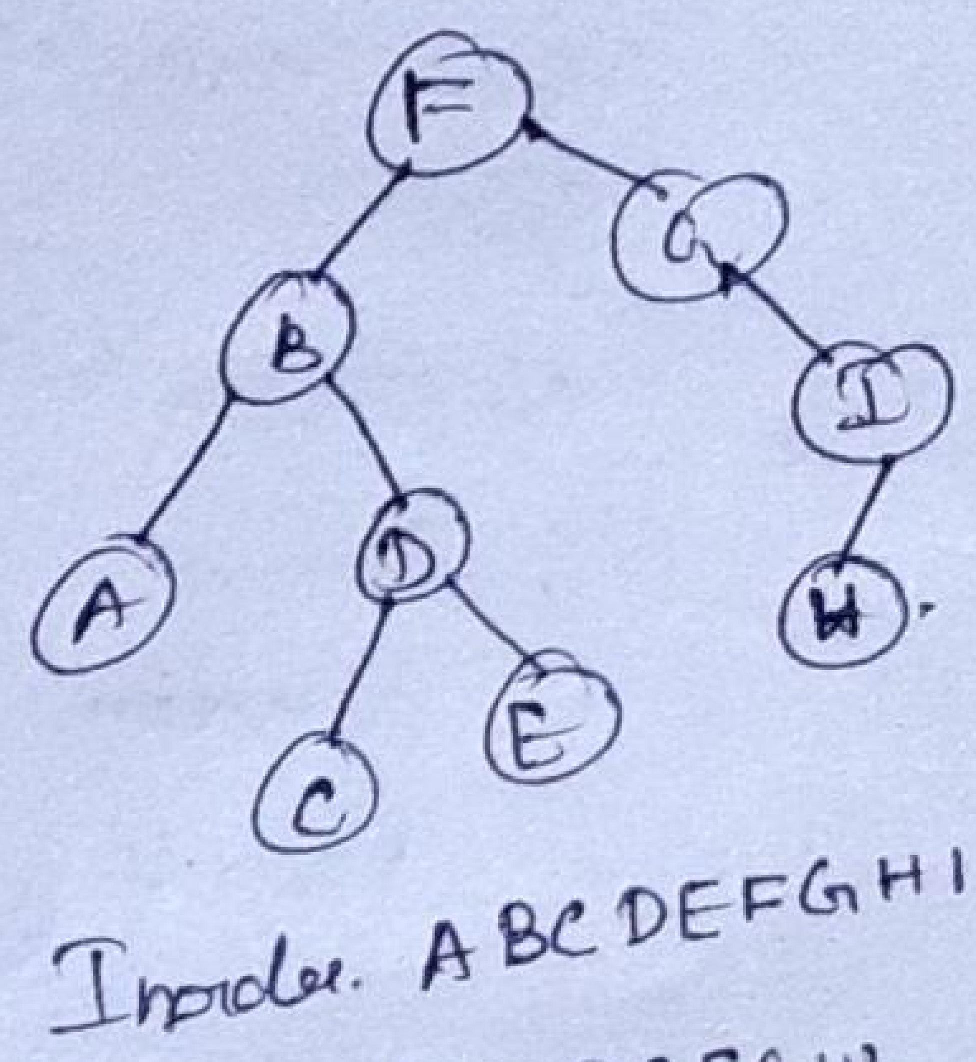
```
Void Preorder (Tree T)
{
 if (T ! = NULL)
  {
   printf ("%d", T->data);
   Preorder (T->left);
   Preorder (T->right);
  }
}
```

Post order traversal.

1. Traverse the left subtree in post order
2. Traverse the right subtree in post order
3. Visit the root.

DEBFCA.

```
Void postorder (Tree T)
{
 if (T != NULL)
  {
   postorder (T-> left);
   postorder (T-> right);
   printf ("%d", T->data);
  }
}
```



Inorder. ABCDEFGHI

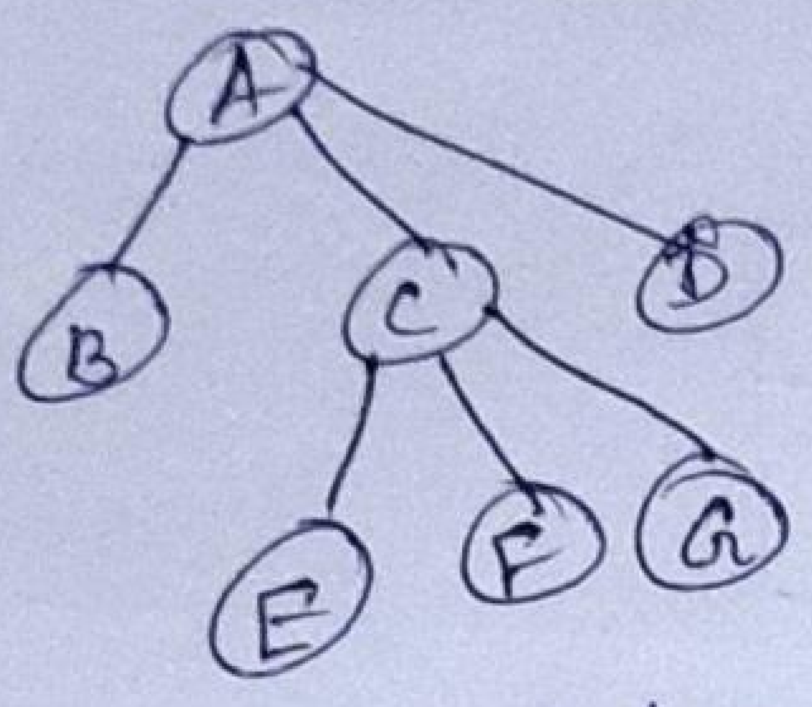
Preorder. FBADCEGIH

Postorder: ACEDBHIGF

Left child Right sibling data.

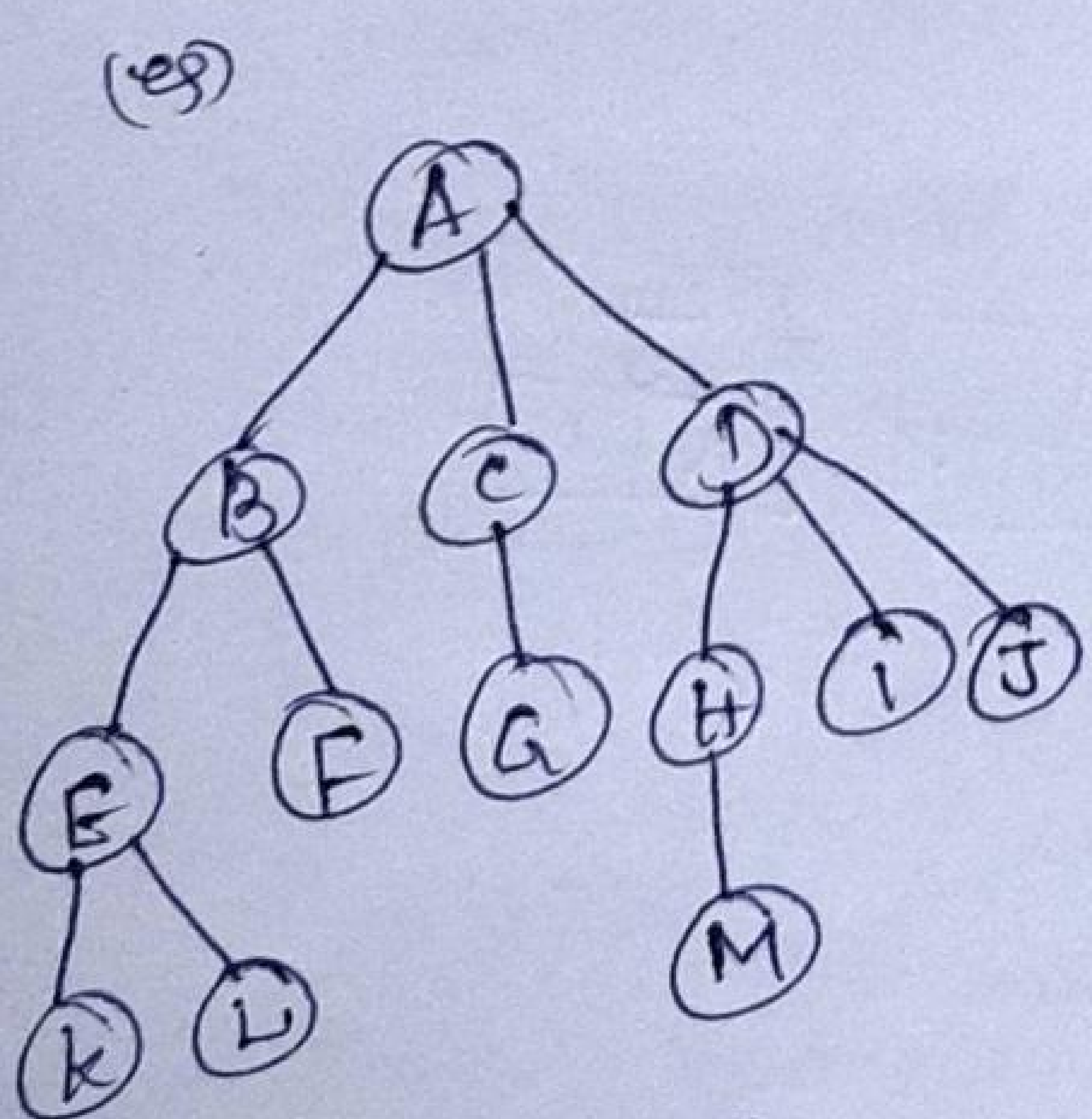
General tree:

In general tree, the root has one or more child nodes. Each child node can be a root of another set of child nodes.



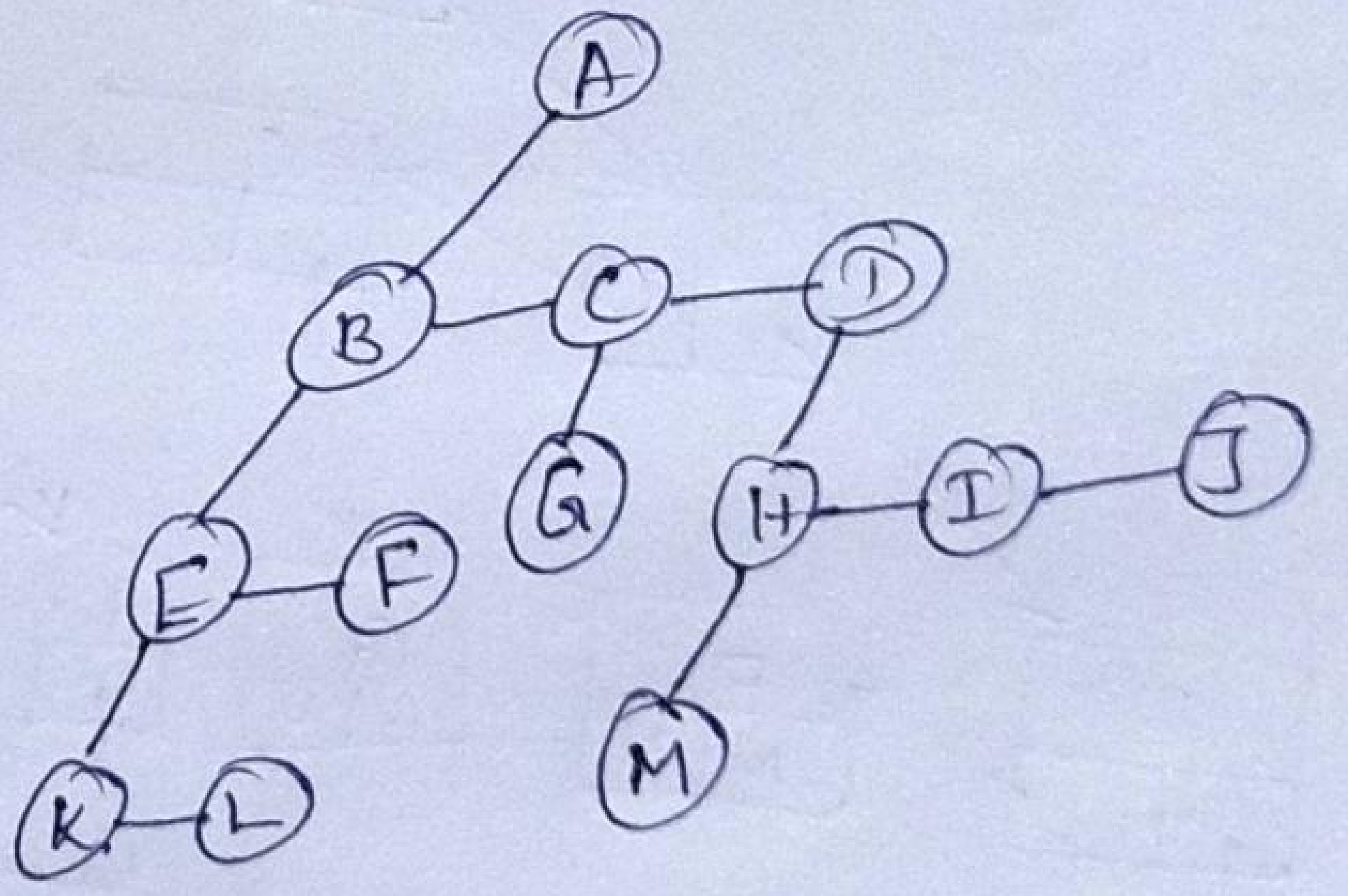
This is also called as first child next sibling DS. Here, the left child is put as such and its sibling is represented using a link from left to right from the left child

(eg)



General tree

⇒

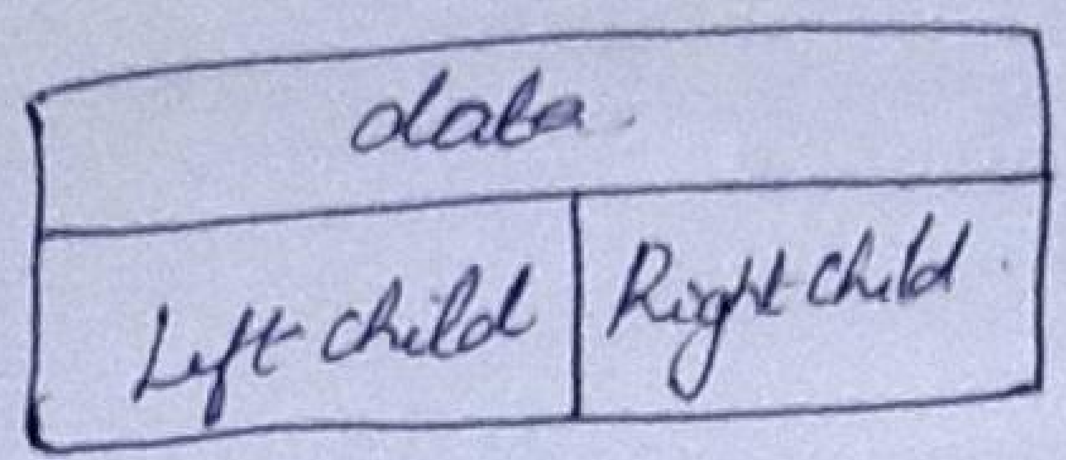


Left child - Right Sibling representation

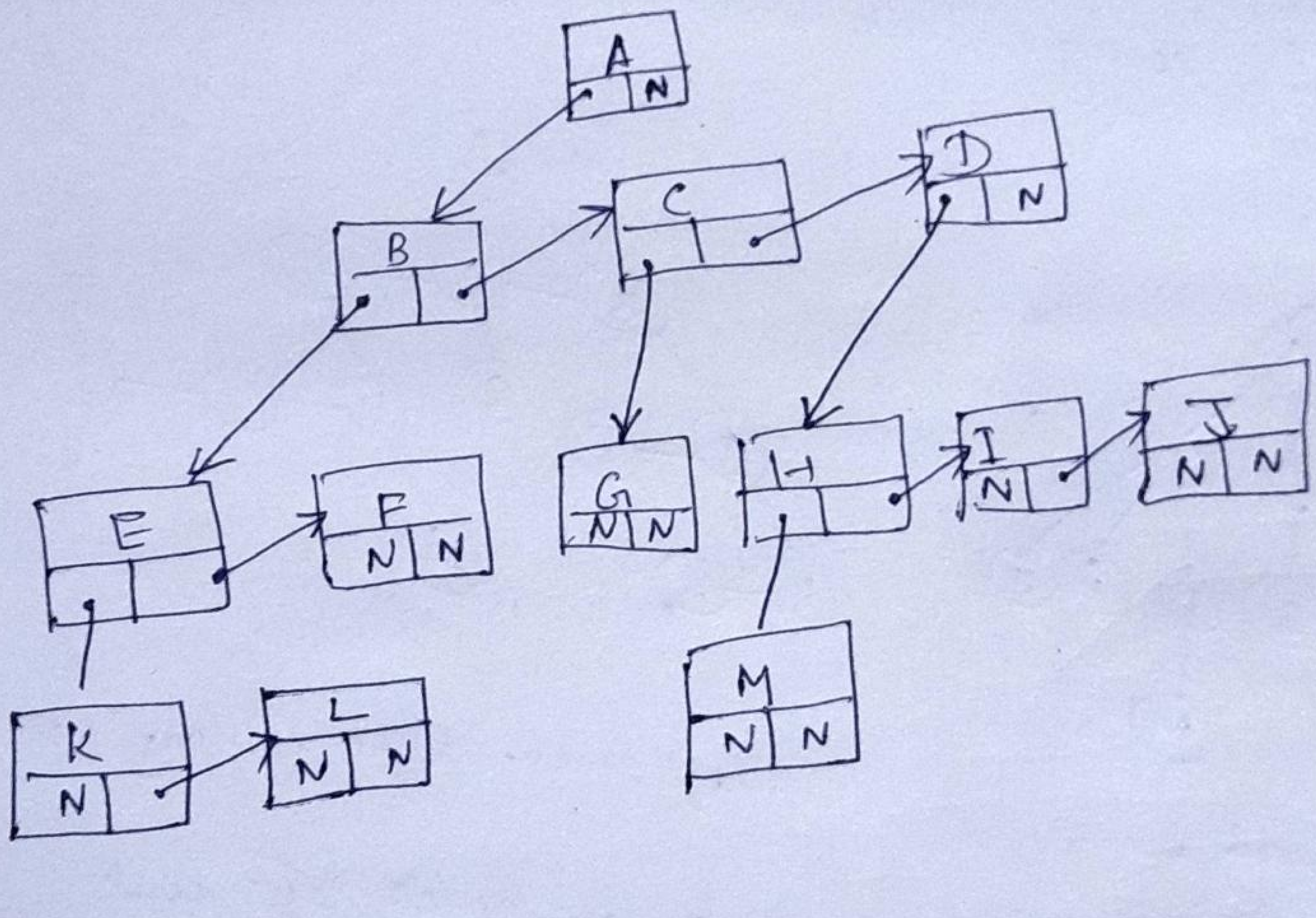
- Put the left child node to the node's left link.
- Put the sibling to the nodes right link.

In this DS, the children of a nodes are maintained in a linked list. Each node has 2 pointers one for its left most child and the other for its right sibling. The edges that points downwards are first child reference and that go left to right are next sibling reference.

Node

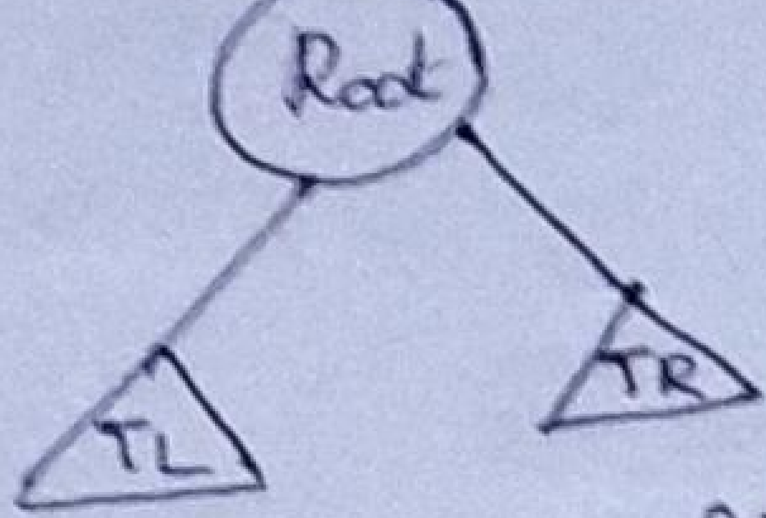


The tree can be represented using linked list as,



Binary tree ADT

A binary tree is a tree in which no node can have more than two children.



Generic binary tree.

The depth of an average binary tree is considerably smaller than N. Average depth of a binary tree is $O(\sqrt{N})$ and that of a binary search tree is $O(\log N)$

Implementation:

- Linear Array implementation
- Linked list implementation

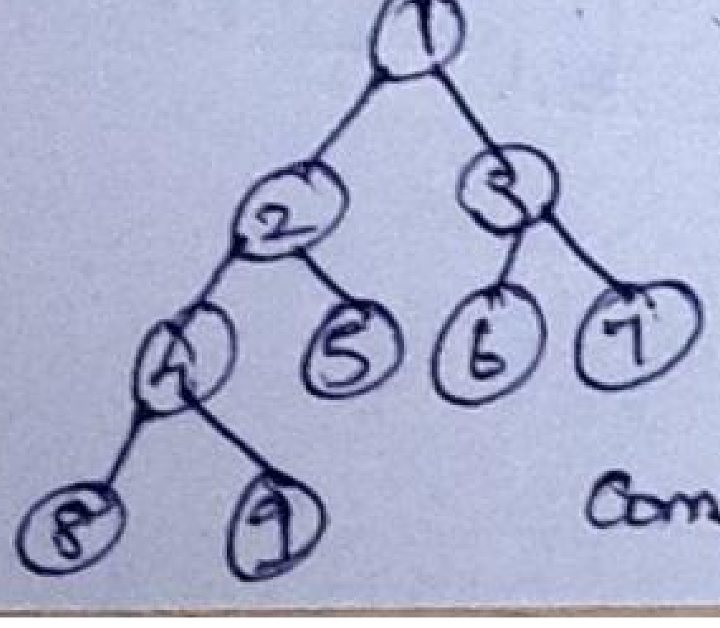
Types:

Full binary tree:

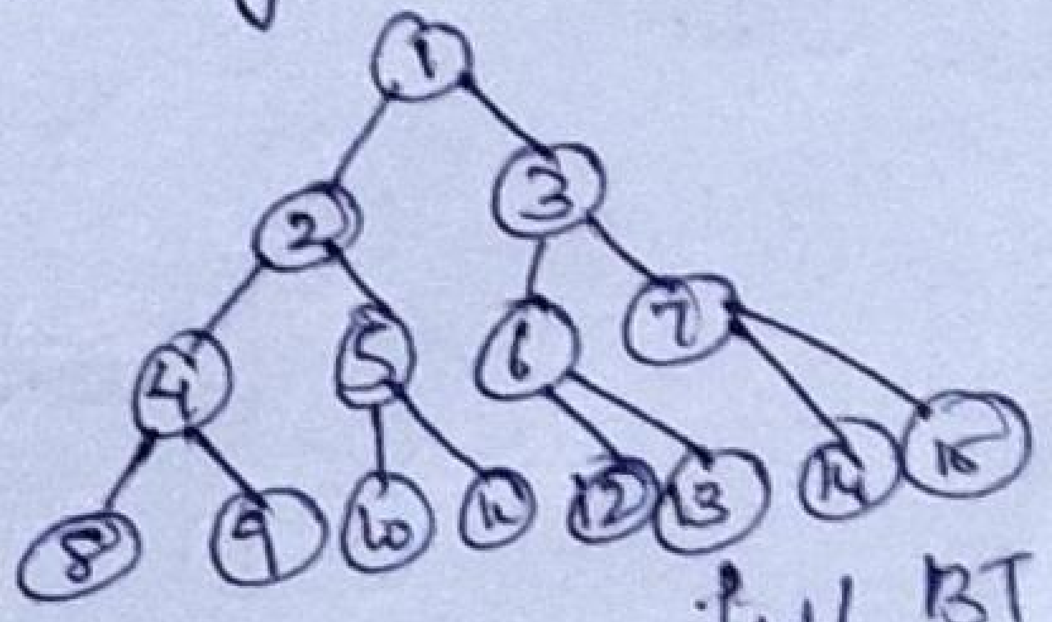
A full binary tree of height h has $2^{h+1}-1$ nodes

Complete binary tree:

A complete binary tree of height h has between 2^h and $2^{h+1}-1$ nodes. In the bottom level the elements should be filled from left to right.



Complete BT

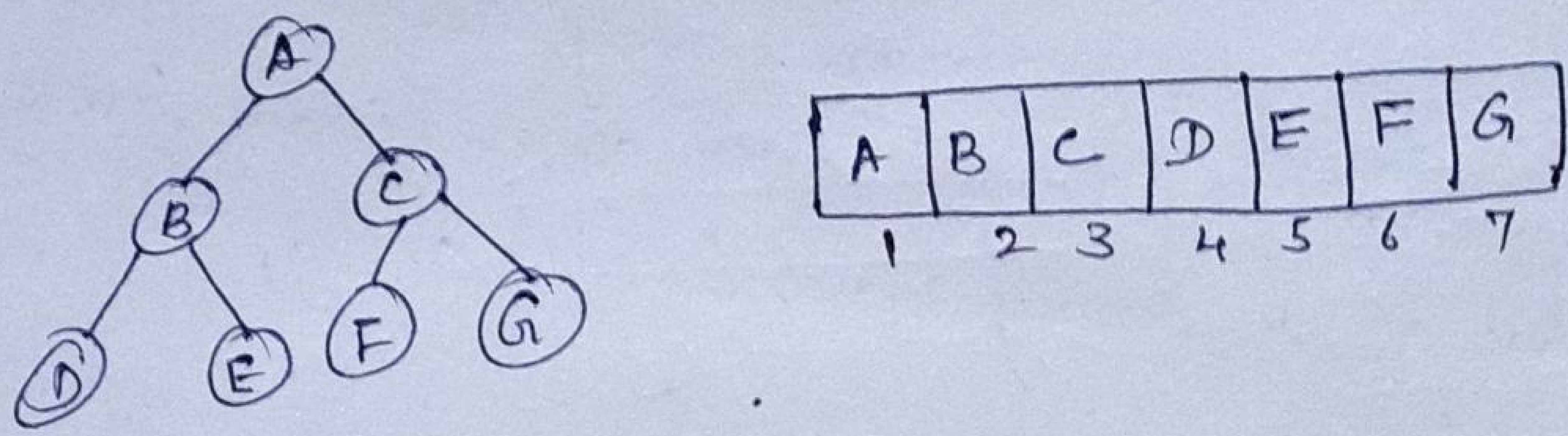


full BT

A full binary tree can be a complete binary tree, but all complete binary tree is not a full binary tree.

Array / Linear Representation:

The elements are represented using arrays.
For any element in position i, the left child is in position 2i, the right child is in position (2i+1) and the parent is in position i/2.

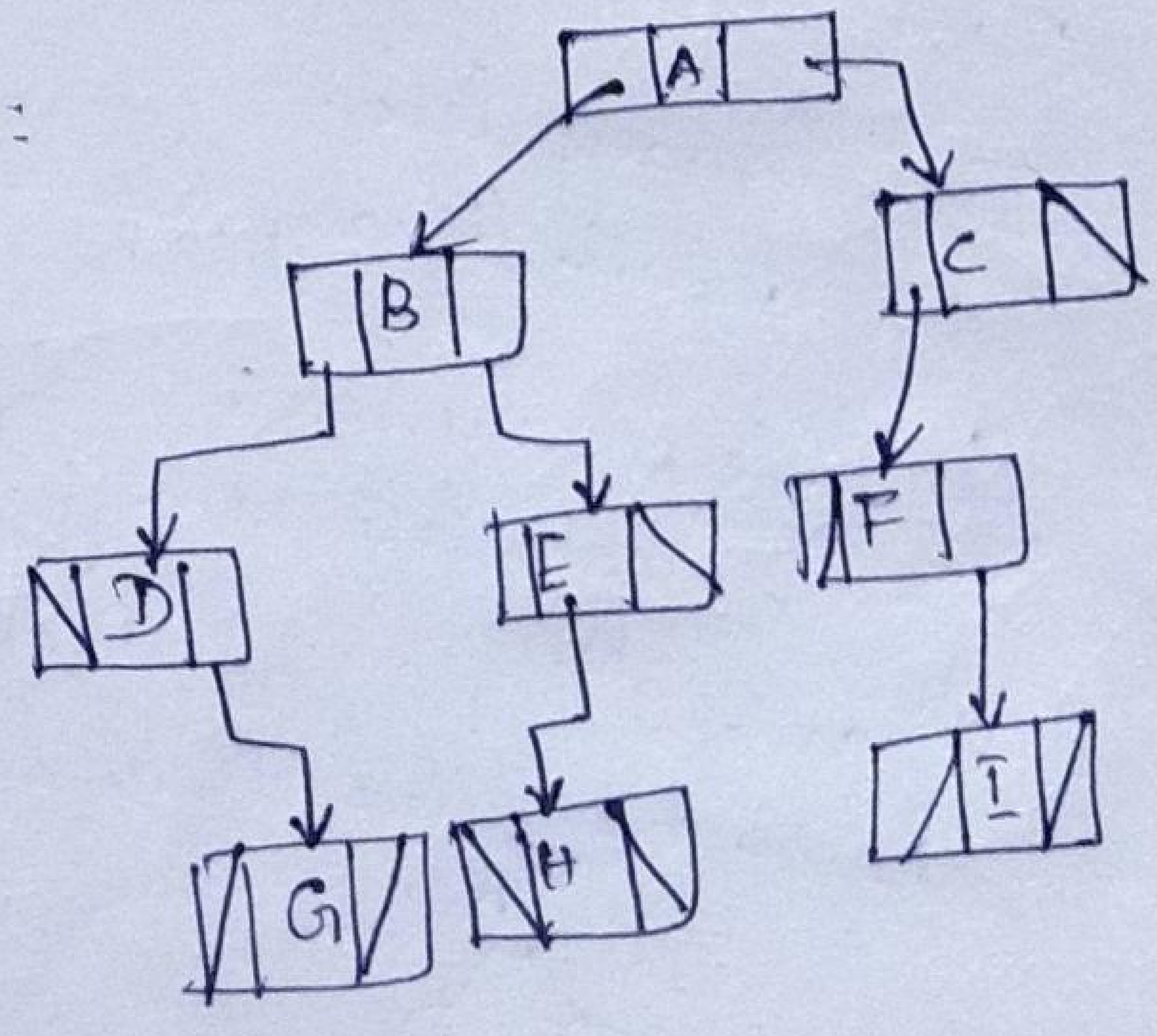


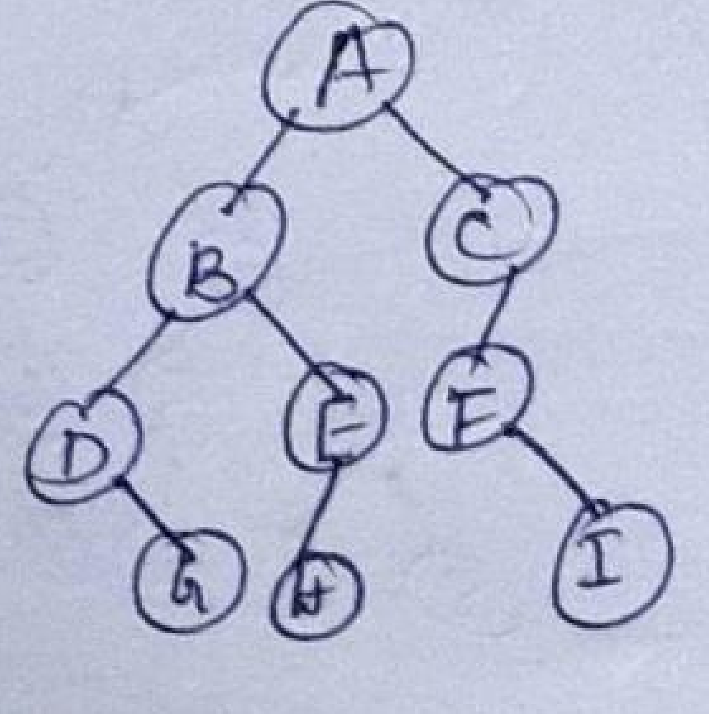
Linked list Representation:

Binary tree can be represented using a doubly LL since it has only 2 pointers to the left node and the right node.

Binary node declaration:

```
Struct TreeNode
{
ElementType Element;
Tree left;
Tree Right;
};
```

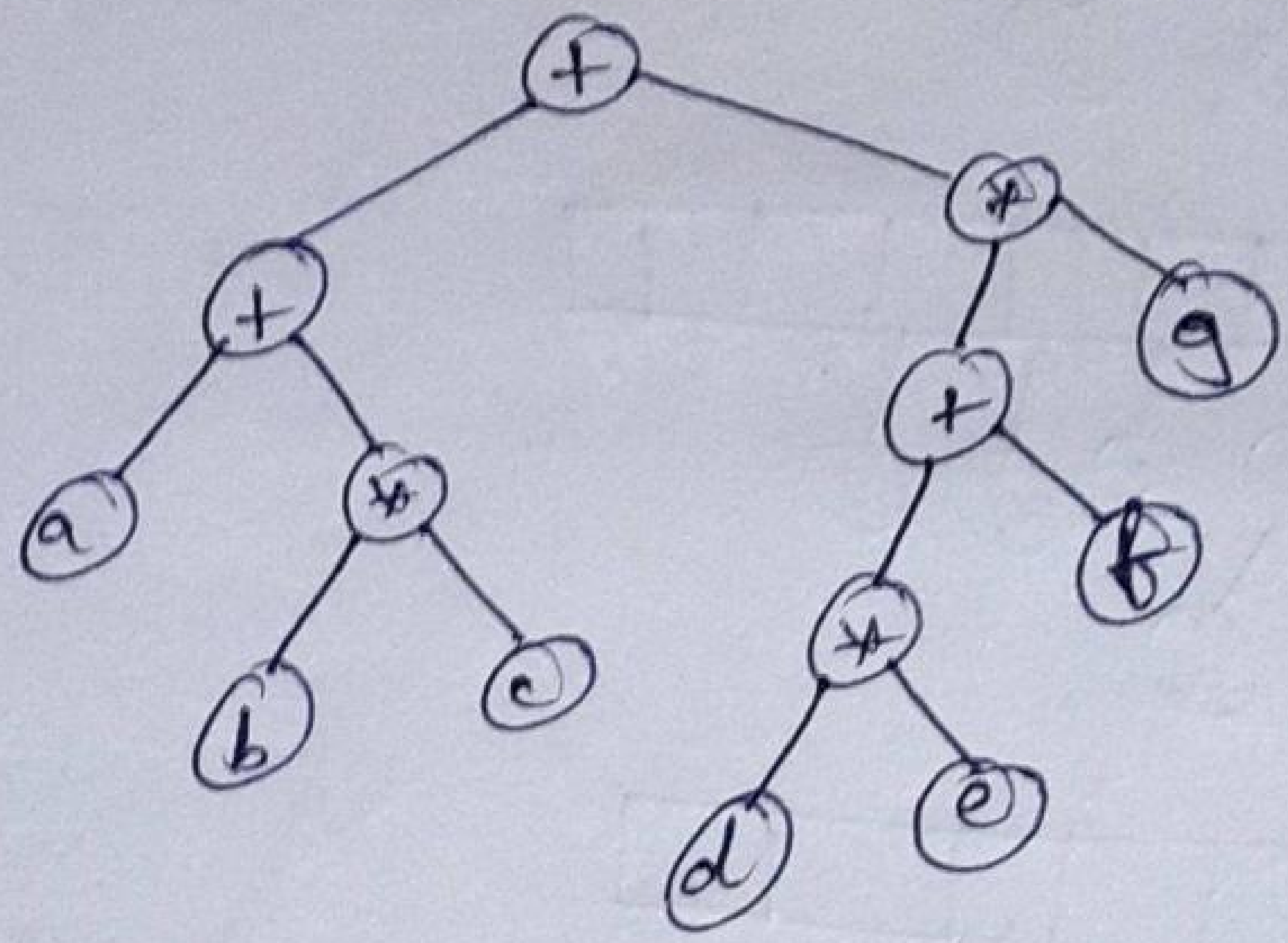




Expression trees:

A expression tree is a tree in which all the leaf nodes contain the operands and the other node contains the operator. Here operands can be any constant or variable. This tree can be a binary tree. Here a root can have more than 2 children. In case of unary operator there will be only one child.

eg. (a + b * c) + ((d * e + f) * g).



Postorder.
abc * + de * f + g * +

Preorder
+ + a * bc * + * defg

Inorder
a+b * c + d * e + f * g

Constructing an expression tree from a postfix expression.

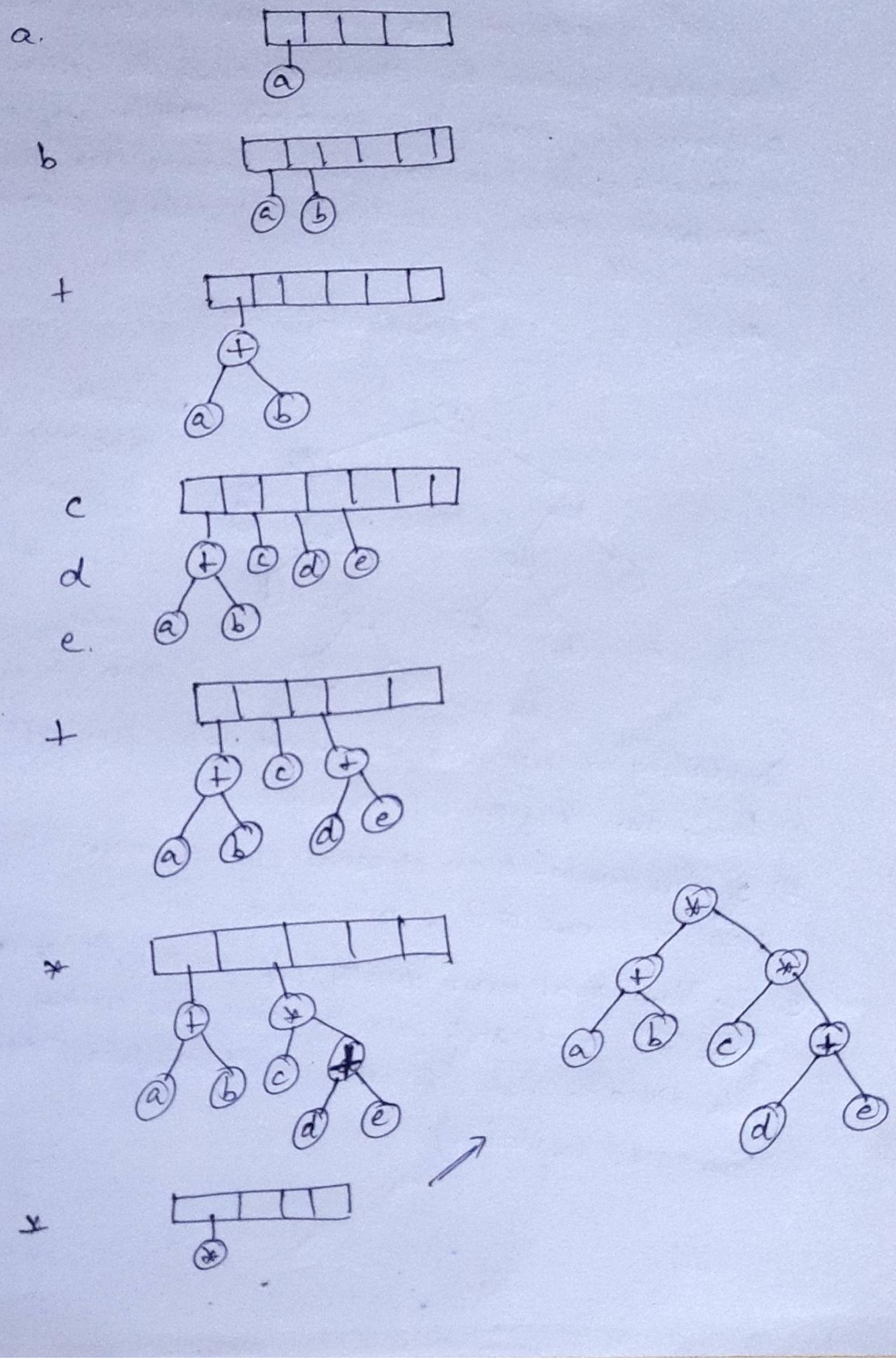
① Read the I/P from left to right

② If the symbol is an operand, create a new node tree and push a pointer to it onto a stack.

③ If the symbol is an operator, pop pointers to 2 trees T_1 & T_2 from the stack and form a new tree whose root is the operator and left and right children point to T_2 & T_1 respectively.

9). ab+cde+**

I/p o/p tree



Applications of Trees.

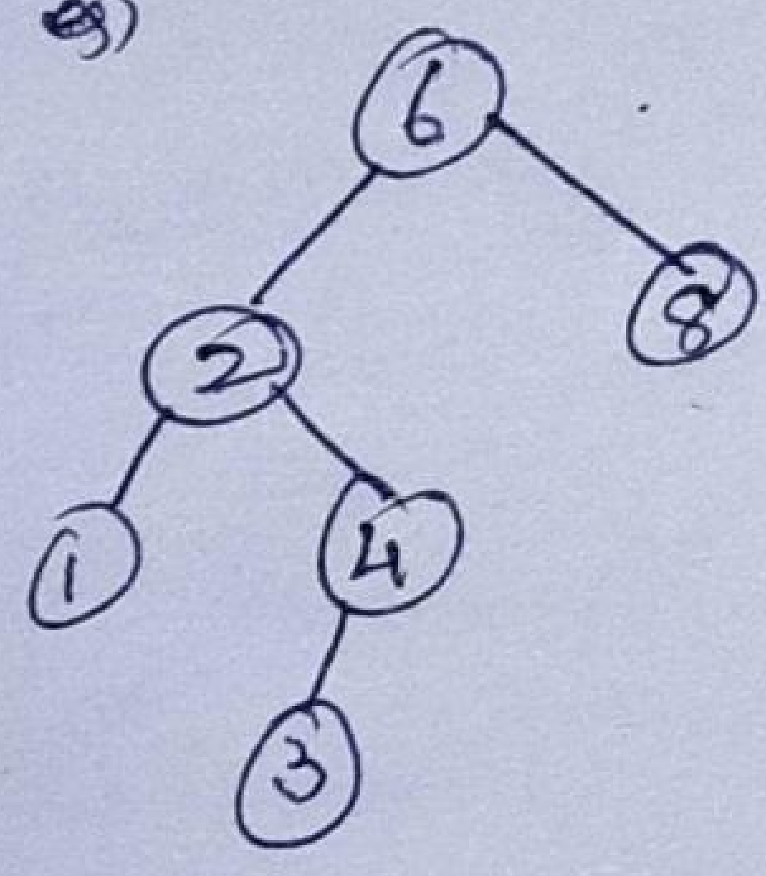
- Expression tree
- Binary search tree.
- Heaps
- Huffman Coding Tree

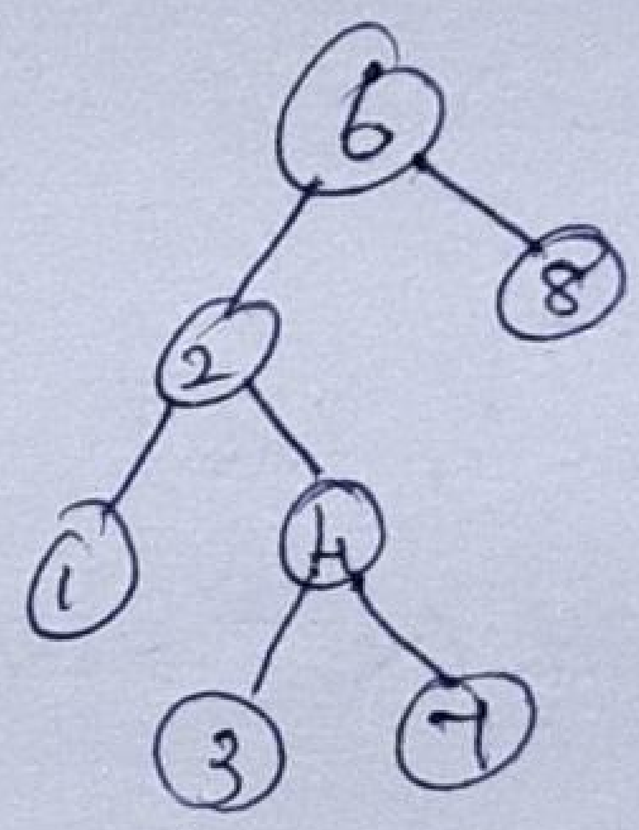
etc.

Binary Search Tree ADT (BST)

A BST is a binary tree which satisfies the following condition, for every node x, the value of all the keys in its left subtree are smaller than the key value in x, and the values of all the key in its right subtree are larger than the key value in x.

eg)





Not a BST
∵ 7 > 6.

Operations on BST

1. Make Empty
2. Find
3. Find Min & Find Max
4. Insert
5. Delete

Make Empty

```
SearchTree MakeEmpty (SearchTree T)
 {
  if (T ! = NULL)
  {
  MakeEmpty (T->Left);
  Make Empty ( T->Right);
  free (T);
  }
 return NULL;
 }
```

Find:

```
Position Find (ElementType X, SearchTree T)
 {
  if (T==NULL)
   return NULL;
  if (X < T-> Element)
   return Find (X, T->Left);
  else
   if ( X > T-> Element)
    return Find (X, T->Right);
  else
   return T;
 }
```

Find Min

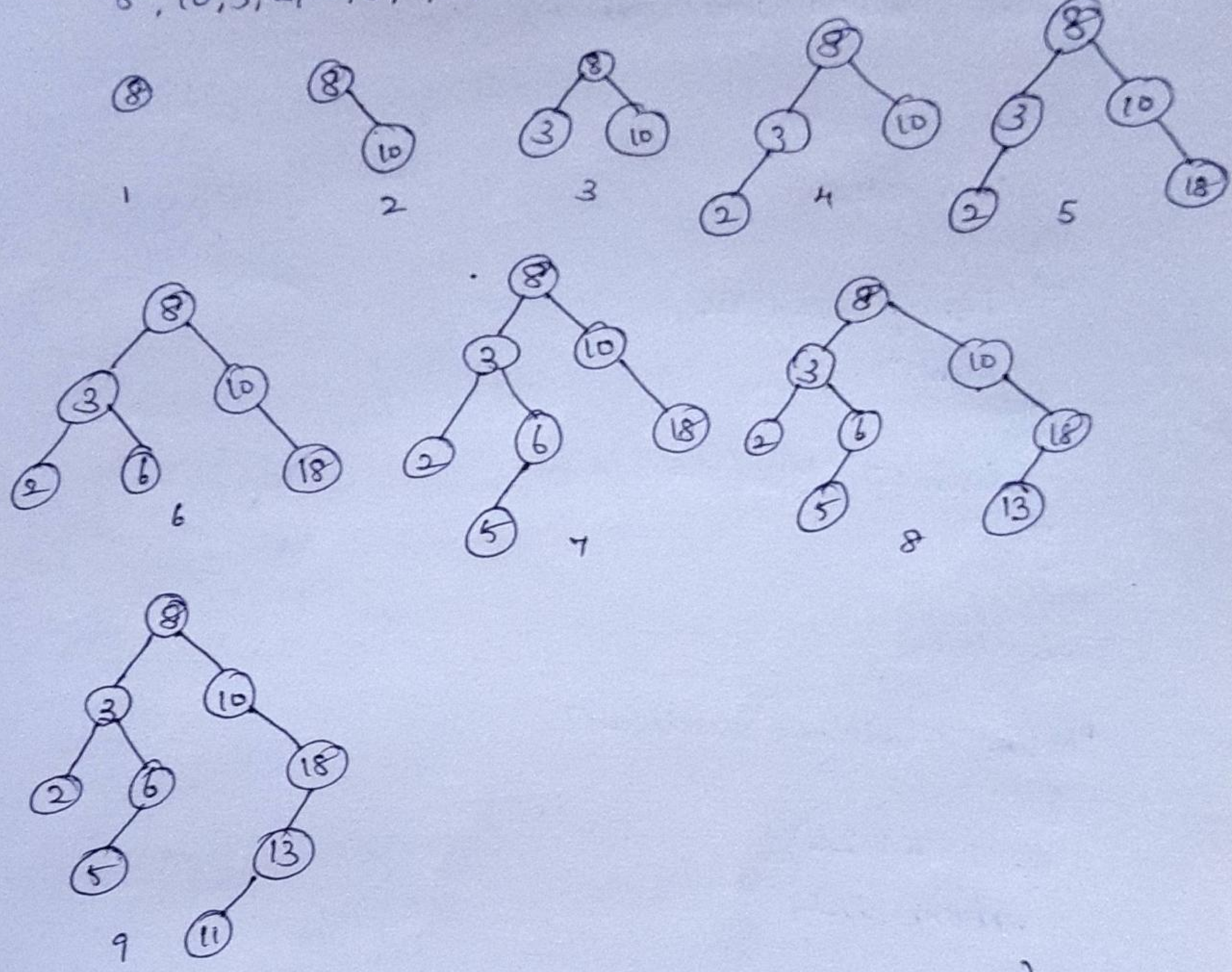
```
Position FindMin(SearchTree T)
{
 if (T == NULL)
   return NULL;
 else
  if (T->Left == NULL)
    return T;
  else
   return FindMin (T->Left);
 }
```

Find Max.

```
Position FindMax (SearchTree T)
 {
  if (T == NULL)
   return NULL;
  else
  if (T->Right == NULL)
    return T;
  else
  return FindMax (T->Right);
  }
```

Insertion.

8, 10, 3, 2, 18, 6, 5, 13, 11




```
SearchTree Insert (ElementType X, SearchTree T)
{
if (T==NULL)
 {
 T=malloc (Sizeof (Struct TreeNode));
 if (T==NULL)
   Error.
 else
  {
  T→Element =X;
  T→Left =NULL;
  T→Right = NULL;
  }
```

Scanned by TapScanner

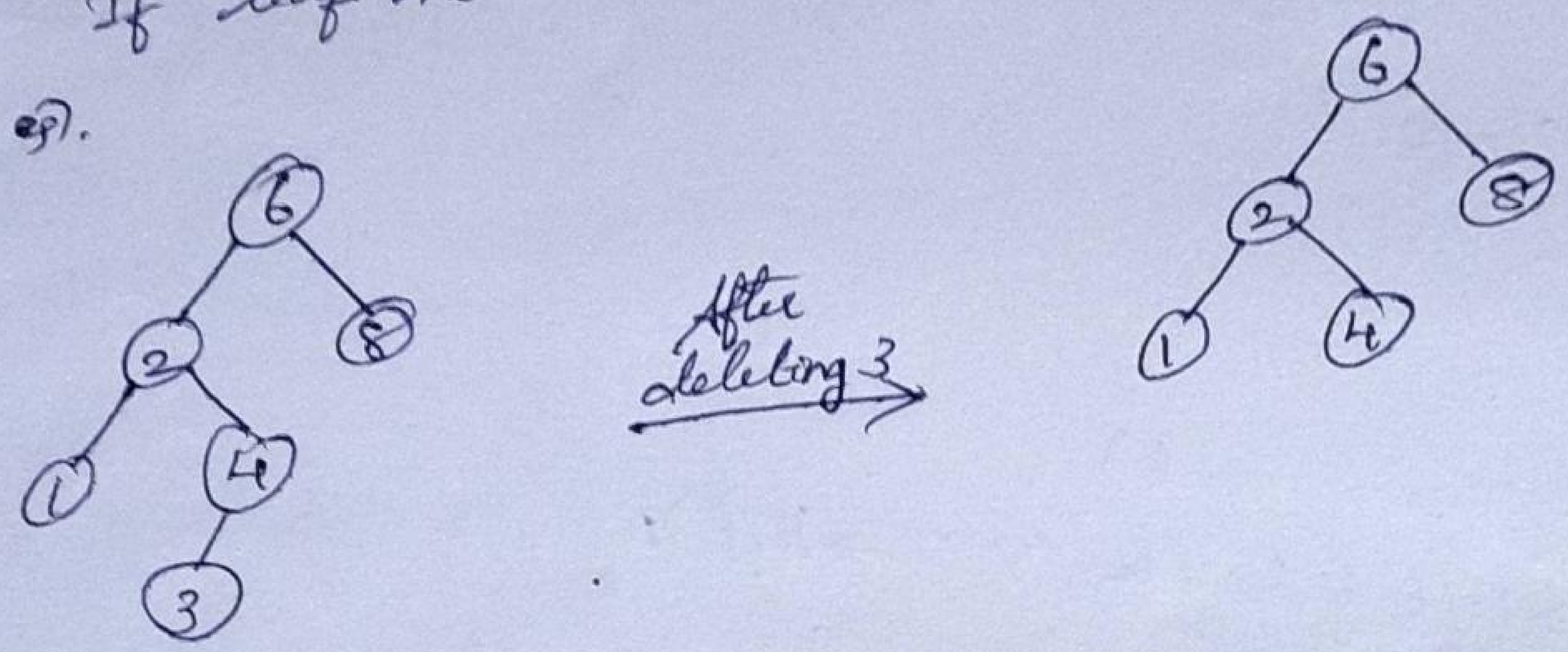
```
else
  if (X > T->Element)
     T->Right = Insert (X, T->Right);
  if (X < T->Element)
     T->Left = Insert (X, T->Left);

  return T;
 }
}
```

Deletion :

case 1 : Node with no children.

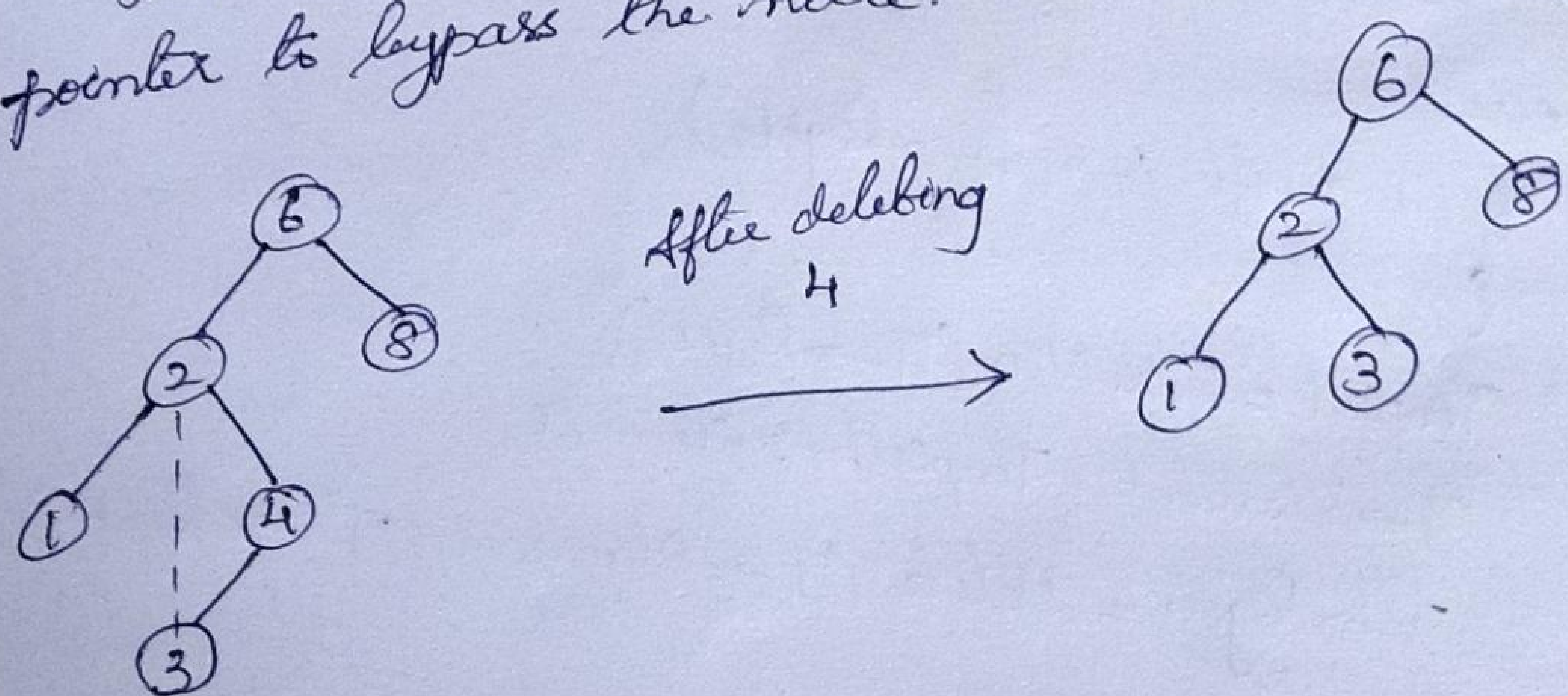
If leaf node delete it immediately.

eg.



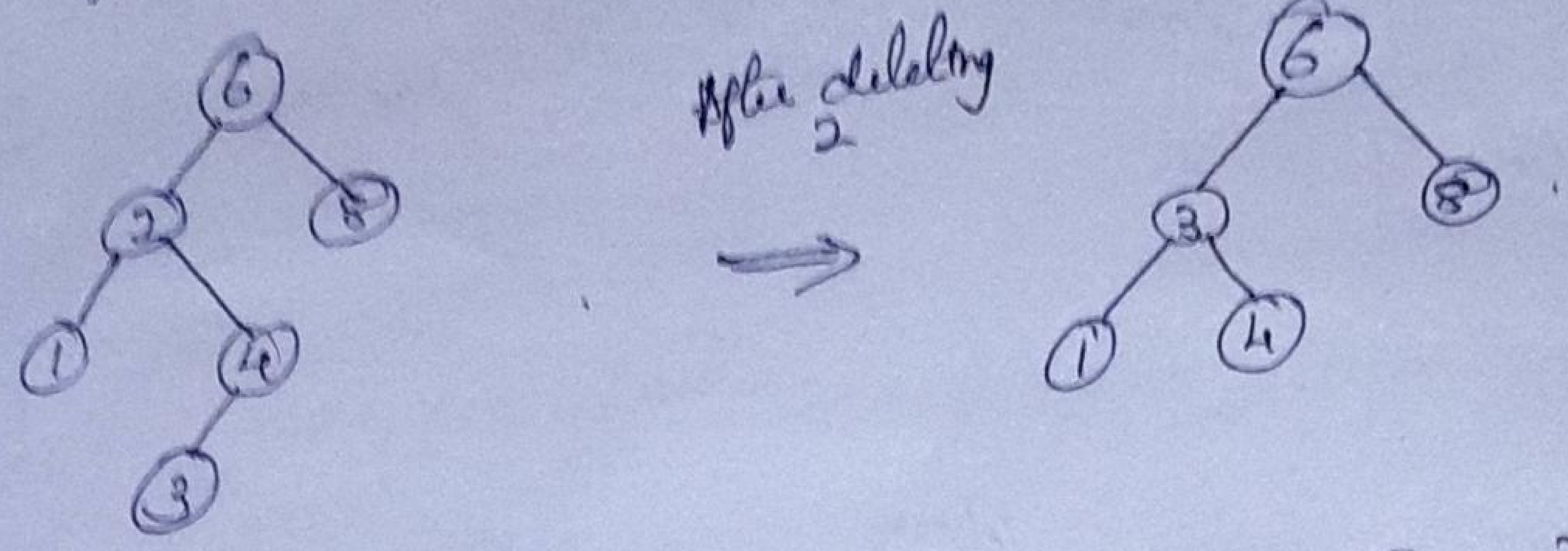
case 2 : Node with one child.

The node can be deleted after its parent adjusts a pointer to bypass the node.



case 3: Node with 2 children.

Find node with smaller value in the right subtree
or find node with bigger value in the left subtree.
Replace the node in place of deleted node




```
SearchTree Delete (ElementType X, SearchTree T)
{
Position Tmpcell;
 if (T == NULL)
    Error
 else
  if (X < T->Element)
    T->Left = Delete (X, T->Left);

  else
   if (X > T->Element)
    T->Right = Delete (X, T->Right)

   else
   if (T->Left && T->Right)
    {
    Tmpcell = Find Min (T->Right);
    T->Element = Tmpcell->Element;
    T->Right = Delete (T->Element, T->Right);
    }
```

```
else
{
 Tmpcell = T;
  if (T -> Left == NULL)
      T = T -> Right;
 else
   if (T -> Right == NULL)
      T = T -> Left;
   free (Tmpcell);
 }
 return T;
}
```

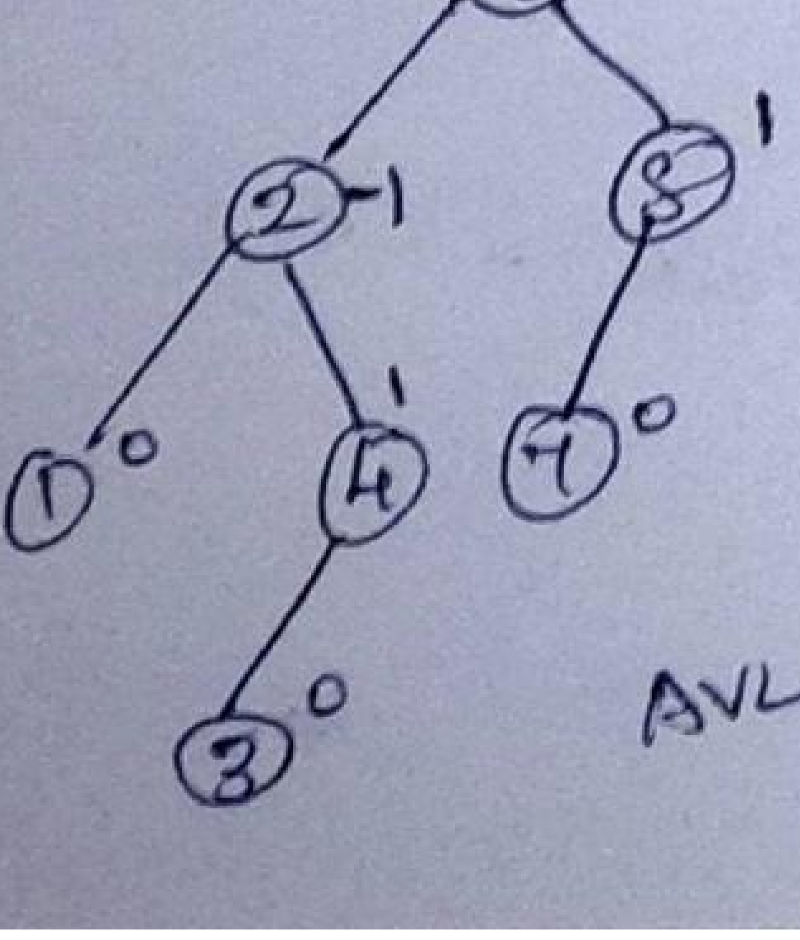
AVL Trees: (Adelson - Velskii & Landis)

- AVL is a binary search tree with a balance condition.
- A balance condition must be easy to maintain & it ensures that the depth of the tree is O(log n).
- The height of the left and right subtrees can differ by atmost 1.

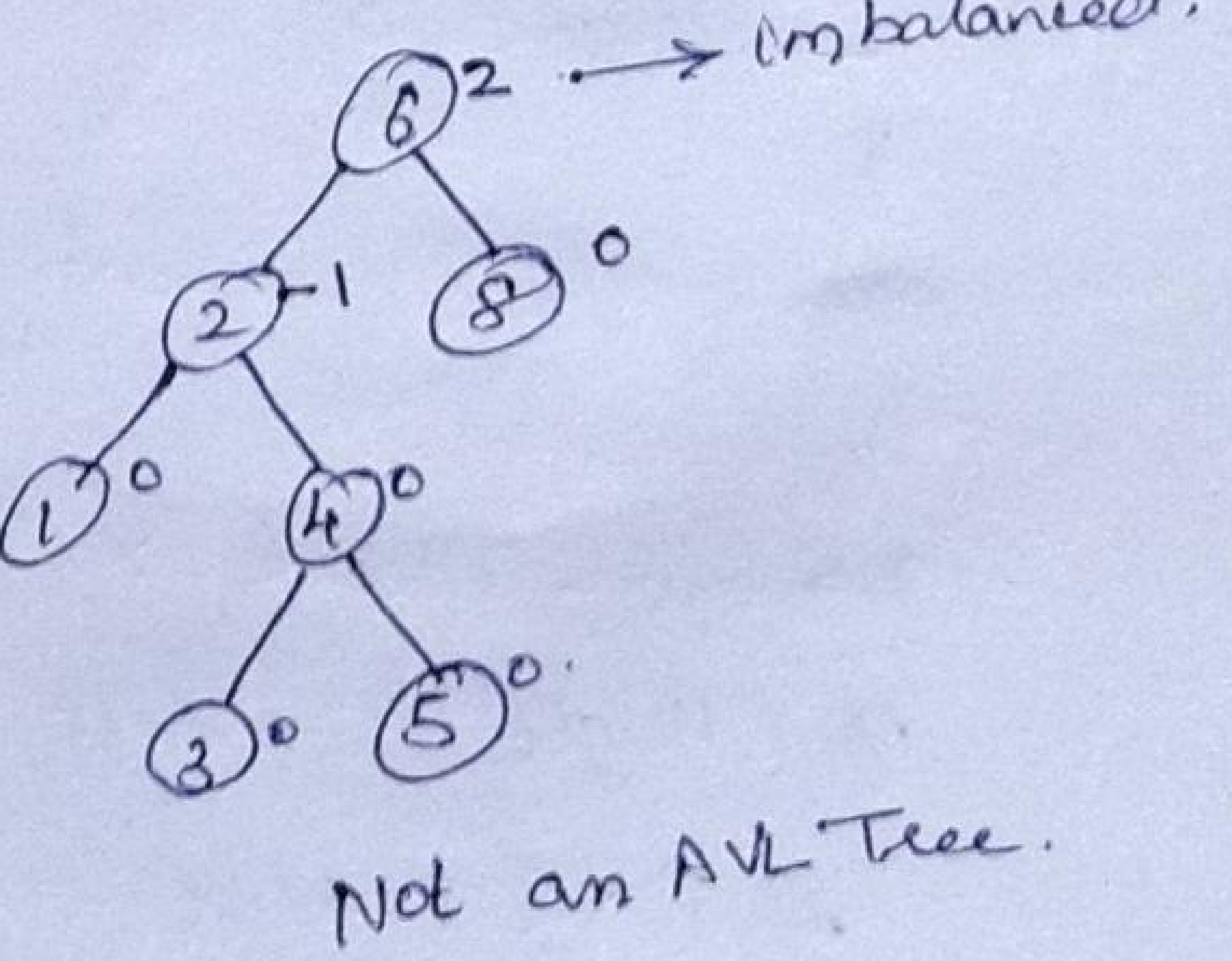
Balance factor = height of LST − height of RST.

- For an AVL tree all balance factor should be +1, 0 or −1.

eg.



AVL Tree



Not an AVL Tree.

Insertion:

An AVL tree causes imbalance when any one of the following conditions occurs.

① An insertion into the left subtree of the left child of node α.

② An insertion into the right subtree of the left child of node α.

③ An insertion into the left subtree of the right child of node α.

④ An insertion into the right subtree of the right child of node α.

Insertion:

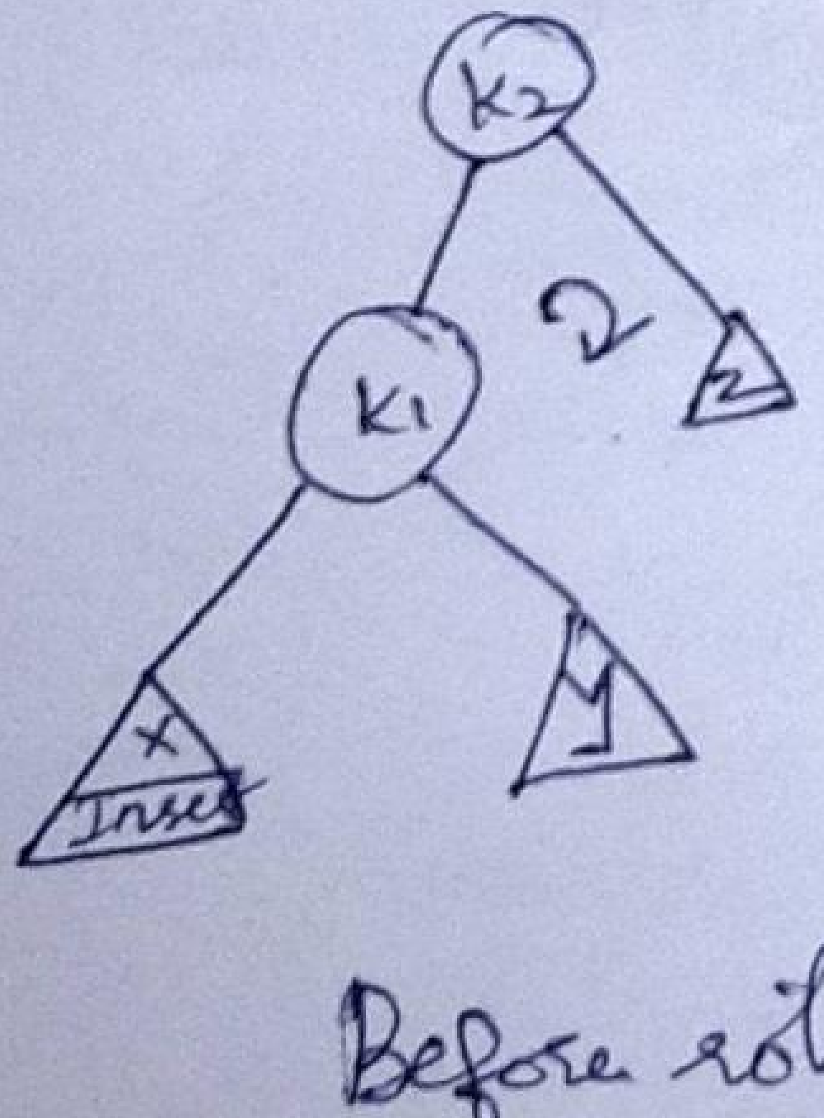
```
Insert (X,T)
{
if (T== NULL)
{
T= malloc (Sizeof (Struct Node));
T→root = x;
T→Left = T→Right = NULL;
T→Height =0;
}
else if (X<T→root )
{
T→left = Insert (X, T→Left);
if ( Height (T→ Left ) - Height (T→Right ) = =2 )
  if ( X< T→Left →root )
    T= SingleRightRotation(T);
```

```
else
 T = DLRRotation (T);
}
else if (x > T -> root)
 {
 T -> Right = Insert (x, T -> Right);
  if (Height (T -> Left) - Height (T -> Right) == 2)
   if (x > T -> Right -> Root;
    T = SingleLeftRotation (T);
   else
     T = DRLRotation (T);
   }
  T -> Height = Max (Height (T -> left), Height (T -> Right)) + 1;
  return T;
}
```

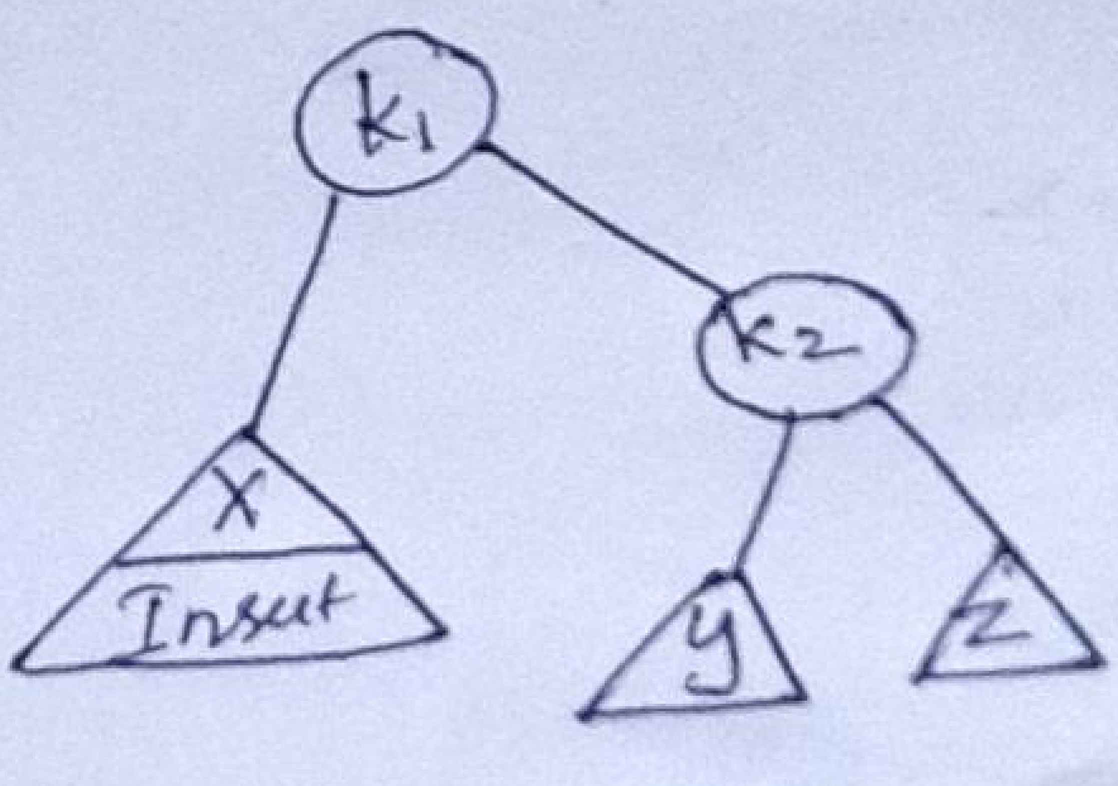
Single Rotation:

Single rotation is performed to fix case 1 & 4

case 1:



Before rotation.

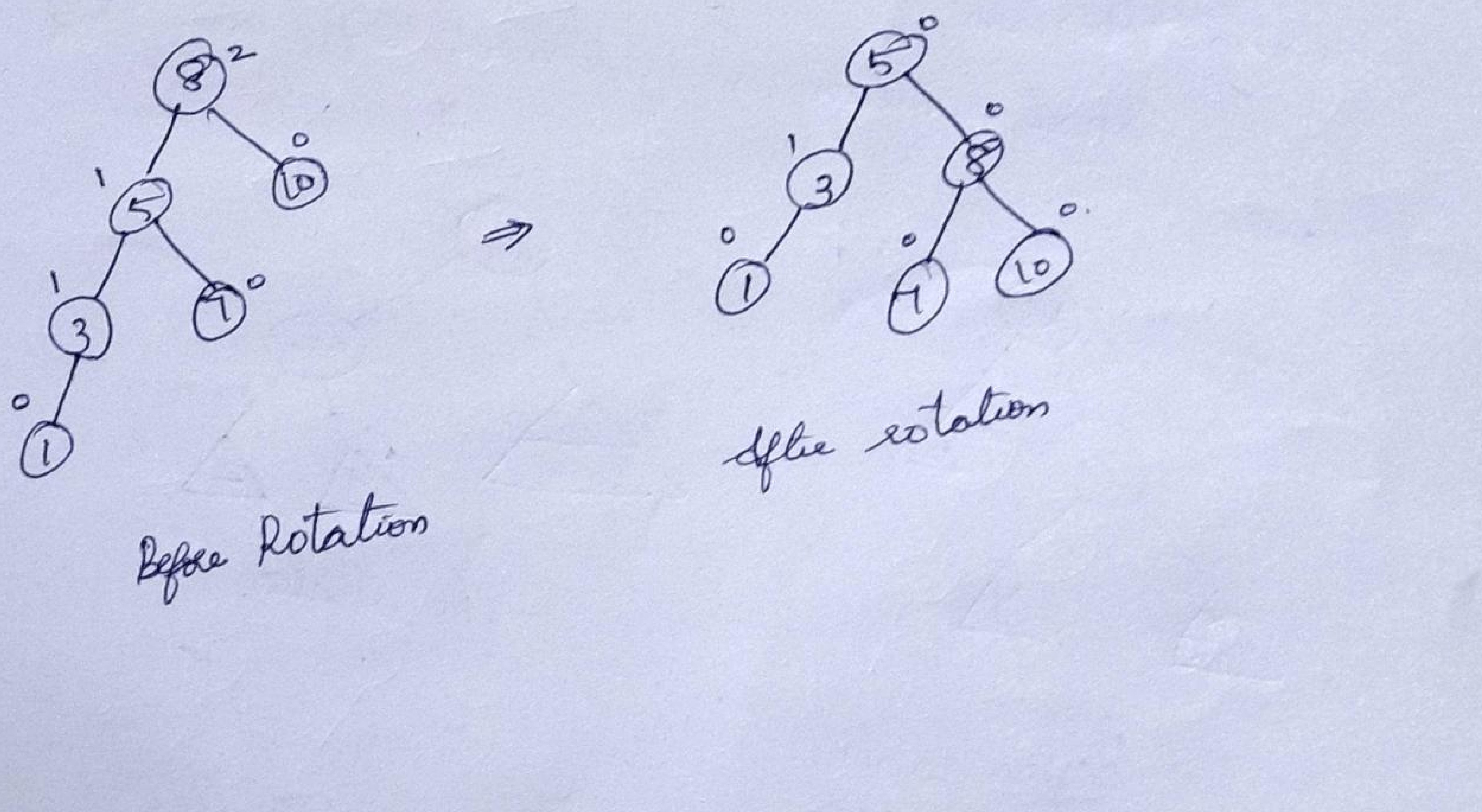


After rotation

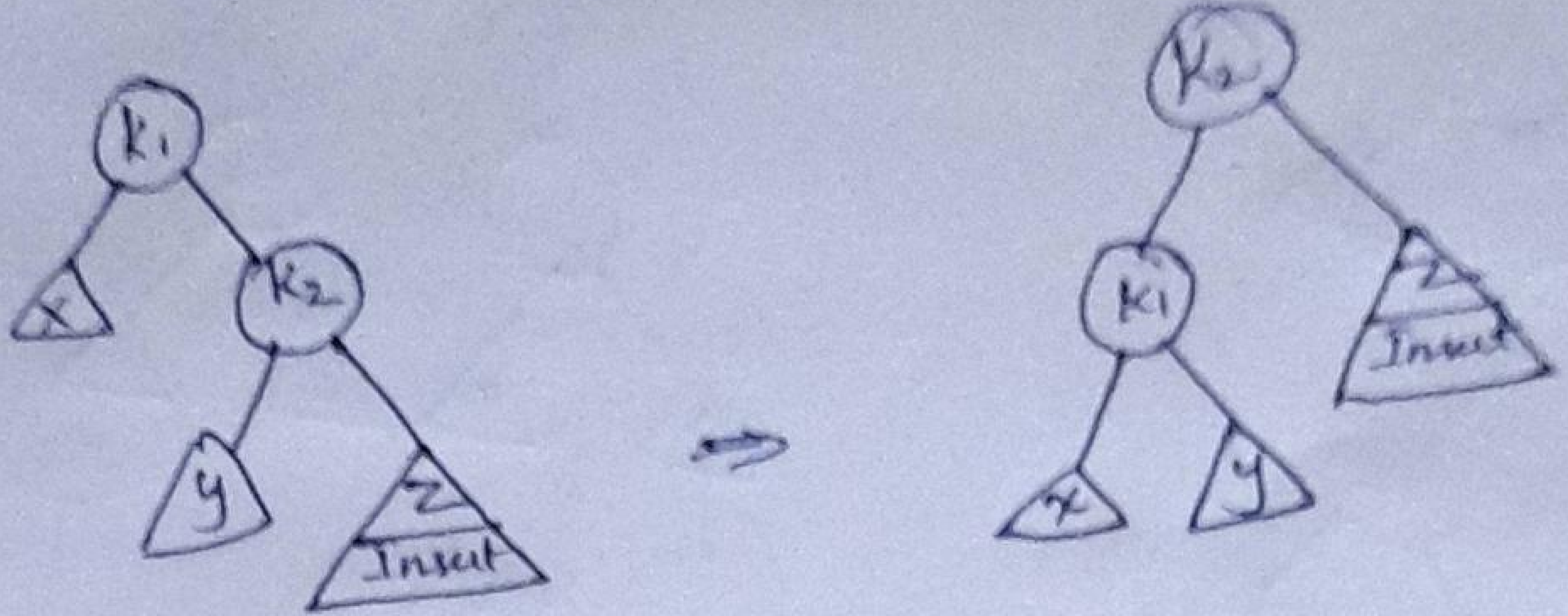
Routine to perform single rotation with left.

```
SingleRotationwithLeft ( Position k2)
{
Position k1;
 k1 = k2 -> Left;
 k2 -> Left = k1 -> Right
 k1 -> Right = k2;
 k2 -> Height = Max (Height (k2 -> Left ), Height (k2 -> Right ))+1;
 k1 -> Height = Max ( Height ( k1 -> Left), Height (k1 -> Right ))+1;
 return k;
}
```

Q) Inserting the value 1 in the following AVL Tree.

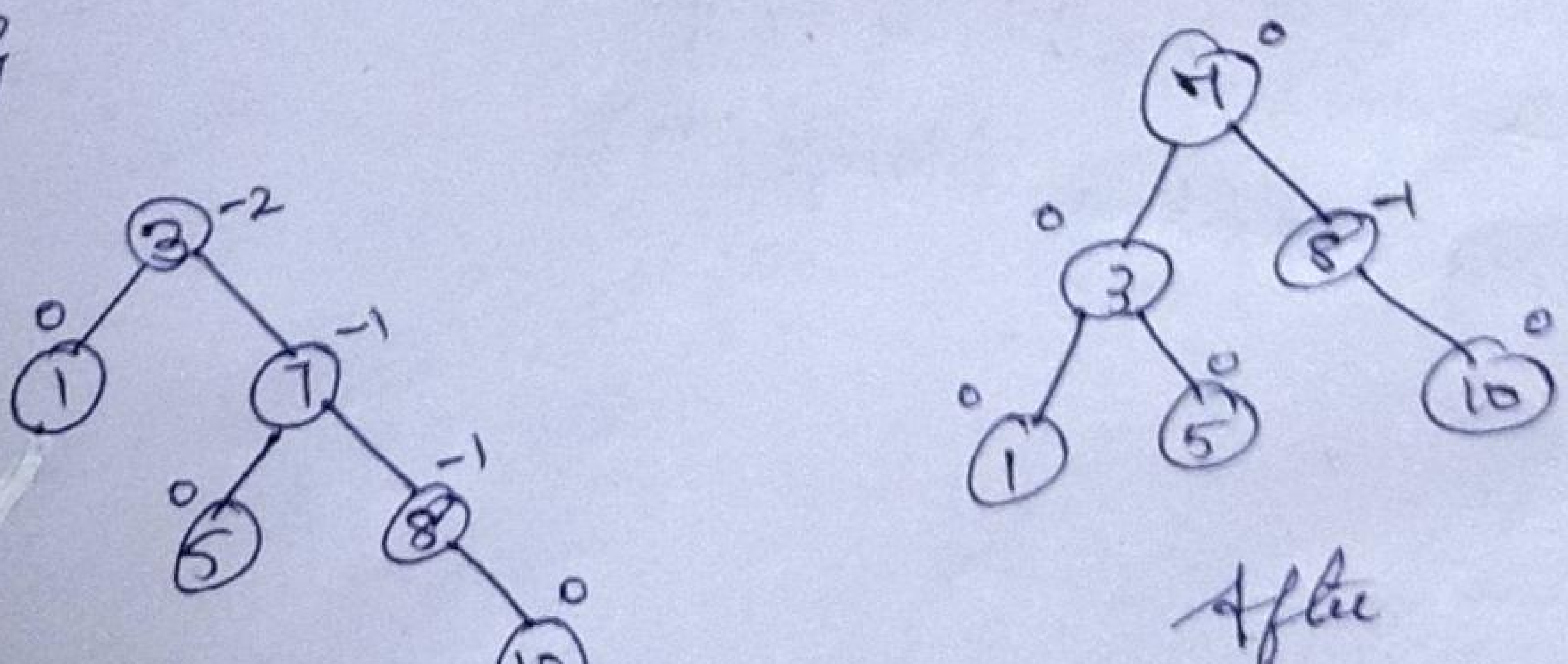


Single Rotation to fix case 4:
Right ST of right child of K_1



Routine

```
SingleRotationWithRight (Position K1)
{
position K2;
 K2 = K1 -> Right
 K1 -> Right = K2 -> Left
 K2 -> Left = K1;
 K2 -> Height = Max (Height (K2 -> Left), Height (K2 -> Right))+1;
 K1 -> Height = Max (Height (K1 -> Left), Height (K1 -> Right))+1;
 Return K2;
}
```

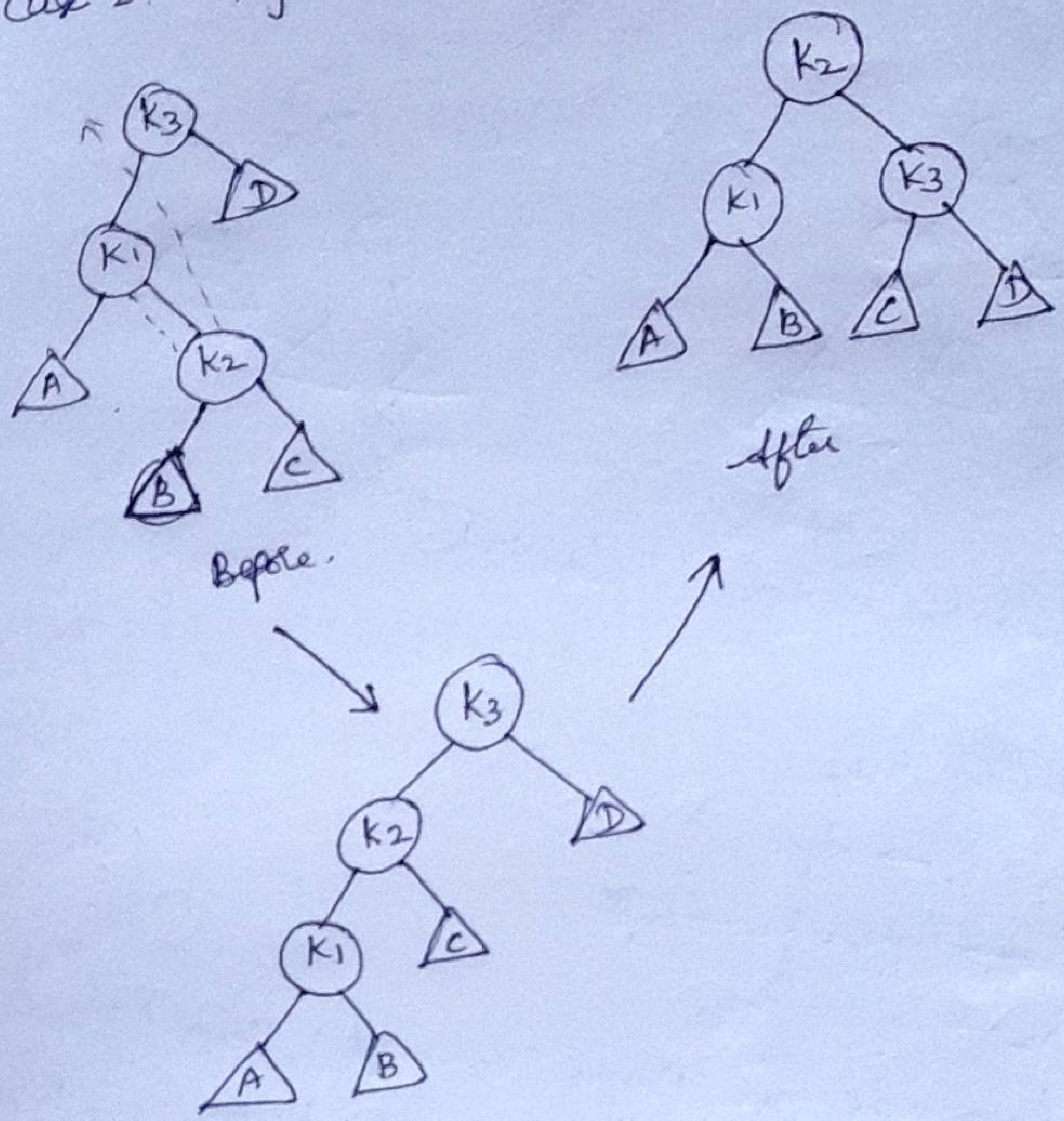


Before

After

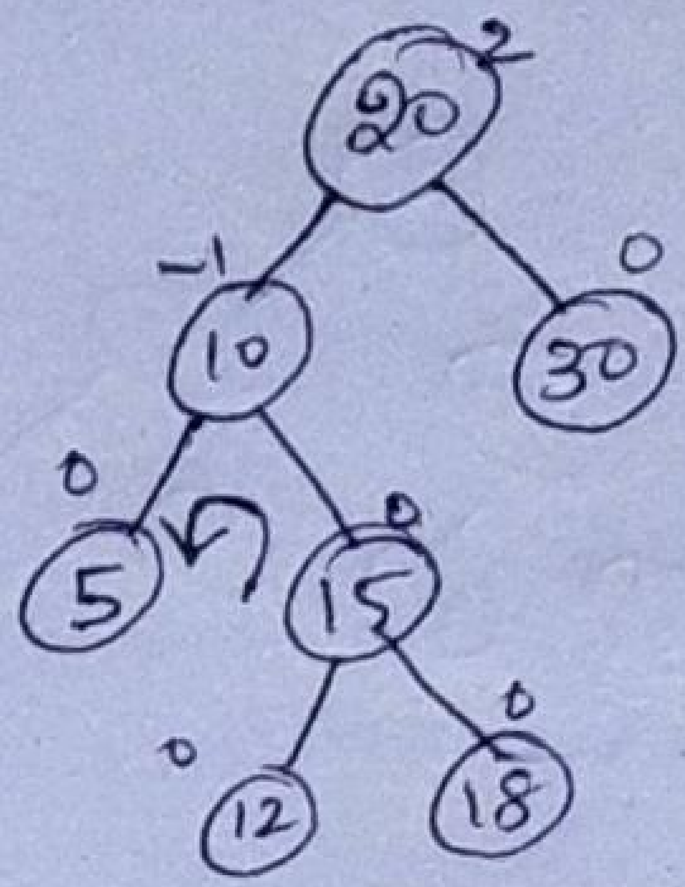
Double Rotation:

Case 2: Right ST of Left Child.

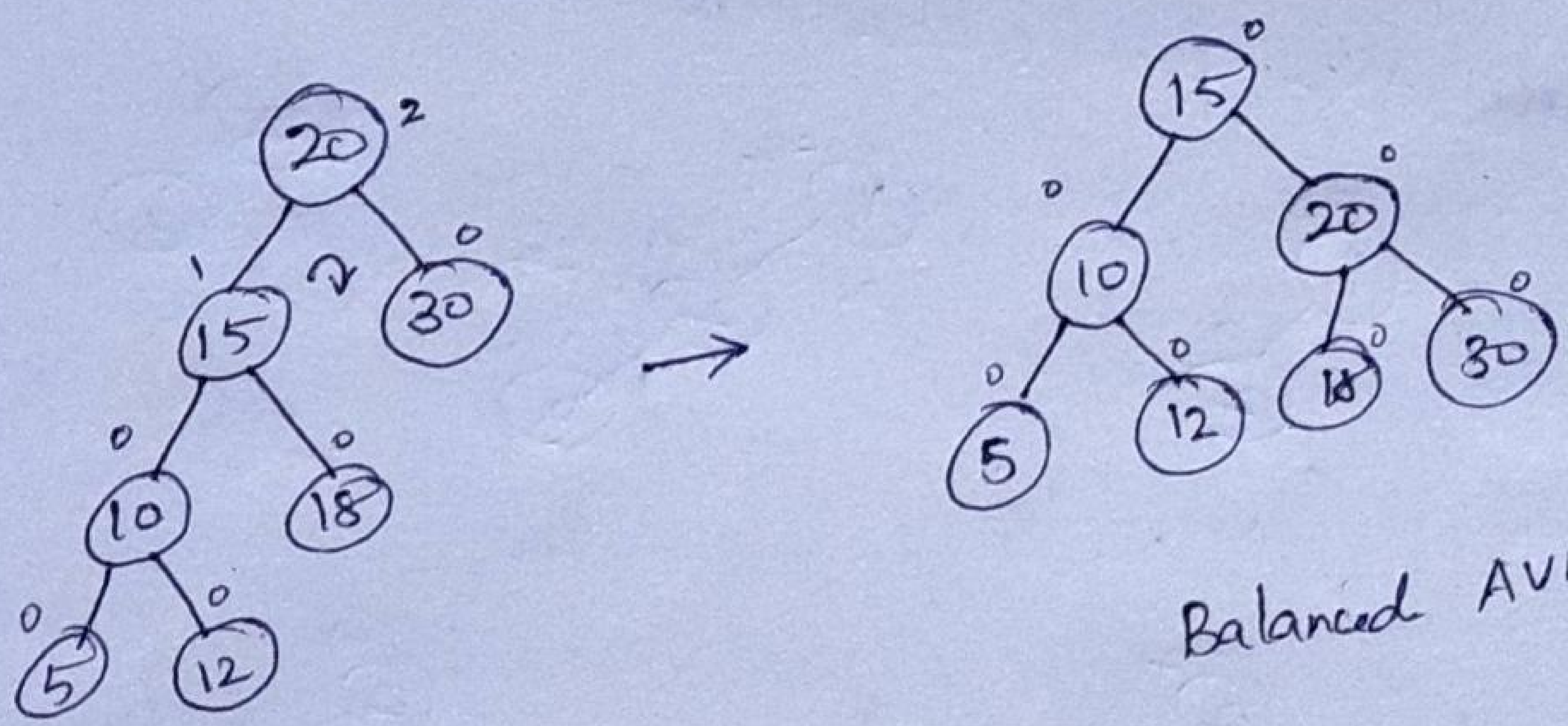


D Rotation with left.

```
DoubleRotation with left (Position K3)
{
  K3 -> Left = SingleRotateWithRight (K3 -> Left);
  Return SingleRotateWithLeft (K3);
}
```

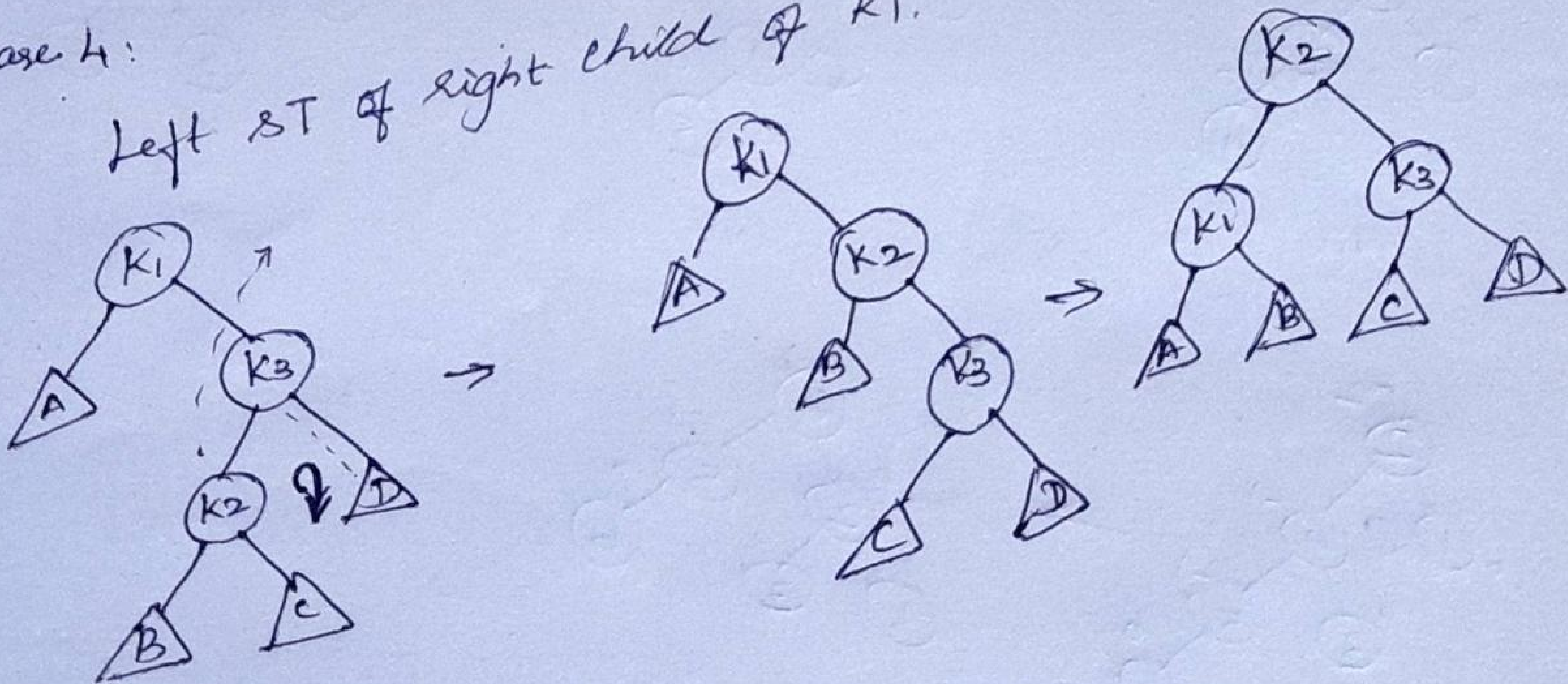


Imbalance is due to (12) or (18) nodes.
This can be done by performing single rotation with right of 10 & then perform the single rotation with left of 20 as,



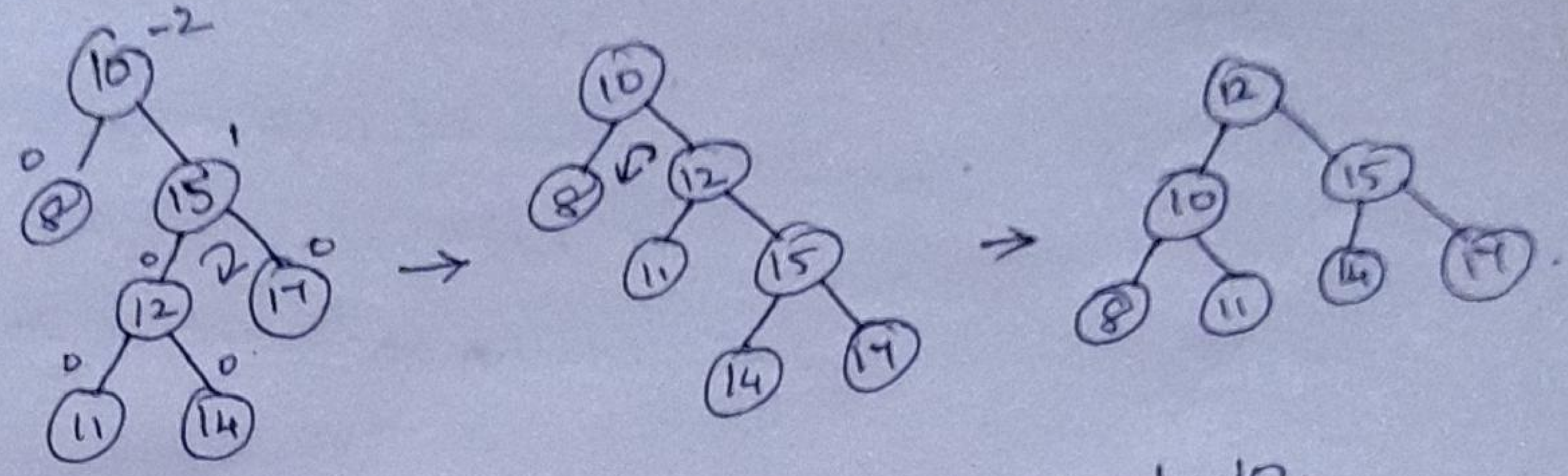
Balanced AVL

Case 4:
Left ST of right child of K1.

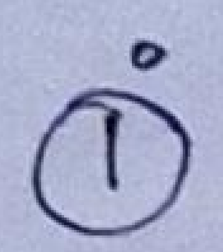


Double Rotation with Right.

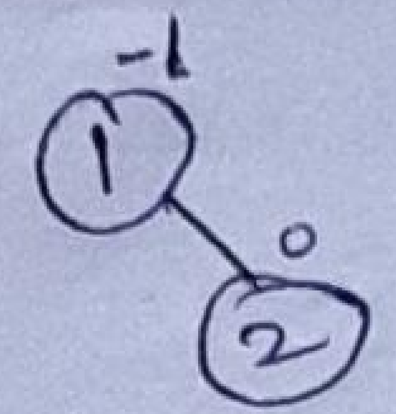
```
DoubleRotateWithRight ( Position K1)
{
K1 -> Right = SingleRotateWithLeft (K1 -> Right);
return SingleRotateWithRight (K1);
}
```



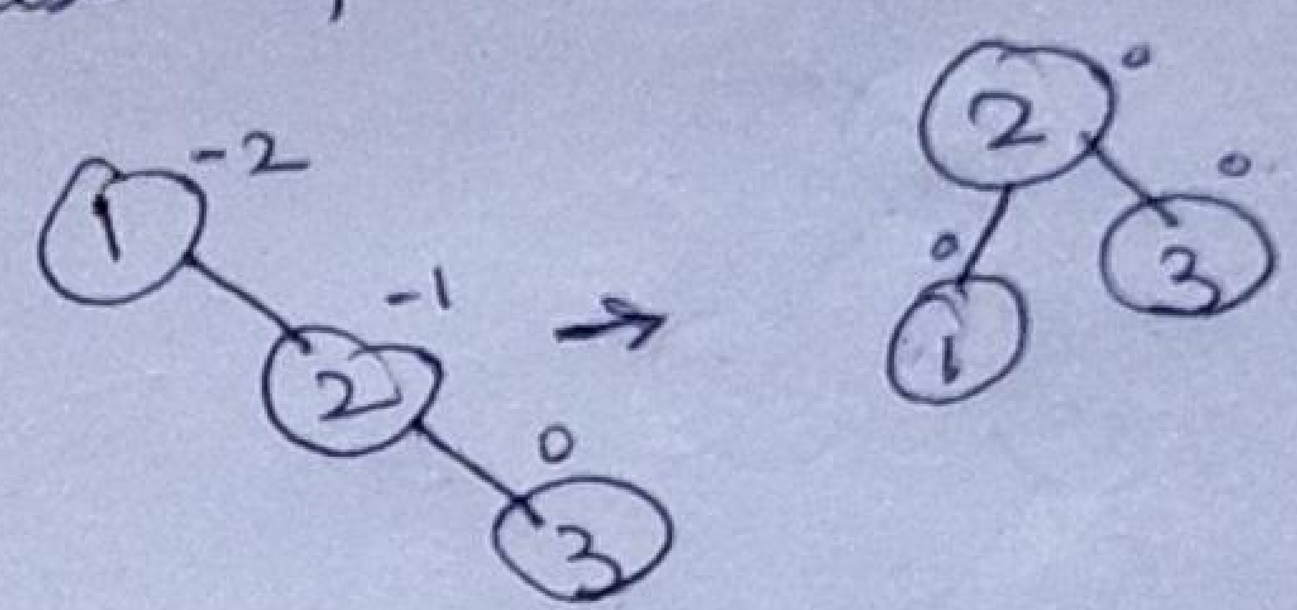
example: Create AVL tree whose input are 1–10.



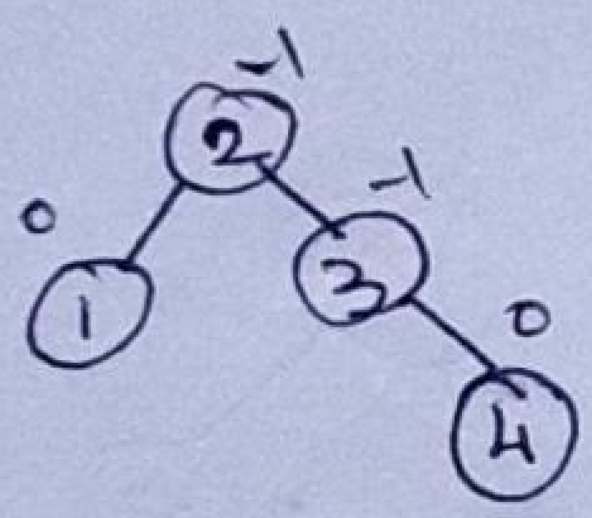
(i)



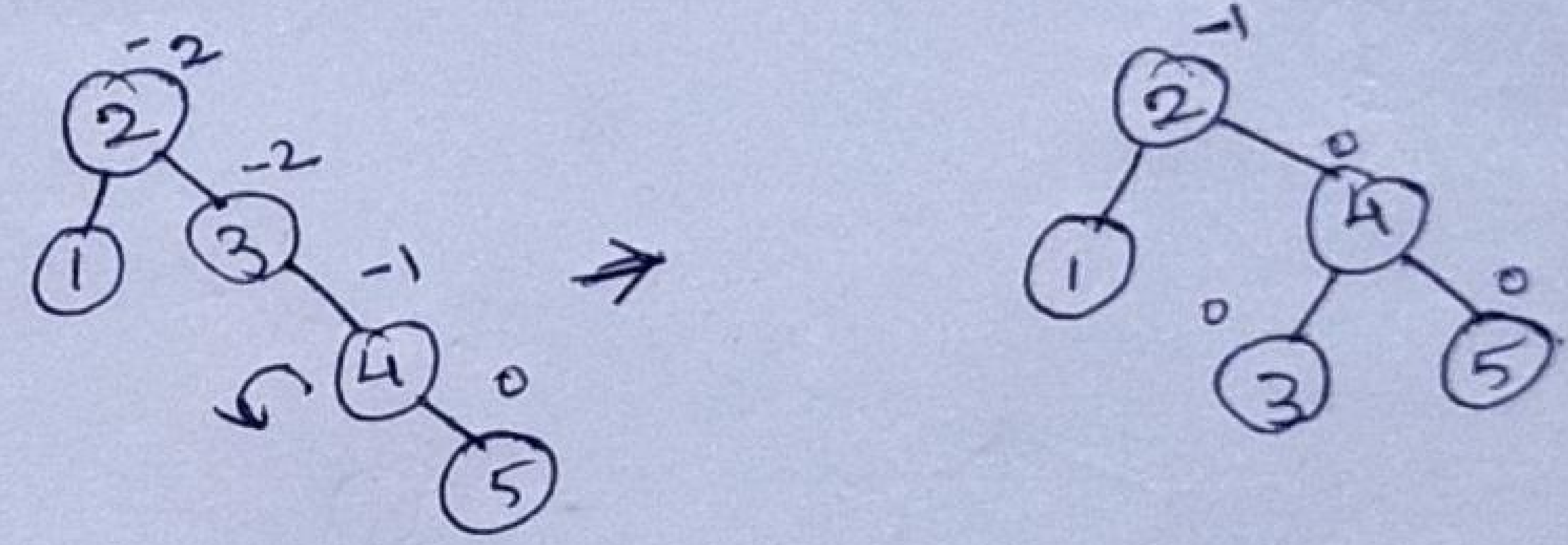
(ii)



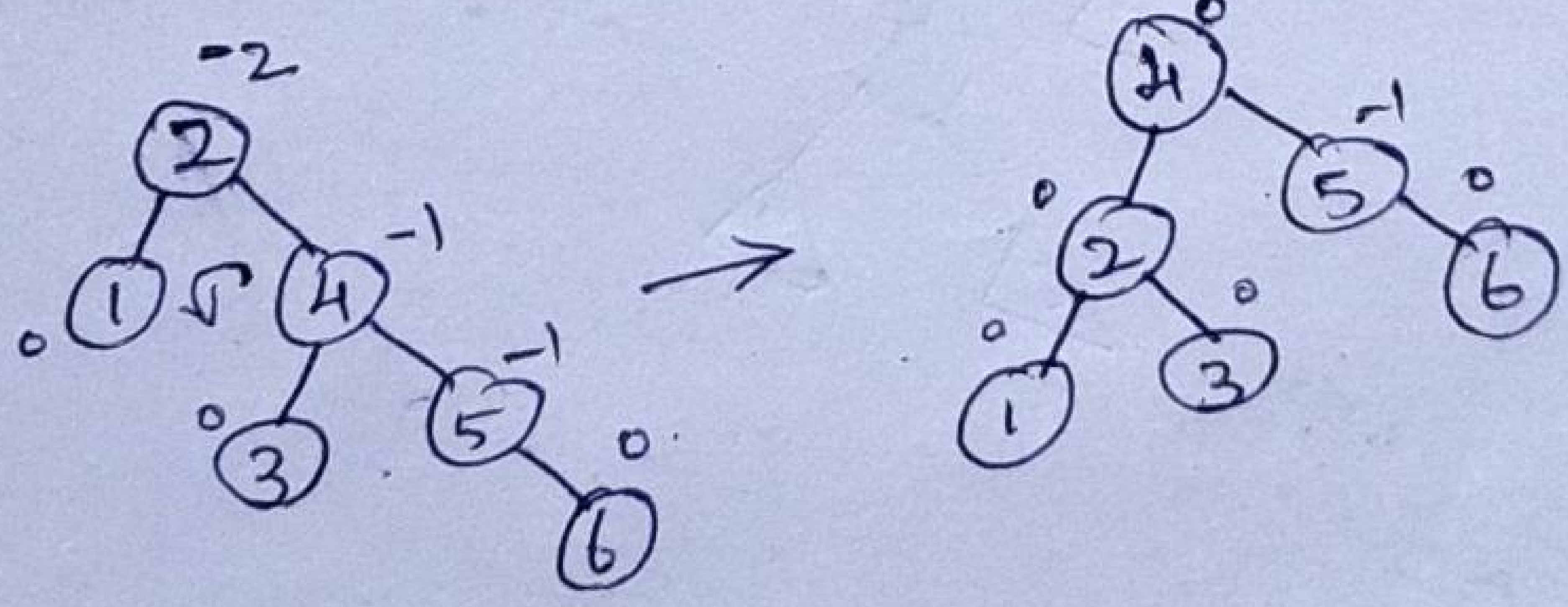
(iii)



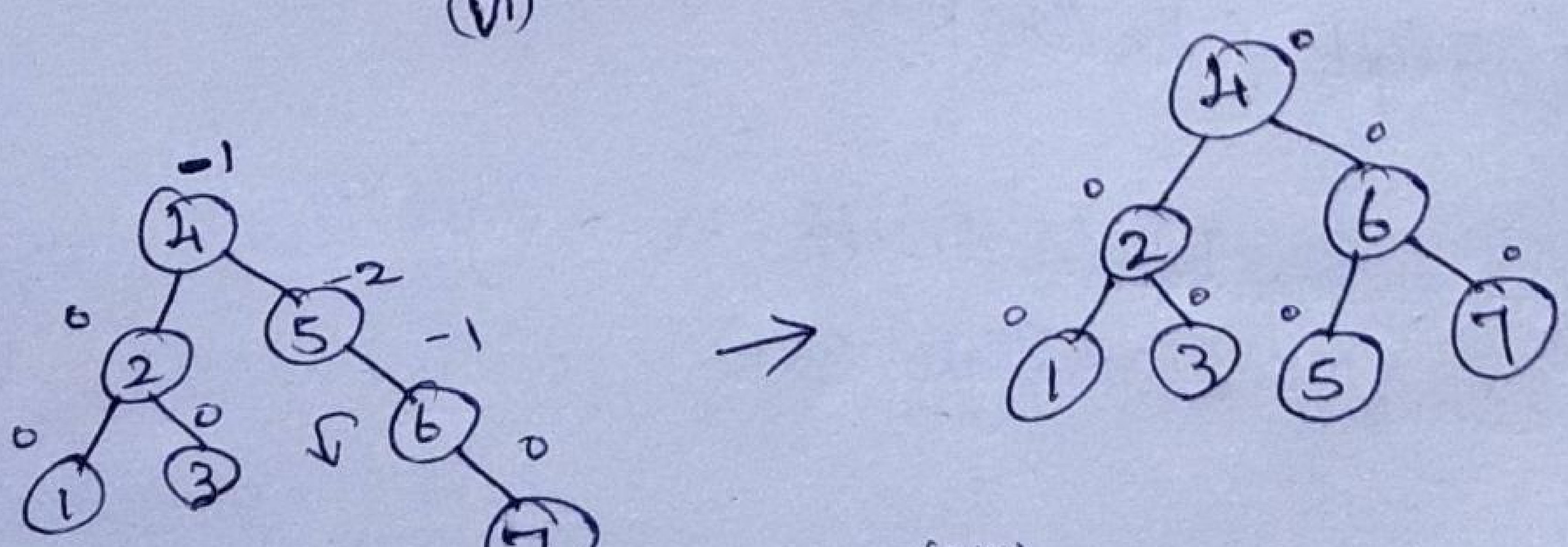
(iv)



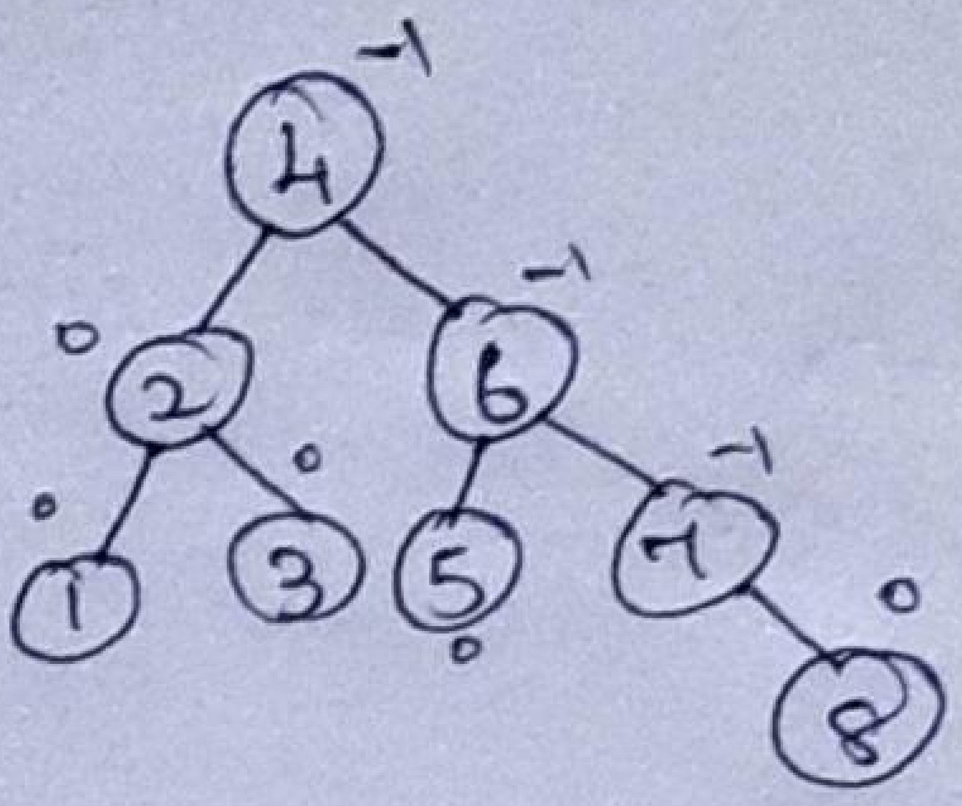
(v)



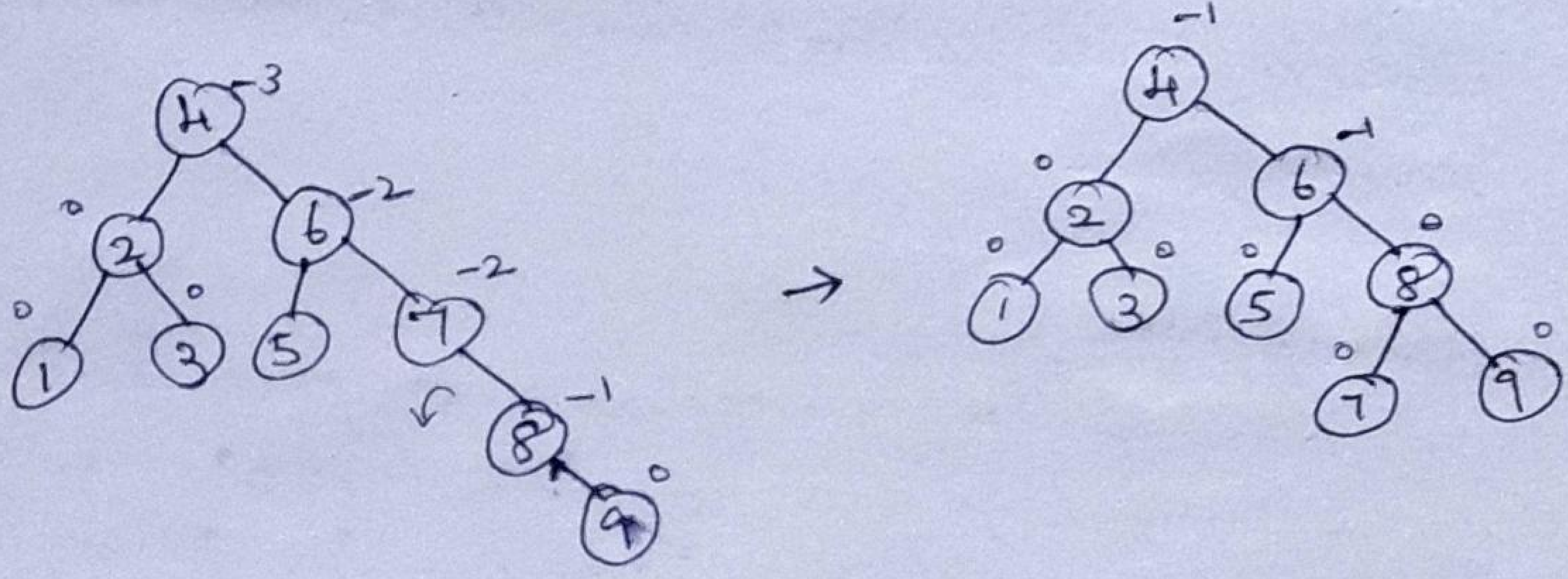
(vi)



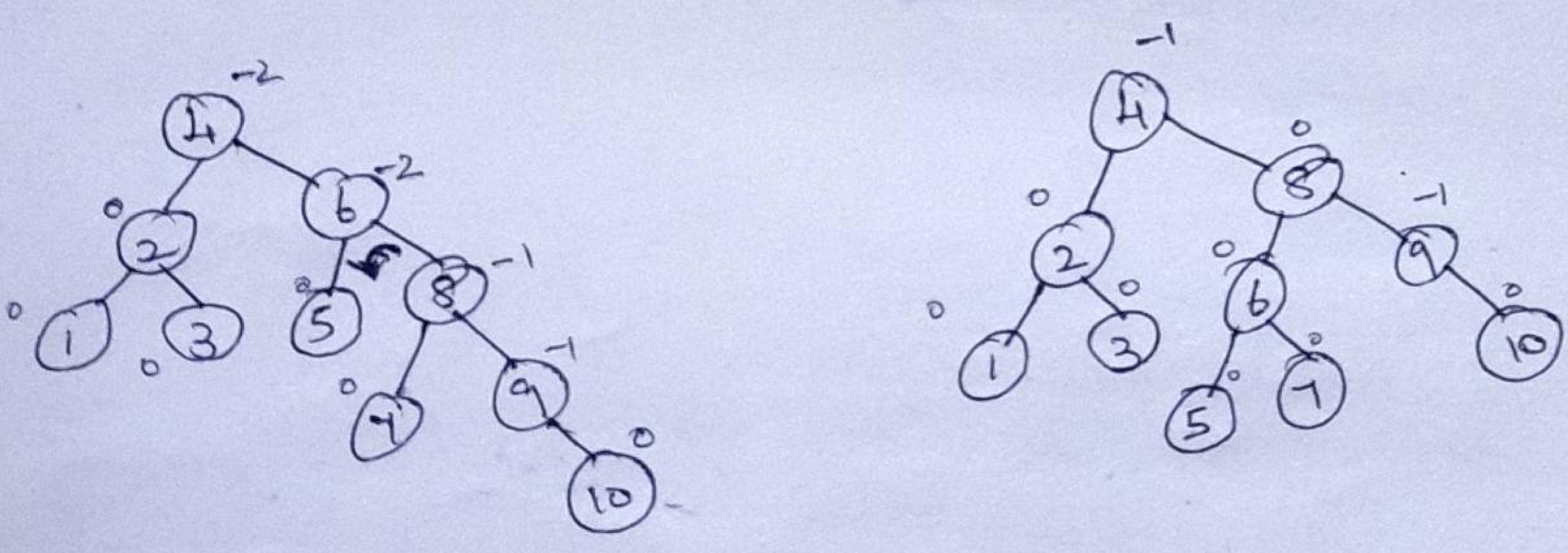
(vii)



(viii)



(ix)



(x)

Priority Queue.

A priority queue is a data structure that allows 2 operations, insert and delete min.

Insert - insert an element in the heap.

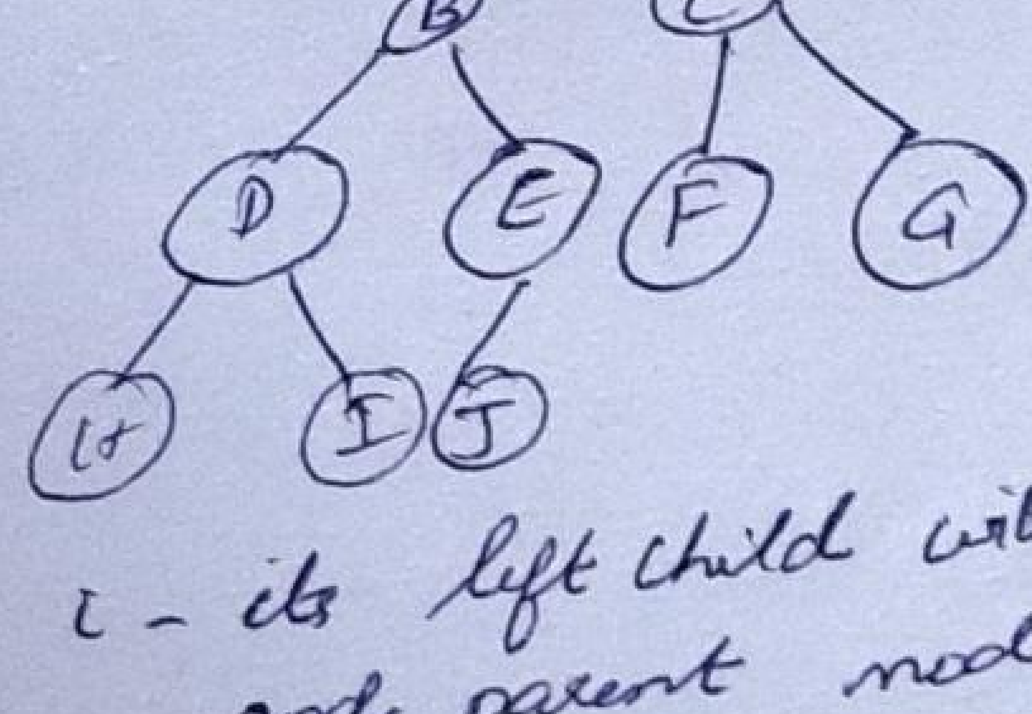
Deletemin. Finds, returns & removes the minimum element in the queue.

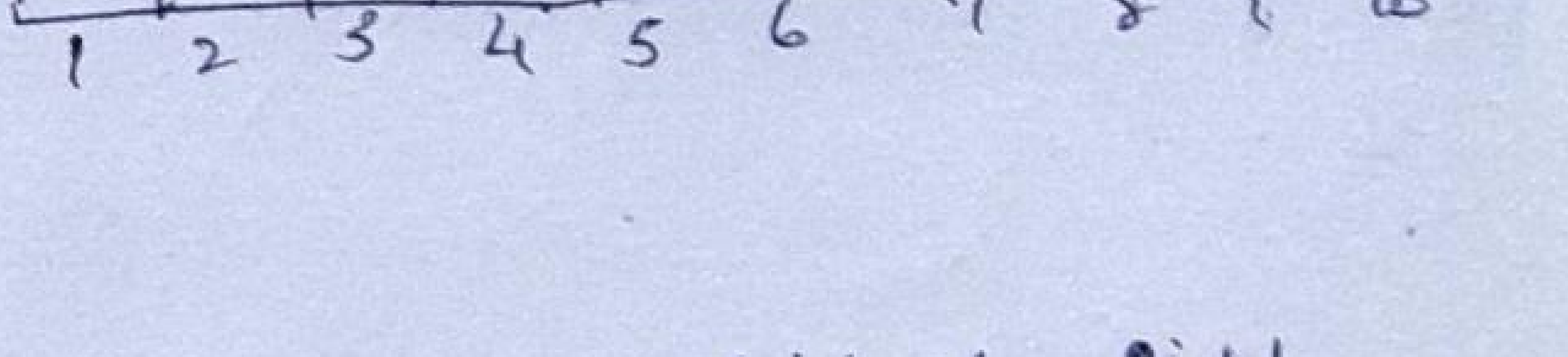
Binary Heap:

- One of the implementation of priority queue.
- Binary heap is a complete binary tree in which every node satisfies the 2 heap order property.
  - structure property - complete binary tree.
  - heap order property.

structure property.

- complete binary tree
- Array representation
- filled from left to right
- No. of nodes will be between 2^h & $2^{h+1}-1$.
  h - is the height of the tree.





For i - its left child will be at 2i & right child at 2i+1 and parent node will be at i/2.

Heap order property.

- For every node x, the key in the parent of x is smaller than the key in x, with the exception of the root.

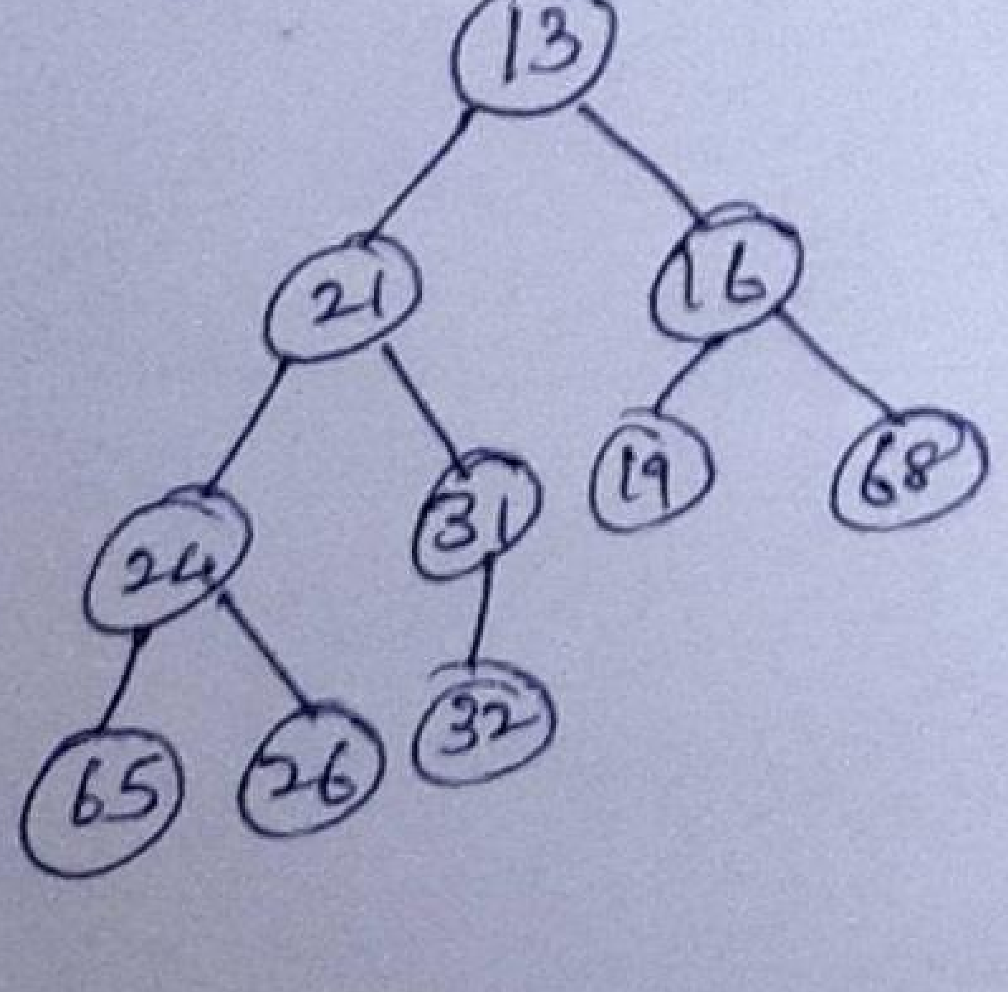
$$A[parent(x)] \leq A[x]$$

- Minimum element should be at the root.

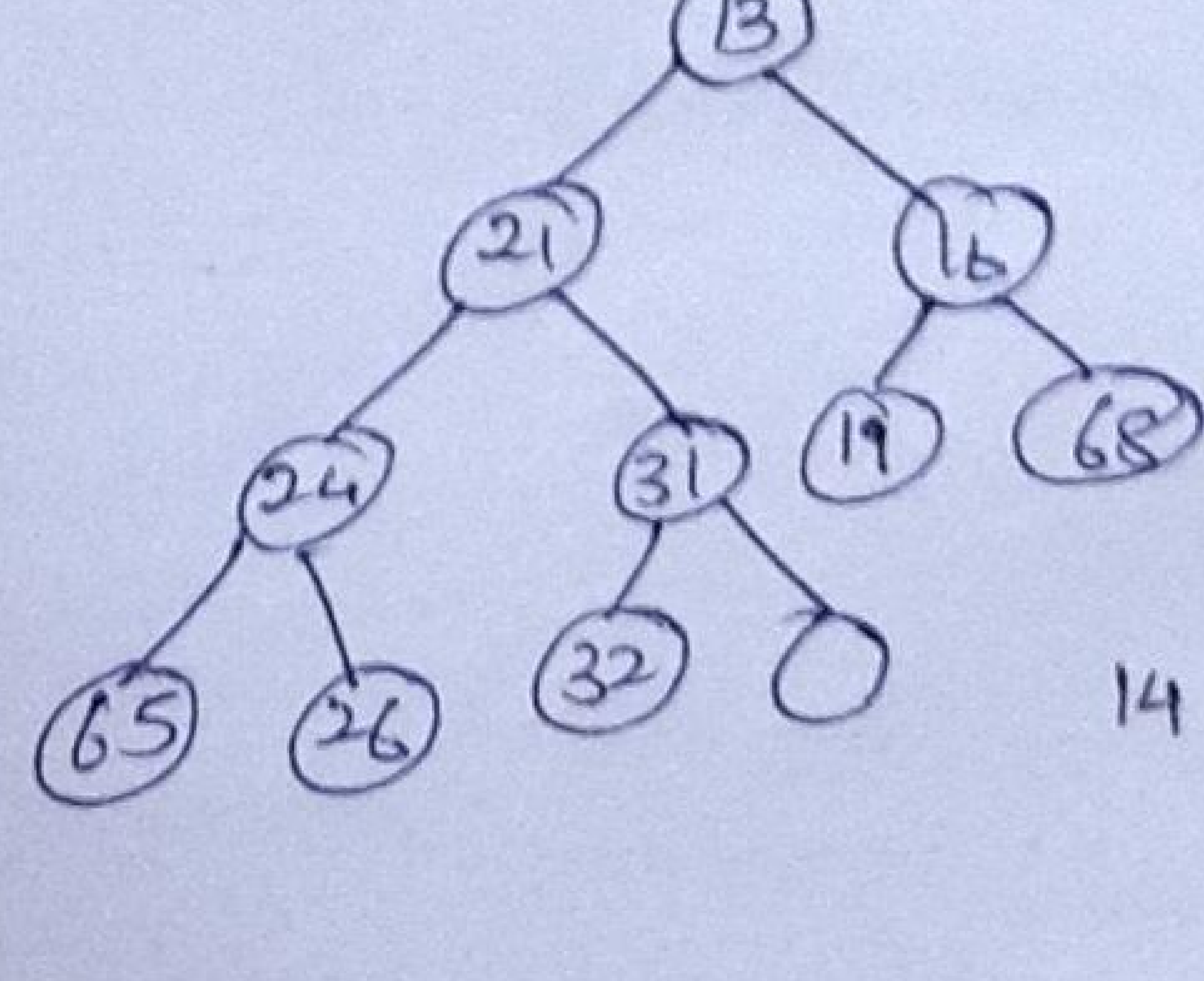
Heap operations are
- Insertion
- Delete min.

Insertion.
- Create a hole in the next available location.
- If x can be placed in the hole without violating heap order, then insert x there.
- Otherwise, slide the element in the hole's parent node into the hole, thus bubbling the hole up towards the root.
- Continue this process until x can be placed in the hole.

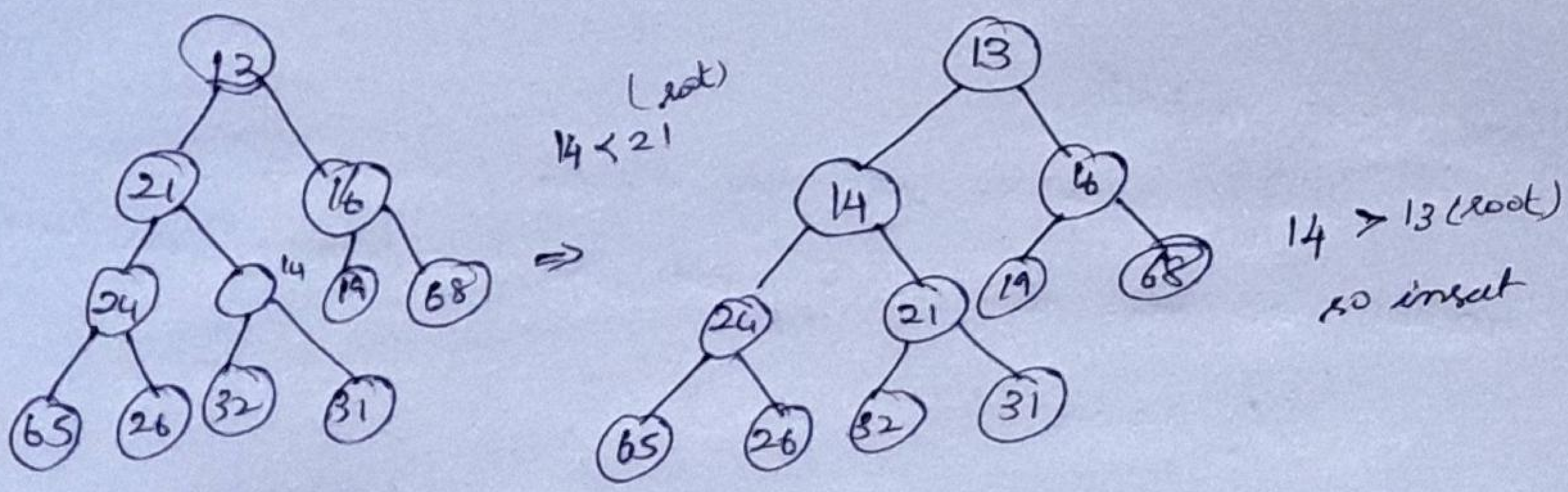
eg.



Insert 14, so create a hole.



14 is smaller than 31 (root)

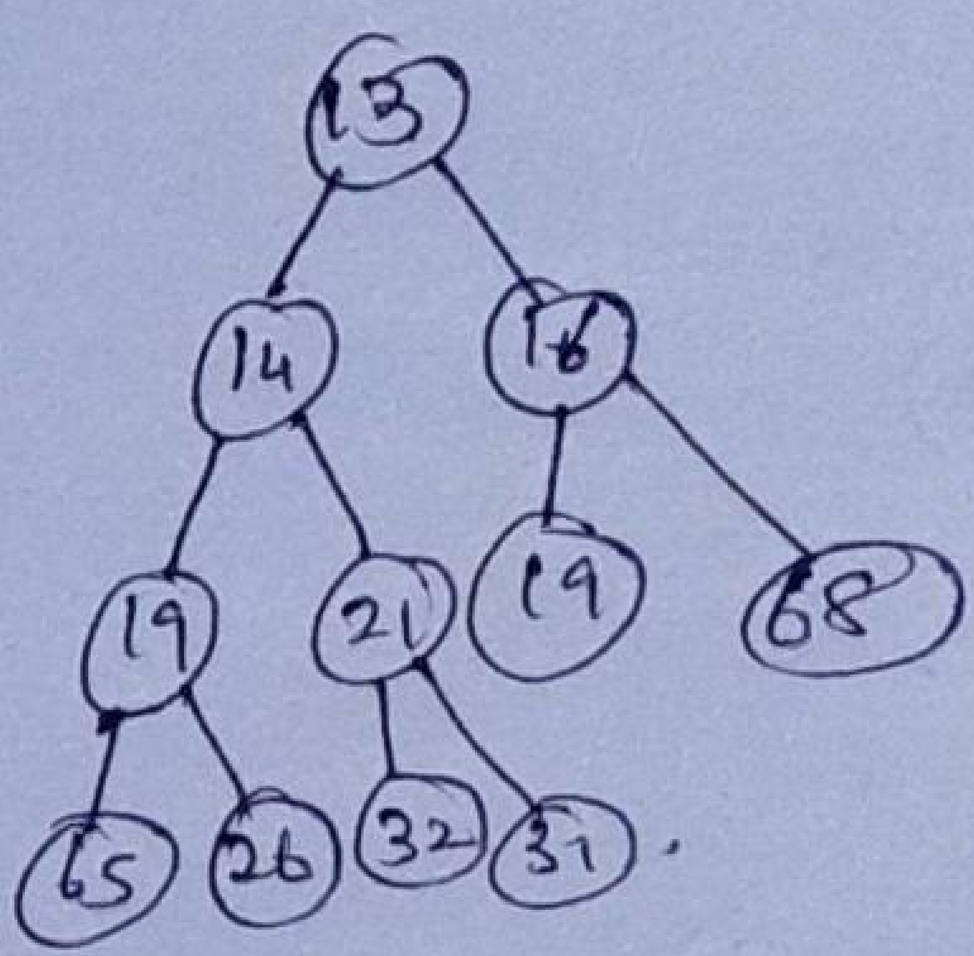


This strategy of bubbling up is known as percolate up.

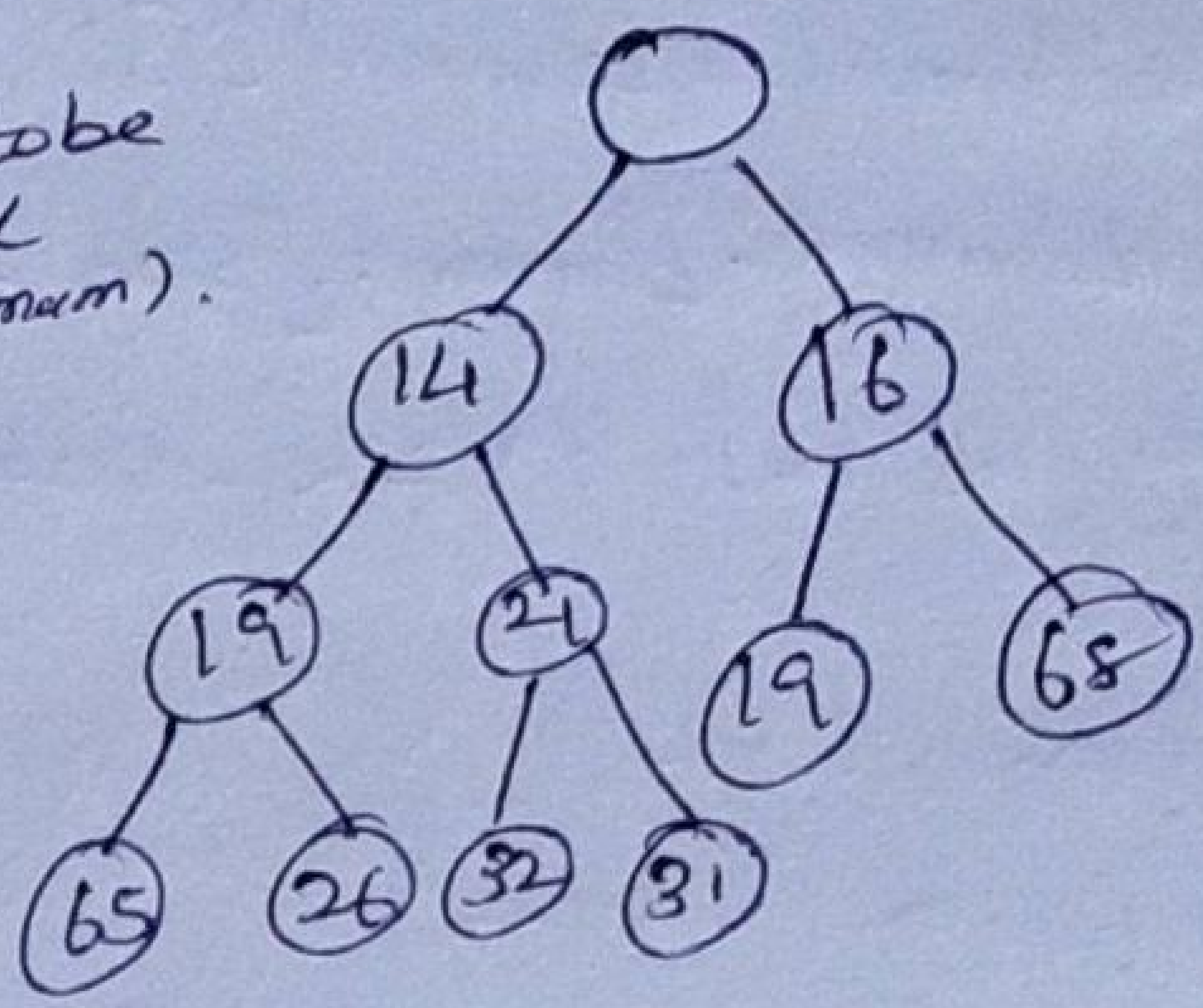
```
void Insert (ElementType x, PriorityQ H)
{
 int i;
 if (Isfull (H))
 {
  Error
  return;
 }
 for (i = ++H -> size ; H -> Element [i/2] > X ; i/=2)
   H -> Element [i] = H -> Element [i/2];
   H -> Element [i] = x;
 }
```

Delete_Min :

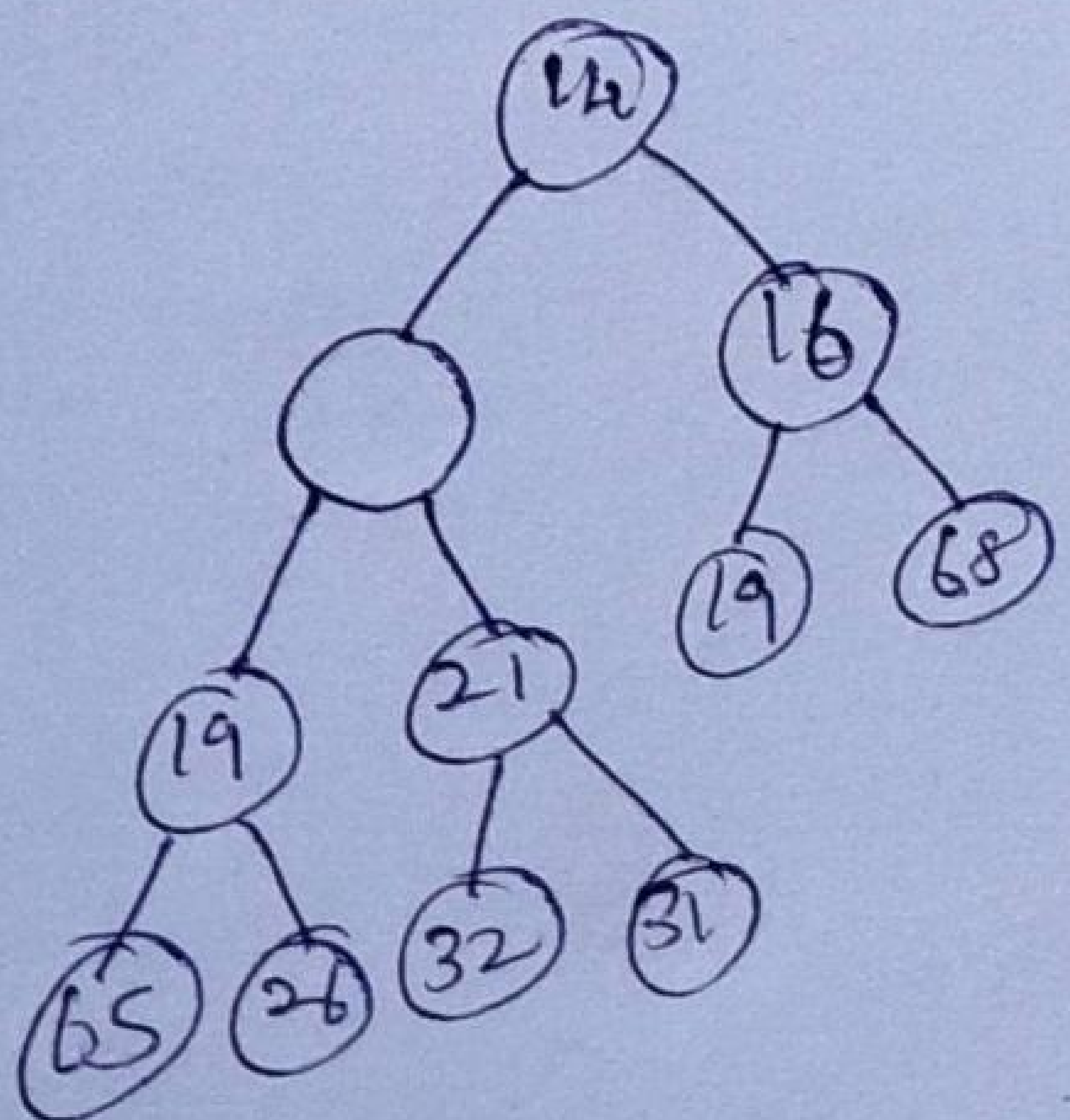
- Min element has to be deleted. i.e, root element.
- Root becomes hole.
- Slide the smaller of the hole's children into the hole, thus pushing the hole down one level.
- Repeat this until X can be placed in the hole

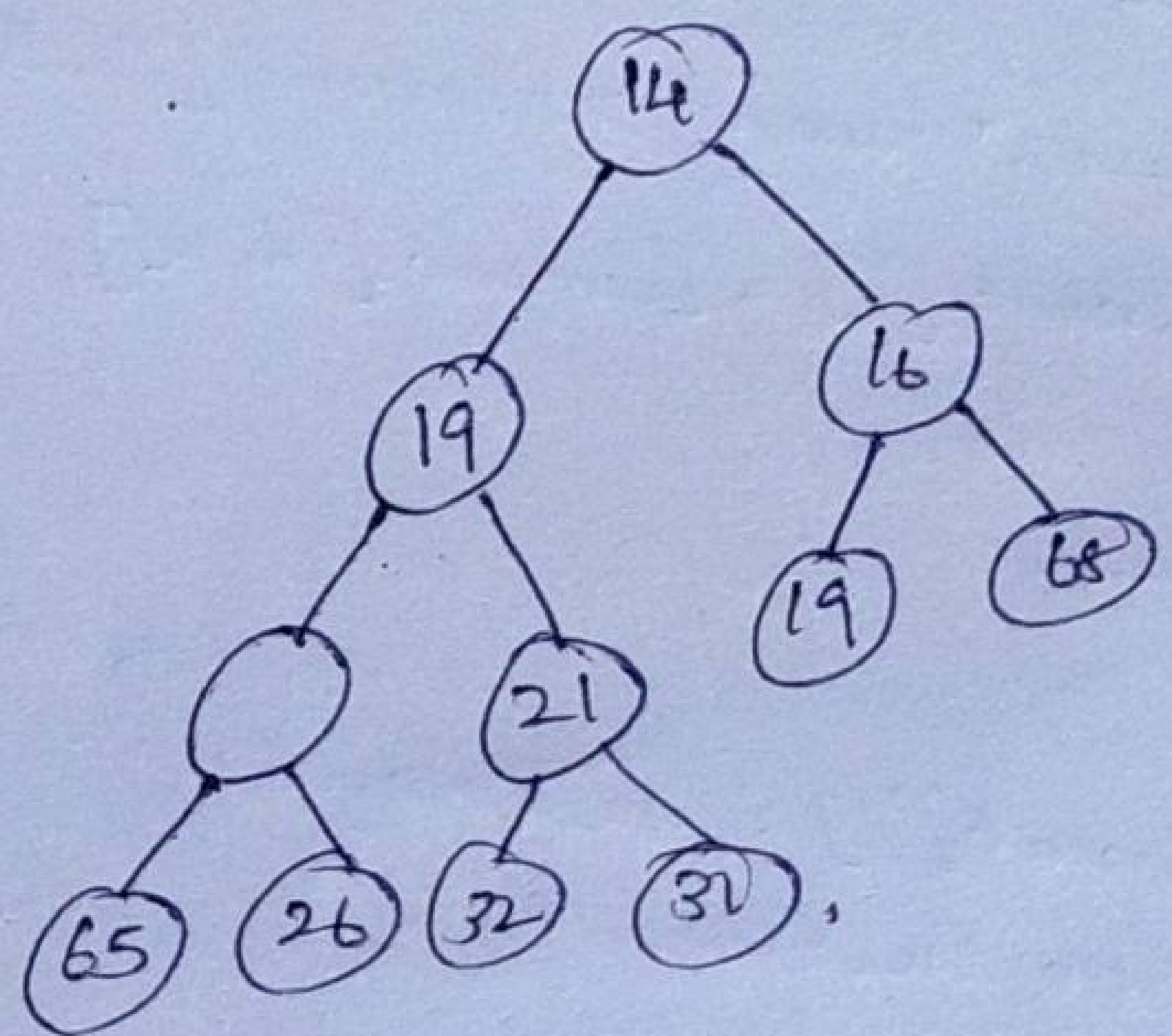


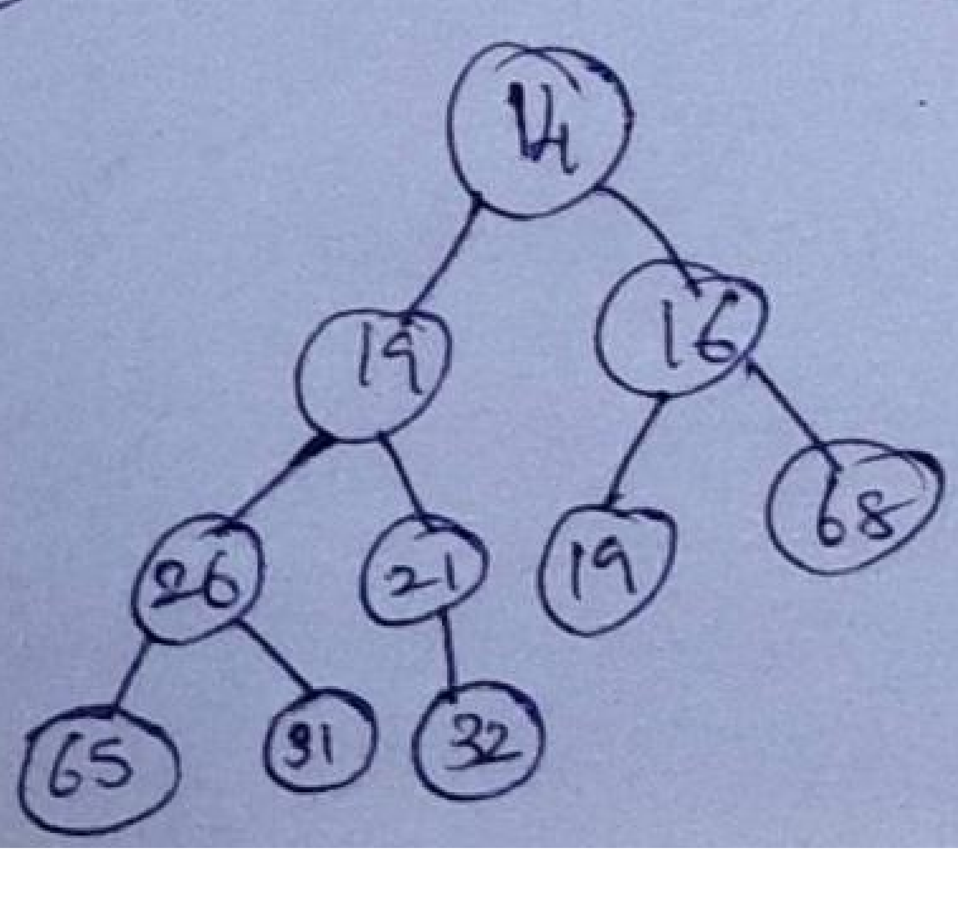
13 is tobe deleted (minimum). ⇒



14 is smaller so put it in the root node.







```
DeleteMin()
{
int child;
ElementType MinElement, LastElement;
if (IsEmpty())
{
Error("Empty");
return(0);
}
MinElement = Element[1];
LastElement = Element[Size--];
for (int i=1; i*2 <= Size; i = child)
{
child = i*2;
if (child != size)
if (Elements[child+1] < Elements[child])
child++;
if (Last_Element > Element[Child])
Element[i] = Element[Child];
else
break;
}
Element[i] = LastElement;
return MinElement;
}
```

Other heap operations:

Decrease Key:

The DecreaseKey(P, Δ, H) operation lowers the value of the key at position P by a positive amount Δ. This might violate the heap order, so it must be fixed by a percolate up.

Increase key.

IncreaseKey (P, Δ, H) operation increases the value of the key at position P by a positive amount Δ. This is done by percolate down.

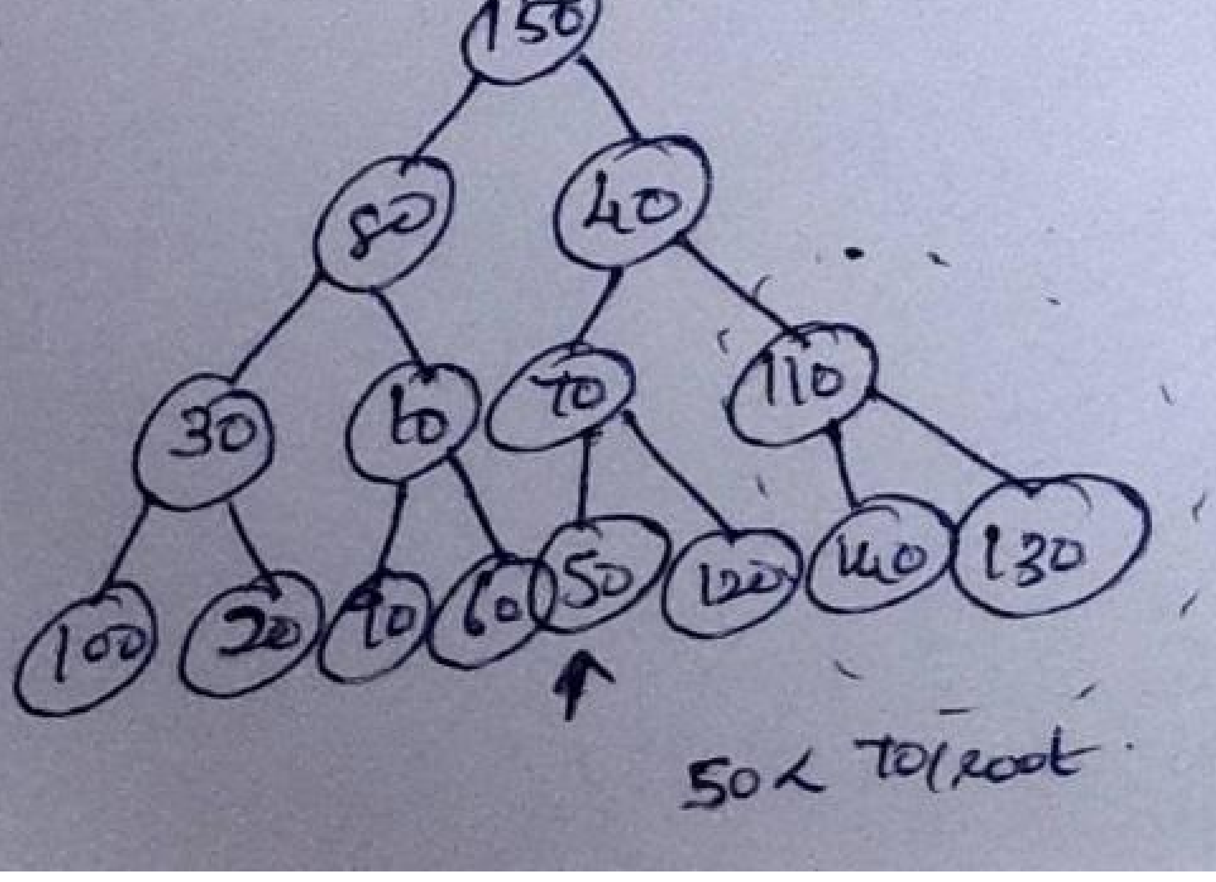
Delete:

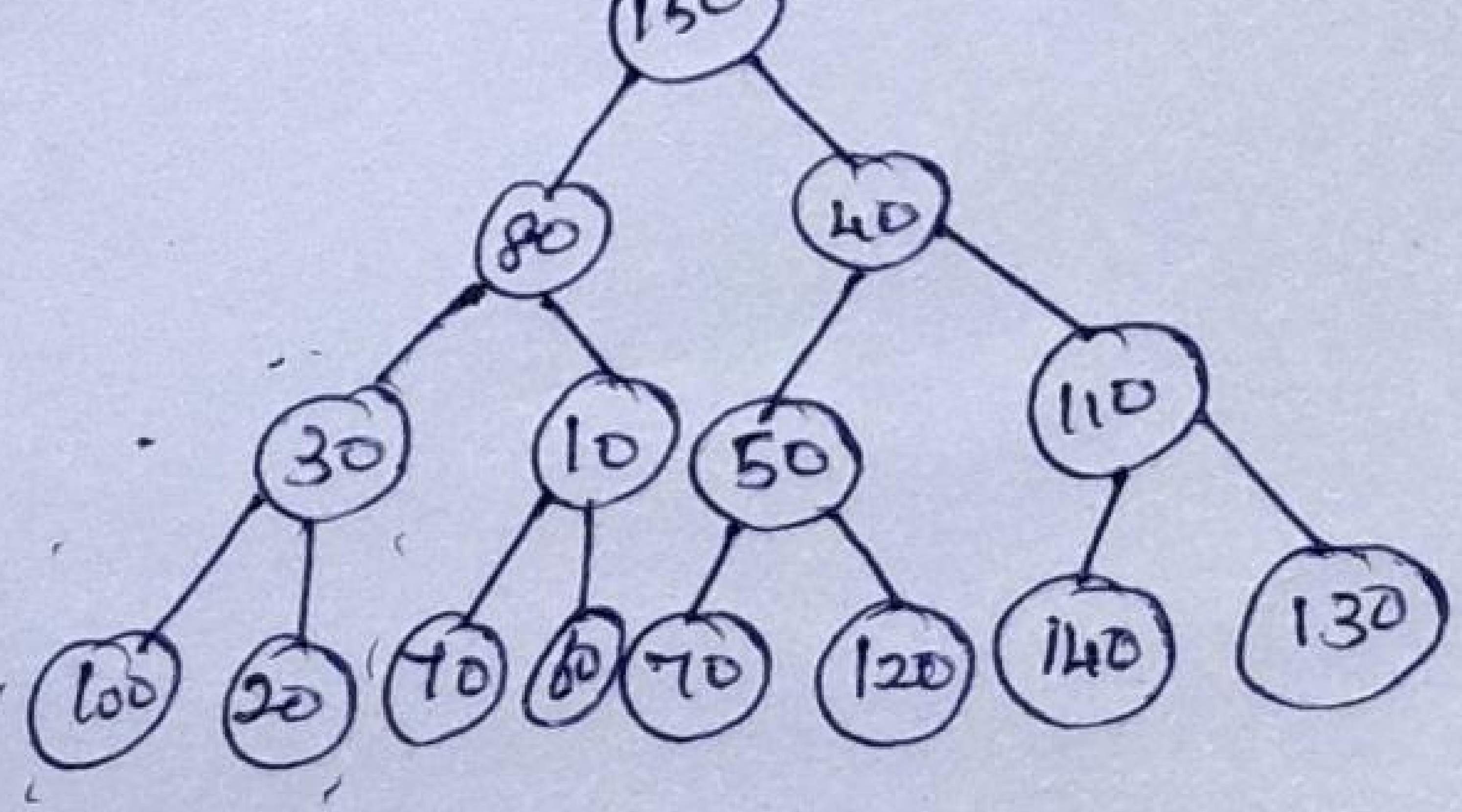
Delete(P, H) operation removes the node at position P from the heap H. First perform DecreaseKey (P, ∞, H) and then perform DeleteMin (H).

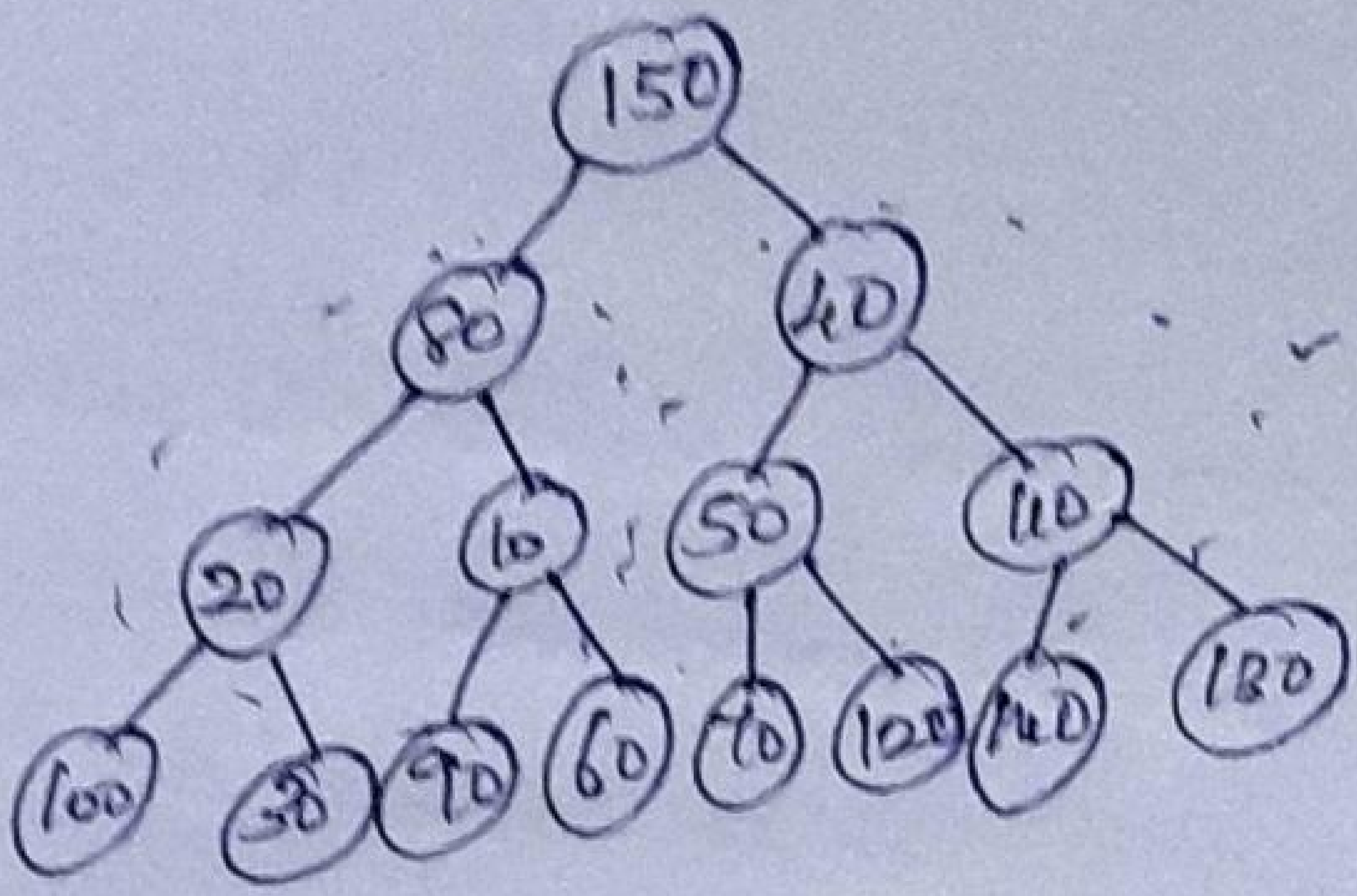
Build Heap.

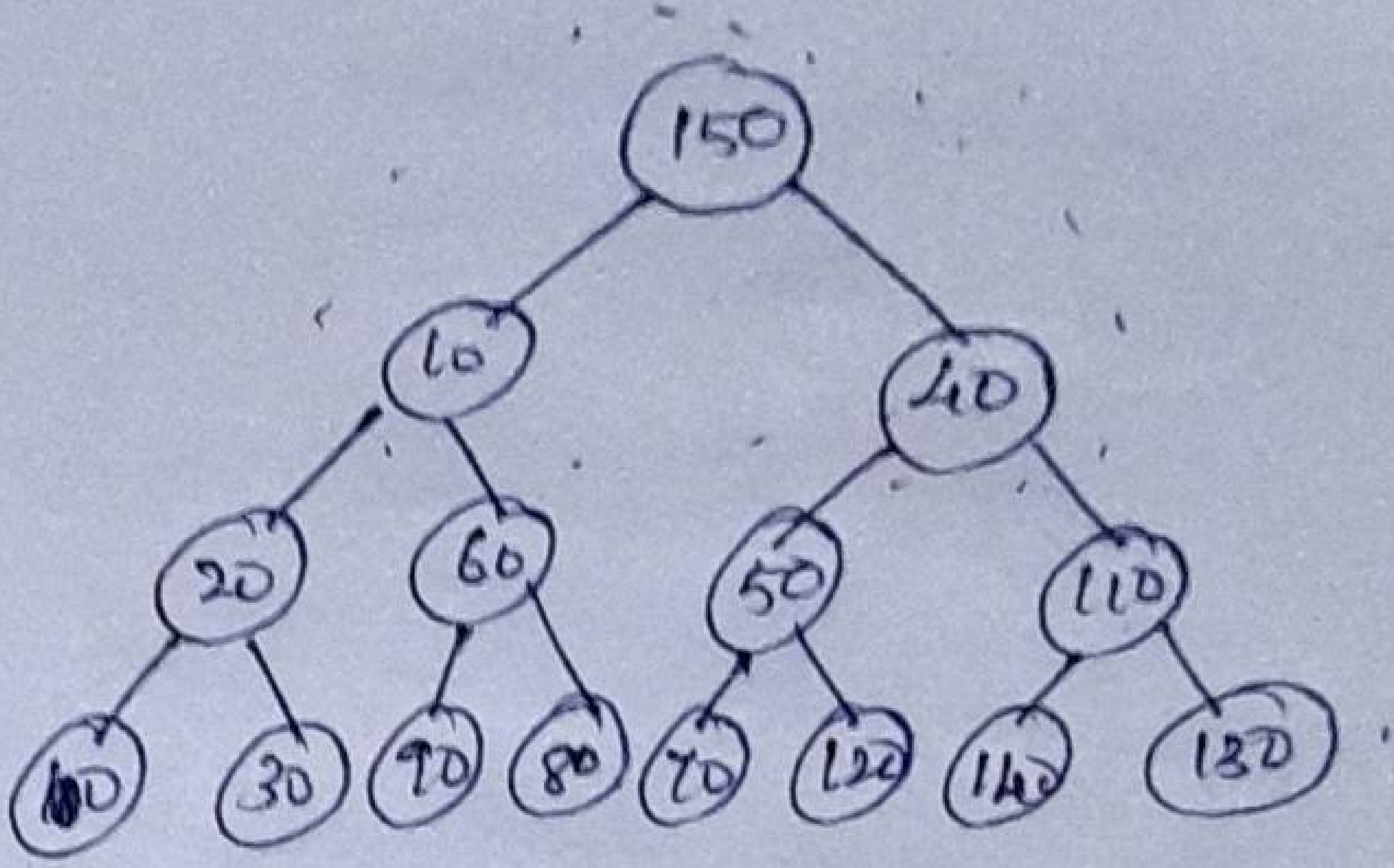
This creates a binary heap from a binary tree. Done by percolate operation.

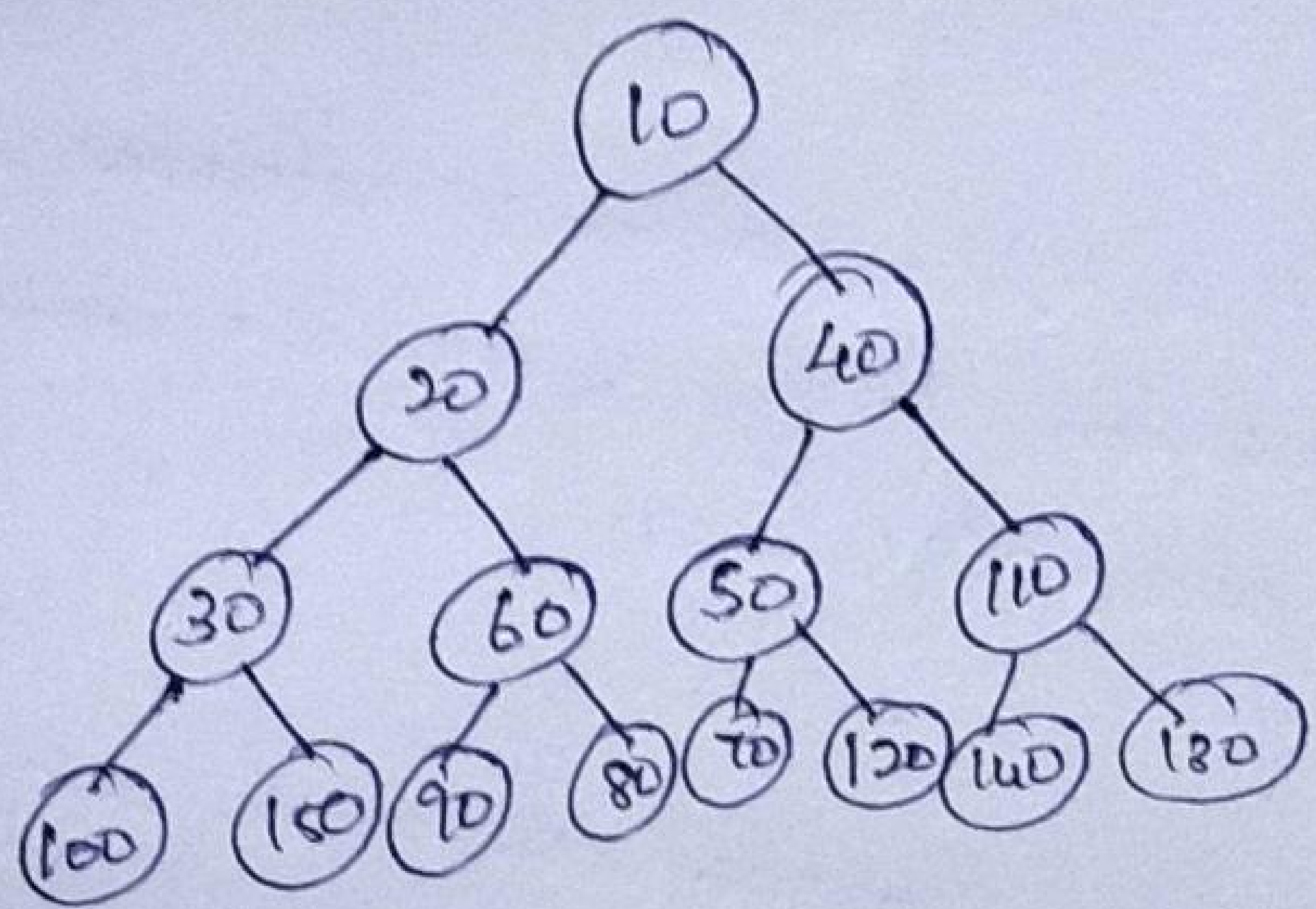
```
for (i = N/2; i>0; i--)
    percolateDown (i);
```






150
80
40
20
10
50
110
100
30
90
60
70
120
140
130


150
10
40
20
60
50
110
100
30
90
80
70
120
140
130


10
20
40
30
60
50
110
100
150
90
80
70
120
140
130

UNIT - IV

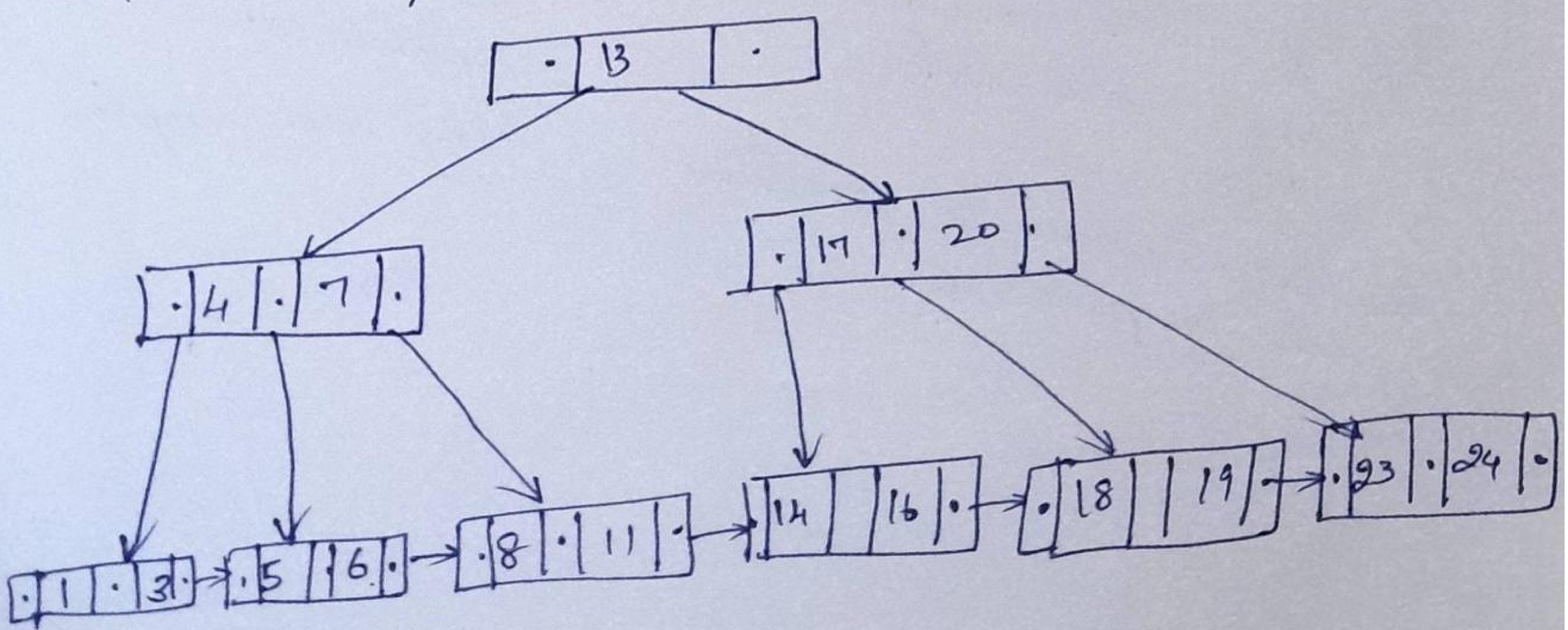
MULTIWAY SEARCH TREES & GRAPHS.

B-TREE.

A B-tree is a search tree but not a binary tree.
A B tree is a order of M with the following structural properties

- The root is either a leaf or has between 2 and M children.
- All non-leaf nodes have b/n [M/2] & M children.
- All leaves are at the same depth.

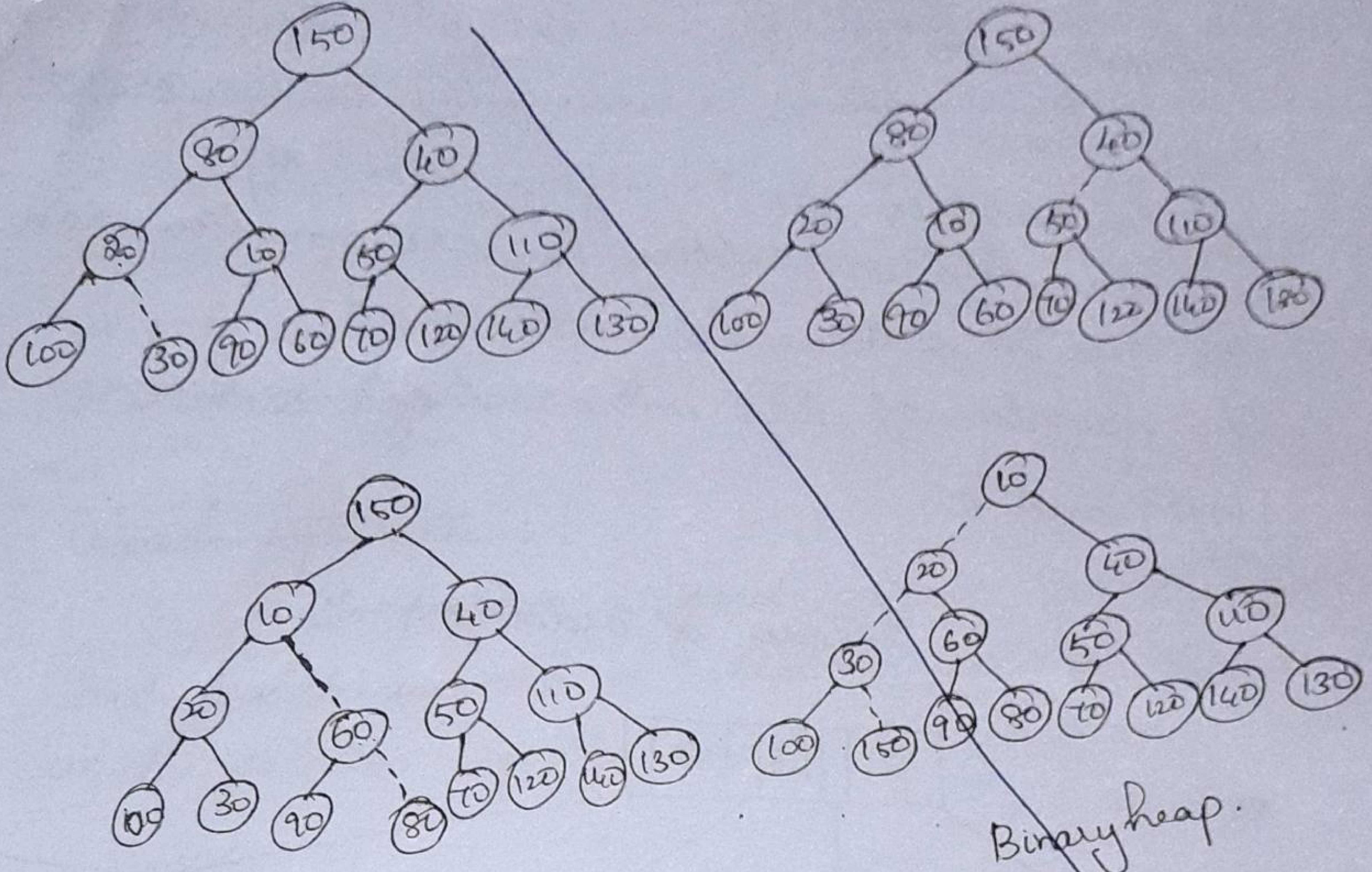
Data are stored at the leaf nodes. Each interior node contains pointers $P_1, P_2 \ldots P_M$ to the children. $K_1, K_2 \ldots K_{M-1}$ are the smallest key values found in the subtree $P_2, P_3 \ldots P_M$ respectively.



B^+-Tree

Comparison between B-Tree and B+ Tree

B-Tree	B+ Tree
1. Search Keys are not repeated.	Stores redundant search Keys.
2. Data is stored in internal or leaf nodes.	Data is stored only in leaf nodes.
3. Searching takes more time as data may be found in a leaf or non-leaf node.	Searching data is very easy as the data can be found in leaf nodes only.
4. Deletion of leaf node is very complicated.	Deletion is simple
5. Leaf nodes cannot be stored using linked lists	Leaf node data are ordered using sequential linked list.
6. The structure & operations are complicated.	They are simple.



Binary heap.

B-TREE

A B-Tree is a search tree but not a binary tree.

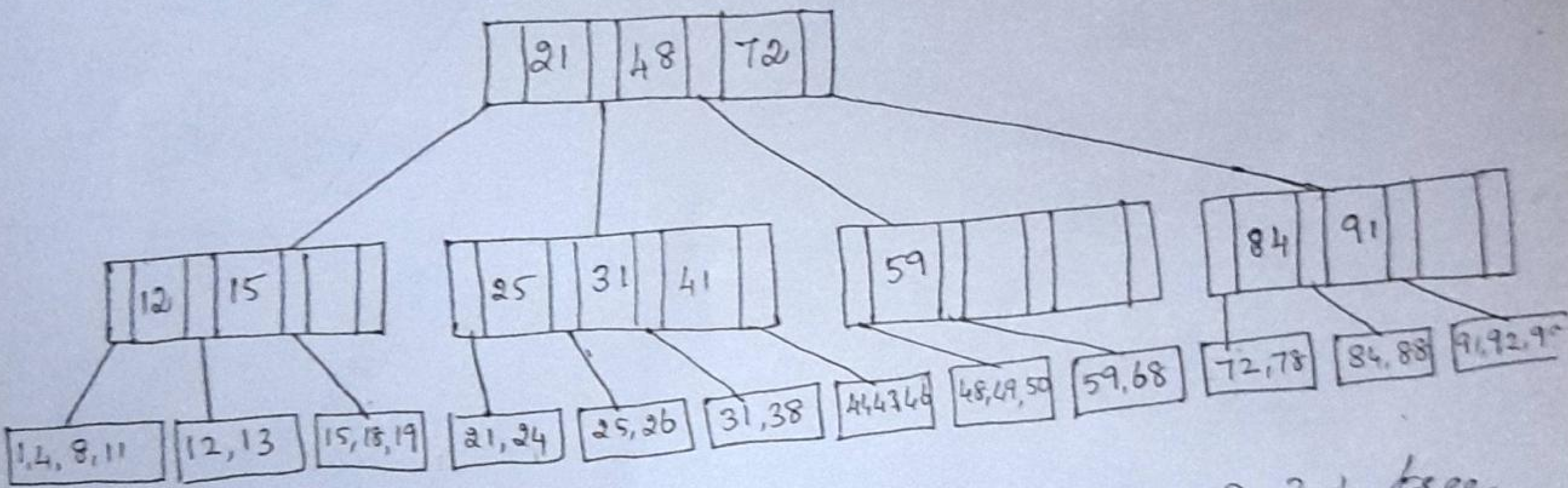
A B-Tree of order M is a tree with the following structural properties.

- The root is either a leaf or has between 2 and M children.
- All non leaf nodes have between [M/2] and M children.
- All leaves are at the same depth.

Datas are stored at the leaf. Each interior node contains pointers $P_1, P_2 .. P_M$ to the children $K_1, K_2 .. K_{M-1}$ are the smallest key values found in the subtree $P_2, P_3 .. P_M$ respectively. All the keys in subtree P_1 are smaller than the keys in subtree P_2 and so on.

The number of keys in a leaf is between $[M/2]$ and M.

B-Tree of order 4.

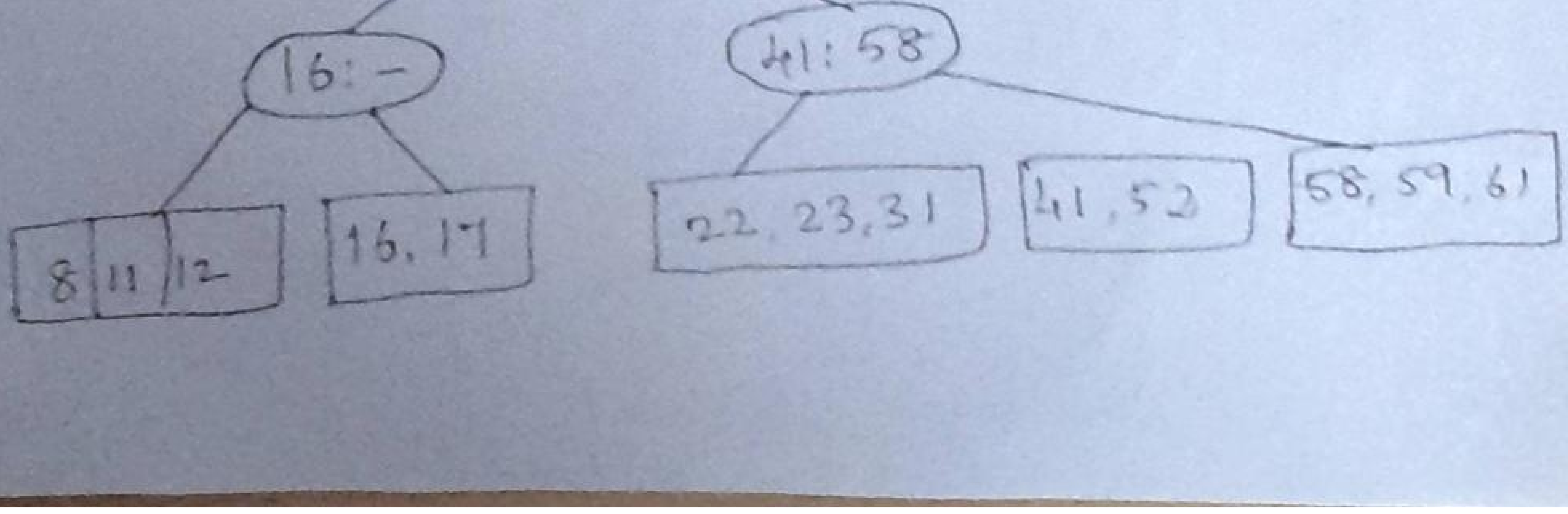


A B.tree of order 4 is known as a 2-3-4 tree and a B.tree of order 3 is known as 2-3 tree.

Operations in B Tree

~~[illegible]~~:

Consider the example,



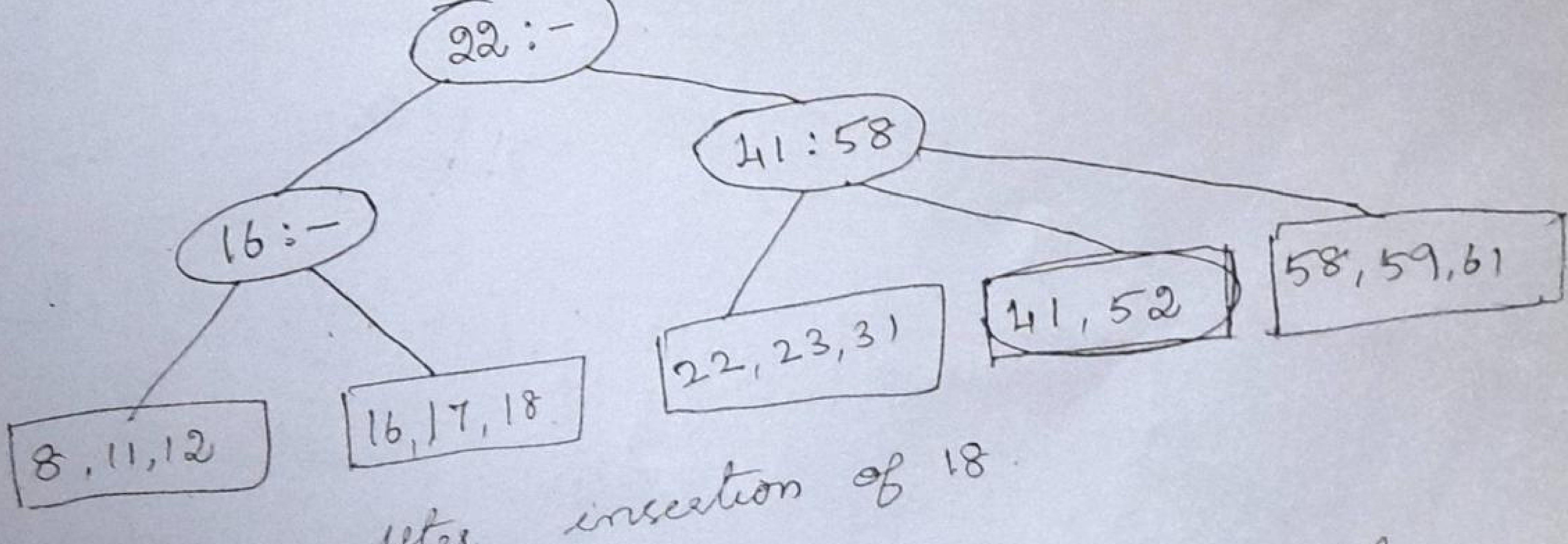
The interior nodes are represented using ellipse. It contains two pieces of data for each node. A dash line as a second piece of information in an interior node indicates that the node has only two children. The leaves are drawn in boxes which contains the key.

i) Find operation:

To perform a find, start at the root and branch in one of three directions, depending on the relation of the key to be searched:

ii) Insert:

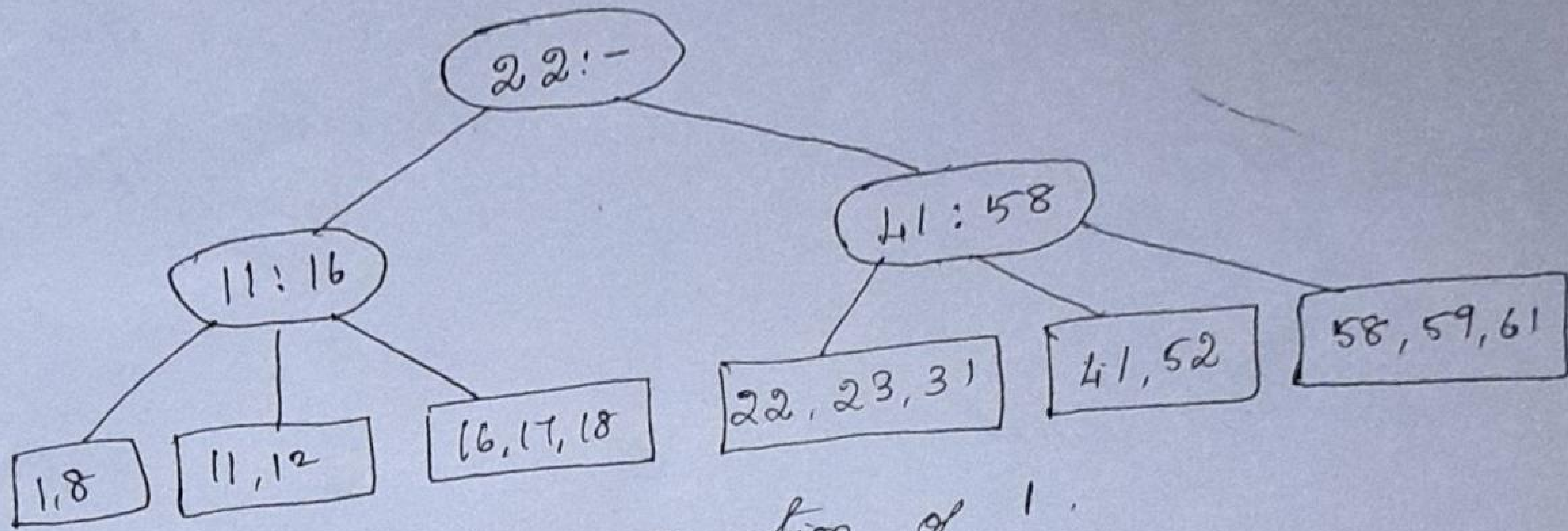
To insert a node first perform the find operation to find the exact position to place the new value to be inserted. To insert a node with key 18, we can just add it to a leaf without causing any violations of the 2-3 tree properties. The result is,



After insertion of 18.

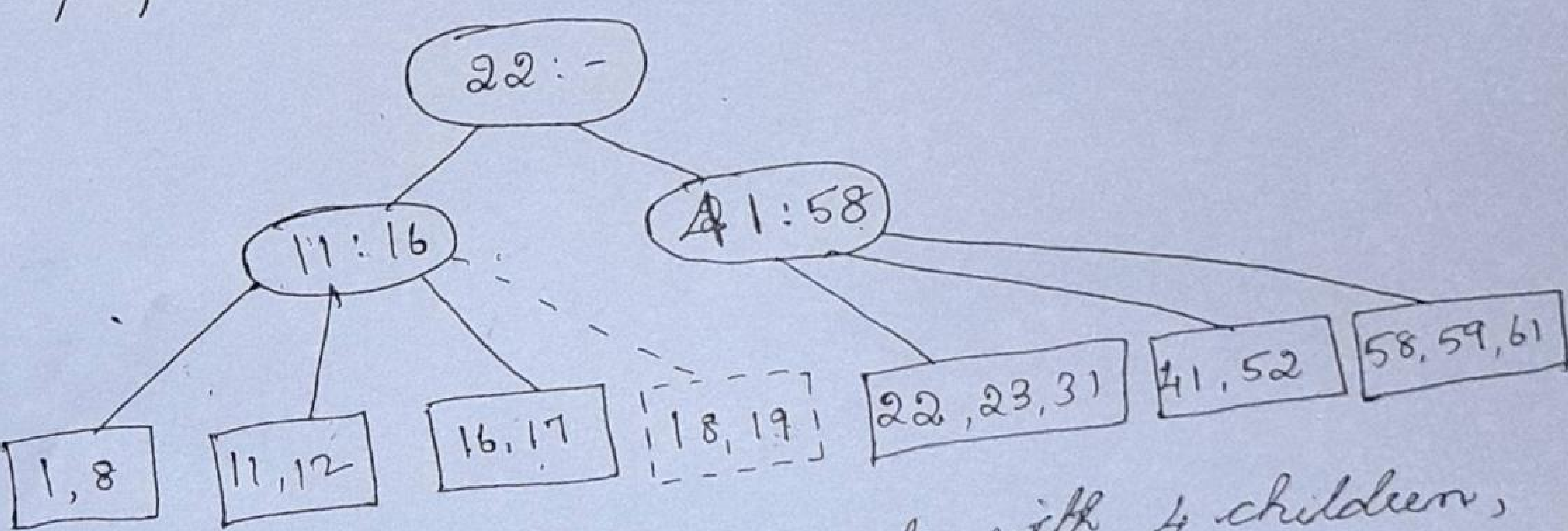
This does not violate the 2-3 tree properties.

If key value 1 is to be inserted to the tree, the first node where the data should be stored is already full. Placing the new key into this node would give it a fourth element, which is not allowed. This can be solved by making two nodes of two keys each and adjusting the information in the parent.



After Insertion of 1.

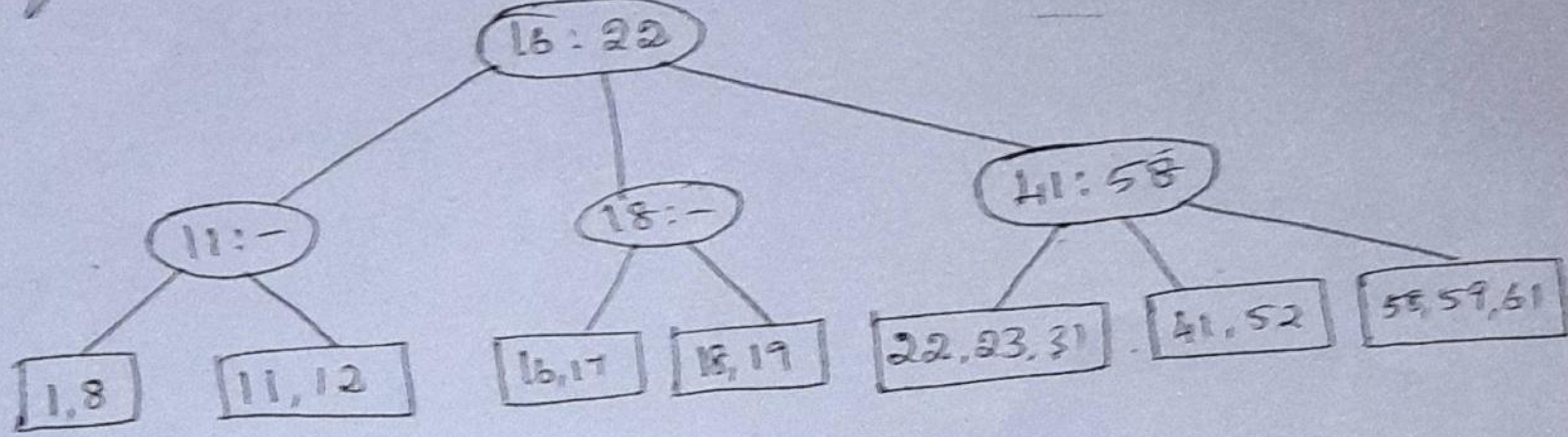
Inserting 19 to the current tree will violate the 2-3 property.



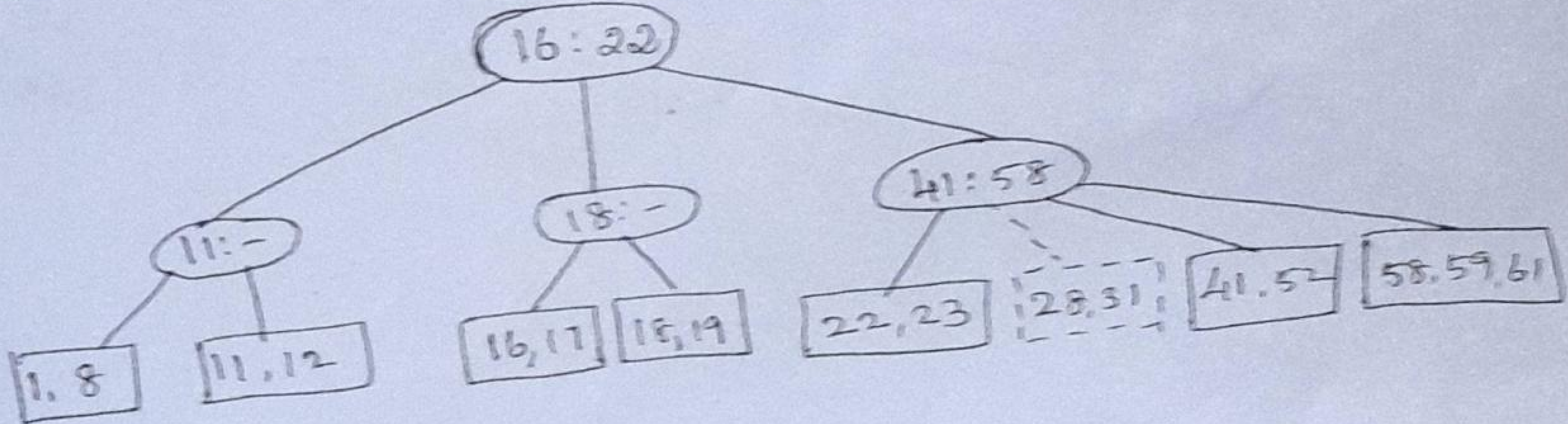
This tree has an internal node with 4 children, but it is allowed only for three node (child)

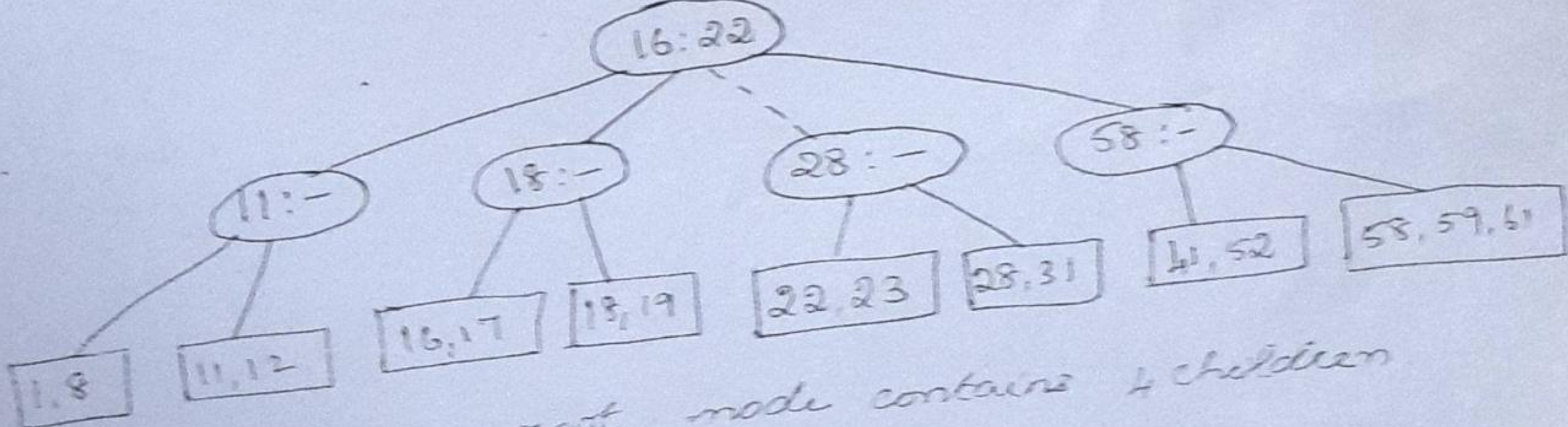
So the intermediate node can be splitted into two with two children each. In some cases this will not result in correct order. So the same process is done to its root node.

The intermediate node can be splitted as follows.

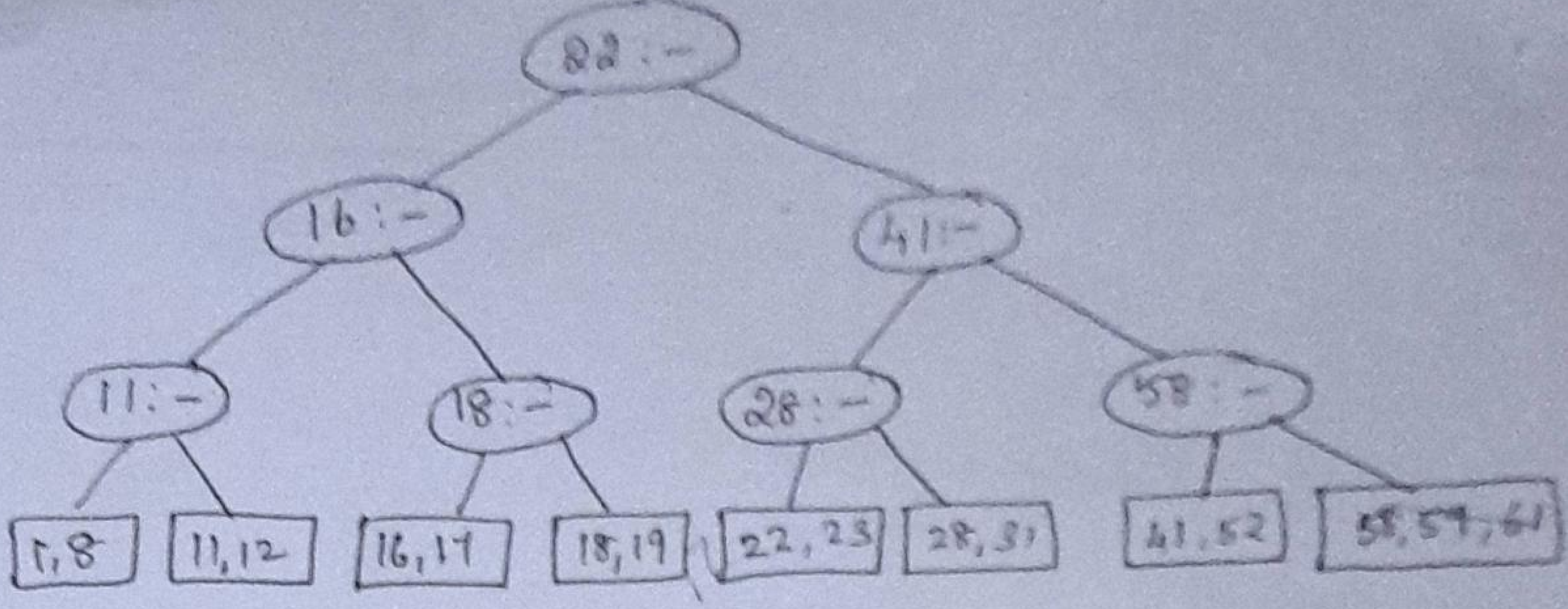


To this tree when a key 28 is inserted the process is much complex.

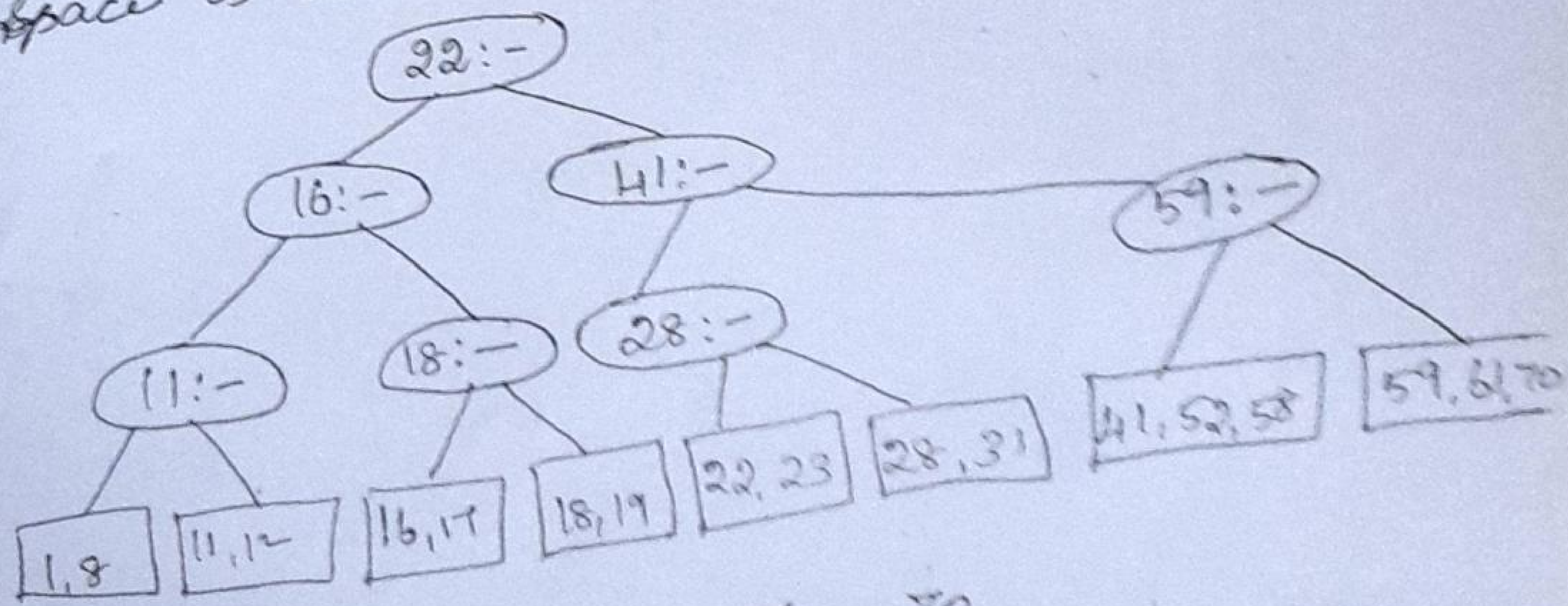




Now the root node contains 4 children



An insertion requires creation of new intermediate node. There is an alternative method which does not need to create an intermediate node. Consider the above figure, when a key value 70 is to be inserted, instead of splitting the node into two, we can first attempt to find a sibling with only two keys. In this example, insert 70 into the node with 59, 61 and move 58 to the leaf containing 41, 52. and adjust the entries in the internal nodes. This strategy will keep the nodes full. The cost will be more but little space is wasted.



After inserting 70.

iii) <u>Deletion</u>:

To delete a key find the key to be deleted and remove it. If a key is deleted from a node containing two Key, then after deletion the leaf node will have only one key.

Solution to the above strategy is that,

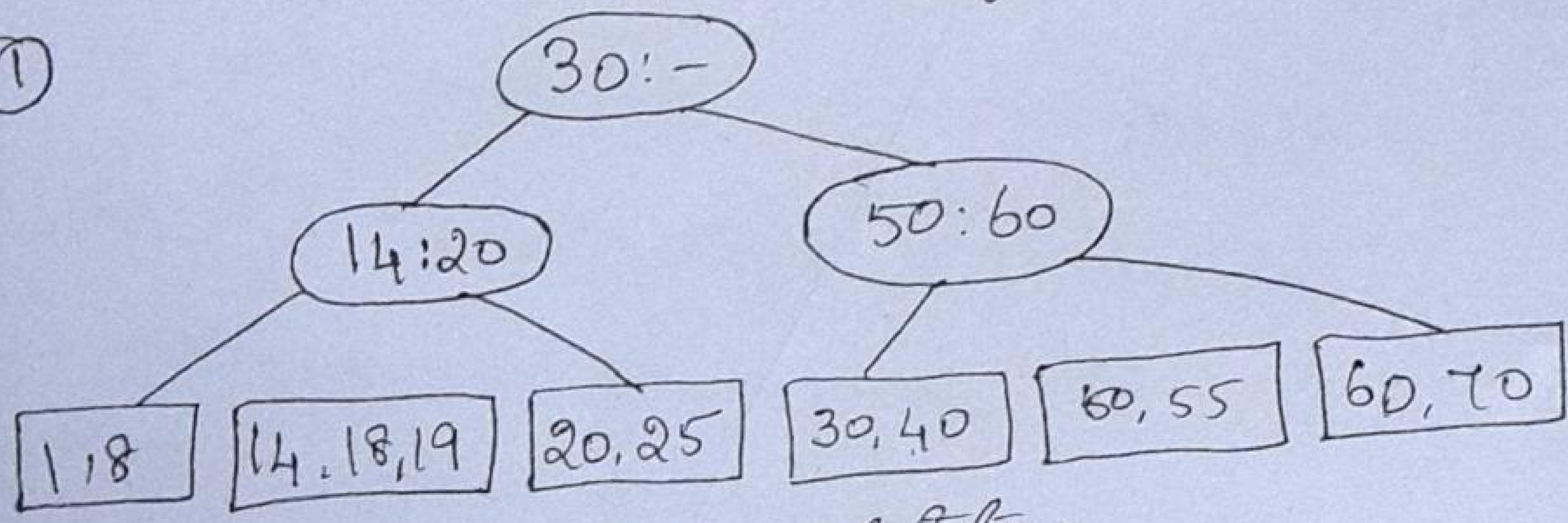
- Combine the leaf with one key with its sibling.

If sibling has 3 keys, steal one key from its sibling.

If sibling has 2 keys, combine two nodes into single node. (1+2) = 3 keys. Then adjust the parent node and update all the internal nodes accordingly.

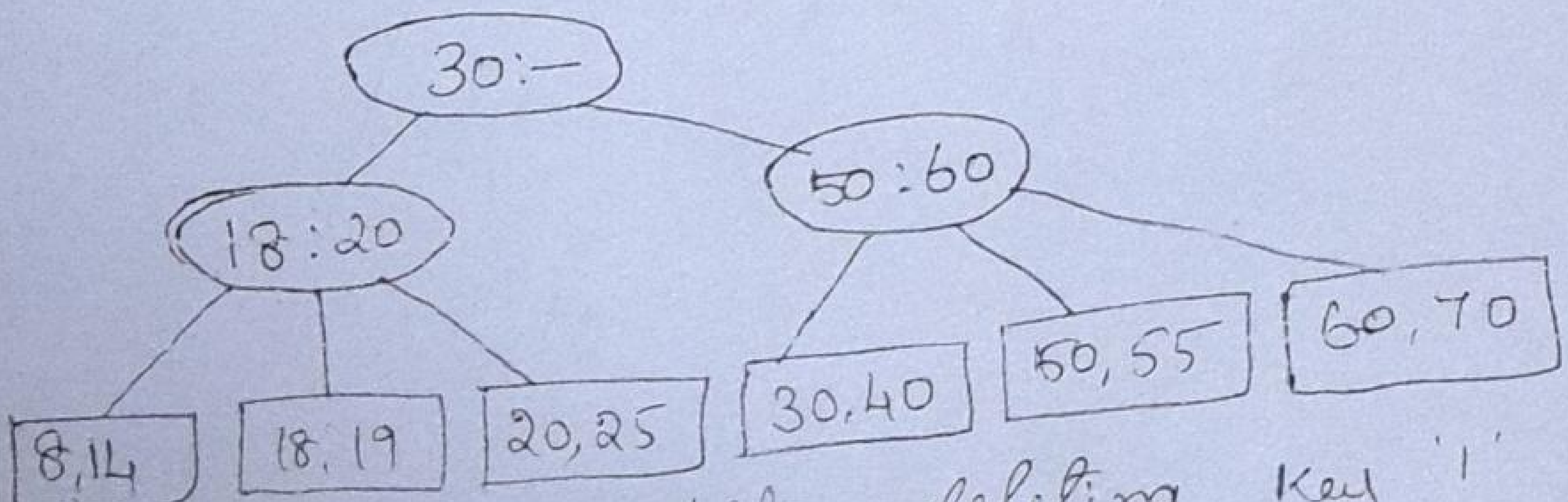
<u>case 1</u>: Deleting a key from leaf with 2 keys. If sibiling has 3 keys.

fig: ①



Before deletion

fig: ②



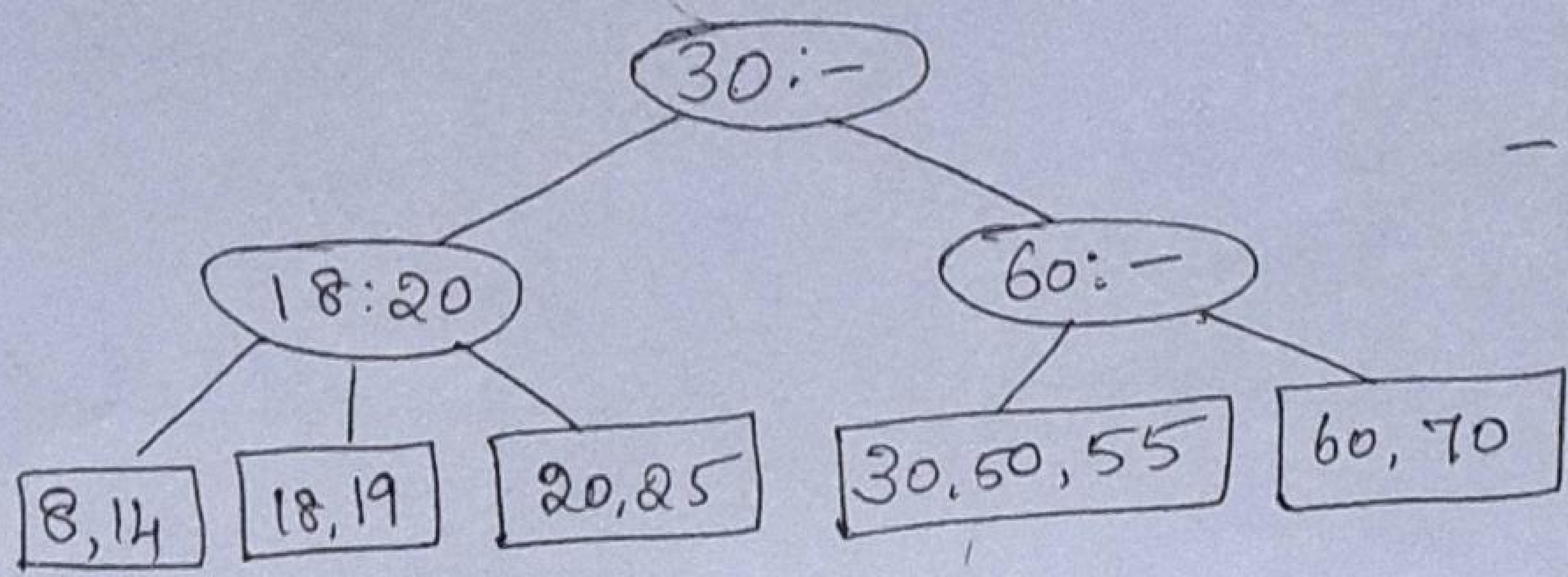
After deleting key '1'

case 2:

If sibling has 2 keys.

Delete 40 from fig: ②

- Combine 30 with its sibling
- Update internal nodes.



After deleting 40

HASHING

The hash table datastructure is an array of fixed size containing the keys. A key is a string with an associated value.

eg) Salary information.

The size of the hash table is refered to TableSize. Every Key is mapped into some number in the range 0 to [TableSize -1] and placed in the appropriate cell. The mapping is called a hash function. A hash function should be simple to compute and should ensure that any two distinct keys get different cells.

GRAPHS

Definitions:

Graphs:

A graph is a collection of vertices and edges.

G = (V, E) V = Set of vertices
E = Set of edges.

Edges (or) Arcs:

Each edge is defined by a pair of vertices. (v, w), where V, W ∈ V. Sometimes edges contains weight or cost.

Vertices:

Vertices are also called as nodes. Vertices contains labels.

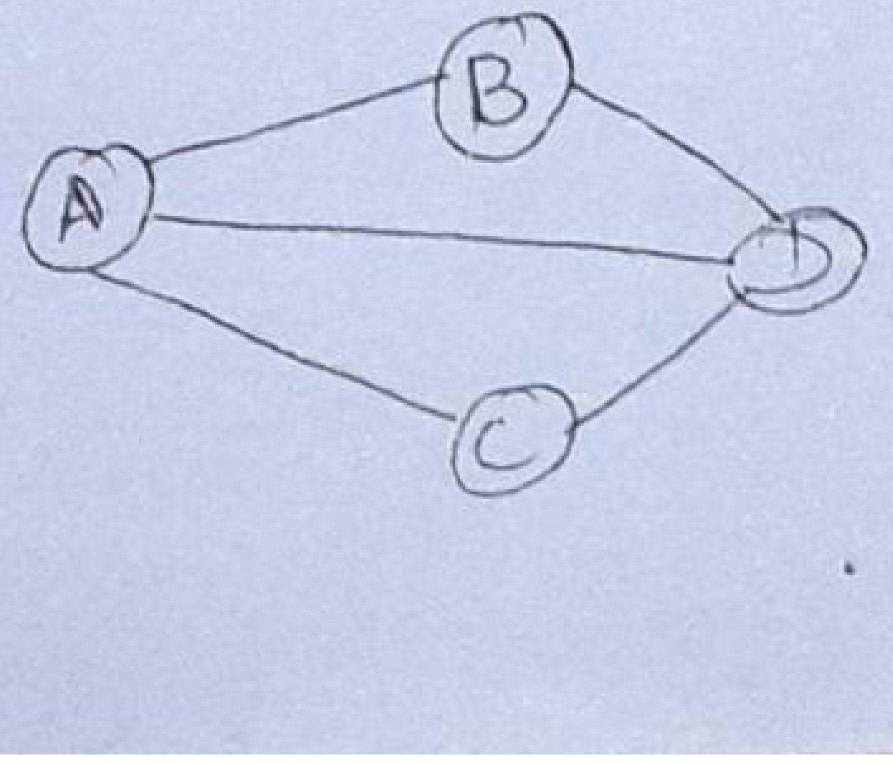
Representation of edges & Vertices:

Edges are represented using lines connected between circles.

Vertices are represented with circle with label in it.

G = (V, E)
V = {A, B, C, D}
E = {(A,B) (B,D) (A,D) (A,C) (C,D)}

eg)



Adjacent Vertices:

Two vertices (A, B) are said to be adjacent if there exists an edge between that two vertices (A & B). From the above graph (A, B) are adjacent.

Path:

Path is a sequence of vertices w_1, w_2, w_3 .. Each pair of successive vertices is connected by edges.

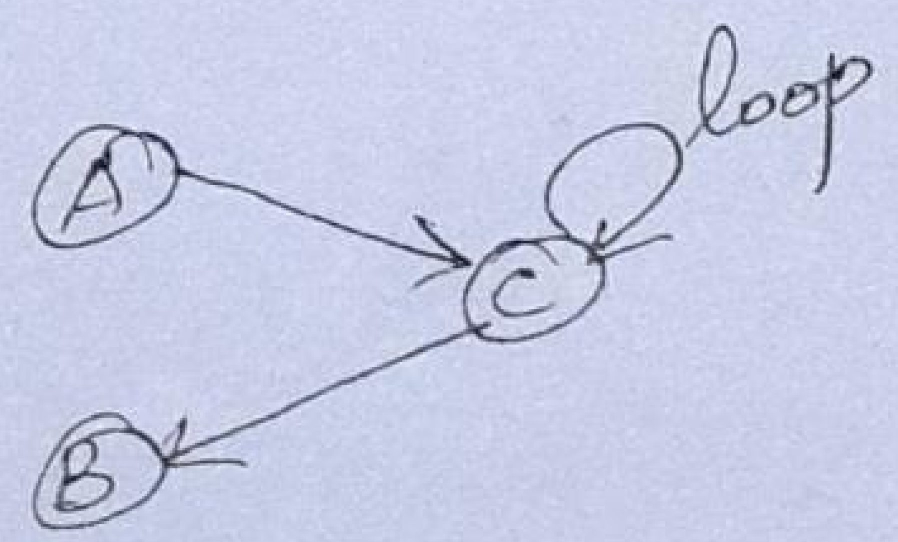
Length of a path:

It is the number of edges in the path. If the edges contains weight or cost then the length of path is the sum of weights of the edges. The length of path is set to zero if there is a path from a vertex to itself.

Loop:

If the graph contains an edge from a vertex to itself, then the path is referred to as a loop.

eg)



path length of c to c is 0.

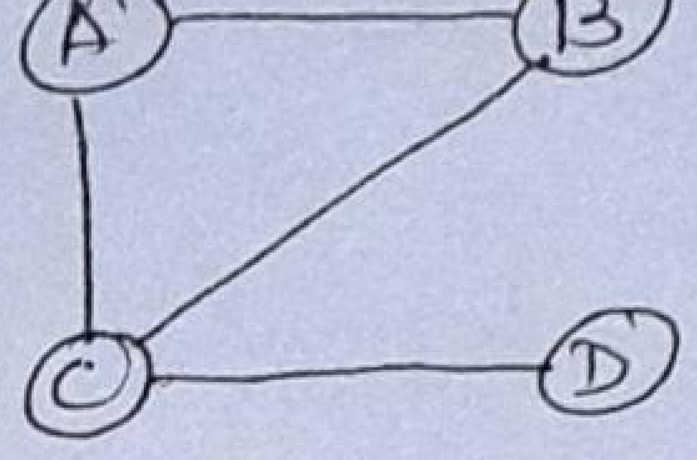
Simple path:

It is a path such that all vertices are distinct, except that the first and last could be the same.

Cycle:

A path that starts and ends on the same vertex.

(eg)



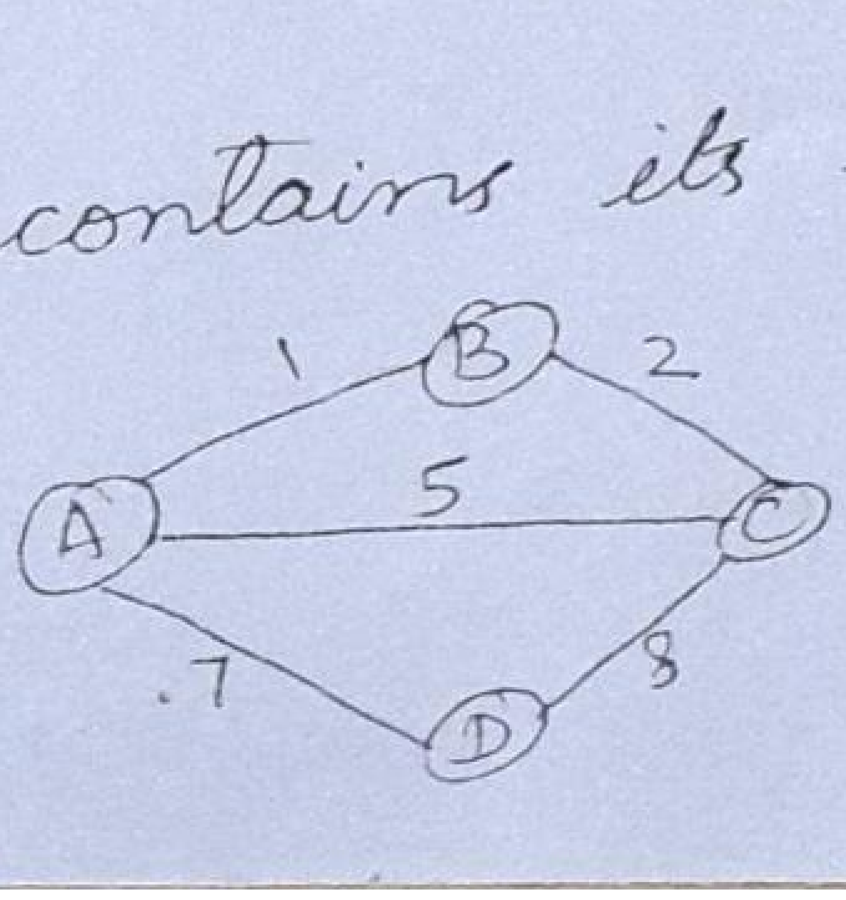
Path:	Simple path	cycle.
ABC	ABCD	ABCA
ABCD	DCA	BACB
ABCABCABCD.	AB	CABC
BACD.	ABC	CBAC

Graph Classification:

1. Weighted or Unweighted graph
2. Directed or Undirected graph
3. Cyclic or Acyclic graph.

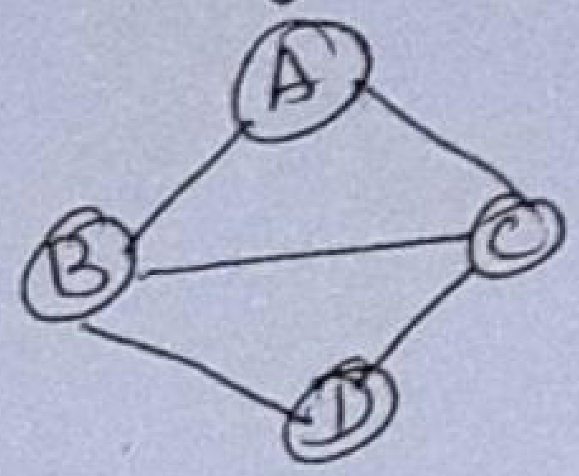
Weighted graph:

The edges contains its weight.



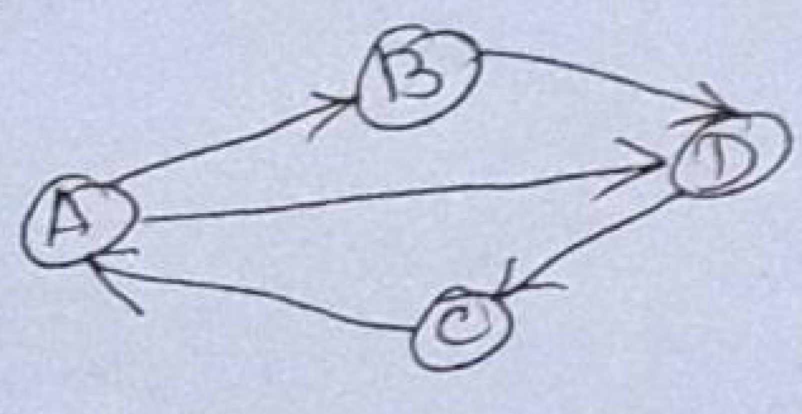
Unweighted graph:

The edges contains no weight.



Directed graph:

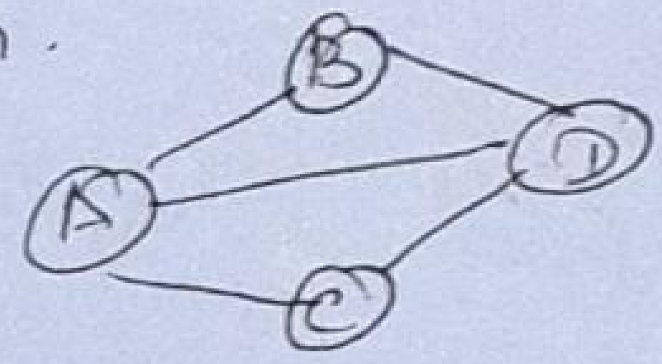
In this each edge can be traversed only in a specified direction through the arrow of the edges. All the edges contains an arrow mark pointing towards the next or adjacent node. This is also known as digraphs.



Undirected graph:

Here the edge doesnot contain any arrow mark. Each edge can be traversed in either direction.

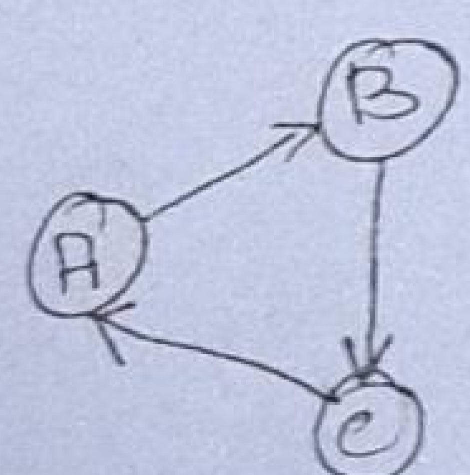
(eg)



Cyclic graph:

If a graph contains cycles it is called as cyclic graphs.

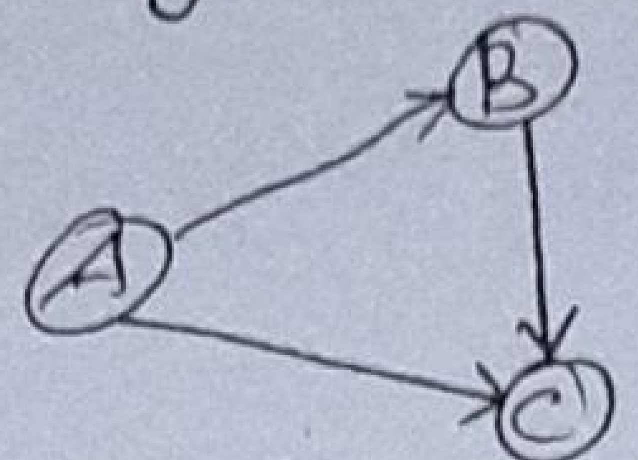
eg)



Acyclic graph:

An Acyclic graph doesnot contain any cycle in it.

(eg)



Directed acyclic graph: (DAG)

It is a directed graph with no cycle.

Degree of a node:

It is the number of edges, the node is used to define.

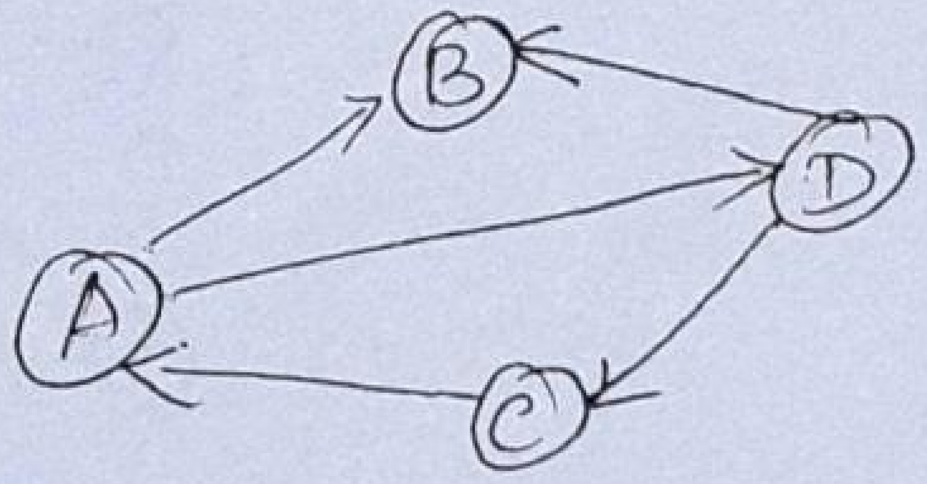
In Degree:

No. of edges pointing to a node. It is the number of incoming edges.

Out degree:

Number of edges pointing from a node. It is the number of outgoing edges.

(eg)



Indegree (A) = 1

Outdegree (A) = 2

ID (B) = 2

OD (B) = 0

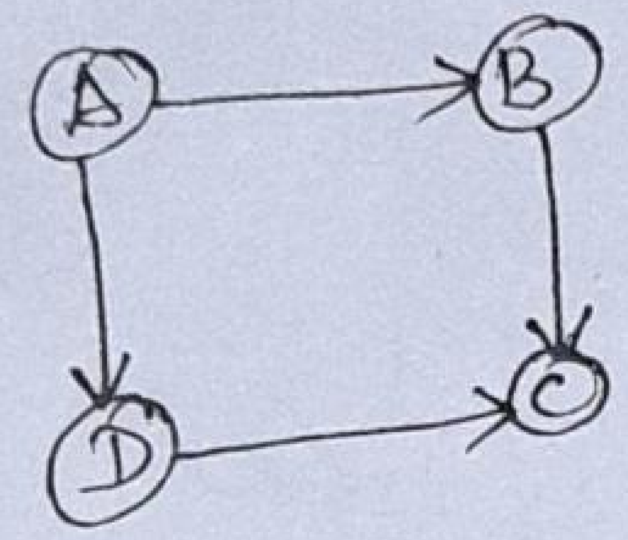
Connected graphs:

An undirected graph is connected if there is a path from every vertex to every other vertex.

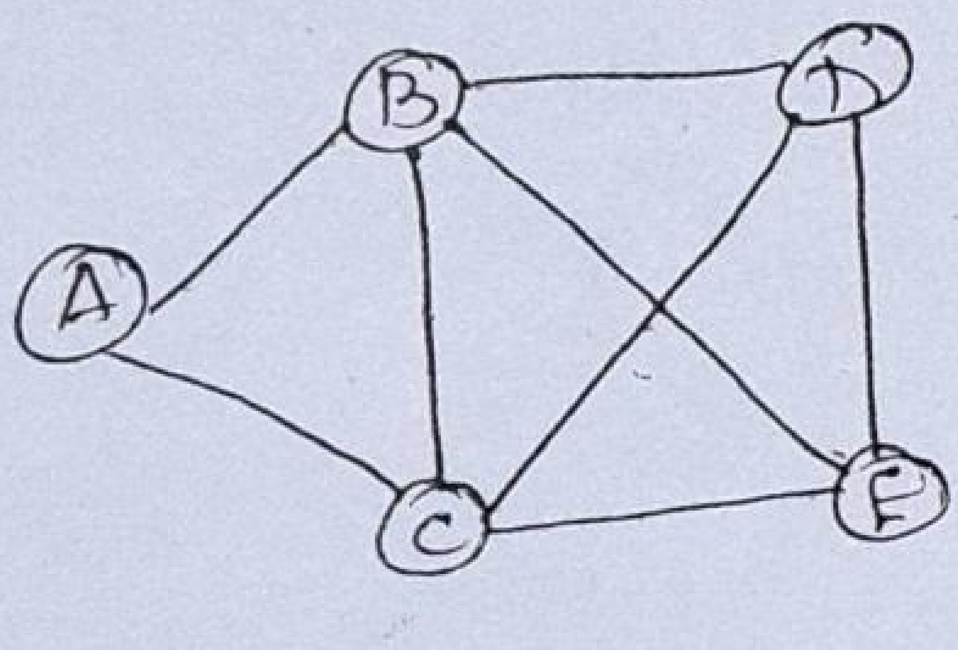
If a directed graph has a path from every vertex to every other vertex then it is known as strongly connected graphs. otherwise it is known as weakly connected graphs.

eg)

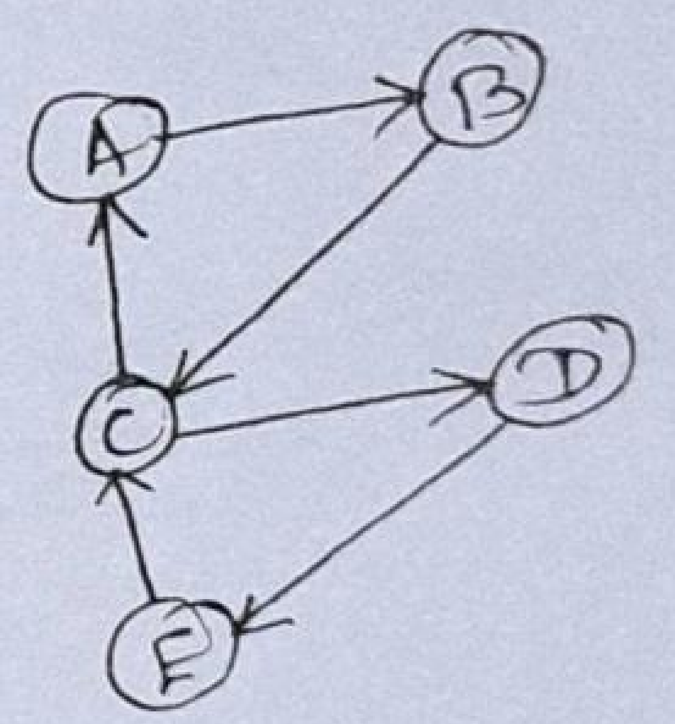
DAG (Directed Acyclic Graph)



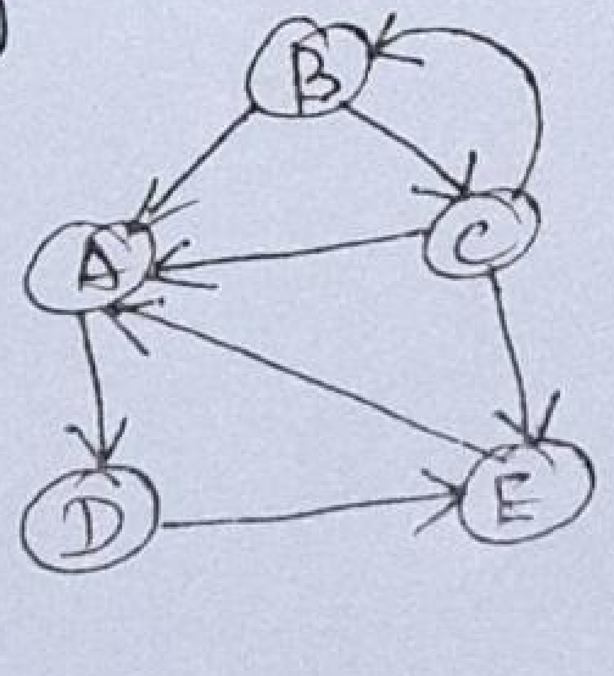
connected graph



strongly connected graph.



Weakly connected graph.



Real Time Example:

Consider an Airport system. Each airport is considered as a vertex. A route from one airport to another airport is declared as an edge. Each edge can have its own weight representing time, distance or cost of the flight. This graph can be assigned as a directed graph. This can be said as strong connected graph because there ~~no~~ exists a route from airport1 to airport2 and vice versa.

Traffic flow can be modeled by a graph. Each street intersection can be represented as vertex and each street is an edge. The weight can be represented with speed limit or a capacity.

Representation of Graphs:

Two types of representation

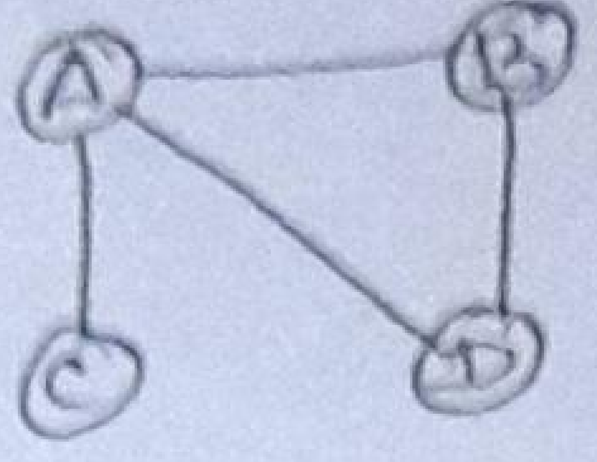
i) Adjacency matrix representation

ii) Adjacency List representation.

Adjacency matrix representation:

A dense graph can be represented using the adjacency matrix representation. This uses a two dimensional array.

(eg)



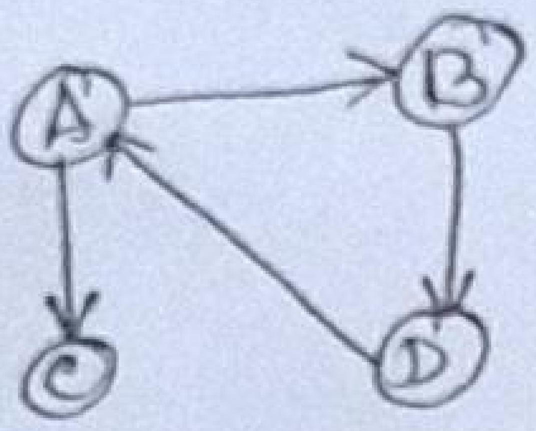
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Vertices : A, B, C, D

Edges : (AC), (AB), (AD), (BD)

Matrix is symmetrical.

eg)



	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	1	0	0	0

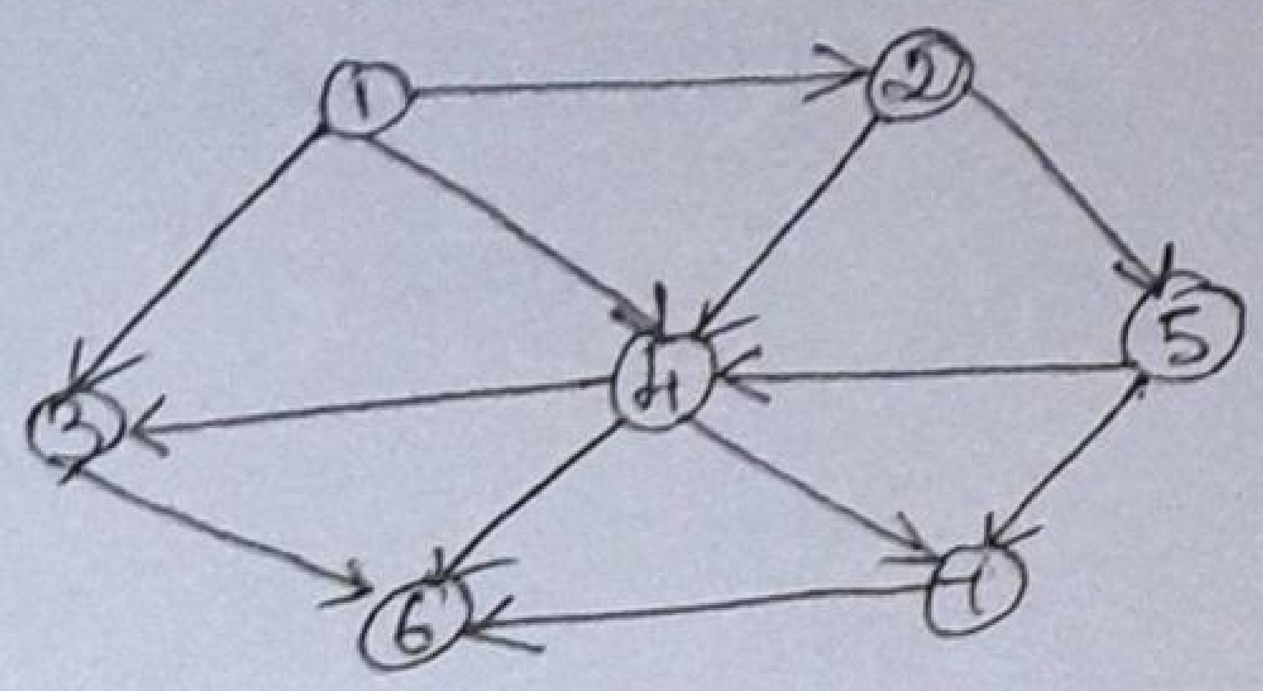
Directed graph.

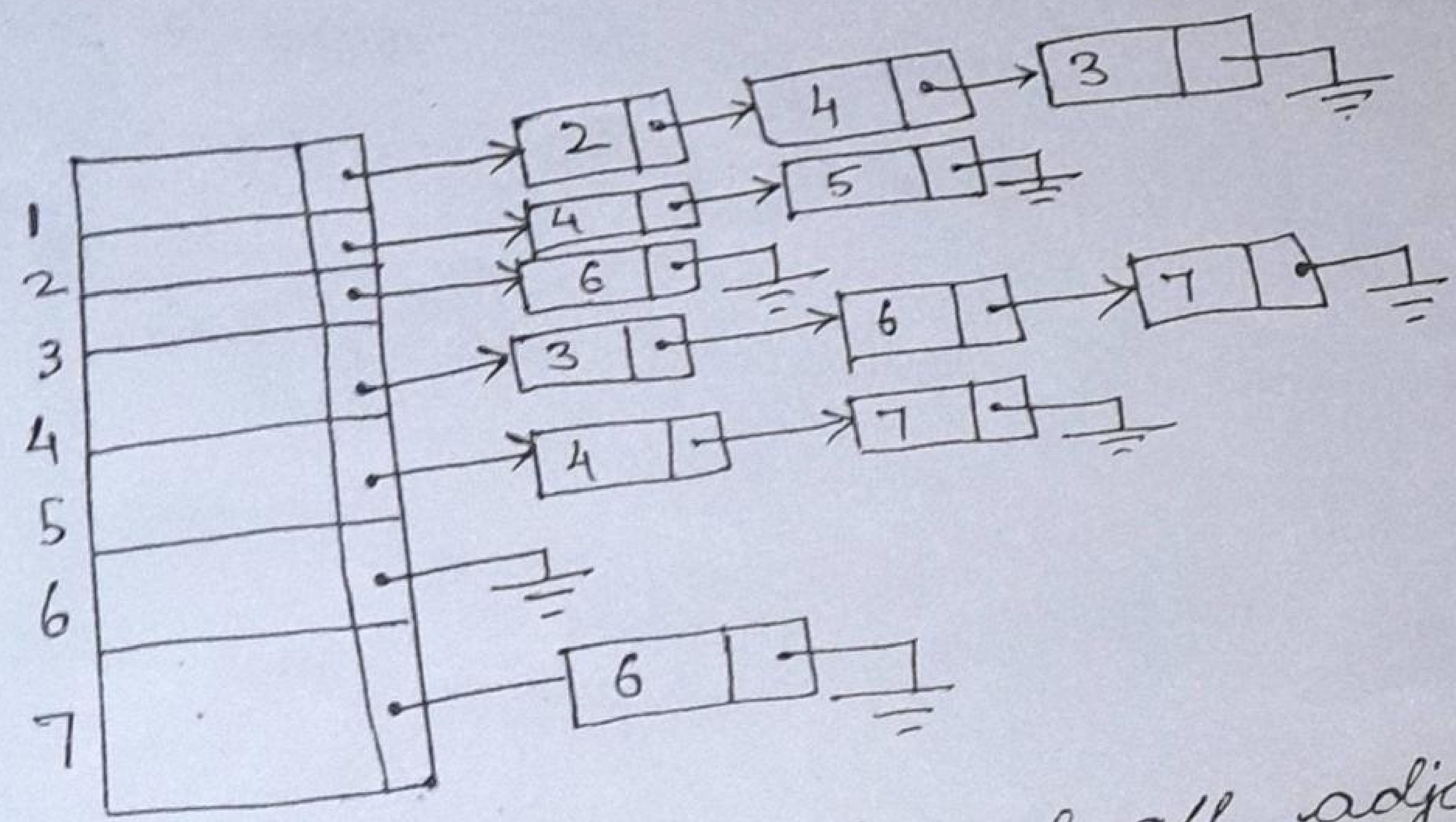
For each edge (u, v), we set A[u][v] = 1; otherwise 0.

Adjacency List Representation:

Adjacency list representation is used for sparse graph. Sparse graph denotes graph with few edges.

eg) Directed graph.





For each vertex a list of all adjacent
Vertices are maintained.
Adjacency lists are standard way to represent
graph. In an undirected graph, each edge (u,v)
appear in two lists, so the space usage
essentially doubles.

Topological Sort:

A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.

Topological ordering is not possible, if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

To implement the topological sort, perform the following steps.

① Find the indegree for every vertex.

② Place the vertices whose indegree is 0 on the empty queue.

③ Dequeue the vertex v and decrement the indegree's of all its adjacent vertices.

④ Enqueue the vertex on the queue if its indegree falls to zero.

⑤ Repeat step 3 & 4 until the queue becomes empty

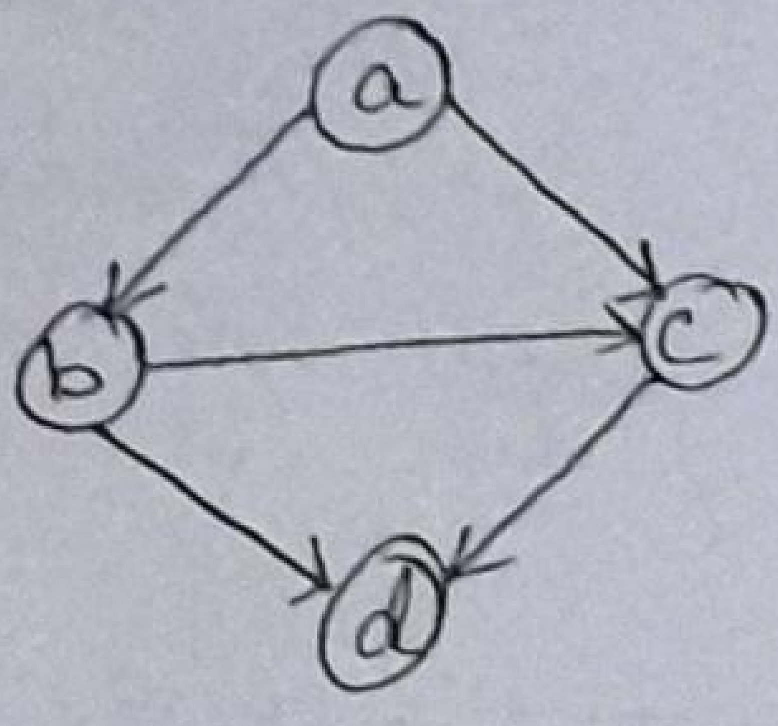
⑥ The topological ordering is the order in which the vertices dequeued.

To find the indegree '0' of a vertex the function FindNewVertexOfIndegreeZero is used. This function returns NotAVertex if no such vertex exists, this indicates that the graph has a cycle.

Pseudocode to perform Topological Sort Using Queue

```
Void TopSort (Graph G)
{
  Queue Q;
  int counter = 0;
  Vertex V, W;
  Q = CreateQueue (NumVertex);
  MakeEmpty (Q);
  for each vertex V
     if ( Indegree [V] = = 0)
        Enqueue (V, Q);
   While (! IsEmpty (Q))
   {
    V = Dequeue (Q);
    TopNum [V] = ++ Counter ;
   for each W adjacent to V
       if ( - - Indegree [W] = = 0 )
          Enqueue (W, Q);
   }
   if (Counter != NumVertex)
       Error ("Graph has a cycle");
   DisposeQueue (Q);
 }
```

Eg for topological sort.



	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

Step 1:

Number of 1's present in each column of adjacency matrix represents the indegree of the corresponding vertex.

ID [a] = 0   ID [b] = 1   ID[c] = 2   ID[d] = 2

Step 2:

Enqueue the vertex whose indegree is 0. Vertex 'a' is 0, so place it in the queue.

Step 3:

Dequeue vertex 'a' and decrement the indegree's of its adjacent vertex 'b' & 'c'.

Now ID[b] = 0 & ID [c] = 1

Now enqueue the vertex 'b' as its indegree becomes zero.

Step 4:

Dequeue vertex 'b' and decrement the indegree's of its adjacent vertex 'c' & 'd'. Now, ID [c] = 0 & ID [d] = 1

Now enqueue 'c' as its indegree falls to zero.

Step 5:

Dequeue vertex 'c' and decrement the indegrees of its adjacent vertex 'd'. Hence ID [d] = 0

Now enqueue the vertex 'd' as its indegree falls to zero.

Step 6:

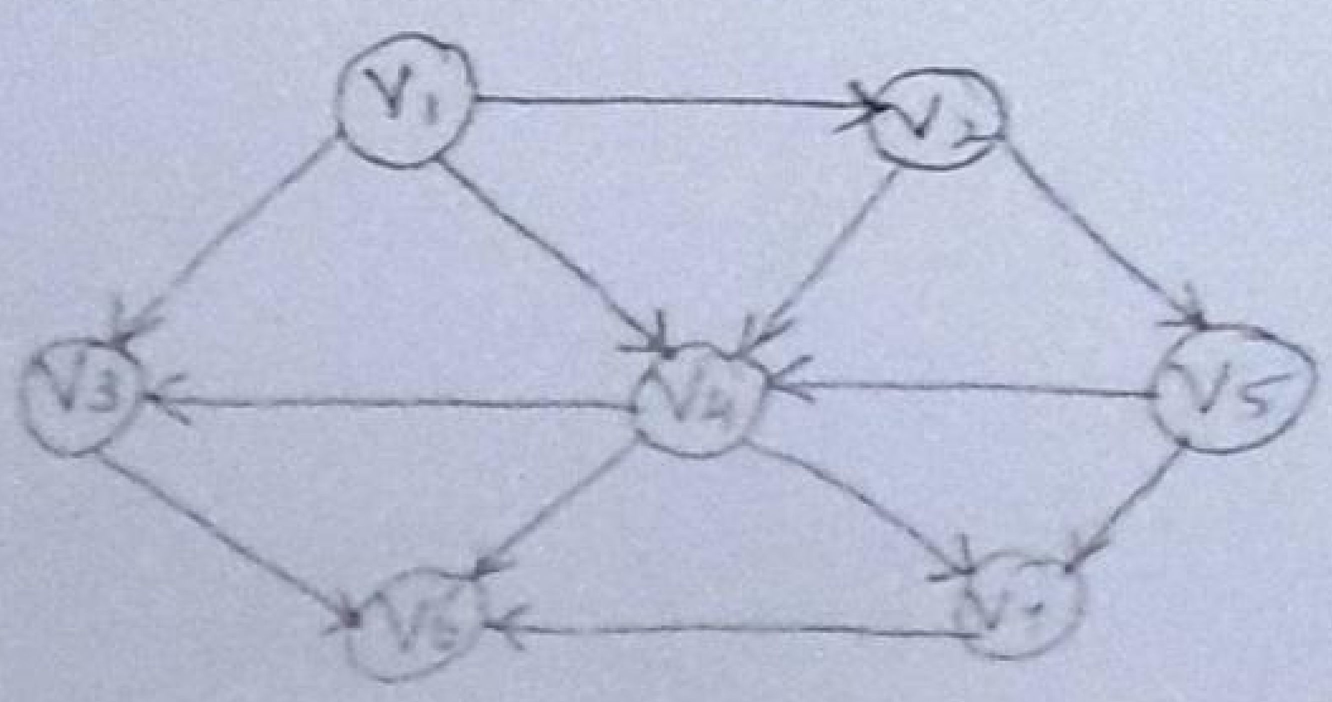
Dequeue the vertex 'd'.

Step 7:

As the queue becomes empty, topological ordering is performed, which is nothing but, the order in which the vertices are dequeued.

Vertex	S1	S2	S3	S4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	0
d	2	2	1	0
Enqueue	a	b	c	d
Dequeue	a	b	c	d

Result of applying topological sort to the graph



V1 V2 V5 V4 V3 V7 V6

Breadth-First Traversal.

A graph traversal is a systematic way of visiting the nodes in a specific order. There are two types of graph traversal namely.

- Depth first traversal
- Breadth first traversal.

Breadth first traversal:

BFS of a graph G starts from an unvisited vertex u. Then all unvisited vertices v_i adjacent to u are visited and then all unvisited vertices w_j adjacent to v_i are visited and so on. The traversal terminates when there are no more nodes to visit. Breadth first search uses a queue data structure to keep track of the order of nodes whose adjacent nodes are to be visited.

Steps to implement breadth first search:

Step 1: Choose any node in the graph, designate it as the search node and mark as visited.

Step 2: Using the adjacency matrix of the graph, find all the unvisited adjacent nodes to the search node and enqueue them into the queue Q.

Step 3: Then the node is dequeued from the queue. Mark that node as visited and designate it as the new node.

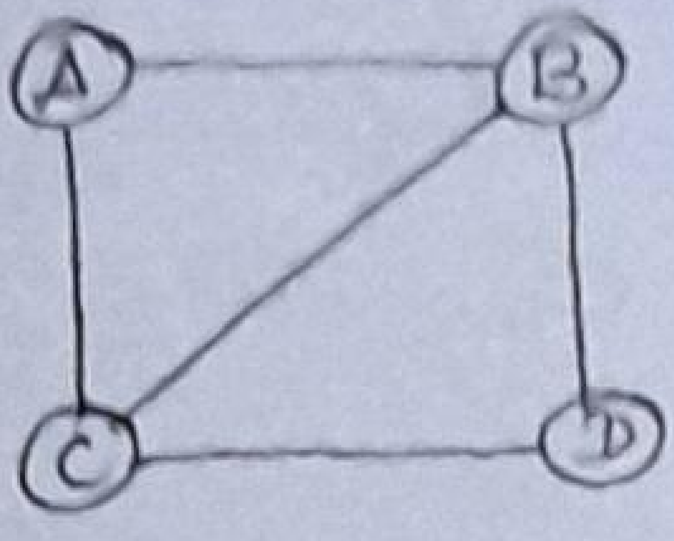
Step 4: Repeat step 2 & 3 using the new search node.

Step 5: This process continues until the queue Q which keeps track of the adjacent nodes is empty.

Routine for BFS:

```
Void BFS (Vertex u)
{
  Initialize Queue Q;
  visited [u] = 1;
  Enqueue (u, Q);
  while ( ! IsEmpty (Q))
  {
    u = Dequeue (Q);
    print u;
    for all vertices v adjacent to u do
      if (visited [v] == 0) then
      {
        Enqueue (v, Q);
        visited [v] = 1;
      }
  }
}
```

Example:



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Adjacency matrix

Implementation:

1. Let 'A' be the source vertex. Mark it as visited.

A	

2. Find the adjacent unvisited vertices of 'A' and enqueue them into the queue.
Here B and C are adjacent nodes of A.
B & C are enqueued

B	C	...

3. Then vertex 'B' is dequeued and its adjacent vertices C & D are taken from the adjacency matrix for enqueuing.
Since vertex C is already in the queue, vertex D alone is enqueued.

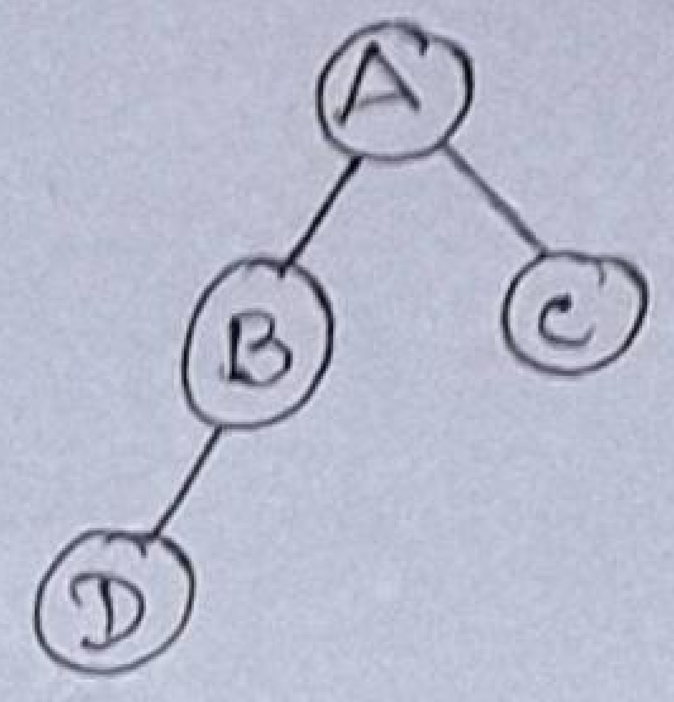
C	D	...

Here B is dequeued, D is enqueued.

4. Then vertex C is dequeued and its adjacent vertices A, B & D are found out. Since vertices A & B are already visited and vertex D is also in the queue. No enqueue operation takes place. Here C is dequeued

D	...

5. Then vertex D is dequeued. This process terminates as all the vertices are visited and the queue is also empty.



Breadth first spanning tree.

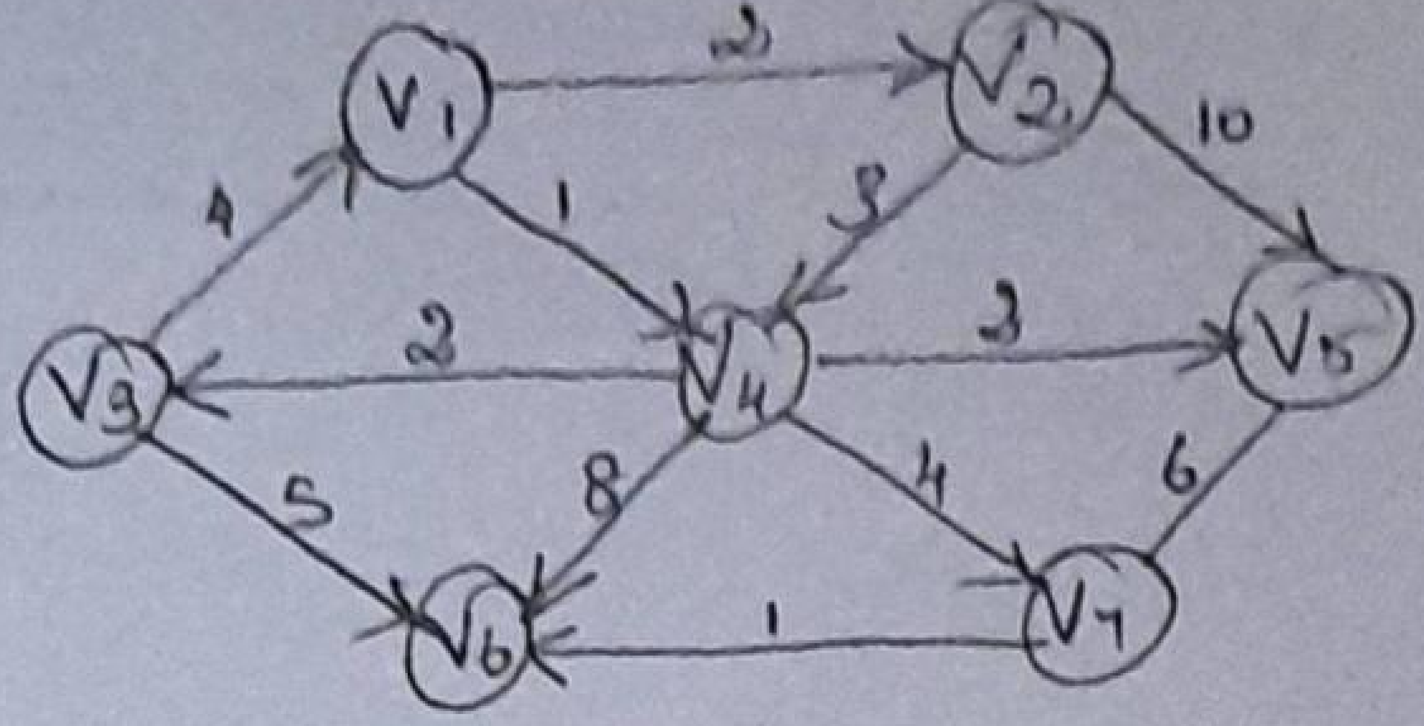
<u>Applications of breadth first search:</u>

- To check whether the graph is connected or not.

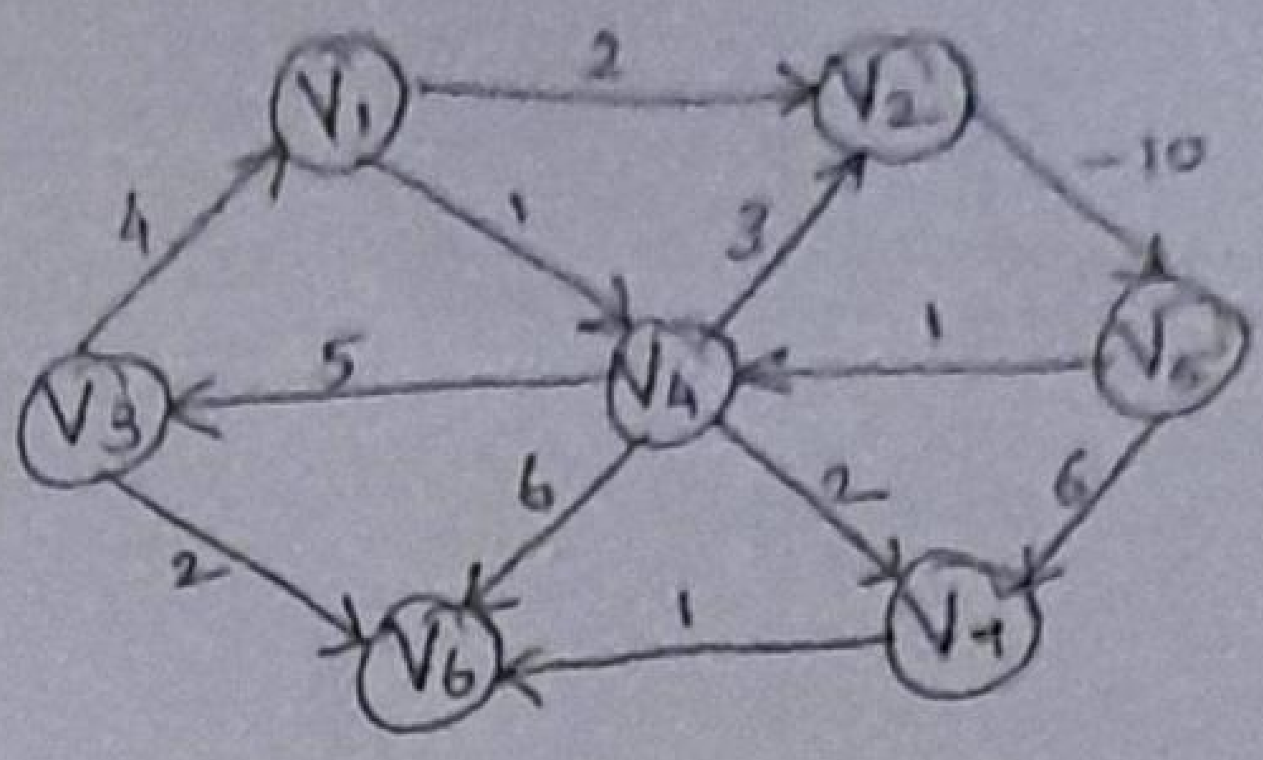
<u>SHORTEST PATH ALGORITHMS</u>

The input to the shortest path algorithm is a weighted graph associated with each edge (V_i, V_j) is a cost $C_{i,j}$ to traverse the arc. The cost of the path $V_1, V_2, \ldots V_N$ is $\sum_{i=1}^{N-1} C_{i,i+1}$. This is refered to as the weighted path length.

In the example(1) the shortest weighted path from V_1 to V_6 has a cost of 6 and goes from V_1 to V_4 to V_7 to V_6. The shortest unweighted path between these vertices is 2



Example ①



Example ②

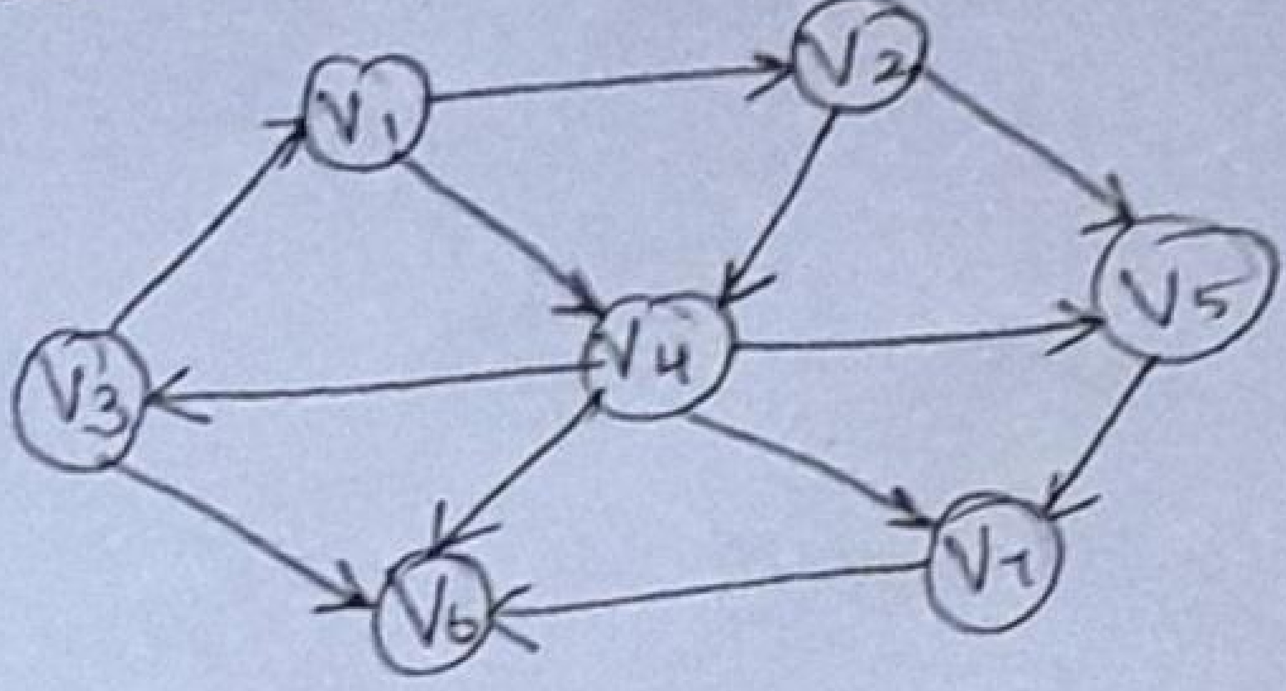
Consider example ② the path from V_5 to V_4 has cost 1, but a shortest path exists by following the loop V_5, V_4, V_2, V_5, V_4 which has cost -5. This path is still not the shortest, because we could stay in the loop arbitrarily long. Thus the shortest path between these two points is undefined. This loop is known as a negative cost cycle.

Types of shortest path algorithms:

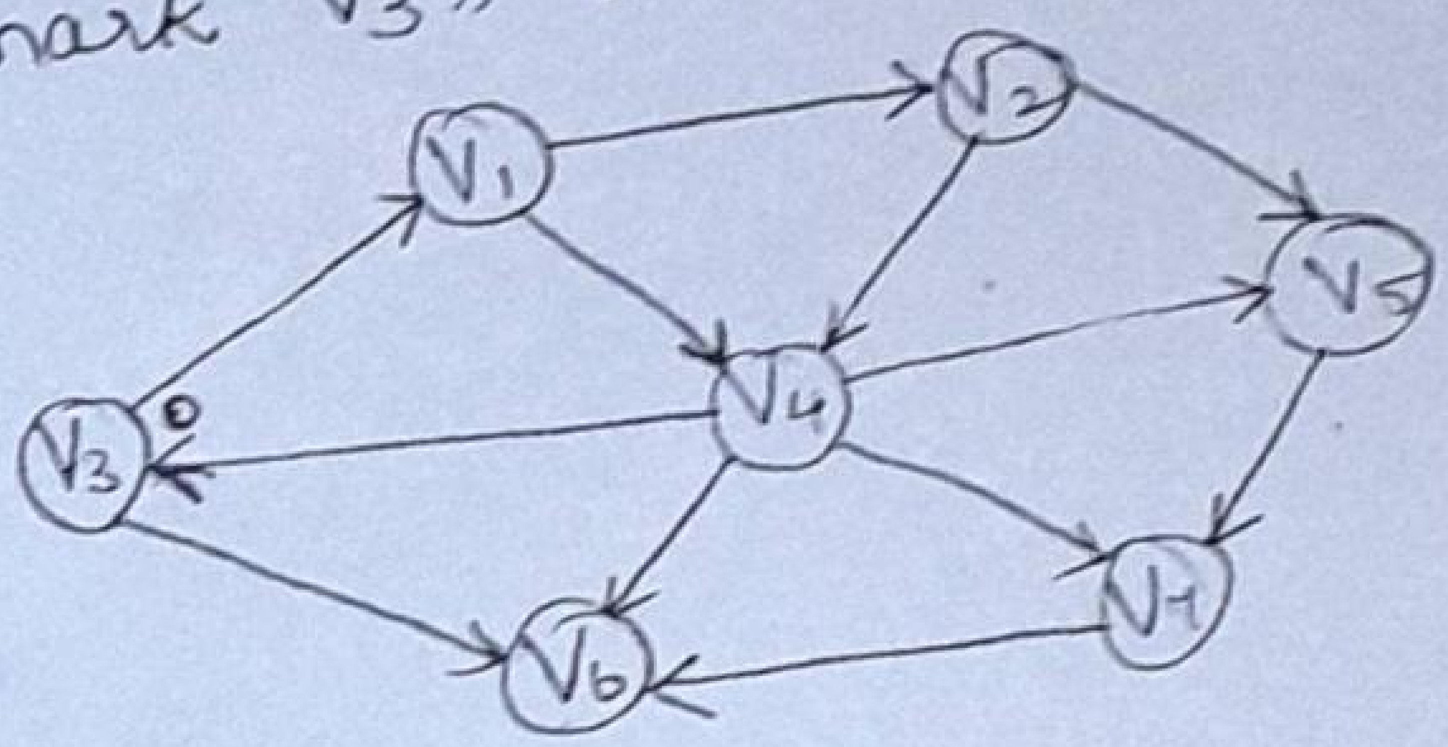
1. Unweighted Shortest path problem
2. Weighted Shortest path problem or Dijkstra's Algm
3. Weighted shortest path with negative edges
4. Acyclic graphs.

Unweighted shortest paths:

For an unweighted graph assign all edge weight as 1. The aim of the problem is to find the shortest path from source 's' to all other vertices.



Choose S to be V_3. So shortest path from S to V_3 is 0. Now mark V_3's distance from S.

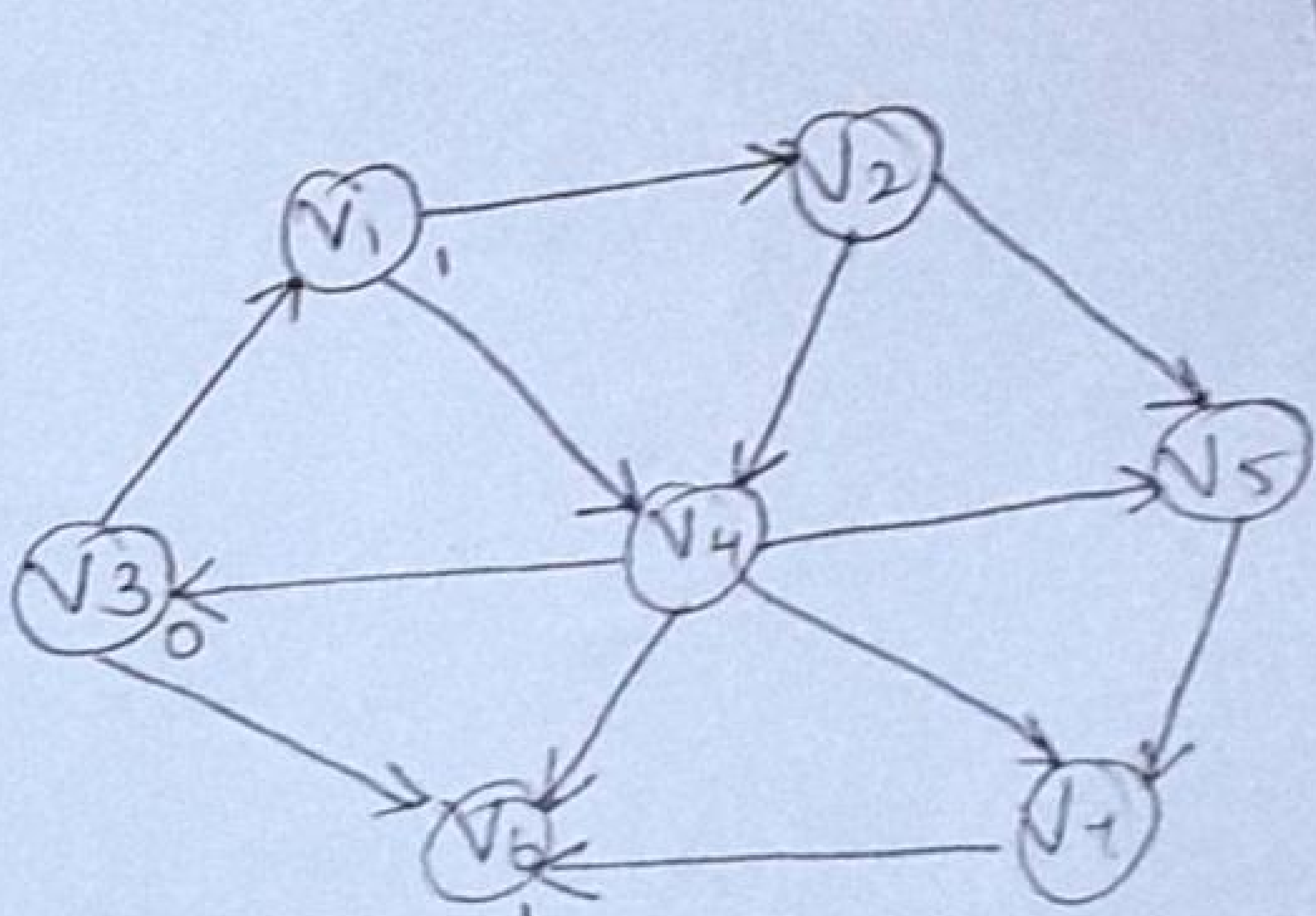


V	Known	d_v	P_v
V_1	0	∞	0
V_2	0	∞	0
V_3	0	0	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

Now mark the adjacent node to S as 1. Since the distance from S to its adjacent node is 1.

V_3 dequeued

V	Known	d_v	P_v
V_1	0	1	V_3
V_2	0	∞	0
V_3	1	0	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	1	V_3
V_7	0	∞	0

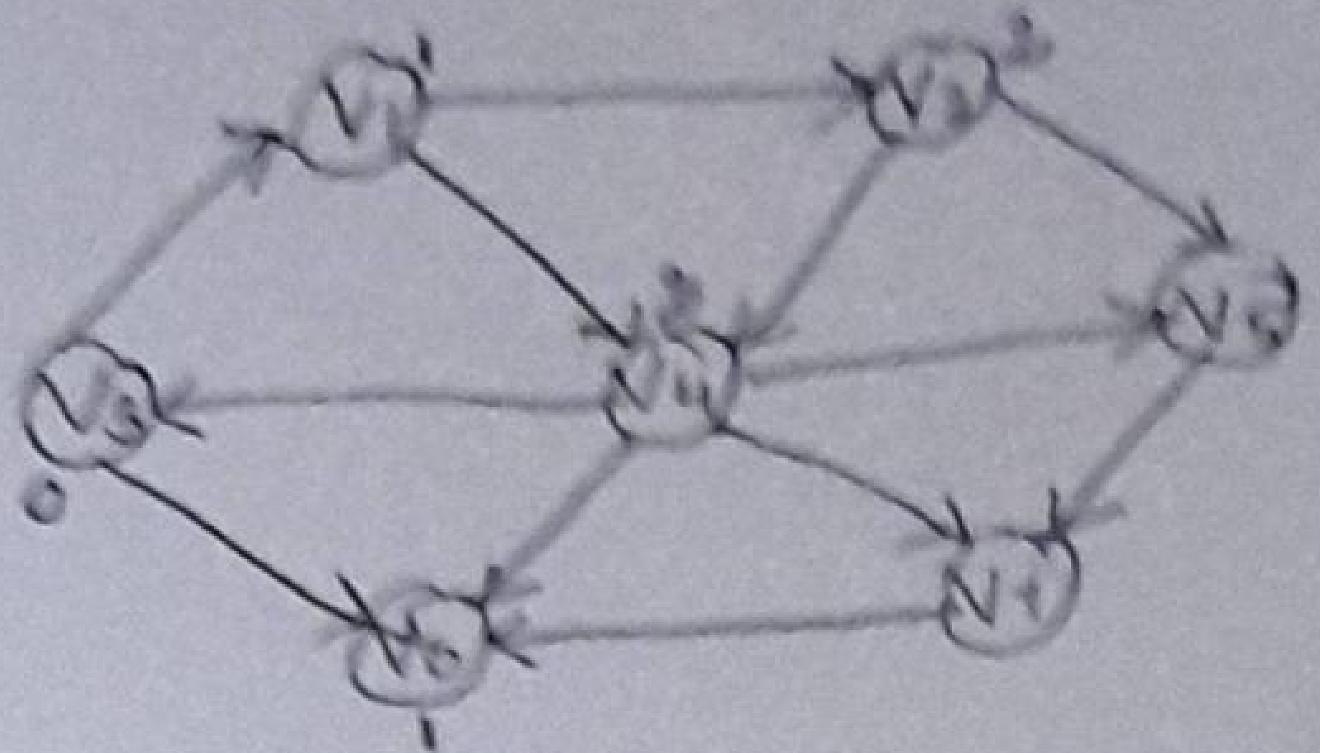


This denotes the distance from source 's' to node V_1 and V_6 are 1.

Now find the vertices whose shortest path from S is exactly 2, by finding all the vertices adjacent to V1 and V6 (i.e. the vertices at distance 1)

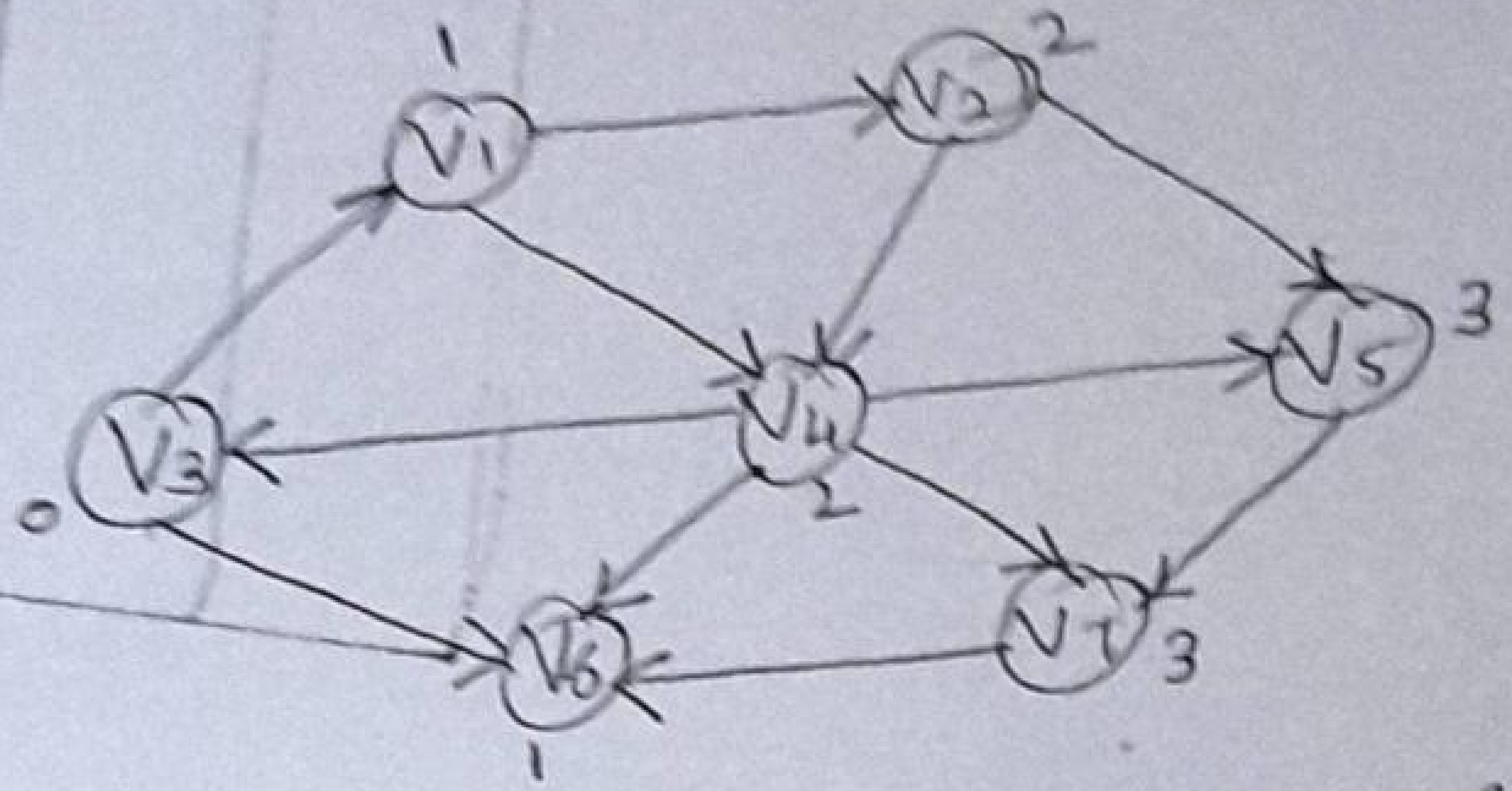
Vdequeue

V	Known	dv	Pv
V1			
V2			
V3			
V4			
V5			
V6			
V7			



The distance of V2 & V4 from S is 2.

~~From~~ the recently evaluated edge vertices find the distance of the adjacent vertices as.



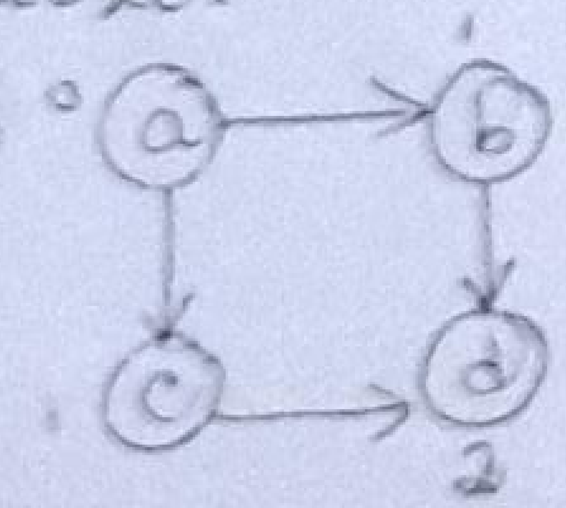
Now all the vertices are marked with its distance, from 'S'.

This strategy for searching a graph is known as breadth-first search.

Initial State				V3 Dequeued			V1 Dequeued			V6 Dequeued			V2 Dequeued			V4 Dequeued			V5 Dequeued			V7 Dequeued		
V	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv
V_1	0	∞	0	0	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3
V_2	0	∞	0	0	∞	0	0	2	V_1	0	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1
V_3	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
V_4	0	∞	0	0	∞	0	0	2	V_1	0	2	V_1	0	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1
V_5	0	∞	0	0	∞	0	0	∞	0	0	∞	0	0	3	V_2	0	3	V_2	1	3	V_2	1	3	V_2
V_6	0	∞	0	0	1	V_3	0	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3
V_7	0	∞	0	0	∞	0	0	∞	0	0	∞	0	0	∞	0	0	3	V_4	0	3	V_4	1	3	V_4
Q_i	V_3			V_1, V_6			V_6, V_2, V_4			V_2, V_4			V_4, V_5			V_5, V_7			V_7			Empty		

Shortest distance from V_3 to $V_1, V_2, V_3, V_4, V_5, V_6$ and V_7 are 1, 2, 0, 2, 3, 1, 3 respectively. The running time of this algorithm is $O(|E| + |V|)$ if adjacency list is used.

eg)



Initial				a Dequeued			b Dequeued			c Dequeued			d Dequeued		
V	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv	Known	dv	Pv
a	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0
b	0	∞	0	0	1	a	1	1	a	1	1	a	1	1	a
c	0	∞	0	0	1	a	0	1	a	1	1	a	1	1	a
d	0	∞	0	0	∞	0	0	2	b	0	2	b	1	2	b
Q_i	a			b, c			c, d			d			Empty		

Scanned by TapScanner

Routine for unweighted Shortest Path algorithm.

```
Void Unweighted (Table T)
{
 int currDist;
 Vertex V, W;
 for (currDist = 0; currDist < numVertex ; currDist ++)
  for each vertex V
    if (!T[V]. Known && T[V] . Dist == CurrDist)
    {
     T[V]. Known = True;
     for each W adjacent to V
       if (T[W]. Dist == Infinity)
       {
        T[W]. Dist = currDist +1;
        T[W]. Path = V;
       }
    }
}
```

Routine for Unweighted Shortest Path algorithm

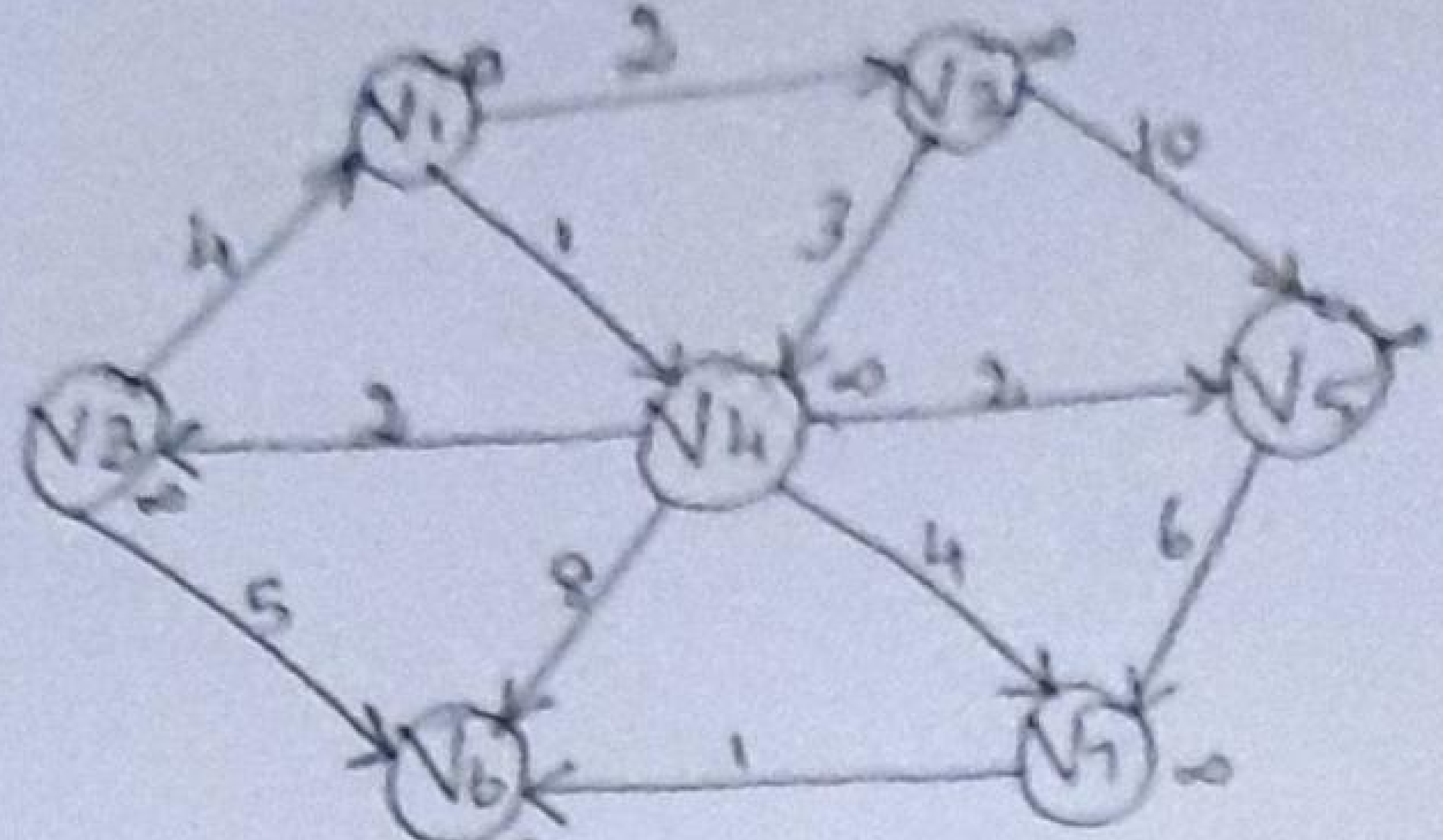
```
Void UnWeighted (Table T)
{
 Queue Q;
 Vertex V, W;
 Q: CreateQueue (NumVertex);
 MakeEmpty (Q);
 Enqueue (S, Q);
 While ( !IsEmpty (Q))
  {
   V: Dequeue (Q);
   T[V].Known = True;
  for each W adjacent to V
    if (T[W].Dist = = Infinity)
    {
      T[W].Dist = T[V].Dist +1;
      T[W].Path = V;
      Enqueue (W, Q);
    }
  }
  DisposeQueue (Q);
 }
```

The running time of this algorithm is,

$O(|E| + |V|)$.

2. Dijkstra's Algorithm:

Dijkstra's algorithm is used to find the shortest path for a directed weighted graph

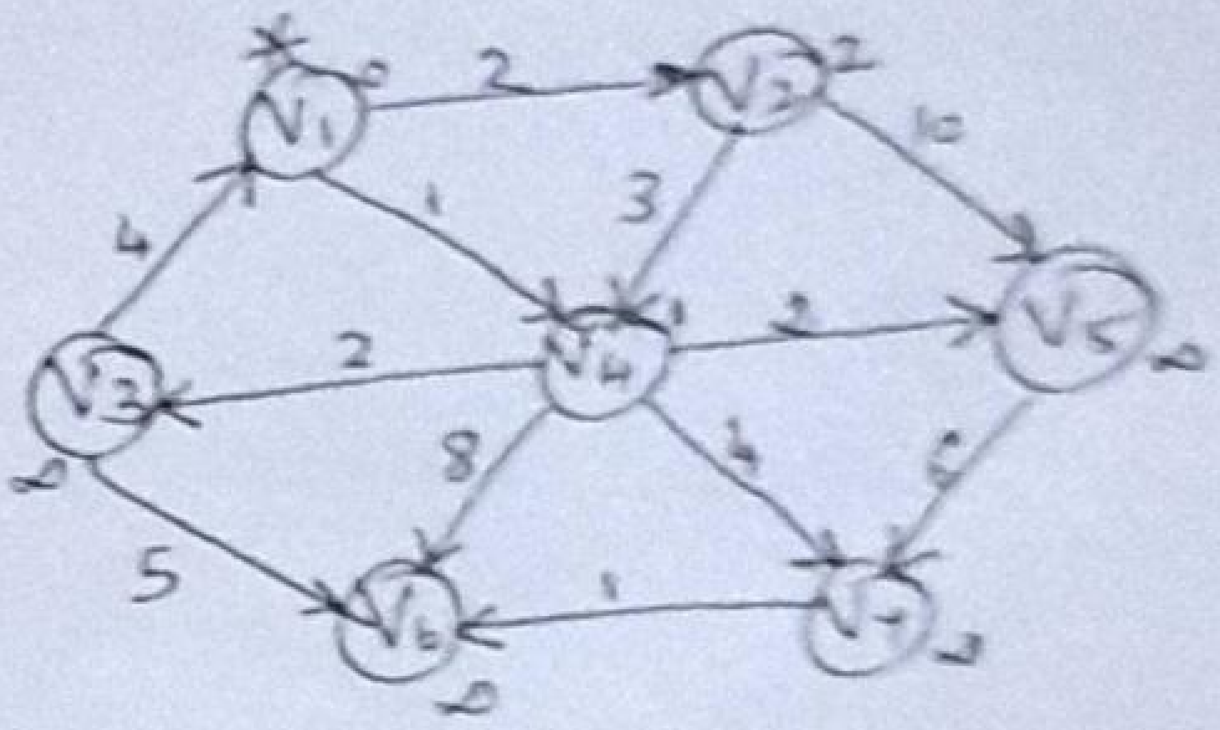


V	Known	dv	Pv
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

In the above graph consider V_1 as the start node.

Declare V_1 as known.

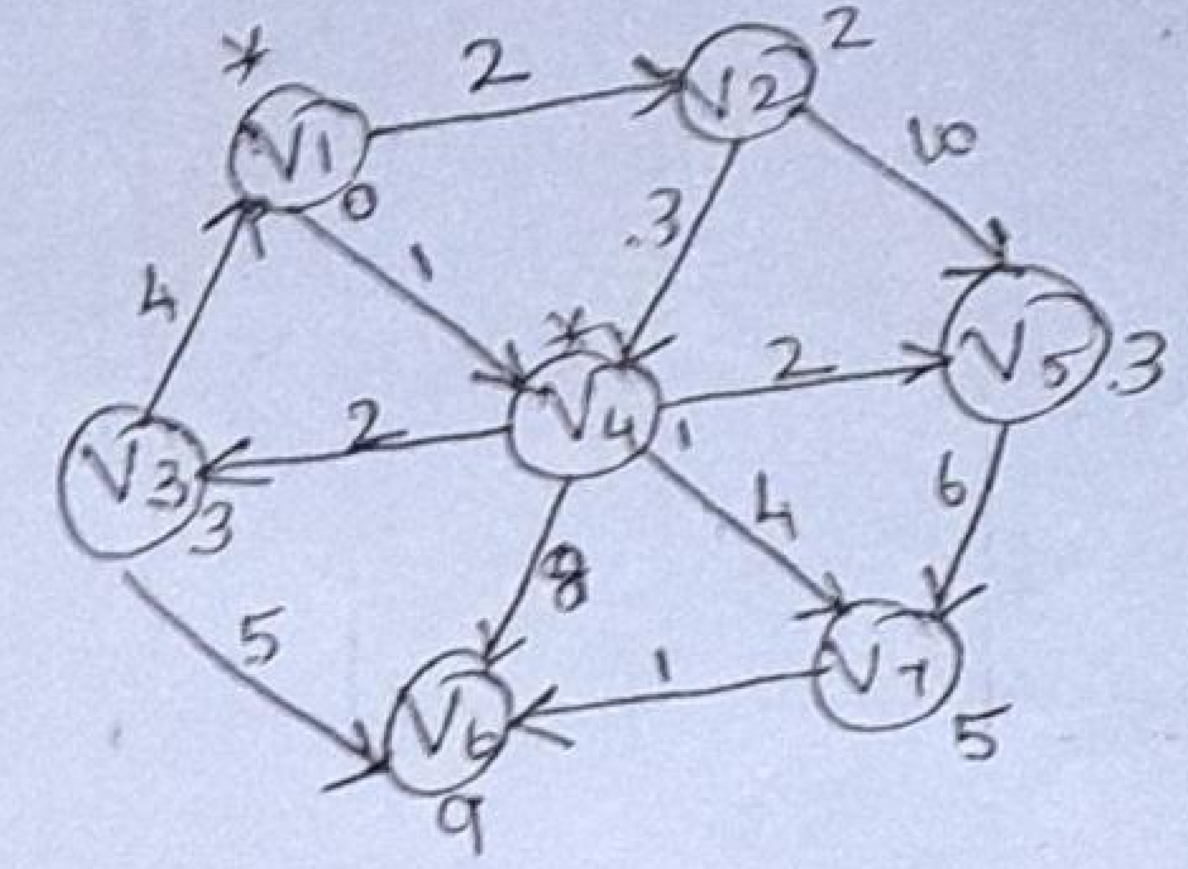
V	Known	dv	Pv.
V_1	1.	0	0
V_2	0	2	V_1
V_3	0	∞	0
V_4	0	1	V_1
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0



Scanned by TapScanner

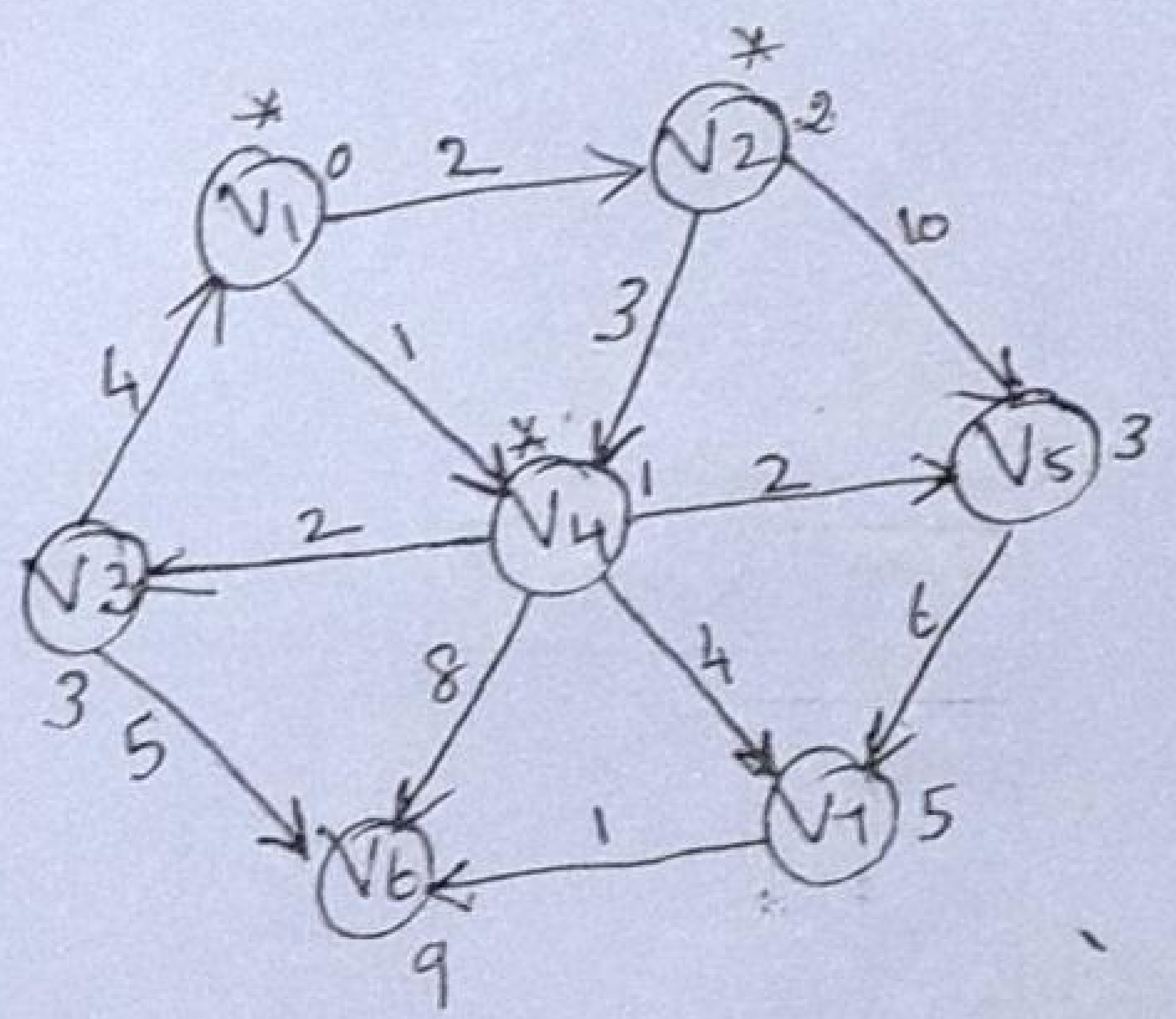
Now the minimum dv value whose vertex is unvisited is V_4. So declare V_4 as known.

V	known	dv	Pv
V_1	1	0	0
V_2	0	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4



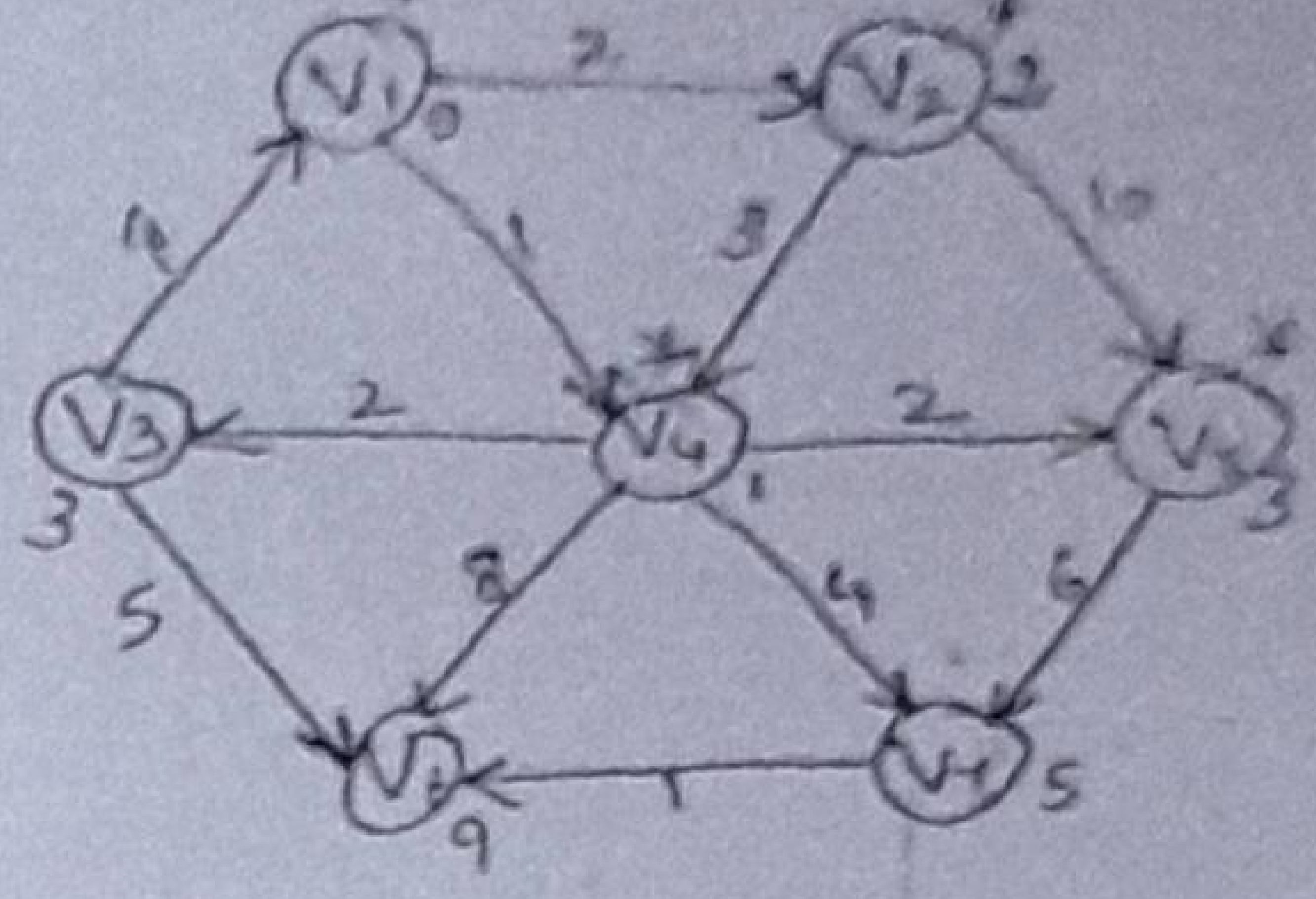
Now mark V_2 as known, because the dv value of V_2 is minimum among the unvisited vertices.

V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

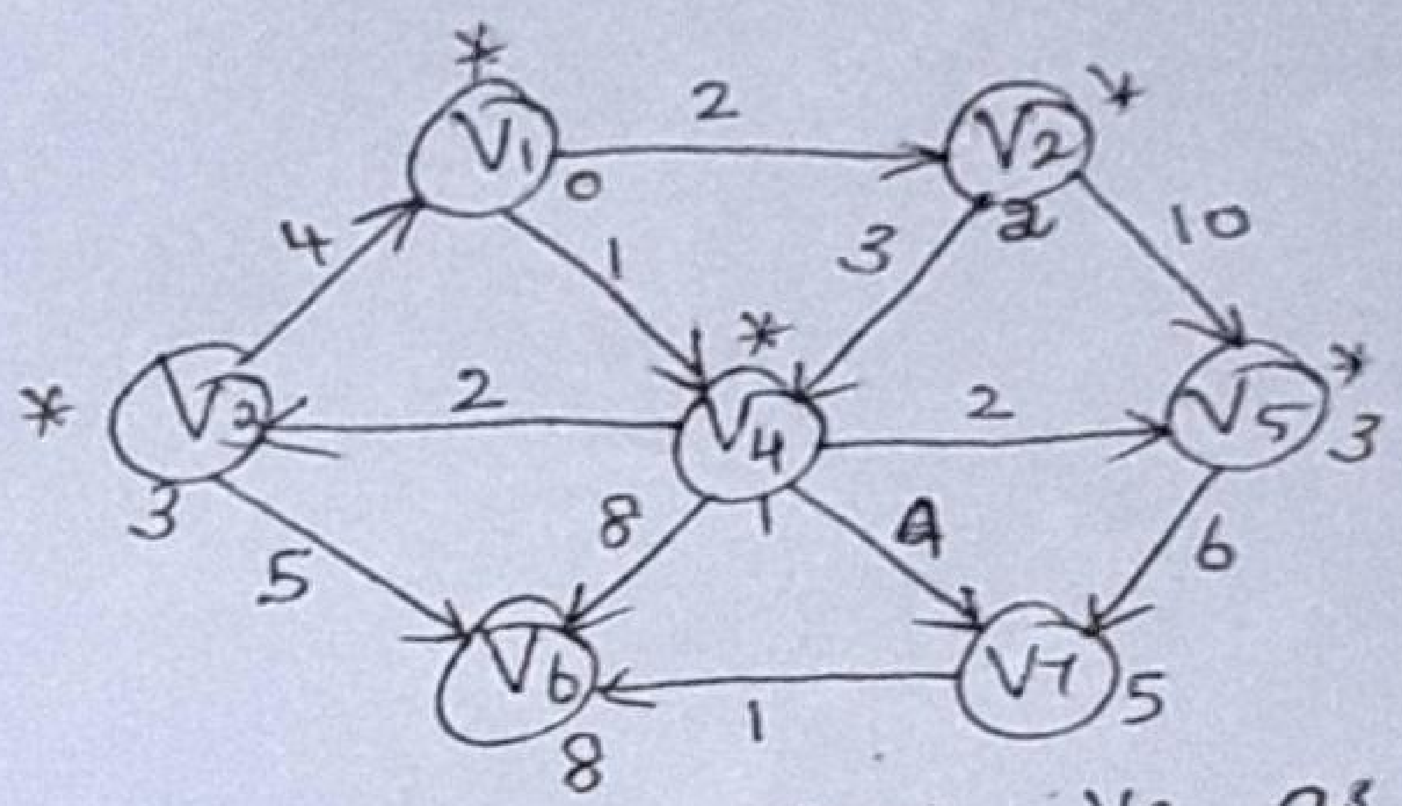


Now mark V_3 and V_5 as known because the dv value of V_3 and V_5 are minimum among the unvisited vertices.

V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	8	V_3
V_7	0	5	V_4



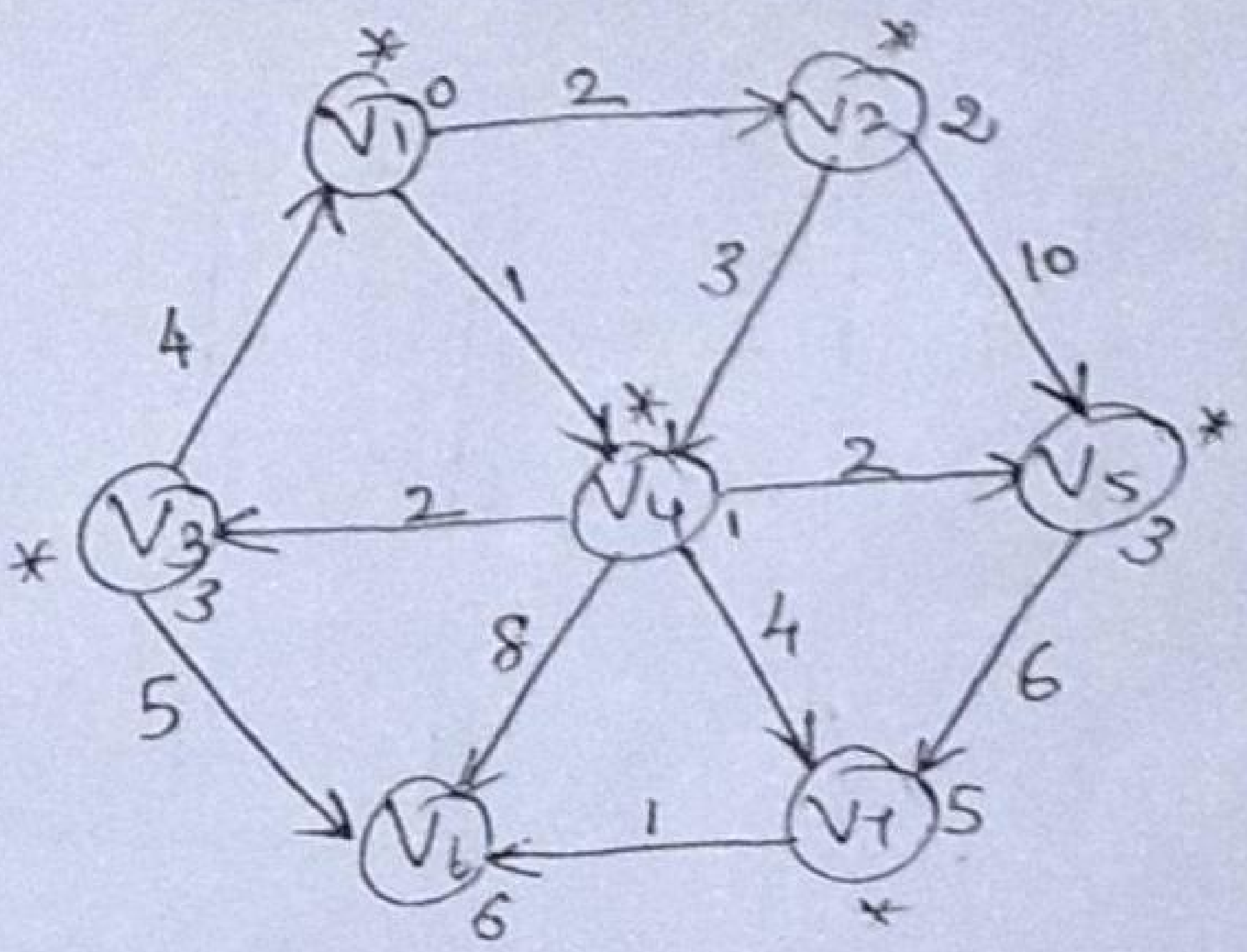
Mark V_5 as known.



Mark V_3 as known.

Now declare V_7 as known.

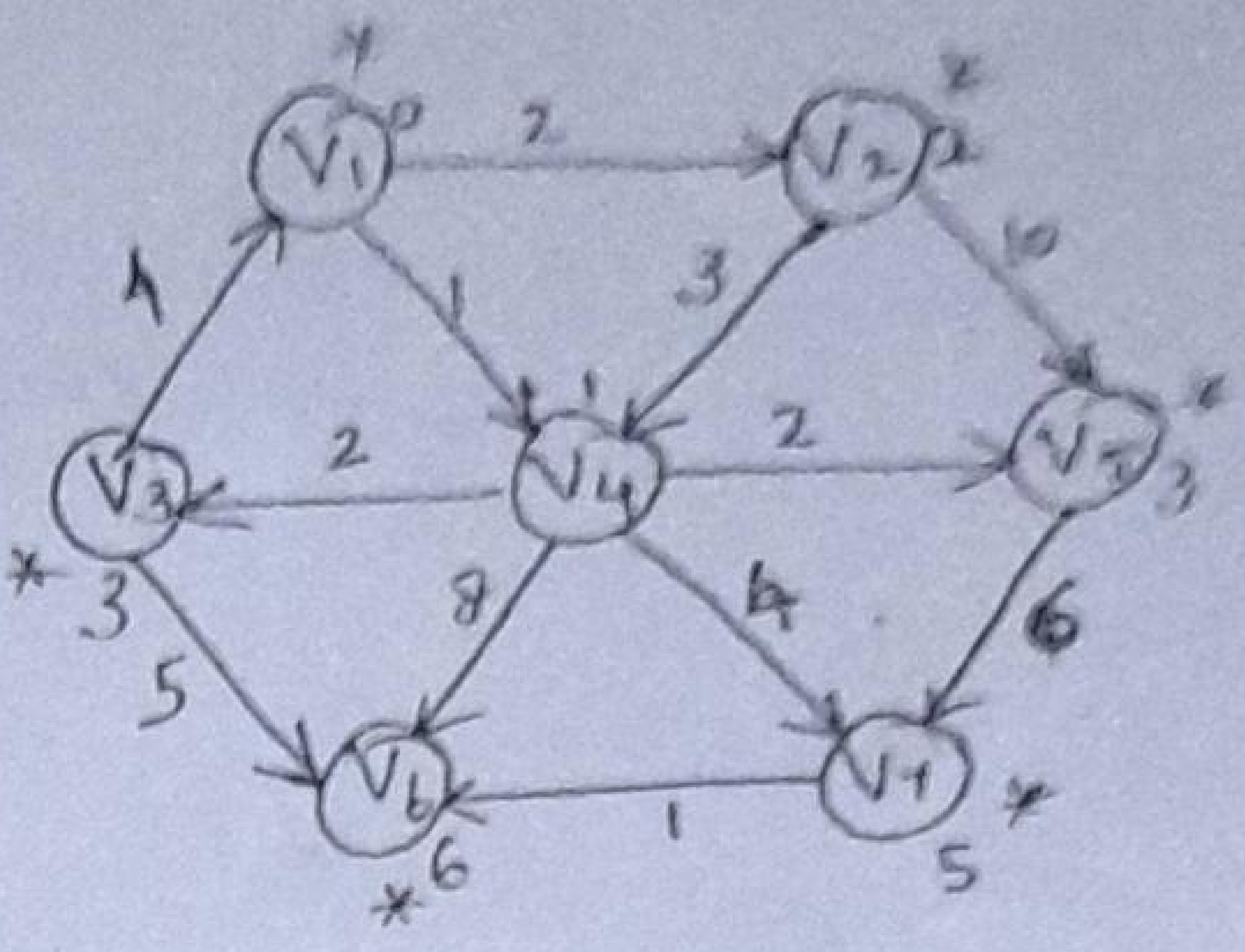
V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	6	V_7
V_7	1	5	V_4



Scanned by TapScanner

Now declare V_6 as marked.

V	known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	1	6	V_7
V_7	1	5	V_4



Declaration for Dijkstra's algorithm.

```
Struct TableEntry
{
 List Header;
 int known;
 DisType Dist;
 Vertex Path;
};
```

Table initialization routine

```
Void InitTable (Vertex Start, Graph G, Table T)
{
int i;
ReadGraph (G, T); // Generates adjacency list
for ( i=0 ; i < NumVertex ; i++)
{
  T[i].known = False;
  T[i].Dist = Infinity
  T[i].Path = NotAVertex;
  }
  T[Start].dist = 0;
  }
```

Routine to print the actual shortest path.

```
Void PrintPath (Vertex V, Table T)
{
 if ( T[V].Path ! = NotAVertex)
    {
     printpath ( T[V].Path, T);
     Printf (" to ");
     }
     printf (" %v", v) ;

    }
```

Routine for Dijkstra's Algorithm:

```
Void Dijkstra ( Table T)
{
 Vertex V, W;
  for ( ; ;)
   {
   V = Smallest unknown distance vertex;
    if (V == NotAVertex)
     break;
    T[V]. Known = True;
     for each W adjacent to V
      if (!T[W]. Known)
      if (T[V]. Dist + CVW < T[W]. Dist)
      {
       Decrease (T[W]. Dist to T[V]. Dist + CVW);
       T[W]. Path = V;
      }
    }
   }
```

This algorithm will work for a graph which does not contain any negative path cost.

The running time of this algorithm is $O(|V|^2)$

Dense graph $|E| = \Theta(|V|^2)$

Sparse graph $|E| = \Theta(|V|)$.

3. Graph With Negative cost:

Dijkstra's algorithm does not work for graph which contains negative edge costs.
A solution to the above problem is to add a constant Δ to each edge cost, to remove the negative edges. Now calculate the shortest path on the new edged graph, then use that result on the original graph.
This solution will not work for a dense graph. A combination of weighted and unweighted algorithms will solve the problem at high run time complexity. The running time of this algorithm is O(|E|·|V|).

```
void weighted_negative( TABLE T )
{
QUEUE Q;
vertex v, w;
 Q = create_queue( NUM_VERTEX );
MakeEmpty( Q );
 Enqueue( s, Q );
while( !IsEmpty( Q ) )
{
v = dequeue( Q );
for each w adjacent to v
if( T[v].dist + cv,w < T[w].dist )
{
T[w].dist = T[v].dist + cv,w ;
T[w].path = v;
 if( w is not already in Q )
enqueue( w, Q );
}
}
dispose_queue( Q );
}
```

4. Acyclic Graph:

Select the vertices to be marked known, in topological order.

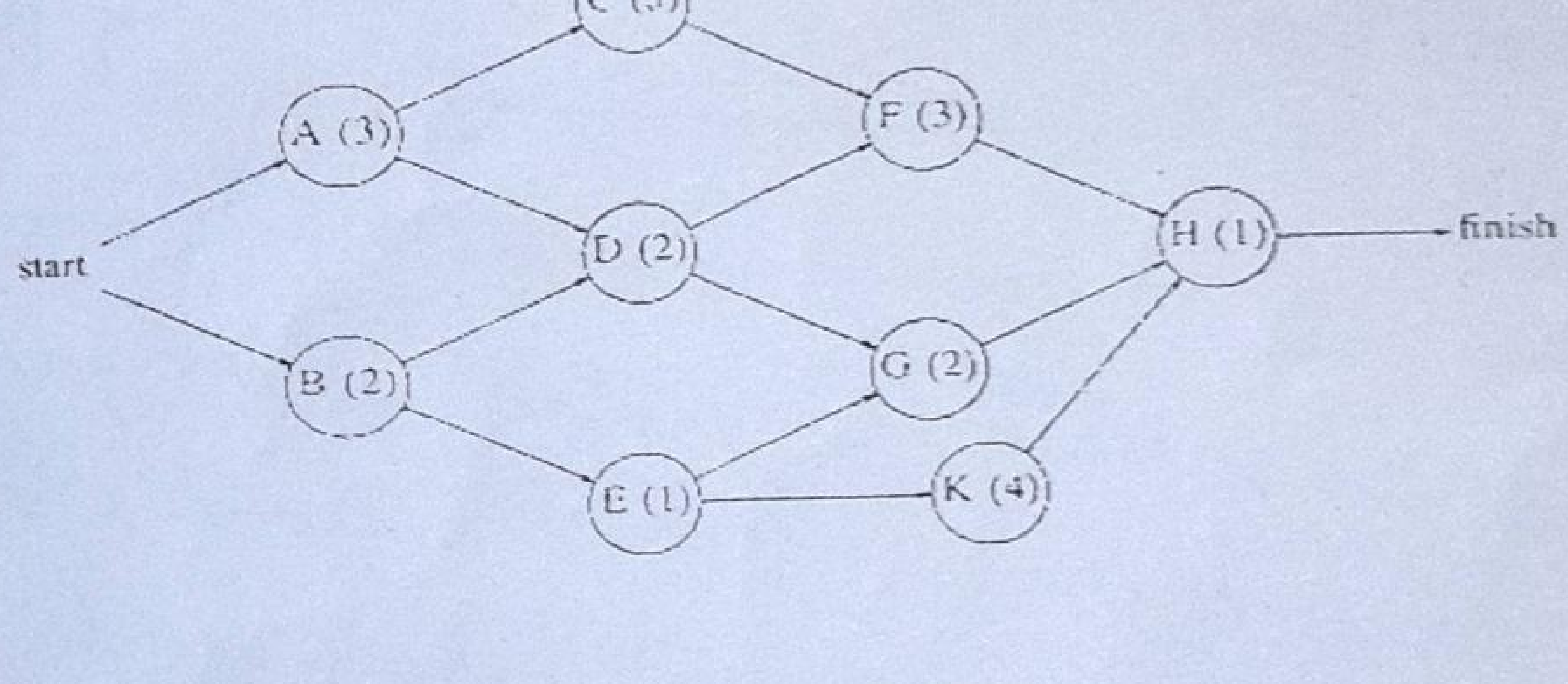
The running time for selecting a vertex is $O(|E| + |V|)$.

Since it is an acyclic graph backward traversing is impossible

Consider each vertex represent a state of an experiment. Edges represent a transaction from one state to another, and edge weight represent the energy release.

The important use of acyclic graph is critical path analysis.

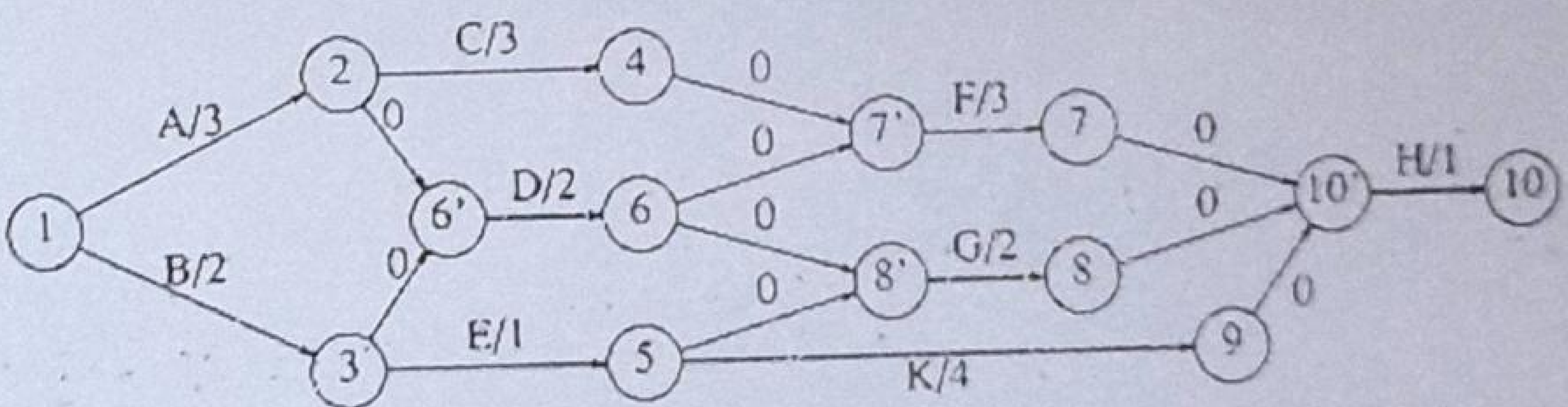
Consider the following example,



Activity-node graph

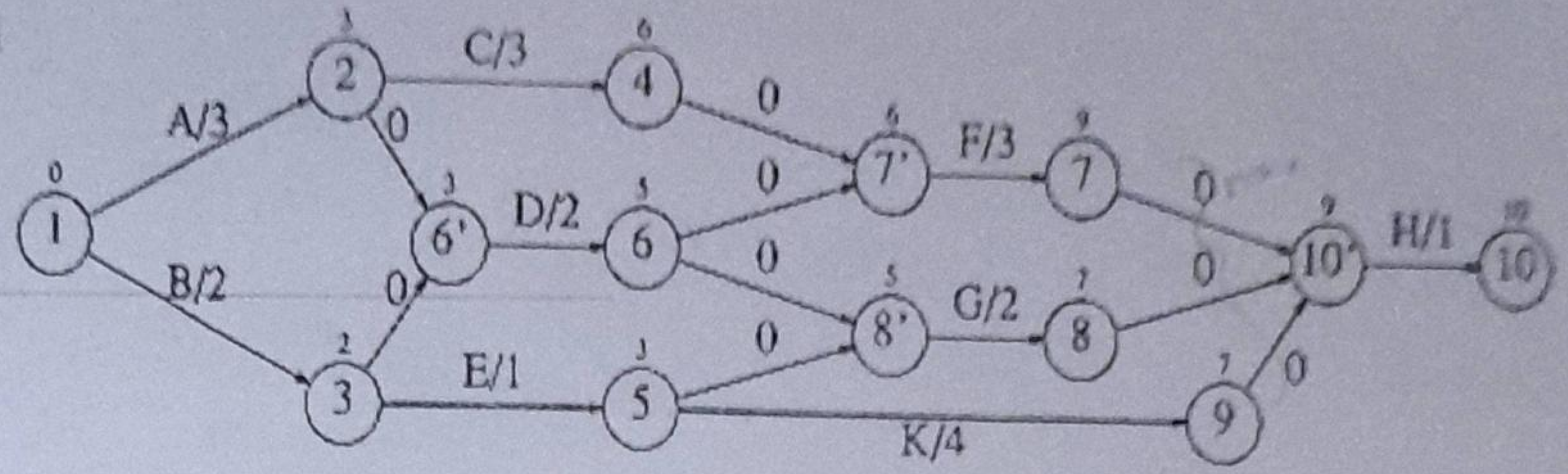
Here each node represent an activity that must be performed, along with the time it takes to complete the activity. This graph is thus known as activity-node graph. The edges represents precedence relationship. i.e, edge (v, w) means that activity v must be completed before activity w may begin.
Any independent activity can be performed in parallel by different servers.

To calculate the earliest completion time (cost of a path), the activity-node graph is converted to an event node graph. This graph can be constructed by adding dummy edges and nodes in case of dependent activities.



Event-node graph for the activity-node graph.

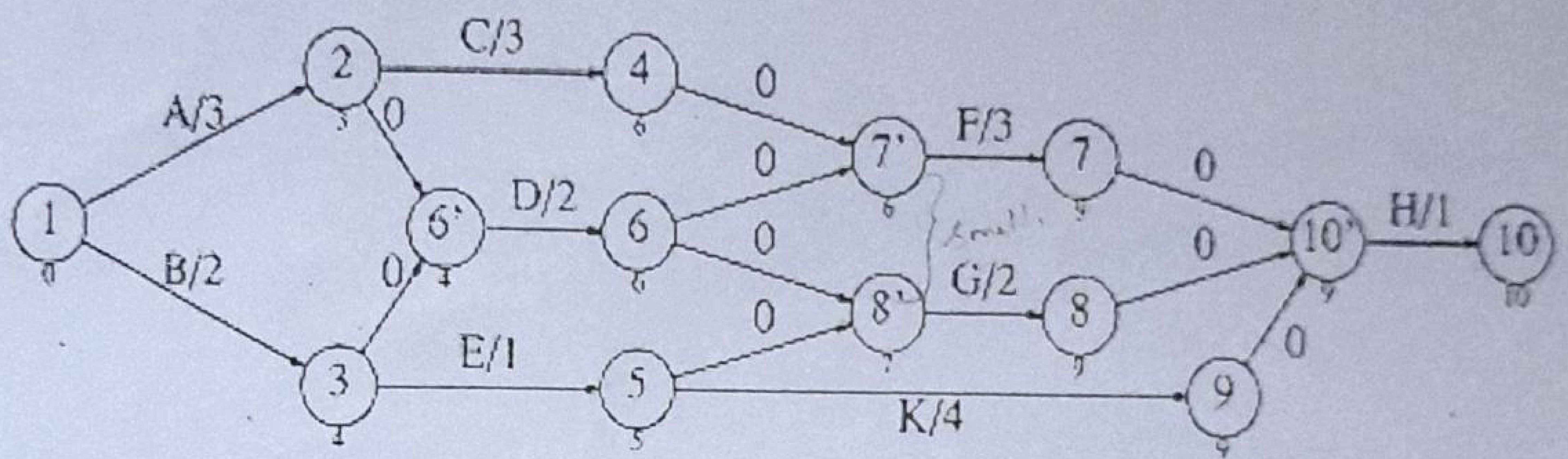
To find the earliest completion of the project, find the length of the longest path from the first event to the last event, the activity node



$EC_1 = 0$

Earliest completion times

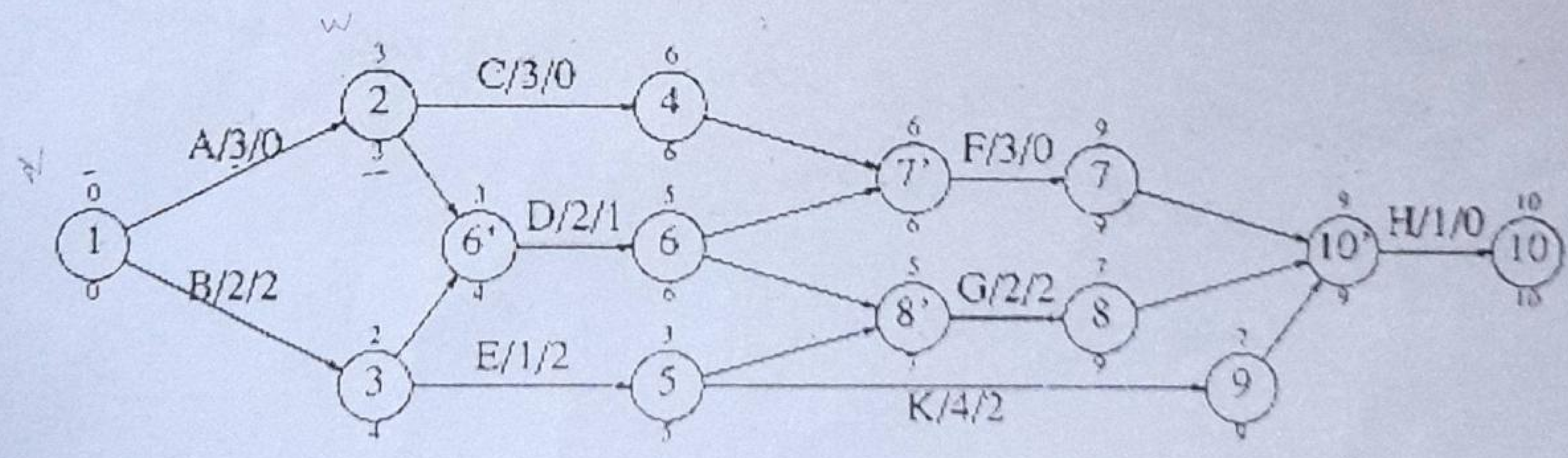
$$EC_w = \max_{(v,w) \in E} (EC_v + C_{v,w})$$



$LC_n = EC_n$

$$LC_v = \min_{(v,v) \in E} (LC_w - C_{v,w})$$

Latest completion times



Earliest completion time, latest completion time, and slack

$$Slack_{(v,w)} = LC_w - EC_v - C_{v,w}$$

graph is converted to event node graph.

From the event node graph the earliest completion time is calculated.

To find earliest completion time, find the length of the longest path from first event to the last event.

EC_i is the earliest completion time.

$$EC_1 = 0$$

$$EC_w = \max_{(v,w) \in E} (EC_v + C_{v,w})$$

Latest time LC_i is the time that each event can finish without affecting the final completion time.

$$LC_n = EC_n$$

$$LC_v = \min_{(v,w) \in E} (LC_w - C_{v,w})$$

Slack time for each edge in the event node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion.

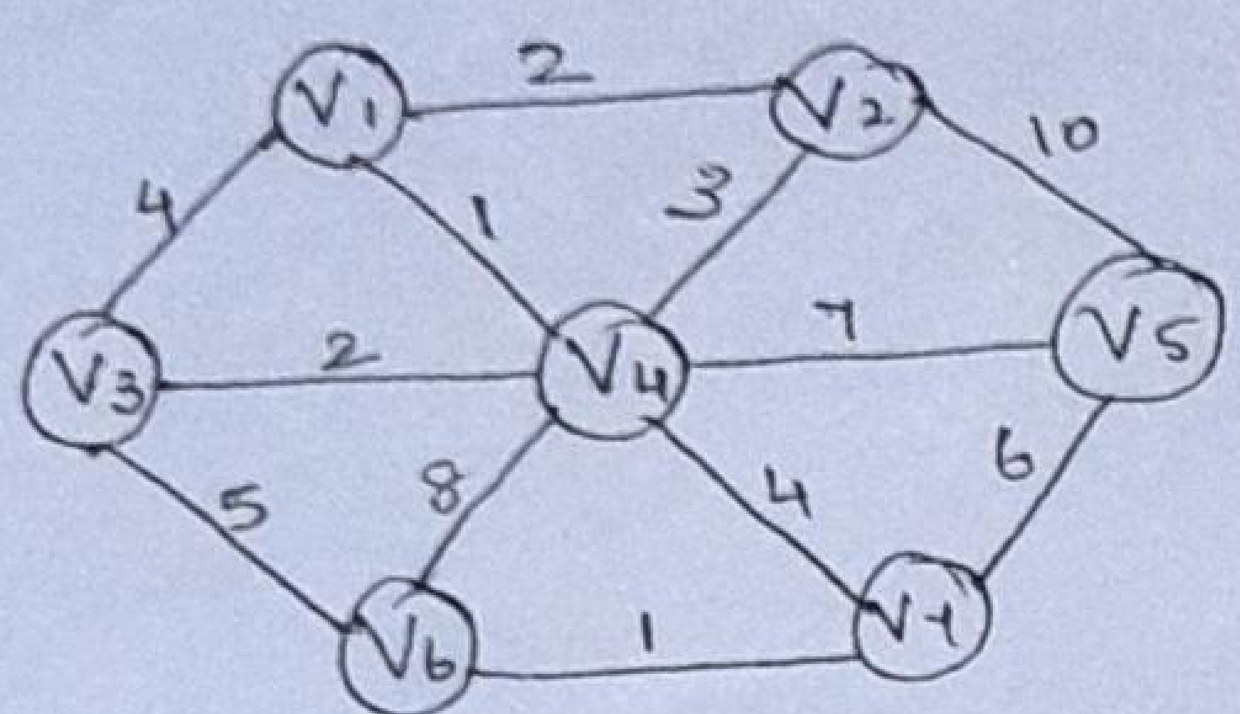
$$slack_{(v,w)} = (LC_w - EC_v - C_{v,w})$$

Scanned by TapScanner

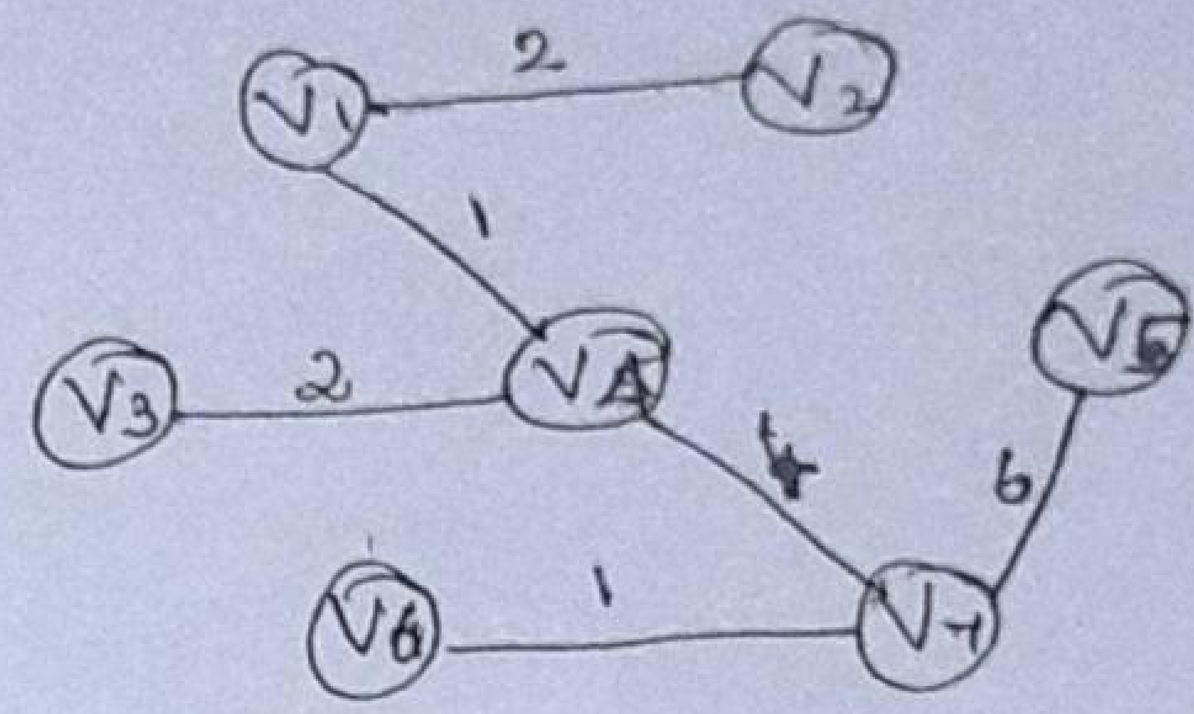
Minimum Spanning Tree.

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost. A minimum spanning tree exists ~~only~~ if and only if G is connected.

(eg)



A Graph G



Minimum spanning Tree

Number of edges in the MST is |V|-1. A MST is a tree because it is acyclic, it is spanning because it covers every vertex, it is minimum for obvious reason.

If an edge is added to a MST then a cycle is generated. If any edge from a MST is removed this will violate the spanning property.

There are two algorithms present to find the minimum spanning tree.

They are,

1) Prim's Algorithm

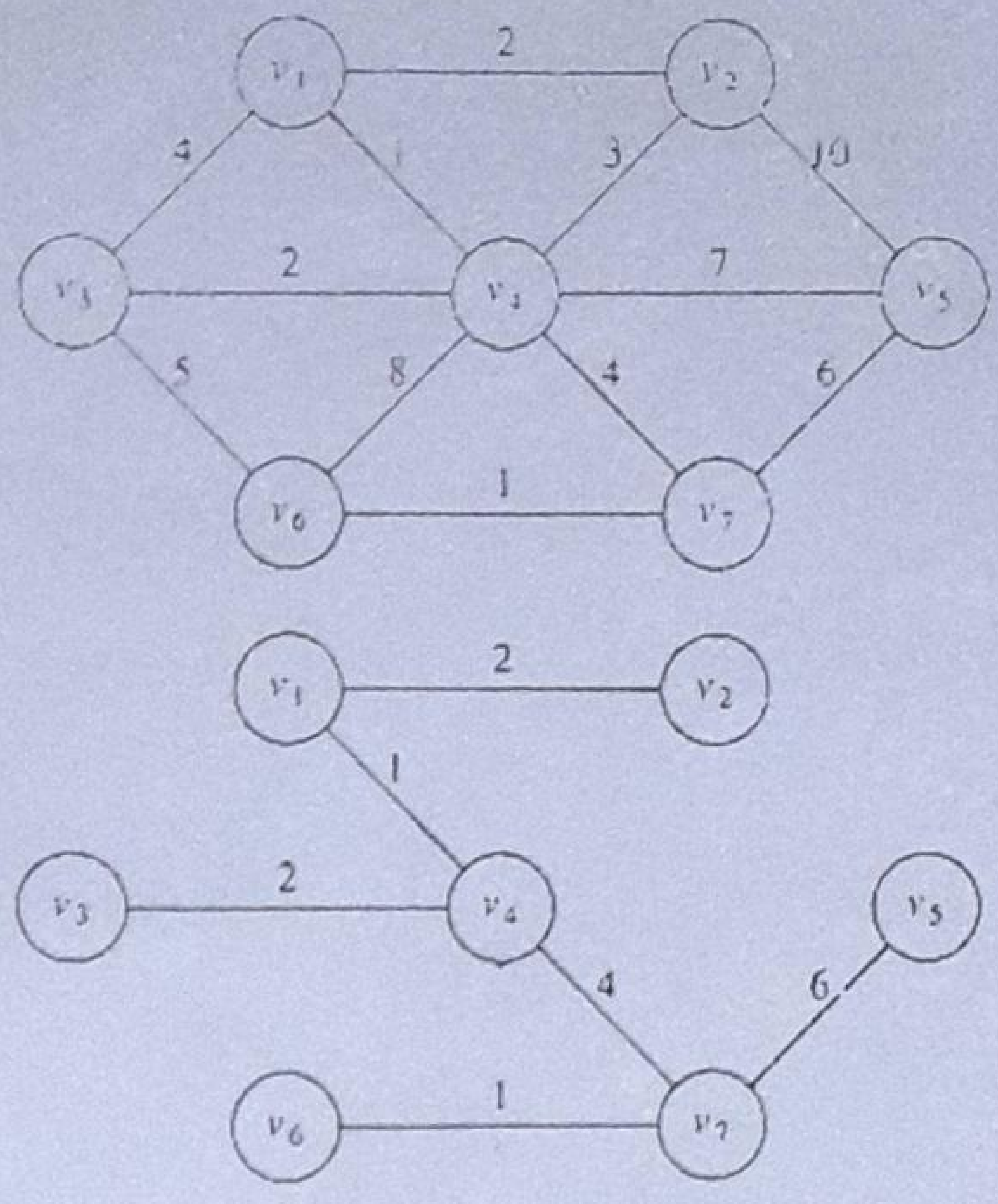
2) Kruskal's Algorithm.

Prims Algorithm.

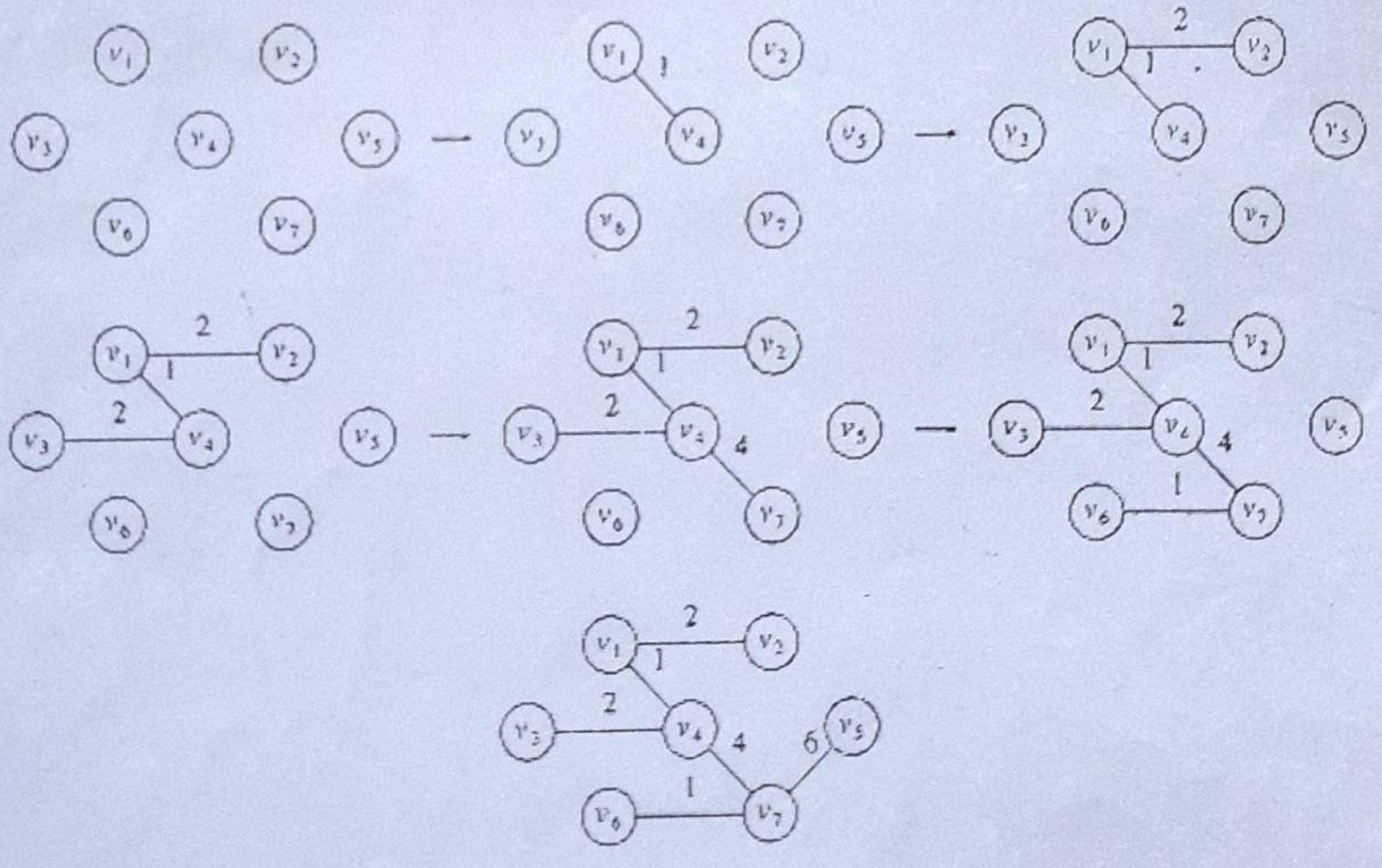
Prims algorithm is one of the way to compute a minimum spanning tree which uses a greedy technique. This algorithm begins with a set U initialised to {1}. It then grows a spanning tree, one edge at a time. At each step, it finds a shortest edge (u,v) such that the cost of (u,v) is the smallest among all edges, where u is in minimum spanning tree and V is not in minimum spanning tree.

Routine for Prims algorithm.

```
Void Prims (Table T)
{
  vertex V, W;
  for (i=0; i< NumVertex; i++)
   {
     T[i].known = False
     T[i].Dist = Infinity
     T[i].Path = 0;
   }
  for ( ; ; )
   { Let V be the start vertex with smallest distance
     T[V].dist=0;
     T[V].known = True;
     for each W adjacent to V
       if (!T[W].known)
       {
         T[W].Dist = Min[T[W].Dist, Cvw);
         T[W].Path = V;
       } } }
```



A graph G and its minimum spanning tree



Prim's algorithm after each stage

v	Known	dv	pv
V1	0	0	0
V2	0	∞	0
V3	0	∞	0
V4	0	∞	0
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

Initial configuration of table used in Prim's algorithm

v	Known	dv	pv
V1	1	0	0
V2	0	2	V1
V3	0	4	V1
V4	0	1	V1
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

The table after v1 **is declared known**

v	Known	dv	pv
V1	1	0	0
V2	0	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4

The table after v4 **is declared known**

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	5	V3
V7	0	4	V4

The table after v2 **and then v3 are declared known**

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	0	1	V7
V7	1	4	V4

The table after v7 **is declared known**

v	Known	dv	pv
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	1	6	V7
V6	1	1	V7
V7	1	4	V4

The table after v6 **and v5 are selected (Prim's algorithm terminates)**

2 Kruskal's algorithm:

A second greedy strategy is continually to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.

The Kruskal's algorithm maintains a collection of ~~two~~ single noded trees. Adding an edge merges two trees into one. When the algorithm terminates there will be only one tree i.e the minimum spanning tree.

The algorithm uses two ~~data structures~~ functions namely find and Union.

Find (u) returns the root of the tree that contains the vertex u.

Union (S, u, v) merge the two trees by making the root pointer of one node point to the root node of the other tree.

The running time of the Kruskal's algorithm is

$O(|E| \log |V|)$.

Edge	weight	Action
(V_1, V_4)	1	Accepted
(V_6, V_7)	1	Accepted
(V_1, V_2)	2	Accepted
(V_3, V_4)	2	Accepted
(V_2, V_4)	3	Rejected
(V_1, V_3)	4	Rejected
(V_4, V_7)	4	Accepted
(V_3, V_6)	5	Rejected
(V_5, V_7)	6	Accepted

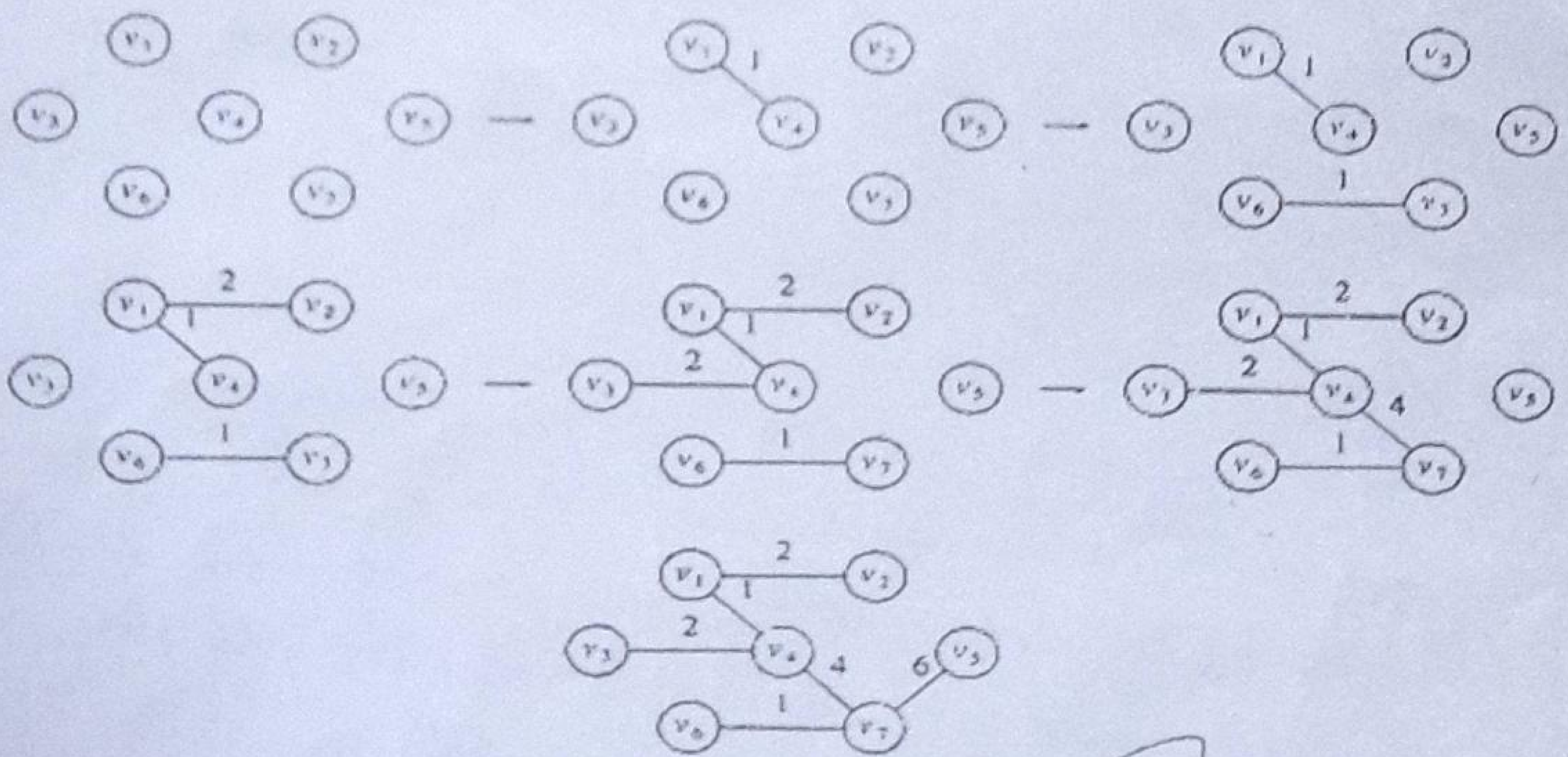
Scanned by TapScanner

```
Void kruskal( graph G )
{
unsigned int edges_accepted;
DISJ_SET S;
PRIORITY_QUEUE H;
vertex u, v;
set_type u_set, v_set;
edge e;
initialize( S );
 read_graph_into_heap_array( G, H );
 build_heap( H );
edges_accepted = 0;
while( edges_accepted < NUM_VERTEX-1 )
{
e = delete_min( H ); /* e = (u, v) */
```



```
u_set = find( u, S );
 v_set = find( v, S );
if( u_set != v_set )
{
edges_accepted++;
set_union( S, u_set, v_set );
}
}
}
```

Pseudocode for Kruskal's algorithm



Kruskal's Algorithm after each stages

Depth First Traversal

Depth first traversal (DFS) is a generalization of preorder traversal. The run time of this algorithm is O(|E| + |V|).

DFS selects one vertex V of G as a start vertex. V is marked visited. Then each unvisited vertex adjacent to V is searched in turn using depth first search recursively. This process continues until a dead end. i.e) a vertex with no adjacent unvisited vertices is encountered. At a dead end the algorithm backsup one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

Implementation steps:

1. Choose any node in the graph, designate it as the search node and mark it as visited.
2. Find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.

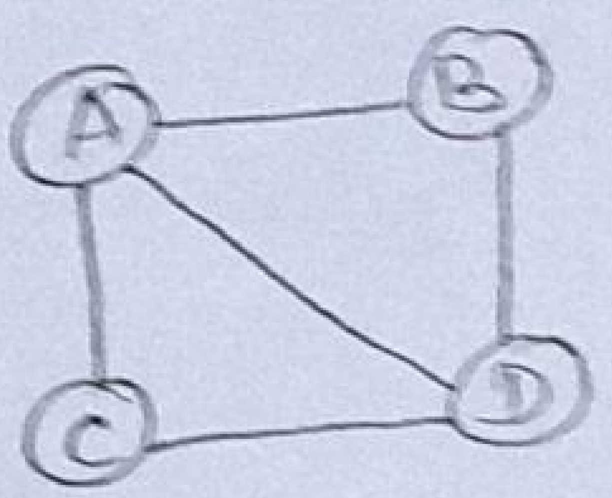
Scanned by TapScanner

3. Repeat step 2 using the current node. If no node satisfying (2) can be found, return to the previous search node and continue from there.
4. When a return to the previous search node in (3) is impossible, the search from the originally chosen search node is complete.
5. If the graph still contains unvisited node, choose any node that has not been visited and repeat step (1) through (4).

Routine for DFS:

```
void DFS (Vertex V)
{
  visited [V] = True;
  for each W adjacent to V
    if ( ! visited [W])
       DFS ( W );
}
```

(eg)



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

Implementation.

1. Let A be the source vertex. Mark it to be visited
2. Find the immediate adjacent unvisited vertex 'B' of A mark it to be visited.
3. From B the next adjacent vertex is 'd'. Mark it as visited.
4. From D the next unvisited vertex is 'c' Mark it to be visited

The resultant tree is called as depth first spanning forest.

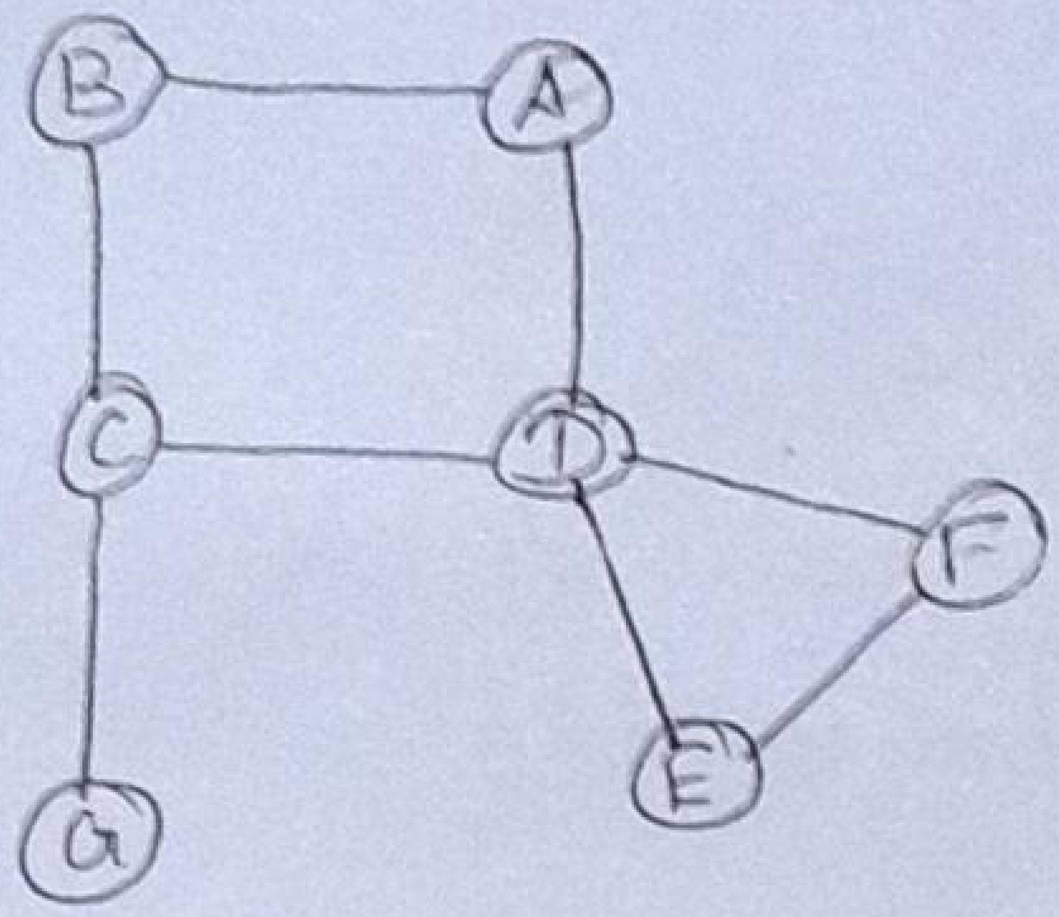
Biconnectivity

A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.

Articulation points

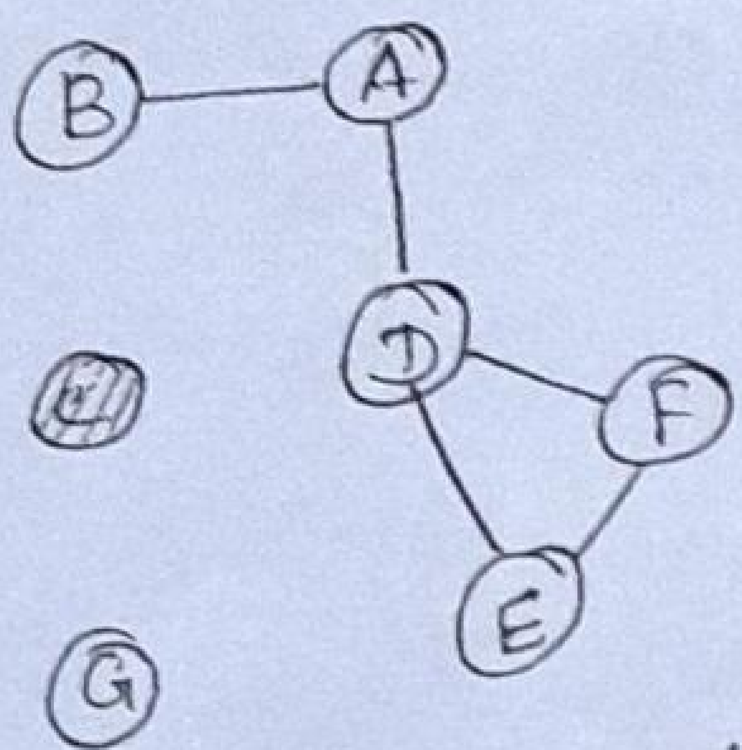
The vertices whose removal would disconnect the graph are known as articulation points

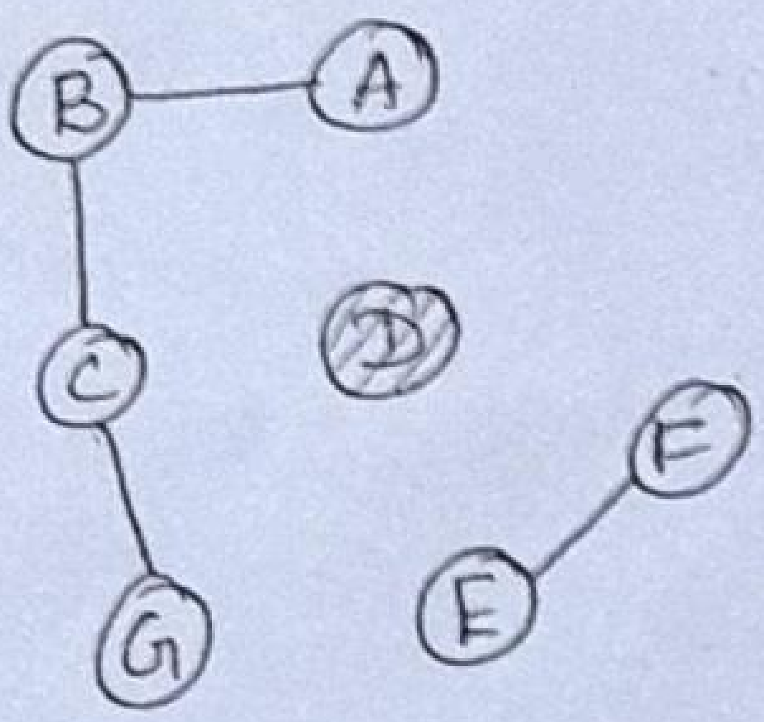
eg)



connected undirected graph.

Removal of 'C' will disconnect G from the graph.
Removal of 'D' will disconnect E & F from the graph.
∴ C & D are articulation points.





The graph is not biconnected, if it has articulation points.
DFS algorithm is used to find the articulation points in a connected graph.

Steps to find articulation points:

1. Perform depth first search, starting at any vertex.
2. Number the vertex as they are visited, as Num (V)
3. Compute the lowest numbered vertex for every vertex V in the depth first spanning tree, which we call as low (W), that is reachable from V by taking zero or more tree edges and then possibly one back edge. By definition, Low (V) is the minimum of
   i) Num (V),
   ii) Lowest Num (W) among all back edges (V,W)
   iii) The lowest Low (W) among all tree edges (V,W)
4. (i) The root is an articulation if and only if it has more than ~~two~~ one children.

   (ii) Any vertex V other than root is an articulation point if and only if V has some child W such that Low (W) $\geq$ Num (V), The time taken to compute this algorithm in a graph is $O(|E| + |V|)$

Note: For any edge (V,W) we can tell whether it is a tree edge ~~or~~ back edge merely by checking Num (V) and Num (W).

If Num (W) > Num (V) then the edge is a back edge.

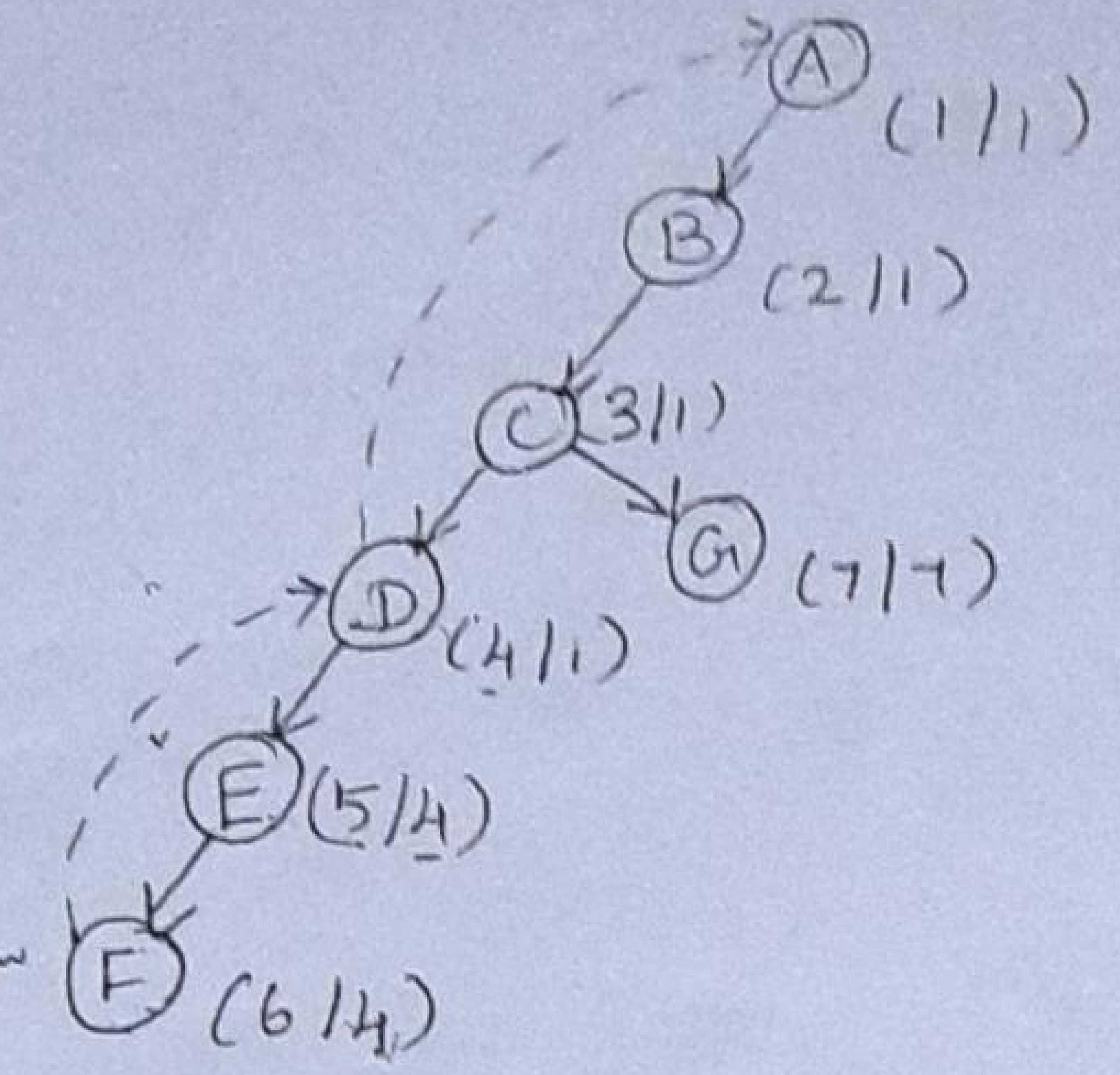
Routine to assign Num to vertices

```
Void AssignNum (Vertex V)
{
  Vertex W;
  Num [v] = counter ++;
  Visited [V] = True;
  for each W adjacent to V
    if (! Visited [W])
    {
      Parent [W] = V;
      AssignNum (W);
    }
}
```

Routine to compute Low & to test for articulation points

```
Void AssignLow (Vertex V)
{
  Vertex W;
  Low [v] = Num [V];        Rule 1
  for each W adjacent to V
  {
    if (Num [W] > Num [v])
    {
      AssignLow (W);
      if (Low [W] >= Num [V])
        printf ("%V is an articulation point \n", v);
        Low [v] = Min (Low [v], Low [W]);        Rule 3
    }
    else if (parent [v] != W)
        Low [v] = Min (Low [v], Num [W]);        Rule 2.
  }
}
```

Depth first tree of the tree is,



Low (F) = Min (Num (F) , Num (D))

= Min (6, 4)

= 4

Low (E) = Min (Num (E) , Low (F))

= Min (5, 4)

= 4

Low (D) = Min (Num (D) , Low (E), Num (A))

= Min (4, 4, 1)

= 1

Low (G) = Min (Num (G))

= 7

Low (C) = Min (Num (C), Low (D), Low (G))

= Min (3, 1, 7)

= 1

Low (B) = Min (Num (B), Low (C))

= Min (2, 1) = 1

Low (A) = Min (Num (A), Low (B))

= Min (1, 1)

= 1

Low (G) ≥ Num (C)

[7 > 3] True

If Low (W) ≥ Num (V) , V is an articulation point

∴ C is an articulation point.

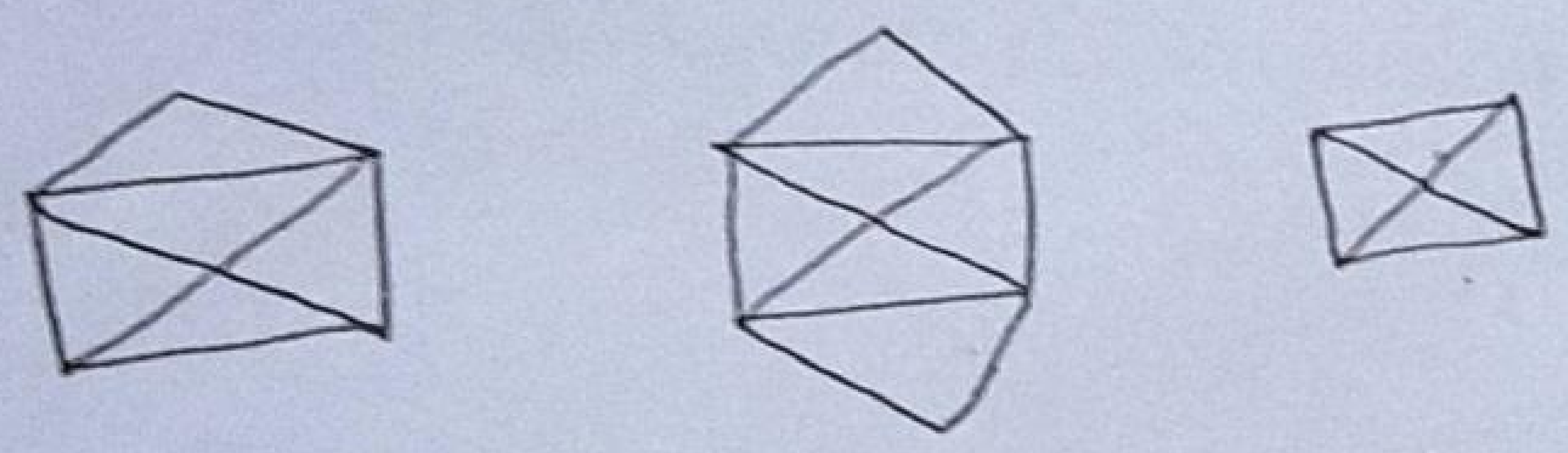
Similarly Low (E) = Num (D).

Hence D is an articulation point.

Euler's Circuit

The Euler's circuit problem is to visit all vertices exactly once, also each edge should be traversed exactly once.

Consider the example.

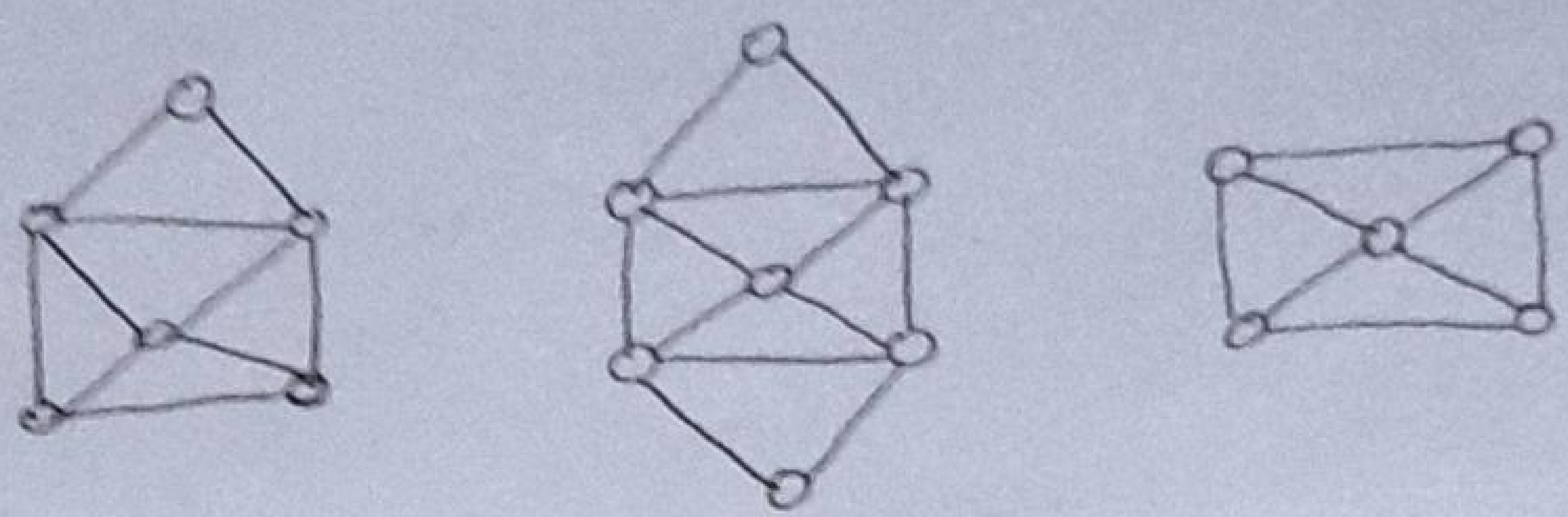


(a) (b) (c)

(a) and (b) can be solved, but (c) cannot be solved.

The rule for this problem is, construct the figures using pen drawing each line exactly once. The pen may not be lifted from the paper.

The problem can be converted to a graph theory problem by assigning a vertex to each intersection.



This problem was solved by Euler. This problem is thus referred to as an Eulerpath or Euler circuit problem.

_ To end at the starting point, the graph should be connected, and each vertex has an even degree.

- If more than two vertices have odd degree, then an Euler's tour is not possible.

Euler Path

A graph is said to be containing an Euler path if it can be traced in 1 sweep without lifting the pencil from the paper and without tracing the same edge more than once. Vertices may be passed through more than once. The starting and ending points need not be the same.

Euler's circuit:

An Euler circuit is similar to an Eulers path, except that the starting and ending points must be the same.

UNIT-V

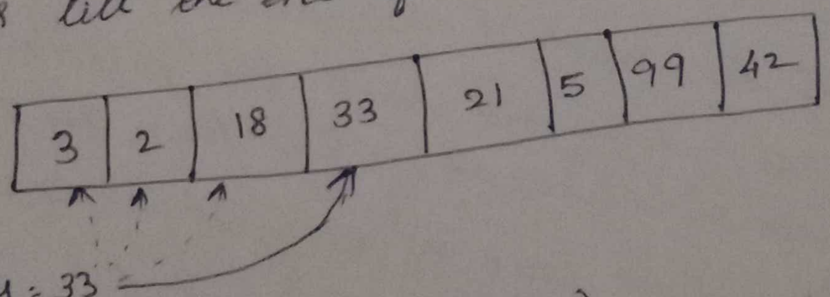
SEARCHING, SORTING, & HASHING TECHNIQUES

Searching:

Searching is the process of determining whether an element is present in a given list of elements or not. If the element is found then the search is succesful otherwise it is an unsuccessful search.

Linear Search:

The search begins at one end of the list and searches for the Key value one by one until the element is found or till the end of the list.




```
int linear (int a[], int n, int key)
{
  int i;
  for (i=0; i<n; i++)
  if (a[i] == Key)
  return a[i];
  }
```

Advantages

- Simple
- It does not require the list to be in sorted
- O(n)

Disadvantage

- Inefficient for large sized list.

Binary Search

- Needs the list to be sorted in ascending order.
- Algm begins by comparing the element in the middle of the list.
- If there is a match then the search ends & the location of the middle element is returned.
- If there is a mismatch with the middle value and the search element is less than the middle element then first part of the list is searched otherwise second part of the list is searched.
- This process is repeated until the search element is equal to the middle element or the list contains only one element that is not equal to the search key.

eg) Search Key : 4.

1, 3, 4, 6, 8, 9, 11

[1, 3, 4] [6] [8 9 11]

$6 \neq 4$ & $6 > 4$ so search in the first half

Next | 1 3 4 |

[1] [3] [4]

3 ≠ 4 So 4>3 So search in Second half

where 4 = 4 So match.

Search is successful.

```
int binarysearch (int a[], int Key, int beg, int end)
{
int mid;
if(beg > end)
return 1;
mid = (beg + end) /2;
if (Key == a[mid])
  return mid;
else if (Key < a[mid])
  return binarysearch (a, Key, beg, mid -1);

else
  return binarysearch (a, Key, mid +1, end);

}
```

Efficiency

Best case = O(1)

Worst case = O(logn).

Advantage
- Requires lesser no. of ilo interactions.
- Faster than linear search.

Disadvantage
- List to be sorted

Sorting Techniques:
Sorting is a process of rearranging the given elements into either ascending or descending order.
There are 2 types of techniques.
① Internal
② External.

① Internal sorting
- Sorting is done in main memory of the computer
eg- Bubble sort, insertion sort, selection sort, quick sort, radix sort, heap sort.

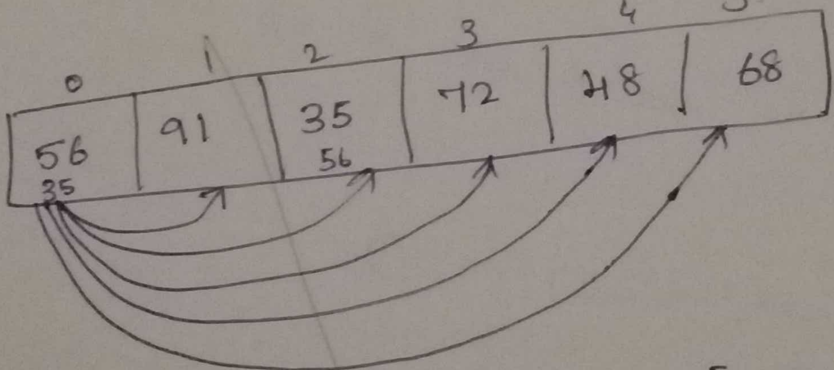
② External sorting:
- Sorting is done in secondary memory.
eg- Multiway merge, polyphase merge.

Bubble sort :

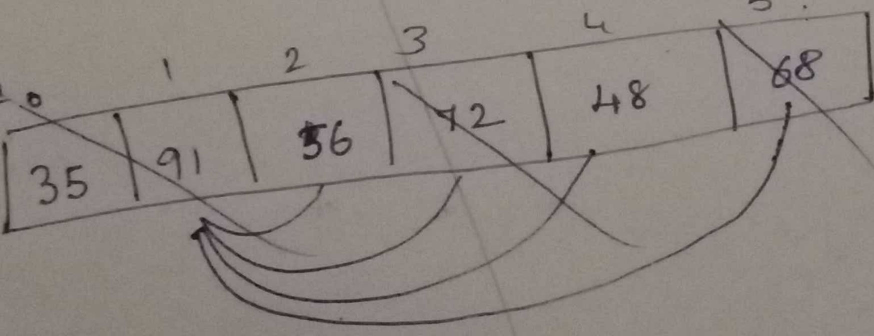
- oldest method
- Also called as sinking sort
- It repeatedly move the smallest element to the lowest index position in the list
- It repeatedly compares two consecutive elements and moves the smallest among them to the left. This process is repeated until all the elements are in the same correct order.

eg. 56, 91, 35, 72, 48, 68.

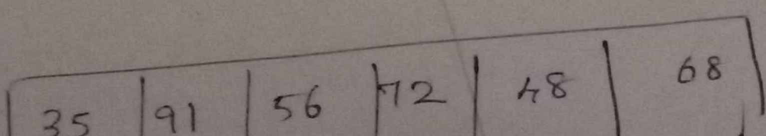
Pass 1.



Pass 2



pass 2



pass 3.

P1 5 1 6 3 4

P2 1 5 6 3 4

P3 1 5 3 6 4

P4 1 5 3 4 6

P5 1 3 5 4 6

P6 1 3 4 5 6

Above is sorted list.

56 91 35 72 48 68

56 35 91 72 48 68

56 35 72 91 48 68

56 35 72 48 91 68

56 35 72 48 68 91

56 35 72 48 68 91

35 56 72 48 68 91

35 56 48 72 68 91

35 56 48 68 72 91

35 48 56 68 72 91

∴ The sorted list is 35 48 56 68 72 91

```
void bubblesort(int a[], int n)
{
 int i, j, tmp;
 for (i=0; i<n; i++)
 {
 for (j=i+1; j<n; j++)
  {
  if (a[i]>a[j])
   {
   tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
   }
  }
 }
```

Selection sort

- Select the smallest element in the list, when found, it is swapped with the first element.
- Then find the second smallest element and swap it with the second element in the list

```
Void SelectSort (int a[ ], int n)
{
 int i, j, k, min;
 for (i=0; i<n; i++)
 {
   k = i;
   min = a[i];
   for (j=i+1; j<n; j++)
   {
    if (a[j]<min)
      min = a[j];
   }
    a[j] = a[i];
    a[i] = min;
  }
}
```

P1	56	91	35	72	48	68
P2	35	91	56	72	48	68
P3	35	48	56	72	91	68
P4	35	48	56	72	91	68
P5	35	48	56	68	91	72
P6	35	48	56	68	72	91

Above is the sorted list

Insertion sort

- Consists of n-1 passes
- we move the element in position P to the left until its correct place is found.

```
Void insertion sort (int a[], int n)
{
  int j, P, temp;
  for (P=1; p<n; p++)
  {
   temp = a[P];
   for (j = P; j>0 && a[j-1] > temp; j--)
   a[j] = a[j-1]
   a[j] = temp;
  }
}
```

The element in position P is saved in temp, and all larger elements are moved one spot to the right. Then temp is placed in the correct spot.

	34	8	64	51	32	21
	8	34	64	51	32	21
P1	8	34	64	51	32	21
P2	8	34	51	64	32	21
P3	8	32	34	51	64	21
P4	8	21	32	34	51	64
P5						

Shell sort:

- Invented by Donald Shell
- It works by comparing elements that are distant
- The distance b/n the comparisons decreases as the algm runs until the last phase. In last phase adjacent elements are compared. So Shell sort is also called as diminishing increment sort.
- The shell sort uses a sequence $h_1, h_2, \ldots h_t$ called the increment sequence.
- After the phase, some increment h_k for every i, $A[i] \le A[i+h_k]$. i.e. all the elements spaced h_k apart are sorted. So the file is said to be h_k sorted.
- The increment sequence suggested by shell is

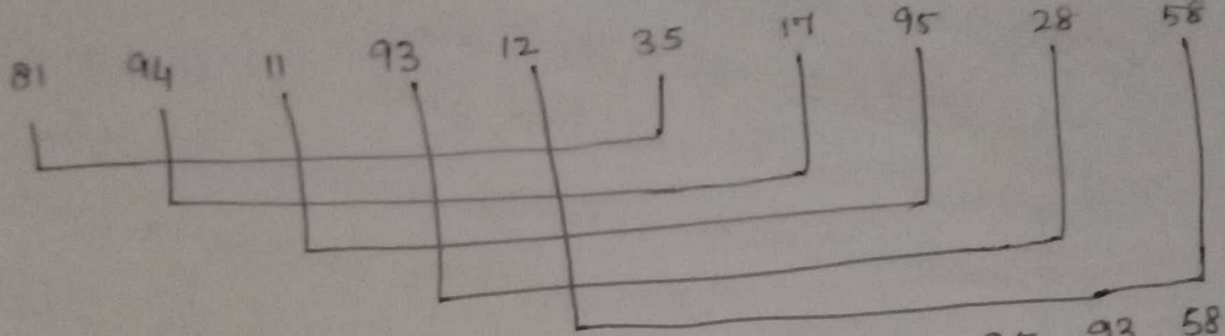
$h_k = N/2$.

```
Void Shellsort (int a[], int n)
{
  int i, j, increment, tmp;
  for (increment = n/2; increment >0; increment /= 2)
  for (i= increment; i<n, i++)
  {
    tmp = a[i];
    for (j = i; j >= increment; j-= increment)
      if (tmp < a[j- increment])
        a[j] = a[j- increment];
      else
        break;
    a[j] = tmp;
  }
}
```

eg. 81 94 11 93 12 35 17 95 28 58

N = 10 K = 10/2 = 5

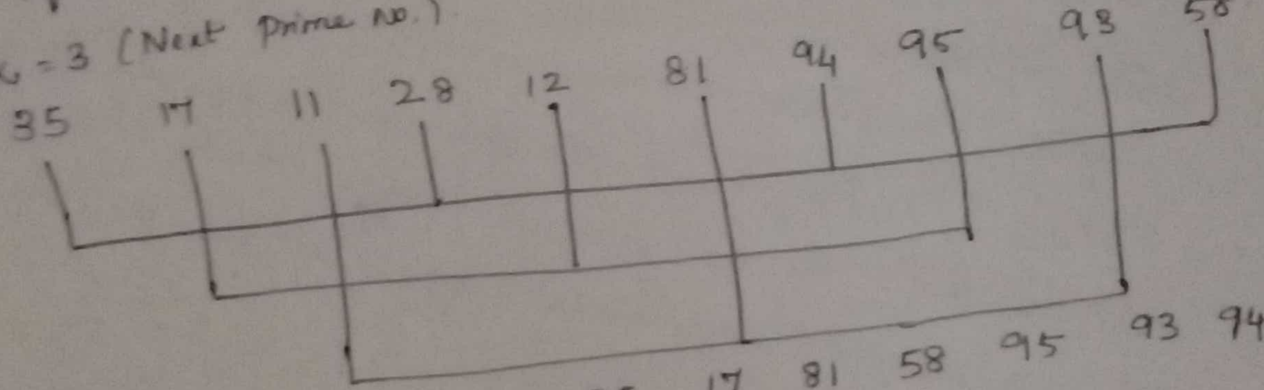
P1.



After P1, 35, 17, 11, 28, 12, 81, 94, 95, 93, 58

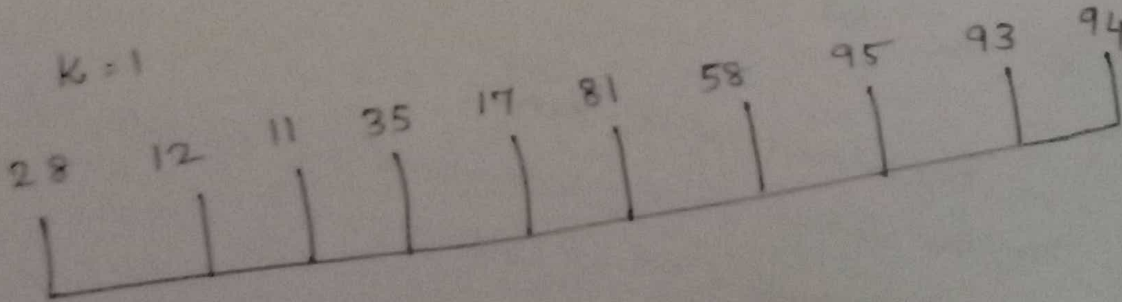
K = 3 (Next prime no.)

P2.



After P2 28 12 11 35 17 81 58 95 93 94

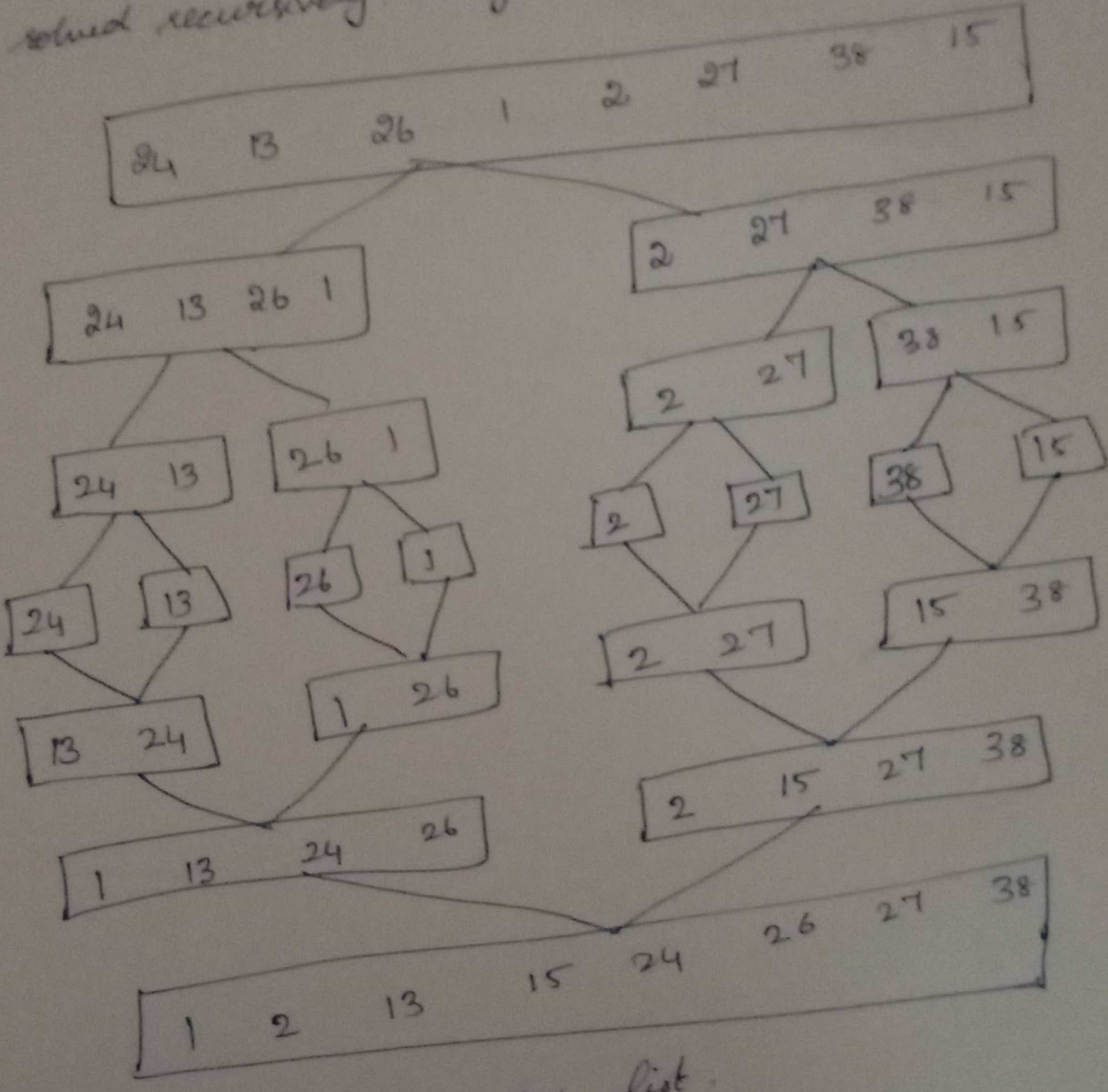
P3 K = 1



Sorted array contains
11, 12, 17, 28, 35, 58, 81, 93, 94, 95

Merge Sort

- Uses divide & Conquer method
- The prm is divided into smaller problem & solved recursively Finally it merges two sorted list



Above is the sorted list.

```
void mergesort (int a[], int n)
{
int *tmp_array;
tmp_array = malloc(n * sizeof (ElementType));
if (tmp-array != NULL)
{
m_sort (a, tmp_array, 0, n-1);
free (tmp_array);
}
else
printf(" No space");

}
```

HASHING

The hash table datastructure is an array of fixed size containing the keys. A key is a string with an associated value.

eg) Salary information

The size of the hash table is refered to TableSize. Every key is mapped into some number in the range 0 to [TableSize -1] and placed in the appropriate cell. The mapping is called a hash function. A hash function should be simple to compute and should ensure that any two distinct keys get different cells.

Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is impossible, and so there is a need for a hash function that distributes the keys evenly among the cells.

eg) A hash Table

0	
1	
2	
3	John 25000
4	Phil 31250
5	
6	Dave 27500
7	Mary 28200
8	
9	

When two keys hash to the same value (ie, same cell) it is known as collision.

__Hash Function:__

If the input keys are integers then the hash value can be calculated using the function.

Key mod TableSize

If the TableSize is 10, all the key value ~~...~~ ends in 0, then the hash value will also be 0 according to the above hash function

So this results in collision. To avoid this the table size can be taken as a prime number.

When the key value is a string, the hash function needs to be chosen carefully, by adding the ASCII values of the characters in the string.

A Simple hash function

```
Index Hash (const char *key, int TableSize)
{
  int HashVal = 0;
  while (*key != '\0')
    HashVal += *key++;
  return HashVal % TableSize;
}
```

If the table size is large, the function does not distribute the keys well.

Good Hash function

```
Index Hash (const char *key, int TableSize)
{
  int HashVal = 0;
  while (*key != '\0')
    HashVal = (HashVal << 5) + *key++;
  return HashVal % TableSize;
}
```

```
Index Hash (const char *key, int TableSize)
{ return (key[0] + 27 * key[1] + [illegible] * key[2])
  % TableSize;
```

Collision Resolve:

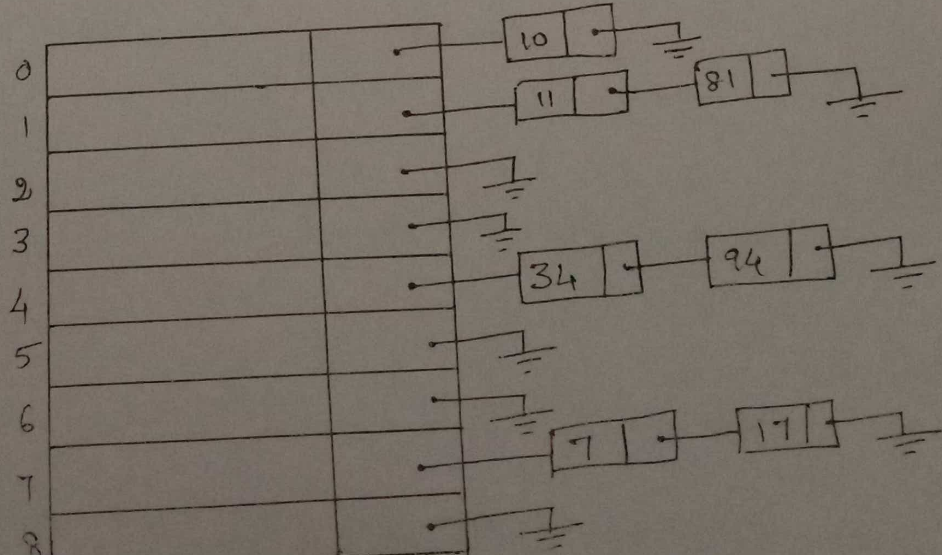
Two methods used to resolve collision are,

1) Separate chaining
2) Open addressing.

Separate chaining:

It is an open hashing technique. A pointer field is added to each record location. When an overflow occurs, this pointer is set to point to overflow blocks making a linked list. In this method, table can never overflow, since the linked lists are only extended upon the arrival of new keys.

eg) Insert 10, 11, 81, 10, 7, 34, 94, 17



Insertion:

To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.

If the elements turns to be a new one, it is inserted either at front of the list or at end of the list.

If it is a duplicate element, an extra field is kept and placed.

The hash function used here is,

Hash (k) = k % tablesize

Insert 10:

Hash (10) = 10 % 10

Hash (10) = 0

So 10 is placed in location '0'.

Insert 11:

Hash (11) = 11 % 10 = 1

Insert 81:

Hash (81) = 81 % 10 = 1

The element 81 collides to same hash value 1. So traverse the list to check whether it is already present. Since it is not present insert at end of the list.

Declaration of separate chaining

```
Struct ListNode
{
 ElementType Element;
 Position Next;
};
Struct HashTbl
{
 int TableSize;
 List * TheLists;
};
```

Initializing routine for hash table (Separate chaining)

```
HashTable InitializeTable (int TableSize)
{
 HashTable H;
 int i;
 H = malloc (SizeOf (Struct HashTbl));
 H -> TableSize = NextPrime (TableSize);
 H -> TheLists = malloc (Sizeof (List) * H -> TableSize);
 for (i = 0; i < H -> TableSize, i++)
 {
  H -> TheLists [i] = malloc (Sizeof (Struct ListNode));
  H -> TheLists [i] -> Next = NULL;
 }
 return H;
}
```

Routine for Insertion:

```
Void insert (int Key, hashtable H)
{
 Position pos, newcell;
 List L;
 pos = Find (Key, H);
 if (Pos = = NULL)
 {
  newcell = malloc (sizeof (Struct Listnode));
  if (newcell ! = NULL)
  {
  L = H -> TheLists [Hash (Key, H -> TableSize)];
  newcell -> next = L -> next;
  newcell -> element = Key;
  L -> next = newcell;
  }
 }
}
```

Find Routine:

```
Position Find (int Key, hashtable H)
{
 position P;
 List L;
 L = H -> TheLists [Hash (Key, H -> tableSize)];
 P = L -> next;
 while (P ! = NULL && P -> element ! = Key)
  P = P -> next;
 return P;
}
```

Advantages:

More number of elements can be inserted as it uses linked list.

Disadvantages:

- It requires pointers which occupies more memory space.
- It takes more effort to perform a search, since it takes time to evaluate hash function and also to traverse list.
- Requires the implementation of a second DS.

OPEN ADDRESSING:

It is also called as closed hashing, which is an alternative to resolve the collision with linked list. In this hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. There are three common collision resolution strategies. They are

i) Linear probing
ii) Quadratic probing.
iii) Double hashing

i) Linear probing:

In this, for the i^{th} probe the position to be tried is $(h(k)+i)$ mod tablesize, where $F(i) = i$, is the linear function

The position in which a key can be stored is found by sequentially searching all position starting from the position calculated by hash function until an empty cell is found. If the end of the table is reached and no empty cells has been found, then the search is continued from the begining of table. It has tendancy to create clusters in the table.

(eg) Insert 42, 39, 69, 21, 71, 55, 33

Empty Table	After 42	After 39	After 69	After 21	After 71	After 55	After 33
0			69	69	69	69	69
1				21	21	21	21
2	42	42	42	42	42	42	42
3					71	71	71
4							33
5						55	55
6							
7							
8							
9		39	39	39	39	39	39

In the above figure first collision occurs whe 69 is inserted which is placed in next available spot namely 0, which is open (free). The next collision occurs when 71 is inserted, which is placed in the next available spot '3', similarly for value 33

Advantages:

- It does not require pointers.

Disadvantages:

- It forms clusters, which degrade the performance of hash table for storing and retrieving data.

Quadratic Probing:

This is a collision resolution method that eliminates the primary clustering problem of linear probing. Here the collision function is quadratic. Here, $F(i) = i^2$.

~~Rear~~ Double hashing:

Here the hashing function is $F(i) = i \cdot hash_2(x)$. This formula, apply a second hash function to x and probe at a distance $hash_2(x)$, $2hash_2(x)$, and so on.

University Questions

2 marks

1. In an AVL tree, at what condition the balancing is to be done?
2. what is the bucket size, when the overlapping and collision occur at same time.
3. What do you mean by heap?
4. Define hashing.
5. What is ment by open addressing.
6. Define load factor of a hash table
7. What is a forest

1. What is an AVL tree? Explain the rotations of an AVL tree (1(
2. Explain binary heap in detail (8)
3. What is hashing? Explain 2 methods to overcome collision problem of hashing (8)
4. Write the functions to insert & delete elements from the AVL tree (16)
5. What is ment by open addressing? Explain the collision resolution strategies in detail.
6. Explain the following in hashing (i) Folding Method (6) (ii) Division method (5) (iii) Linear Probing (5)
7. Explain B tree with its properties. (6)