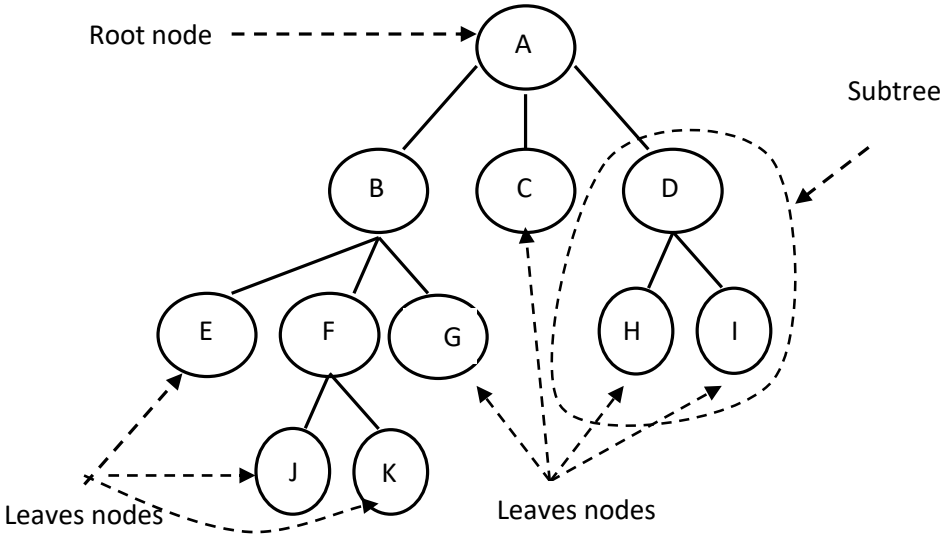
Second Term (Data structure and Algorithm Analysis)

Lecture 1 / 2 May 2020 / Non Linear Data Structures

1. Tree
2. Graph
3. Network

1. Tree

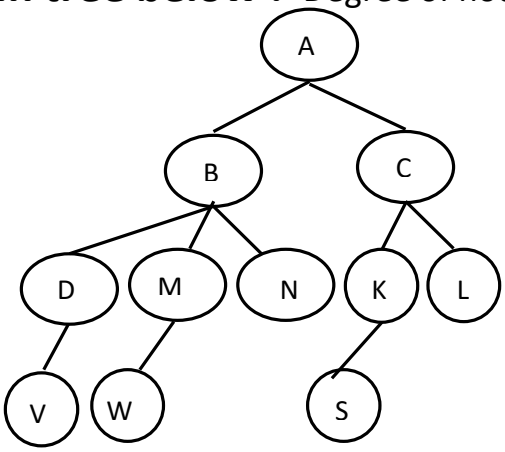
A Tree: is a branching structure such as following figure:



- The point at which lines come together in a tree called "**nodes**".
- The top most node at the tree is called a "**root**" and the bottom most node is called a "**leaf**". The lines connecting the nodes are called "**branches**".
- ;A node is said to be the "**parent**" of these bellow it immediately which are said to be "**children**", children at the same parent called "**brothers**" or "**siblings**" or "**twins**".

- Any node of the tree is itself a tree, which is said to be "**sub tree**" at the origin tree.
- The entire tree is sub tree of itself.
- Each node is a sub tree consisting of only a single node.
- Tree can be represented in a computer memory by linked structure that corresponds directly to the diagram used to represent trees on printed page.
- We say that a node is visited when that node is processed. A visit to every node in a tree is said to be "**traversed**".
- The direction from the node to the leaf is down and the opposite direction is up. Coming from the leaves to the root called "**climbing**" the tree while going from the root to the leaf is called "**descending**" the tree.
- "**Ancestor**", A node's parent is its first ancestor, the parent of the parent is the next ancestor, and so on. The root is an ancestor of each other node.
- The "**level**" of a node refers to its distance from the root.
- Maximum number of levels is called the "**depth**" of tree .
- The "**degree**" of the node is the number of children that branched from it.

- The "**degree**" of the tree is the maximum degree of nodes.
- **In tree below :** Degree of node A= 2 , degree of node B=3, degree of node K=1



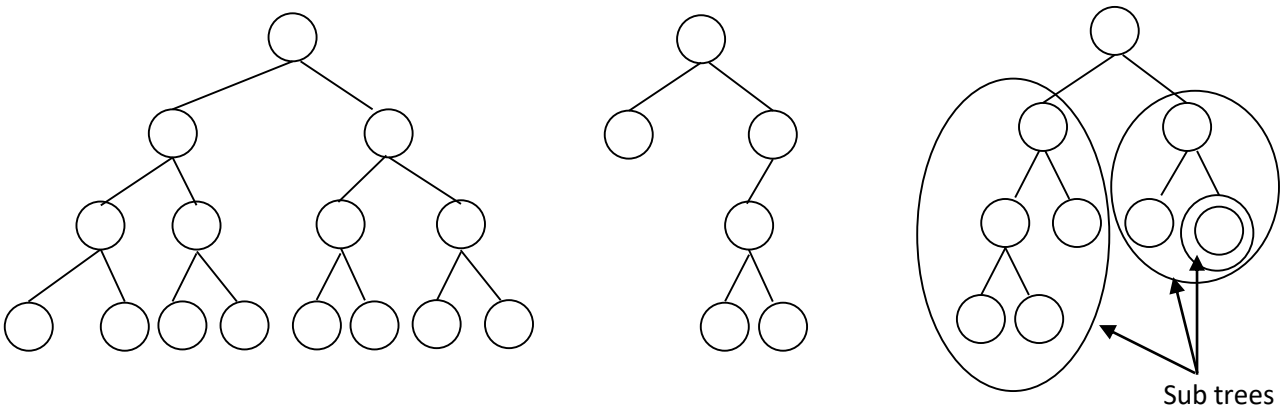
General tree (degree of tree= ?)

Binary Tree

A **binary tree** is a finite set of "**nodes**". The set might be empty (no nodes, which is called **empty tree**). But if the set is not empty, it follows these rules:

There is one special node called the root.

1. Each node may be associated with up to two other different nodes, called its "**left child**" and its "**right child**". If a node *c* is the child of another node *p*, then we say that "*p* is *c*'s parent".
2. Each node, except the root, has exactly one parent; the root has no parent.

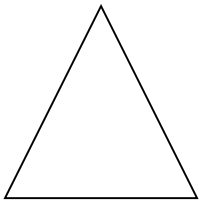


Binary Search Tree

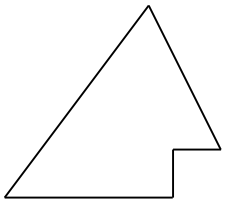
Binary Search Tree: is a binary tree, in which left child (if any) of any node contains a smaller value than does the parent node and the right child (if any) contains a larger value than does the parent node.

Types of Binary Trees

1- Full Binary Tree: is a binary tree in which all of the leaves are on the same level and every non leaf node has two children. The basic shape of a full binary tree is triangular.



2- Complete Binary Tree: is the binary tree that is either full or full through the next to the last level, which leaves on the last level as far left as possible. The shape of complete binary tree is either triangular (if tree is full) or something like the following:

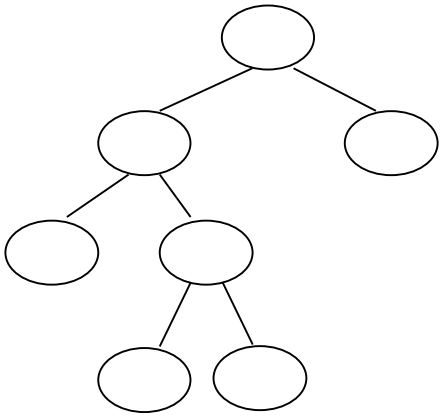


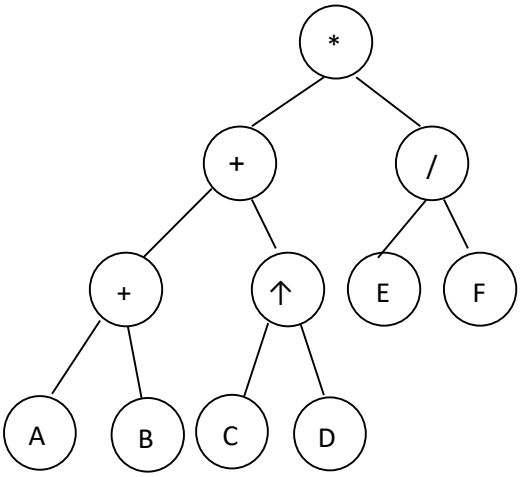
3-Heap Tree: is a data structure that satisfies two properties one concerning it's shape: A heap tree must be a complete tree, and the other concerning the order of its elements: For every node in the heap (Max Heap), the value stored in that node is greater than or equal to the value in each of its children. The root node will always contains the largest value in the heap and thus we always know where the maximum value is in the root.

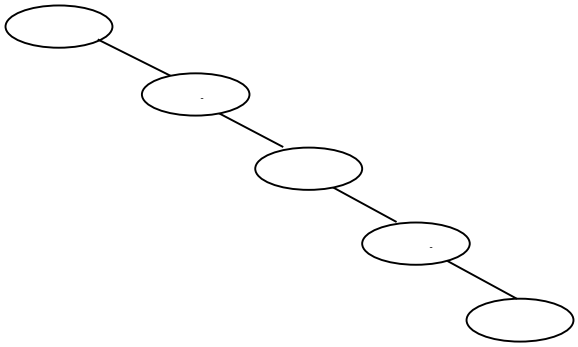
4-Strictly Binary Tree: is a binary tree in which each node except the leaf has two children. A strictly binary tree which has (n) leaves always contains (2*n-1) nodes.

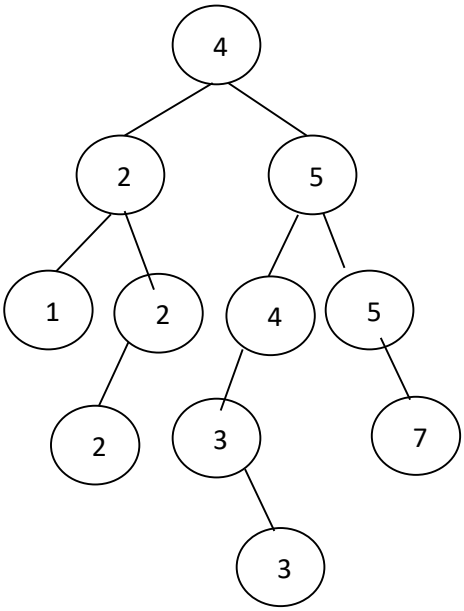
5-Balanced Binary Tree: In a binary tree, each node has a factor called "**balance factor**". Balance factor of a node is the height of the left subtree minus the height

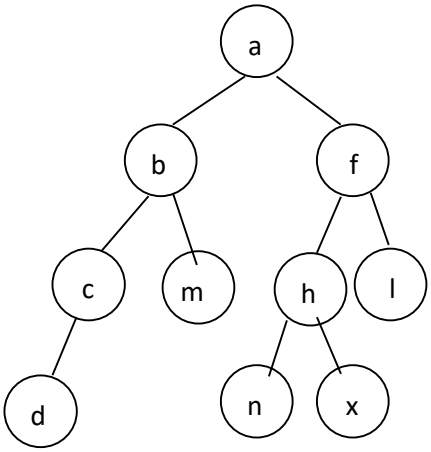
of the right subtree. If each node in the binary tree has a balance factor equal to -1 or 0 or 1 then this binary tree is called "**balanced**".

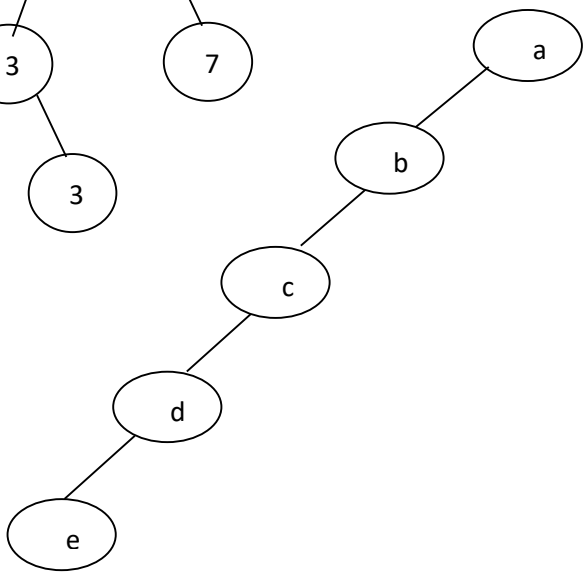


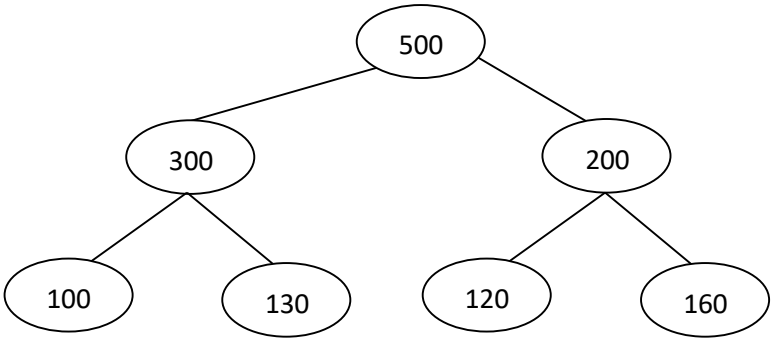


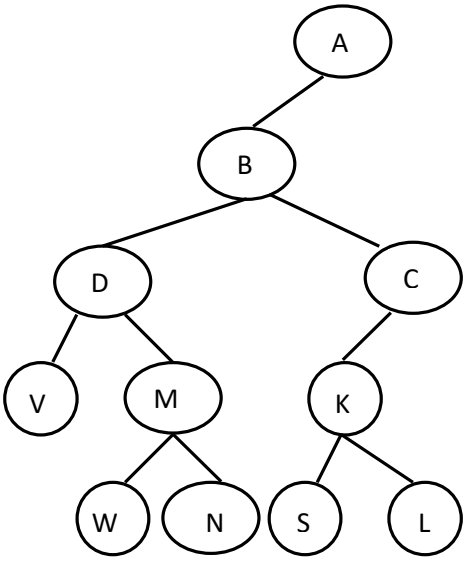


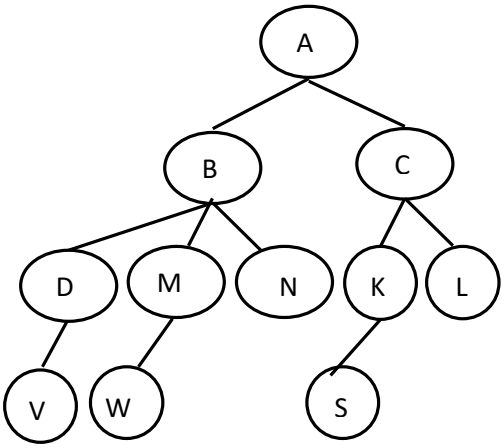


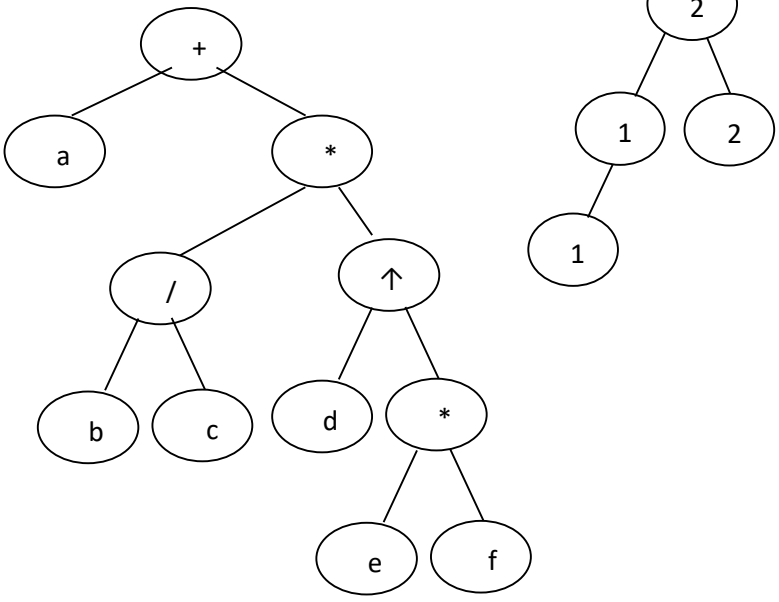




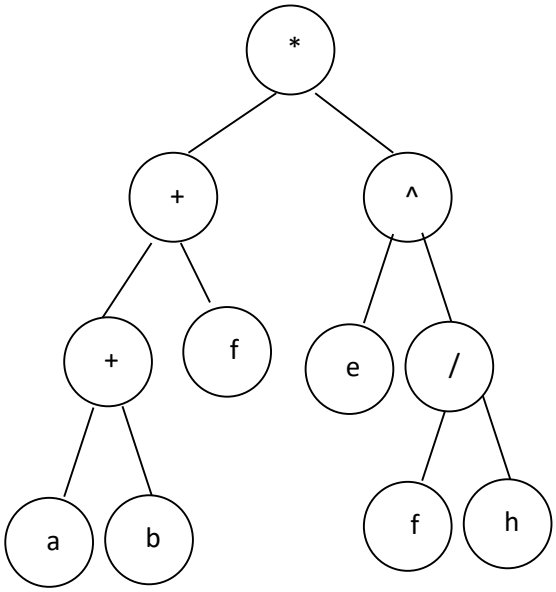

500
300
200
100
130
120
160

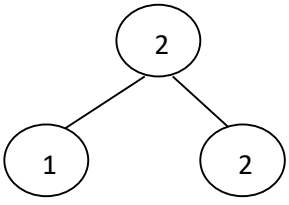

A
B
D
C
V
M
K
W
N
S
L

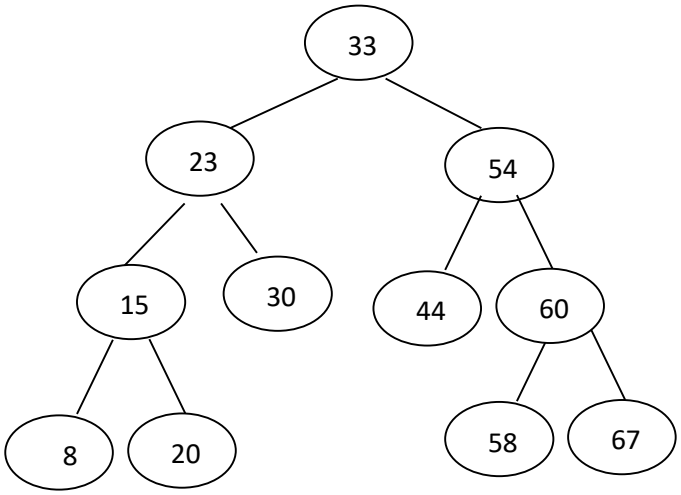

A
B
C
D
M
N
K
L
V
W
S


+
a
*
/
↑
b
c
d
*
e
f

2
1
2
1


*
+
^
+
f
e
/
a
b
f
h


2
1
2


33
23
54
15
30
44
60
8
20
58
67

Term 2 (Data structure and Algorithm Analysis)

Lecture 2 / 9 may 2020

Application of binary tree

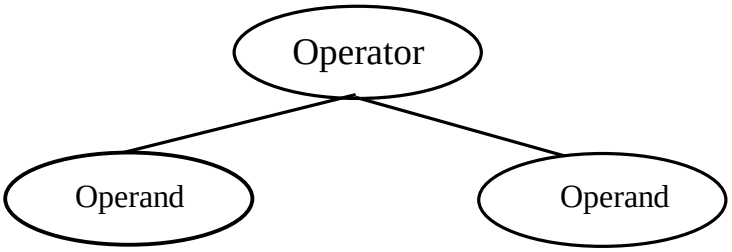
1- To represent any arithmetic expression.

2- To obtain sorted data by building Binary Search Tree (BST).

3- To find all duplicated data.

1- To represent any arithmetic expression:

Arithmetic expression can be represented as tree, such expression trees are particularly interested because the prefix and postfix (suffix) notations for the arithmetic expression correspond to important ways of traversing. When we use a binary tree to represent an expression, parentheses are not needed to indicate precedence. The levels of nodes in the tree indicate the relative precedence of evaluation implicitly. The operations at higher levels of the tree are evaluated later than those below them. The operation at the root will always be the last opearation performed.

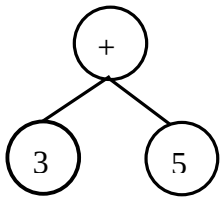


The simplest arithmetic expression consisting of single constant, The corresponding tree consists of a single node.

Y=3

The simplest expression consisting of two constants compound by any operator such as 3+5. We can represent the binary expression as a two level binary tree. The root of tree is operator and its two children (subtrees) are operands.

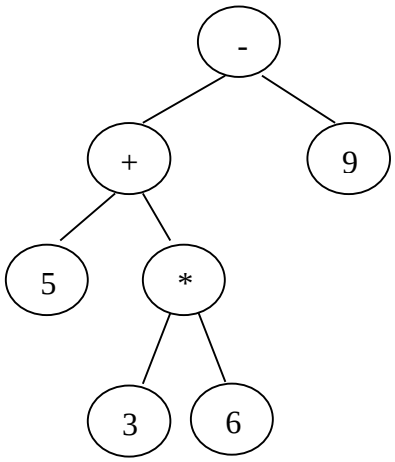
3+5



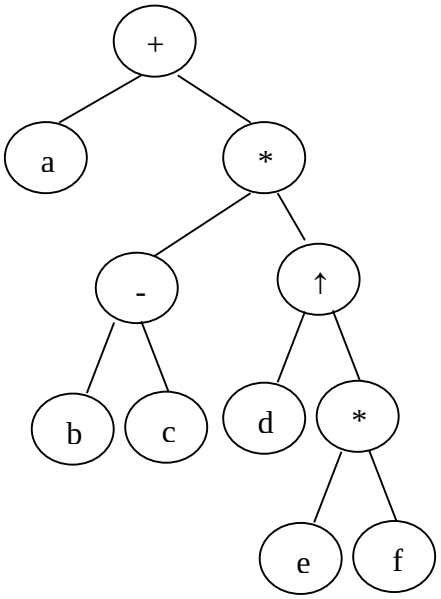
A more complex expression can be represented as any root in a tree or any subtree is operator and leaves contain operands (variables and constants).

The following examples show the representation of expressions as trees:-

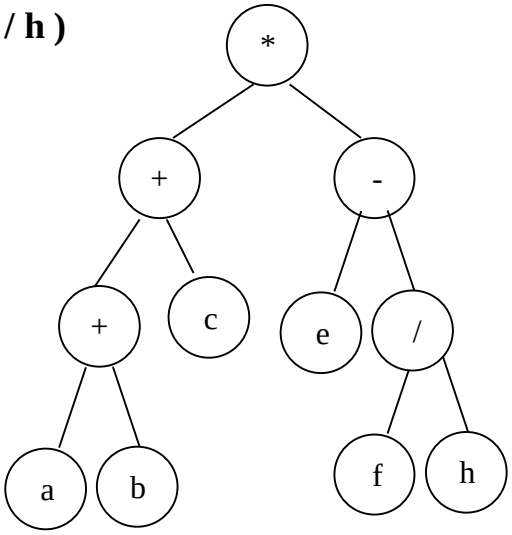
1- **5+3*6-9**



2- **a + (b – c) * d ↑ (e * f)**



3- **(a + b + c) * (e – f / h)**



Tree traversals

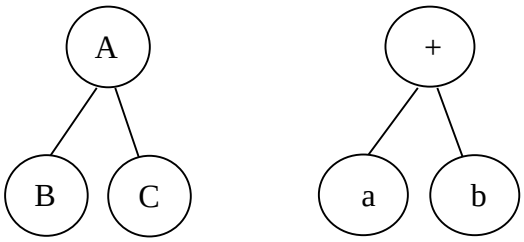
Tree traversals means visit all the nodes in a tree only once. There are 3 common ways for traversals of binary tree: in-order traversal, pre-order traversal, and post-order traversal.

1- Inorder traversal: each node is visited in between its left and right subtrees.

   a. left subtree
   b. root
   c. right subtree

   B A C

   a + b (infix expression)



2- Preorder traversal: each node is visited before its left and right subtrees.

   a. Root
   b. left subtree
   c. right subtree

   A B C

   + a b (prefix expression)

3- Postorder traversal: each node is visited after its left and right subtrees.

   a. left subtree
   b. right subtree
   c. root

   B C A

   a b + (suffix expression)

Note that there are another methods to traverse general tree, a tree if we convert left by right.

1- Converse inorder: each node is visited in between its right and left subtrees.
   a. right subtree
   b. root
   c. left subtree
2- Converse preorder: each node is visited after its right and left subtrees.
   a. Root
   b. right subtree
   c. left subtree
3- Converse postorder: each node is visited before its right and left subtrees.
   a. right subtree
   b. left subtree
   c. root

Note that there are two methods depend on level concept to traverse general tree.

1- Level by Level:

A. Top-Down: Nodes are visited starting from top level (level 0) down to the last level. In each level we start from left to right.

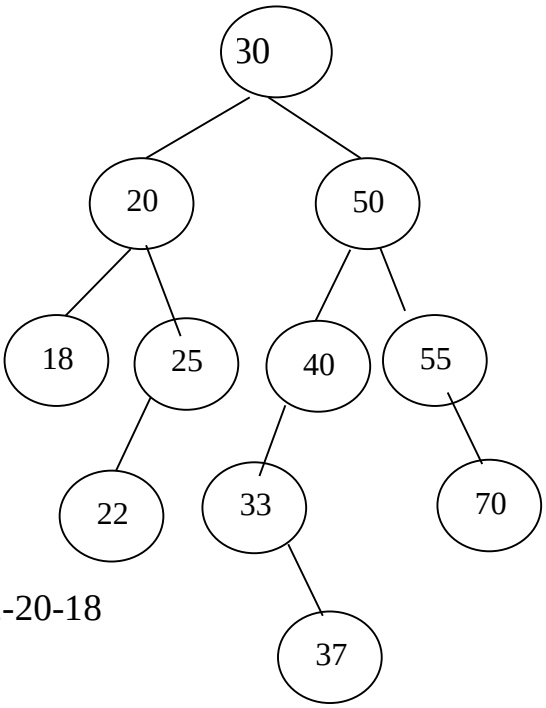
B. Bottom-Up: Nodes are visited from the last level up to the top level (level 0). In each level we start from left to right.

2- Converse Level by Level:

A. Top-Down: Nodes are visited starting from top level (level 0) down to the last level. We start from right to left.

B. Bottom-Up: Nodes are visited from the last level up to the top level (level 0). In each level we start from right to left.

Example 1: Write all the traversals for the following trees:



Inorder: 18-20-22-25-30-33-37-40-50-55-70

Preorder: 30-20-18-25-22-50-40-33-37-55-70

Postorder: 18-22-25-20-37-33-40-70-55-50-30

Converse Inorder: 70-55-50-40-37-33-30-25-22-20-18

Converse Preorder: 30-50-55-70-40-33-37-20-25-22-18

Converse Postorder: 70-55-37-33-40-50-22-25-20-18-30

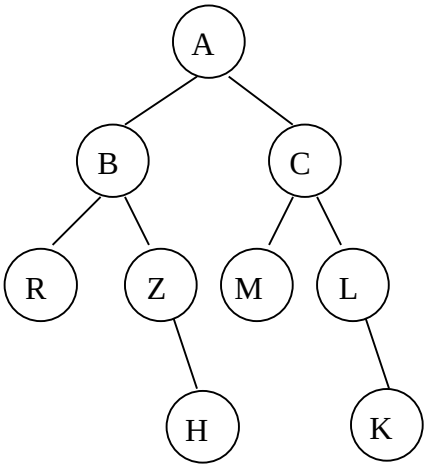
Level by Level (top-down): 30-20-50-18-25-40-55-22-33-70-37

Level by Level (bottom-up): 37-22-33-70-18-25-40-55-20-50-30

Converse Level by Level (top-down): 30-50-20-55-40-25-18-70-33-22-37

Converse Level by Level (bottom-up): 37-70-33-22-55-40-25-18-50-20-30

Example 2: Given the following tree, write the inorder, postorder and preorder traversals?



R B Z H A M C L K (inorder)

A B R Z H C M L K (preorder)

R H Z B M K L C A (postorder)

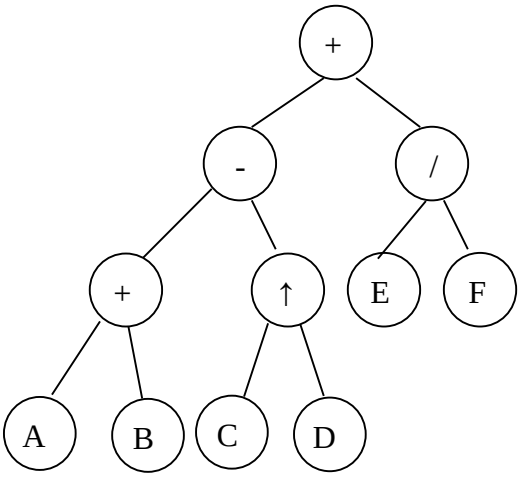
Example 3: **((A + B) - (C ↑ D)) + (E / F)**

Given the above expression, draw the expression tree then use the preorder and preorder traversals to obtain prefix and suffix forms(notations)?

Inorder Traversal: A + B – C ↑ D + E / F

Preorder Traversal: + - + A B ↑ C D / E F

Postorder Traversal: A B + C D ↑ - E F / +



Note that :

The preorder traversal obtain prefix form & postorder traversal obtain suffix form

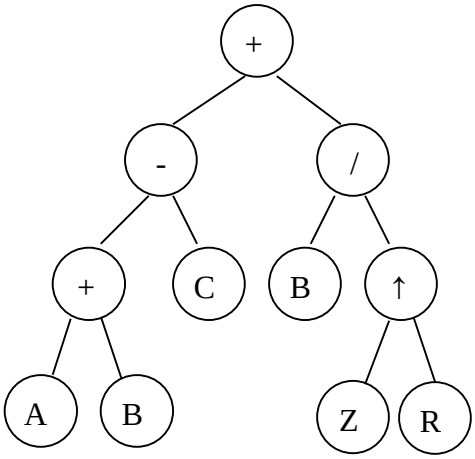
The Inorder traversal give the original expression without parenthesis .

Example 4: Given the following tree, write the inorder, postorder, preorder traversals and the original arithmetic expression?

A + B – C + B / Z ↑ R (inorder)

A B + C – B Z R ↑ / + (postorder)

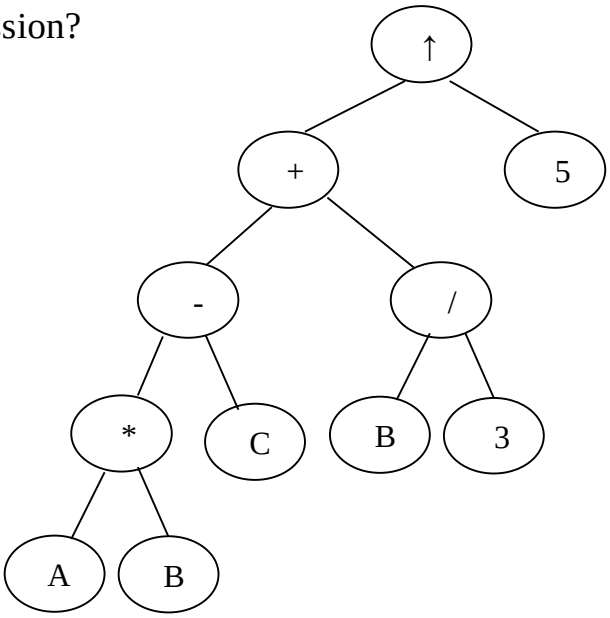
+ - + A B C / B ↑ Z R (preorder)



(((A + B) – C) + (B / (Z ↑ R))) (original arithmetic expression)

Example 5: Given the following tree:

Write the original arithmetic expression?



Solution:

(((A * B) – C) + (B / 3)) ↑ 5

or

(A * B – C + B / 3) ↑ 5

(Remove not needed parenthesis)

Exercise:

Write the suffix and prefix forms to

A given tree.

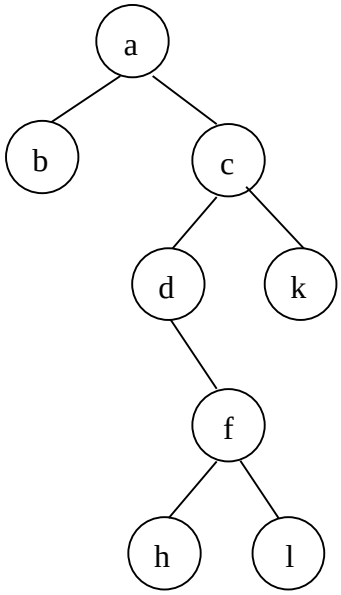
Example 6: Given the following trees, write the inorder, postorder and preorder traversals?

1)

b a d h f l c k (inorder traversal)

b h l f d k c a (postorder traversal)

a b c d f h l k (preorder traversal)



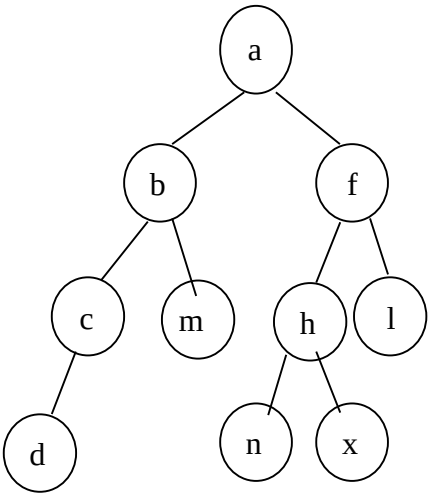
---

2)

d c b m a n h x f l (inorder traversal)

d c m b n x h l f a (postorder traversal)

a b c d m f h n x l (preorder traversal)

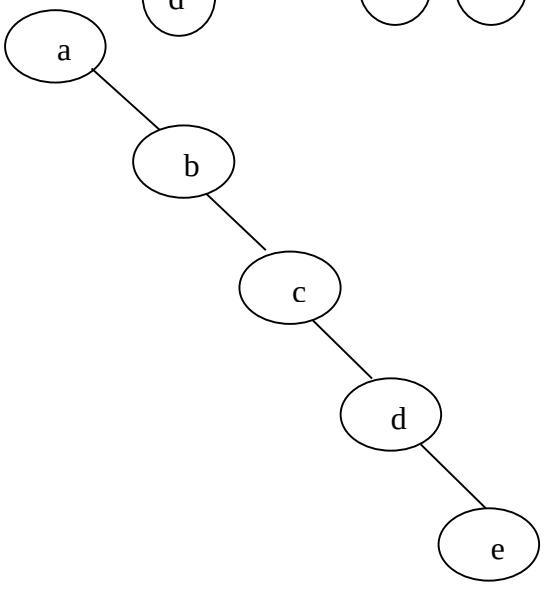


3)

a b c d e (inorder traversal)

e d c b a (postfix traversal)

a b c d e (prefix traversal)

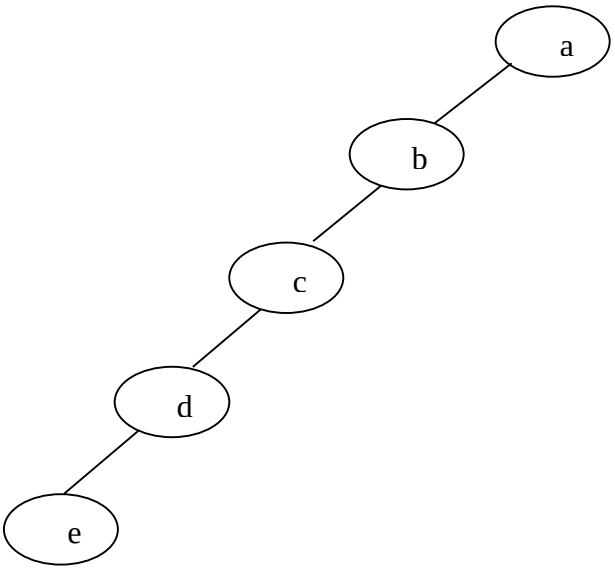


4)

e d c b a (inorder traversal)

e d c b a (postfix traversal)

a b c d e (prefix traversal)



Remark1: We can know the root from postorder traversal then we can know the left subtree and the right subtree from the inorder traversal.

Remark2: If the tree is in one side left or right then there are two equal traversals:

1- If inorder = preorder means that the tree is on the right.
2- If inorder = postorder means that the tree is on the left.
3- If inorder=postorder=preorder means that the tree consisted from one node.

Remark3: If the word left interchanged by right in the traversal, we obtain convert inorder, converse preorder and converse postorder.

Note that: We can drawing a tree from the suffix or prefix forms.

Exercises : Draw a trees corresponding to a given expressions ,

then perform its prefix (or suffix) forms in addition to the original expressions

1- A B + C – B Z R ↑ / +

2- G B H * + T W / M ^ -

3- + - A B * / C D ^ E F

4- * / ^ 8 H / - R B + C D ^ k 2

5- / / * T R + K L ^ – 10 R + C B

Note that : An expression tree must be strictly tree ? Why ?

Home work (1)

Q2) Draw a binary tree to represent the expressions below ,
show how obtain the suffix and prefix forms ,
then allocate the tree in a suitable array.

1- (A+B/K) / L^(N* 5 *G)

2- (A*B+K) / 4 + (N* L*G)

3- H^B /(E+ H/3)*N^5

4- A B / E H C / - * N R ^ +

5- - + + A* B C / D E

6- (A/B+K)^4^(N- L*G)

state the relationship between parents and sons

Home work (1)

Q1) Draw a binary tree to represent the expressions below
then write the original forms:

1- / + * A R ^ Z F - B ^ H L

2- Z R / B A * H / + M 2 ^ -

3- / + R 5 F * B - N * H L

4- A Z ^ B C * H / - L N / +

Q2) Draw a binary tree to represent the expressions below , show how obtain the suffix and prefix forms , then allocate the tree in a suitable array.

1- (A+B/K) / L^(N* 5 *G)

2- (A*B+K) / 4 + (N* L*G)

3- H^B /(E+ H/3)*N^5

4- A B / E H C / - * N R ^ +

5- - + + A* B C / D E

6- (A/B+K)^4^(N- L*G)

state the relationship between parents and sons

Q) How a Binary tree represented in main memory ?

Ans. A binary tree can be represented in memory using two different methods

1- Sequential Representation (Static Allocation)

2- Linked Representation (Dynamic Allocation)

1-The Sequential Representation of Binary Tree

The complete binary Search Tree can be numbered from 1 to n so that the number assigned to the left son is twice the number assigned to it's parent and the number assigned to the right son is one more than twice the number assigned to it's parent. (i.e) the node in position (p) is the parent of the nodes in position (2p) and (2p +1). If the left child at position (p) then it's right brother in position (p+1) and if the right child at position (p) then it's left brother in position (p-1). If any node in position (k) then it's parent in position trunc (k/2). Then we will store the tree in the array level by level from left to right.

(the root in location 1 its left son in location 2 & its right son in location 3 and so on).

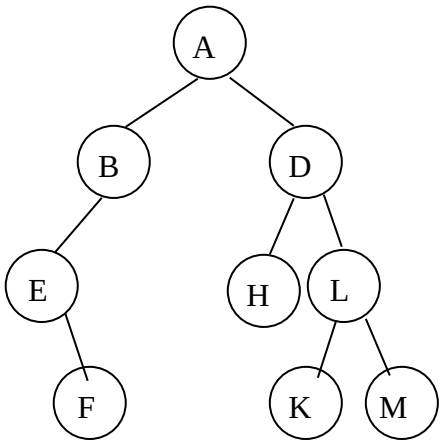
Example 1:

Size of array = $2^N - 1$

Where N no. of levels

4 level , $2^4 - 1$

16-1 =15 location

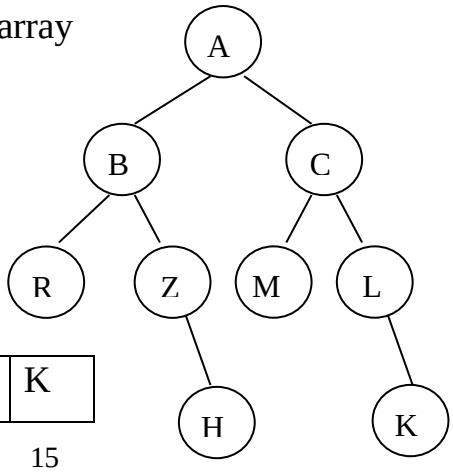


A	B	D	E		H	L		F					K	M
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example 2: Represent a given the following tree in suitable array

The array representation of the tree

Array size = 15 location



A	B	C	R	Z	M	L		…		H		…		K
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

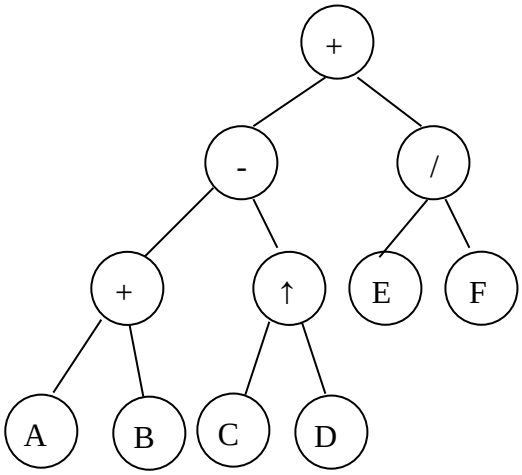
Example 3: ((A+B)-(C↑D))+(E/F)

Given the above expression, draw the expression tree then use the preorder and preorder traversals to obtain prefix and suffix forms(notations)? Then allocate the tree in Suitable array .

Inorder Traversal: A+B-C↑D+E/F

Preorder Traversal: +-+AB↑CD/EF

Postorder Traversal: AB+CD↑-EF/+



+	-	/	+	^	E	F	A	B	C	D		…		…
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

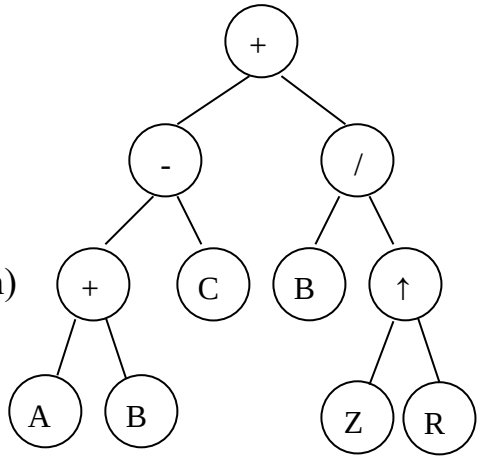
Example 4: Given the following tree, write the inorder, postorder, preorder traversals and the original arithmetic expression?

A+B-C+B/Z↑R (inorder)

AB+C-BZR↑/+ (postorder)

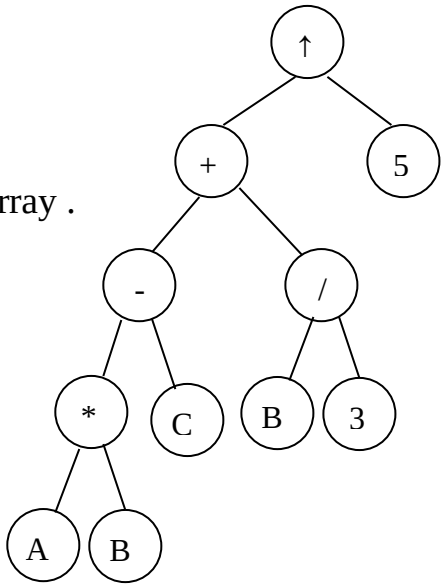
+-+ABC/B↑ZR (preorder)

(((A+B)-C)+(B/(Z↑R))) (original arithmetic expression)



+	-	/	+	C	B	^	A	B					Z	R
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example 5: Given the following tree, write the original Expression and allocate the tree in suitable array .



Solution :

(((A*B)-C)+(B/3))↑5

(A*B-C+B/3)↑5

^	+	5	-	/			*	C	B	3	…	A	B	…
1	2	3	4	5	6	7	8	9	10	11	…	16	17 …	31

Note that : The binary tree can be obtained from its array representation

Example 6: Given the following array ,draw a binary tree then traversed it using the inorder, postorder and preorder traversals?

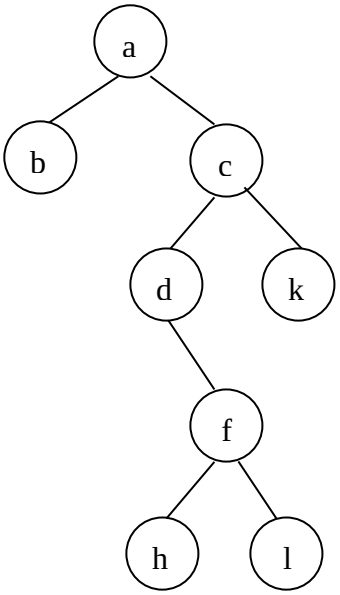
a	b	c	…	…	d	k	…	f	…	h	l	…	
1	2	3	4	5	6	7		13		26	27	…	31

Solution

b a d h f l c k (inorder traversal)

b h l f d k c a (postorder traversal)

a b c d f h l k (preorder traversal)



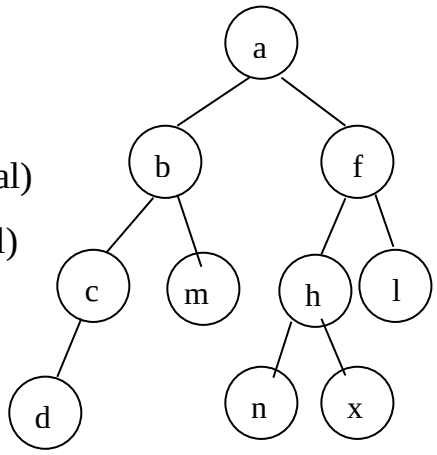
Example 7:

Given the following arrays ,draw a corresponding binary tree then traversed it using the inorder, postorder and preorder traversals?

A)

a	b	f	c	m	h	l	d	…	…	n	x	…	
1	2	3	4	5	6	7	8	9	…	12	13	14	15

Solution :



d c b m a n h x f l (inorder traversal)

d c m b n x h l f a (postorder traversal)

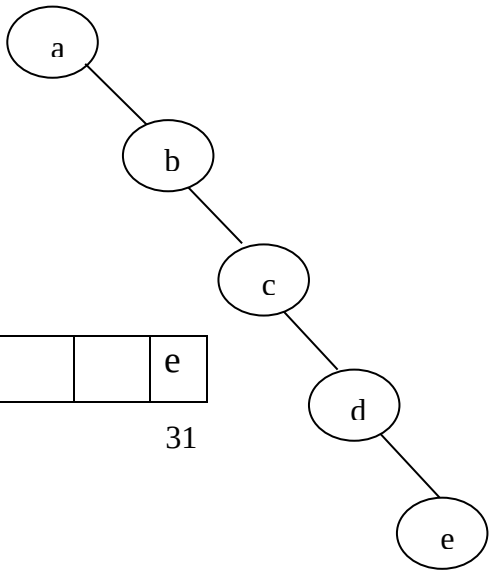
a b c d m f h n x l (preorder traversal)

B)

N=5

The array size = $2^N - 1$

32- 1 = 31 location



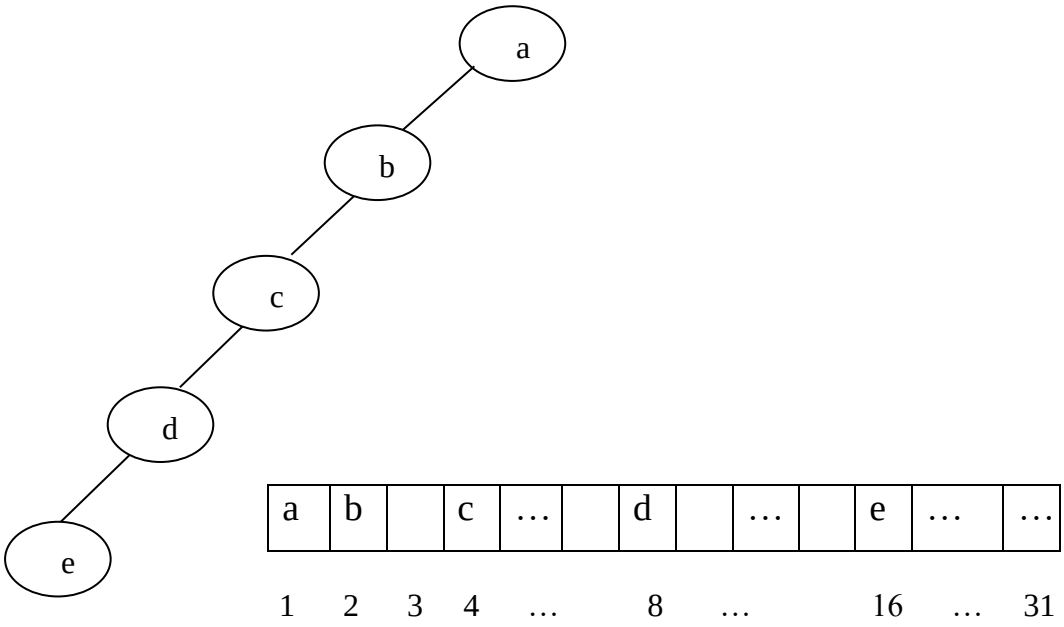
a	…	b		…		c	…			d	…			e
1	2	3		…		7	…			15	…			31

a b c d e (inorder traversal)

e d c b a (postfix traversal)

a b c d e (prefix traversal)

C)

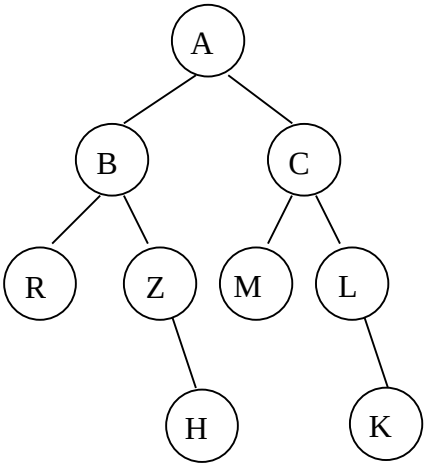


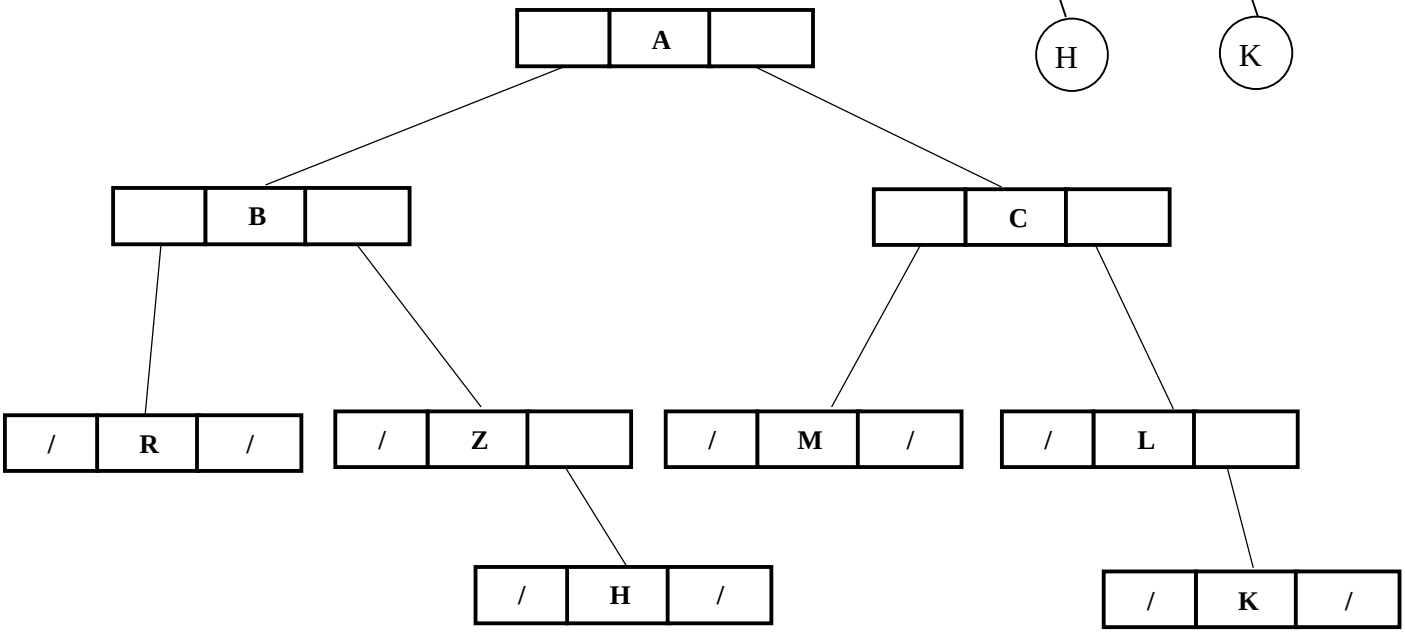
e d c b a (inorder traversal)

e d c b a (postfix traversal)

a b c d e (prefix traversal)

Linked Allocation of a Binary Tree



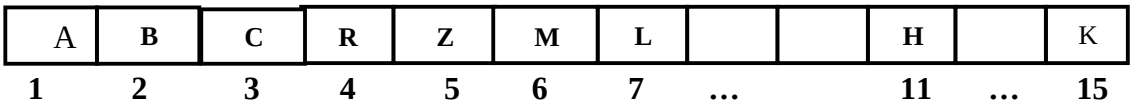


Note that :

Any binary tree contain N nodes have (**N+1)** Null pointers

The tree above contain **(9)** nodes then it have **(10)** Null pointers

(4) Level then the array size **(15)** location.



The Static Allocation of the given tree

Root in location **(1)** its left sub tree in location **(2)** and right sub tree in location **(3)**

The children of node contain B in locations **(4)** and **(5)**

The children of node contain C in location **(6)** and **(7)**

The right child of node contain Z in location **(11)**

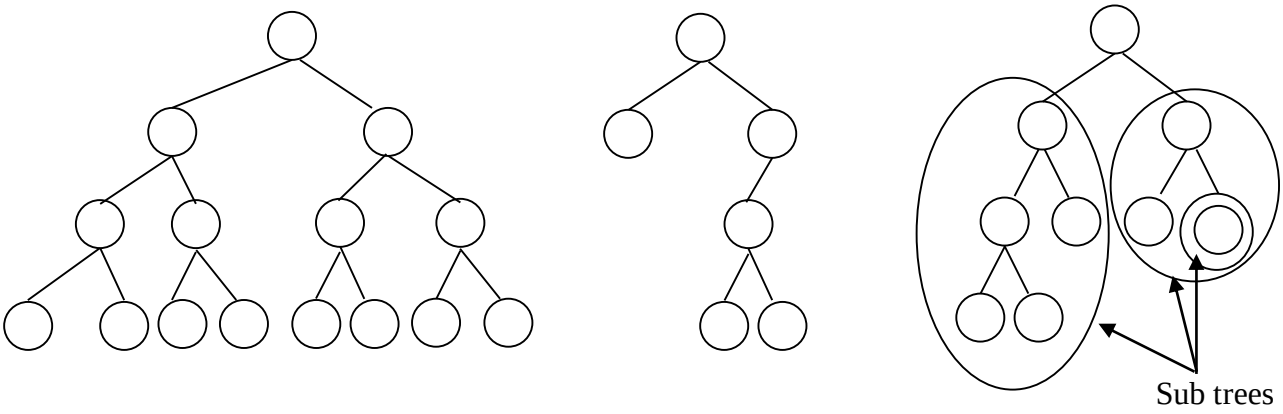
The right child of node contain L in location **(15)**

Q) Given a Binary tree with N levels , Count the minimum & maximum No. of :

	Max.	Min.
1- Nodes	$2^N - 1$	N (Why ?)
2- Leaves	$(2^{N-1} - 1) + 1$	1 (Why ?)
3- No of empty sub trees (no. of Null pointers)	$2^N - 1 + 1$	N +1 (Why ?)

4- Interior nodes = No. of Nodes – No. of Leaves

$$(2^N - 1) - ((2^{N-1} - 1) + 1) \qquad N - 1$$



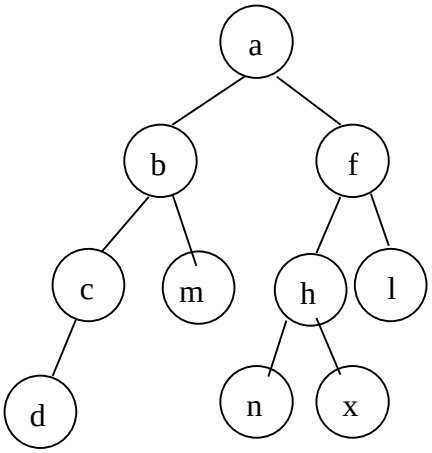
No. of Levels = 4 then

Max No. of Nodes = 15	Min No. of Nodes = 4
Max No. of leaves = 8	Min No. of Leaves = 1
Max No. of Interior nodes = 7	Min No. of Interior nodes = 3
Max No. of empty sub tree = 16	Min No. of empty sub tree = 5

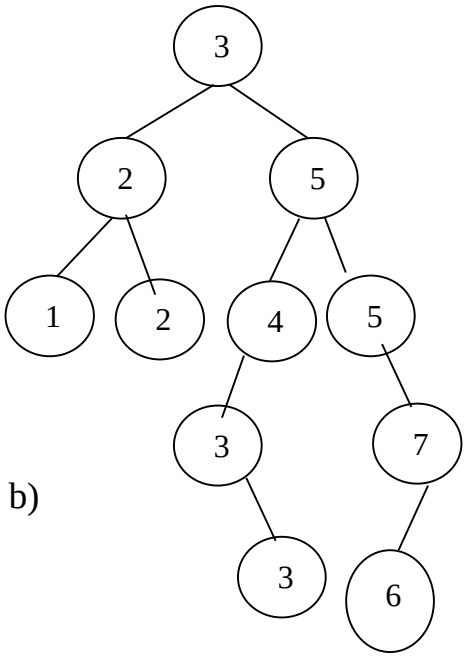
Home Work :

Count Max . & Min. No. of Nodes ,Leaves , Interior nodes , empty sub trees .
then allocate the tree in a suitable Array to the trees below :

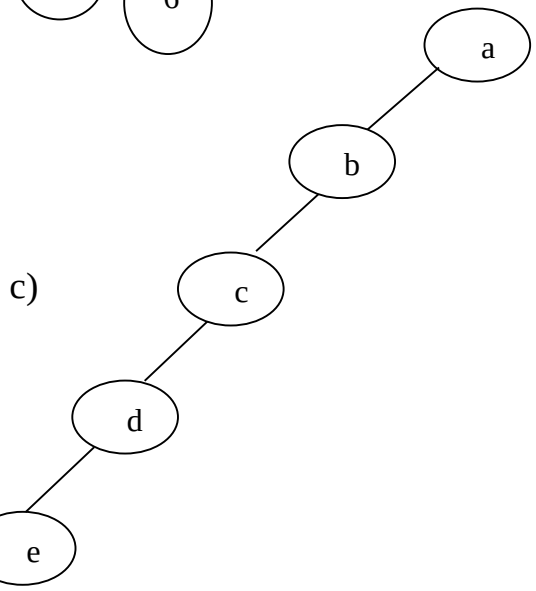
a)



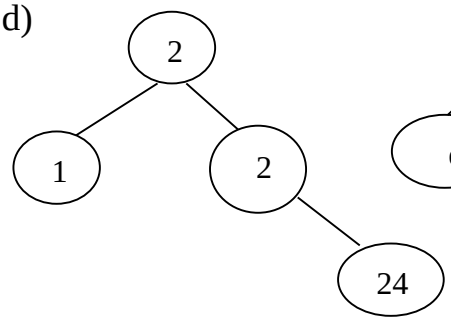
b)



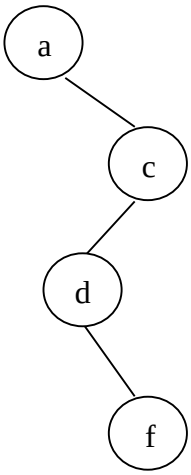
c)



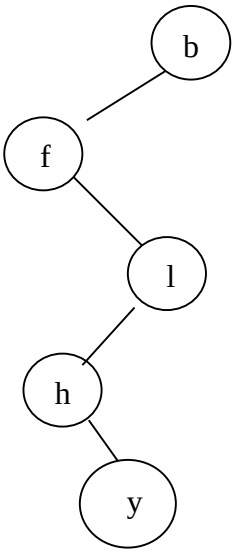
d)



e)



f)



Exercises : Draw a trees corresponding to a given expressions ,

then perform its prefix (or suffix) forms in addition to the original expressions

1- A B + C – B Z R ↑ / +

2- G B H * + T W / M ^ -

3- + - A B * / C D ^ E F

4- * / ^ 8 H / - R B + C D ^ k 2

5- / / * T R + K L ^ – 10 R + C B

Note that : An expression tree must be strictly tree ? Why ?

Home work (1)

Q2) Draw a binary tree to represent the expressions below ,
show how obtain the suffix and prefix forms ,
then allocate the tree in a suitable array.

1- (A+B/K) / L^(N* 5 *G)

2- (A*B+K) / 4 + (N* L*G)

3- H^B /(E+ H/3)*N^5

4- A B / E H C / - * N R ^ +

5- - + + A* B C / D E

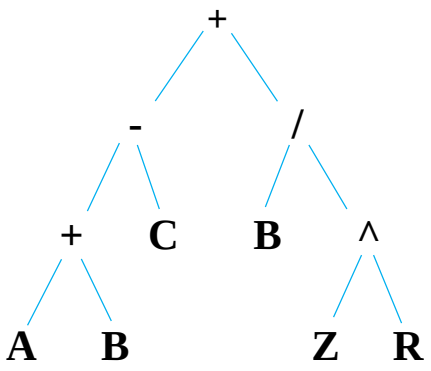
6- (A/B+K)^4^(N- L*G)

state the relationship between parents and sons

ملاحظه : من الممكن رسم الأشجار بدون دوائر للسرعه وخصوصا بالامتحانات وكما مبين ادناه

Exercises :

1- A B + C – B Z R ↑ / +

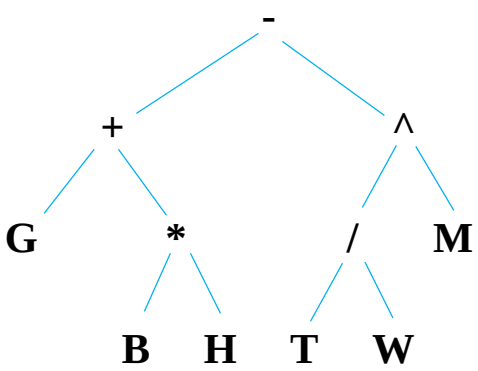


Prefix form : + - + A B C / B ^ Z R

Original Expression : ((A + B) – C) + (B / (Z^ R))

A + B – C + B / (Z ^ R)

2- G B H * + T W / M ^ -

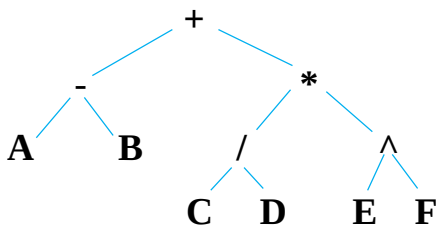


Prefix form : - + G * B H ^ / T W M

Original exp. (G + B* H) – ((T / W) ^ M)

G + B * H – (T / W) ^ M

3- + - A B * / C D ^ E F

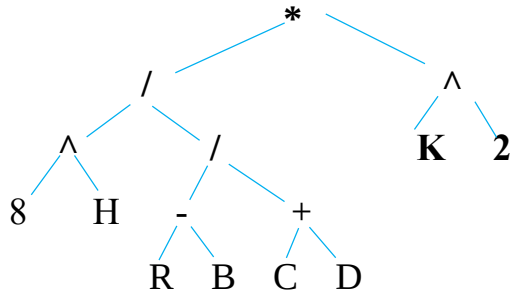


Suffix form : A B – C D / E F ^ * +

Original Exp. (A – B) + ((C / D) * (E ^ F))

A – B + (C / D * (E ^ F))

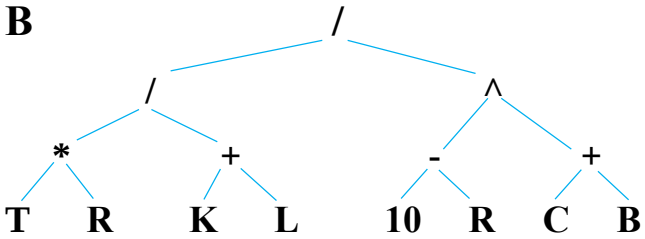
4- * / ^ 8 H / - R B + C D ^ k 2



Suffix form : 8 H ^ R B – C D + / / K 2 ^ *

Original Exp. ((8 ^ H) / ((R – B)/ (C + D))) * (K ^ 2)

5- / / * T R + K L ^ – 10 R + C B

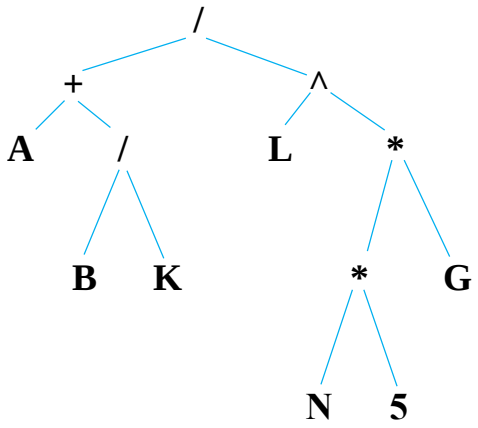


Suffix form : T R * K L + / 10 R – C B + ^ /

Original Exp. ((T* R) / (K + L)) / ((10-R) ^ (C + B))

Home work solution (Q2)

1- （A + B / K） / L ^（N * 5 * G）



Preorder traversal give prefix form :

/ + A / B K ^ L * * N 5 G

postorder traversal give suffix form :

A B K / + L N 5 * G * ^ /

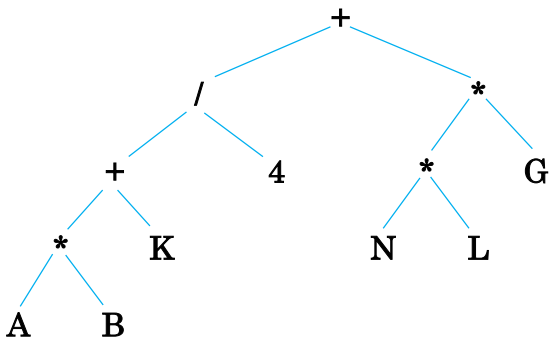
5

Array size = 2 - 1 = 32 -1

=31 location

/	+	^	A	/	L	*	…	B	K	…	*	G	…	N	5	…
1	2	3	4	5	6	7	…	10	11	…	14	15	…	28	29 …	31

2- （A * B + K）/ 4 + （N * L *G）



Preorder traversal give prefix form :

+ / + * A B K 4 * * N L G

postorder traversal give suffix form :

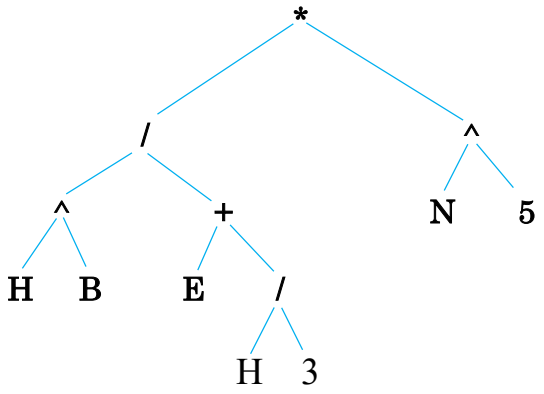
A B * K + 4 / N L * G * +

Array size = $2^5 - 1 = 32 - 1$

=31 location

+	/	*	+	4	*	G	*	K	…	N	L	…	A	B	…	
1	2	3	4	5	6	7	8	9	…	12	13	…	16	17	…	31

3- H ^ B /(E + H / 3) * N ^ 5



Preorder traversal give prefix form :

***/ ^ H B + E / H 3 ^ N 5**

postorder traversal give suffix form :

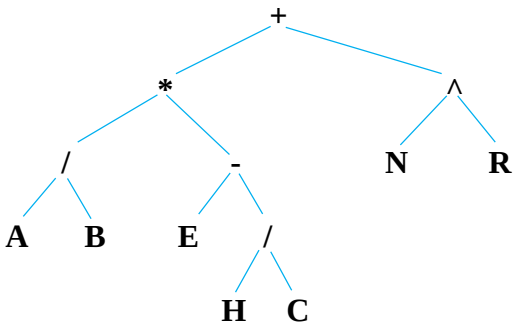
H B ^ E H 3 / + / N 5 ^ *

Array size = $2^5 - 1 = 32 - 1$

=31 location

*	/	^	^	+	N	5	H	B	E	/	…	H	3	…	
1	2	3	4	5	6	7	8	9	10	11	…	22	23	…	31

4- A B / E H C / - * N R ^ +



Preorder traversal give prefix form :

+ * / A B – E / H C ^ N R

Add parenthesis for each sub tree to obtain the general form :

((A / B) * (E – H / C)) + (N ^ R)

Note that : The inorder traversal give the original but without parentheses

A / B * E – H / C + N ^ R

Array size = $2^5 - 1 = 32 - 1$

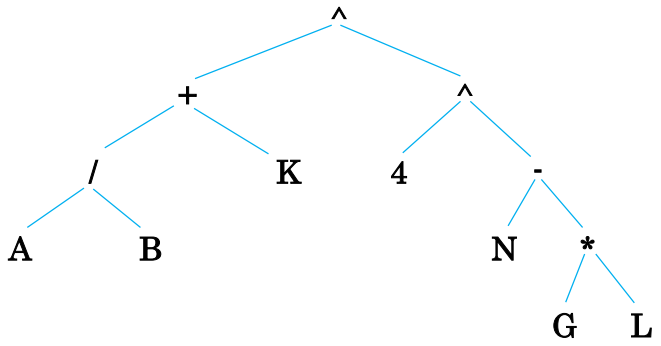
=31 location

+	*	^	/	-	N	R	A	B	E	/	…	H	C	…	
1	2	3	4	5	6	7	8	9	10	11	…	22	23	…	31

5 - + + A * B C / D E

Invalid expression (no, of operator = no. of operands)
The valid expression must have operator less than operands by one.

6- (A / B + K) ^ 4 ^ (N - G * L)



Preorder traversal give prefix form :

^ + / A B K ^ 4 – N * G L

postorder traversal give suffix form :

A B / K + 4 N G L * _ ^ ^

Note that : the exponentiation is a right associative operator.

Array size = $2^5 - 1 = 32 - 1$

=31 location

^	+	^	/	K	4	-	A	B	…	N	*	…	G	L
1	2	3	4	5	6	7	8	9	…	14	15	…	30	31

Q) State the relationship between parents and sons .

The answer : (all the parents are operators and its children (sons) are operands.
The expression tree satisfy the equation below:
No. of nodes = (2 * No. of leaves -1)

Home work solution (Q1)

Q1) Draw a binary tree to represent the expressions below then write the original forms:

1- / + * A R ^ Z F - B ^ H L

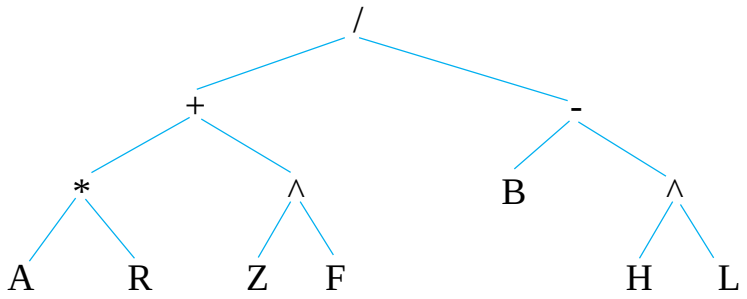
2- Z R / B A * H / + M 2 ^ -

3- / + R 5 F * B - N * H L

4- A Z ^ B C * H / - L N / +

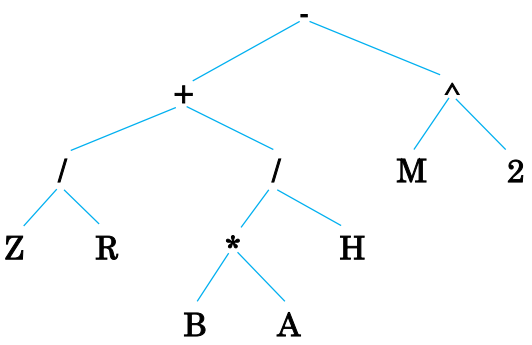
Solution :

/ + * A R ^ Z F - B ^ H L



The original Exp. ((A* R) + (Z ^ F)) / (B – (H ^ L))
Equivalent to : (A* R + Z ^ F) / (B – H ^ L)

Z R / B A * H / + M 2 ^ -

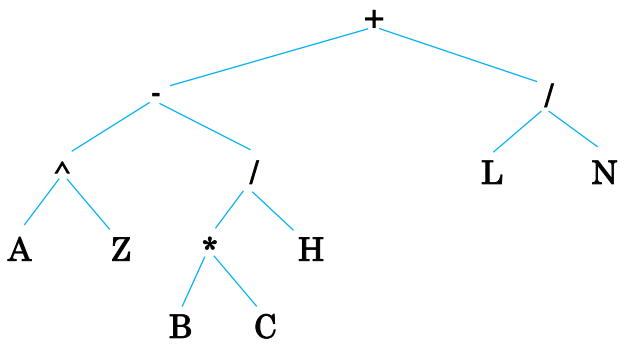


The original Exp. ((Z / R) + (B * A / H)) – (M ^ 2)
Equivalent to : Z / R + B * A / H – M ^ 2

/ + R 5 F * B - N * H L

Invalid expression (7 operands and 5 operators) (to become valid add operator or delete operands in suitable location)

A Z ^ B C * H / - L N / +

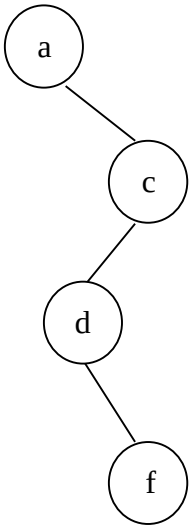


The original Exp. ((A ^ Z) – (B * C / H)) + (L / N)

Equivalent to : A ^ Z – B * C / H + L / N

Home work Lec . 2 : Write the preorder ,inorder and postorder traversel to the trees below :

a)

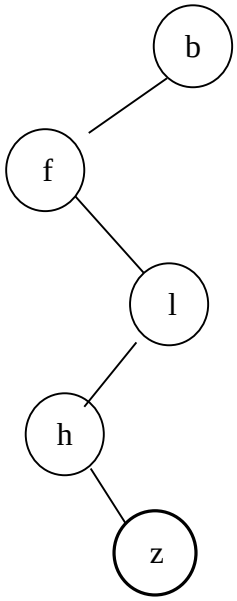


Preorder traversal : a , c, d, f

Inorder traversal : a, d, f, c

Post order traversal :f, d, c, a

b)

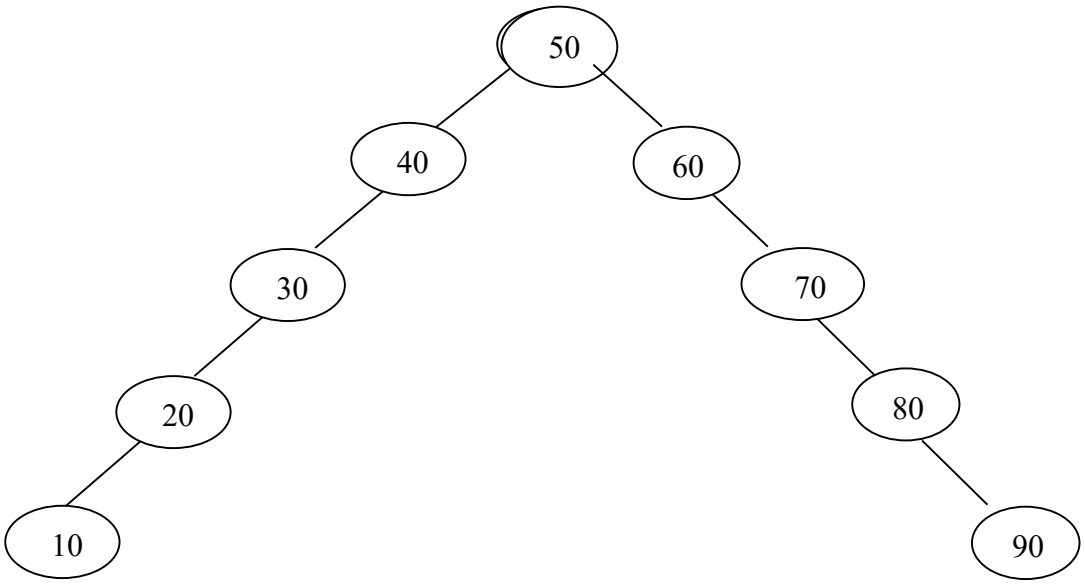


Preorder traversal : a , f , l , h , z

Inorder traversal : f , h , z , l , b

Post order traversal : z , h , l , f , b

c)



Preorder traversal : 50 , 40 , 30 , 20 , 10 , 60, 70 , 80, 90

Inorder traversal : 10, 20, 30, 40, 50, 60 , 70 , 80, 90

Post order traversal : 10 , 20, 30, 40, 90, 80, 70, 60 , 50

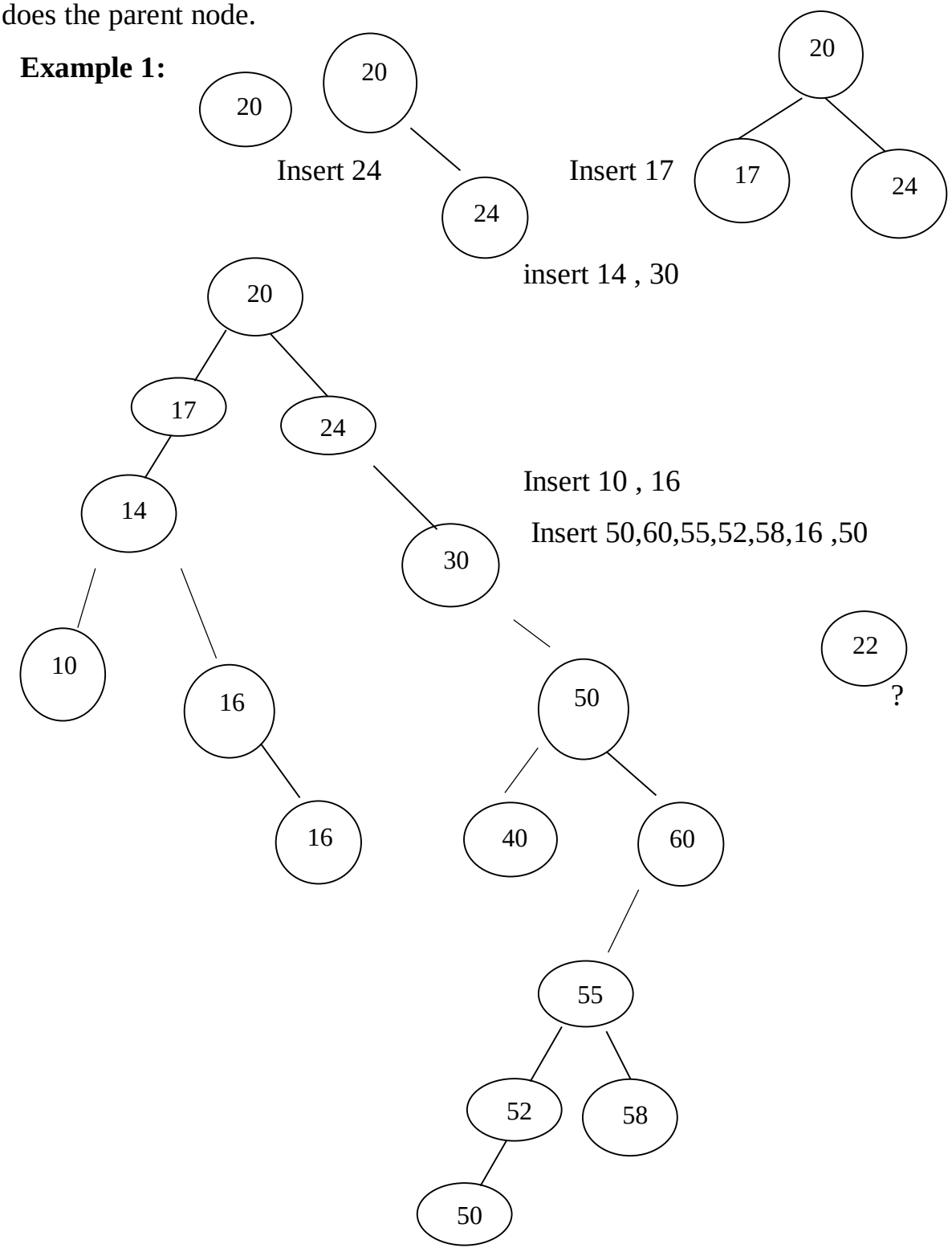
-

-

2- Using binary tree to obtain sorted data by building binary search tree (BST):

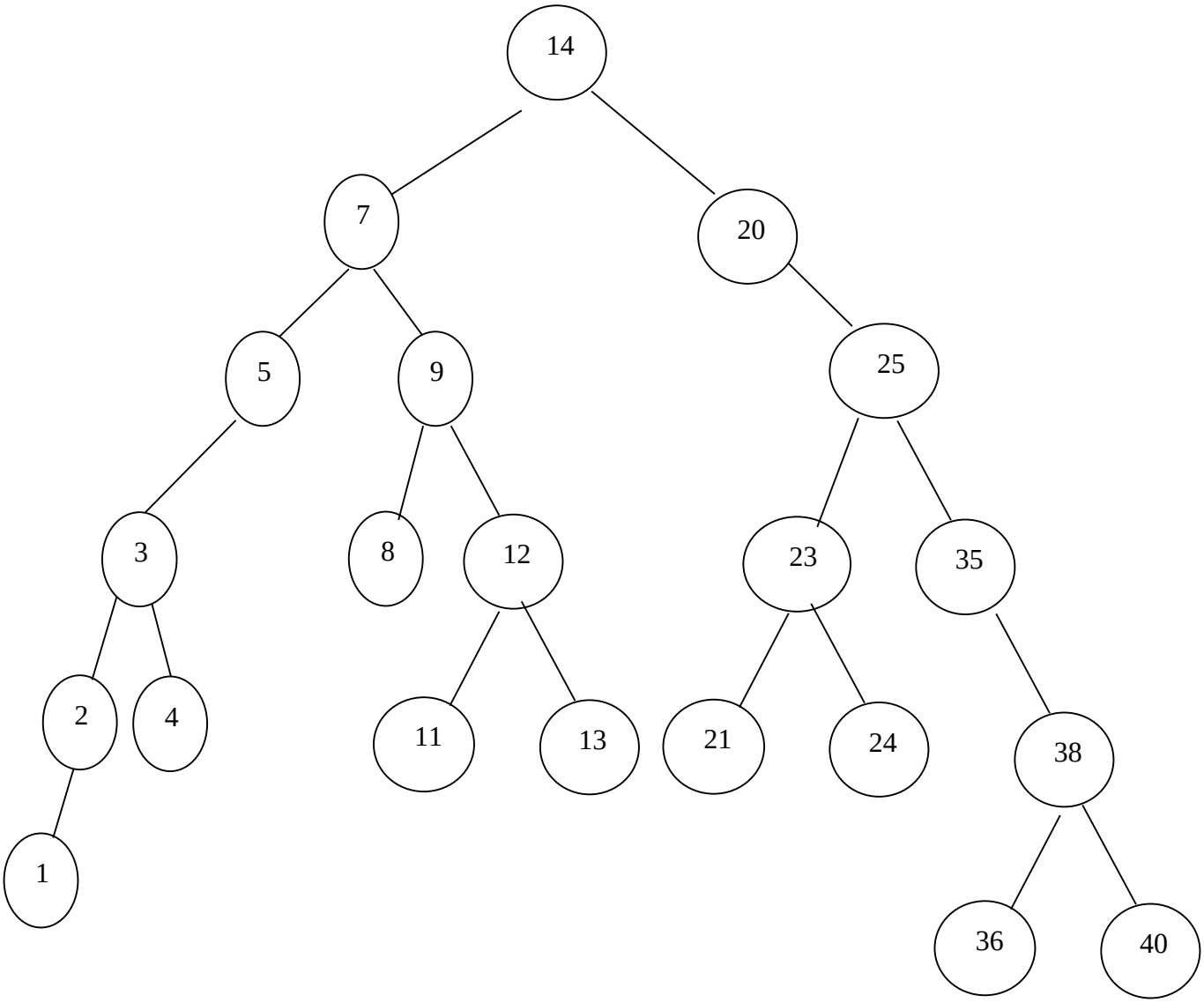
Binary Search Tree: is a binary tree, in which left child (if any) of any node contains a smaller value than does the parent node and the right child (if any) contains a larger value than does the parent node.

Example 1:



The inorder traversal of any BST gives data in ascending order while the convert inorder traversal of any BST gives data in descending order.

Example 2: 14, 7, 9, 20, 5, 12, 8, 11, 25, 35, 24, 21, 11, 3, 4, 13, 2, 38, 40, 1, 36



The first value 14 at the root

Insert 7 (less than 14) then (left branch) reside as left son to the root (14)

Insert 9 (less than 14) then (left branch) reside as left son to the root (14)

(9 > 7) set as a right son to the node contain (7)

Insert 20 (greater than 14) then (right branch) reside as Right son to the root (14).

Insert 5 (less than 14) then reside as left son to the root (14), (5< 7) set as a left son to the node contain (7).
Insert 12 (less than 14) then reside as left son to the root (14), (12 > 7) set as a right son to the node contain (7), (12 > 9) then set as right son to the node contain (9) . and so on

Inorder Traversal: 1 2 3 4 5 7 8 9 11 12 13 14 20 21 23 24 25 35 36 38 40

Converse Inorder Traversal:40 38 36 35 25 24 23 21 20 14 13 12 11 9 8 7 5 4 3 2 1

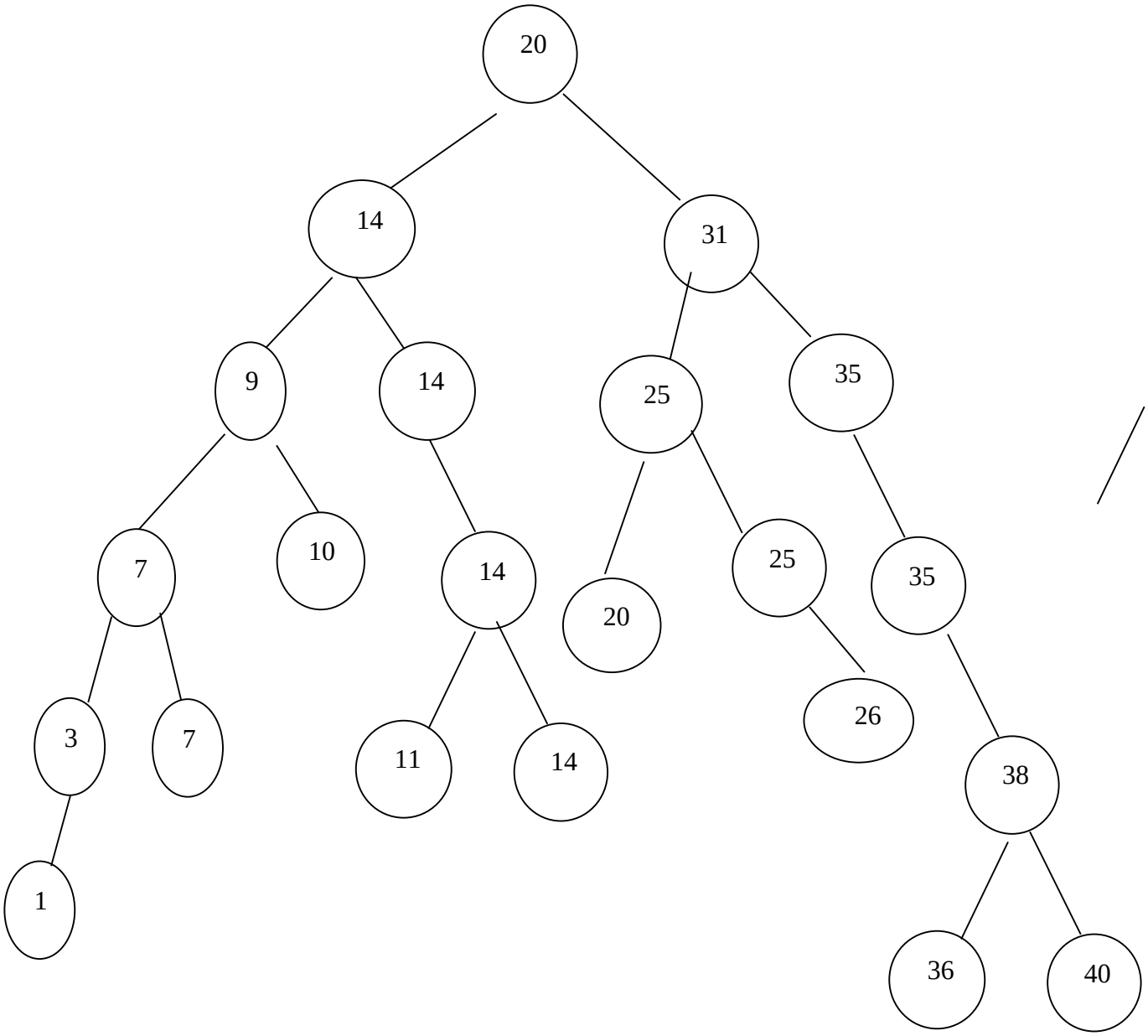
Note that: all the value less than the root inserted in left sub tree and all values greater than or equal to the root value insert at the right sub tree.

3- Using binary tree to Find duplicated data:

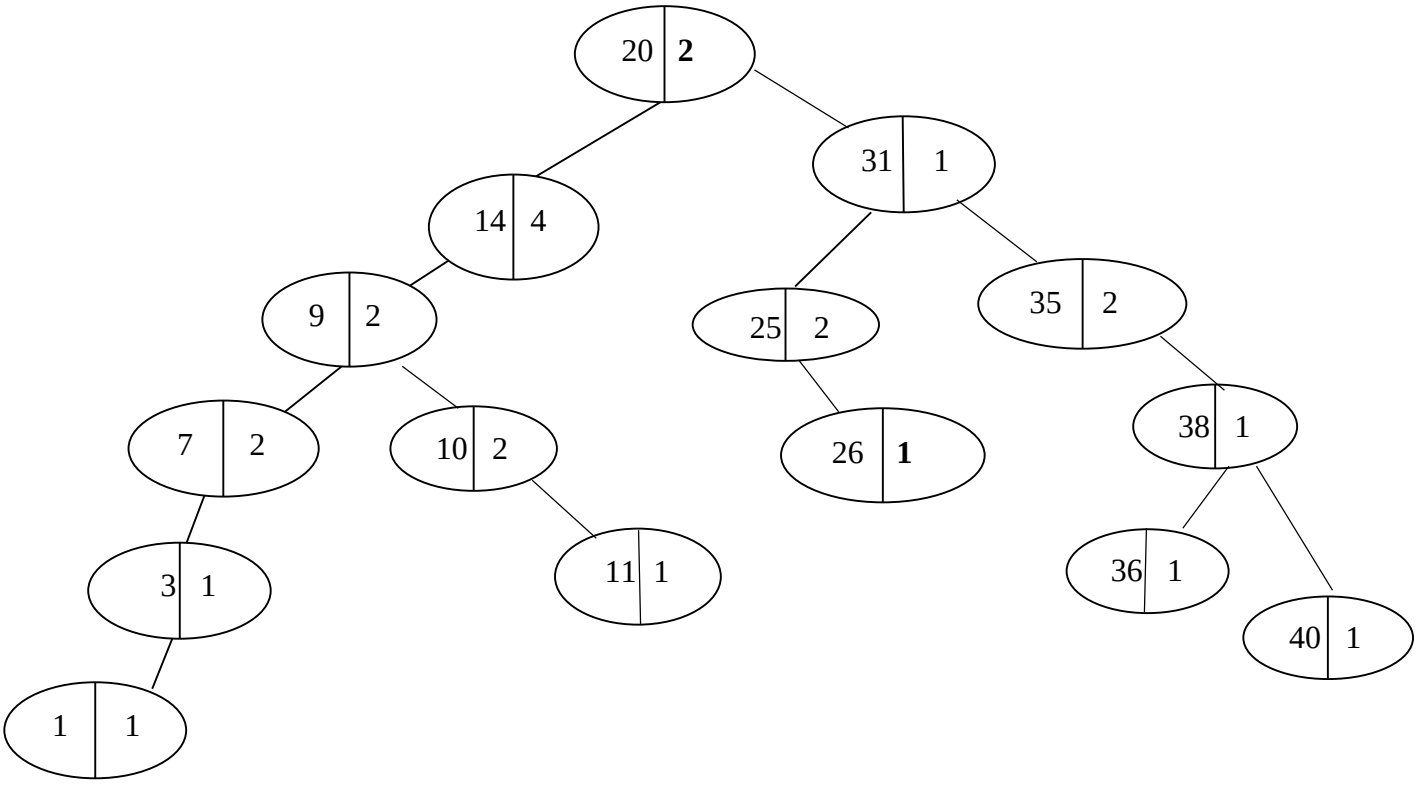
To find duplicated data, another field must be added to the node to compute the redundancy of each element of the tree.

Example 3: Find the duplicated data if you given the following data:

20, 14 , 31 , 14 , 9 , 7 , 25 , 35 , 10 , 3 , 14 , 7 , 25 , 35 , 1 , 14 , 26 ,38 ,11,36,40 ,20



No. of nodes = ?



No. of nodes = ?

The inorder traversal to a **(BST)**

No.	Occ.	No.	Occ.
1	1	35	2
3	1	36	1
7	2	38	1
9	2	40	1
10	2		
11	1		
14	4		
20	2		
25	2		
26	1		
31	2		

Note that : A Binary search tree used to search for any given value

The search algorithm same as insertion

Questions :

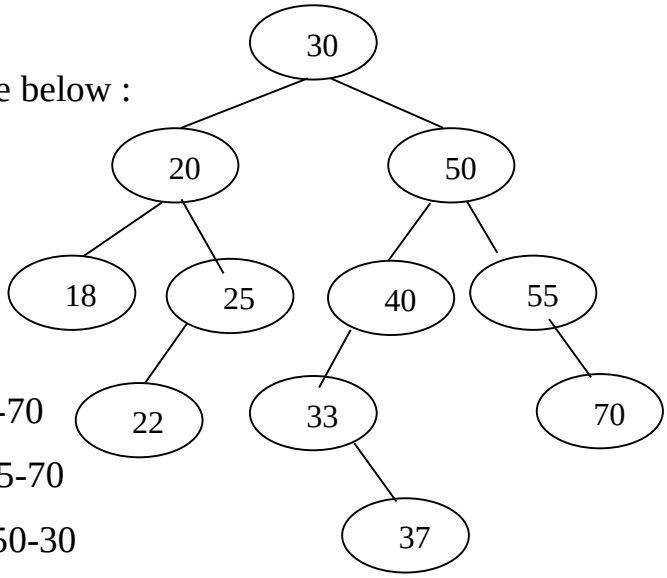
1- Which method is the best to store duplicate values allocate a node to each value or use occurrence field ?

2- In which case prefer to use occurrence field in stead of store values ?

3- What about the number of nodes in each methods ?

Example 4:

Write all the traversals for the Binary Search Tree below :



Inorder: 18-20-22-25-30-33-37-40-50-55-70

Preorder: 30-20-18-25-22-50-40-33-37-55-70

Postorder: 18-22-25-20-37-33-40-70-55-50-30

Converse Inorder: 70-55-50-40-37-33-30-25-22-20-18

Converse Preorder: 30-50-55-70-40-33-37-20-25-22-18

Converse Postorder: 70-55-37-33-40-50-22-25-20-18-30

Level by Level (top-down): 30-20-50-18-25-40-55-22-33-70-37

Level by Level (bottom-up): 37-22-33-70-18-25-40-55-20-50-30

Converse Level by Level (top-down): 30-50-20-55-40-25-18-70-33-22-37

Algorithms

1- Algorithm Create Tree

[This algorithm is to create a tree with single node (Root) contains x value]

If (root =NULL) **{**

p←avail (obtain new node)
avail ← Link (avail)
Info(p)← x (set info and link fields)
Lptr(p)←NULL
Rptr(p)←NULL
Root←p **} (save the root node)**

2- Algorithm Create node

[This algorithm is to create node called (po) contains information called (x), so its input is value of (x) and return node called (po)]

{ po←avail
avail ← Link (avail)
Info(po)← x
Lptr(po)←NULL
Rptr(po)←NULL **}**

3- Algorithm Rinsert

[Given a binary search tree pointed by root, this is a recursive algorithm to insert a new node to the tree contains a new information called (x)]

1- [check left subtree]

```
If (x < info (root)) then {
        If (Lptr(root)= NULL ) //insert a new node as a left subtree
                {
                 Newnode← create node
                 Lptr(root)←new node
                }
        Else
         Call Rinsert (Lptr(root),x)
         }
```

2- [check right subtree]

```
If x ≥ info (root) then {
        If (Rptr(root)= NULL) //insert a new node as a right subtree
                {
                 Newnode← createnode
                 Rptr(root)←newnode
                }
         Else
        Call Rinsert (Rptr(root),x) }
```

4- algorithm Rpreorder

[This algorithm is recursive algorithm for printing the contents of each node in any binary tree traversed in preorder]

```
If (root = NULL) // check empty tree
        print message "empty tree"
        Else {
                print (info(root))
                If (Lpte(root)≠NULL) // print out left subtree
                    Rpreorder (Lptr(root))
               If (Rptr(root)≠NULL) // print out right subtree
                    Rpreorder (Rptr(root))
              }
```

5- algorithm Rinorder

[This algorithm is recursive algorithm for printing the contents of each node in any binary tree traversed in inorder]

```
       If (root = NULL) // check empty tree
             print message "empty tree"
        Else {
                If (Lpte(root)≠NULL) // print out left subtree
                     Rinorder (Lptr(root))
               print (info(root))
               If (Rptr(root)≠NULL) // print out right subtree
                    Rinorder (Rptr(root))
              }
```

6- algorithm Rpostorder

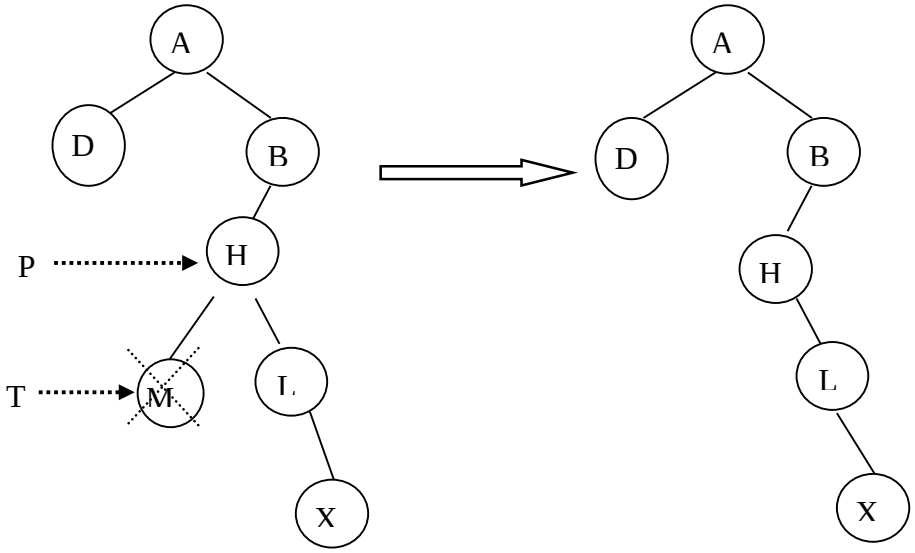
[This algorithm is recursive algorithm for printing the contents of each node in any binary tree traversed in postorder]

```
If (root = NULL) // check empty tree
        print message "empty tree"
    Else {
        If (Lpte(root)≠NULL) // print out left subtree
            Rpostorder (Lptr(root))
        If (Rptr(root)≠NULL) // print out right subtree
            Rpostorder (Rptr(root))
            print (info(root))
        }
```

5- Algorithm delete [there are 3 cases in deletion]

a- if the deleted node has no sons (it is a leaf)

- if deleted node denoted by pointer variable (T) then it's parent denoted by (P). If (T) is left child:-

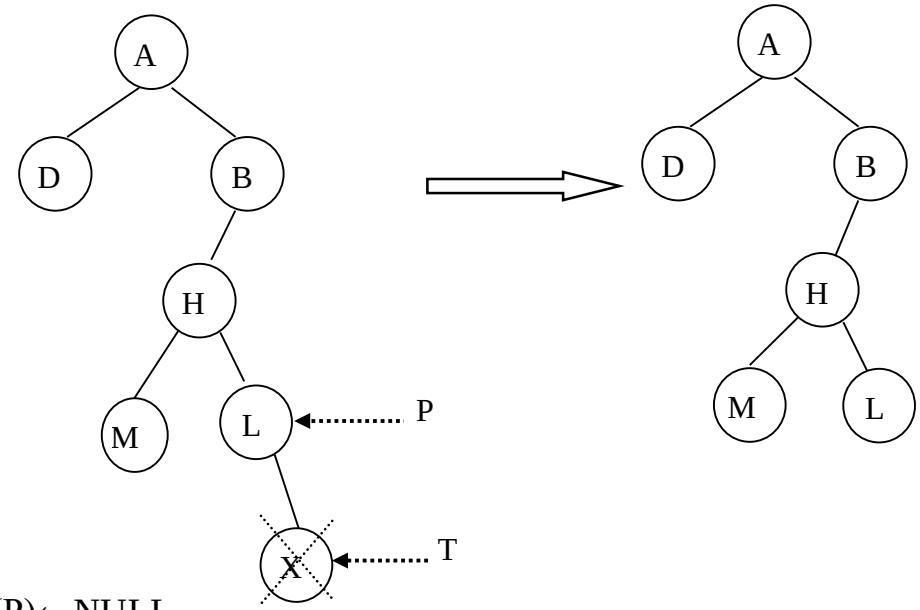


Lptr(P)←NULL

Link(T) ← avail

avail ← T

- if deleted node denoted by pointer variable (T) then it's parent denoted by (P). If (T) is right child:-

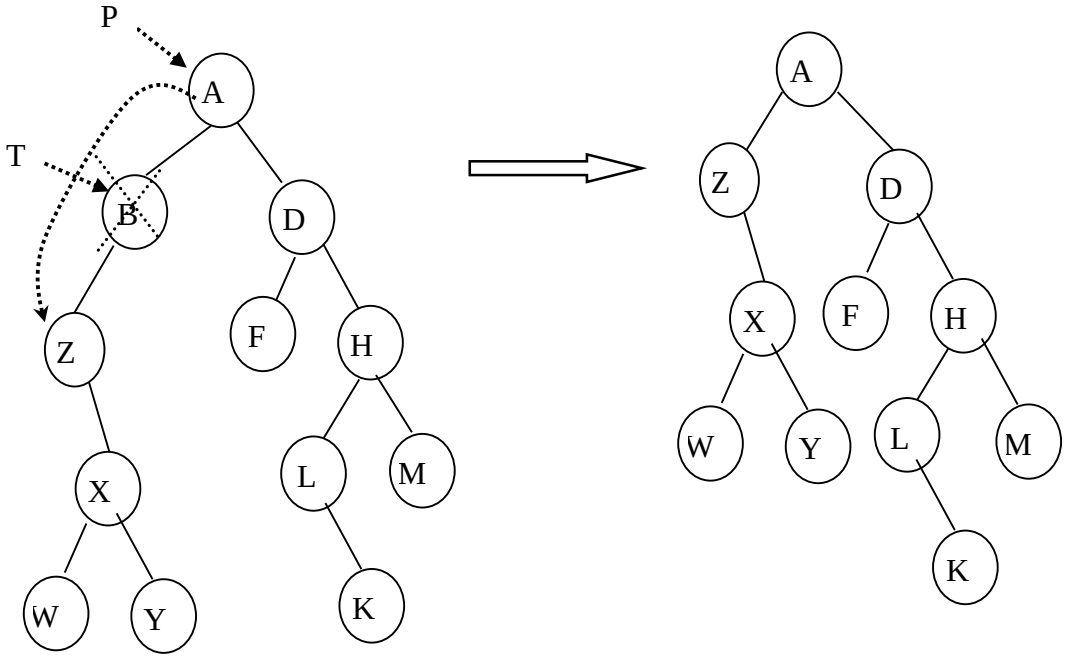


Rptr (P)←NULL

Link(T) ← avail , avail ← T

if the deleted node have left or right subtree and denoted by (T) then It's son can be moved to take it's place.

- If (T) has left subtree and (P) is T's parent:-

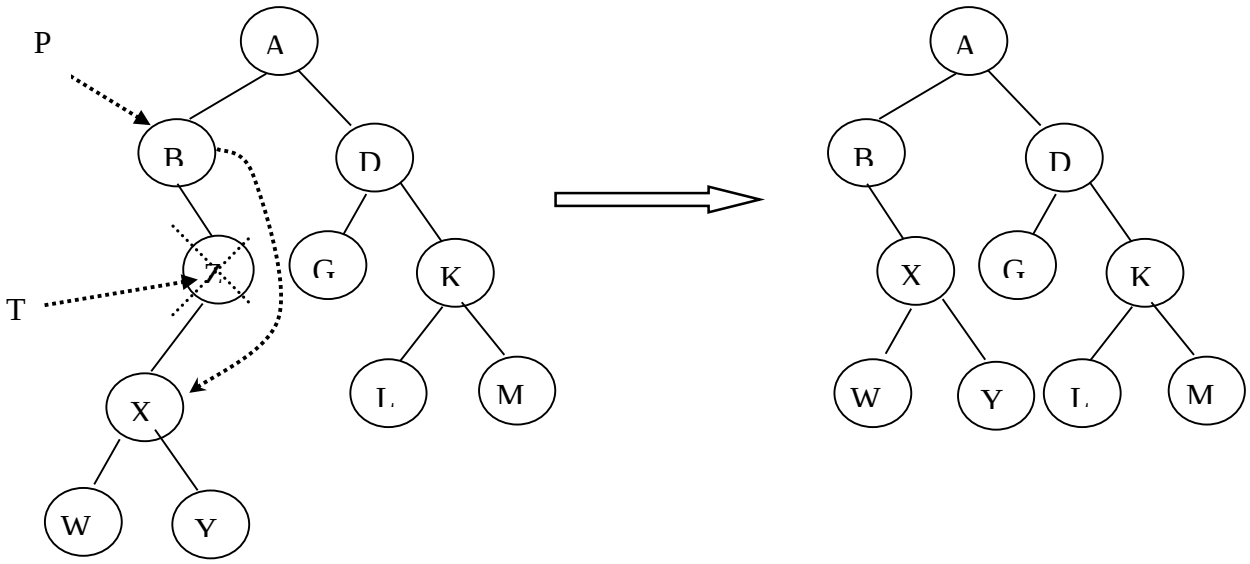


Lptr(P)←Lptr(T)

Link(T) ← avail

avail ← T

- If (T) has right subtree and (P) is T's parent:-



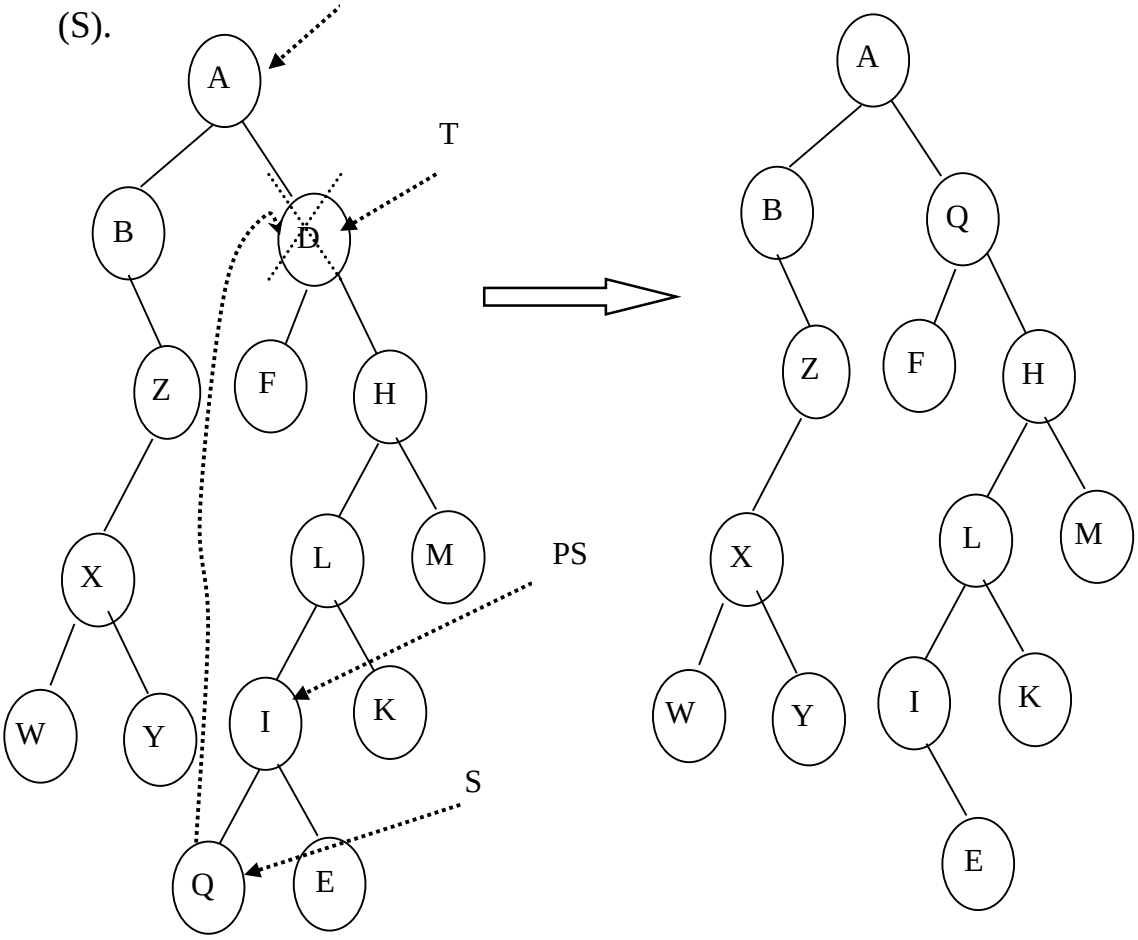
Rptr (P)←Lptr(T)

Link(T) ← avail

avail ← T

c- If the deleted node has two subtrees

If deleted node has left subtree and right subtree then it's successor in inorder traversal must take it's place. Let (T) means deleted node then (PT) will be parent of (T) and let (S) n PT successor of (T) in inorder traversal then (PS) is parent of (S).



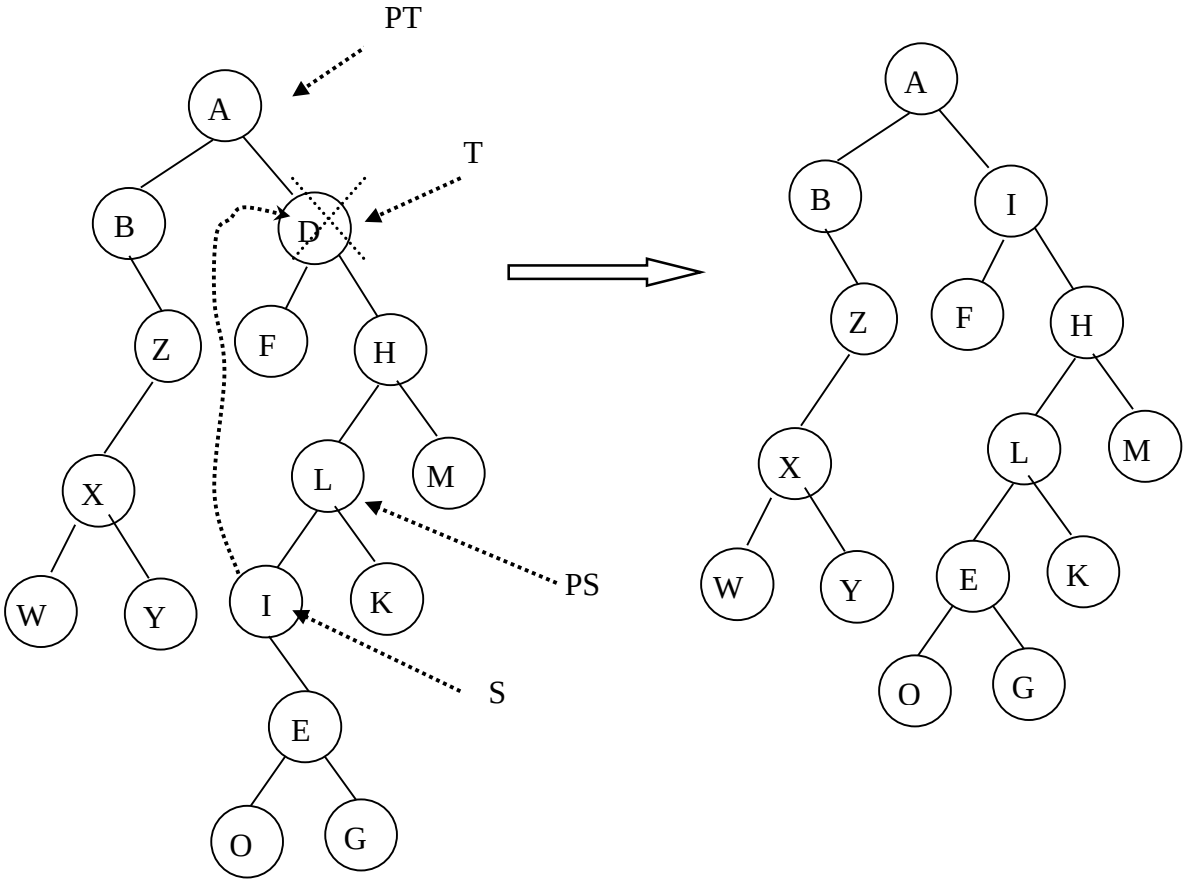
- If (S) is a leaf node.

Lptr(PS)←NULL

Lptr(S)←Lptr(T)

Rptr(S)←Rptr(T)

Rptr(PT)←S



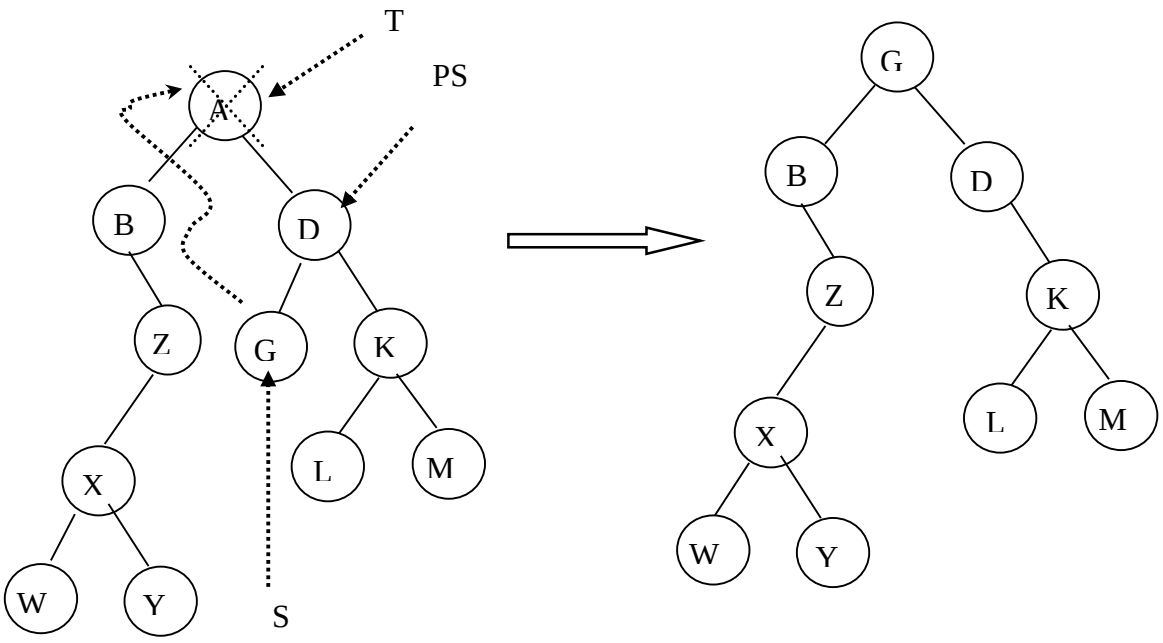
- If (S) is a node that has one child or one subtree.

Lptr(PS)←Rptr(S)

Lptr(S)← Lptr(T)

Rptr(S)← Rptr(T)

Rptr(PT)←S



- If (T) is a node that has two children or two subtrees (right and left) and (T is a root node).

  Lptr(PS)←NULL

  Lptr(S)←Lptr(T)

  Rptr(S)←Rptr(T)

  Root←S

- **Advantages of Tree:**

  Search operation of any node in a Binary Search Tree takes half time of search in linear data structures like queue because it searches either in right subtree or in left subtree.

- **Disadvantages of Tree:**

  1- Its programming is difficult operation in the languages that not support the pointers.

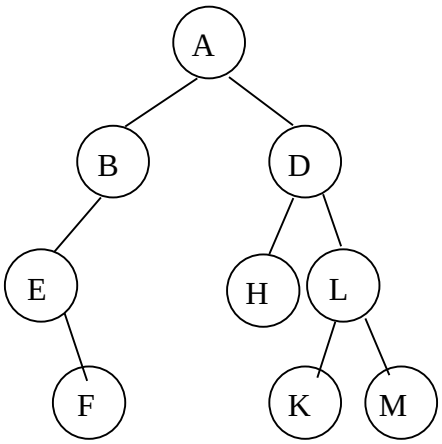
  2- It takes more space to store the pointers.

  3- There is a difficulty to reach to the parents.

The Sequential Representation of Binary Tree

The complete binary Search Tree can be numbered from 1 to n so that the number assigned to the left son is twice the number assigned to it's parent and the number assigned to the right son is one more than twice the number assigned to it's parent. (i.e) the node in position (p) is the parent of the nodes in position (2p) and (2p +1). If the left child at position (p) then it's right brother in position (p+1) and if the right child at position (p) then it's left brother in position (p-1). If any node in position (k) then it's parent in position trunc (k/2). Then we will store the tree in the array level by level from left to right.

Example:



A	B	D	E		H	L		F					K	M
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- In sequential allocation, the returning to the root is easy. While in linked allocation, it is hard to return to the root, also there is increasing in reserving space since there is a need to reserve the left and right pointers.

Sorting of Tree

Two sorting methods are based on a tree representation of data:

1- The straight forward Binary Search Tree (BST).

2- The Heap Sort which is involving binary tree in much more complex structure called "**Heap**".

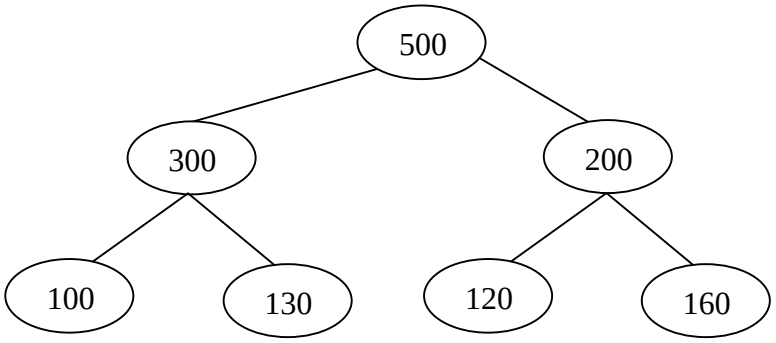
Heap Structure: is a complete tree with some of right most leaves removed. There are two types of heap sort: 1- Max heap and Min heap. The Max heap represents a table of records that satisfies the following properties:-

1- (max heap) $k_j \leq k_i$ for $2 \leq j \leq n$ and $i = \lfloor j/2 \rfloor$

2- The record with largest key is at the root of the tree called the top of the heap.

3- Any path from the root to a leaf is sorted list in descending order.

While Min heap represents a table of records that satisfies the opposite properties of max heap.



Heap Sort algorithm:

1- Build heap structure: arrange the data using tree then put the data of resulted tree in an array.

2- Reheaping: is the process of printing the data of the heap by deleting root until the tree became empty tree.

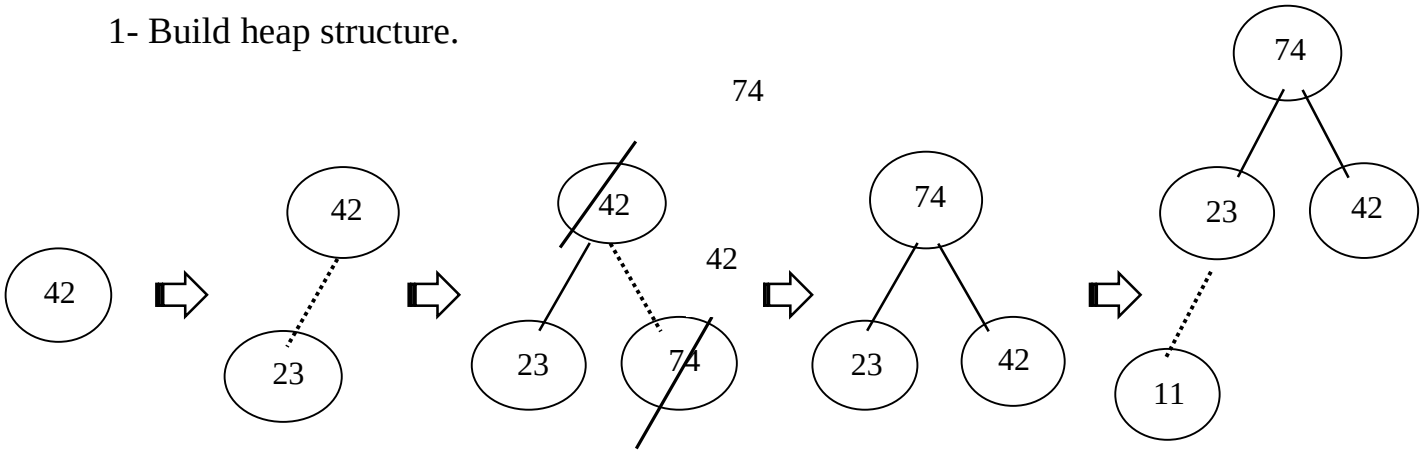
Note that:

- The heap structure must be represented in a sequential allocation.
- If we build a Min heap then the data will be sorted in ascending order.

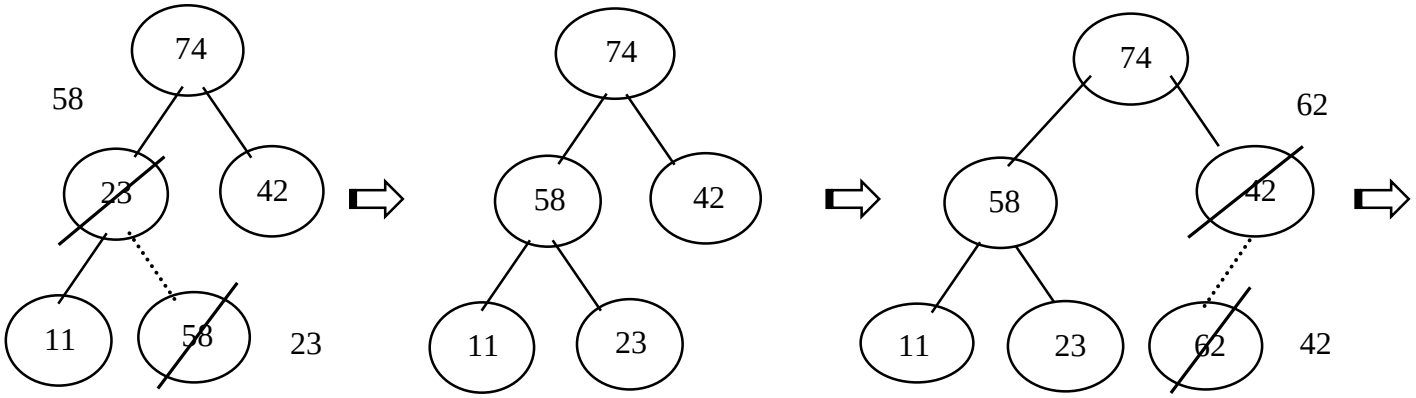
Example:

Use the following data to construct the heap structure then sort the data in descending order: 42, 23, 74, 11, 58, 62, 94, 99, 87, 36.

1- Build heap structure.

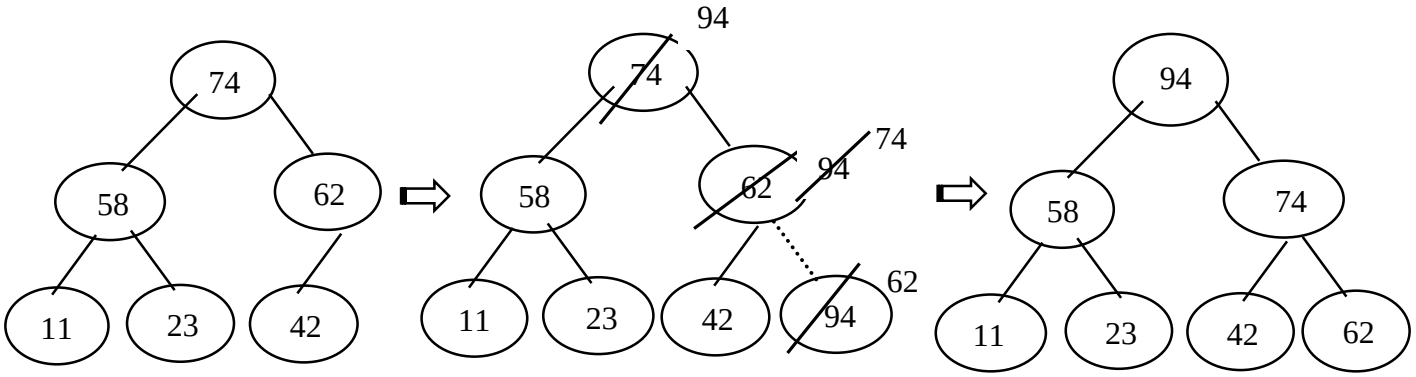


insert “42” insert “23” insert “74” insert “11”

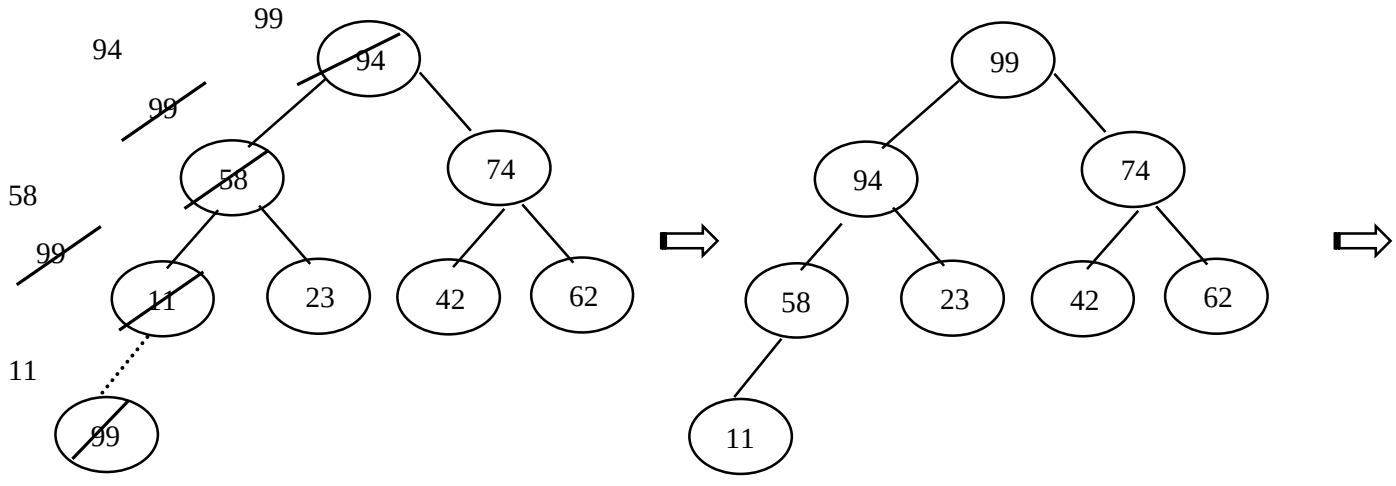


insert “58”

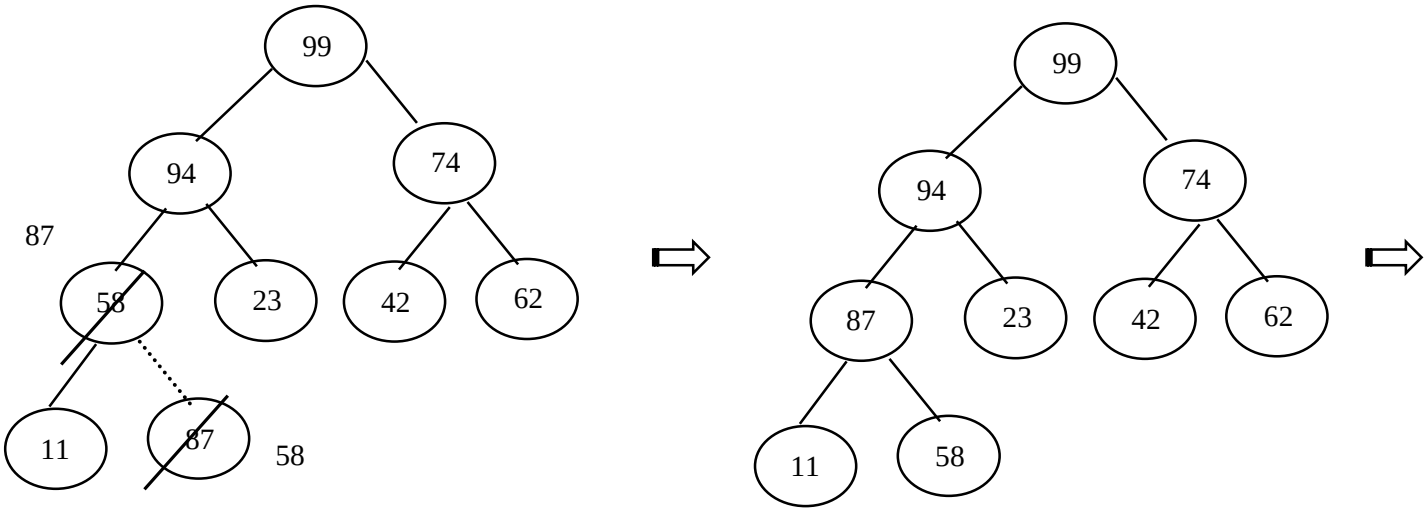
insert “62”



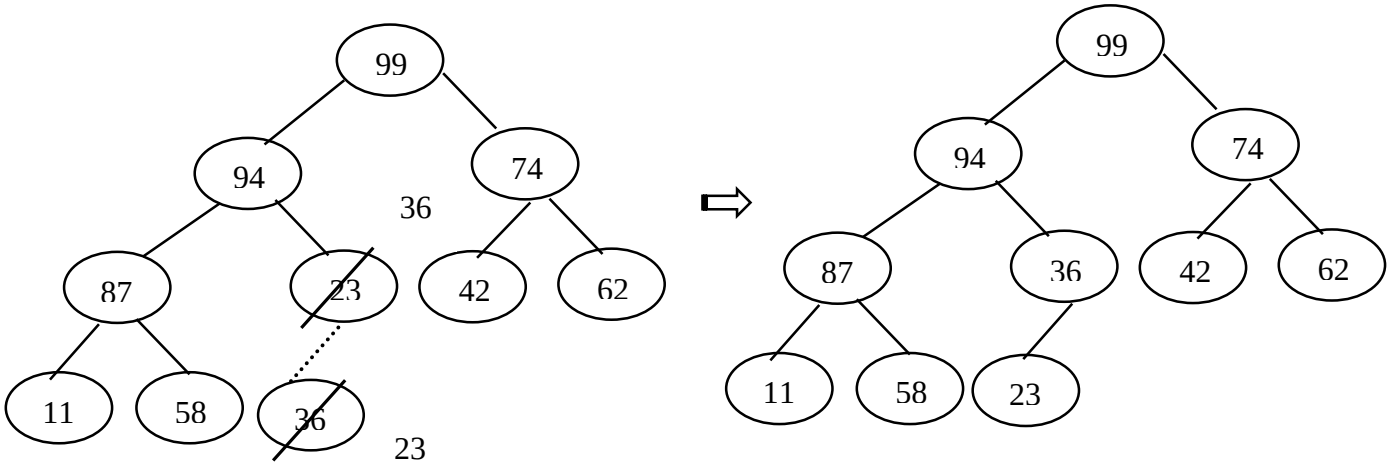
insert “94”



insert “99”



insert “87”



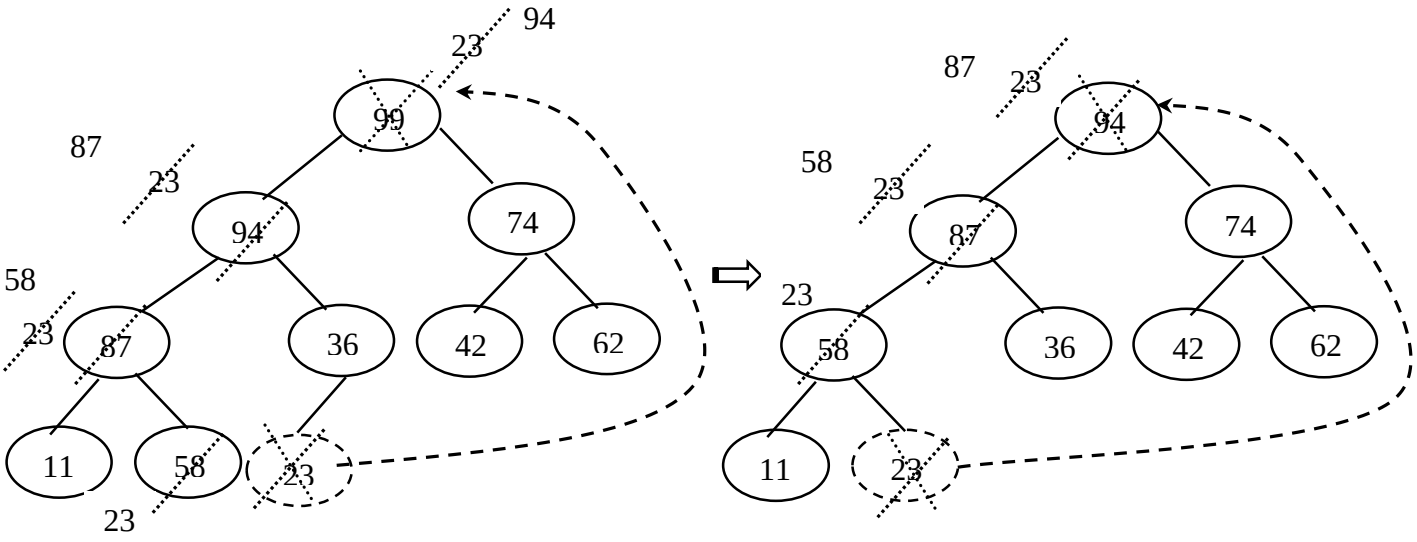
insert “36”

Q) How a heap represent in memory ? Why ?

Answer : The heap represent in memory as array (Static) because the relation between a parent and its child are twice and half.(easy to reach to any location directly)

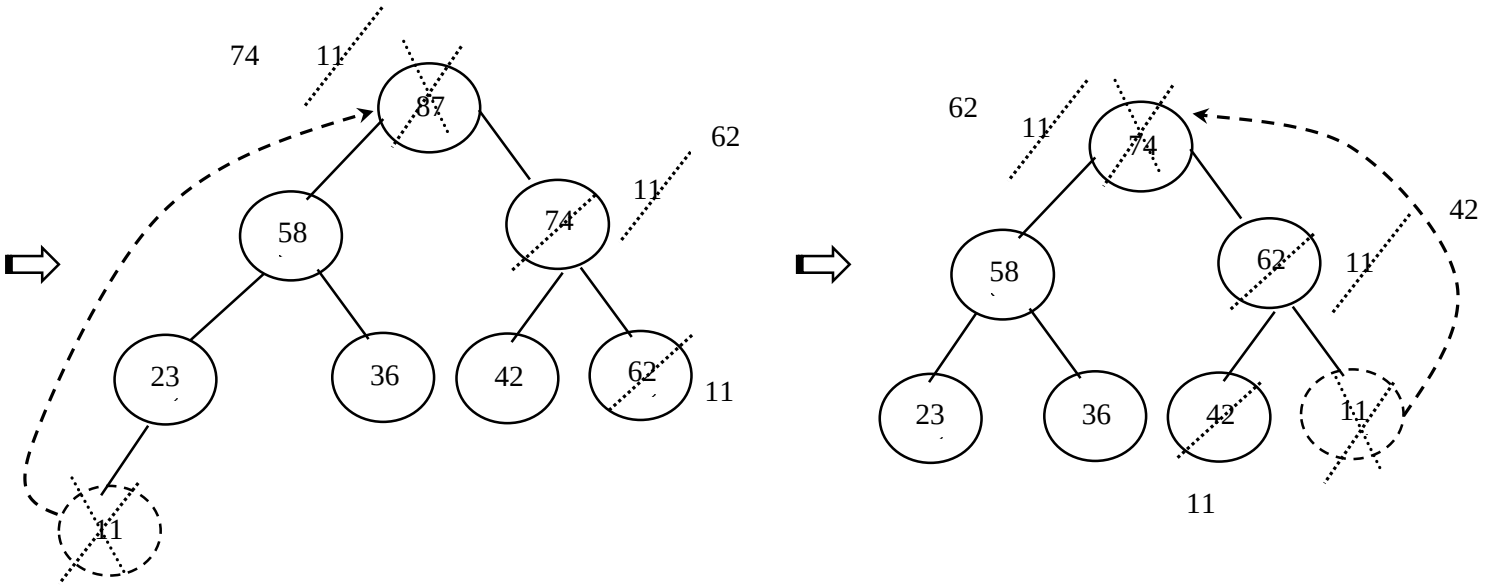
99	94	74	87	36	42	62	11	58	23

2- Reheaping



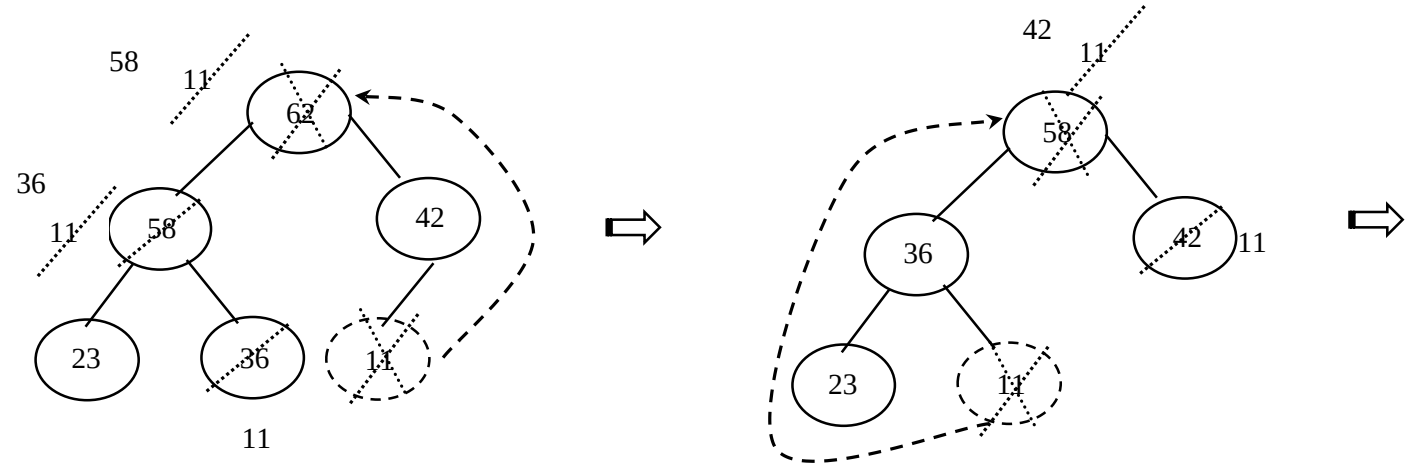
delete “99”

delete “94”



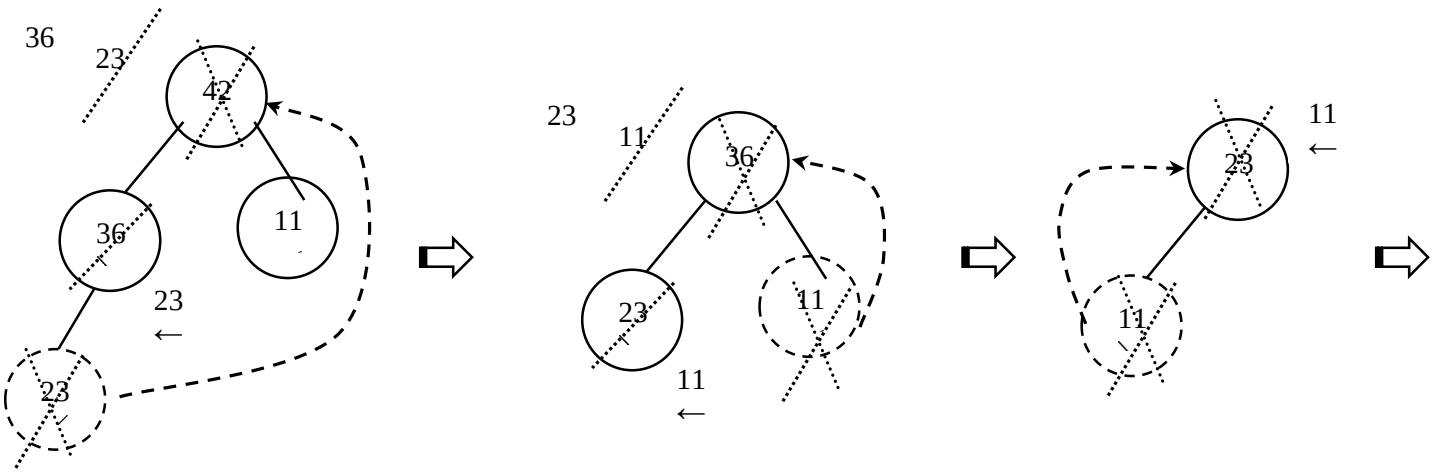
delete “87”

delete “74”



delete ”62”

delete “58”

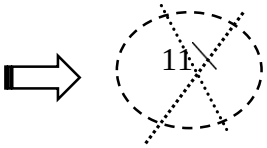


delete “42”

delete “36”

delete “23”

delete “11”



Data structure / Second class / Second Term

Assist. Prof. Makia K. Hamad.

Makia 4.4.2015

Sorting methods Sorting and searching Trembe CH-6

Def. :- sorting operation is the arrange of the record in table into some sequential order according to the key values, the sort either ascending or descending and the key may be either numeric or non-numeric (alphanumeric), we shall divide the subject into two parts, (internal and external sort).

Sorting algorithms are designed with the following objectives:-

1- minimize exchange or movement of the data.
2- move data from secondary storage to main memory in large blocks, this is the key part of external sorting.
3- if possible to return all the data in main memory, in this case random access into array can be effectively sed this is the key part of internal sorting.

Programmer considerations

There are three main considerations which should effect a programmer decision to chose from a varity of sorting methods:-

1- programming time.
2- execution time of the program.
3- memory or auxiliary space needed for the program environment.

The three general methods for sorting : exchange, selection, insertion.

Some common sorting method:-

1- selection sort
2- bubble sort
3- insertion sort
4- merge sort
5- shell sort
6- radix sort
7- quick sort
8- heap sort
9- bin sort

selection sort

One of the easiest ways to sort a table is by selection, the general form:

```
for i := 1 to n-1 do
    for j := i+1 to n do
        if a[i] > a[j] then swap ( a[i] , a[j] )
```

in this method we search for the record with next smallest element is called a pass. There are n-1 passes required in order to perform the sort of n elements, this because each pass places one record into its proper location.

The maximum number of interchanges in this sort is (n-1).

In first pass (n-1) element are compared in the second pass (n-2) … and so on.

In general for the ith pass (n-i) comparison are required, the total sum of comparisons there is $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.

Bubble sort

One of the well known sorting methods is the bubble sort. It differ from the selection sort is instead of finding the smallest record and then performing interchange, two records are interchange immediately upon discovering that they are out of order, when this approach are used there are (n-1) passes required. This method will cause records with small key to move or bubble up after the first pass, the record with largest key will be in the n-th position.

The general loop:

```
For i := 1 to n-1 do
    For j := 1 to n-i do
        If a[j] > a[j+1] then swap (a[j] , a[j+1])
```

Another form:

```
For i := 1 to n-1 do
    For j := n downto i+1 do
        If a[j] < a[j-1] then swap (a[j] , a[j-1])
```

Note:- in bubble sort we can use a Boolean flag to shut off loop with no interchange are made. This cause increasing run time efficiently.

```
Procedure bubble2 (n:integer);
var   i:integer;
      t:  k;        same as the type of an array
      nochange:boolean;
begin
      i:=0;
      repeat  nochange:=true;
              i:=i+1;
             for j:=1 to n-I do
                   if a[j] > a[j+1]  then  begin
                                           t:=a[j];
                                           a[j]:=a[j+1];
                                           a[j+1]:=t;    no change := false ;
                                           end;
      until   (i=n-1) or nochange
end;
```

Insertion sort

The main idea behind the insertion is to insert in the i-th pass the ith element in a[1], a[2], a[3], …, a[i] in its right place.

The following step essentially defined the insertion sort as applied to sorting in ascending order array A contain N elements.

1- set j = 2.

2- Check if A[j] < A[j-1] if so interchange then
set j ← j-1
until j = 1

3- set j = 3, 4, 5, …, N and keep on executing step2.

```
Begin
      If n >= 2 then begin
                     For i := 2 to n do
                     begin
                           flag := true;
                           j := i ;
                     while (j >= 2) and flag do
                     if a[j] < a[j-1] then begin
                                           t:=a[j];
                                           a[j]:=a[j-1];
```

3.

```
                    a[j-1]:=t;
                    end;
            else  flag := false;
            end;
        end;
end;
```

The efficiency of sorting method:
The insertion sort always better than the bubble sort, the time in both methods is approximately the same. The number of interchanges needed in both method is on the average $n^2/4$ and in the worse case $n^2/2$ where the data is partially ordered, the insertion sort take less time than the bubble. The insertion sort is highly efficient in the array is already in almost sorted order.

Merge sort
This sort is used to merge two ordered list and combine them to produce a single order list .
Ex:-
List1 10 12 24 30 55 80
List2 5 7 14 18 25 ...
New list 5 7 10 12 14 18 24 25 30 ...

Bin sort
Suppose key type is an integer and the values of the keys are known to be in the range 1 to n, with no duplicates where n is the number f elements. Then if A and B are two arrays of size n then n elements to be sorted are initially in A, we can place in B sorted in order of key value by the following loop:

```
For i := 1 to n do
    B[a[i]] := a[i] ;
```

But if we want this method works correctly in one array (origin of array) the following form is used:

```
For i := 1 to n do
While a[i] <> I do
    Swap (a[i] , a[a[i]]);
```

Ex:-use bin sort algorithm to sort the following data

a = {3, 5, 2, 4, 10, 6, 8, 9, 7, 1}

Sol.:- the general form in bin sort algorithm:

```
For i := 1 to n do
While a[i] <> I do
    Swap (a[i] , a[a[i]]);
```

i = 1 a[1] = 3 swap (a(3), a(1))

→ a = {2, 5, 3, 4, 10, 6, 8, 9, 7, 1}

a[1] = 2 swap (a(1), a(5))

→ a = {5, 2, 3, 4, 10, 6, 8, 9, 7, 1}

A[1] = 5 swap (a(1), a(5))

→ a = {10, 2, 3, 4, 5, 6, 8, 9, 7, 1}

A[1] = 10 swap (a(1), a(10))

→ a = {1, 2, 3, 4, 5, 6, 8, 9, 7, 10}

i = 2 a(2) = 2

.

.

.

i = 6 a(6) = 6

i = 7 a(7) = 8 → a = {1, 2, 3, 4, 5, 6, 9, 8, 7, 10}

a[7] = 9 → a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

i = 8 a(8) = 8

i = 9 a(9) = 9

i = 10 a(10) = 10

Radix Sort (Pocket Sort):-

The radix sort is used for numerical data this methode consist of number of passes, in pass(1) we sort on the lower digit of the number in pockets '0' to '9', the pocket '0' on the bottom and pocket '9' at the top.
On pass(2) we sort the higher order digit, by combining the ten pockets same as pass(1) we complete the sort.

Ex:- sort the following data using pocket (radix) sort method:

42 23 74 11 65 87 58 94 36 99

					94					
Pass(1)		11	42	23	74	65	36	87	58	99
	0	1	2	3	4	5	6	7	8	9
Pass(2)		11	23	36	42	58	65	74	87	94
										99

Note:-
the sequential allocation techniques are not practical in representing the pocket since we do not know how many numbers record will occupy a particular pocket. Each pocket is saved by using linked allocation each pocket can represent as linked FIFO queue.

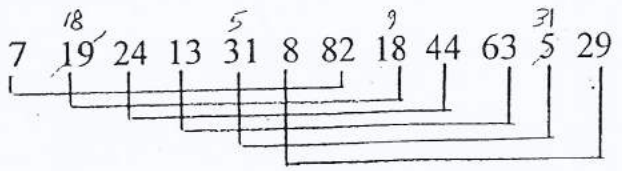
The general algorithm:-

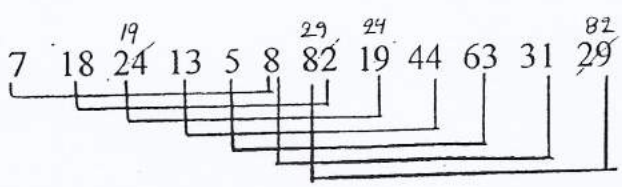
1- repeat through step 6 for each digit in the key.
2- initialize the pocket.
3- repeat through step 5 until the end of linked list.
4- obtain the next digit of the key.
5- insert the record in appropriate place (pocket).
6- combine the pocket to form a new list.

Shell Sort:-

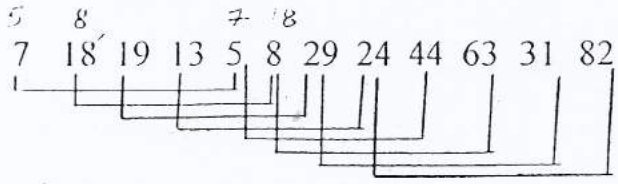
The technique is used by shell sort (named for it's inventor Donald Shell) easy to program and run fairly quickly. Shell sort sorts a list whose entries are intermingled in the whole list.

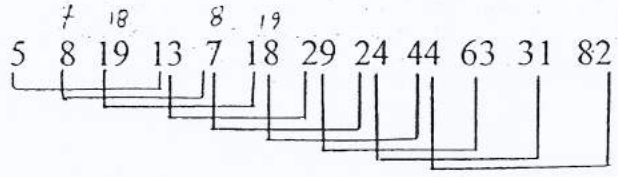
Ex:- Ascending order

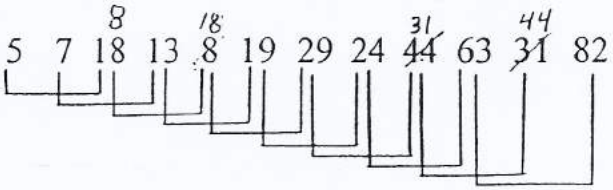


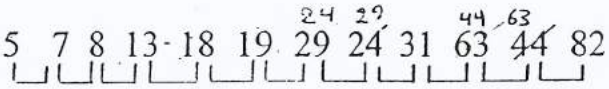


6







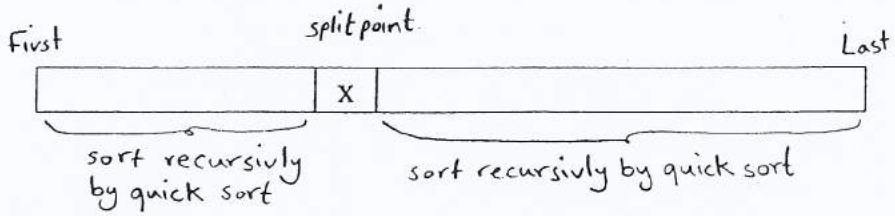


5 7 8 13 18 19 24 29 31 44 63 82

The best choice for h is approximately $1.72\sqrt[3]{n}$ and with this choice the average running time is proportional to $n^{5/3}$.
This increment 1 is the same as doaning with insertion sort.

Partition Exchange Sort (Quick Sort):-

This sort is one of the fasted methods, in this method it divide the list to be sorted into two partitions such that all records in the first partition come before all the records in the partition two. The quick sort procedure is called it self recursively to sort each of two partitions…each time the table of values is subdivided into two sublist one of these is processed with other stored so that it can processed later. A stack is used for this purpose the boundaries of sub table can be stacked while the other is processed.



Ex:- use quick sort algorithm to sort the following values, use first value as a split point.

42 23 74 11 65 58 94 36 99 87

42 23 36 11 65 58 94 74 99 87

11 23 36 42 65 58 94 74 99 87

Ex:- use quick sort algorithm to sort the following data

62 22 58 28 33 74 77 95 18 120 14 181 87 24 36 56 99 71 90

Efficiency of quick sort:-

1- running time of quick sort is depends on the size of data (n) and number of comparison made on each level is depend on number of levels and number of exchange. In the best case and the average there are about log(n) calles to quick sort.
2- There are several other strategies for choosing split point such as choose random value or the medium of L(first), L(first+last/2) and L(last).
3- Quick sort is good for large table but it's not particularly good for small list…this problem can be remedied by choosing small size list and sorting by simple non recursive sort…for example bubble, insertion…

Procedure sequential ~~sort~~ Search
If you have n elements in array a. we search for element item.

```
procedure research ;
begin
      i := n ;
      While (i >=1 ) and (item <> a[i]) do
            i := i-1 ;
      If i > 0 then writeln ('the item is found in location', i)
      Else writeln ('the item is not found');
end;
```

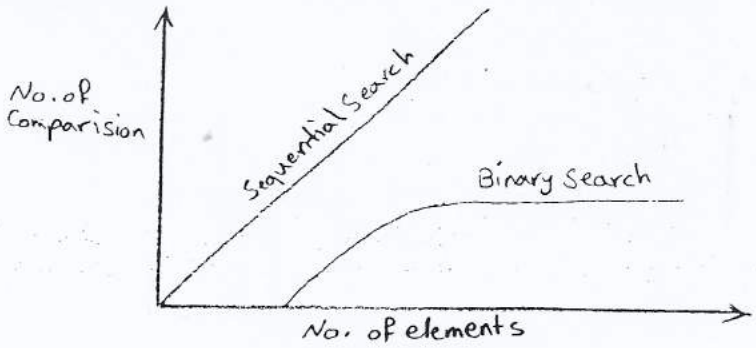
Ex:- a = 5, 10, 3, 8, 20, 34, 6, 18 if item = 20 n = 8
i = 8 → {20 <> 18}
i = 7 → {20 <> 6}
i = 6, i = 5 → {20 = 20} → while give false
i > 0 → found.

Binary searching

Another relatively simple method of accessing a table is the binary search method, the entries in the table are sorted in alphabetically or numerically increasing order.

The algorithm of binary search vector k with n elements for the value x

1- (initialize) low ← 1, high ← n
2- (perform search)
Repeat thru step4 while low≤high
3- (obtain index for midpoint of interval)
mid ← (low+high)/2
4- (compare)
if x < k(mid) then high ← mid-1
else if x > k(mid) then low ← mid+1
else writeln ('successful search')
return (location)
5- (unsuccessful search)
writeln ('not found')
return (0).



Note that the binary search is efficient and fast if n is large, and if n is small the sequential search is better than a binary search.

Hashing

There is a popular class of search method commonly known as hashing.

primary key value $\xrightarrow[\text{function}]{\text{Hash}}$ address

hashing algorithm consist of two components:

1- hashing function which define the mapping.

2- Collision resolution arise when more than one record key is mapped to the same location (address).

There are two methods to solve the conflict:

1- Open addressing.

2- Chain method.