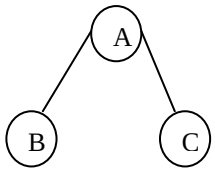
Introduction to Tree

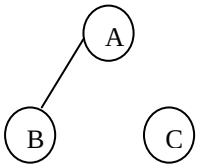
A tree is a connected acyclic graph. An acyclic connected graph where each node has zero or more *children* nodes, and at most one *parent* node. Furthermore, the children of each node have a specific order. The following figure is an example of a tree.



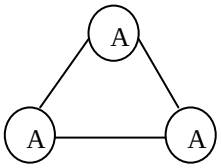
Tree

Connected acyclic graph

Some of the graphs that are not connected acyclic as shown in the following:



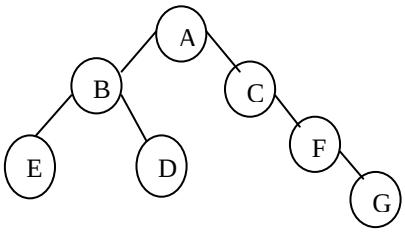
Disconnected graph



Connected cyclic graph

These are not tree.

<u>Properties of tree:</u>



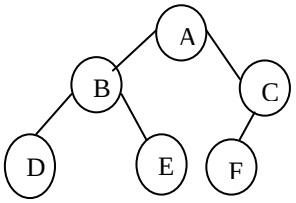
Tree

<u>Children</u>	<u>Father</u>	<u>Grand Father</u>
A->B,C	B,C->A	F->A
B->E,D	E,D->B	
C->F	F->C	

1) A is an ancient of G
2) B, C siblings and E, D Siblings
3) B, C, D, E, F, G descendant of A
4) E, D, G leaf node (no child)

Data Structure for Trees

Tree can be represented by several data structures. The following tree is characterized by two type's data structure such as array and linked list.



Tree

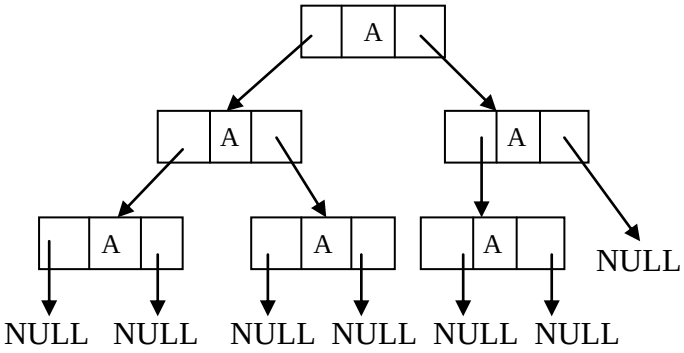
Array representation:

A

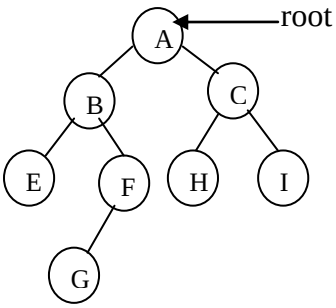
A	B	C	D	E	F
1	2	3	4	5	6

i=1, left =2i=2* 1=2 =>B ;
right =2i+1=2* 1+1=3 =>C
i=2, left =2i=2* 2=4 =>D
right =2i+1=2*2+ 1=5 =>E
i=3, left =2i=2* 3=6 =>F

Linked list representation:



Family Structure of a Tree



root of tree=A
B,C are children of A
E,F are children of B
H,I children of C
G children of F
E, G, H, I have no child (leaf node)
B and C are siblings
E and F are siblings
H and I are siblings
A is grandfather of E, F, H, I
A is grand- grandfather of G
A is ancestor G

Different Types of Trees

There are several types of trees discussed in the following:

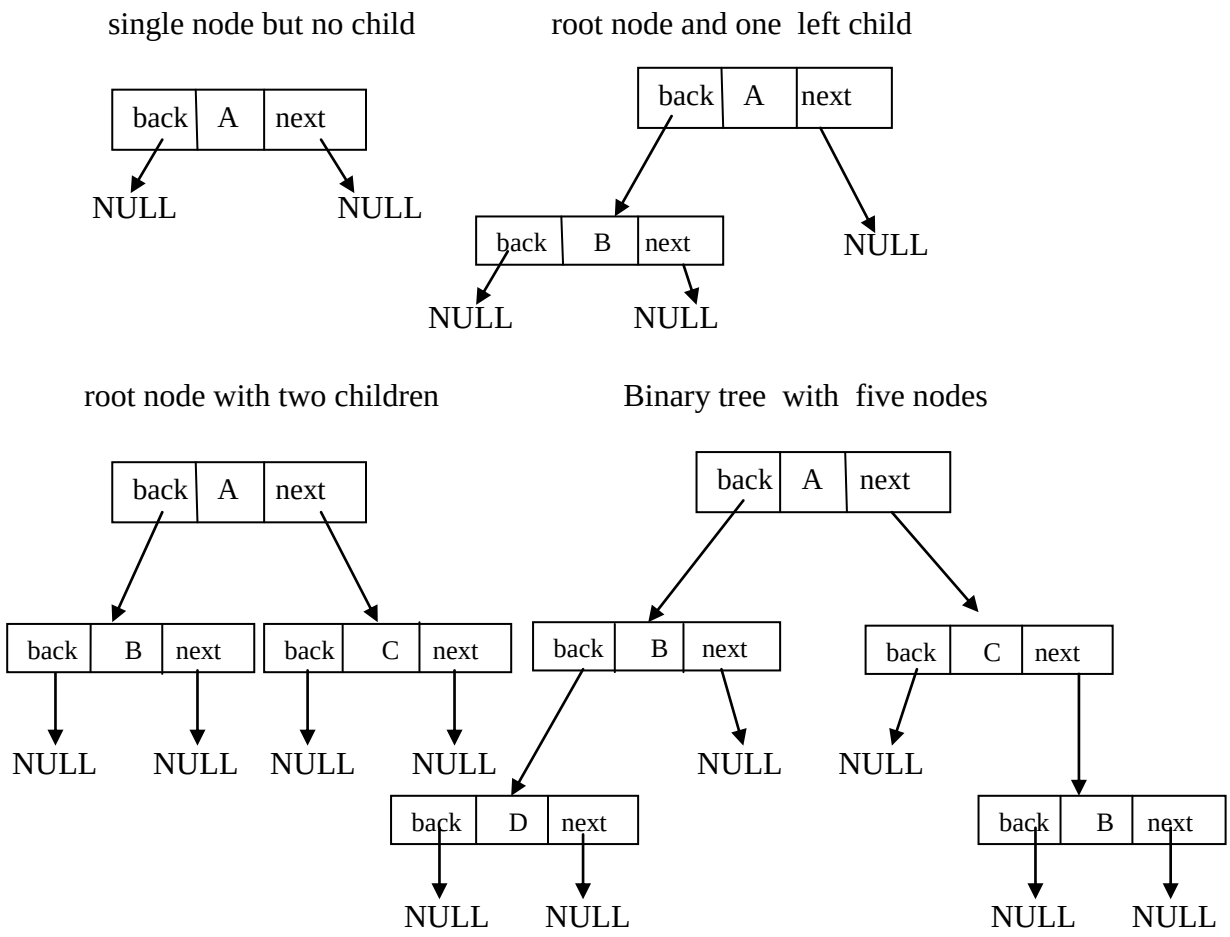
Binary tree
Ternary tree
Full tree
Complete binary tree
Extended binary tree
Forest tree

Binary Tree

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The left and right pointers of topmost node recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

A binary tree is defined as:
(i) A binary tree is either empty (represented by a null pointer) or
(ii)A binary tree is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

A binary tree may have as follows:
(1) Tree (T) has no node (called the null tree of empty tree)
(2) Tree (T) has single node but no child
(3) Tree (T) has root node and one child (left or right)
(4) Tree (T) has at most two sub-trees such as left sub-tree (T_l) and right sub-tree(T_r) [each sub-tree may has zero child, one child or two children and so on]
That means we can say, any node including root of binary tree may have zero node, one node or at most two nodes in the same order of level.



However, another graphical representation of binary tree is as follows?

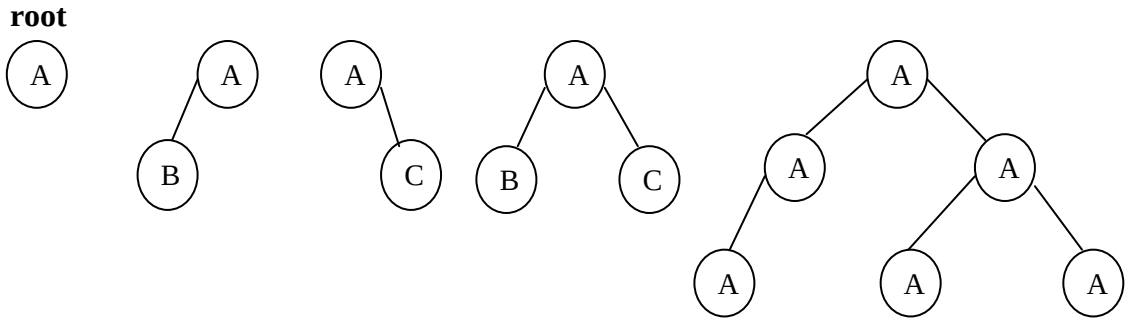


Fig: Binary tree

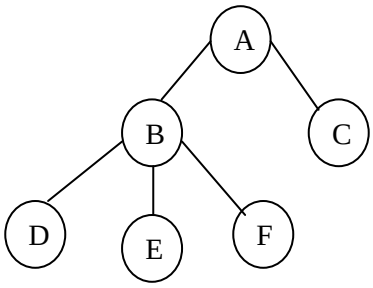
Write an algorithm to create a binary tree

```
void BinaryTree(node *root){
node *l,*r; char ans;
if(root!=NULL){
      printf("\n Do you want to create left child(Y/N):=");
      ans=toupper(getch());
```

```
if(ans=='Y'){
   l=(node*)malloc(sizeof(node));
      printf("Enter left child of %c",root->data);
     l->data=getch();
    l->left=NULL;
    l->right=NULL;
    root->left=l;
   BinaryTree(l);
  }
   printf("\n Do you want to create right child (Y/N):=");
    ans=toupper(getch());
    if(ans=='Y') {
      r->data=getch();
      r->left=NULL;
     r->right=NULL;
     root->right=r;
    BinaryTree(r);
     }
   }
}
```

Ternary Tree

A ternary tree is a tree data structure in which each node has at most three child nodes, usually distinguished as "left", "mid" and "right". Nodes with children are parent nodes, and child nodes may contain references to their parents. Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists. Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left, mid or right child.



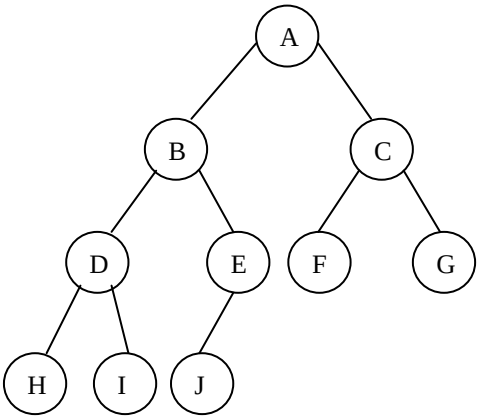
Write an algorithm to create a ternary tree

Complete and Balanced Binary Tree

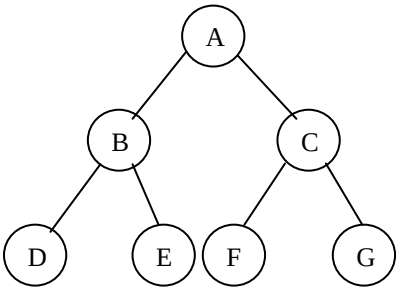
The tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appears as far as possible.

A balanced tree is a tree where every leaf is "not more than a certain distance" away from the root than any other leaf.

In other word, a binary tree is a balance tree, where no leaf is more than a certain amount farther from the root than any other. After inserting or deleting a node, the tree may be rebalanced with "rotations."

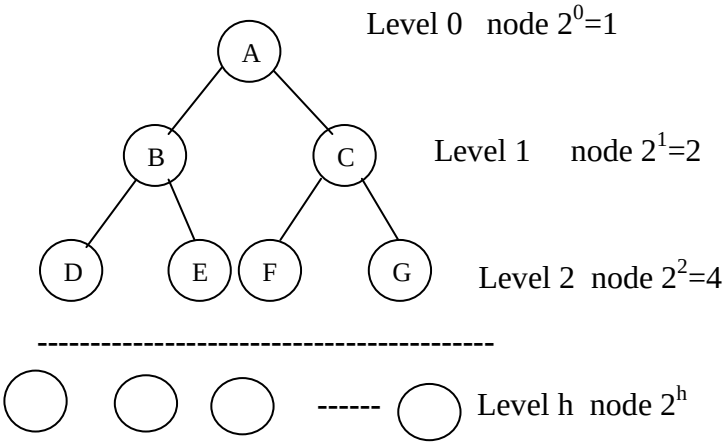


Complete Binary Tree



Balanced Binary Tree

Problem : Prove that the maximum number of nodes in a perfect binary tree is $2^{h+1}-1$.



Total nodes= $1+2+4+\ldots\ldots\ldots+ 2^h= 1(2^{h+1}-1)/(2-1) = 2^{h+1}-1$ (proved)

[$a+ar+ar^2+\ldots..+a^{n-1}=a(r^n-1)/(r-1)$]

Problem: Height of a perfect/balanced binary tree

The number of nodes (n) for height (h) of a perfect binary tree= $2^{h+1}-1$

Therefore, $n= 2^{h+1}-1$

- ⇨ $n+1=2.2^h$
- ⇨ $2^h=(n+1)/2$
- ⇨ $\log_2 2^h=\log_2(n+1)/2$
- ⇨ $h= \log_2(n+1)/2$

Thus the height of a perfect binary with n nodes $h= \log_2(n+1)/2$

$h= \lg(n+1)/2$

Problem: prove that the total number of non-leaf node $=2^h-1$

Total number of non-leaf node =Total number of node – Totasl number opf leaves

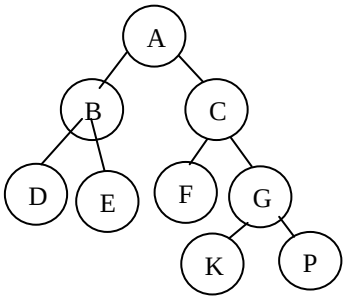
$=2^{h+1}-1 - 2^h$

$= 2^h(2-1)-1$

$= 2^h-1$ (Proved)

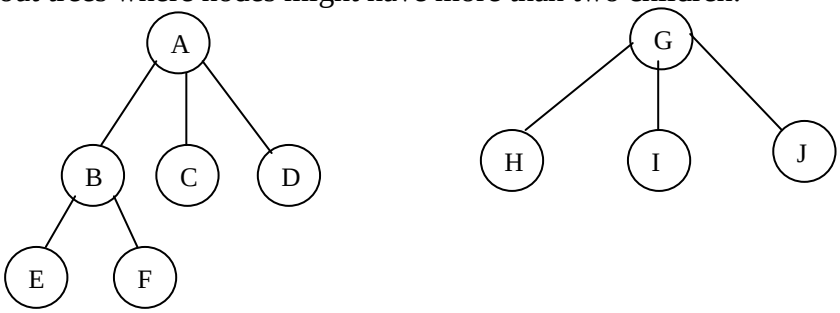
Full binary tree:

Every node has two children except the leaves



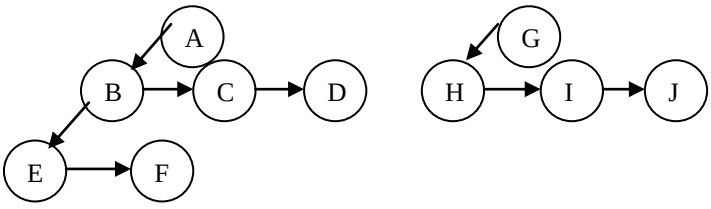
Forest and Natural Correspondence

Forest is a collection of many trees. A general tree as the root of a forest, and a forest is a set of zero or more general trees. This mutually recursive definition of general trees and forests allows us to talk about trees where nodes might have more than two children.

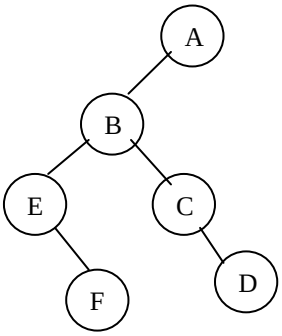


Forest

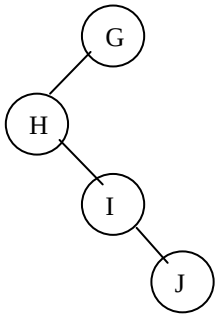
We can convert the above forest into binary tree using natural correspondence in the following way:



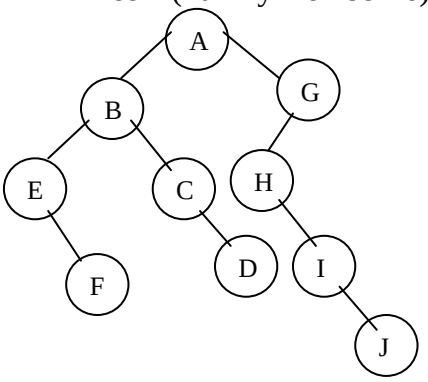
Tree 1



Tree 2



Tree 1 (Family member=6)



Tree 2 (Family member=4)

Tree Traversal Techniques

There are four types of tree traversal technique. They are as follows:

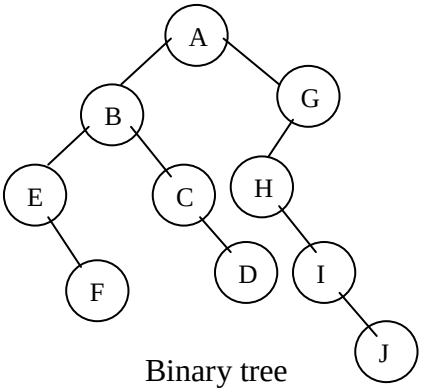
1) Pre-order (root-left-right)

2) In-order (left-root-right)

3) Post-order (left-right-root)

4) Level-order (Level by level)

We can traverse the following binary tree using tree traversal techniques.



Binary tree

1) Pre-order (root-left-right)

A B E F C D G H I J

Pseudo code

```
void preorder(node * root){
  if(root!=NULL){
     printf("%c",root->data);
   if(root->left!=NULL)
     preorder(root->left);
   if(root->right!=NULL)
     preorder(root->right);
   }
}
```

```
void inorder(node * root){
  if(root!=NULL){
   if(root->left!=NULL)
     inorder(root->left);
     printf("%c",root->data);
   if(root->right!=NULL)
     inorder(root->right);
   }
}
```

```
void postorder(node * root){
  if(root!=NULL){
     if(root->left!=NULL)
       postorder (root->left);
     if(root->right!=NULL)
     postorder (root->right);
    printf("%c",root->data);
   }
}
```

2) In-order (left-root-right)
E F B C D A H I J G

3) Post-order (left-right-root)
F E D C B J I H G A

4) Level-order (Level by level)
A B G E C H F D I J

Level order algorithm

```
queue :=root
while(queue!=empty) {
       v:=queue
       printf   v
      if (left->child (v) exists)
          queue := left->child (v)
      if (right->child (v) exists)
          queue := right->child (v)
}
```

Binary Tree Construction from Traversal Sequences

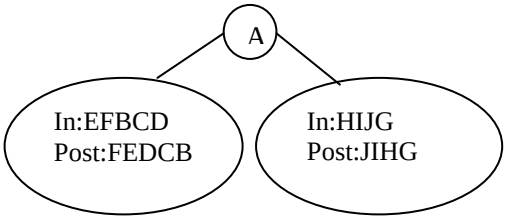
The in-order and post-order node sequences are in the following:

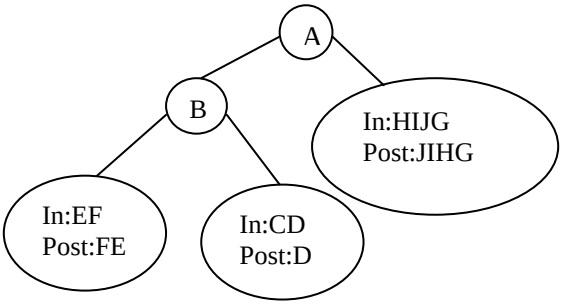
In-order: E F B C D A H I J G

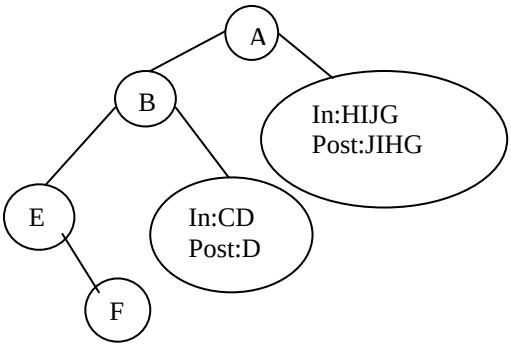
Post-order: F E D C B J I H G A

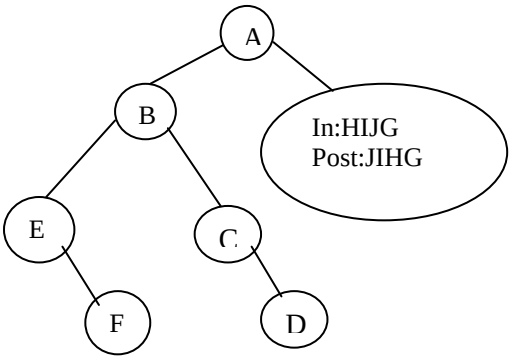
We can construct a binary tree from those traversal sequences in the following way:

We know that in-order and post-order sequences are “left-root-right” and “left-right-root” respectively.

root	left	right	tree
A	In-order: EFBCD Post-order: FEDCB	In-order: HIJG Post-order: JIHG	

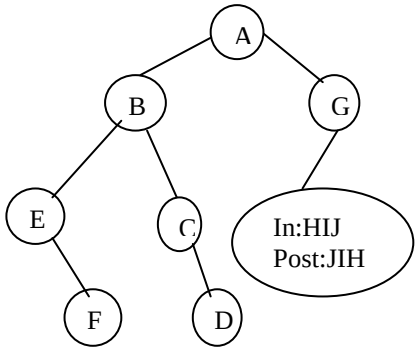
B	In-order: EF Post-order: FE	In-order: CD Post-order: DC	

E	In-order: Post-order:	In-order: F Post-order: F	

C	In-order: Post-order:	In-order: D Post-order: D	

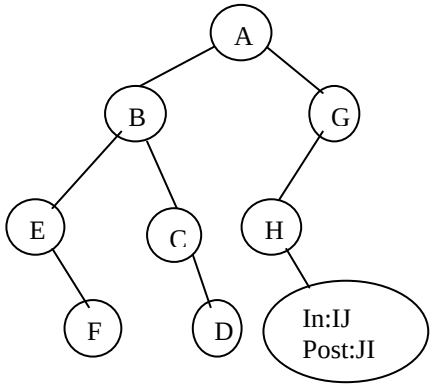
G In-order: HIJ In-order:

Post-order: JIH Post-order:



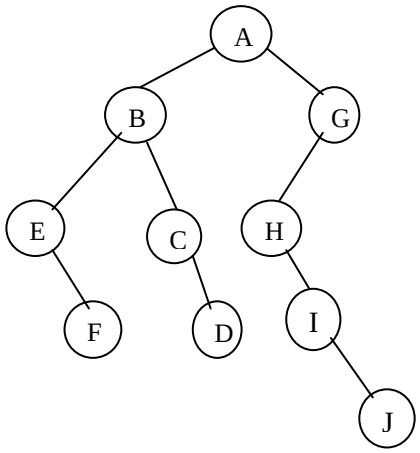
H In-order: In-order: IJ

Post-order: Post-order: JI



I In-order: In-order: J

Post-order: Post-order: J

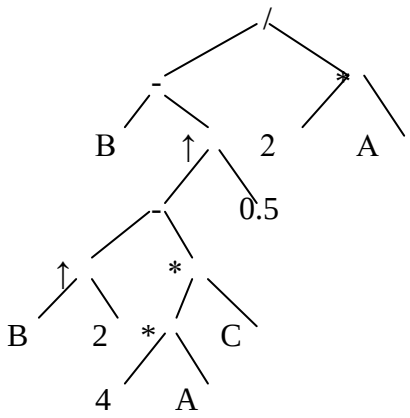


Binary Expression Tree

A binary expression tree is a specific application of a binary tree to evaluate certain expressions. In general, expression trees are a special kind of binary tree. A binary tree is a tree in which all nodes contain zero, one or two children. This restricted structure simplifies the programmatic processing of Expression trees. Binary expression trees can represent **expressions** that contain both unary and **binary** operators. We can take an arithmetic expression and write it in the obvious way as a binary tree, and then the post order traversal of that tree produces the postfix form of the arithmetic expression.

For example, consider the expression (B-(B↑2-4*A*C) ↑ 0.5) / (2*A)

As binary expression tree we would write it as fig ( ):



A postorder traversal of this tree would visit the node in the order BB2↑4A*C*-0.5↑-2A*/, exactly the postfix of the expression considered. Similarly the preorder traversal of such a tree corresponds to prefix form of the expression, while the inorder traversal corresponds to the normal infix form with the parenthesis deleted.

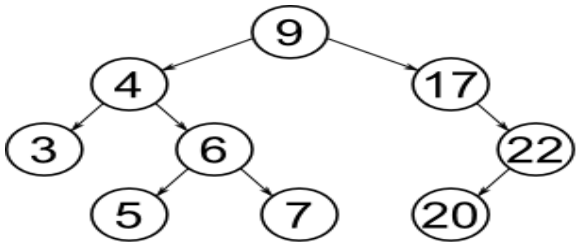
Quantative Aspect of Trees

Binary Search Tree Construction

A binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node.

The number of BSTs possible with n nodes is given by the Catalan Number

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

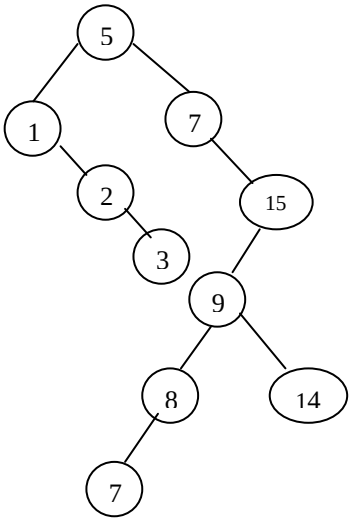


The BST data structure is the basis for a number of highly efficient sorting and searching algorithms, and it can be used to construct more abstract data structures including sets, multi-sets, and associative arrays.

Suppose we have an array consisting data are as follows:

5	7	1	15	9	2	14	8	7	3
1	2	3	4	5	6	7	8	9	10

The binary search tree is as follows:



Binary search tree

Binary search tree construction:

Let an array A [ ] consisting 4 elements: A

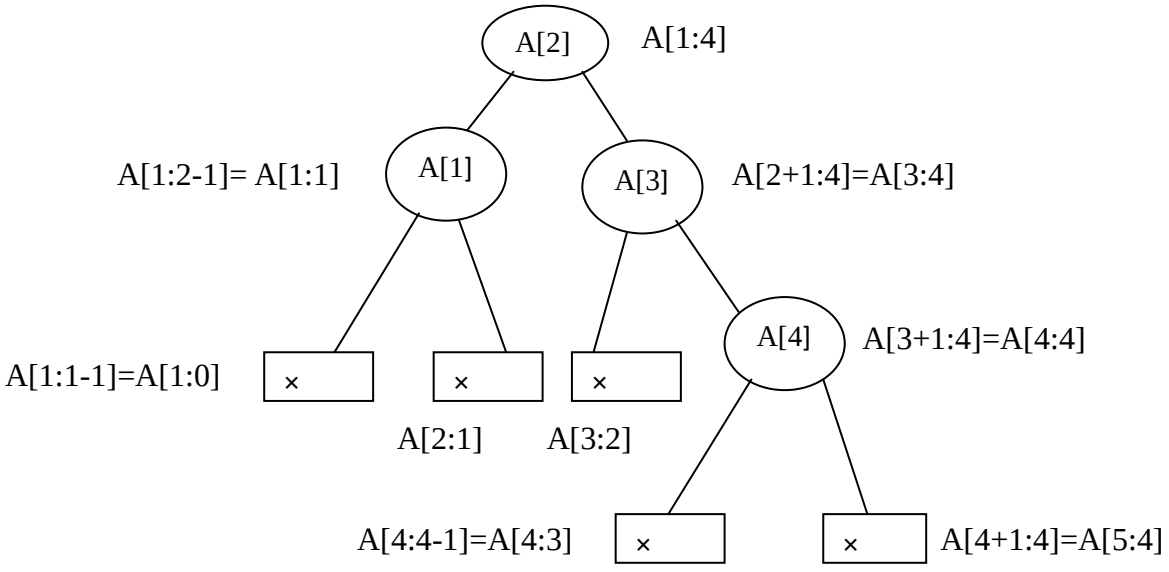
10	20	30	40
1	2	3	4

Found element represents circular node

Not found element represents square node

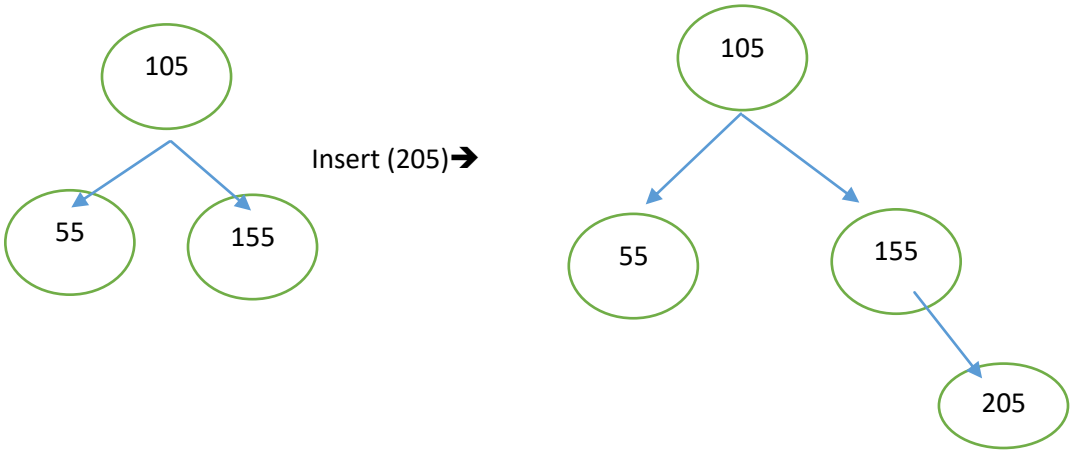
First element location+ Last element= ⌊1+4⌋ =2

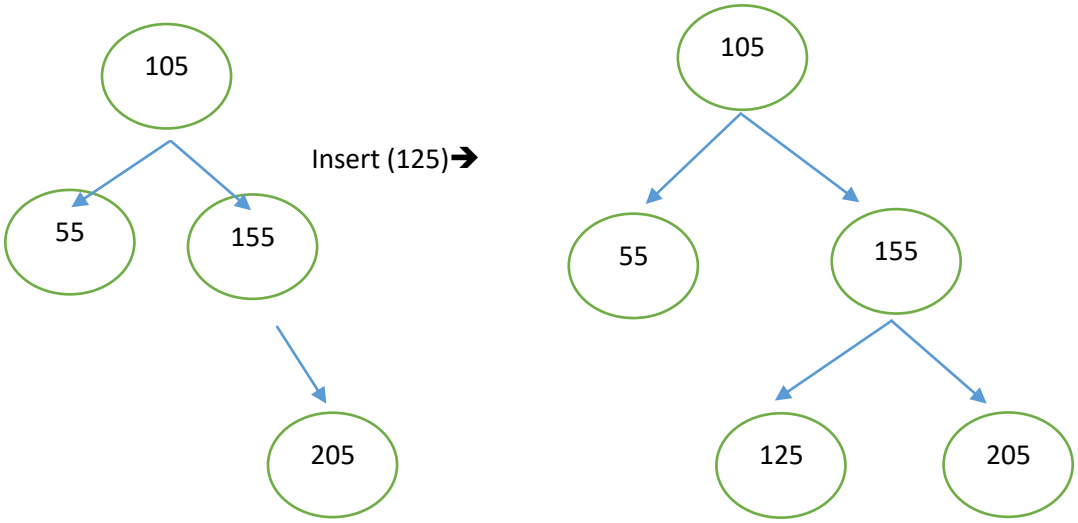
So the root element location is A[2].



The internal and external nodes are as follows of the tree:
Total Internal nodes, I(n)=4
Total external nodes, E(n)=5
Therefore, E (n) =I (n) +1
Hence Total external nodes= Total Internal nodes+1

Inserting a node in a binary search tree

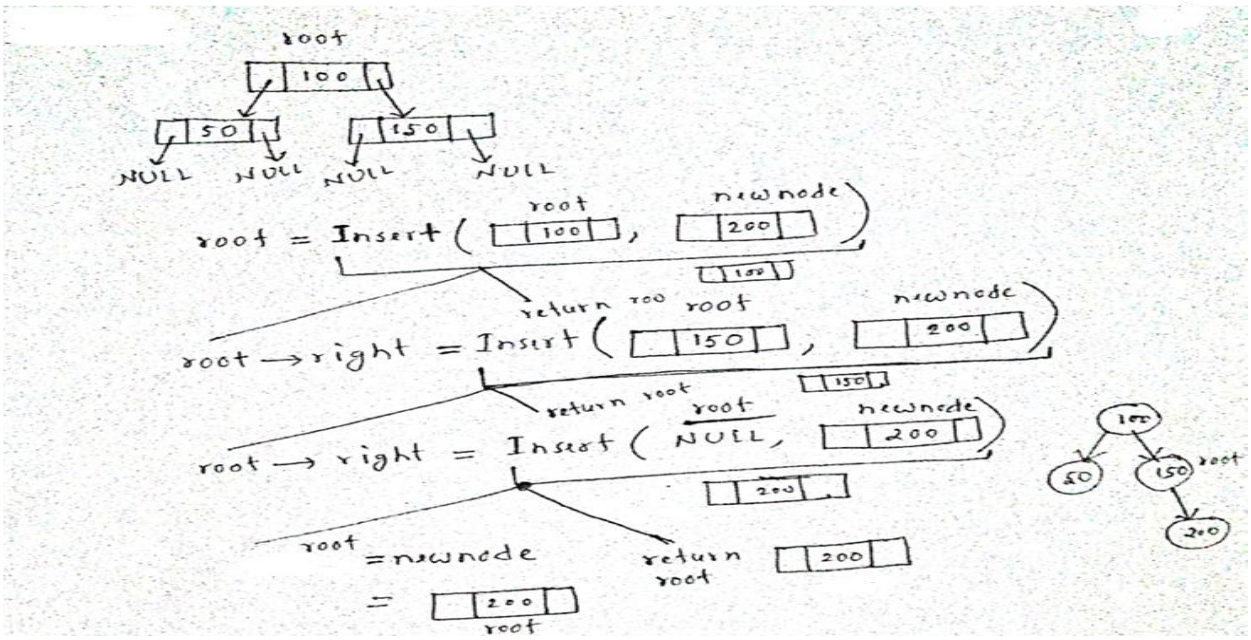




Recursive algorithm:

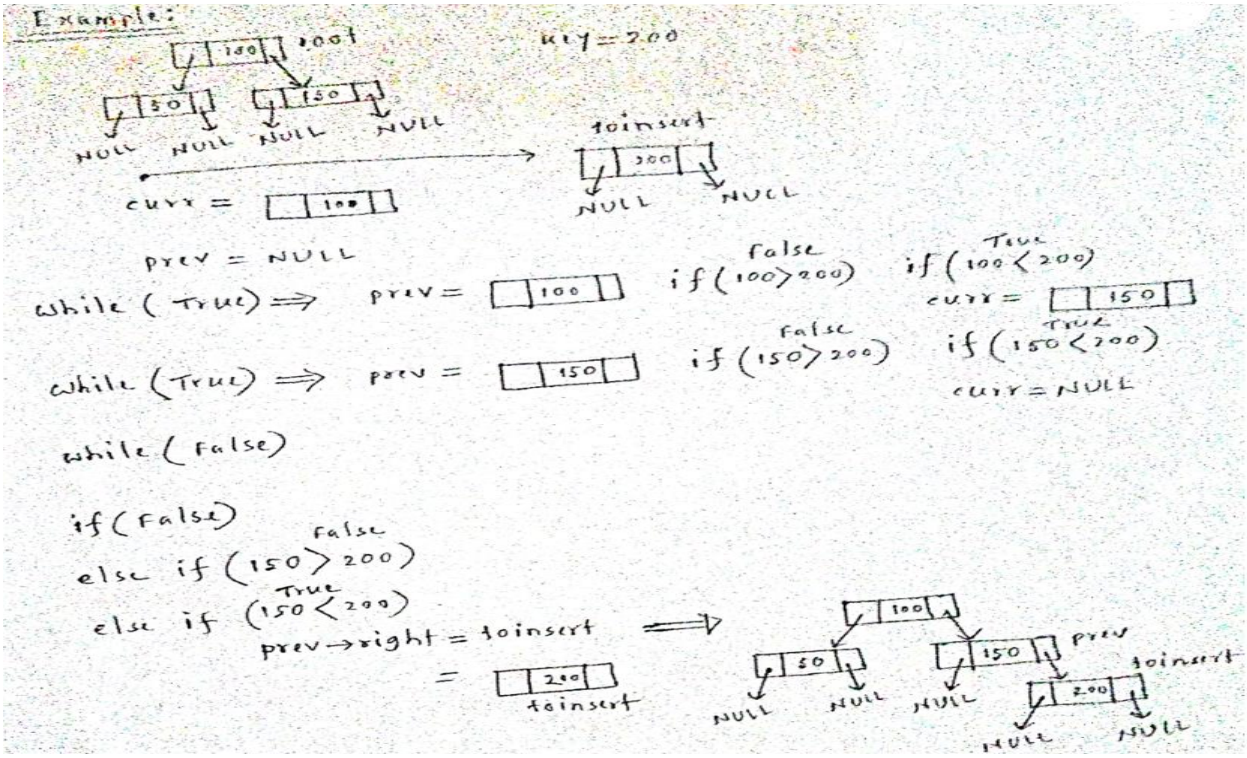
```
node *insert (node  *root, node  *newnode) {
 if( root== NULL)
     root=newnode;
else if (root->data<newnode->data)
    root->right=insert(root->right, newnode);
else if (root->data>newnode->data)
     root->left=insert(root->left, newnode);
return root;
}
```

Recursion tree:



Iterative Algorithm

```
node *insert( node  *root, int   key) {
node         *toinsert, *curr, *prev;
toinsert=(node*)malloc(sizeof(node));
toinsert->data=value;
toinsert->left=NULL;
toinsert->right=NULL;
curr=root;
prev=NULL;
while (cuur!=-=NULL){
prev=curr;
if(curr->data>value)
      curr=curr->left;
else
        curr=curr->right;
}
if (prev==NULL)
     root=toinsert;
else if(prev->data>value)
     prev->left=toinsert;
else
    prev->right=toinsert;
  return root;
}
```

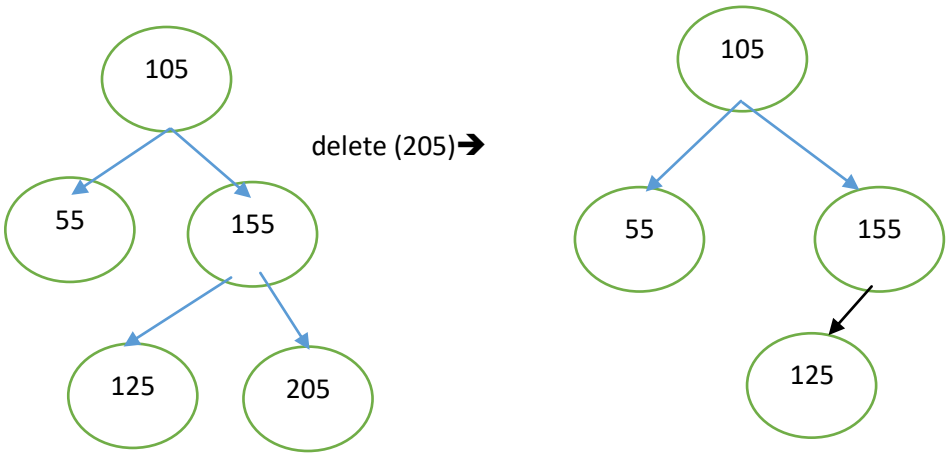


Deletion form Binary search tree

1. Leaf node:

   If the node is leaf remove the node directly and free its memory.

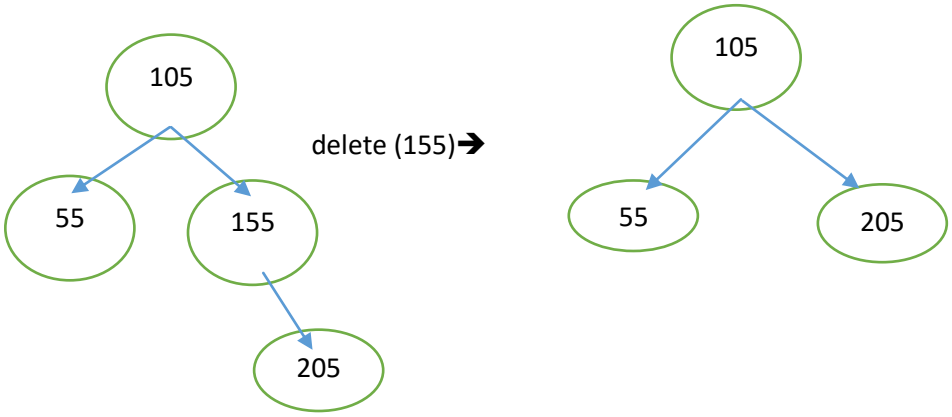
   Example:



2. Node with right child:

If the node has only right child, make the node points to the right node and free the node.

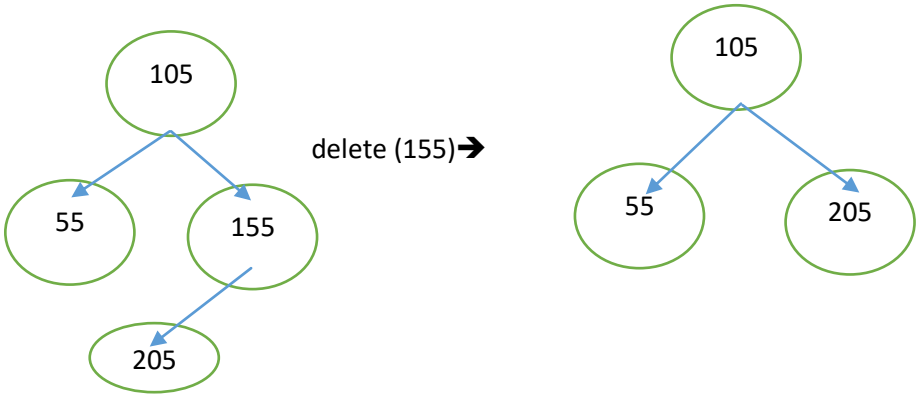
Example:



3. Node with left child:

If the node has only left child, make the node points to the left node and free the node.

Example:



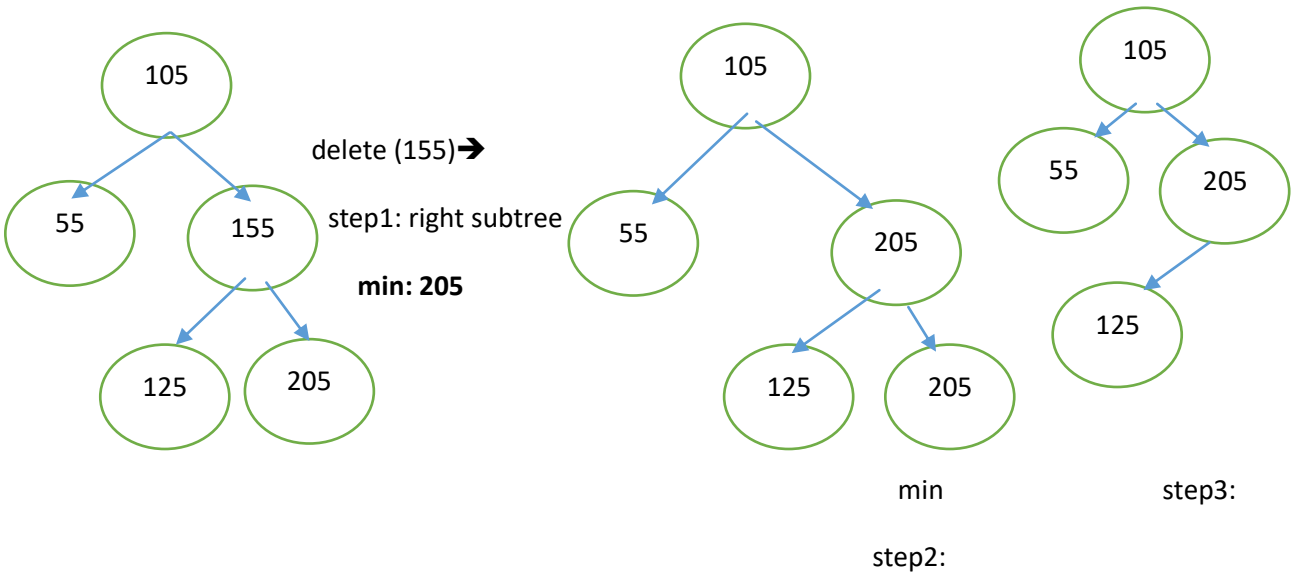
4. Node has both left and right child:

i) Find the smallest node from the right subtree

ii) Make node ->data= min

iii) Delete the min node

Example:



Deletion Function

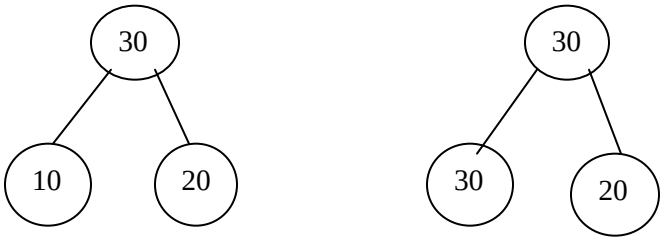
```
node   *delete ( node *root,  int     val) {
    if (root==NULL) {return NULL;}
    if (root->data<val)
       root->right=delete(root->right, val)
else if (root->data>val)
      root->left=delete(root->left, val)
else{
    if (root->left==NULL && root->right==NULL){
         free(root);
         return NULL;
      }
     else if(root->left==NULL){
           node *temp=root->right; free(root); return temp; }
     else if(root->right==NULL){
           node *temp=root->left; free(root); return temp; }
      else {
           int rightMin=getRightMin(root->right)
           root->data=rightMin;
```

```
            root->right=delete( root->right, rightMin)
return root;
 }
int   getRightMin (node *root){
   node *temp =root;
     while(temp->left!=NULL)
             temp=temp->left;
     return temp->data;
}
```

Heap data structure and Heap sort:

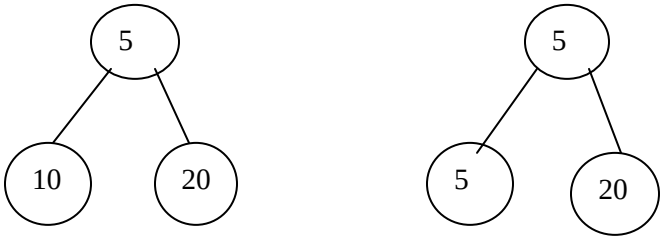
Heapsort: Heapsort is an in-place algorithm. ,but is not a stable sort. Heapsort was invented by J.W.J. Williams in 1964. Heap sort is an elegant sorting algorithm.

Classification of heapsort:
->Max heap
->Min heap
Max heap tree:



For max heap parent>=child

Min heap tree:

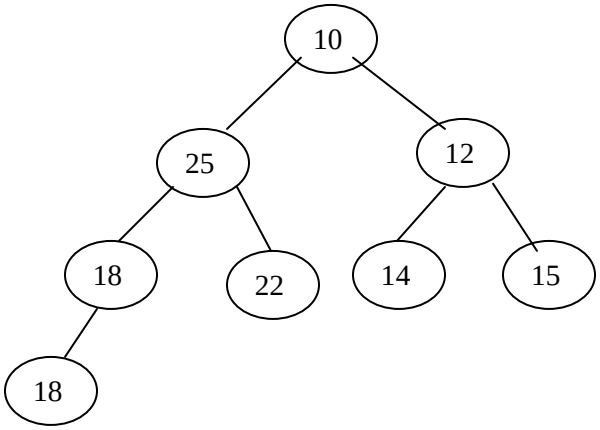


For Min heap parent<=child

Max heap construction:

A

10	25	12	18	22	14	15	18
1	2	3	4	5	6	7	8



$i=\lfloor n/2 \rfloor=4$

Therefore item=A[4]=18

Compare Item->left and item->right

Item->left is larger because of item->right is null

Compare Item->left and Item

if (item<item->left)

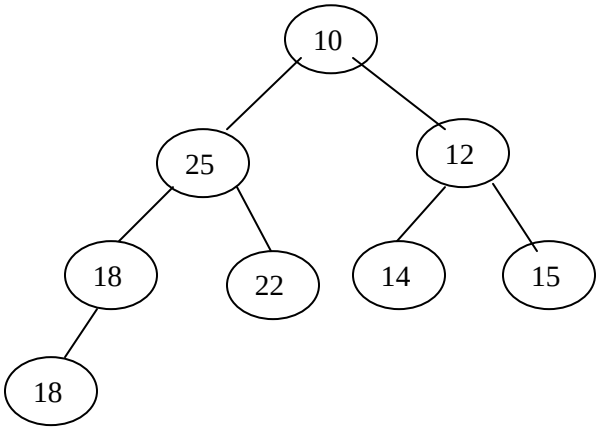
Item->left <---> Item

else

no interchange

A

10	25	12	18	22	14	15	18
1	2	3	4	5	6	7	8

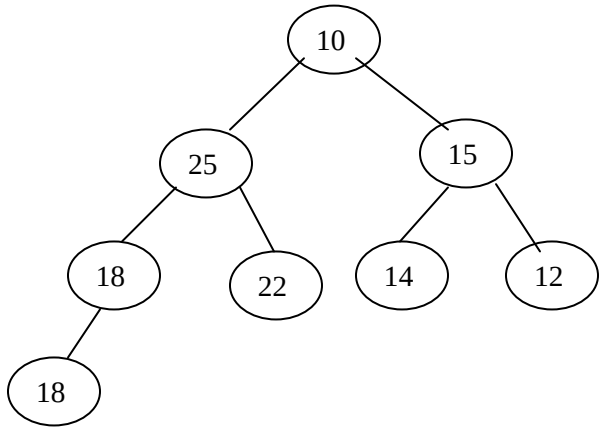


i=i-1=4-1=3

Therefore item=A[3]=12

Compare Item->left and item->right
Item->right is larger than item->left
Compare Item->right and Item
if (item<item->right)
Item->right <---> Item
else
no interchange

A	10	25	15	18	22	14	12	18
	1	2	3	4	5	6	7	8



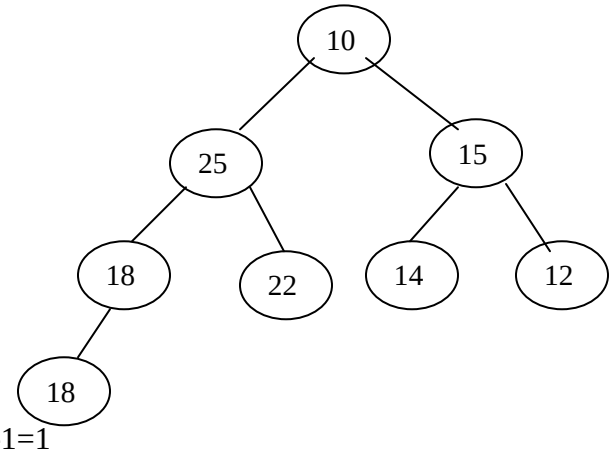
i=i-1=3-1=2

Therefore item=A[2]=25

Compare Item->left and item->right
Item->right is larger than item->left
Compare Item->right and Item
if (item<item->right)
Item->right <---> Item
else
no interchange

A

10	25	15	18	22	14	12	18
1	2	3	4	5	6	7	8

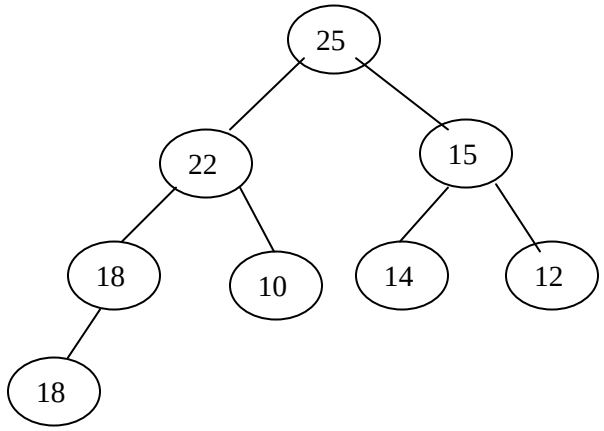


i=i-1=2-1=1

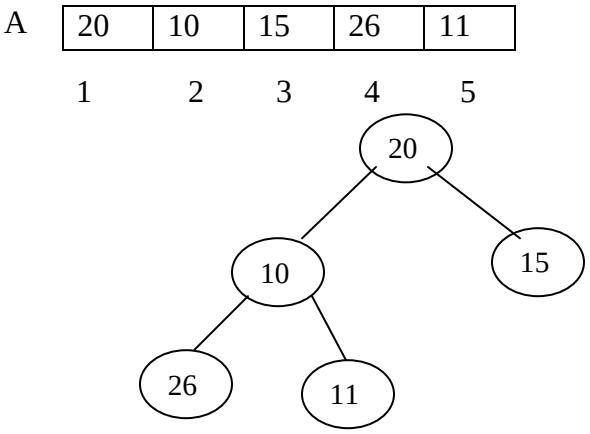
Therefore item=A[1]=10

Compare Item->left and item->right
Item->left is larger than item->right
Compare Item->left and Item
if (item<item->left)
Item->left <---> Item
else
no interchange

25	22	15	18	10	14	12	18
1	2	3	4	5	6	7	8

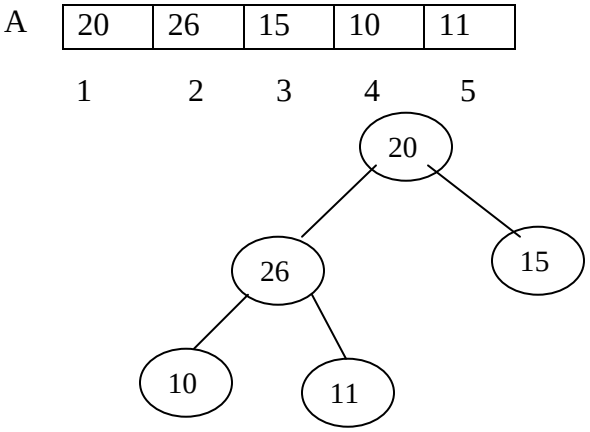


Heap sort Mechanism:



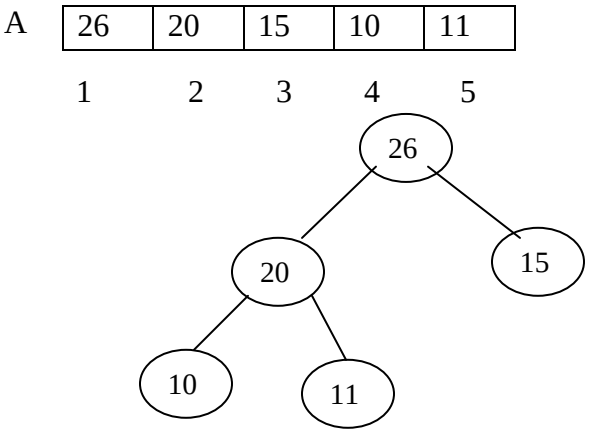
i=⌊n/2⌋=2

Item=A[2]=10



i=i-1= 1

item=A[1]=20

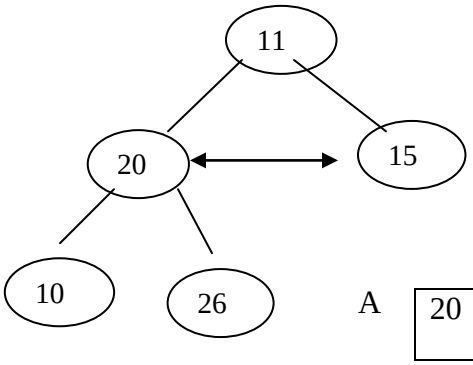


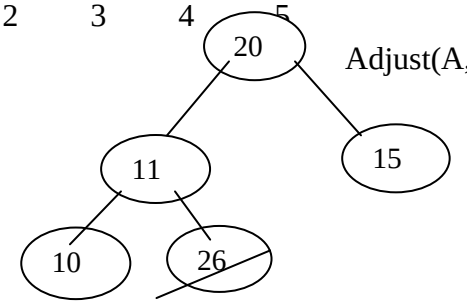
when i=5
A[i]<----->A[1]

A

11	20	15	10	26

1 2 3 4 5





Adjust(A,1,i-1)

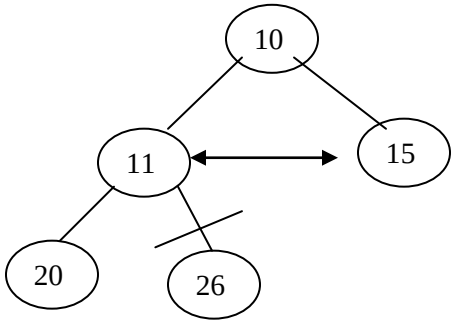
A

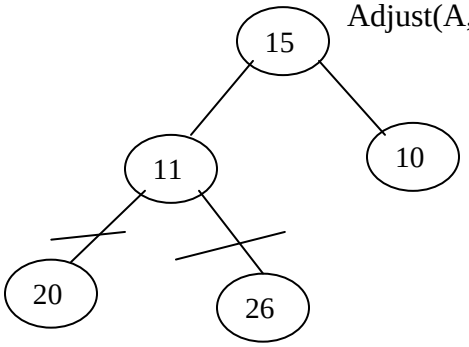
20	11	15	10	**26**

1 2 3 4 5

when i=4
A[i]<----->A[1]

10	11	15	20	**26**





Adjust(A,1,i-1)

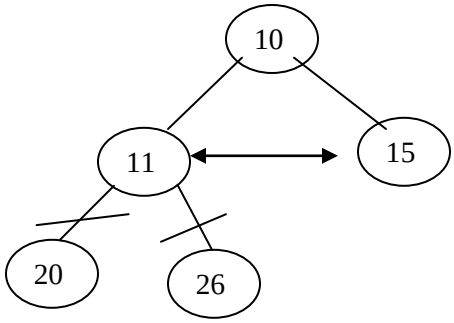
A

15	11	10	**20**	**26**

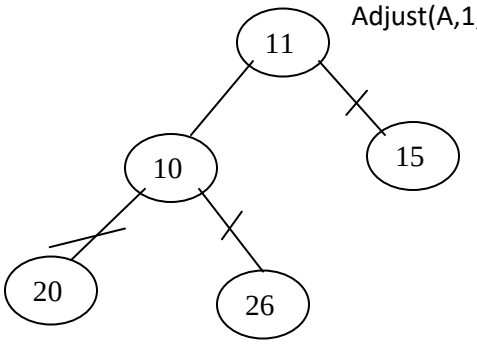
when i=3
A[i]<---->A[1]

A

10	11	15	20	26



Adjust(A,1,i-1)



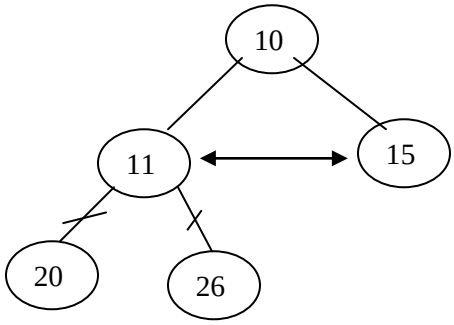
A

11	10	**15**	**20**	**26**

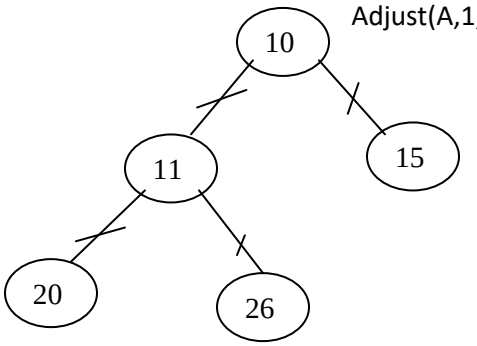
when i=2
A[i]<---->A[1]

A

10	11	15	20	26



Adjust(A,1,i-1)



A

10	11	15	20	26

Algorithm of Heap sort:

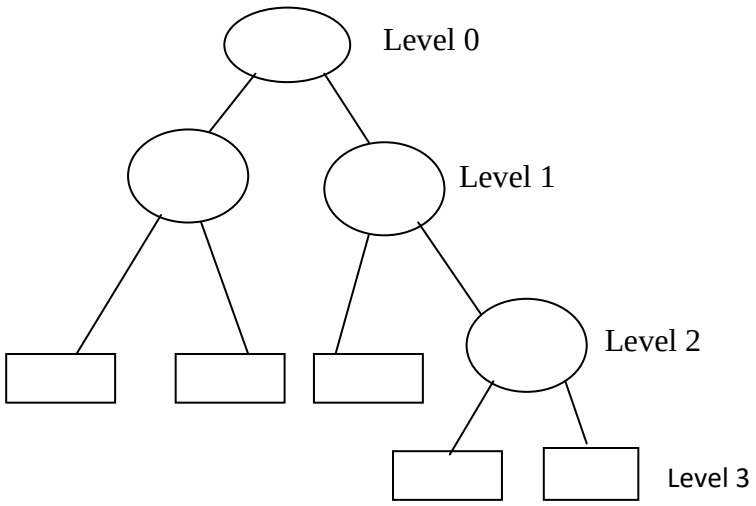
step1: Construct a binary tree with given list of elements
step2: Transform the binary tree into Max heap
step3: Delete the root element from Max heap using Heapify method
step4: Put the deleted element into the sorted list
step5: Repeat the same until the Max-heap becomes empty
step6: Display the sorted list

Algorithm of Heap sort:

```
procedure heapsort (A,n)
    for (i=⌊n/2⌋) to down to 1
        adjust (A, i, n)
    end for
    for (i=n down to 2)
      A[1]<--→A[i]
      adjust (A, 1,i)    // for heapify
    end for
end procedure ( )
procedure adjust(A,i,n)
  j=2*i; item=A[i];
  while ( j<=n)
    if ((j<n) and (A[j]<A[j+1]))
        j=j++;
    end if
    if (item >=A[j])
       exit loop;
    end if
    A[j/2]=A[j];
     j=2*j;
   end while
 A[j/2]=item;
end procedure ( )
```

Internal and External Path Length Determination

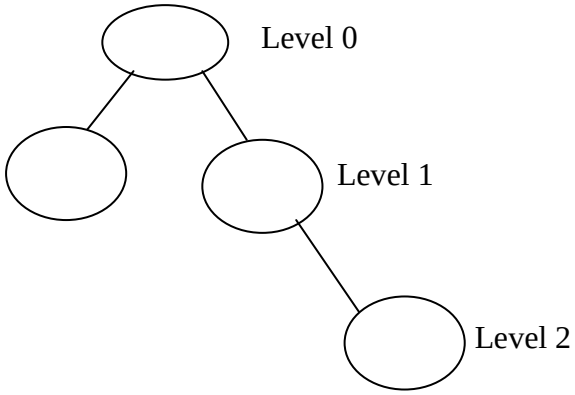
The internal and external path lengths of this tree are as follows:



Internal path lengths I(T)=0+1+1+2 =4
External path lengths E(T)=2+2+2+3+3 =12
Relation between external and internal path lengths:
E(T)-E(I)=12-4
=8
=2*4
=2*circular node
Hence, E(T)-E(I)= 2*circular node
Internal path length is the summation of level of internal nodes, whereas external path length is the summation of level of external nodes.

Height Calculation

There are several methods to calculate the height of a tree. They are shown as follows:



Method 1: (Level oriented)
Height=max(Level i)
=max(level 0,level 1,level 2)
=2
Method 2: (Formula oriented)

Height=$\lfloor \log_2 n \rfloor$

=$\lfloor \log_2 4 \rfloor$

=$\lfloor 2\log_2 2 \rfloor$

=2*1
=2
If n=5

Height=$\lfloor \log_2 n \rfloor$

=$\lfloor \log_2 5 \rfloor$

=$\lfloor \log_e 5/\log_e 2 \rfloor$=$\lfloor$ 1.609/0.693$\rfloor$ =$\lfloor$ 2.321$\rfloor$ =2

Method 3: (Recursive)

Height=max(height(T_l),height(T_r))+1

=max(0, max(height(T_{rl}),height(T_{rr}))+1)+1

= max(0, max(-1,0)+1)+1

= max(0,0+1)+1

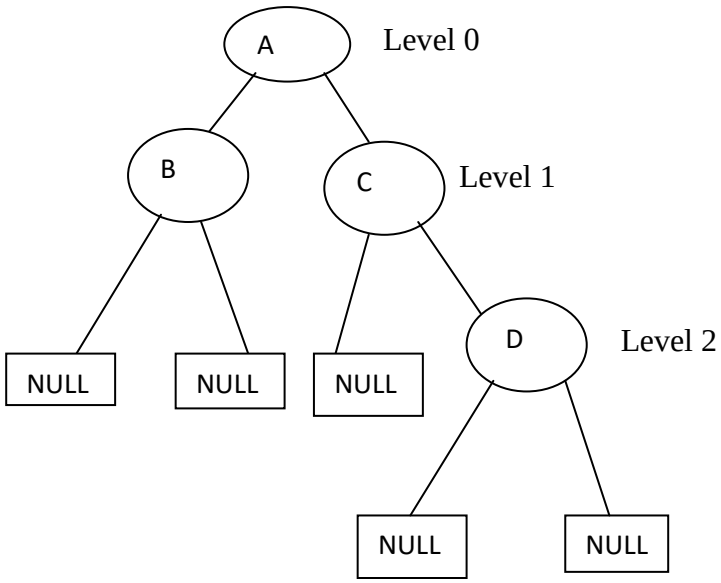
= max(0,1)+1= 1+1=2

Design an algorithm to find height of a binary tree

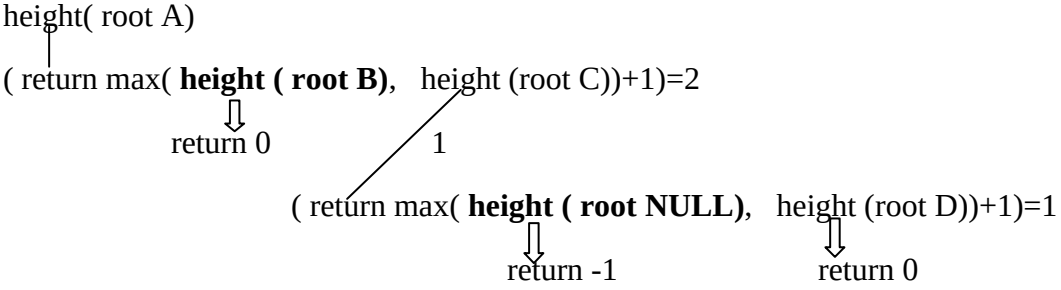
```
int height (node* root) {
        if(root==NULL)
                return -1;
        else if  ((root->left==NULL) &&(root->right==NULL))
                return 0;
         else
               return max(height(root->left),height(root->right))+1;
        end if
}
int max(int x, int y) {
   if(x>y)
        return x;
   else
       return y;
end if
}
```

Recursion tree for height calculation:

The internal and external path lengths of this tree are as follows:

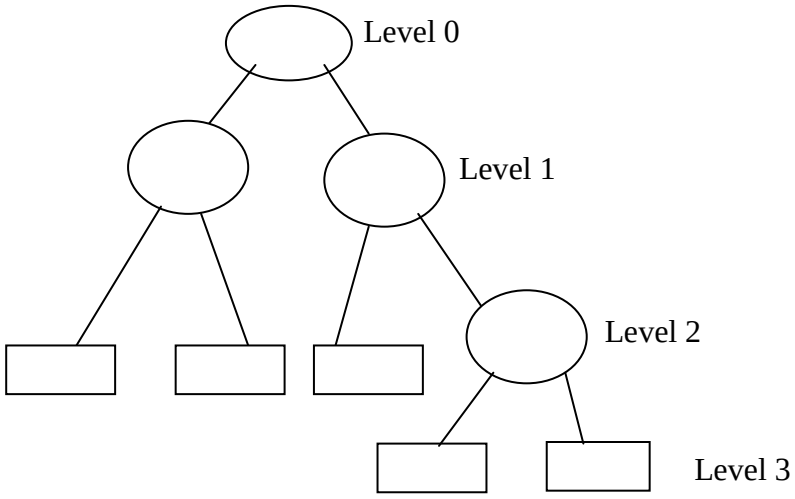


Recursion tree



Successful and Unsuccessful Comparisons

The average number of successful and unsuccessful comparison S(n) and U(n) are as follows for the following tree.



Average number of successful comparisons $S(n)= \sum (l_i+1) /n$

$=\{(0+1)+(1+1)+(1+1)+(2+1)\}/4$

$=8/4$

$=2$

Average number of unsuccessful comparisons $U(n)= \sum (l_i) /(n+1)$

$=(2+2+2+3+3)/5$

$=12/5$

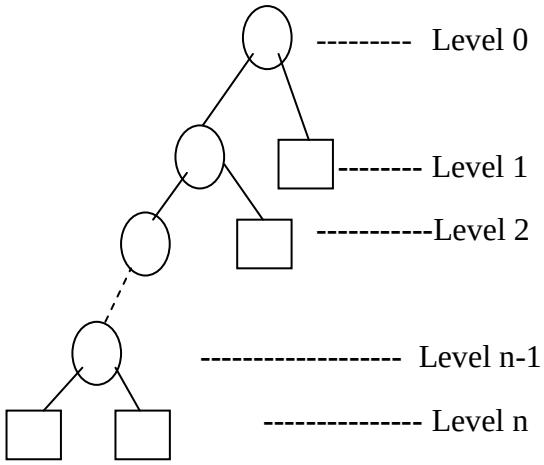
$=2.4$

One Sided Binary Search Tree

Definition: Meta Binary Search, also known as One-Sided Binary Search, is a variation of the binary search algorithm that is used to search an ordered list or array of elements. This algorithm is designed to reduce the number of comparisons needed to search the list for a given element.

The relation between external and internal path length for one sided binary search tree is as shown in the following:

Let the tree is as follows:



Internal path length ,I(T)=0+1+2+-----+(n-1)

=(n-1)(n-1+1)/2

=n(n-1)/2 ----------(i)

External path length ,E(T)=1+2+-----+n+n

=n(n+1)/2 + n

=n(n+3)/2 ----------(ii)

We have equ(ii)-equ(i)=> E(T) - I(T)= n(n+3)/2 - n(n-1)/2

$=n^2/2+3n/2- n^2/2+n/2$

=3n/2+n/2

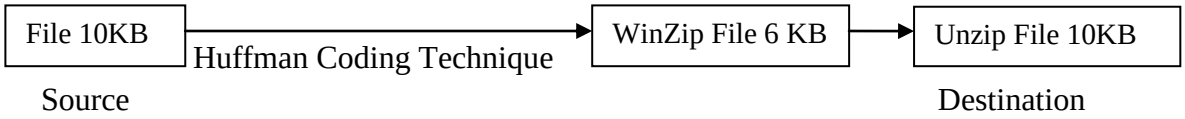
=4n/2

E(T) - I(T)=2n

Huffman Coding

A Huffman code is an optimal prefix code found using the algorithm developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes". The process of finding and/or using such a code is called Huffman coding and is a common technique in entropy encoding, including in lossless data compression. The algorithm's output can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). Huffman's algorithm derives this table based on the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in linear time to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

Huffman coding:



The example of Huffman coding is as follows:

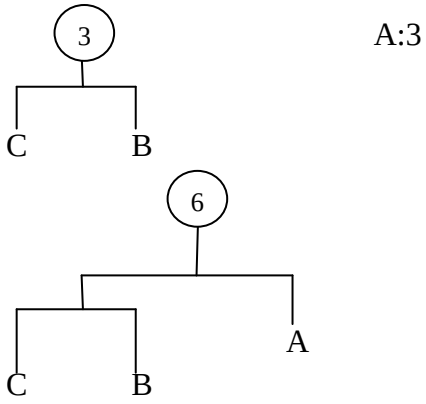
Let, the word ABBACA is 6 bytes as source file need to compressed for sending to the destination.

Step1: Frequency count:

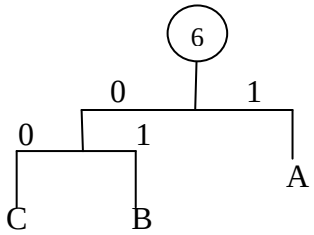
Character	count
A	3
B	2
C	1

Step2: Huffman tree construction:

Priority Queue: C:1 B:2 A:3



Step3: Code generation:



Characters	Code
A	1
B	01

C 00

Step4: Code replacement:

A B B A C A

1 01 01 1 00 1

Step5: code compression

10101100 1

172 1

¬ SOH

Therefore, compressed code: ¬SOH

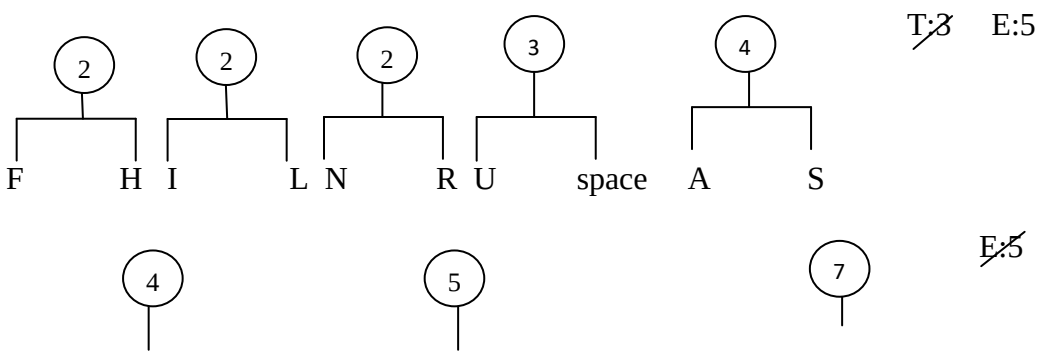
Example: THE ESSENTIAL FEATURE

Step1: Frequency count:

Character	count
T	3
H	1
E	5
S	2
N	1
I	1
A	2
L	1
F	1
U	1
R	1
space	2

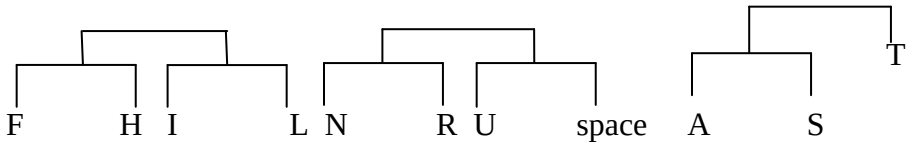
Step2: Huffman tree construction:

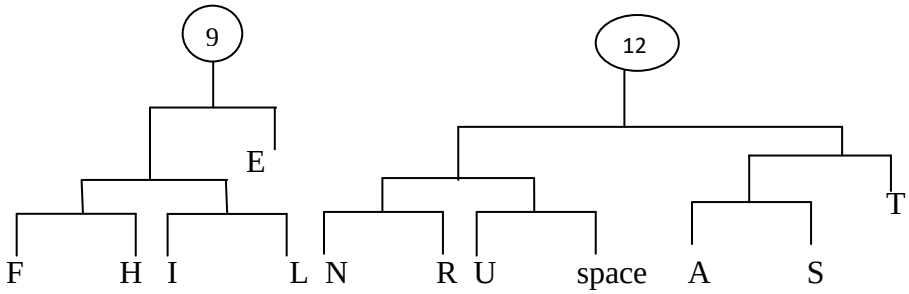
Priority Queue: ~~H:1~~ ~~N:1~~ ~~I:1~~ ~~L:1~~ ~~F:1~~ ~~U:1~~ ~~R:1~~ ~~S:2~~ ~~A:2~~ ~~space:2~~ T:3 E:5

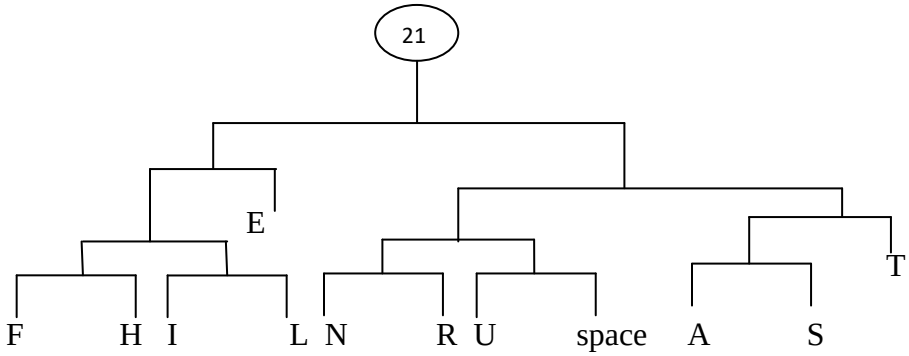


~~T:3~~ E:5

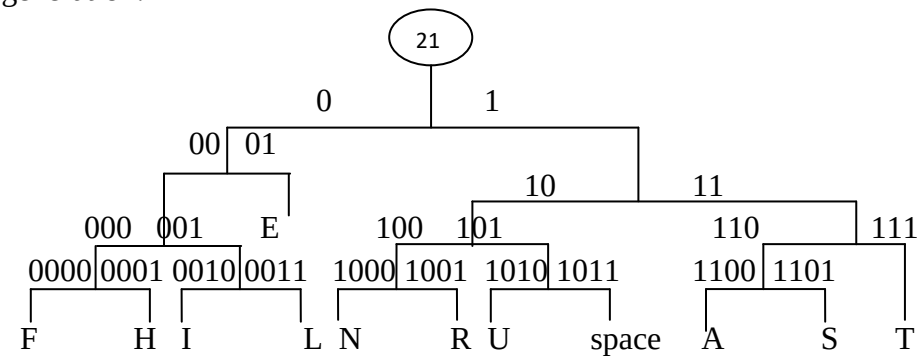
~~E:5~~







Step3: Code generation:



Characters	Code
A	1100
E	01
F	0000
H	0001
I	0010

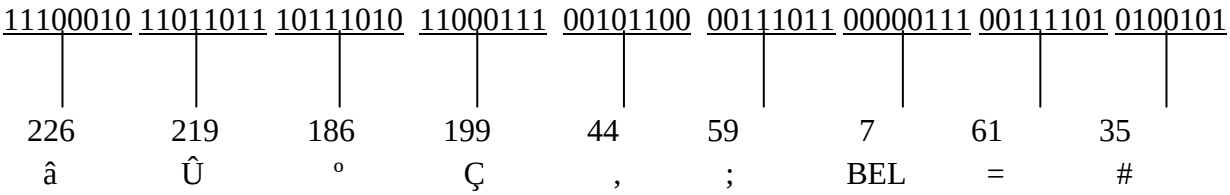
L	0011
N	1000
R	1001
S	1101
T	111
U	1010
space	1011

Step4: Code replacement:

T H E E S S E N T I A L F E A T U R E

11100010110110111011101011000111001011000011101100000111001111010100101

Step5: code compression



Therefore, compressed code: âÛºÇ,;BEL=#