SWE 123

Discrete Mathematics

Horipriya Das Arpita

2019831068

Software Engineering

Shahjalal University of Science and Technology

Tree

What is Tree?

A tree is a connected undirected graph with no simple circuits.

- A graph is said to be circuit-free if, and only if, it has no circuits.
- A graph is called a tree if, and only if, it is circuit-free and connected.
- A trivial tree is a graph that consists of a single vertex.
- A graph is called a forest if, and only if, it is circuit-free and not connected.

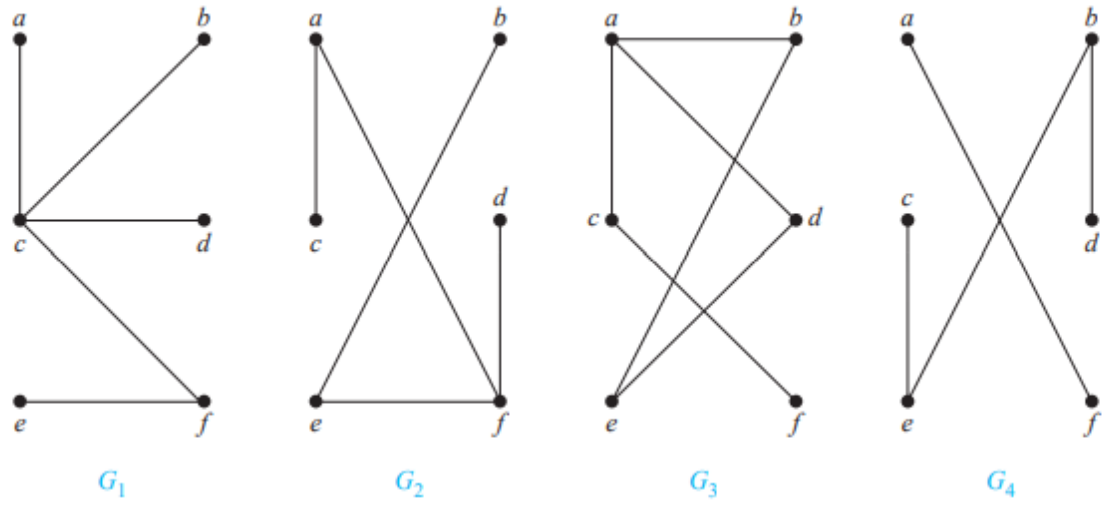


Figure: 1

Which of the graphs are trees shown in Figure 1?

Solution:

G1 and G2 are trees, because both are connected graphs with no simple circuits. G3 is not a tree because e, b, a, d, e is a simple circuit in this graph. Finally, G4 is not a tree because it is not connected.

Forest:

A disjoint collection of trees is known as forest.

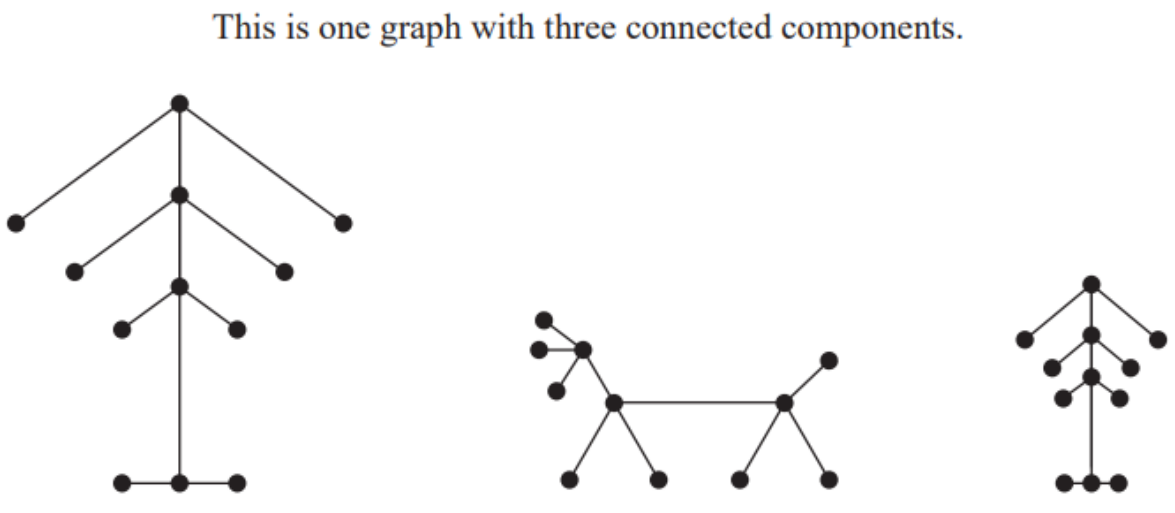


Figure: 2

Theorem 1:

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Proof:

First assume that T is a tree. Then T is a connected graph with no simple circuits. Let x and y be two vertices of T. Because T is connected, by Theorem 1 there is a simple path between x and y. Moreover, this path must be unique, for if there were a second such path, the path formed by combining the first path from x to y followed by the path from y to x obtained by reversing the order of the second path from x to y would form a circuit. This implies that there is a simple circuit in T. Hence, there is a unique simple path between any two vertices of a tree. Now assume that there is a unique simple path between any two vertices of a graph T. Then T is connected, because there is a path between any two of its vertices. Furthermore, T can have no simple circuits. To see that this is true, suppose T had a simple circuit that contained the vertices x and y. Then there would be two simple paths between x and y, because the simple circuit is made up of a simple path from x to y and a second simple path from y to x. Hence, a graph with a unique simple path between any two vertices is a tree.

Theorem 2:

For any positive integer n, if G is a connected graph with n vertices and n-1 edges, then G is a tree.

Proof 1:

By induction on n.

Base Case: Let n = 1, this is the trivial tree with 0 edges. So true for n = 1.

Inductive Step: Let n = k and assume true for k, i.e., every tree with k vertices has k – 1 edges.

Let T be a tree with k + 1 vertices. Let v be a leaf of V (exists by previous Lemma). Consider the tree T' obtained by deleting v from T.

T' has k vertices, so by the inductive hypothesis it has k – 1 edges. But v is a leaf, and so has degree 1, thus T has one more edge than T'.

Proof 2:

Induction on the size of the graph. Assume you have a connected graph of n vertices and m edges. Remove the edges until your graph splits in two parts. By inductive hypothesis both parts have at least n1−1 and n2−1 edges (where n1+n2=n), so your graph had at least (n1−1) + (n2−1) + 1 = n−1 edges (the additional one denotes the last edge you removed before the graph stopped being connected).

Proof 3:

Let G be a connected Graph: If G has no cycles=> then G is a Tree. So, G has n-1 edges.

If G has cycles: and G is connected then for every two vertices there is a path between them. Assuming that G have only one cycle: let's look at the path: v1, v2… vn, v1. We can remove the edge v1, v1 and we will get a connected sub-graph v1, v2 with no cycles and E(H)+1=E(G). So, E(G)=n.

And by induction you will get that for every number of cycles n E(G) ≥ n.

So, if G has cycles E(G) = n−1 else E(G) ≥ n.

Proof 4:

By contradiction. Let G be a connected graph with n vertices and n − 1 edges. Suppose that G is not a tree, thus G has a circuit. Remove an edge from this circuit to get a new graph G1. If G1 still has circuits repeat this process of removing edges from circuits until we get a circuit free graph. Note that removing edges from a circuit never disconnects the graph. Suppose we have repeated this process k times, we are left with a tree with n vertices and n−1−k edges. But this contradicts the previous Theorem.

Rooted Tree:

A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

Definition

- A rooted tree is a tree with a distinguished vertex called the root.
- The level of a vertex in a rooted tree is the number of edges from the vertex to the root.
- the height of a rooted tree is the maximum level of any vertex in the tree.
- Given an internal vertex in a rooted tree its children are those vertices adjacent to it and one level higher.
- If u and v are vertices in a rooted tree, with u a child of v, then v is called the parent of u.
- Two vertices which are children of the same vertex are called siblings.
- Given two vertices u and v in a rooted tree, if u lies on the path from v to the root, then u is called an ancestor of v, and v is called a descendent of u.

Definitions:

Parent:

Suppose that T is a rooted tree. If v is a vertex in T other than the root, the parent of v is the unique vertex u such that there is a directed edge from u to v.

Child:

If u is the parent of v, then v is called a child of u.

Siblings:

Vertices with the same parent are called siblings.

Ancestors:

The ancestors of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.

Descendants:

The descendants of a vertex v are those vertices that have v as an ancestor.

Leaf:

A vertex of a tree is called a leaf if it has no children.

Internal Vertices:

Vertices that have children are called internal vertices.

Subtree:

If a is a vertex in a tree, the subtree with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

m-ary Tree:

A rooted tree is called an m-ary tree if every internal vertex has no more than m children. The tree is called a full m-ary tree if every internal vertex has exactly m children.

An m-ary tree with m = 2 is called a binary tree.

- **Are the rooted trees in Figure 3 full m-ary trees for some positive integer m?**

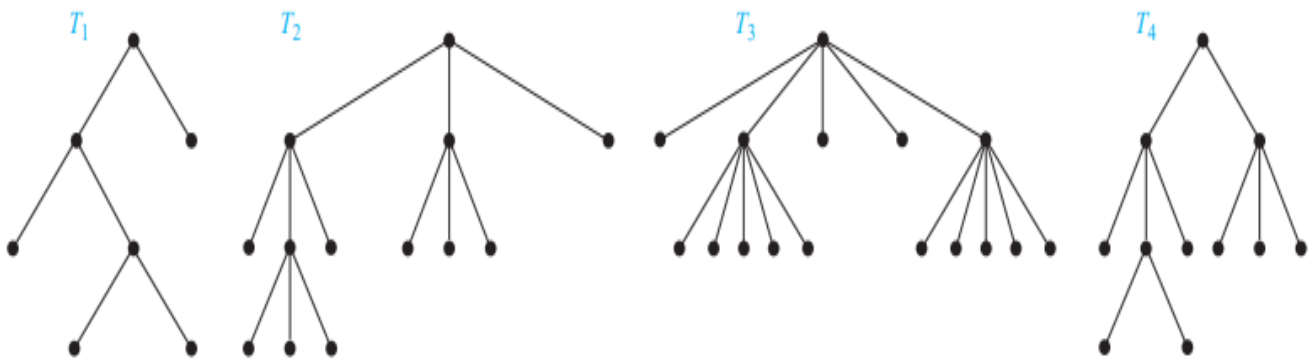


Figure: 3

Solution:

T1 is a full binary tree because each of its internal vertices has two children. T2 is a full 3-ary tree because each of its internal vertices has three children. In T3 each internal vertex has five children, so T3 is a full 5-ary tree. T4 is not a full m-ary tree for any m because some of its internal vertices have two children and others have three children.

- **Answer these questions about the rooted tree illustrated in Figure 4.**

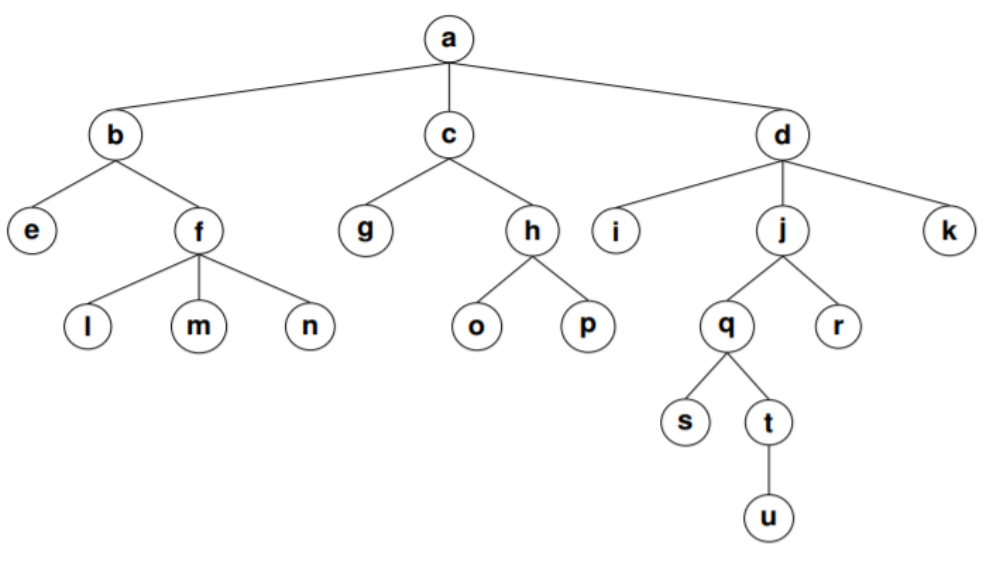


Figure: 4

a) Which vertex is a root?
   Ans: a is the root of the tree.
b) Which vertices are internal?
   Ans: The internal vertices are a, b, c, d, f, h, j, q, t.
c) Which vertices are leaves?
   Ans: The leaves are e, g, i, k, l, m, n, o, p, r, s, u.
d) Which vertices are children of j?
   Ans: The children of j are q and r.
e) Which vertex is the parent of h?
   Ans: The parent of h is c.
f) Which vertices are siblings of o?
   Ans: The sibling of o is p.
g) Which vertices are ancestors of m?
   Ans: The ancestors of m are f, b, a.
h) Which vertices are descendants of b?
   Ans: The descendants of b are e, f, l, m, n.

Ordered Root Tree:

An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered.

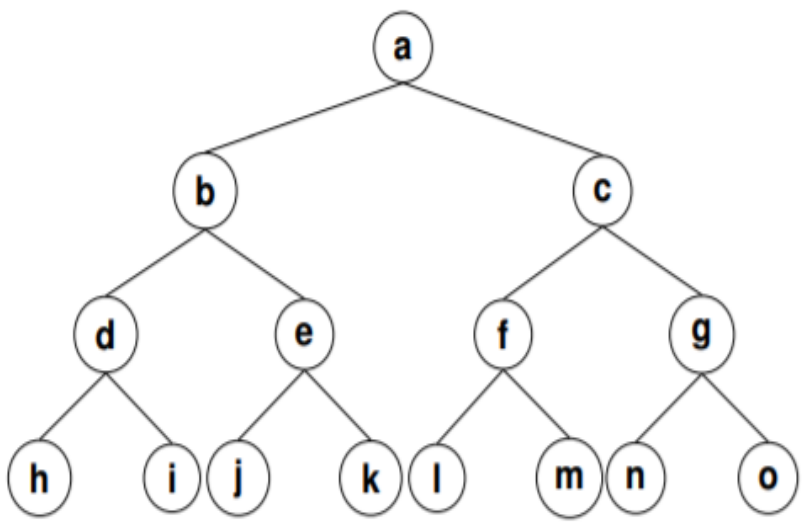


Figure: 5

Left and Right Child:

In an ordered binary tree, the first child is called the left child and the second child is called the right child.

Left and Right Subtree:

The tree rooted at the left child is called the left subtree and the tree rooted at the right child is called the right subtree.

Theorem 3:

A full m-ary tree with i internal vertices contains n = mi + 1 vertices.

Proof:

Every vertex, except the root, is the child of an internal vertex. Because each of the i internal vertices has m children, there are mi vertices in the tree other than the root. Therefore, the tree contains n = mi + 1 vertices.

Theorem 4:

A full m-ary tree with

1. n vertices has $i = \frac{n-1}{m}$ internal vertices and $l = \frac{(m-1)n+1}{m}$ leaves.
2. i internal vertices has n = mi + 1 vertices and l = (m − 1) i + 1 leaves.
3. l leaves has $n = \frac{ml-1}{m-1}$ vertices and $i = \frac{l-1}{m-1}$ internal vertices.

Proof:

Let n represent the number of vertices, i the number of internal vertices, and l the number of leaves. The three parts of the theorem can all be proved using the equality given in Theorem 3, that is, n = mi + 1, together with the equality n = l + i, which is true because each vertex is either a leaf or an internal vertex. We will prove part (i) here. The proofs of parts (ii) and (iii) are left as exercises for the reader. Solving for i in n = mi + 1 gives i = (n − 1)/m. Then inserting this expression for i into the equation n = l + i shows that l = n − i = n − (n − 1)/m = [(m − 1) n + 1]/m.

Level:

The level of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.

Height:

The height of a rooted tree is the maximum of the levels of vertices.

In other words, the height of a rooted tree is the length of the longest path from the root to any vertex.

Balanced Tree:

A rooted m-ary tree of height h is balanced if all leaves are at levels h or h − 1.

Theorem 5:

There are at most m^H leaves in an m-ary tree of height h.

There are at most 2^H leaves in a binary tree of height H.

Proof:

The proof uses mathematical induction on the height. First, consider m-ary trees of height 1. These trees consist of a root with no more than m children, each of which is a leaf. Hence, there are no more than $m^1 = m$ leaves in an m-ary tree of height 1. This is the basis step of the inductive argument.

Now assume that the result is true for all m-ary trees of height less than h; this is the inductive hypothesis. Let T be an m-ary tree of height h. The leaves of T are the leaves of the subtrees of T obtained by deleting the edges from the root to each of the vertices at level 1, as shown in Figure 6 (below).

Each of these subtrees has height less than or equal to h – 1. So, by the inductive hypothesis, each of these rooted trees has at most m^{h-1} leaves. Because there are at most m such subtrees, each with a maximum of m^{h-1} leaves, there are at most $m \times m^{h-1} = m^h$ leaves in the rooted tree. This finishes the inductive argument.

Corollary 1:

If an m-ary tree of height h has l leaves, then $\boldsymbol{h \geq [\log_m l]}$. If the m-ary tree is full and balanced, then $\boldsymbol{h = [\log_m l]}$.

If a Binary tree of height h has l leaves, then $\boldsymbol{h \geq [\log_2 l]}$. If the binary tree is full and balanced, then $\boldsymbol{h = [\log_2 l]}$.

Proof:

We know that $\boldsymbol{l \leq m^h}$ from Theorem 5. Taking logarithms to the base m shows that $\log_m l \leq h$. Because h is an integer, we have $h \geq [\log_m l]$. Now suppose that the tree is balanced.

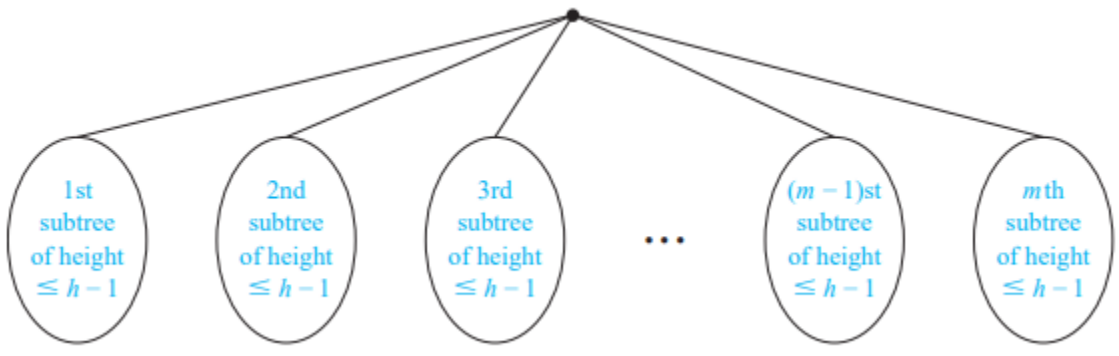


Figure: 6

Then each leaf is at level h or h – 1, and because the height is h, there is at least one leaf at level h. It follows that there must be more than m^{h-1} leaves. Because $l \leq m^h$, we have $m^{h-1} < l \leq m^h$. Taking logarithms to the base m in this inequality gives $h - 1 < \log_m l \leq h$. Hence, $h = [\log_m l\,]$.

We have discussed-

- Tree is a non-linear data structure.
- In a tree data structure, a node can have any number of child nodes.

Tree Traversal Algorithms

Procedures for systematically visiting every vertex of an ordered rooted tree are called traversal algorithms.

We will describe three of the most commonly used such algorithms:

1. **Preorder traversal**
2. **Inorder traversal**
3. **Postorder traversal.**

Find the preorder, inorder and postorder traversal for the following ordered rooted tree.

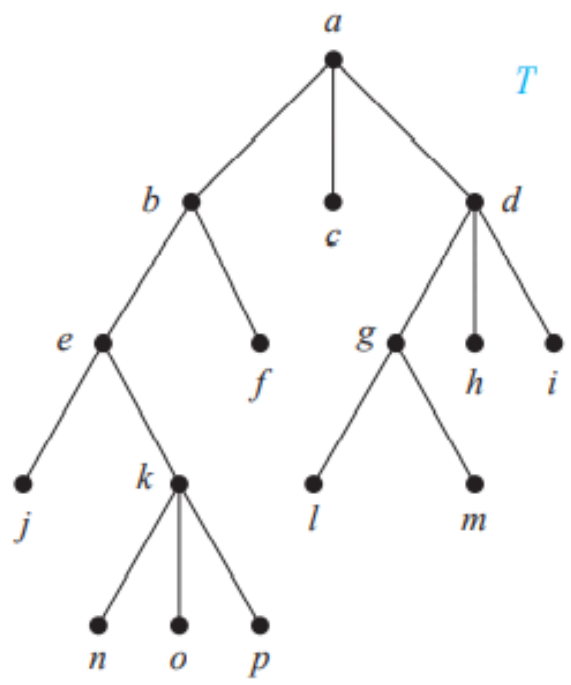


Figure: 8

Preorder Traversal:

Definition:

Let T be an ordered rooted tree with root r. If T consists only of r, then r is the preorder traversal of T. Otherwise, suppose that T1, T2, …Tn are the subtrees at r from left to right in T. The preorder traversal begins by visiting r. It continues by traversing T1 in preorder, then T2 in preorder, and so on, until Tn is traversed in preorder.

ALGORITHM 1 Preorder Traversal.

procedure *preorder*(T: ordered rooted tree)
r := root of T
list r
for each child c of r from left to right
  $T(c)$:= subtree with c as its root
  preorder($T(c)$)

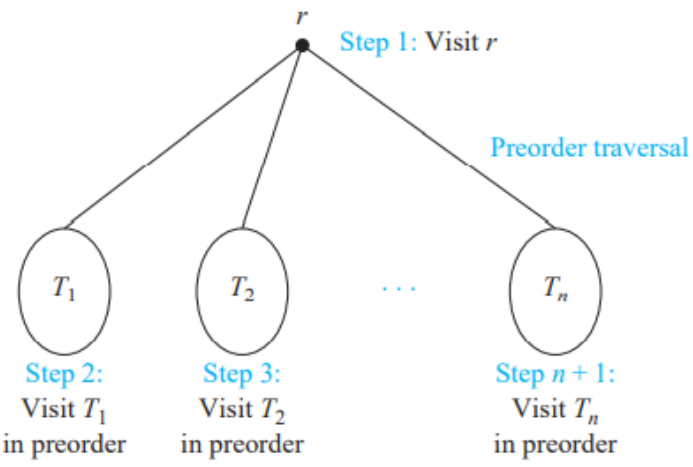
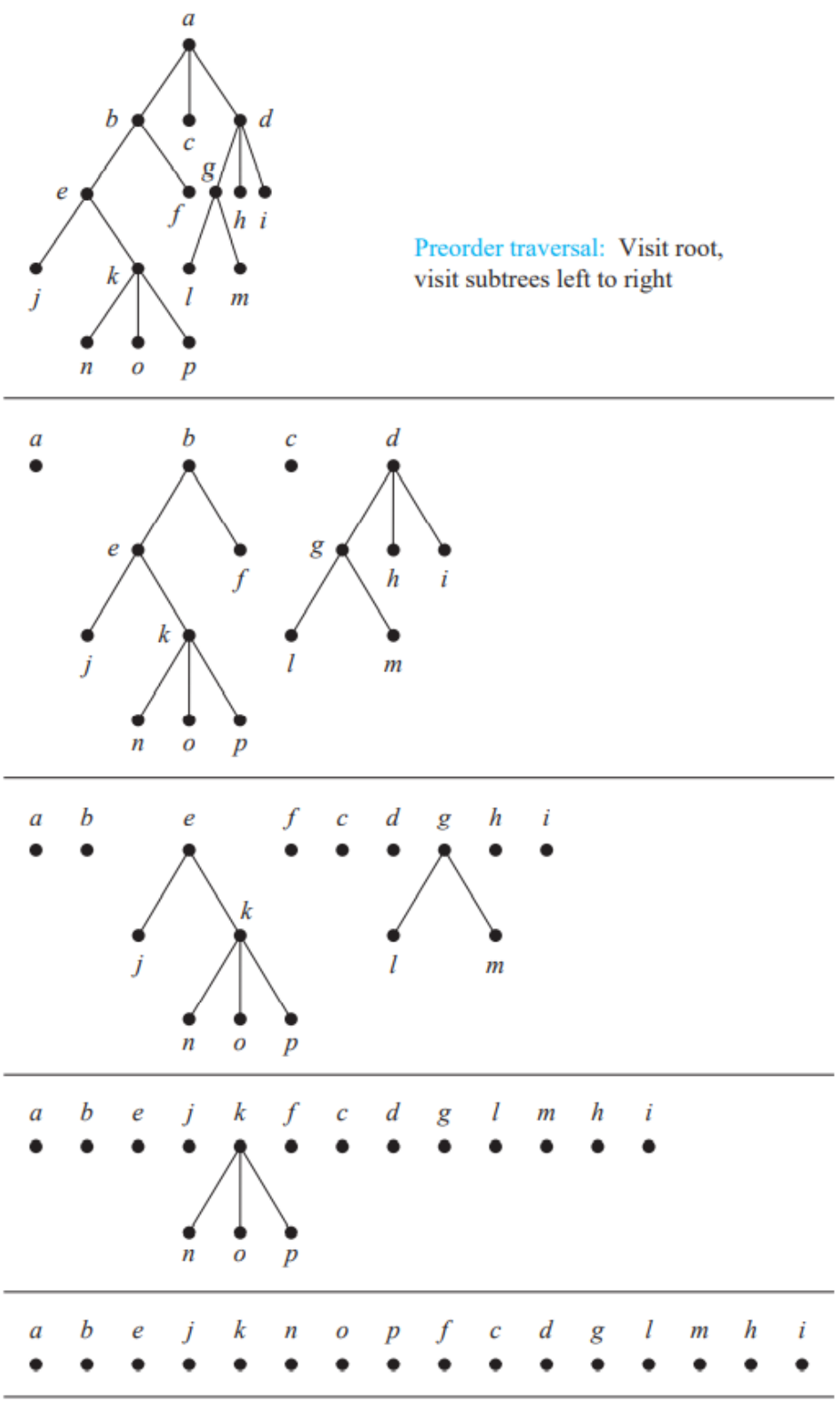


Figure: 9

Steps are shown below:



Preorder Traversal:

Definition:

Let T be an ordered rooted tree with root r. If T consists only of r, then r is the inorder traversal of T. Otherwise, suppose that T1, T2, … Tn are the subtrees at r from left to right. The inorder traversal begins by traversing T1 in inorder, then visiting r. It continues by traversing T2 in inorder, then T3 in inorder, ..., and finally Tn in inorder.

ALGORITHM 2 Inorder Traversal.

procedure *inorder*(T: ordered rooted tree)
r := root of T
if r is a leaf **then** list r
else
  l := first child of r from left to right
  $T(l)$:= subtree with l as its root
  inorder($T(l)$)
  list r
  for each child c of r except for l from left to right
    $T(c)$:= subtree with c as its root
    inorder($T(c)$)

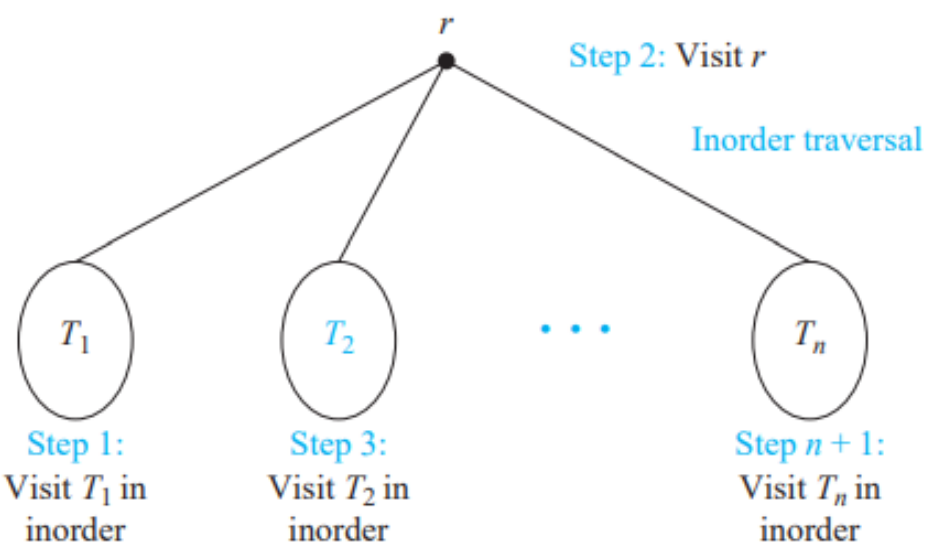
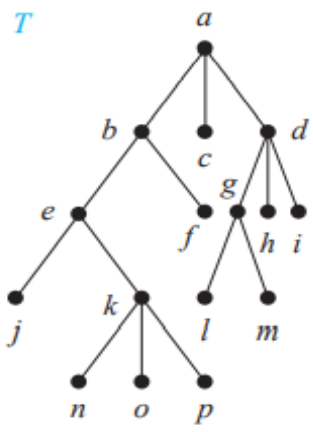
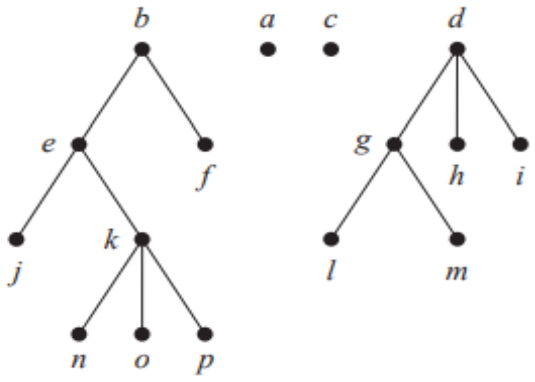


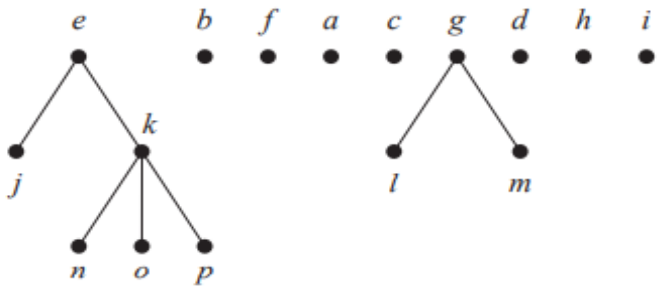
Figure: 10

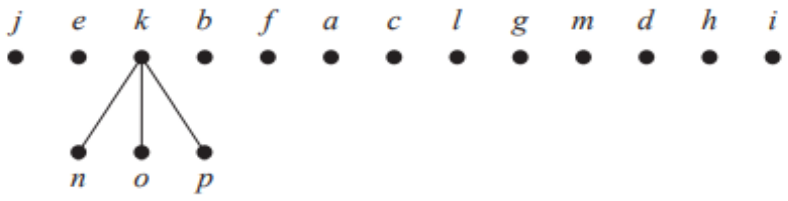
Steps are shown below:

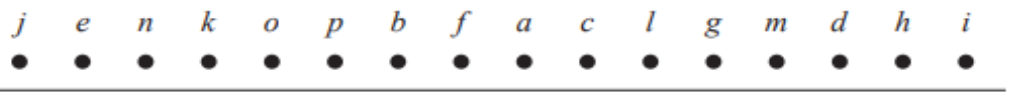


Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right









Postorder Traversal:

Definition:

Let T be an ordered rooted tree with root r. If T consists only of r, then r is the postorder traversal of T. Otherwise, suppose that T1, T2, ..., Tn are the subtrees at r from left to right. The postorder traversal begins by traversing T1 in postorder, then T2 in postorder, ..., then Tn in postorder, and ends by visiting r.

ALGORITHM 3 Postorder Traversal.

procedure *postorder*(*T*: ordered rooted tree)
r := root of *T*
for each child *c* of *r* from left to right
  T(*c*) := subtree with *c* as its root
  postorder(*T*(*c*))
list *r*

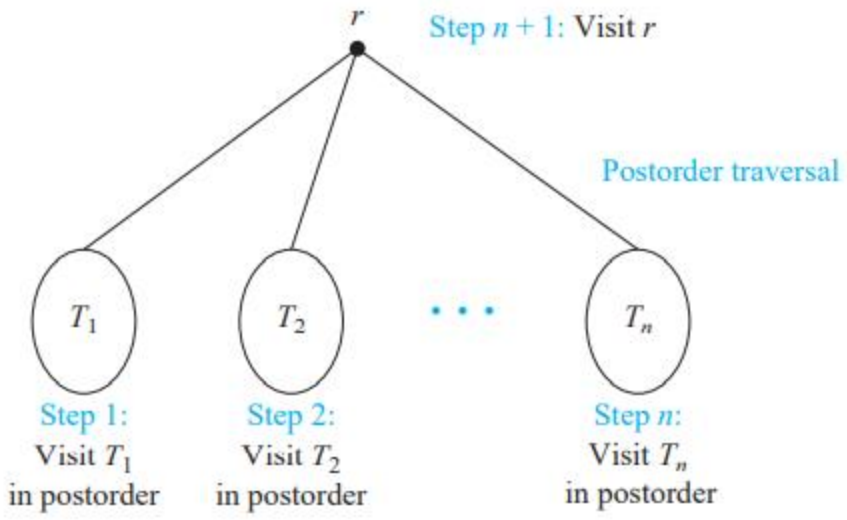
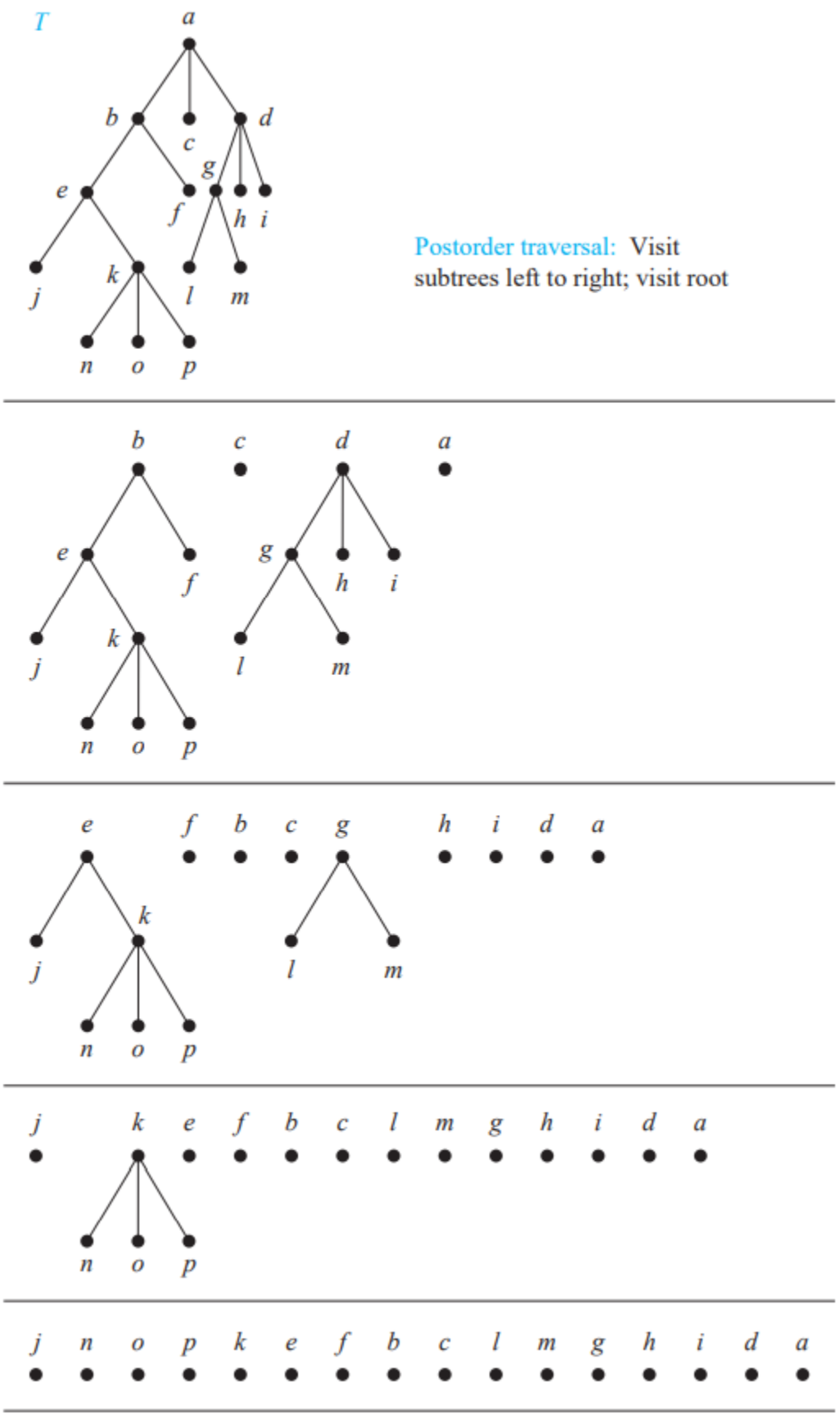


Figure: 11

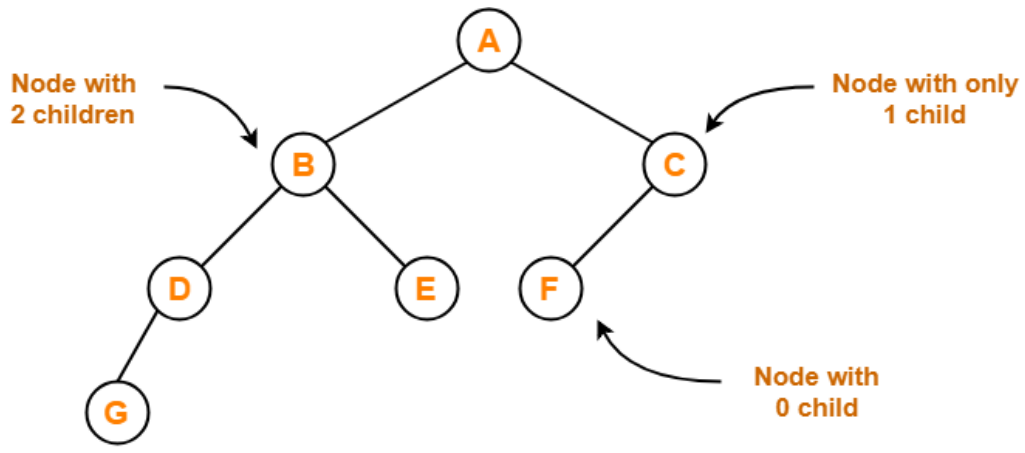
Steps are shown below:



Binary Tree

A Binary tree is a special tree data structure in which each node can have at most 2 children.
Thus, in a binary tree, each node has either 0 child or 1 child or 2 children.

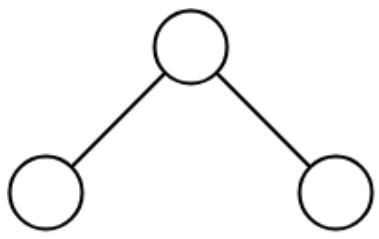
Example:



Binary Tree Example

Unlabeled Binary Tree:

A Binary tree is unlabeled if its nodes are not assigned any label.



Unlabeled Binary Tree

$$\text{Number of different Binary Trees possible with 'n' unlabeled nodes} = \frac{{}^{2n}C_n}{n+1}$$

Example:

Consider we want to draw all the binary trees possible with 3 unlabeled nodes.

Using the above formula, we have-

Number of binary trees possible with 3 unlabeled nodes

$= {}^{2 \times 3}C_3 / (3 + 1)$

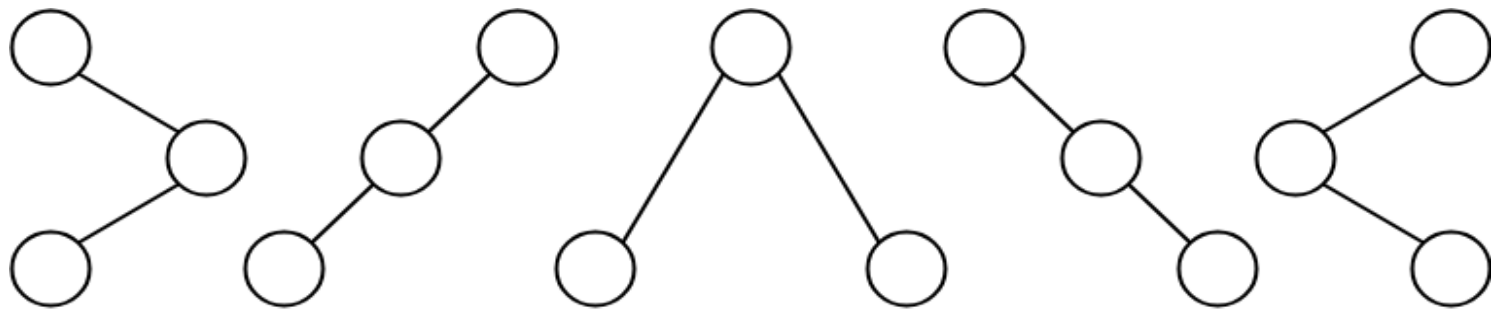
$= {}^{6}C_3 / 4$

$= 5$

Thus,

- With 3 unlabeled nodes, 5 unlabeled binary trees are possible.

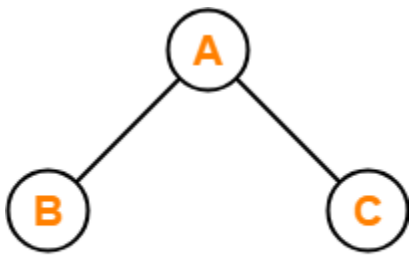
- These unlabeled binary trees are as follows-



Binary Trees Possible With 3 Unlabeled Nodes

Labeled Binary Tree:

A Binary tree is labeled if all its nodes are assigned a label.



Labeled Binary Tree

Number of different Binary Trees possible with 'n' labeled nodes	$= \frac{{}^{2n}C_n}{n+1} \times n!$

Example:

Consider we want to draw all the binary trees possible with 3 labeled nodes.

Using the above formula, we have-

Number of binary trees possible with 3 labeled nodes

$= \{{}^{2 \times 3}C_3 / (3 + 1)\} \times 3!$

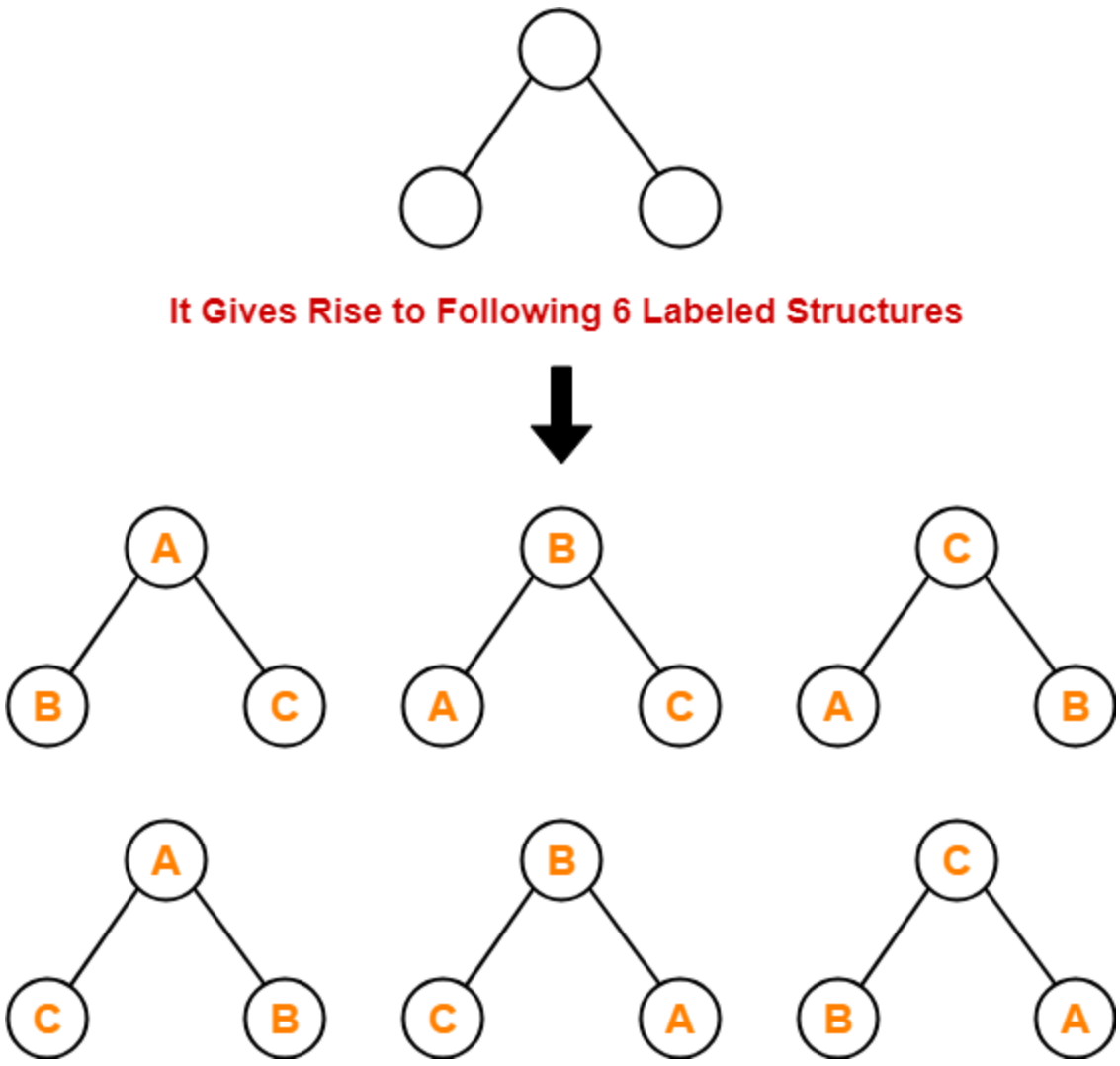
$= \{{}^{6}C_3 / 4\} \times 6$

$= 5 \times 6$

$= 30$

Thus,

- With 3 labeled nodes, 30 labeled binary trees are possible.
- Each unlabeled structure gives rise to 3! = 6 different labeled structures.

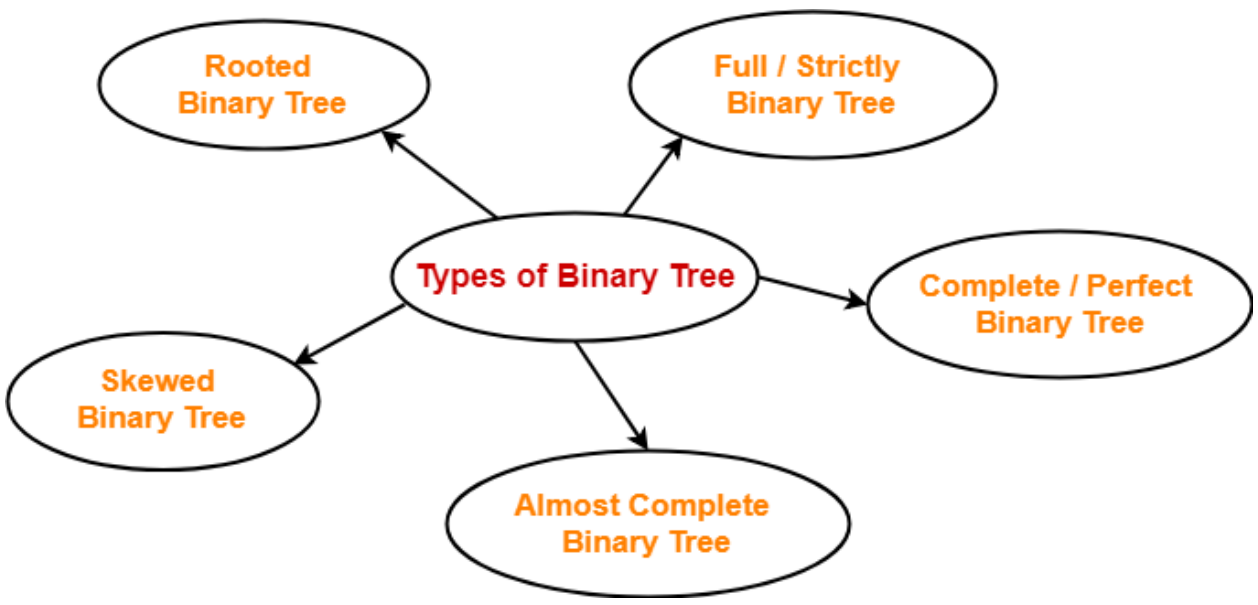


Similarly,

- Every other unlabeled structure gives rise to 6 different labeled structures.
- Thus, in total 30 different labeled binary trees are possible.

Types of Binary Tree

Binary trees can be of the following types-



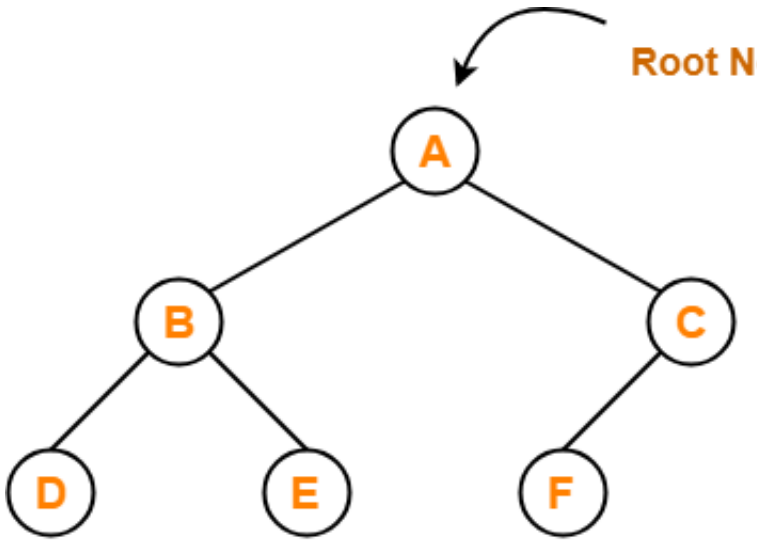
1. Rooted Binary Tree
2. Full / Strictly Binary Tree
3. Complete / Perfect Binary Tree
4. Almost Complete Binary Tree
5. Skewed Binary Tree

1.Rooted Binary Tree:

A rooted binary tree is a binary tree that satisfies the following 2 properties-

- It has a root node.
- Each node has at most 2 children.

Example:

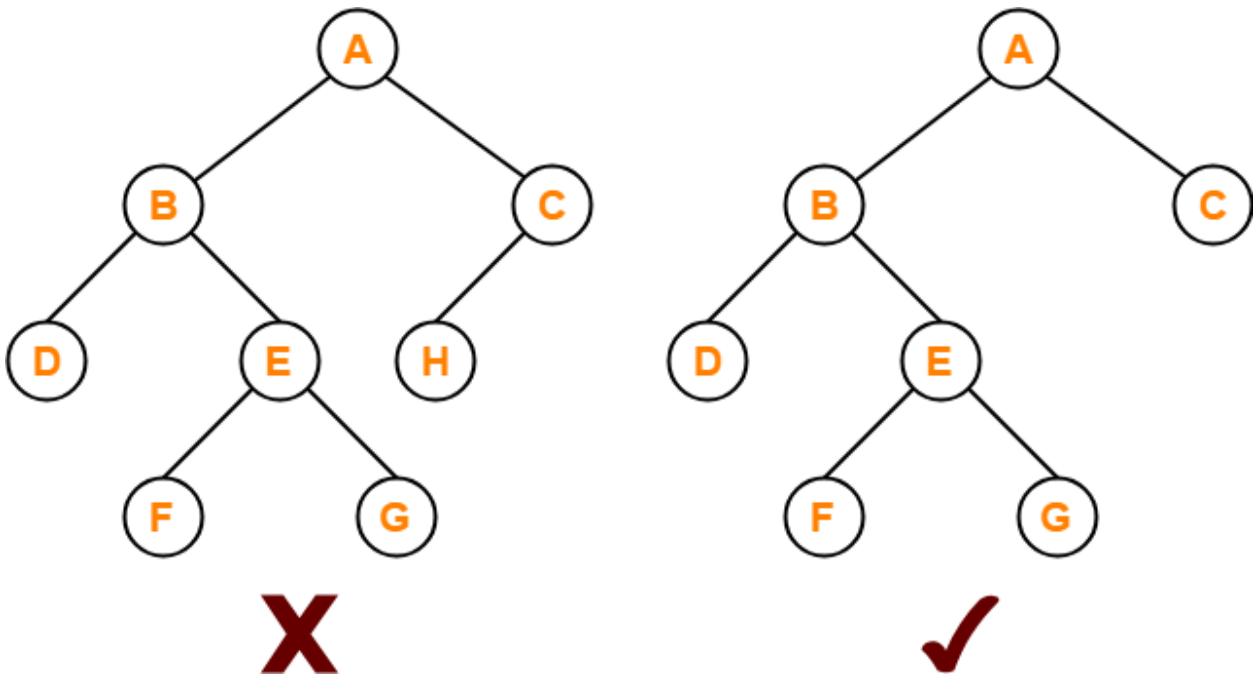


Rooted Binary Tree

2.Full / Strictly Binary Tree:

- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full binary tree is also called as Strictly binary tree.

Example:



Here,

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

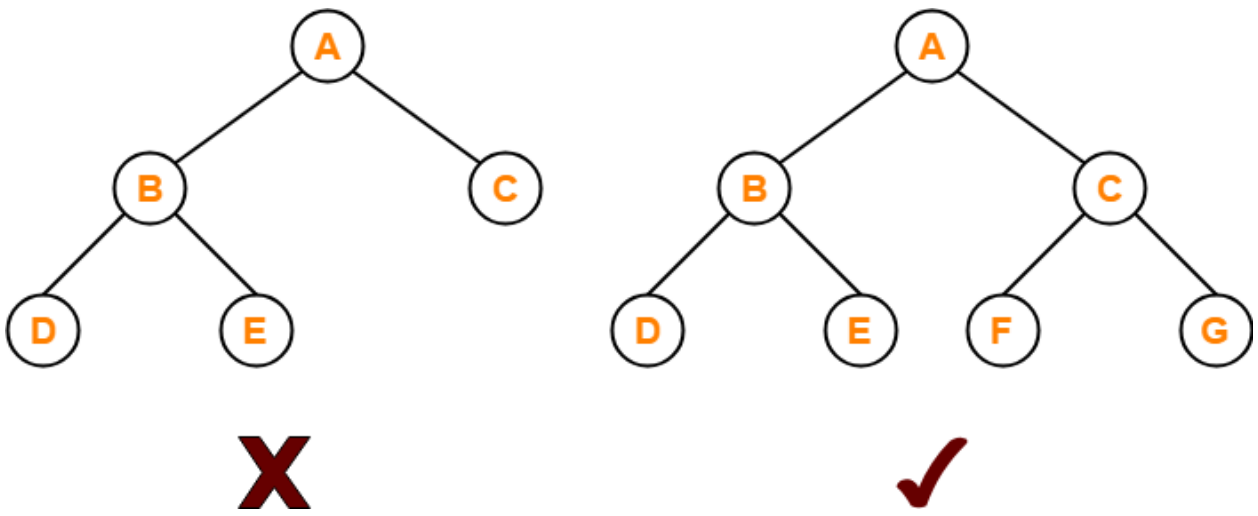
3.Complete / Perfect Binary Tree:

A complete binary tree is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

Complete binary tree is also called as Perfect binary tree.

Example:



Here,

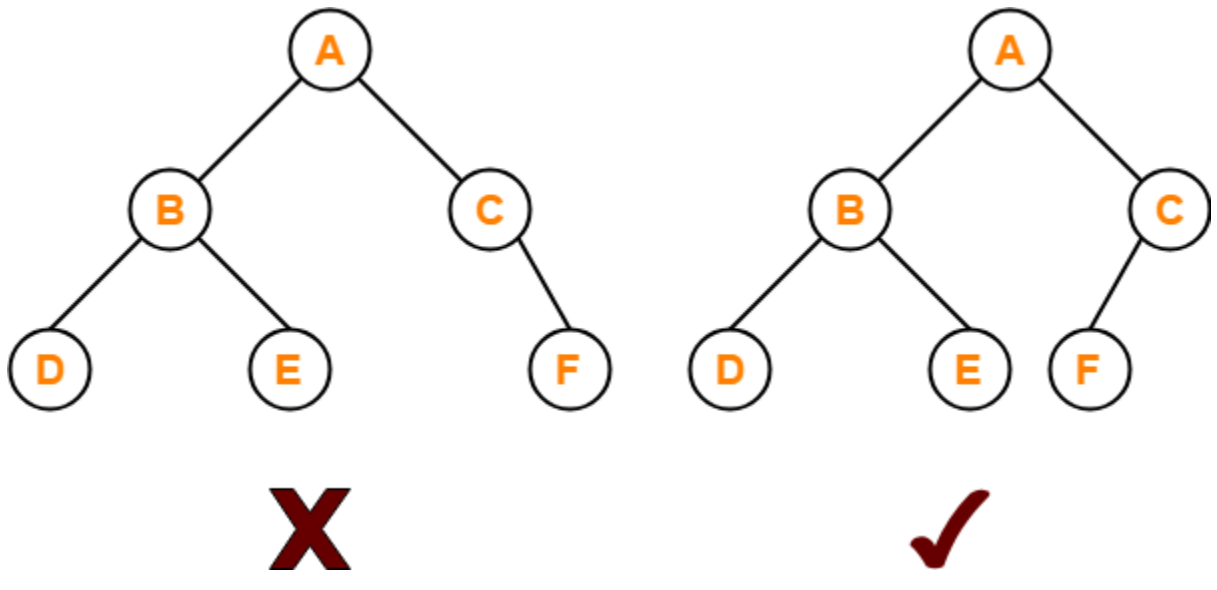
- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

4.Almost Complete Binary Tree:

An almost complete binary tree is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.

Example:



Here,

- First binary tree is not an almost complete binary tree.
- This is because the last level is not filled from left to right.

5.Skewed Binary Tree:

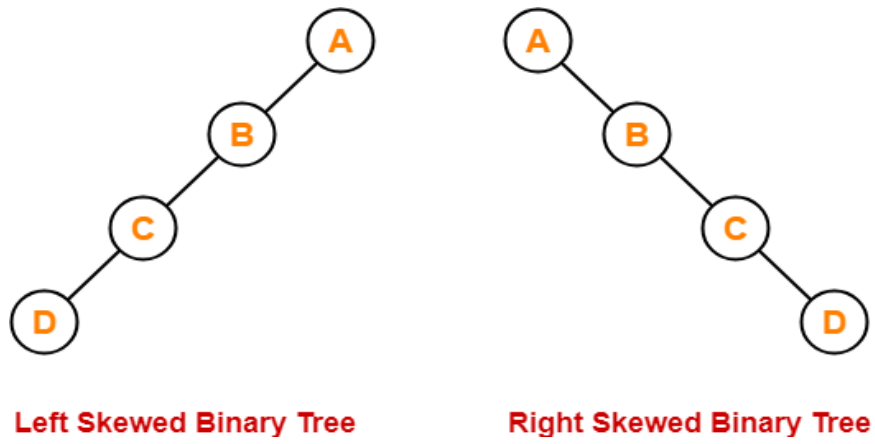
A skewed binary tree is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

A skewed binary tree is a binary tree of n nodes such that its depth is (n-1).

Example:



Left Skewed Binary Tree Right Skewed Binary Tree

Binary Tree Properties

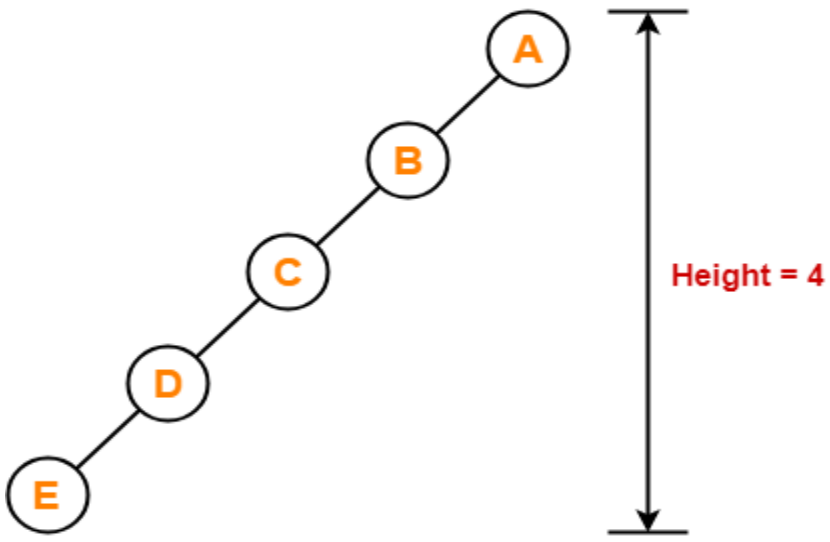
Important properties of binary trees are-

Property – 01:

Minimum number of nodes in a binary tree of height H = H + 1

Example:

To construct a binary tree of height = 4, we need at least 4 + 1 = 5 nodes.



Property – 02:

Maximum number of nodes in a binary tree of height H = $2^{H+1} - 1$

Example:

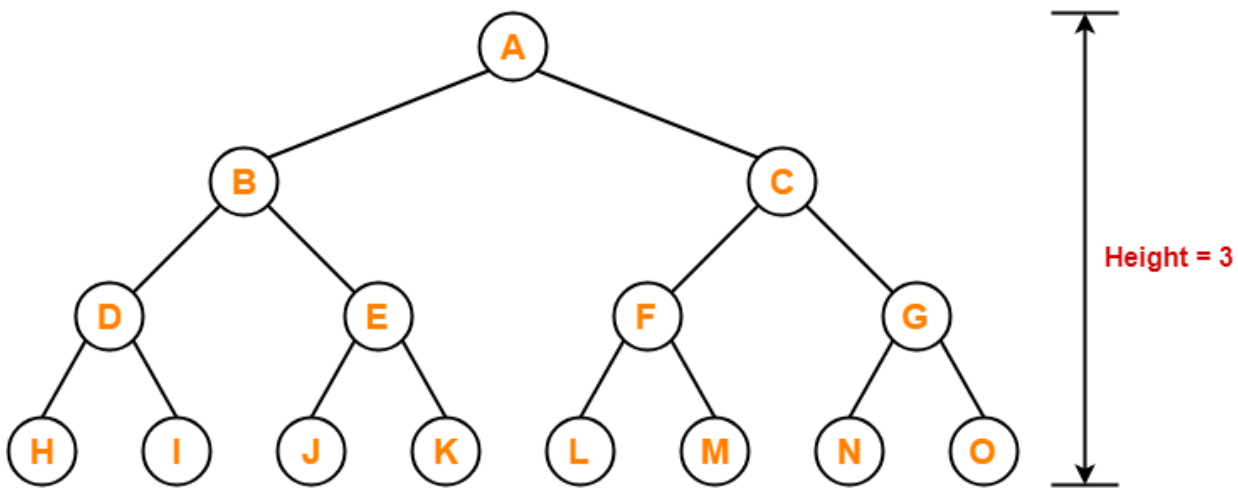
Maximum number of nodes in a binary tree of height 3

= $2^{3+1} - 1$

= 16 – 1

= 15 nodes

Thus, in a binary tree of height = 3, maximum number of nodes that can be inserted = 15.



We cannot insert a greater number of nodes in this binary tree.

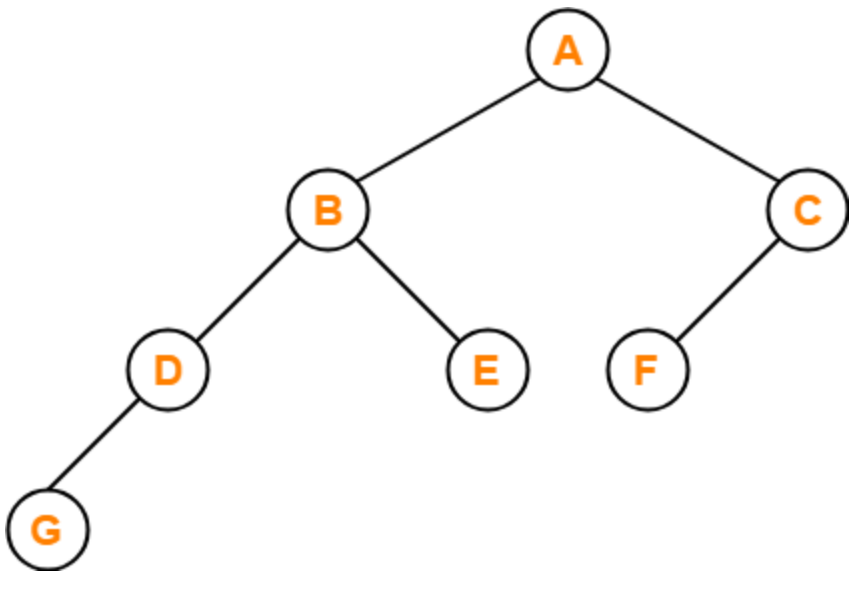
Property – 03:

Total number of leaf nodes in a Binary Tree

= Total Number of nodes with 2 children + 1

Example:

Consider the following binary tree-



Here,

- Number of leaf nodes = 3
- Number of nodes with 2 children = 2

Clearly, number of leaf nodes is one greater than number of nodes with 2 children.

This verifies the above relation.

NOTE

It is interesting to note that-

Number of leaf nodes in any binary tree depends only on the number of nodes with 2 children.

Property – 04:

Maximum number of nodes at any level 'L' in a binary tree = 2^L

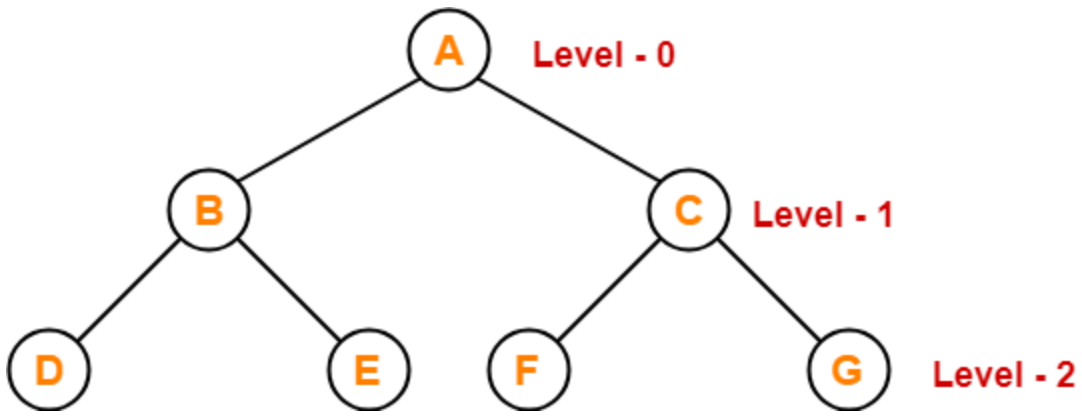
Example:

Maximum number of nodes at level-2 in a binary tree

= 2^2

= 4

Thus, in a binary tree, maximum number of nodes that can be present at level-2 = 4.



PRACTICE PROBLEMS BASED ON BINARY TREE PROPERTIES-

Problem-01:

A binary tree T has n leaf nodes. The number of nodes of degree-2 in T is _____?

a) $\log_2 n$

b) n-1

c) n

d) 2^n

Solution-

Using property-3, we have-

Number of degree-2 nodes

= Number of leaf nodes – 1

= n – 1

Thus, Option (B) is correct.

Problem-02:

In a binary tree, for every node the difference between the number of nodes in the left and right subtrees is at most 2. If the height of the tree is h > 0, then the minimum number of nodes in the tree is _____?

a) 2^{h-1}

b) $2^{h-1} + 1$

c) $2^h - 1$

d) 2^h

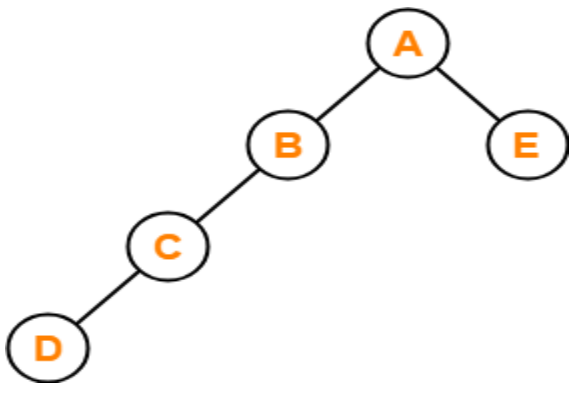
Solution-

Let us assume any random value of h. Let h = 3.

Then the given options reduce to-

1. 4
2. 5
3. 7
4. 8

Now, consider the following binary tree with height h = 3



- This binary tree satisfies the question constraints.
- It is constructed using minimum number of nodes.

Thus, Option (B) is correct.

Problem-03:

In a binary tree, the number of internal nodes of degree-1 is 5 and the number of internal nodes of degree-2 is 10. The number of leaf nodes in the binary tree is _____?

a) 10

b) 11

c) 12

d) 15

Solution-

Using property-3, we have-

Number of leaf nodes in a binary tree

= Number of degree-2 nodes + 1

= 10 + 1

= 11

Thus, Option (B) is correct.

Problem-04:

The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is _____?

1. 2^h
2. $2^{h-1} - 1$
3. $2^{h+1} - 1$
4. 2^{h+1}

Solution-

Using property-2, Option (C) is correct.

Problem-05:

A binary tree T has 20 leaves. The number of nodes in T having 2 children is _____?

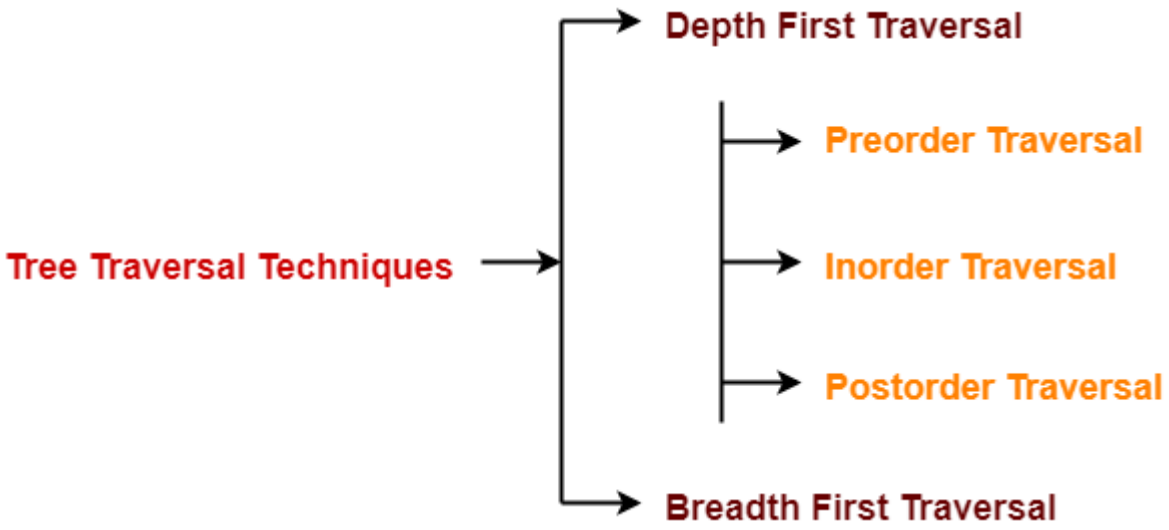
Solution-

Using property-3, correct answer is 19.

Binary Tree Traversal

Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

Various tree traversal techniques are-



Depth First Traversal-

Following three traversal techniques fall under Depth First Traversal-

1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

1. Preorder Traversal-

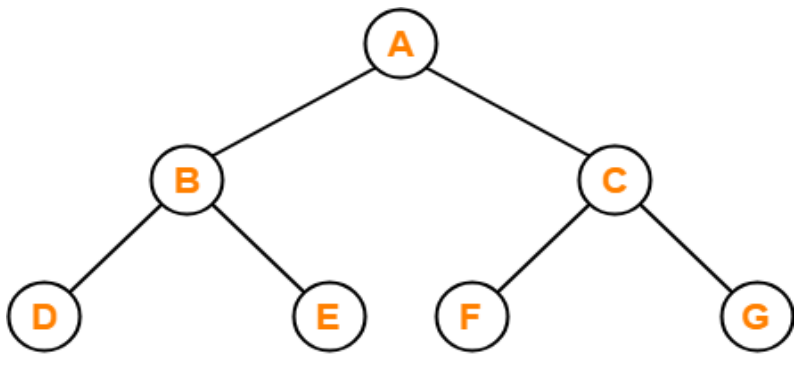
Algorithm-

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

Example-

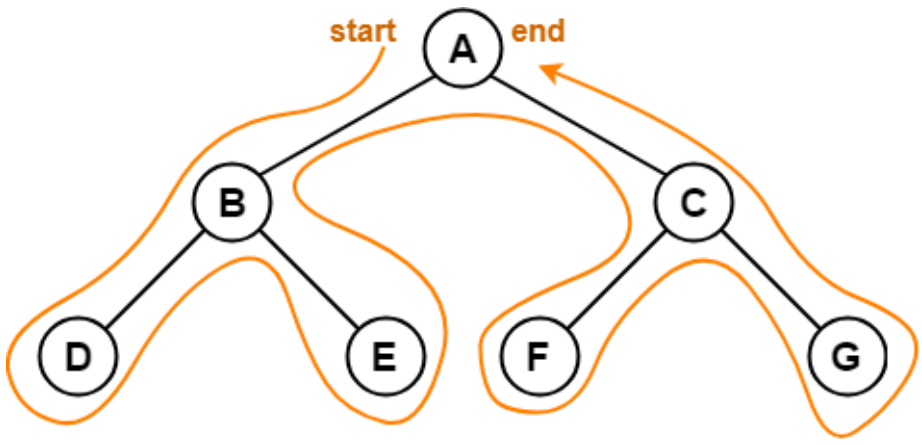
Consider the following example-



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal Shortcut:

Traverse the entire tree starting from the root node keeping yourself to the left.



Preorder Traversal : A , B , D , E , C , F , G

Applications-

- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.

2. Inorder Traversal-

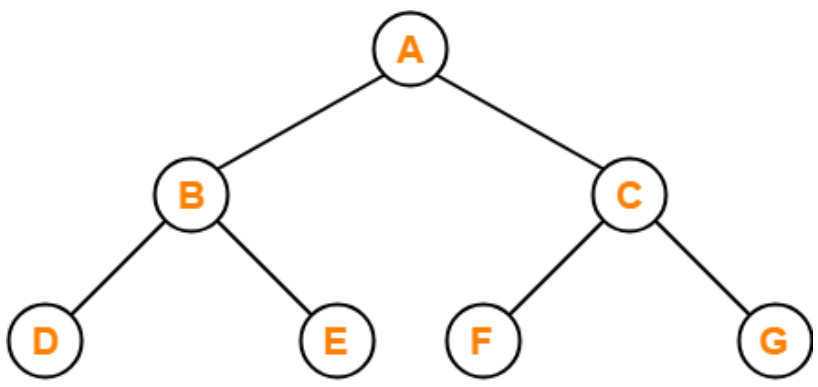
Algorithm-

1. Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root
3. Traverse the right sub tree i.e. call In order (right sub tree)

Left → Root → Right

Example-

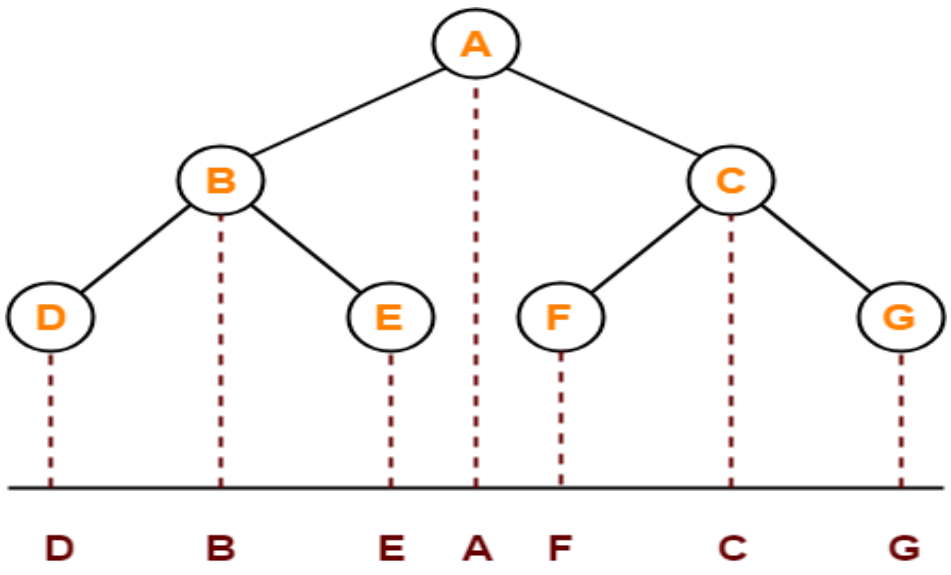
Consider the following example-



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal Shortcut:

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

Applications-

- Inorder traversal is used to get infix expression of an expression tree.

3. Postorder Traversal-

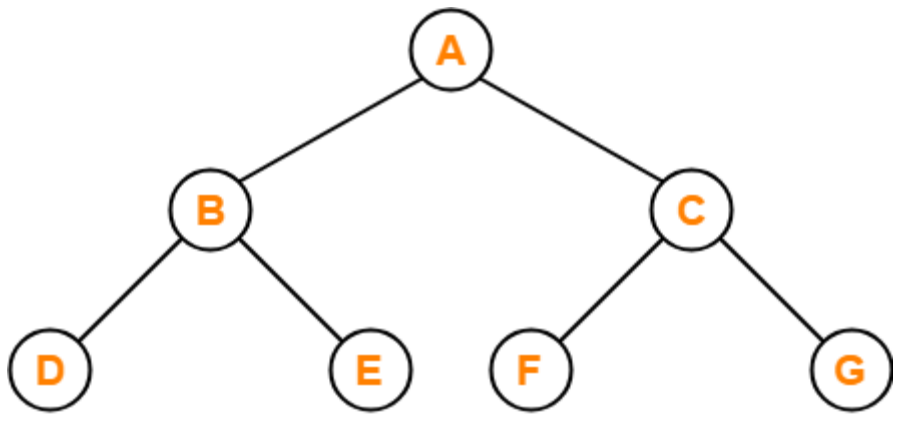
Algorithm-

1. Traverse the left sub tree i.e., call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Example-

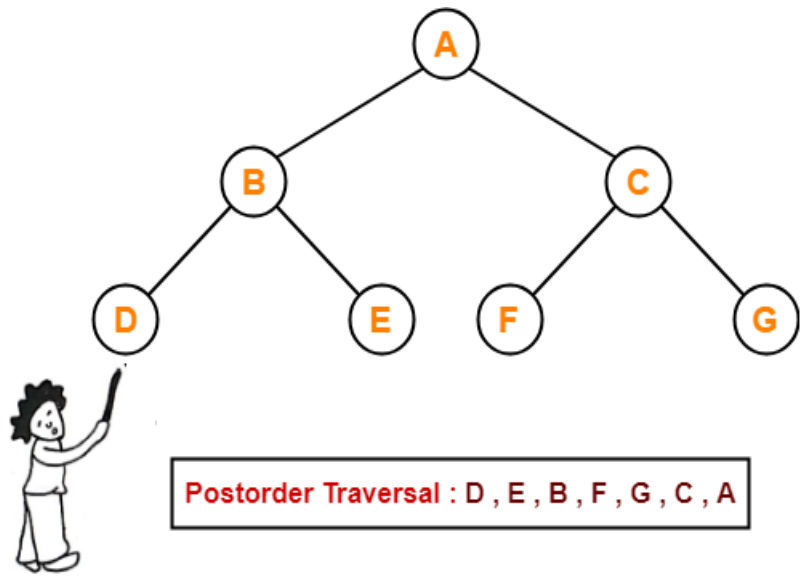
Consider the following example-



Postorder Traversal : D , E , B , F , G , C , A

Postorder Traversal Shortcut:

Pluck all the leftmost leaf nodes one by one.



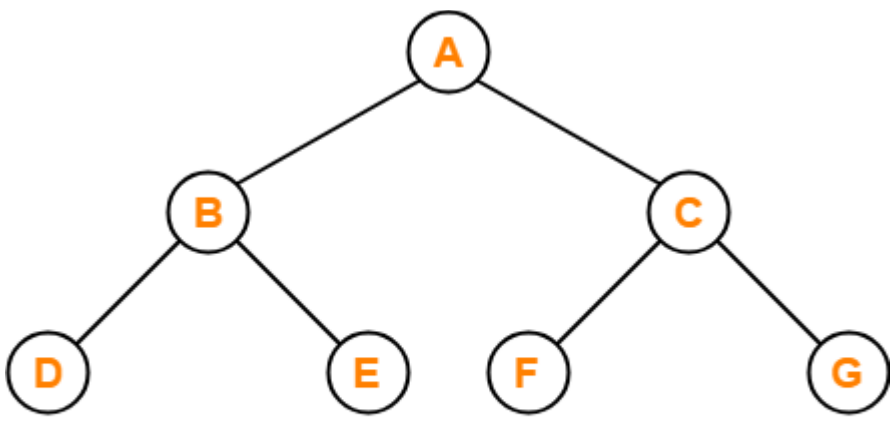
Applications-

- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree.
- This is because it deletes the children first and then it deletes the parent.

Breadth First Traversal-

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.

Example-



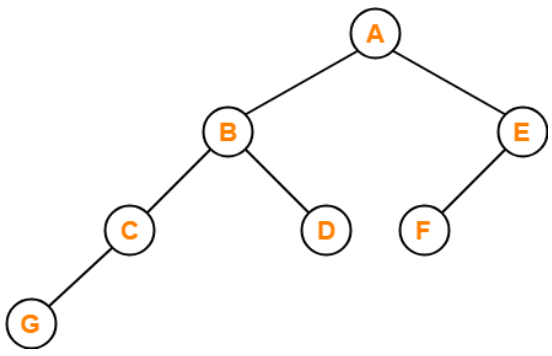
Applications-

- Level order traversal is used to print the data in the same order as stored in the array representation of complete binary tree.

PRACTICE PROBLEMS BASED ON TREE TRAVERSAL-

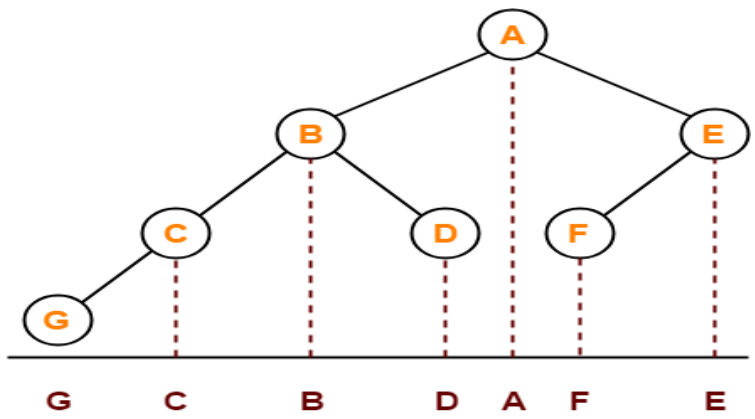
Problem-01:

If the binary tree in figure is traversed in inorder, then the order in which the nodes will be visited is ____?



Solution-

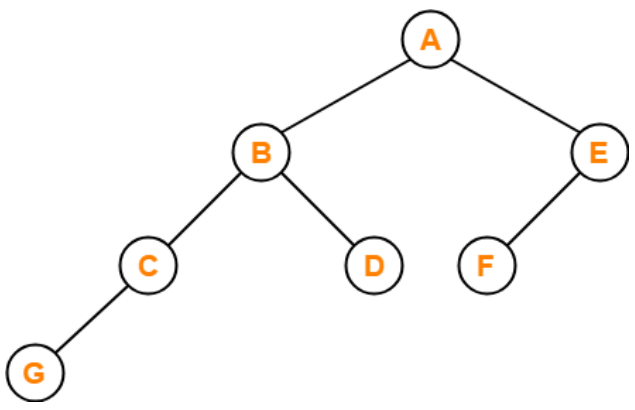
The inorder traversal will be performed as-



Inorder Traversal : G , C , B , D , A , F , E

Problem-02:

Which of the following sequences denotes the postorder traversal sequence of the tree shown in figure?



a) FEGCBDBA

b) GCBDAFE

c) GCDBFEA

d) FDEGCBA

Solution-

Perform the postorder traversal by plucking all the leftmost leaf nodes one by one.

Then, Postorder Traversal: G, C, D, B, F, E, A

Thus, Option (C) is correct.

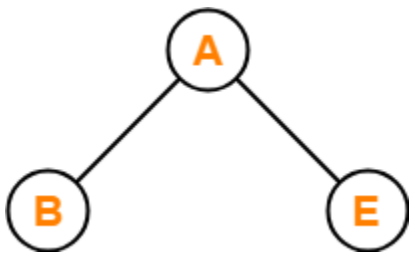
Problem-03:

Let LASTPOST, LASTIN, LASTPRE denote the last vertex visited in a postorder, inorder and preorder traversal respectively of a complete binary tree. Which of the following is always true?

a) LASTIN = LASTPOST
b) LASTIN = LASTPRE
c) LASTPRE = LASTPOST
d) None of these

Solution-

Consider the following complete binary tree-



Preorder Traversal: A, B, E

Inorder Traversal: B, A, E

Postorder Traversal: B, E, A

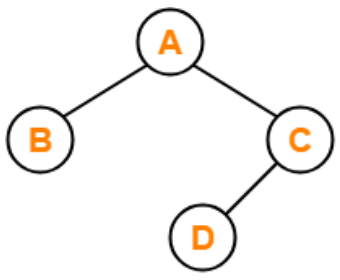
Clearly, LASTIN = LASTPRE.

Thus, Option (B) is correct.

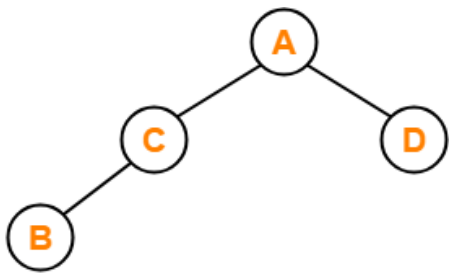
Problem-04:

Which of the following binary trees has its inorder and preorder traversals as BCAD and ABCD respectively-

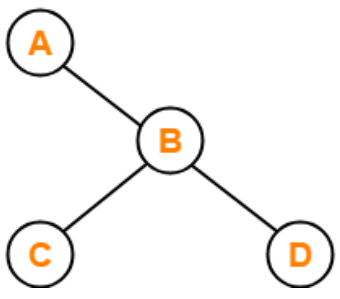
A)



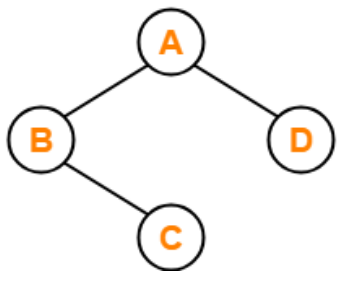
B)



C)



D)



Solution-

Option (D) is correct.

Infix, Prefix, and Postfix Notation

Outlines

- Construct tree from mathematical expression (postfix, infix and prefix).
- Convert mathematical expression (postfix, infix and prefix) using binary tree.
- Value of mathematical expression (postfix, infix and prefix) using binary tree.

EXAMPLE:

What is the ordered rooted tree that represents the expression ((x + y) ↑ 2) + ((x – 4) / 3)?

Solution:

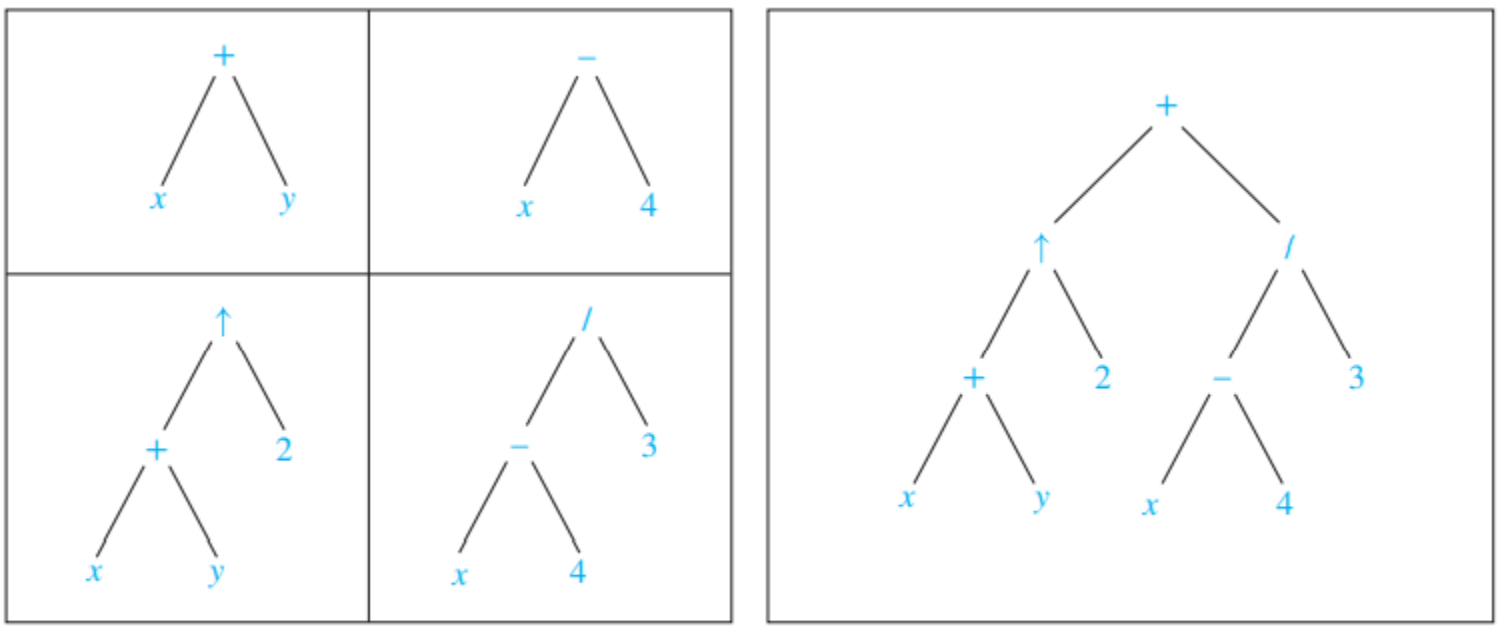


FIGURE 10 A binary tree representing $((x + y) \uparrow 2) + ((x - 4)/3)$.

EXAMPLE:

What is the **prefix** form for ((x + y) ↑ 2) + ((x – 4) / 3)?

Solution:

We obtain the prefix form for this expression by traversing the binary tree that represents it in **preorder**, shown in Figure 10.

This produces **+ ↑ + x y 2 / − x 4 3**.

EXAMPLE:

What is the **postfix** form of the expression ((x + y) ↑ 2) + ((x – 4)/3)?

Solution:

The postfix form of the expression is obtained by carrying out a post-order traversal of the binary tree for this expression, shown in Figure 10.

This produces the postfix expression: **x y + 2 ↑ x 4 − 3 / +**

EXAMPLE:

What is the value of the **prefix** expression + − ∗ 2 3 5 ∕ ↑ 2 3 4?

Solution:

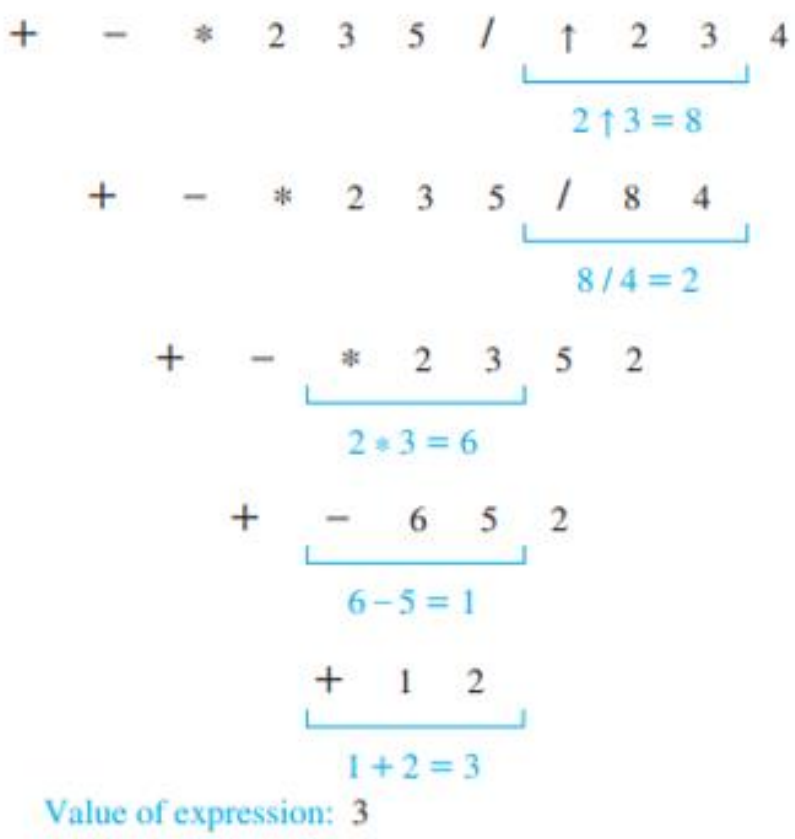


FIGURE 12 **Evaluating a prefix expression.**

EXAMPLE:

What is the value of the **postfix** expression 7 2 3 ∗ −4 ↑ 9 3∕+?

Solution:

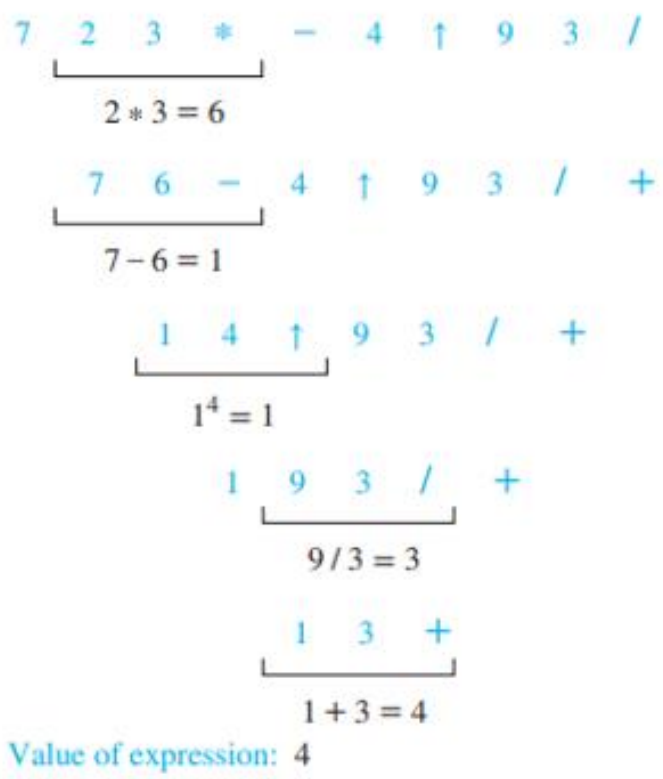


FIGURE 13 **Evaluating a postfix expression.**

EXAMPLE:

Find the ordered rooted tree representing the compound proposition (¬ (p ∧ q)) ↔ (¬p ∨ ¬q). Then use this rooted tree to find the prefix, postfix, and infix forms of this expression.

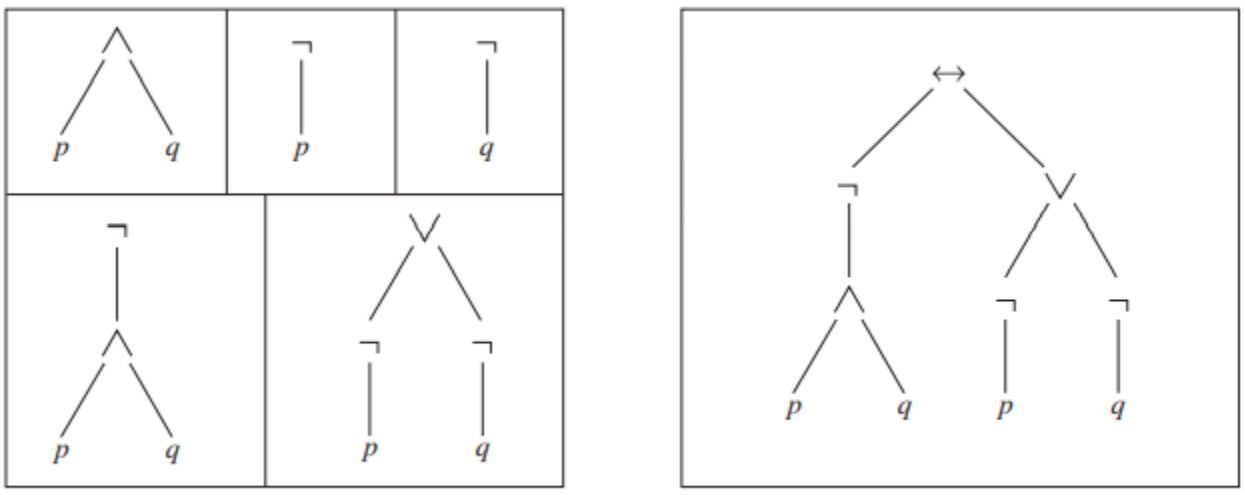


FIGURE 14 **Constructing the rooted tree for a compound proposition.**

Solution:

The rooted tree for this compound proposition is constructed from the bottom up. Examples First, subtrees for ¬p and ¬q are formed (where ¬ is considered a unary operator). Also, a subtree for p ∧ q is formed. Then subtrees for ¬(p ∧ q) and (¬p) ∨ (¬q) are constructed.

Finally, these two subtrees are used to form the final rooted tree. The steps of this procedure are shown in Figure 14.

The prefix, postfix, and infix forms of this expression are found by traversing this rooted tree in preorder, postorder, and inorder (including parentheses), respectively. These traversals give

- ↔ ¬ ∧ pq ∨ ¬p¬q,
- pq ∧ ¬p¬q¬ ∨ ↔, and
- (¬(p ∧ q)) ↔ ((¬p) ∨ (¬q)), respectively

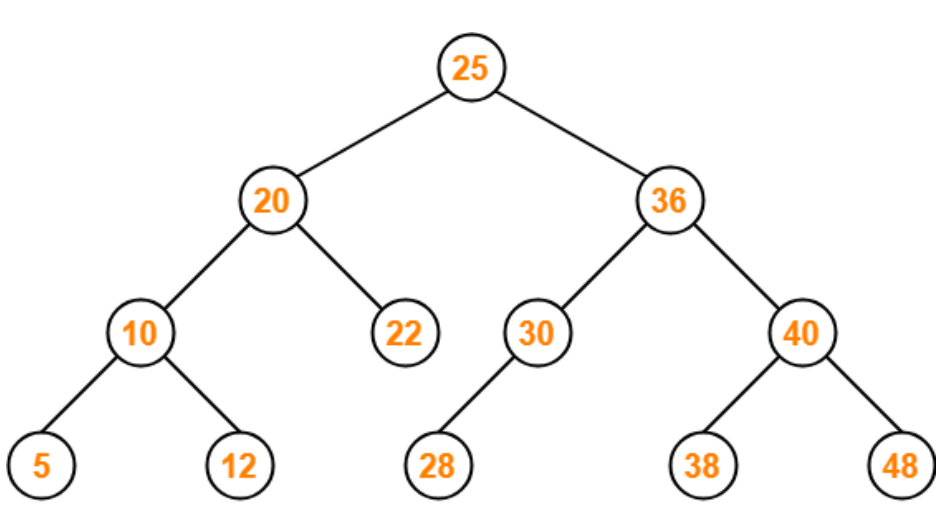
Binary Search Tree

Binary Search Tree is a special kind of binary tree in which nodes are arranged in a specific order.

In a binary search tree (BST), each node contains-

- Only smaller values in its left sub tree
- Only larger values in its right sub tree

Example-



Binary Search Tree

Number of Binary Search Trees-

Number of distinct Binary Search Trees possible with n distinct keys = $\frac{{}^{2n}C_n}{n+1}$

Example-

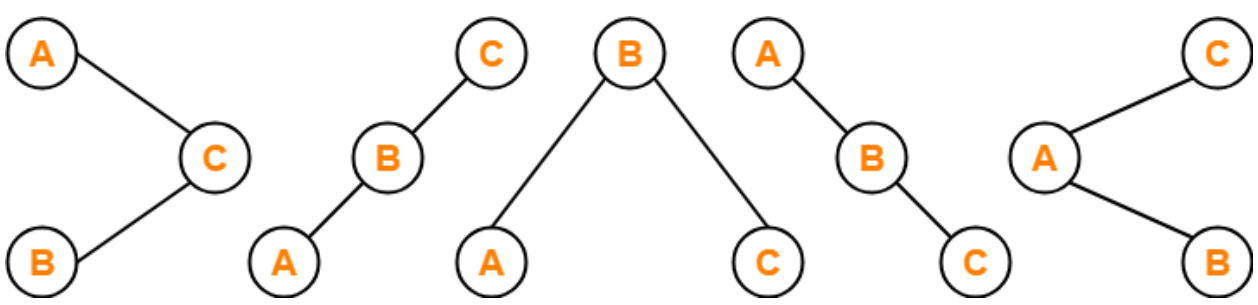
Number of distinct binary search trees possible with 3 distinct keys

$= {}^{2\times3}C_3 / 3+1$

$= {}^{6}C_3 / 4$

$= 5$

If three distinct keys are A, B and C, then 5 distinct binary search trees are-



Binary Search Tree Construction-

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

procedure *insertion*(T: binary search tree, x: item)
$v :=$ root of T
{a vertex not present in T has the value *null*}
while $v \neq null$ and $label(v) \neq x$
  if $x < label(v)$ **then**
    if left child of $v \neq null$ **then** $v :=$ left child of v
    else add *new vertex* as a left child of v and set $v := null$
  else
    if right child of $v \neq null$ **then** $v :=$ right child of v
    else add *new vertex* as a right child of v and set $v := null$
if root of $T = null$ **then** add a vertex v to the tree and label it with x
else if v is null or $label(v) \neq x$ **then** label *new vertex* with x and let v be this new vertex
return v {$v =$ location of x}

Let us understand the construction of a binary search tree using the following example-

Example:

Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

When elements are given in a sequence,

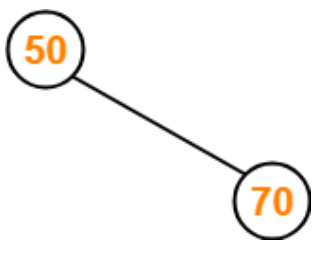
- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

The binary search tree will be constructed as explained below-

Insert 50-

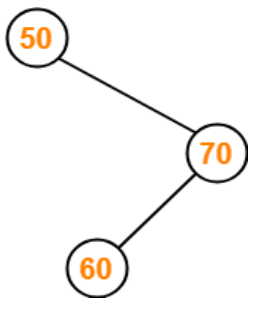
Insert 70-

- As 70 > 50, so insert 70 to the right of 50.



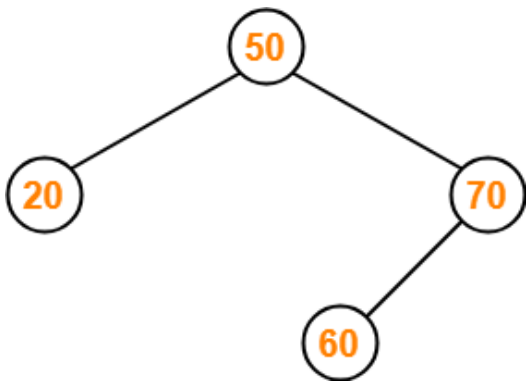
Insert 60-

- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.



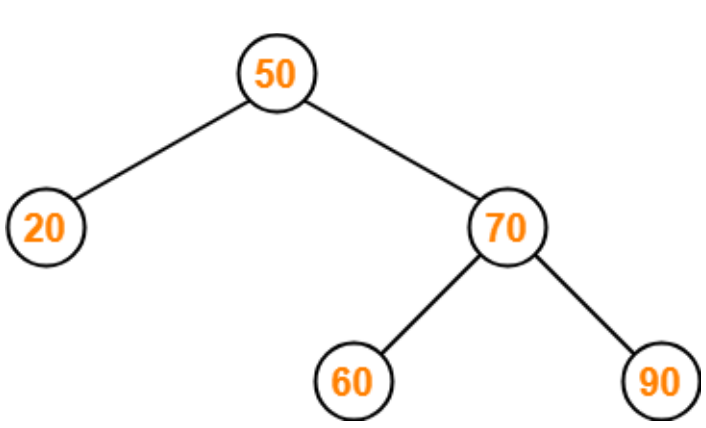
Insert 20-

- As 20 < 50, so insert 20 to the left of 50.



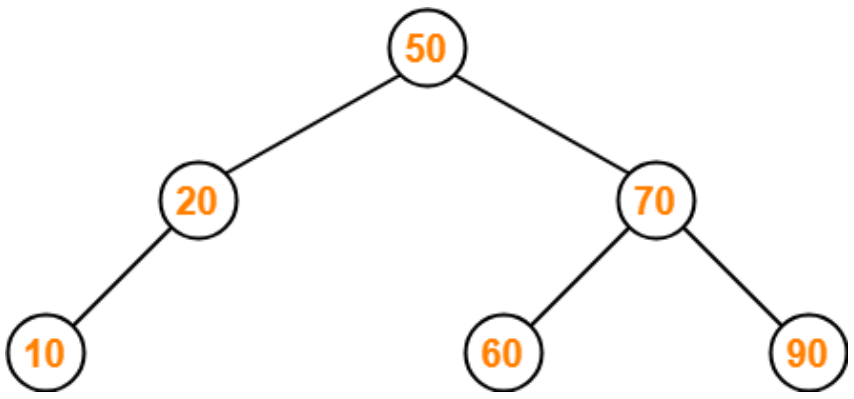
Insert 90-

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



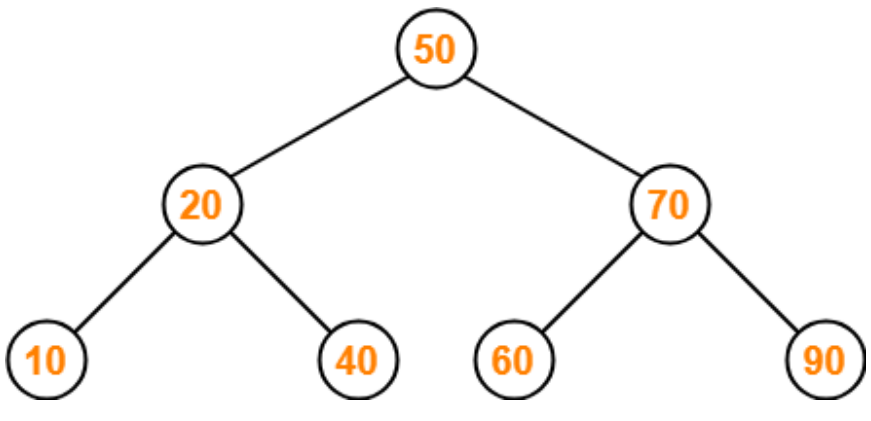
Insert 10-

- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



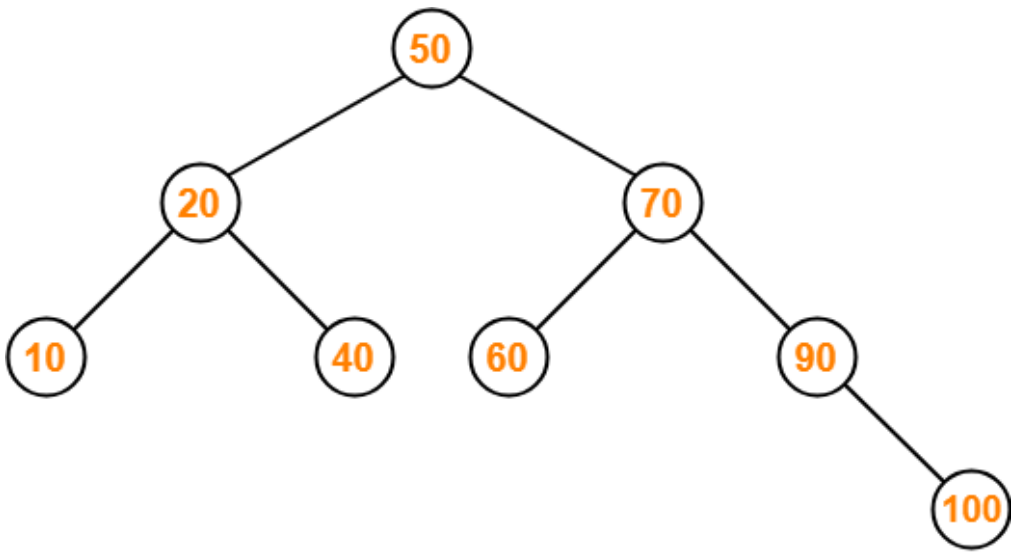
Insert 40-

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



Insert 100-

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.



Binary Search Tree

PRACTICE PROBLEMS BASED ON BINARY SEARCH TREES-

Problem-01:

A binary search tree is generated by inserting in order of the following integers-

50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

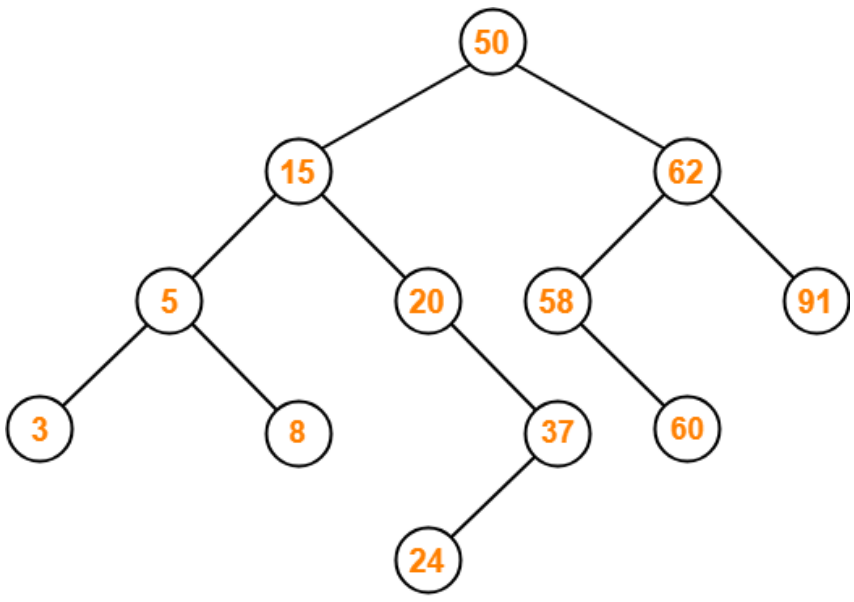
The number of nodes in the left subtree and right subtree of the root respectively is _____.

a) (4, 7)
b) (7, 4)
c) (8, 3)
d) (3, 8)

Solution-

Using the above discussed steps, we will construct the binary search tree.

The resultant binary search tree will be-



Binary Search Tree

Clearly,

- Number of nodes in the left subtree of the root = 7
- Number of nodes in the right subtree of the root = 4

Thus, Option (B) is correct.

Problem-02:

How many distinct binary search trees can be constructed out of 4 distinct keys?

a) 5
b) 14
c) 24
d) 35

Solution-

Number of distinct binary search trees possible with 4 distinct keys

$= {}^{2n}C_n / n+1$

$= {}^{2\times4}C_4 / 4+1$

$= {}^{8}C_4 / 5$

$= 14$

Thus, Option (B) is correct.

Problem-03:

The numbers 1, 2, …, n is inserted in a binary search tree in some order. In the resulting tree, the right subtree of the root contains p nodes. The first number to be inserted in the tree must be-

a) p
b) p+1
c) n-p
d) n-p+1

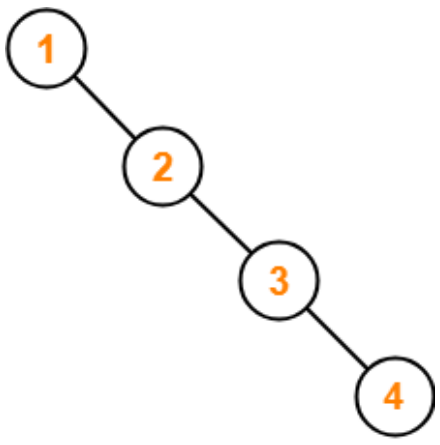
Solution-

Let n = 4 and p = 3.

Then, given options reduce to-

1. 3
2. 4
3. 1
4. 2

Our binary search tree will be as shown-



Binary Search Tree

Clearly, first inserted number = 1.

Thus, Option (C) is correct.

Problem-04:

We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with given set so that it becomes a binary search tree?

a) 0
b) 1
c) n!
d) $\frac{2n_{C_n}}{n+1}$

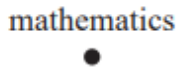
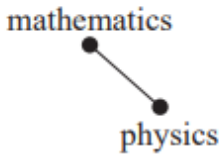
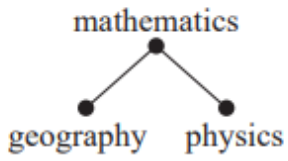
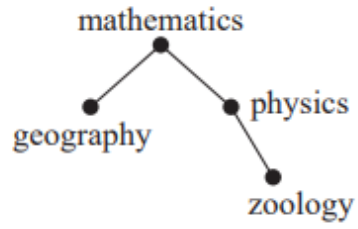
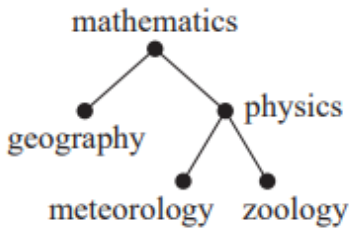
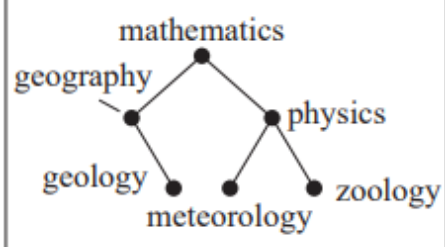
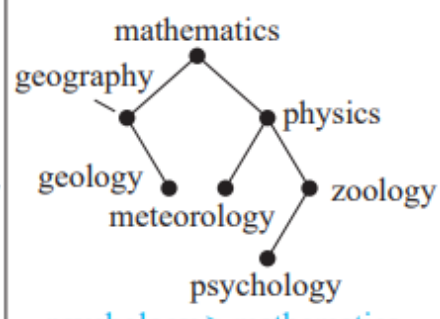
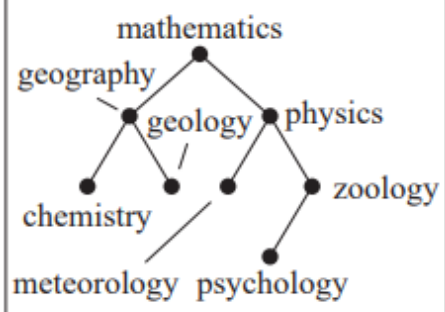
Solution-

Option (B) is correct.

Problem-05:

Form a binary search tree for the words mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetical order).

Solution:

	 physics > mathematics	 geography < mathematics	 zoology > mathematics zoology > physics
 meteorology > mathematics meteorology < physics	 geology < mathematics geology > geography	 psychology > mathematics psychology > physics psychology < zoology	 chemistry < mathematics chemistry < geography

Binary search tree (BST) is a special kind of binary tree where each node contains-

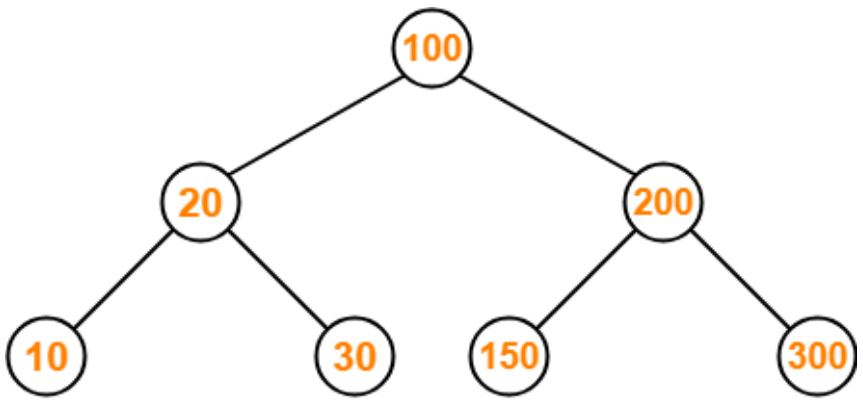
- **Only larger values in its right subtree.**
- **Only smaller values in its left subtree.**

Binary Search Tree Traversal-

- A binary search tree is traversed in exactly the same way a binary tree is traversed.
- In other words, BST traversal is same as binary tree traversal.

Example-

Consider the following binary search tree-



Binary Search Tree

Now, let us write the traversal sequences for this binary search tree-

Preorder Traversal- 100, 20, 10, 30, 200, 150, 300

Inorder Traversal- 10, 20, 30, 100, 150, 200, 300

Postorder Traversal- 10, 30, 20, 150, 300, 200, 100

Important Notes-

Note-01:

- Inorder traversal of a binary search tree always yields all the nodes in increasing order.

Note-02:

Unlike Binary Trees,

- A binary search tree can be constructed using only preorder or only postorder traversal result.
- This is because inorder traversal can be obtained by sorting the given result in increasing order.

Construction of unique Binary tree

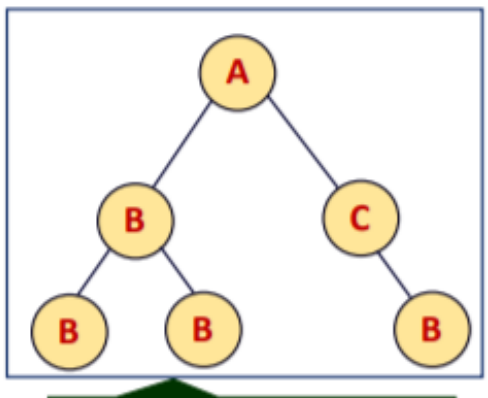
Let us suppose one of your friends wants to send you a binary tree via a network. So how he/she will send the binary tree?

Don't think that just take a snap-shot of that binary tree and send the picture. We are not studying data structure to do that. We have to think optimal.

We cannot just send the picture because this will waste the network bandwidth unnecessarily as sending a picture consume more network bandwidth.

So more optimal way is to write the traversal of binary tree in a text file and then send it. This will save a lot of network bandwidth.

Let us see the difference



Size of this image is approximate 16-18 KB

Inorder Travesal : 1234567

This will take 26 byte when stored in a text file

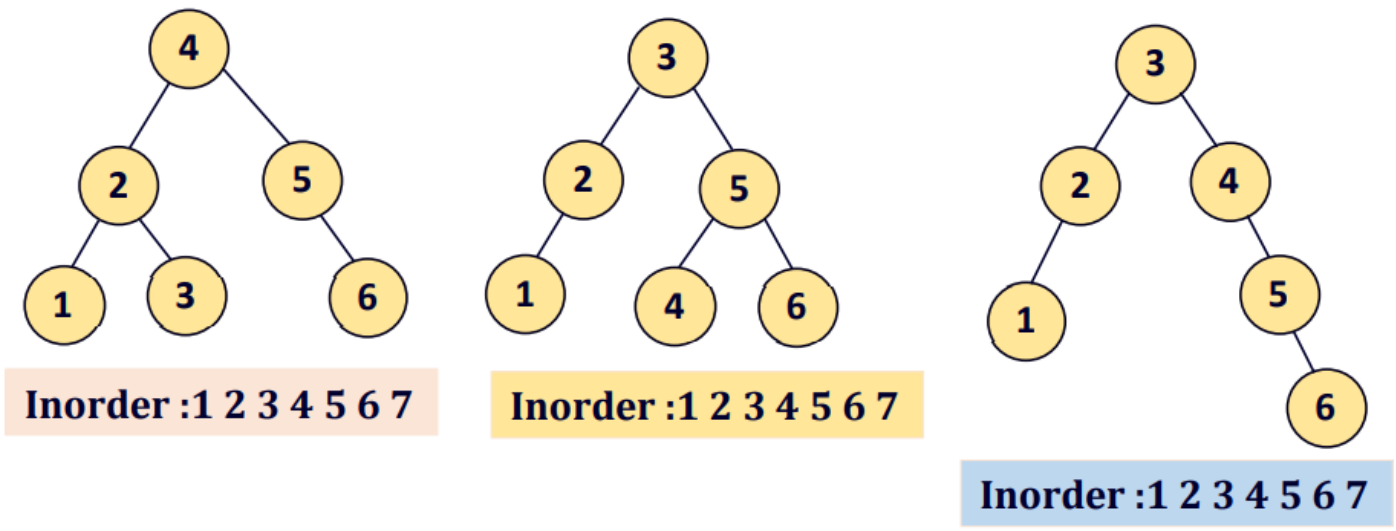
So, your friend must opt the second method which is more optimal. Now let us suppose that you have received the inorder traversal. You don't know how original binary tree look alike. So, can you able to reconstruct the original binary tree? If yes. Then please go ahead.

Actually, we cannot reconstruct original binary tree if we have only inorder traversal available. In general, an inorder traversal does not uniquely define the structure of the tree.

Let us see why we cannot reconstruct original binary tree if we have only inorder traversal available.

Inorder traversal that you have received from your friend is: 1 2 3 4 5 6 7

For this inorder traversal multiple binary trees exist. So, this will cause ambiguity that which binary tree your friend had sent to you.

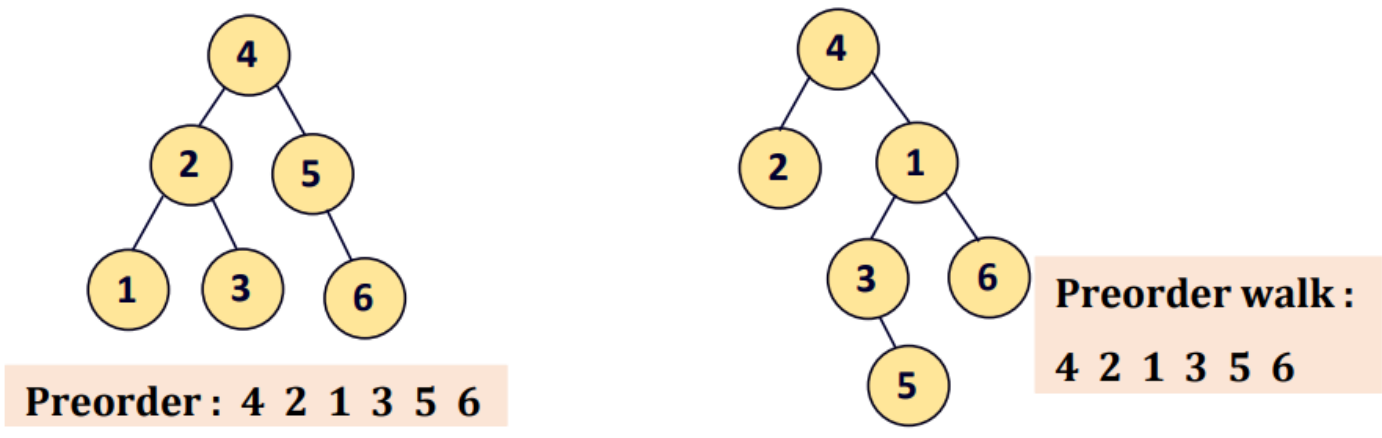


So, which binary tree out of these three, your friend had sent you. This cannot be predicted. The only information you had, is inorder traversal. But more than one binary tree has same in-order traversal.

So unique binary tree cannot be constructed from a given inorder traversal.

Now let us suppose instead of sending inorder traversal, your friend had sent you a preorder traversal. Now you have only the information is preorder traversal as: 4 2 1 3 5 6

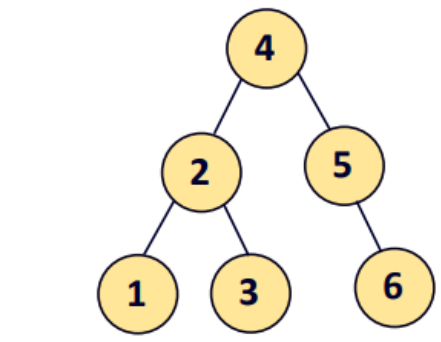
So now can you able to reconstruct the original binary tree? No. let us see why?



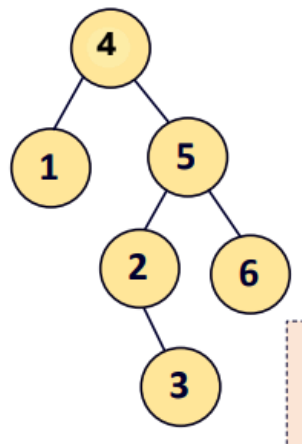
Both the tree have the same pre-order traversal. So, the same ambiguity is also present in case of pre-order traversal.

So unique binary tree cannot be constructed from a given pre-order traversal.

Same ambiguity is present in case of post-order traversal also.



Postorder : 1 3 2 6 5 4



Postorder walk

1 3 2 6 5 4

So, we cannot construct unique binary tree from a given in-order or pre-order or post-order traversal.

To uniquely construct a Binary Tree, In-order traversal together with either Post-order or Pre-order must be given.

We will discuss how to construct a unique binary tree from the given traversals if In-order traversal together with either Post-order or Pre-order must be given.

BST Construction from Given Traversals

As we know we cannot construct unique binary tree if any one out of below traversals are given individually.

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

To construct a unique binary Inorder traversal together with either Postorder or Preorder must be given.

Let's suppose one or two traversals are given, whether we can construct Binary tree or not.

Let's Rephrase.

We can Construct Unique Binary Tree if following are given:-

- **Inorder and Preorder.**
- **Inorder and Postorder.**
- **Inorder and Level-order.**

We cannot Construct Unique Binary tree if following are given:-

- **Pre-order Traversal**
- **Post-order Traversal**
- **Level-order Travesal**
- **In-order Traversal**
- **Postorder and Preorder.**
- **Levelorder and Postorder.**

So, if any one out of four (Pre, Post, in-order or Level) are given, Binary Tree cannot be constructed.

Construction of unique BST from Pre-order and in-order

Let us suppose we have given in-order and pre-order traversal of a binary tree. We don't know how original binary tree look alike. So, we'll try to reconstruct original binary tree from given in-order and pre-order traversal.

Example:

You are given two traversal Inorder and Preorder of Binary search tree as:

Inorder: g d h b e a f j c

Preorder: a b d g h e c f j

What is height of original binary tree?

Solution:

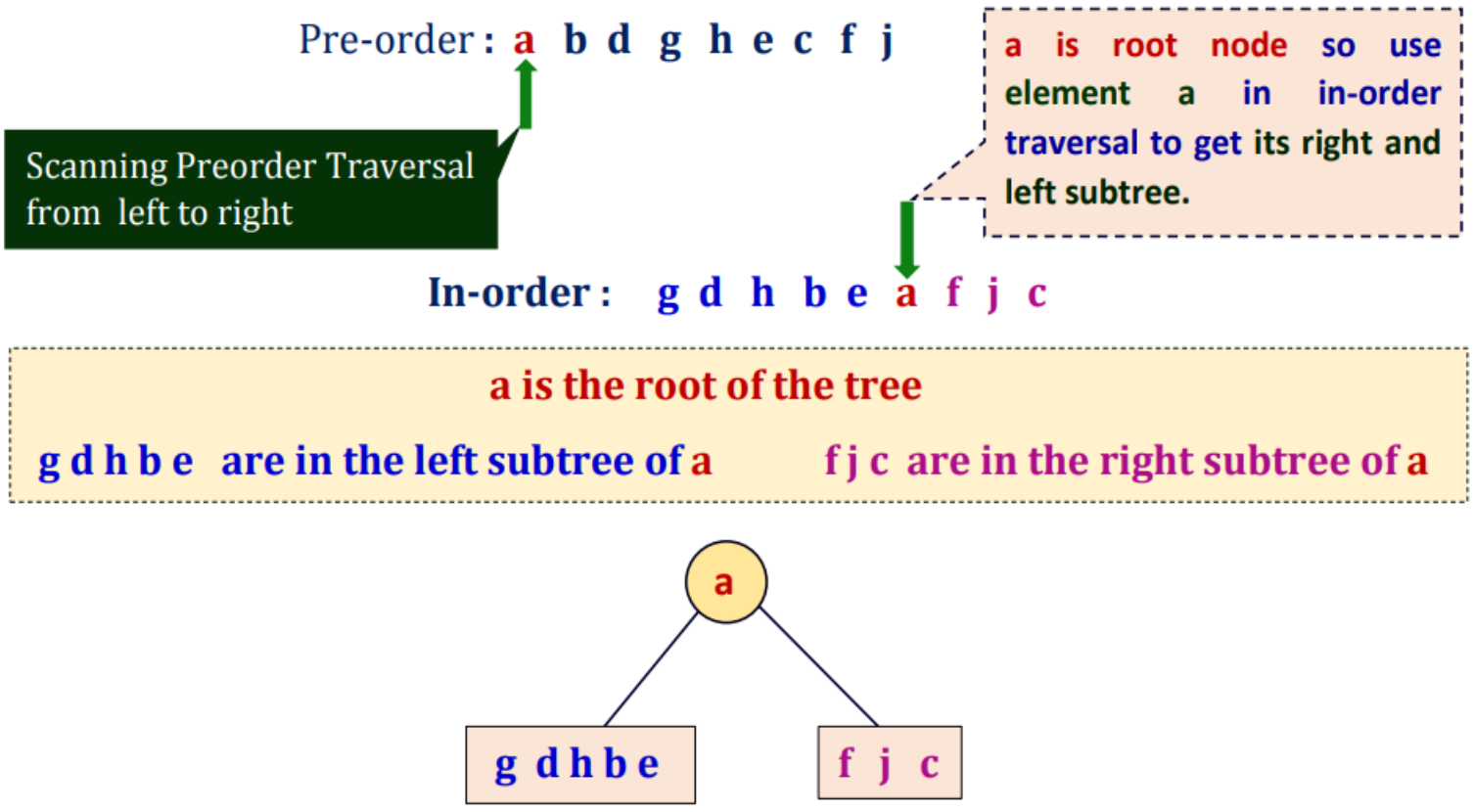
Inorder: g d h b e a f j c

Preorder: a b d g h e c f j

As we know first element in preorder traversal is always a root node. So we got our first clue that a is root tree.

Step 1

Scan the preorder traversal from left to right one element at a time. Use that currently scanned element in in-order traversal to get its left subtree and right subtree.



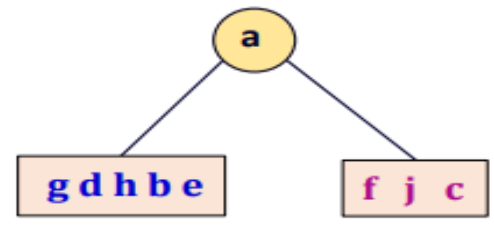
Step 2

Next element under scan is b. Use currently scanned **element b** in left sub-tree of a to get further its left sub-tree and right sub-tree.

Preorder : **a b d g h e c f j**

Current element under scan i.e **element b**

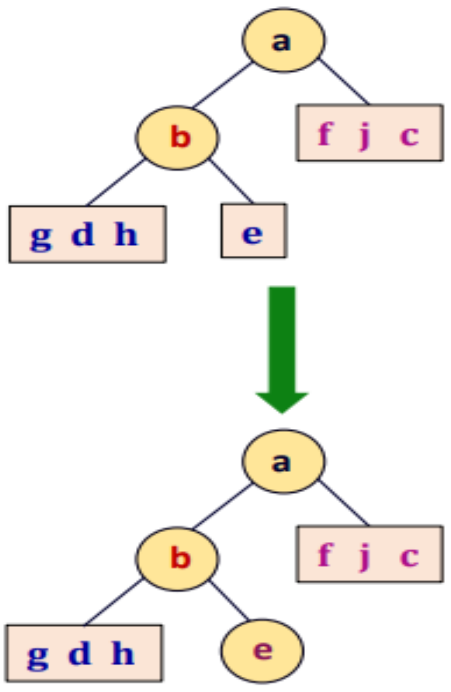
b is root node in the left subtree of a.



Left Subtree of a : g d h b e

b is the root

g d h are in the left subtree of b **e i is in the right subtree of b**

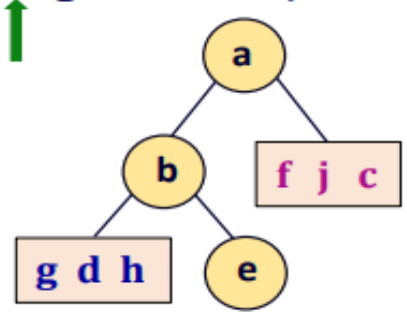


Step 3

Next element under scan is d. Use currently scanned **element d** in left sub-tree of b to get further its left sub-tree and right sub-tree.

Preorder : **a b d g h e c f j**

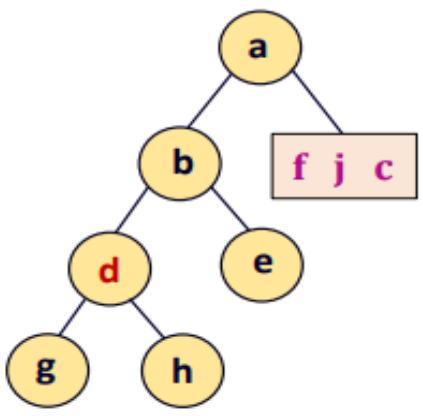
d is root node in the left subtree of b.



Left Subtree of b : g d h

d is the root

g is in the left subtree of d **h is in the right subtree of d**

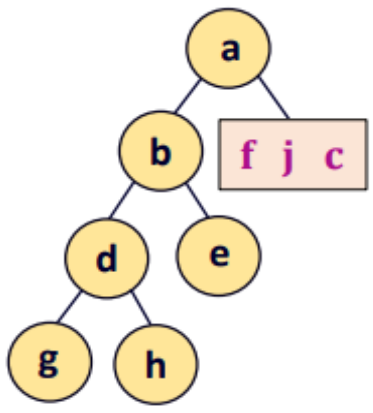


Step 4

Next scanned element is g , and **g has no left or right child.** So stop and go for scanning next element.

Preorder : **a b d g h e c f j**

g is a single value node. Node g has no left or right child. So move to next step.

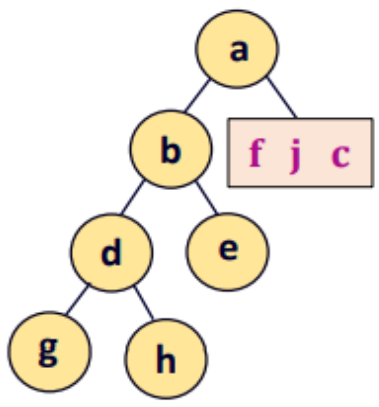


Step 5,6

Next scanned elements are h and e. Element h and e has no left or right child. So stop and go for scanning next element.

In Step 5 Preorder : **a b d g h e c f j**

In Step 6 Preorder : **a b d g h e c f j**



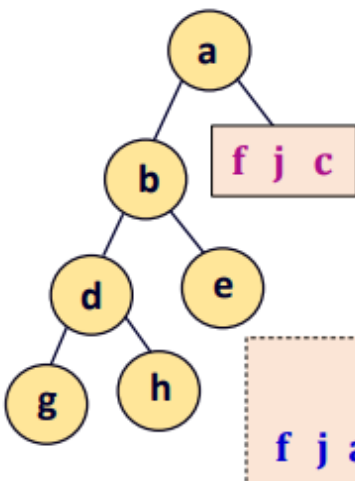
Step 7

Next element under scan is c. Use currently scanned **element c** in right sub-tree of a to get further its left sub-tree and right sub-tree.

Preorder : **a b d g h e c f j**

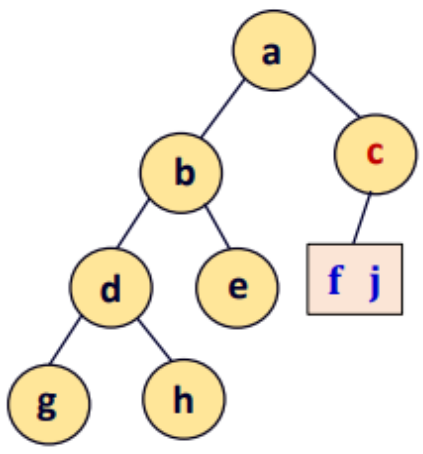
c is root node in the right subtree of a.

Right Subtree of a : f j c



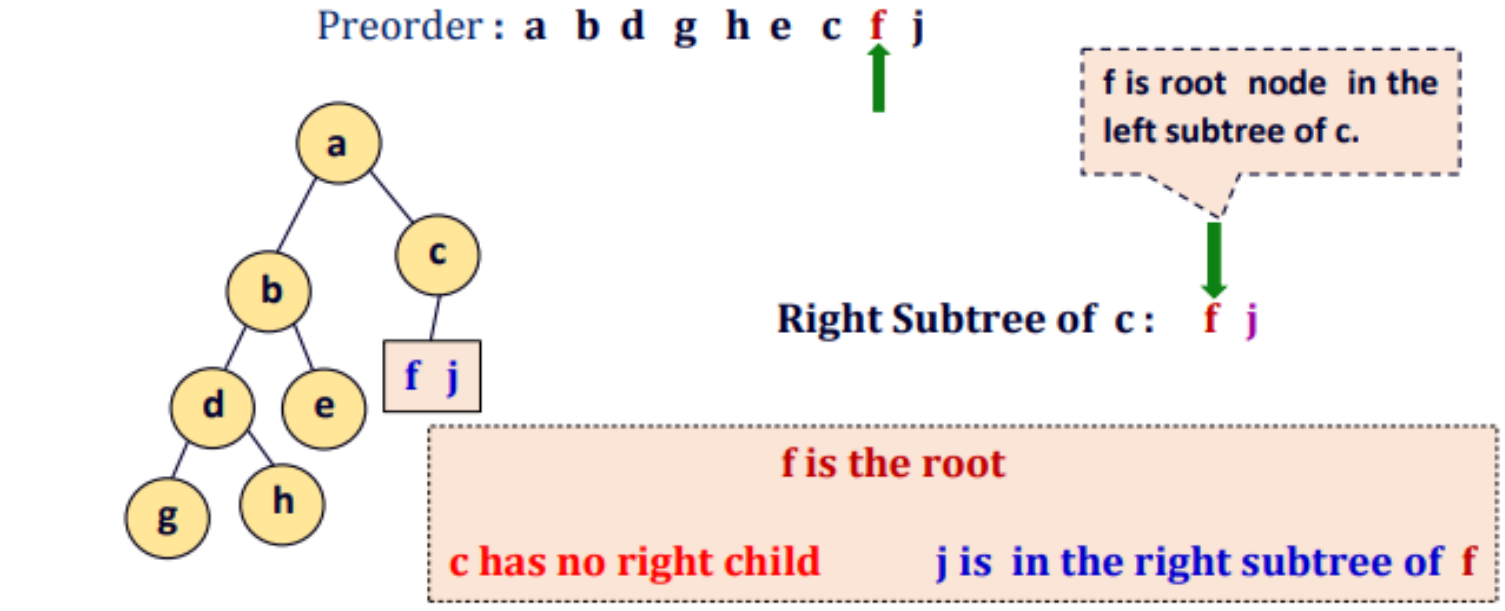
c is the root

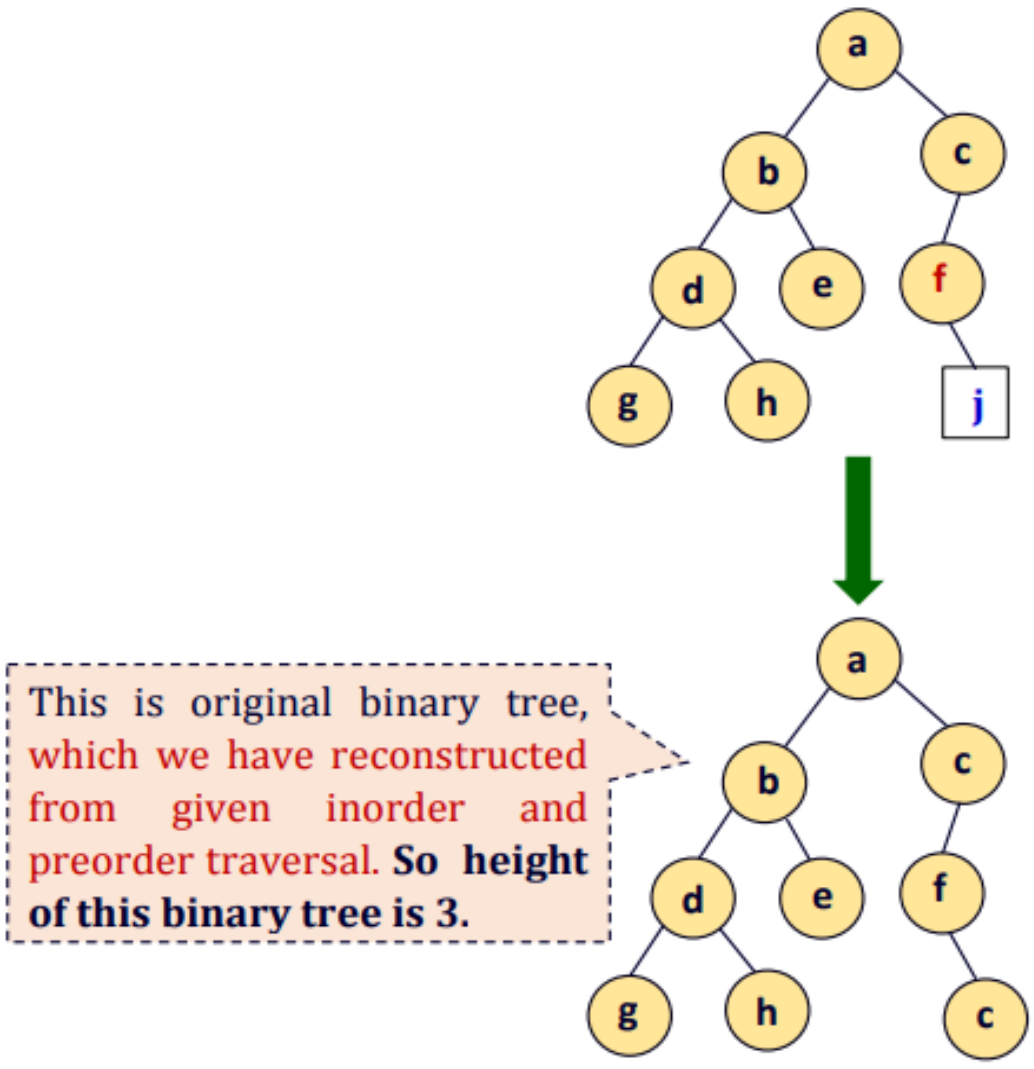
f j are in the left subtree of c **c has no right subtree**



Step 8

Next element under scan is f. Use currently scanned **element f** in left sub-tree of c to get further its left sub-tree and right sub-tree(if any).





Example 2:

Construct a binary tree whose preorder traversal is 5 7 4 8 6 1 9 2 3 and inorder traversal is 4 7 5 6 1 8 2 9 3. The post-order traversal of the binary tree is?

(A) 4 7 1 6 2 3 9 8 5

(B) 4 7 1 6 2 3 9 8 5

(C) 4 7 1 6 2 3 9 8 5

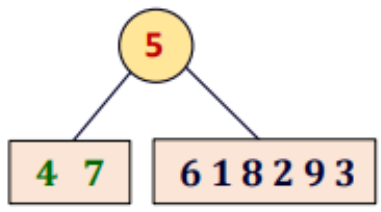
(D) 4 7 1 6 2 3 9 8 5

Solution:

Step 1

Preorder traversal : 5 7 4 8 6 1 9 2 3

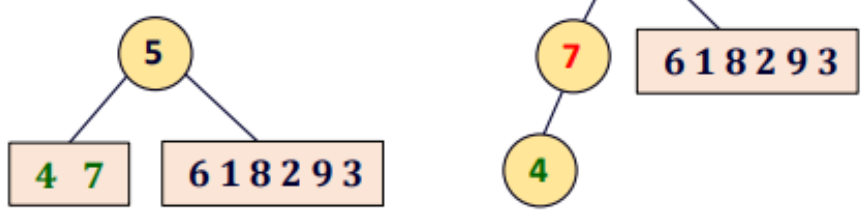
Inorder traversal : 4 7 5 6 1 8 2 9 3



Step 2

Preorder traversal : 5 7 4 8 6 1 9 2 3

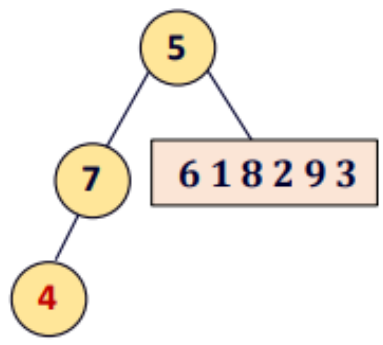
Left subtree of 5: 4 7



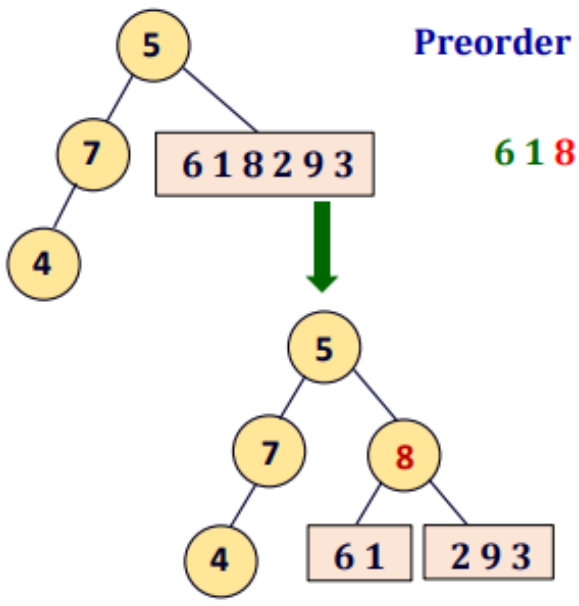
Step 3

Preorder traversal : 5 7 4 8 6 1 9 2 3

Node 4 don't have any child (right or left)



Step 4

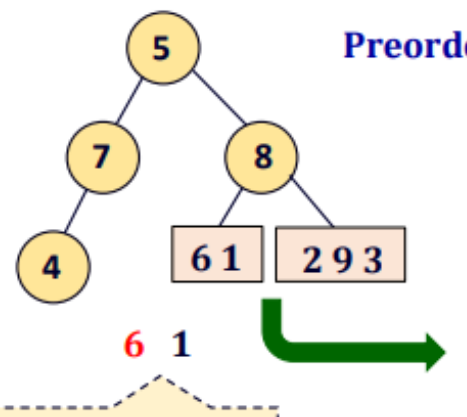


Preorder traversal : 5 7 4 8 6 1 9 2 3

6 1 8 2 9 3

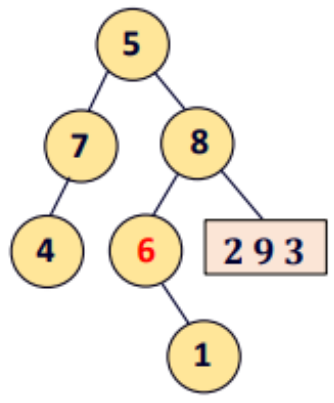
Right subtree of node 5. So 8 become the root in Right Subtree of node 5

Step 5



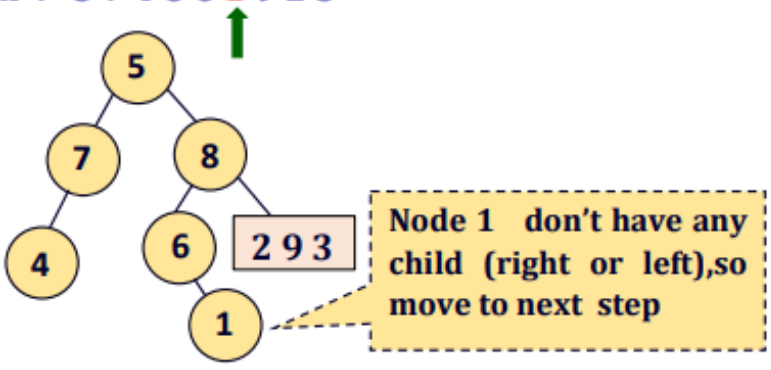
Preorder traversal : 5 7 4 8 6 1 9 2 3

Left subtree of node 8. So 6 become the root in left Subtree of node 8

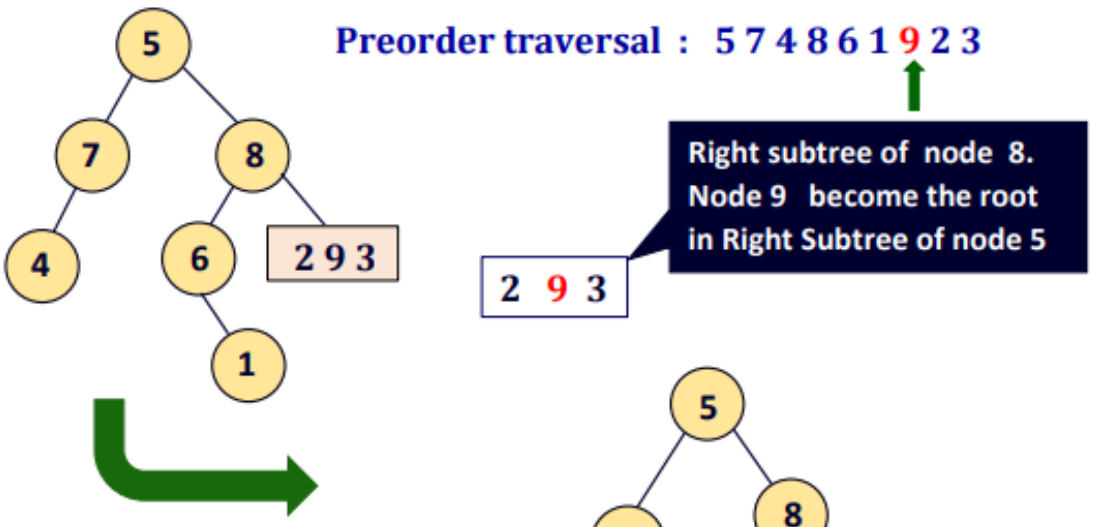


Step 6

Preorder traversal : 5 7 4 8 6 1 9 2 3



Step 7



This is final binary tree which we have reconstructed from given inorder and preorder.

So post-order traversal is given as:

4 7 1 6 2 3 9 8 5

Example:

The inorder traversal of a binary tree is d b e a f c g and pre-order traversal of same binary tree is as: a b d e c f g. The post-order traversal of the binary tree is

(A) d e b f g c a

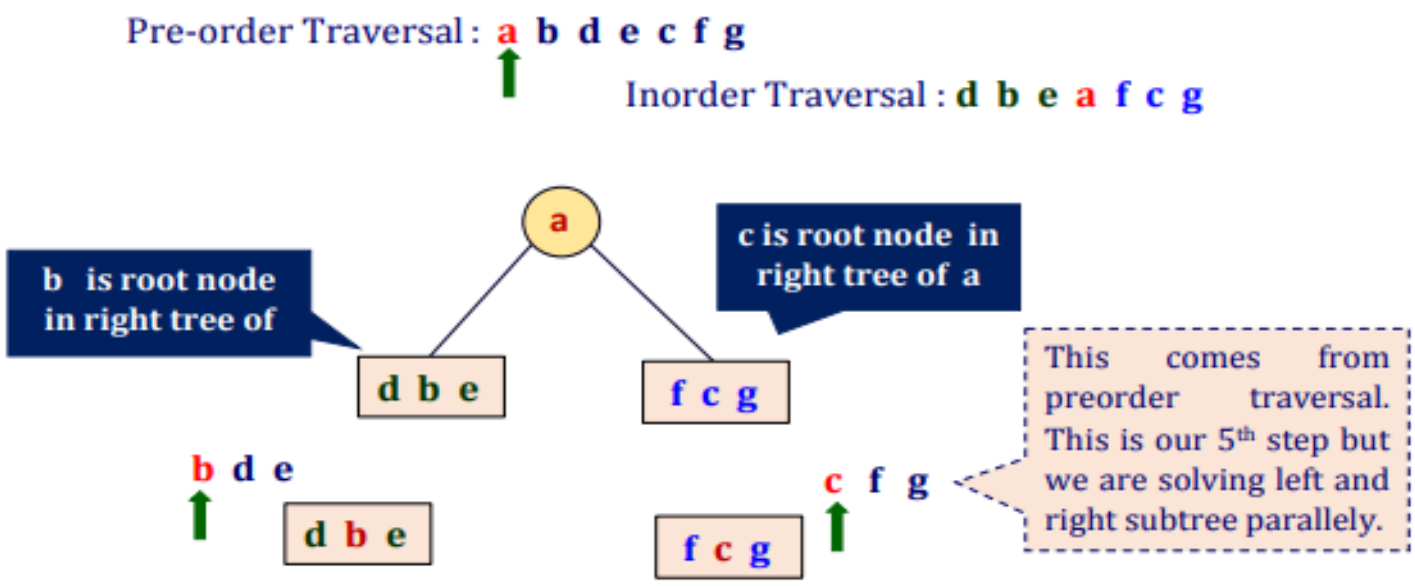
(B) e d b g f c a

(C) e d b f g c a

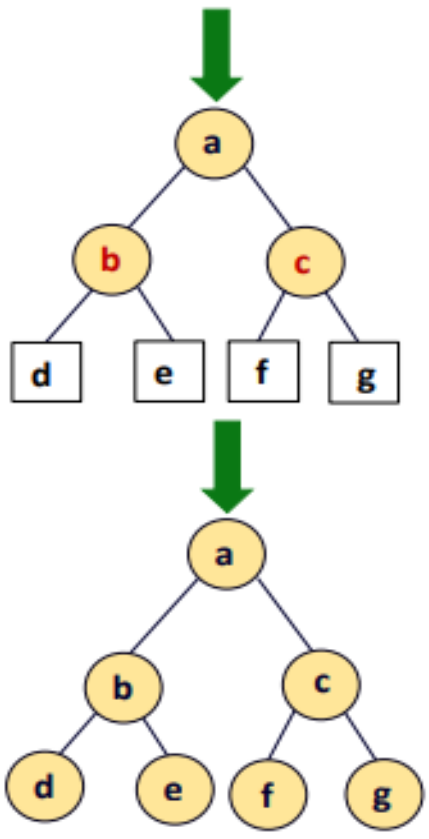
(D) d e f g b c a

Solution:

First Try to solve this Example by yourself. Don't worry its solution is also given. We will follow the same approach to solve this problem also as we have solved above two problems but for a competition point of view, we have to choose a fast approach to solve any king problem. So, see how to do it fast.



Right subtree of **node a** contain **three element f c and g.** Now for these three elements, write the same element from inorder traversal in the order as they present in inorder traversal. (Both for left and right subtree of a)



Post-order traversal of above binary tree : d e b f g c a

So correct option is A

Construction of unique BST from Postorder and inorder

Construction of binary tree from given Postorder and inorder traversal is same as that constructing binary tree from preorder and inorder, but in case of postorder traversal we scan the given traversal from right to left and in case of preorder we scan the given traversal from left to right. So, I will not discuss in much detail. We will solve the right and left subtree in parallel. I hope you will not face any problem.

Example:

You are given two traversal Inorder and Postorder of Binary tree as: Inorder: 3 2 8 5 7 9 6 4 1 Postorder: 2 3 5 8 6 1 4 9 7 Binary search trees with its Preorder traversal is?

(A) d e b f g c a

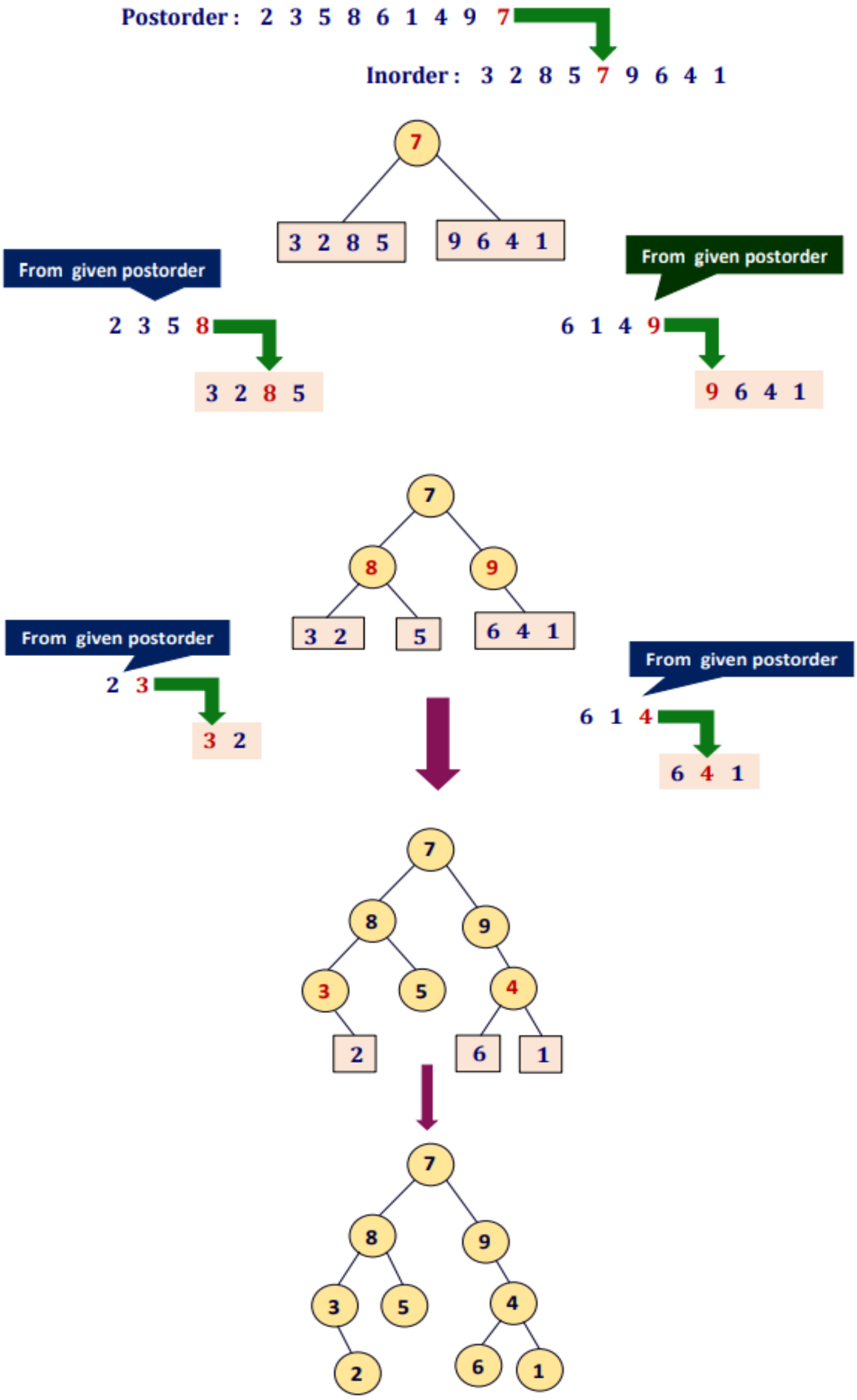
(B) e d b g f c a

(C) e d b f g c a

(D) d e f g b c a

Solution:

As we know last element in preorder traversal is always a root node. So, we got our first clue that a is root tree.



Preorder traversal is: 7 8 3 2 5 9 4 6 1

✓ So the correct option is B

PRACTICE PROBLEMS BASED ON BINARY SEARCH TREE (BST) Traversal-

Problem-01:

Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers.

What is the inorder traversal sequence of the resultant tree?

a) 7, 5, 1, 0, 3, 2, 4, 6, 8, 9
b) 0, 2, 4, 3, 1, 6, 5, 9, 8, 7
c) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
d) 9, 8, 6, 4, 2, 3, 0, 1, 5, 7

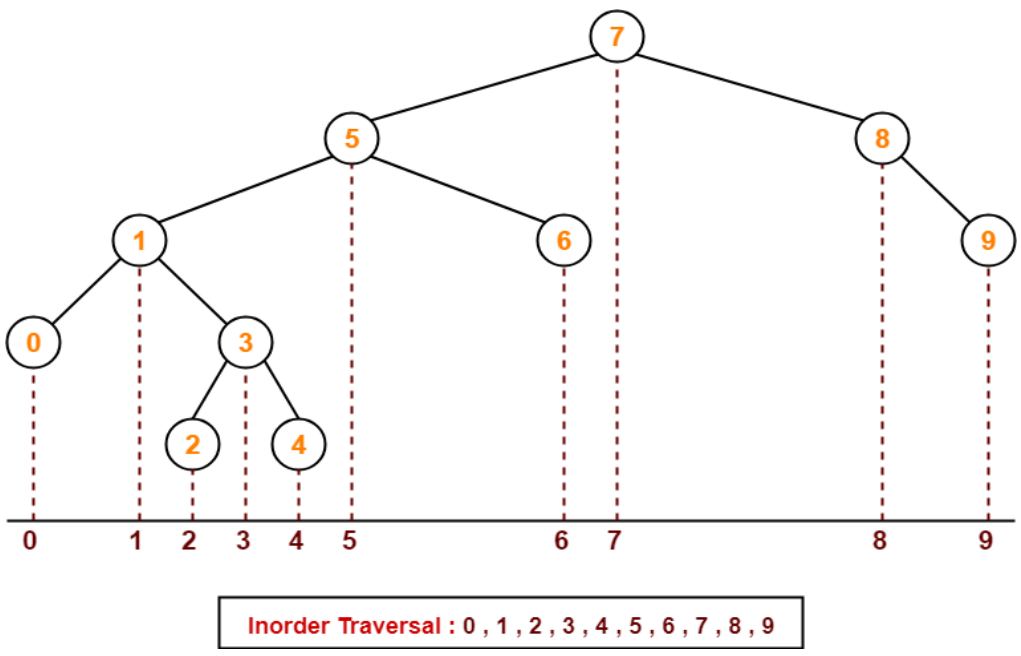
Solution-

This given problem may be solved in the following two ways-

Method-01:

- We construct a binary search tree for the given elements.
- We write the inorder traversal sequence from the binary search tree so obtained.

Following these steps, we have-



Thus, Option (C) is correct.

Method-02:

We know, inorder traversal of a binary search tree always yields all the nodes in increasing order.

Using this result,

- We arrange all the given elements in increasing order.
- Then, we report the sequence so obtained as inorder traversal sequence.

Inorder Traversal:

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

Thus, Option (C) is correct.

Problem-02:

The preorder traversal sequence of a binary search tree is-

30, 20, 10, 15, 25, 23, 39, 35, 42

What one of the following is the postorder traversal sequence of the same tree?

a) 10, 20, 15, 23, 25, 35, 42, 39, 30
b) 15, 10, 25, 23, 20, 42, 35, 39, 30
c) 15, 20, 10, 23, 25, 42, 35, 39, 30
d) 15, 10, 23, 25, 20, 35, 42, 39, 30

Solution-

In this question,

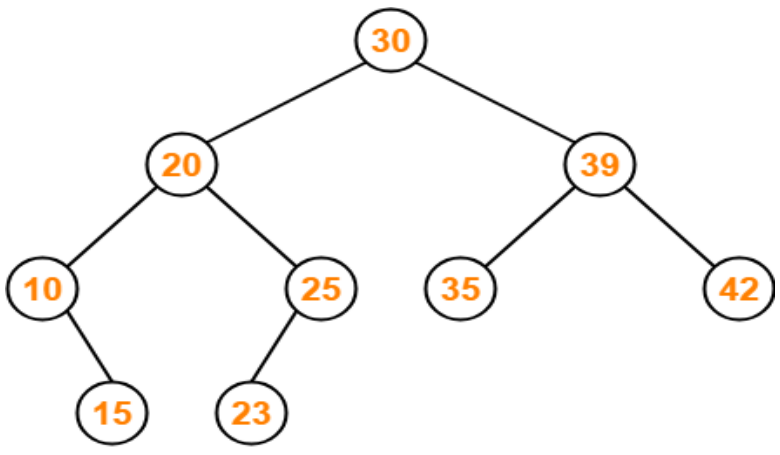
- We are provided with the preorder traversal sequence.
- We write the inorder traversal sequence by arranging all the numbers in ascending order.

Then-

- Preorder Traversal: 30, 20, 10, 15, 25, 23, 39, 35, 42
- Inorder Traversal: 10, 15, 20, 23, 25, 30, 35, 39, 42

Now,

- We draw a binary search tree using these traversal results.
- The binary search tree so obtained is as shown-



Binary Search Tree

Now, we write the postorder traversal sequence-

Postorder Traversal:

15, 10, 23, 25, 20, 35, 42, 39, 30

Thus, Option (D) is correct

Binary Search Tree Operations-

Commonly performed operations on binary search tree are-

1. Search Operation
2. Insertion Operation
3. Deletion Operation

1. Search Operation-

Search Operation is performed to search a particular element in the Binary Search Tree.

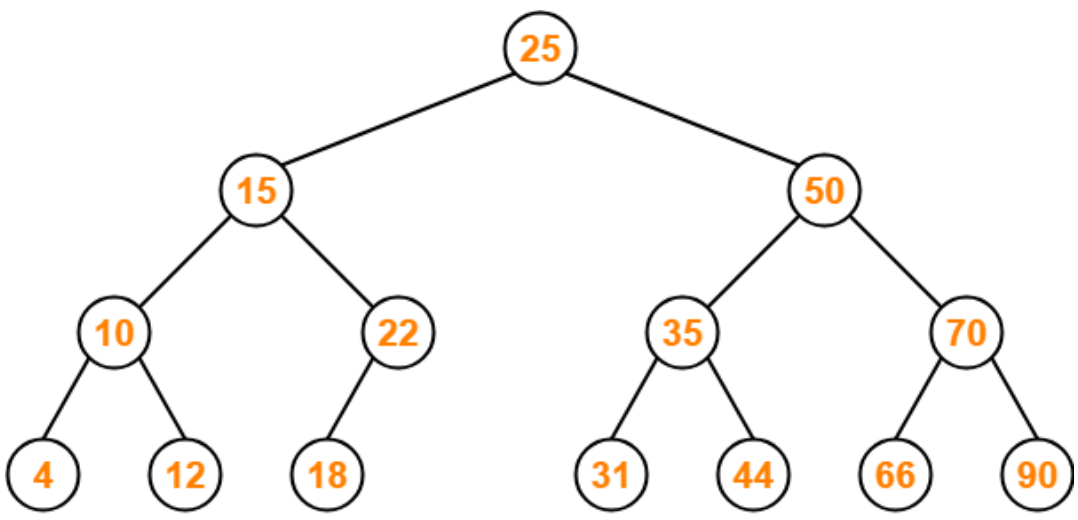
Rules-

For searching a given key in the BST,

- Compare the key with the value of root node.
- If the key is present at the root node, then return the root node.
- If the key is greater than the root node value, then recur for the root node's right subtree.
- If the key is smaller than the root node value, then recur for the root node's left subtree.

Example-

Consider key = 45 has to be searched in the given BST-



Binary Search Tree

- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

2. Insertion Operation-

Insertion Operation is performed to insert an element in the Binary Search Tree.

Rules-

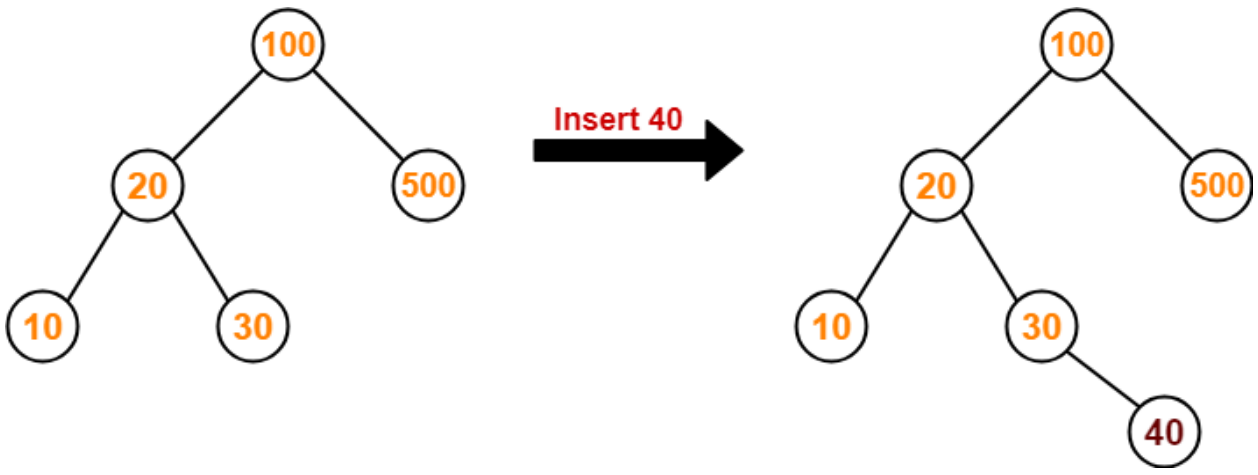
The insertion of a new key always takes place as the child of some leaf node.

For finding out the suitable leaf node,

- Search the key to be inserted from the root node till some leaf node is reached.
- Once a leaf node is reached, insert the key as child of that leaf node.

Example-

Consider the following example where key = 40 is inserted in the given BST-



- We start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

3. Deletion Operation-

Deletion Operation is performed to delete a particular element from the Binary Search Tree.

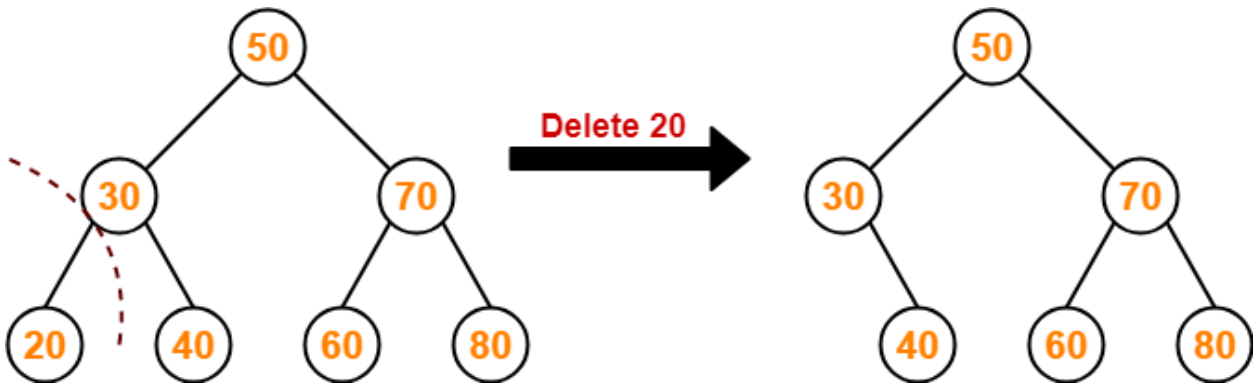
When it comes to deleting a node from the binary search tree, following three cases are possible-

Case-01: Deletion of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to deleted from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-

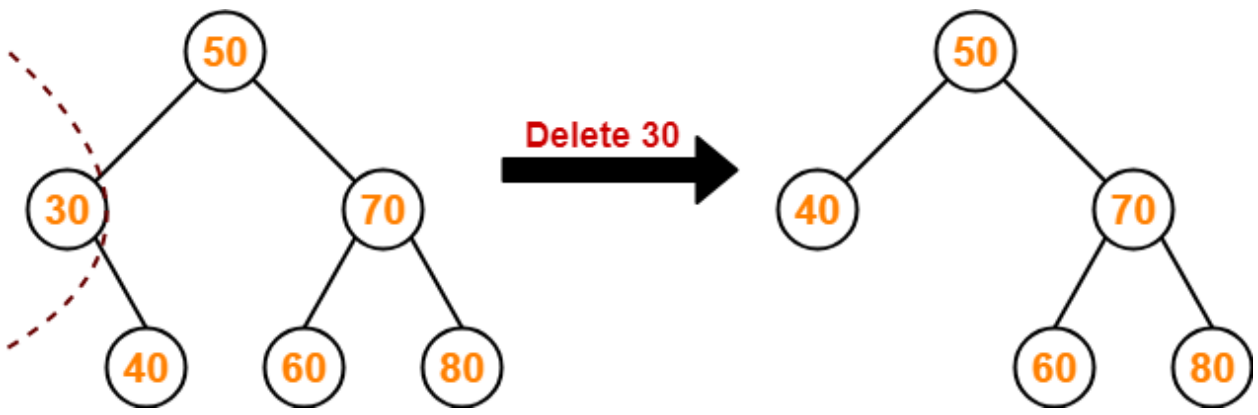


Case-02: Deletion Of A Node Having Only One Child-

Just make the child of the deleting node, the child of its grandparent.

Example-

Consider the following example where node with value = 30 is deleted from the BST-



Case-03: Deletion of A Node Having Two Children-

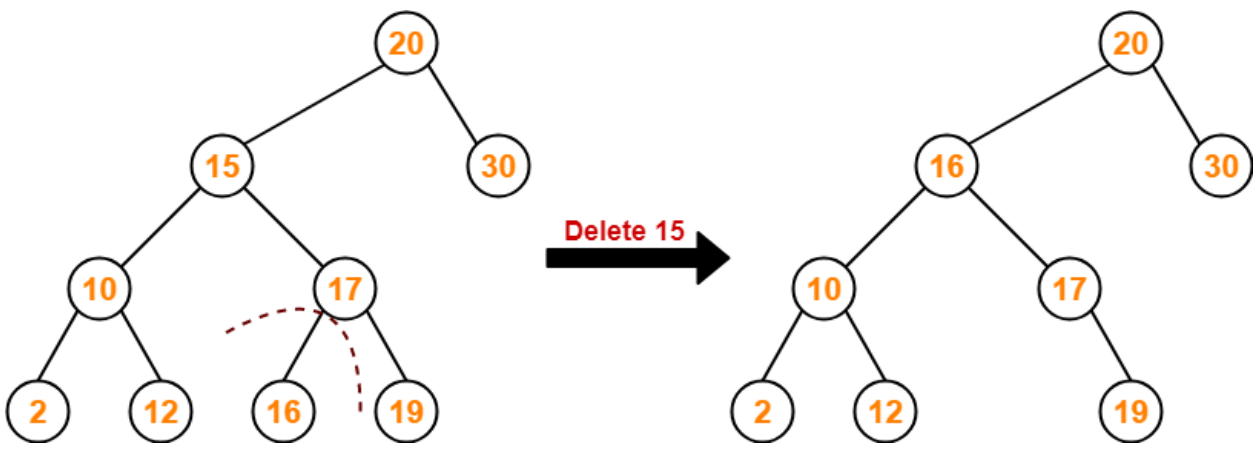
A node with two children may be deleted from the BST in the following two ways-

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-

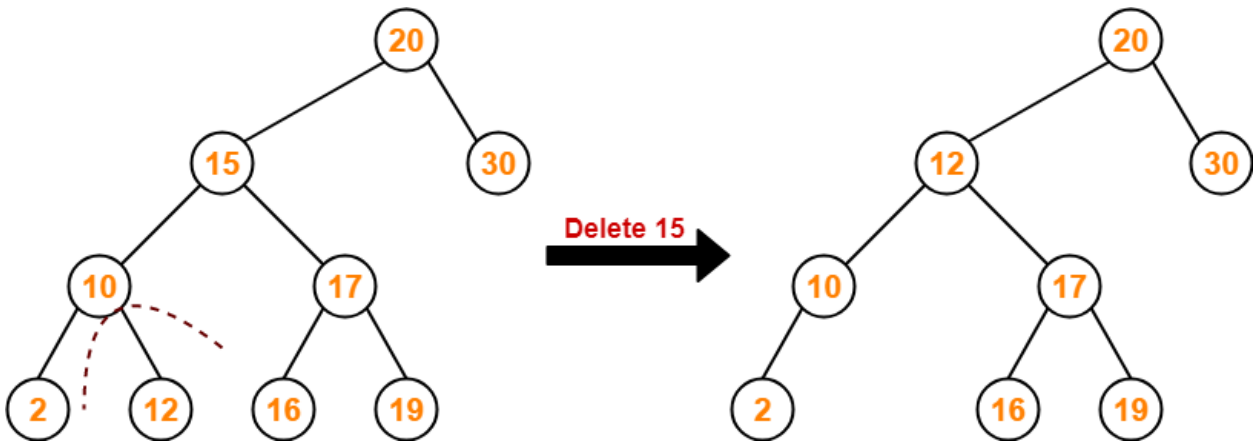


Method-02:

- Visit to the left subtree of the deleting node.
- Pluck the greatest value element called as inorder predecessor.
- Replace the deleting element with its inorder predecessor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-



Time Complexity-

- Time complexity of all BST Operations = O(h).
- Here, h = Height of binary search tree

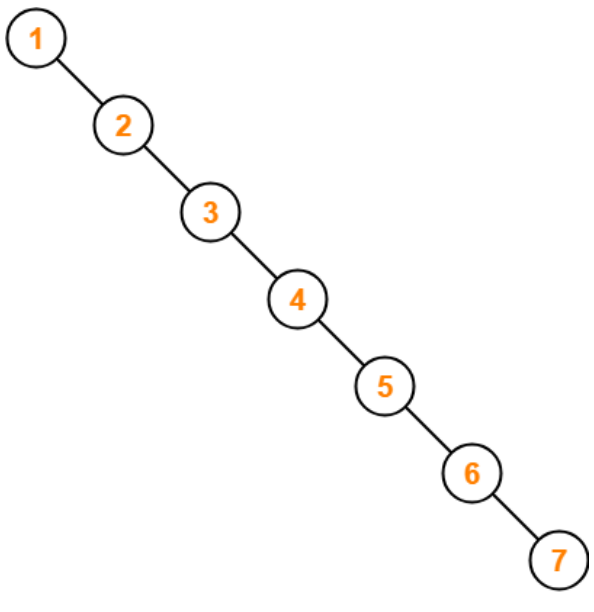
Now, let us discuss the worst case and best case.

Worst Case-

In worst case,

- The binary search tree is a skewed binary search tree.
- Height of the binary search tree becomes n.
- So, Time complexity of BST Operations = O(n).

In this case, binary search tree is as good as unordered list with no benefits.

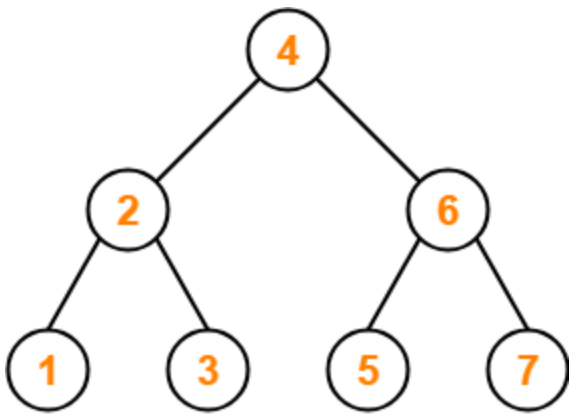


Skewed Binary Search Tree

Best Case-

In best case,

- The binary search tree is a balanced binary search tree.
- Height of the binary search tree becomes log(n).
- So, Time complexity of BST Operations = O(logn).



Balanced Binary Search Tree

Spanning Trees and a Shortest Path Algorithm

What is a Spanning Tree?

A spanning tree for a graph G is a subgraph of G that contains every vertex of G and is a tree.

Given an undirected and connected graph G = (V, E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

Question:

Find all spanning trees for the graph G pictured below.

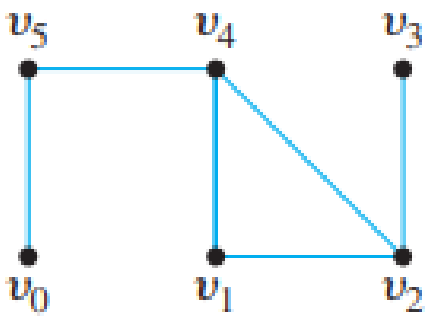


Figure: 12

Solution:

The graph G has one circuit v2v1v4v2, and removing any edge of the circuit gives a tree. Thus, as shown below, there are three spanning trees for G.

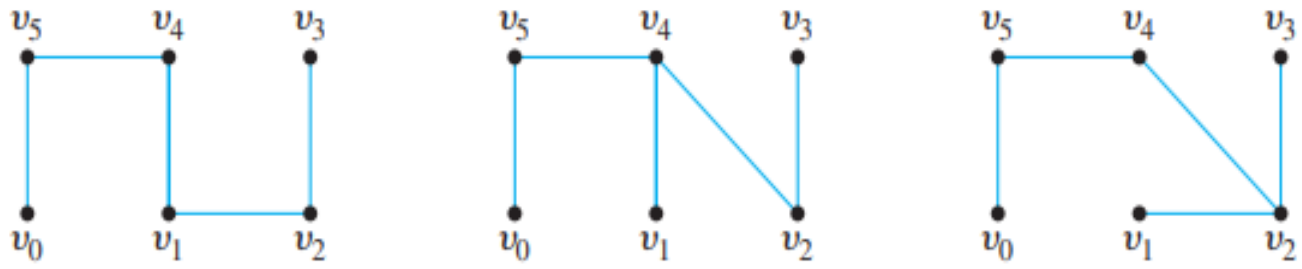


Figure: 13

Depth First Search:

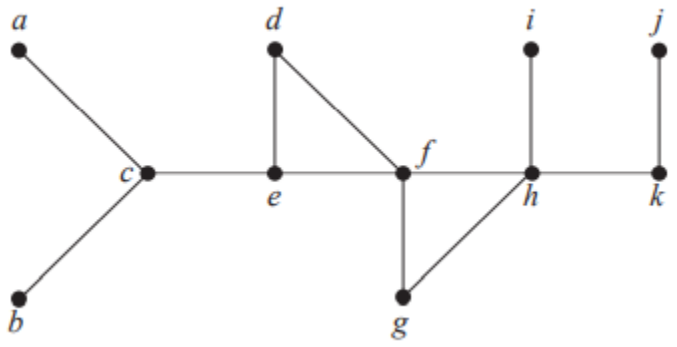
DFS starts the traversal from the root node and visits nodes as far as possible from the root node (i.e., depth wise).

Idea:

1. mark all vertices as unvisited
2. start at an arbitrary vertex
3. go to an unvisited neighbor
4. repeat until you have seen all unvisited neighbors
5. go back

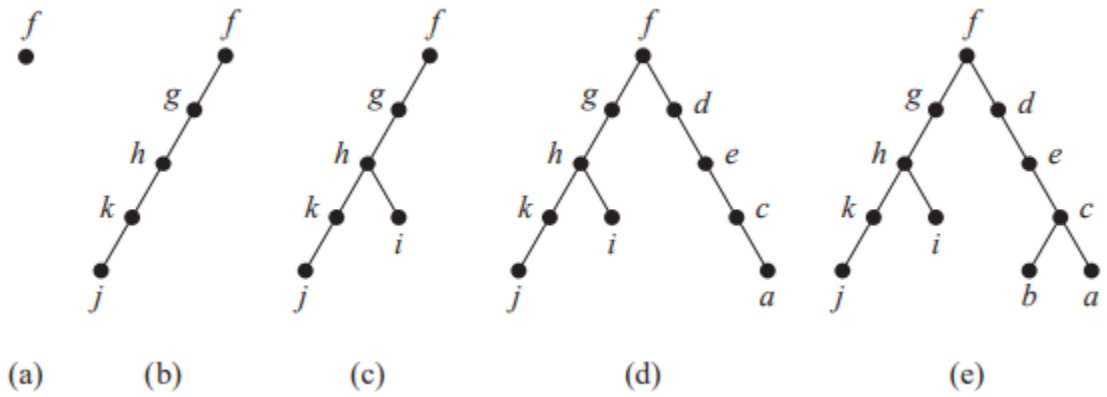
This approach is called a **depth first search**.

Consider the following graph:

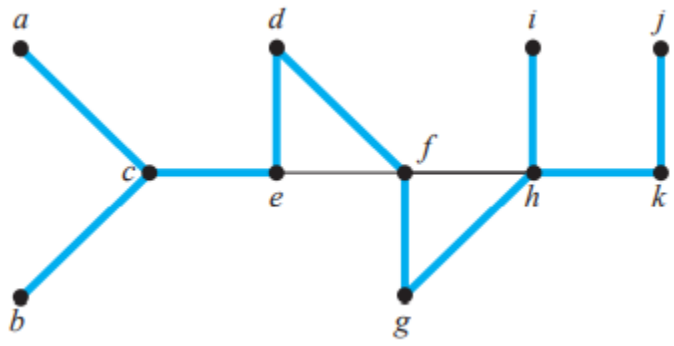


A depth first search proceeds as follows:

1. start at (for example) f
2. visit f, g, h, k, j
3. backtrack to k
4. backtrack to h
5. visit i.
6. backtrack to h
7. backtrack to f
8. visit f, d, e, c, a
9. backtrack to c
10. visit c, b



Notice that this produces a spanning tree, since all nodes are visited and there are no cycles (since having a cycle would mean visiting an already-visited node).



ALGORITHM 1 Depth-First Search.

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
$T :=$ tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(v: vertex of G)
for each vertex w adjacent to v and not yet in T
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

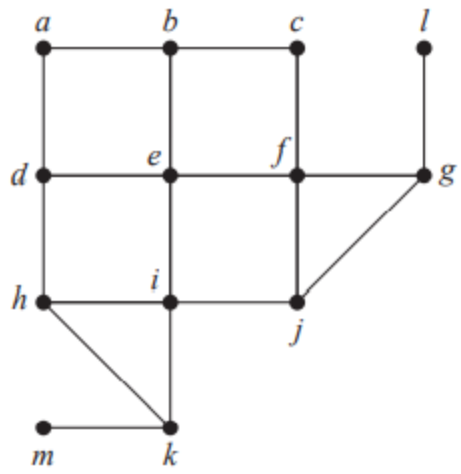
Breadth-First Search

BFS starts traversal from the root node and visits nodes in a level-by-level manner (i.e., visiting the ones closest to the root first).

Instead of always going as deep as possible (as in depth first search), we can try to explore gradually at specific distances from the starting vertex. Such a search is called a **breadth first search** and proceeds as follows:

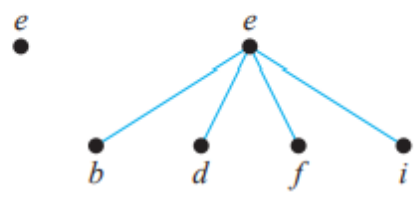
1. keep a list of vertices seen so far, initially containing an arbitrary vertex
2. add all adjacent vertices to the end of the list
3. take the first vertex off the list and visit it
4. add its unvisited neighbors to the end of the list
5. repeat previous two steps until list is empty

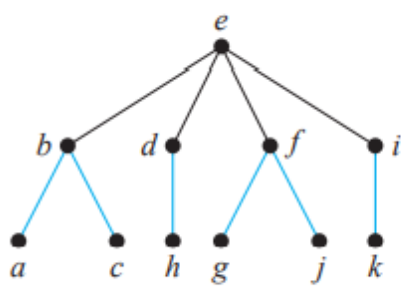
Consider the following graph:

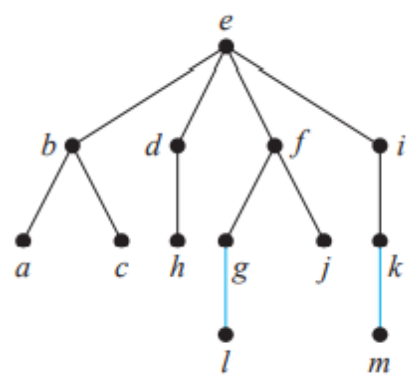


A breadth first search proceeds as follows:

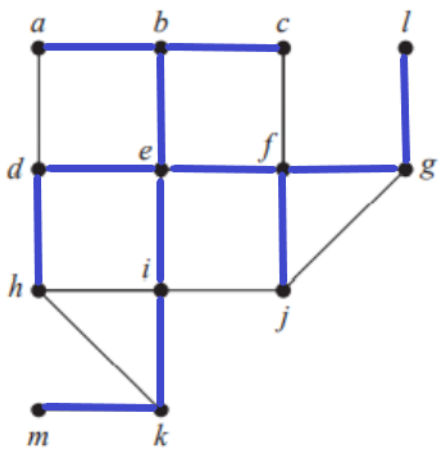
1. start at (for example) e
2. visit b, d, f, i (the vertices at distance 1 from e)
3. visit a, c, h, j, g, k (the vertices at distance 2 from e)
4. visit l, m (the vertices at distance 3 from e)







The process also produces a spanning tree. In fact, this also gives the path with the fewest edges from the start node to any other node. This is the same as the shortest path if all edges are the same length or have the same cost.



ALGORITHM 2 Breadth-First Search.

procedure *BFS* (G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
$T :=$ tree consisting only of vertex v_1
$L :=$ empty list
put v_1 in the list L of unprocessed vertices
while L is not empty
  remove the first vertex, v, from L
  for each neighbor w of v
    if w is not in L and not in T **then**
      add w to the end of the list L
      add w and edge $\{v, w\}$ to T

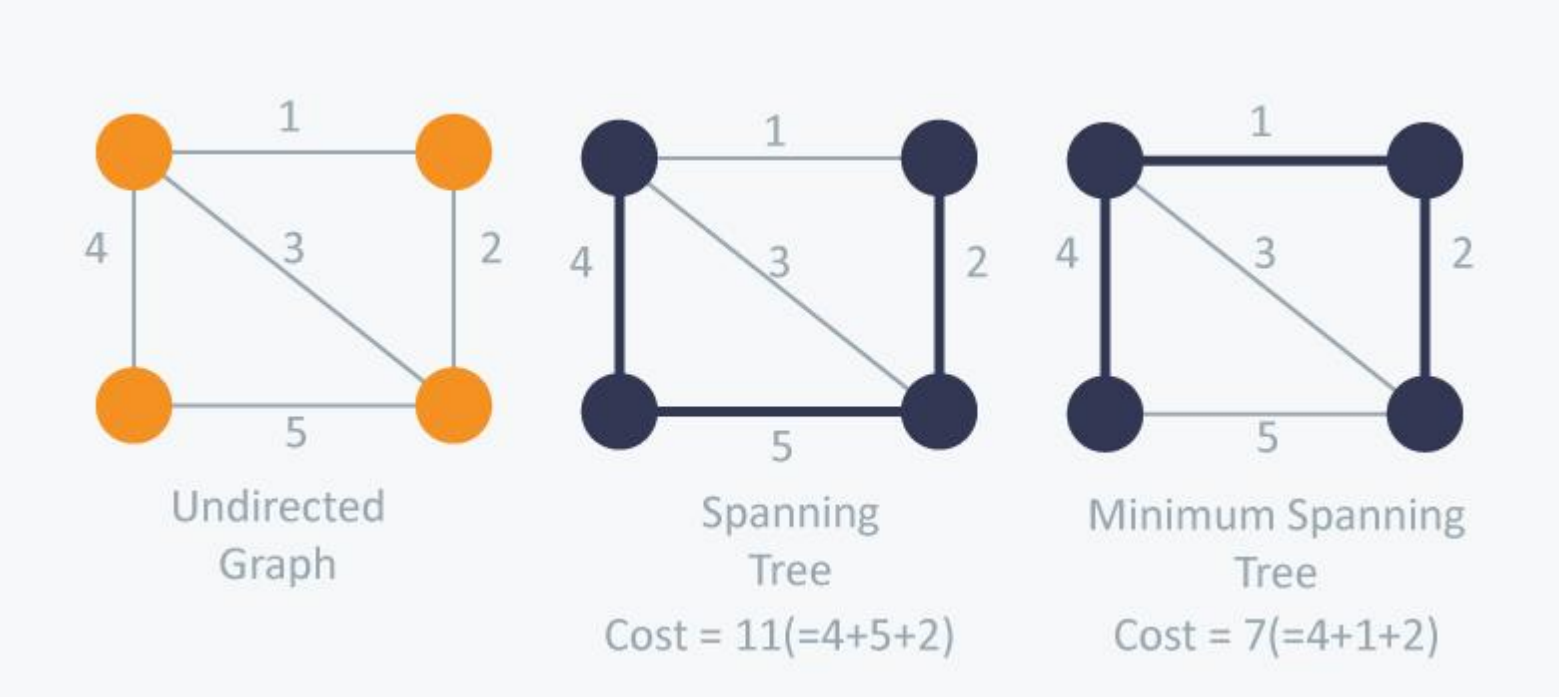
What is a Minimum Spanning Tree?

A minimum spanning tree for a connected, weighted graph is a spanning tree that has the least possible total weight compared to all other spanning trees for the graph.

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle, edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So, the best solution is **"Disjoint Sets"**: Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
  e := any edge in G with smallest weight that does not form a simple circuit
    when added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

Consider following example:

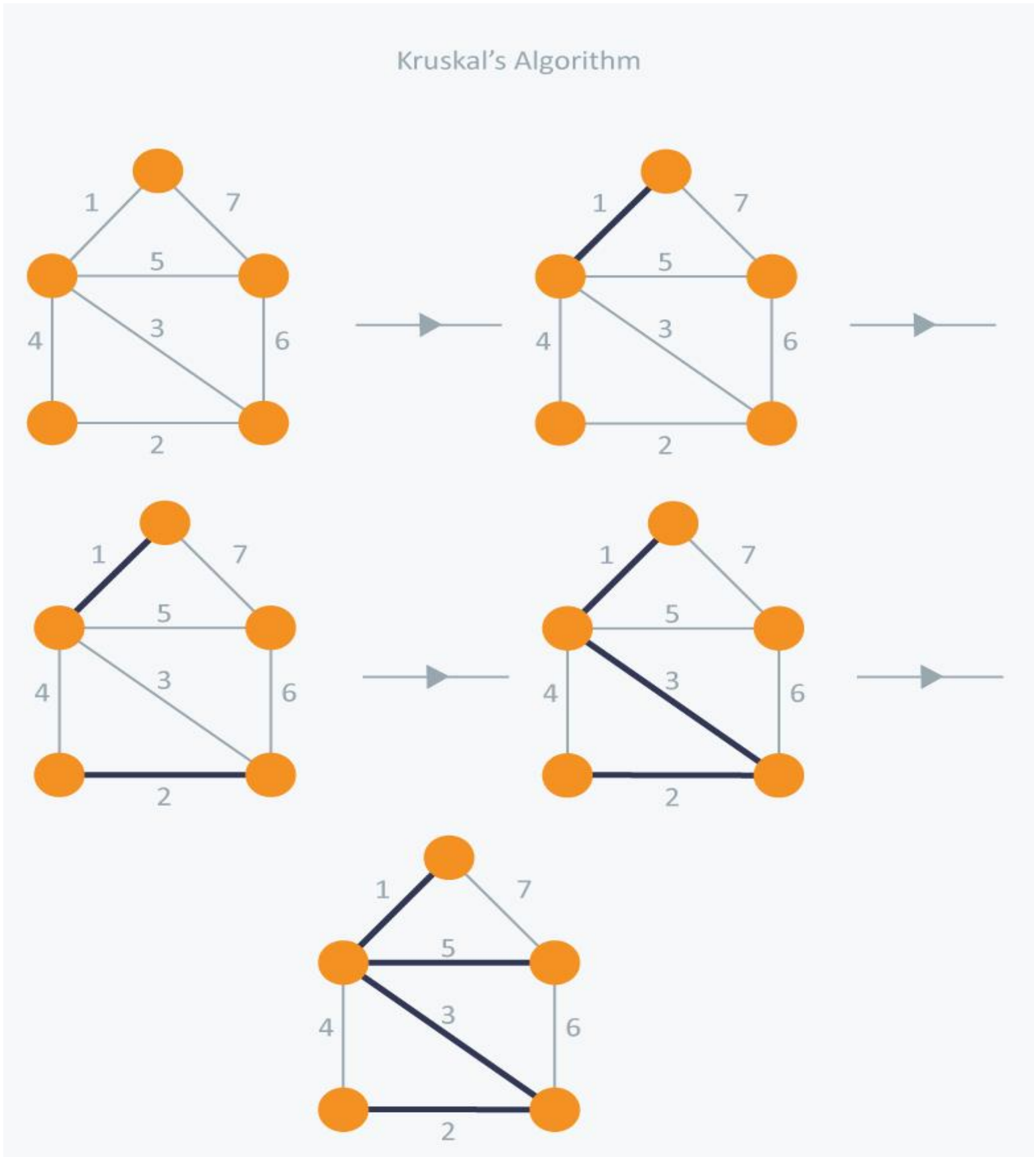


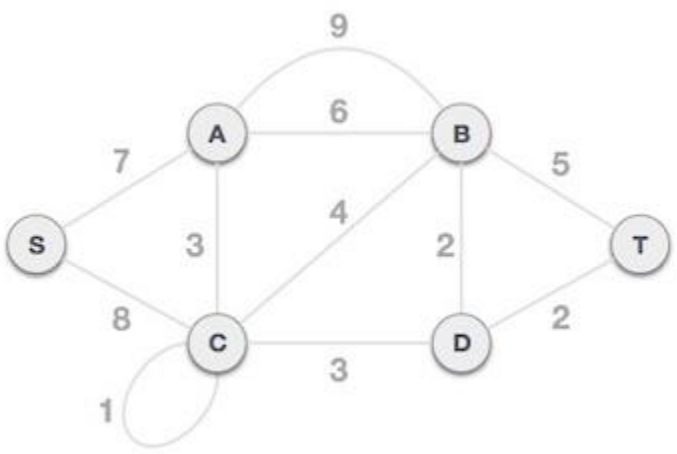
Figure: 14

In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So, we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 = (1 + 2 + 3 + 5).

Time Complexity:

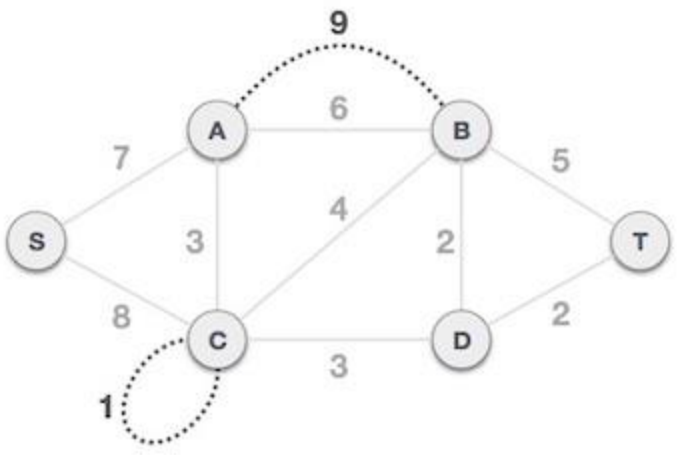
In Kruskal's algorithm, most time-consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

Example:

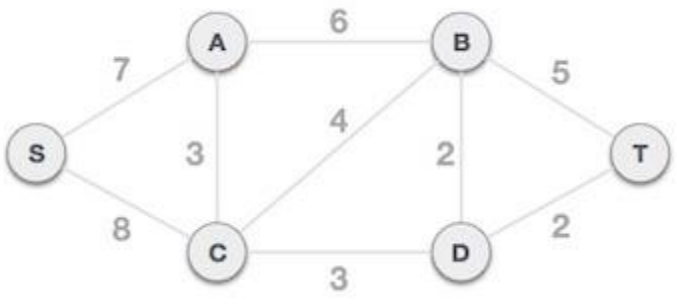


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



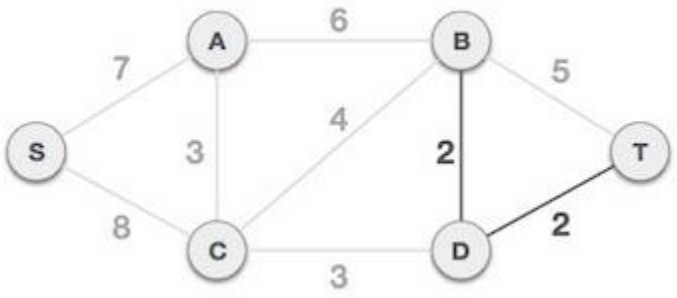
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

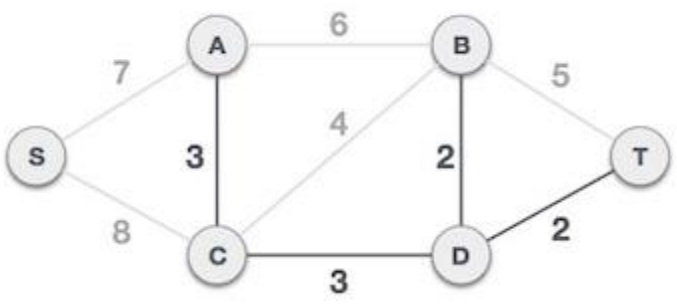
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

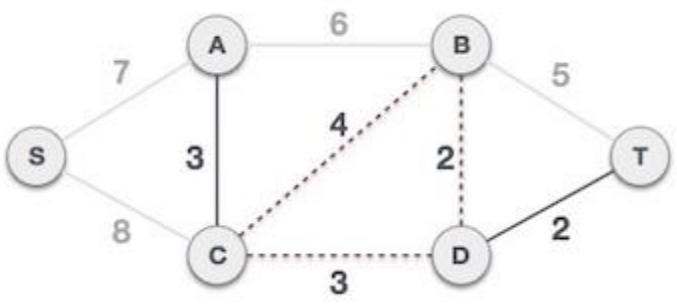


The least cost is 2 and edges involved are B, D and D, T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

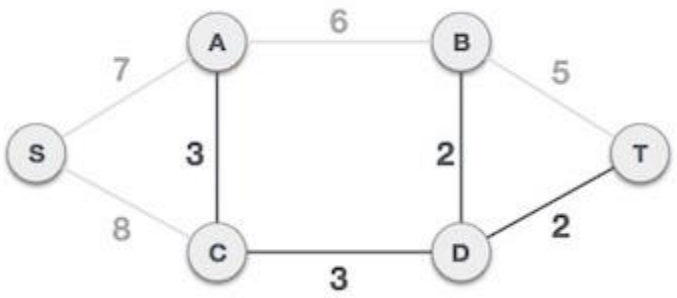
Next cost is 3, and associated edges are A, C and C, D. We add them again –



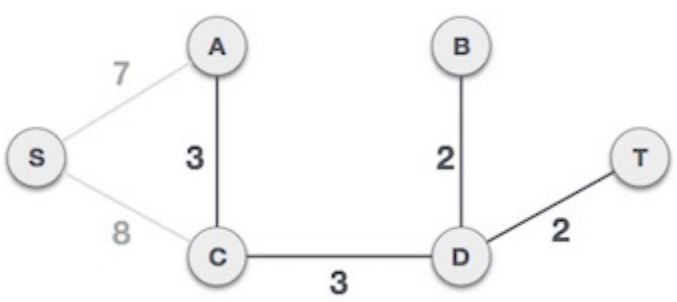
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



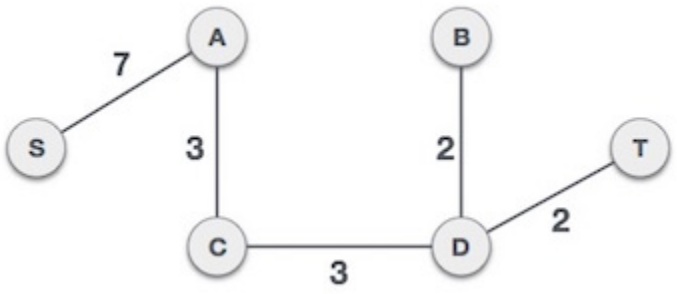
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S, A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Algorithm

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
  e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

Consider the example below:

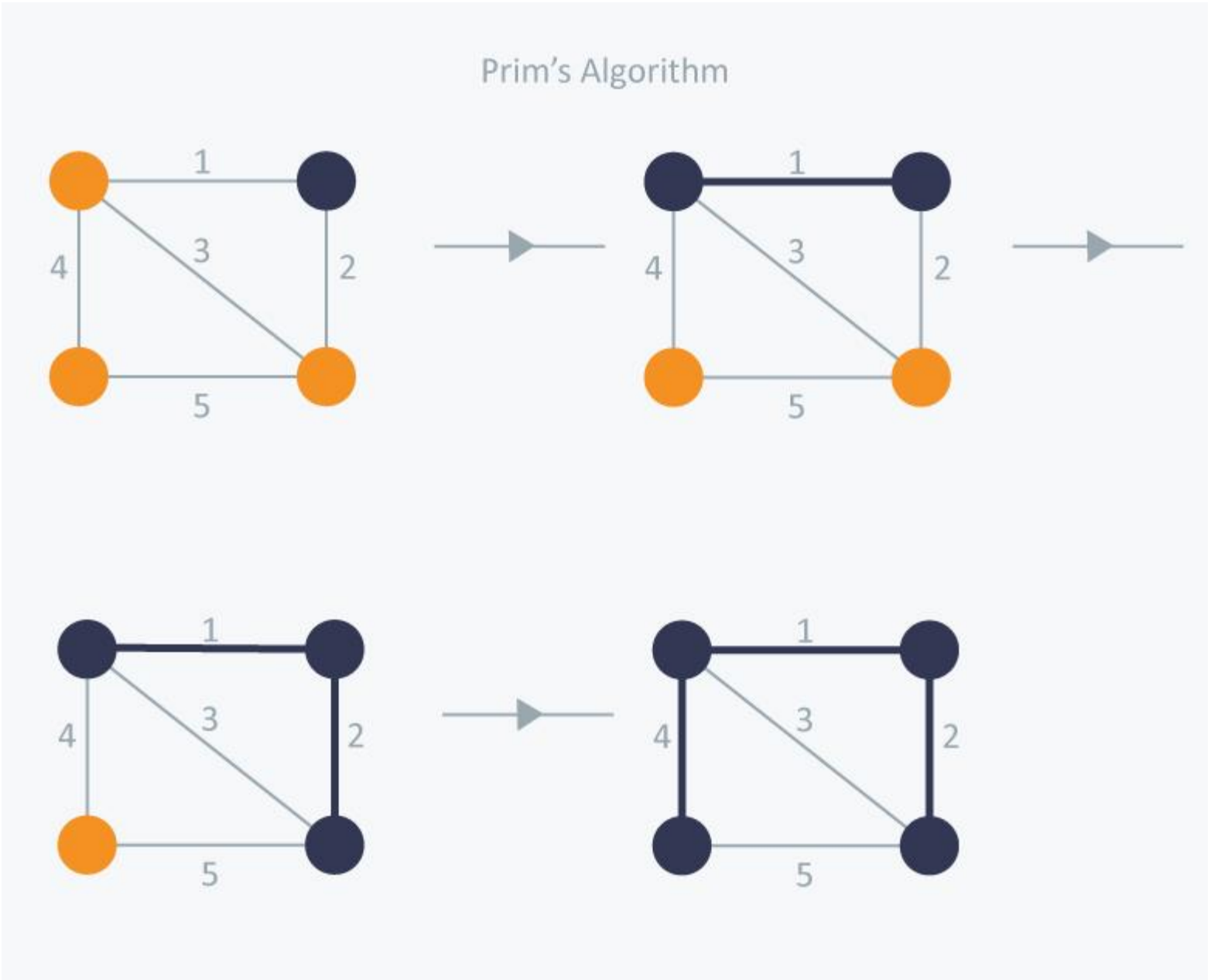


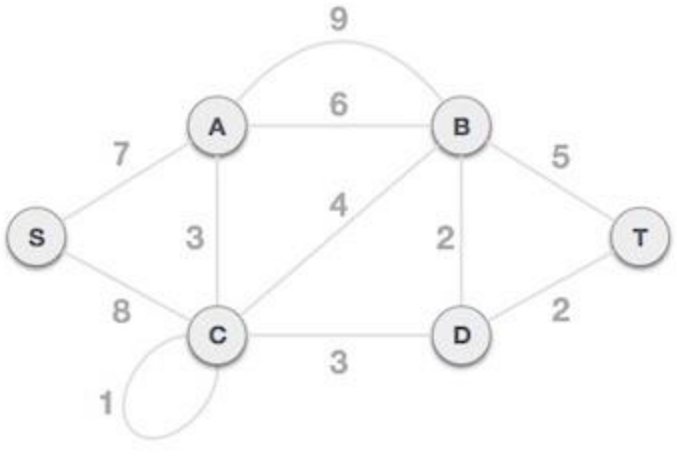
Figure: 15

In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So, we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So, we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 = (1 + 2 +4).

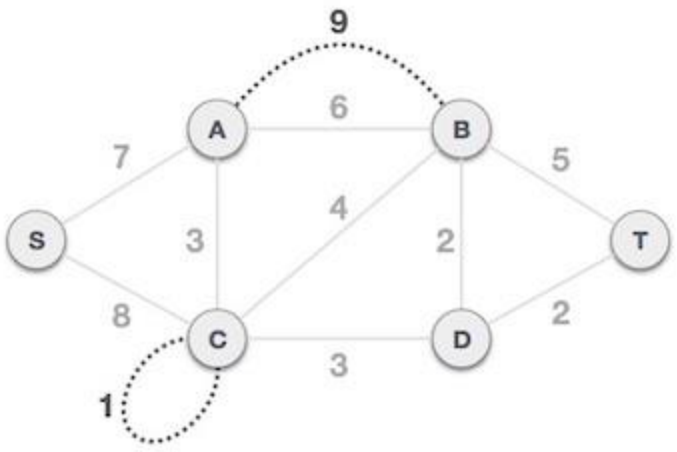
Time Complexity:

The time complexity of the Prim's Algorithm is O((V+E) logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

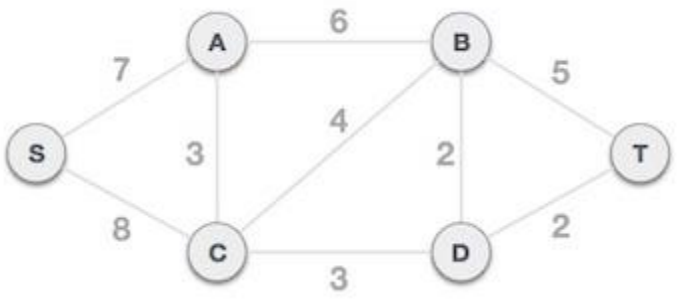
Example :



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

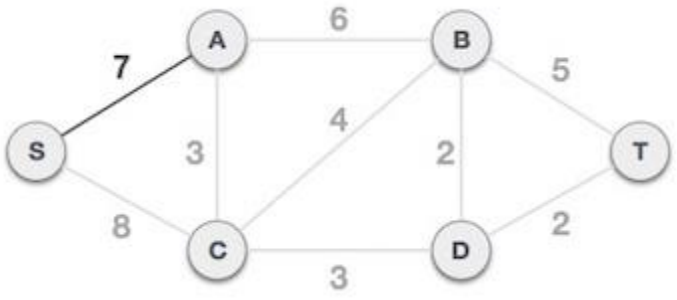


Step 2 - Choose any arbitrary node as root node

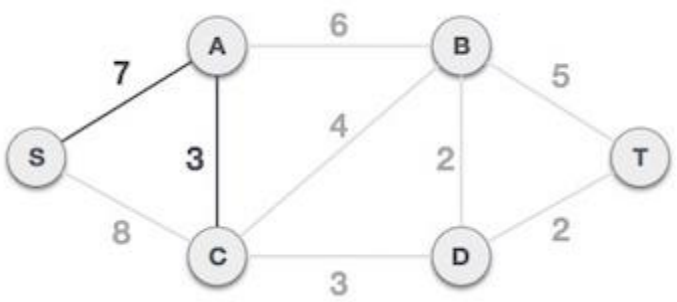
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So, the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

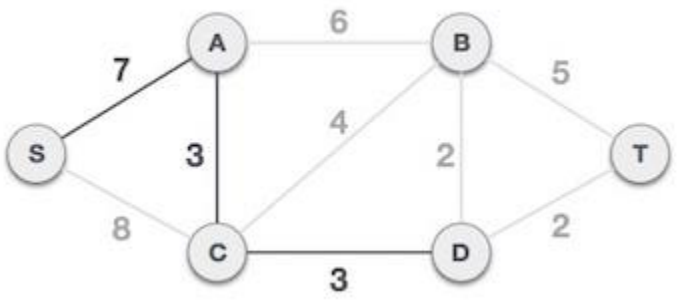
After choosing the root node **S**, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.



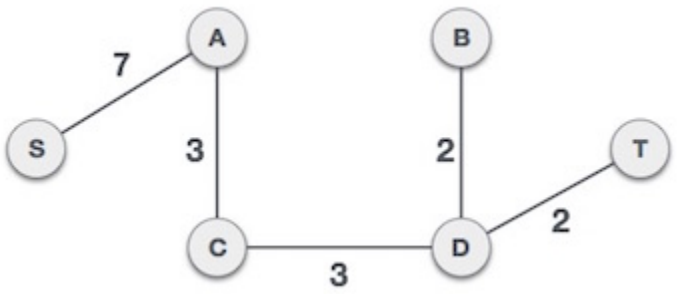
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e., D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Question:

Find all minimum spanning trees for the following graph. Use Kruskal's algorithm and Prim's algorithm starting at vertex a. Indicate the order in which edges are added to form each tree.

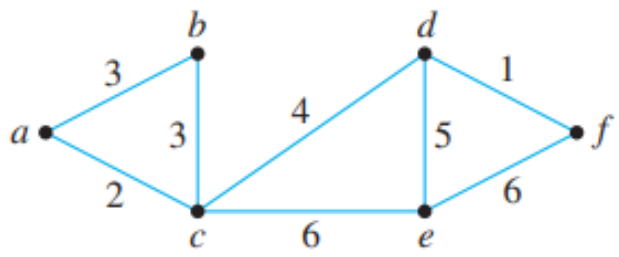


Figure: 16

Solution:

When Kruskal's algorithm is applied, edges are added in one of the following two orders:

1. {d, f}, {a, c}, {a, b}, {c, d}, {d, e}

2. {d, f}, {a, c}, {b, c}, {c, d}, {d, e}

When Prim's algorithm is applied starting at a, edges are added in one of the following two orders:

1. {a, c}, {a, b}, {c, d}, {d, f}, {d, e}

2. {a, c}, {b, c}, {c, d}, {d, f}, {d, e}

Thus, as shown below, there are two distinct minimum spanning trees for this graph.

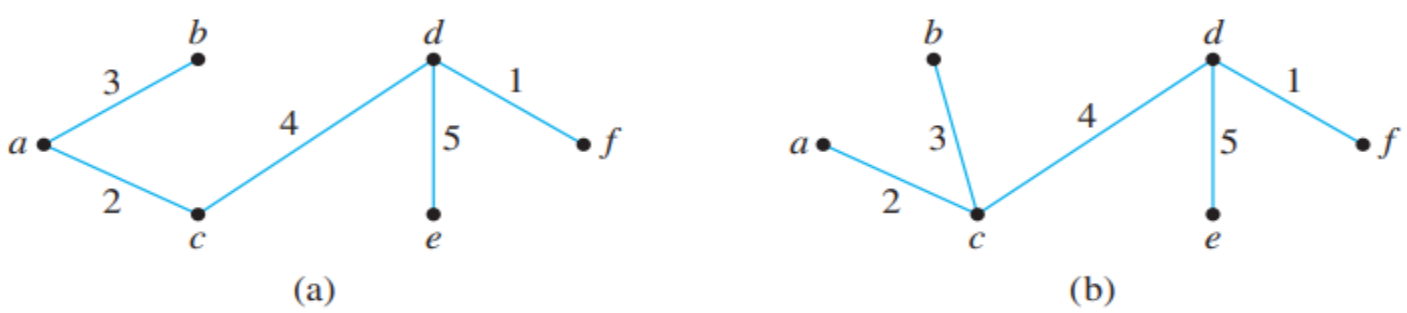


Figure: 17

Dijkstra's Shortest Path Algorithm

Although the trees produced by Kruskal's and Prim's algorithms have the least possible total weight compared to all other spanning trees for the given graph, they do not always reveal the shortest distance between any two points on the graph.

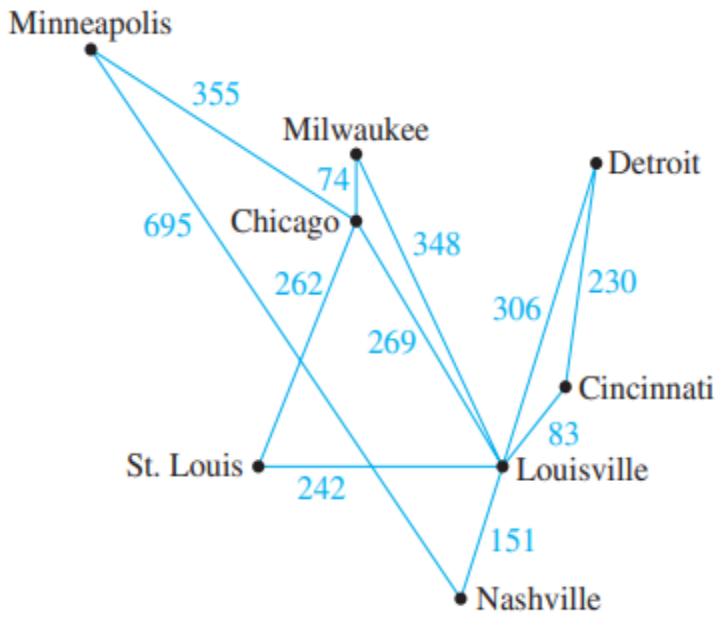


Figure 18: The graph of routes

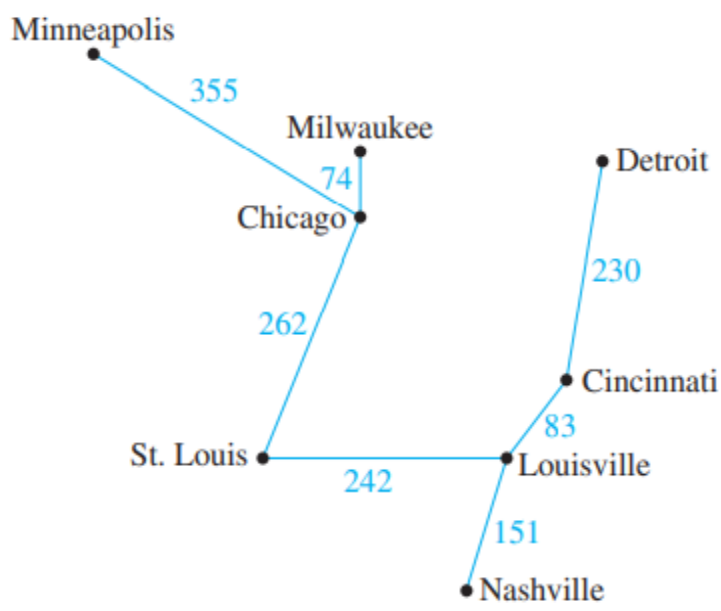


Figure 19: Tree produced by Kruskal's Algorithm

For instance, according to the system shown in Figure 18, one can fly directly from Nashville to Minneapolis for a distance of 695 miles, whereas if you use the minimum spanning tree shown in Figure 19, the only way to fly from Nashville to Minneapolis is by going through Louisville, St. Louis, and Chicago, which gives a total distance of 151+242+262+355 = 1,010 miles and the unpleasantness of three changes of plane.

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

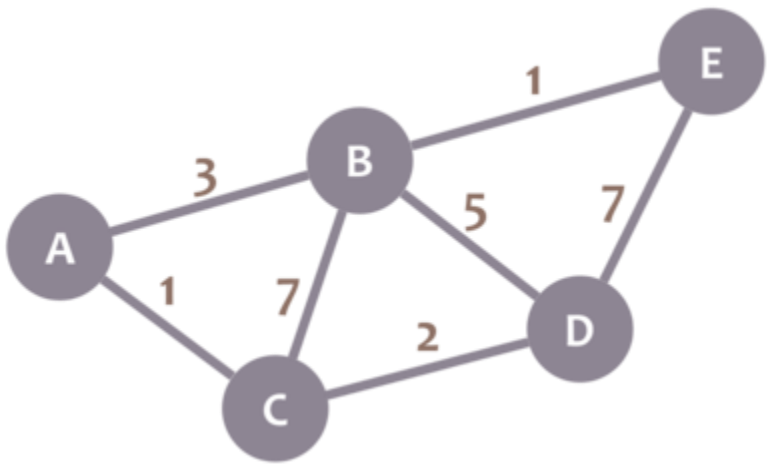
- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance, vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

ALGORITHM 1 Dijkstra's Algorithm.

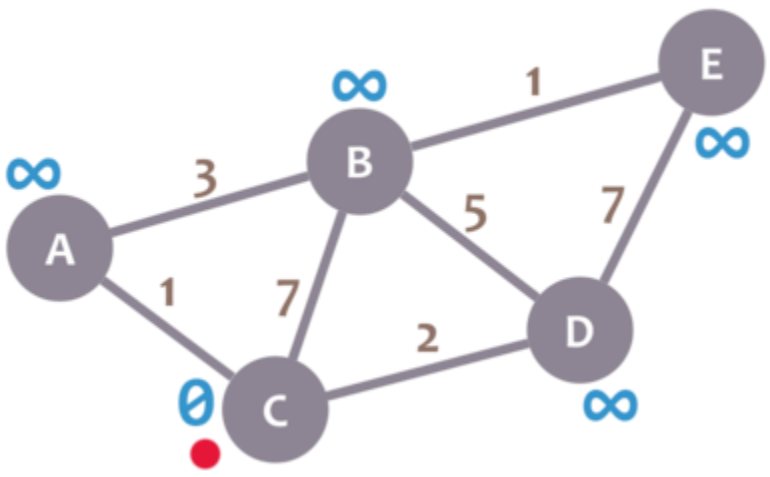
procedure *Dijkstra*(G: weighted connected simple graph, with all weights positive)
{G has vertices $a = v_0, v_1, \ldots, v_n = z$ and lengths $w(v_i, v_j)$ where $w(v_i, v_j) = \infty$ if $\{v_i, v_j\}$ is not an edge in G}
for $i := 1$ **to** n
  $L(v_i) := \infty$
$L(a) := 0$
$S := \emptyset$
{the labels are now initialized so that the label of a is 0 and all other labels are ∞, and S is the empty set}
while $z \notin S$
  $u :=$ a vertex not in S with $L(u)$ minimal
  $S := S \cup \{u\}$
  for all vertices v not in S
    if $L(u) + w(u, v) < L(v)$ **then** $L(v) := L(u) + w(u, v)$
    {this adds a vertex to S with minimal label and updates the labels of vertices not in S}
return $L(z)$ {$L(z) =$ length of a shortest path from a to z}

Example:

Dijkstra's Algorithm allows you to calculate the shortest path between one node (you pick which one) and *every other node in the graph*. You'll find a description of the algorithm at the end of this page, but let's study the algorithm with an explained example! Let's calculate the shortest path between node C and the other nodes in our graph:

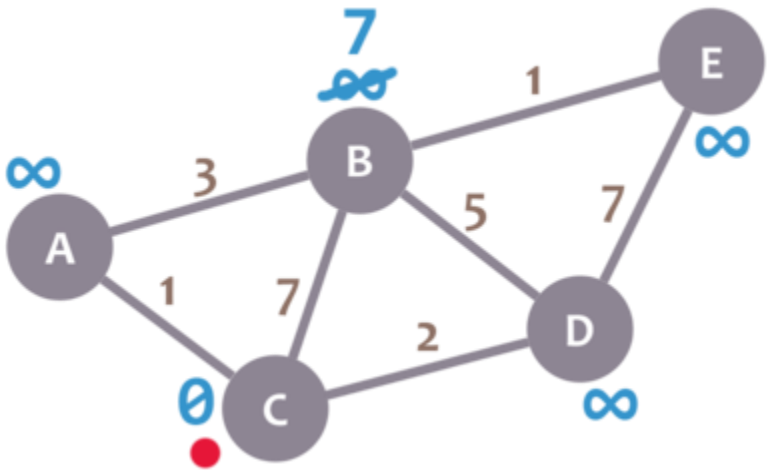


During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):

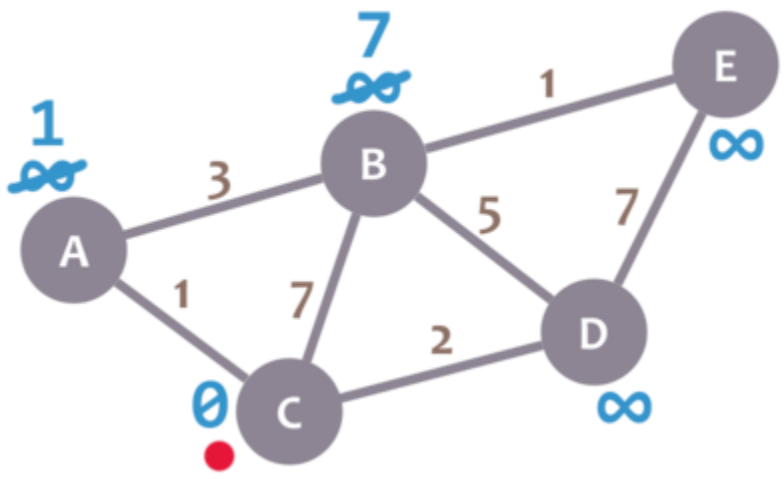


We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

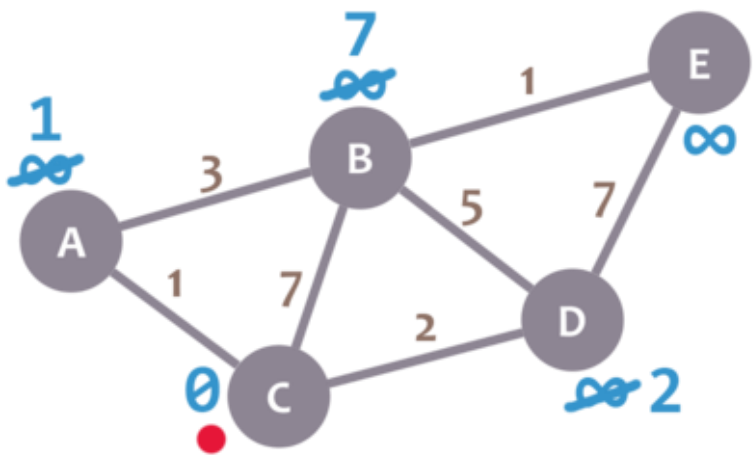
Now, we check the neighbors of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



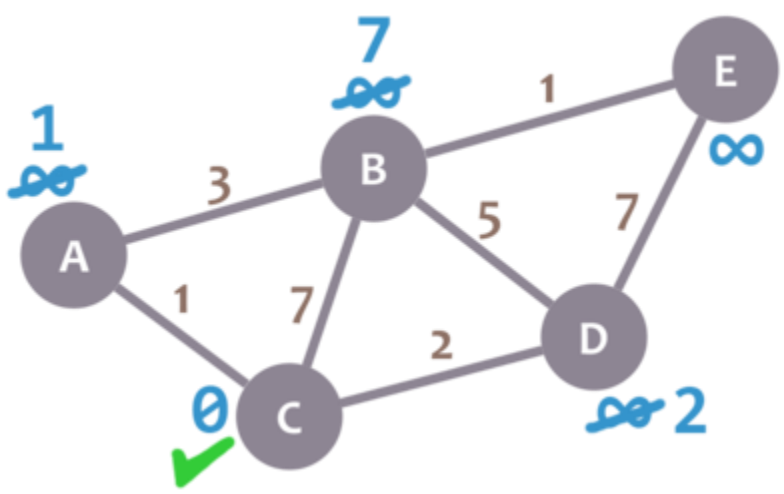
So far, so good. Now, let's check neighbor A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



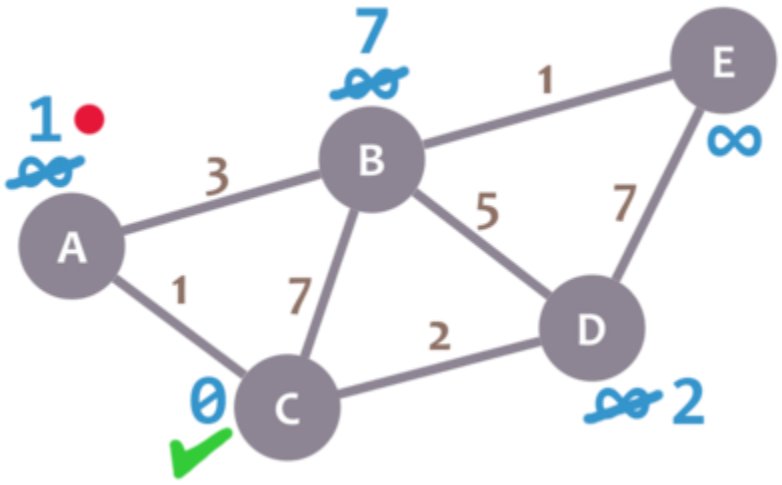
OK. Repeat the same procedure for D:



Great. We have checked all the neighbors of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:

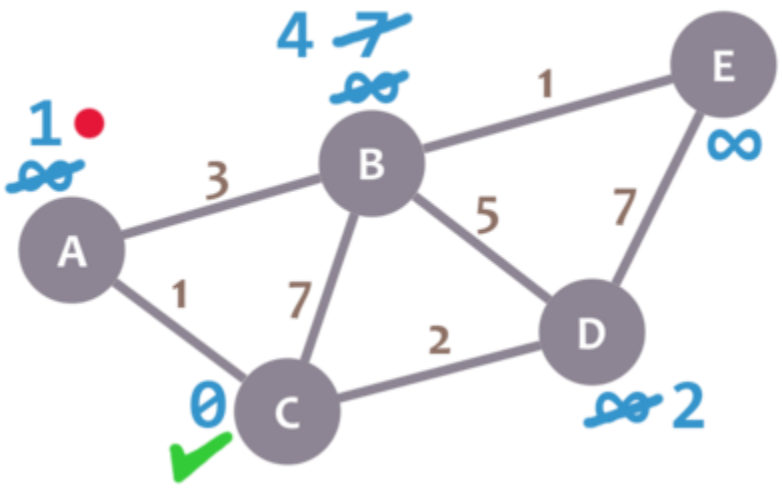


We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

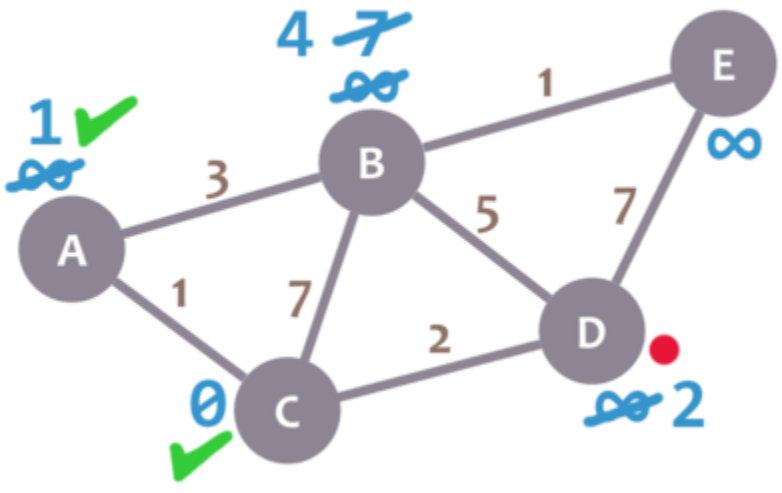


And now we repeat the algorithm. We check the neighbors of our current node, ignoring the visited nodes. This means we only check B.

For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



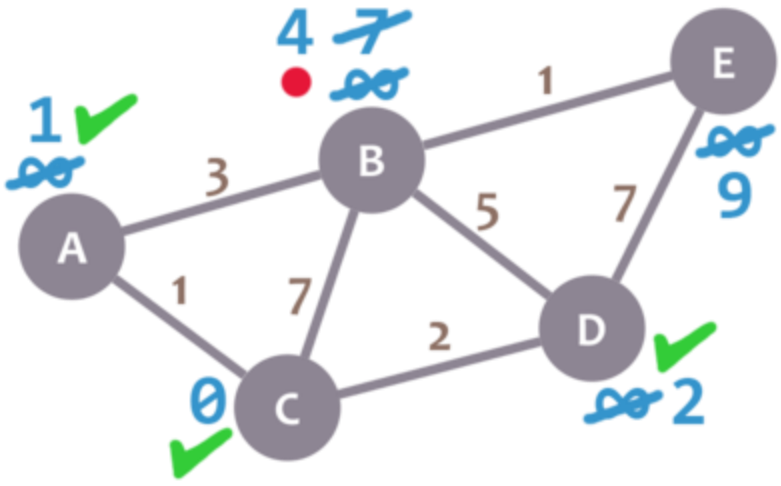
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



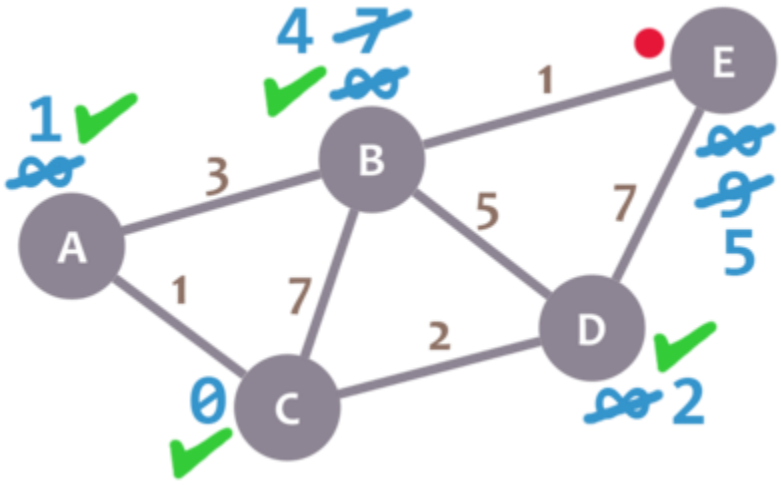
We repeat the algorithm again. This time, we check B and E.

For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

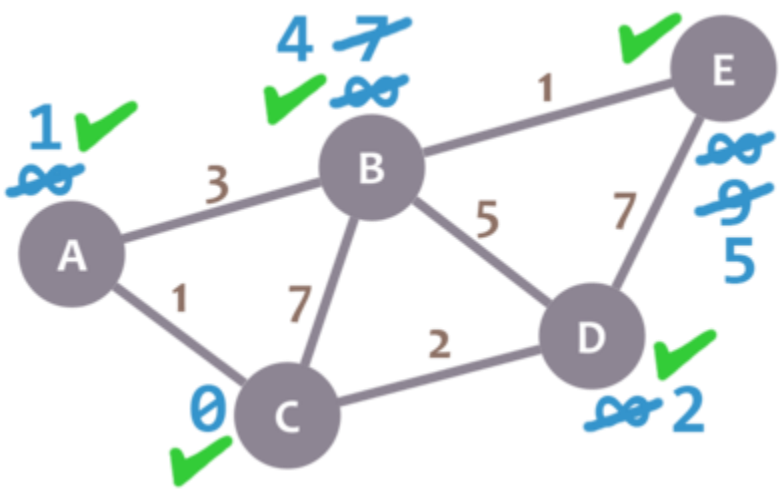
We mark D as visited and set our current node to B.



Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbors, so we don't need to check anything. We mark it as visited.



As there are not unvisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node-to-node C (the node we picked as our initial node)!

Here's a description of the algorithm:

1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node C.
3. For each neighbor N of your current node C: add the current distance of C with the weight of the edge connecting C-N. If it's smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. If there are non-visited nodes, go to step 2.

Example 2:

Use Dijkstra's algorithm to find the length of a shortest path between the vertices a and z in the weighted graph displayed in Figure 20(a).

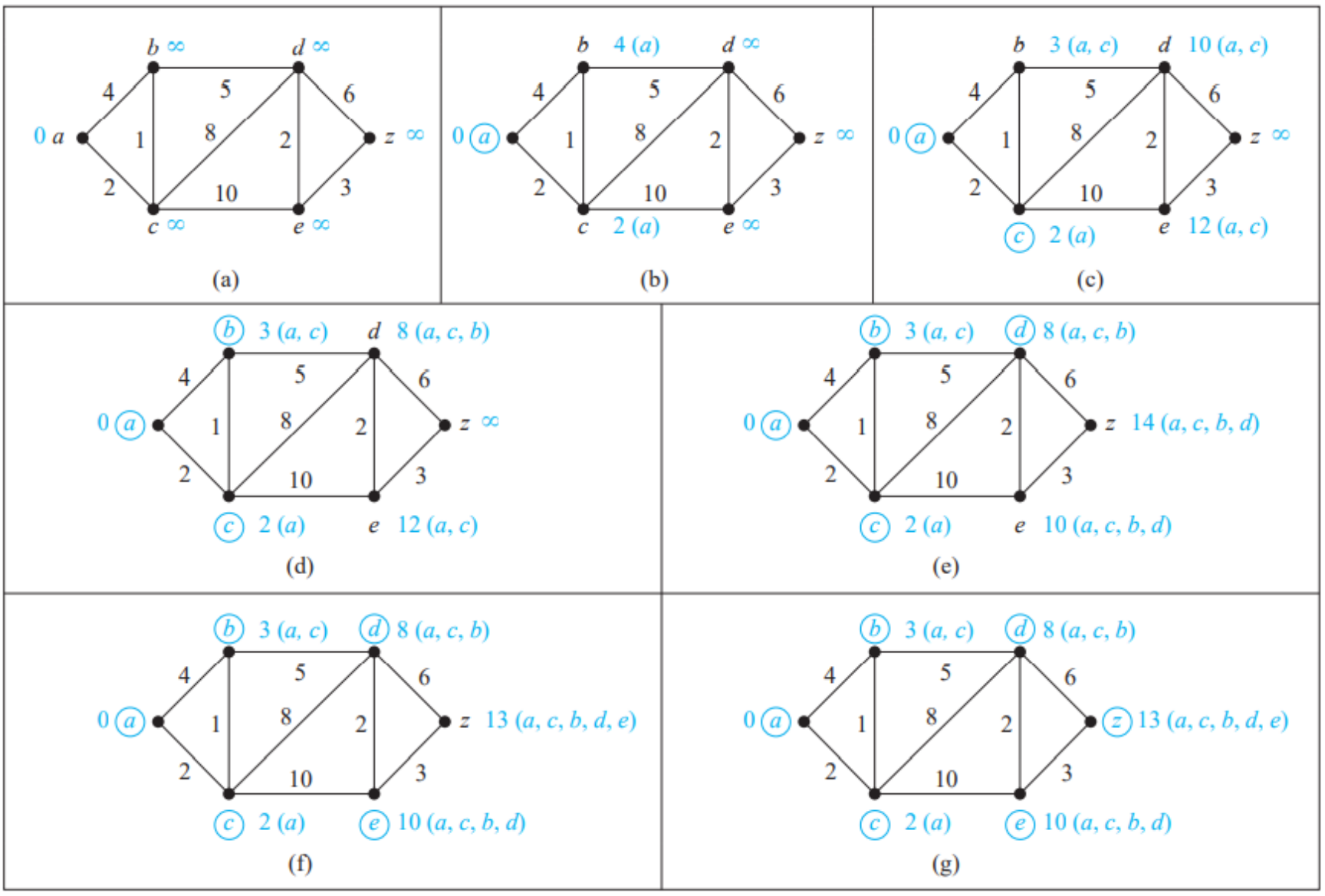


Figure 20: Using Dijkstra's Algorithm to Find a Shortest Path from a to z.

Action of Dijkstra's Algorithm

Show the steps in the execution of Dijkstra's shortest path algorithm for the graph shown below with starting vertex a and ending vertex z.

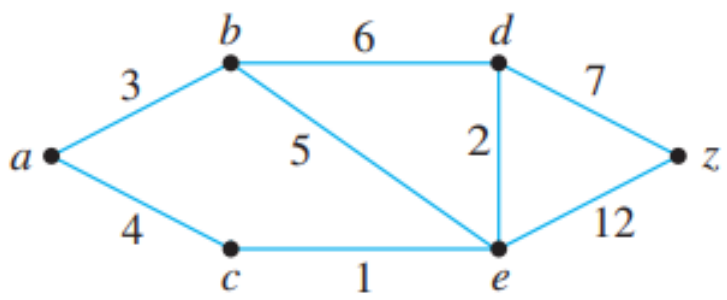
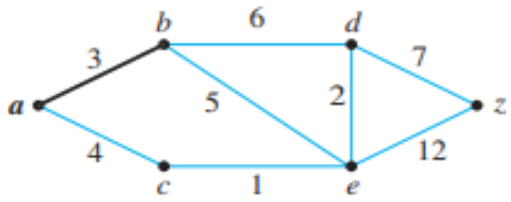


Figure: 21

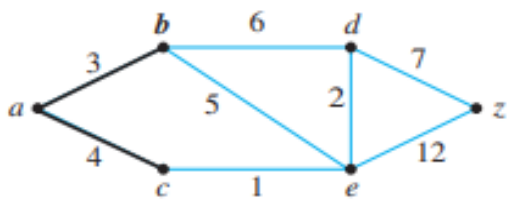
Solution

Step 1: Going into the **while** loop: $V(T) = \{a\}$, $E(T) = \varnothing$, and $F = \{a\}$



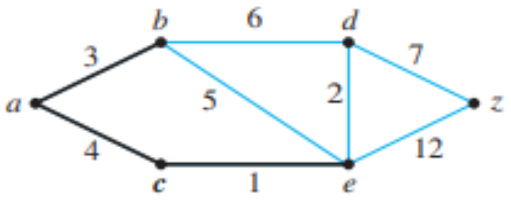
During iteration:
$F = \{b, c\}$, $L(b) = 3$, $L(c) = 4$.
Since $L(b) < L(c)$, b is added to $V(T)$, $D(b) = a$, and $\{a, b\}$ is added to $E(T)$.

Step 2: Going into the **while** loop: $V(T) = \{a, b\}$, $E(T) = \{\{a, b\}\}$



During iteration:
$F = \{c, d, e\}$, $L(c) = 4$, $L(d) = 9$, $L(e) = 8$.
Since $L(c) < L(d)$ and $L(c) < L(e)$, c is added to $V(T)$, $D(c) = a$, and $\{a, c\}$ is added to $E(T)$.

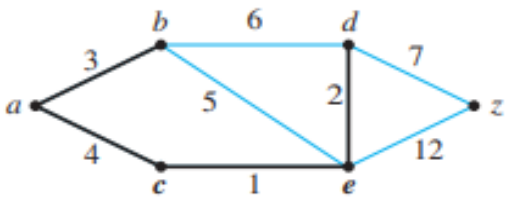
Step 3: Going into the **while** loop: $V(T) = \{a, b, c\}$, $E(T) = \{\{a, b\}, \{a, c\}\}$



During iteration:
$F = \{d, e\}$, $L(d) = 9$, $L(e) = 5$
$L(e)$ becomes 5 because *ace*, which has length 5, is a shorter path to e than *abe*, which has length 8.

Since $L(e) < L(d)$, e is added to $V(T)$, $D(e) = c$, and $\{c, e\}$ is added to $E(T)$.

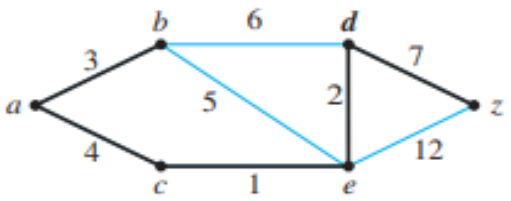
Step 4: Going into the **while** loop: $V(T) = \{a, b, c, e\}$,
$E(T) = \{\{a, b\}, \{a, c\}, \{c, e\}\}$



During iteration:
$F = \{d, z\}$, $L(d) = 7$, $L(z) = 17$
$L(d)$ becomes 7 because *aced*, which has length 7, is a shorter path to d than *abd*, which has length 9.

Since $L(d) < L(z)$, d is added to $V(T)$, $D(d) = e$, and $\{e, d\}$ is added to $E(T)$.

Step 5: Going into the **while** loop: $V(T) = \{a, b, c, e, d\}$,
$E(T) = \{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}\}$



During iteration:
$F = \{z\}$, $L(z) = 14$
$L(z)$ becomes 14 because *acedz*, which has length 14, is a shorter path to d than *abdz*, which has length 17.

Since z is the only vertex in F, its label is a minimum, and so z is added to $V(T)$, $D(z) = d$, and $\{d, z\}$ is added to $E(T)$.

Execution of the algorithm terminates at this point because $z \in V(T)$. The shortest path from a to z has length L(z) = 14

Keeping track of the steps in a table is a convenient way to show the action of the algorithm. Table-1 does this for the graph in Example.

Table -1:

Step	$V(T)$	$E(T)$	F	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(z)$
0	$\{a\}$	$\varnothing$	$\{a\}$	**0**	∞	∞	∞	∞	∞
1	$\{a\}$	$\varnothing$	$\{b, c\}$	0	**3**	4	∞	∞	∞
2	$\{a, b\}$	$\{\{a, b\}\}$	$\{c, d, e\}$	0	3	**4**	9	8	∞
3	$\{a, b, c\}$	$\{\{a, b\}, \{a, c\}\}$	$\{d, e\}$	0	3	4	9	**5**	∞
4	$\{a, b, c, e\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}\}$	$\{d, z\}$	0	3	4	**7**	5	17
5	$\{a, b, c, e, d\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}\}$	$\{z\}$	0	3	4	7	5	**14**
6	$\{a, b, c, e, d, z\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}, \{e, z\}\}$							

In step 1, D(b) = a; in step 2, D(c) = a; in step 3, D(e) = c; in step 4, D(d) = e; and in step 5, D(z) = e. Working backward gives the vertices in the shortest path. Because D(z) = d, D(d) = e, and D(c) = a, the shortest path from a to z is – a, c, e, d, z.

It is clear that Dijkstra's algorithm keeps adding vertices to T until it has added z. The proof of the following theorem shows that when the algorithm terminates, the label for z, L(z), is the length of the shortest path to z from a.

Huffman Coding (Variable Length Coding)

A B B C D B C C D A A B B E E E B E A B -> 20 Characters

A = 65 = 01000001 -> 8 bits

Total bits required = 20 x 8 = 160 (for fixed length coding)

With n bits, we can use 2^n characters.

Example: with 2 bits, four combinations (00,01,10,11) are possible.

For the above example, 3 bits are required as there are 5 (5 > 2^2 = 4 and 5 < 2^3 = 8) different characters.

So, total bits required for this message = 20 x 3 = 60

Encryption Table	
A – 000 B – 001 C – 010 D – 011 E – 100	For the table, For these alphabets, 5 x 8 = 40 bits For the codes, 5 x 3 = 15 bits

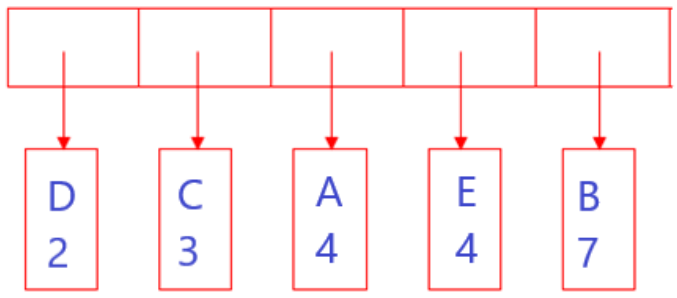
Total = 60 + 40 + 15 = 115 bits

Drawbacks of fixed-length codes

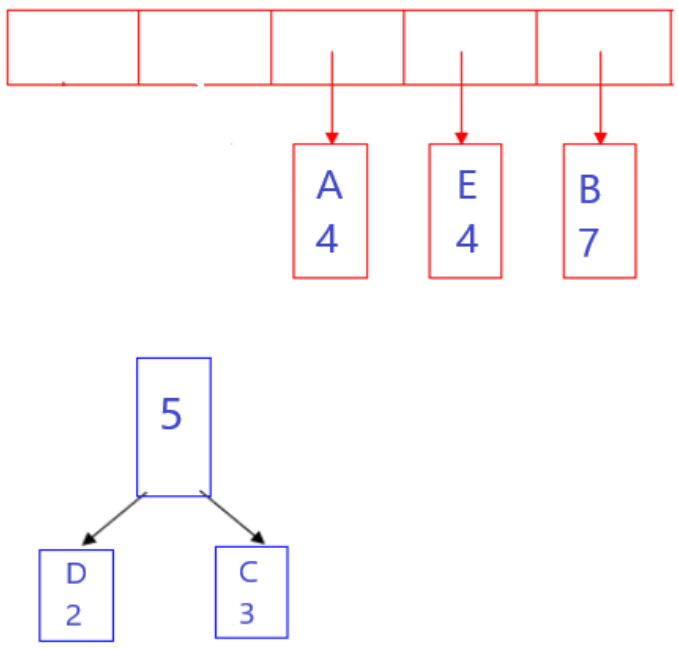
- Wasted space:
  - Unicode uses twice as much space as ASCII.
    - inefficient for plain-text messages containing only ASCII characters.
- Same number of bits used to represent all characters.
  - 'a' and 'e' occur more frequently than 'q' and 'z'.
- **Potential solution**: use variable-length codes
  - variable number of bits to represent characters when frequency of occurrence is known.
  - short codes for characters that occur frequently.

Characters	Counts
A	4
B	7
C	3
D	2
E	4

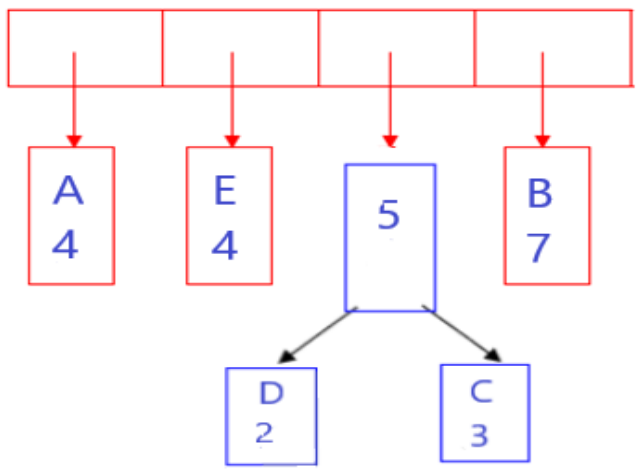
Step 1:



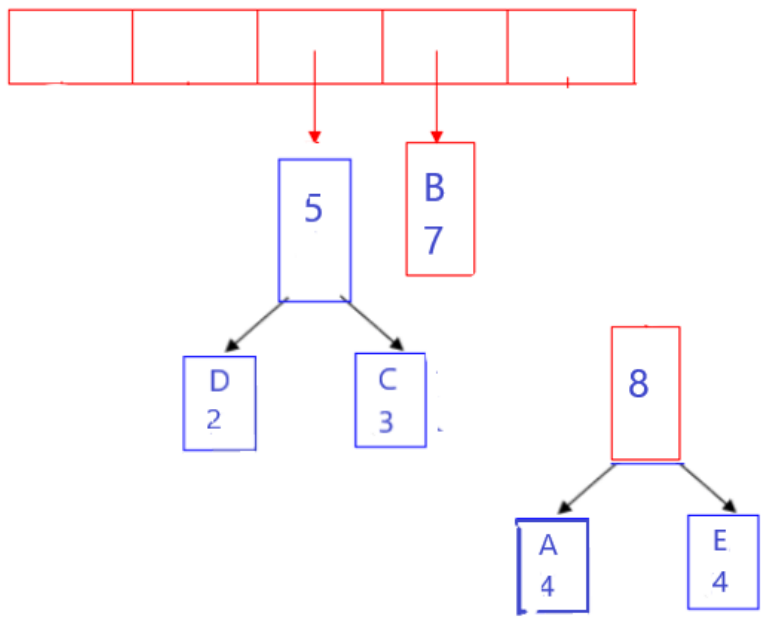
Step 2:



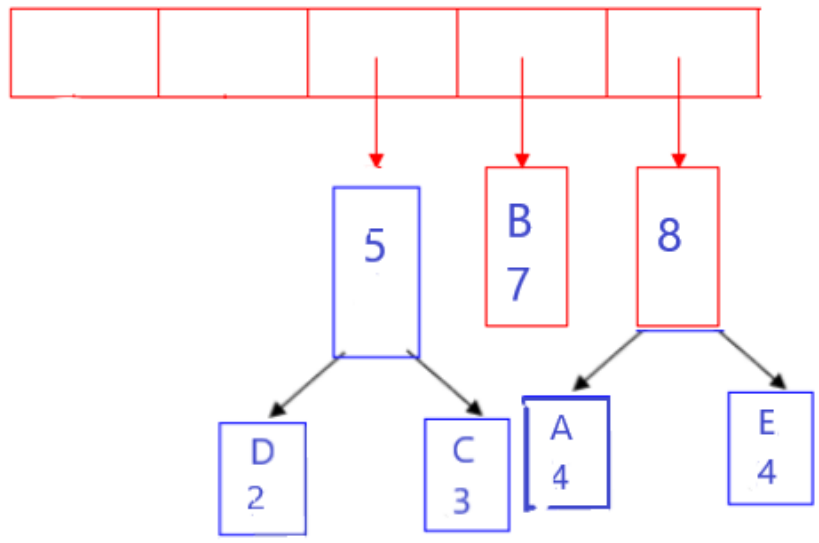
Step 3:



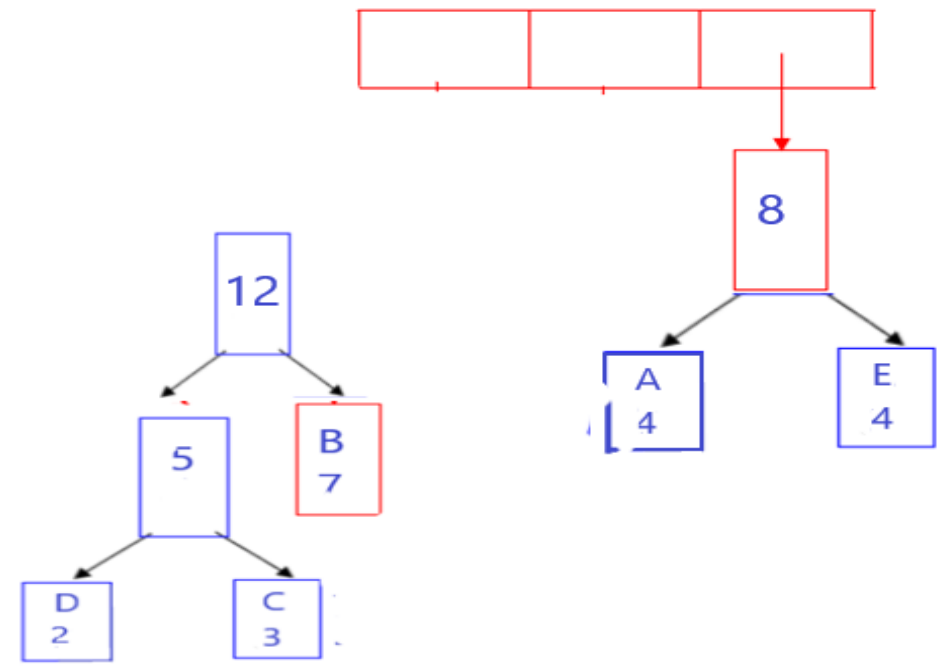
Step 4:



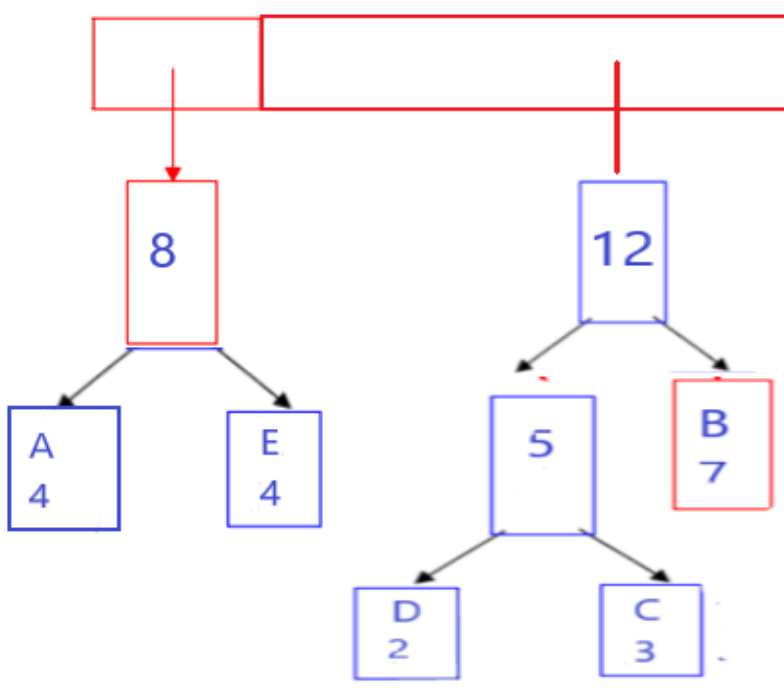
Step 5:



Step 6:

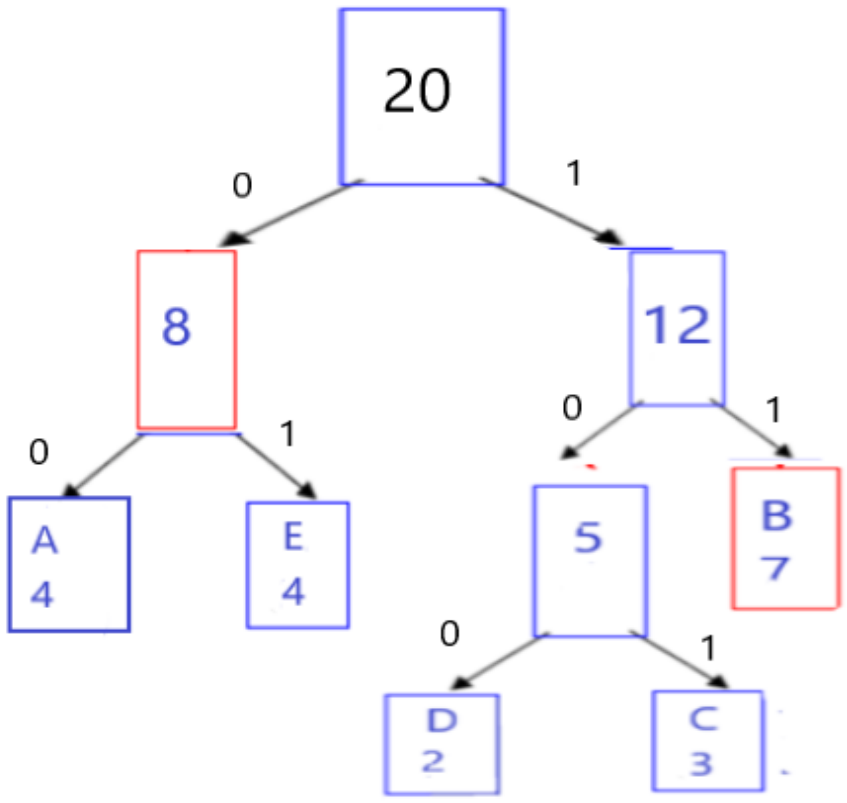


Step 7:



For left subtree 0, and for right subtree assign 1.

So, the binary tree is:



Encryption Table	
A – 00 B – 11 C – 101 D – 100 E – 01	For the table; For these alphabets, 5 x 8 = 40 bits For the codes, 2+2+3+3+2= 12 bits Total = 40 + 12 = 52

Characters	Counts	Code
A	4	00
B	7	11
C	3	101
D	2	100
E	4	01

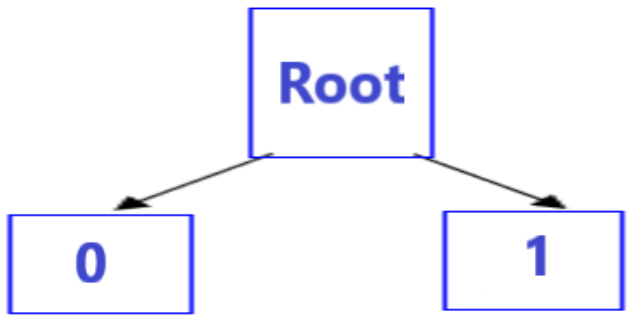
Total bits required for this message = **4x2 + 7x2 + 3x3 + 2x3 + 4x2 = 47**

Total = **47 + 52 bits = 99 bits**

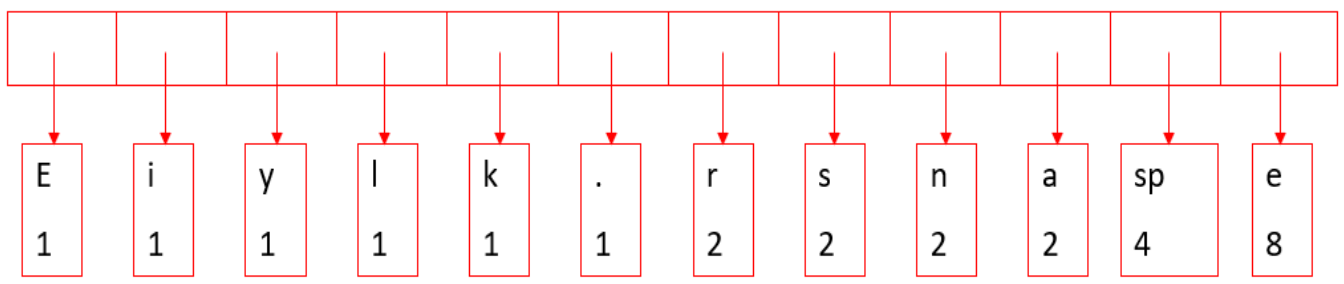
Time complexity: O(n logn).

The character with large frequency produces small code and the character with small frequency produces large code.

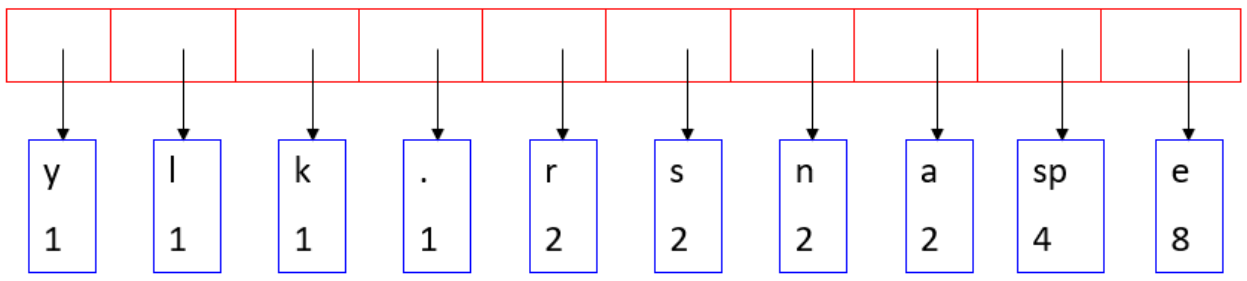
Building a Binary Tree:

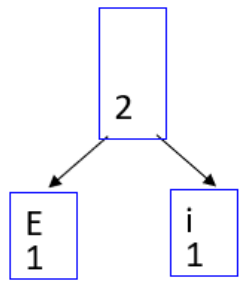


Step 1:

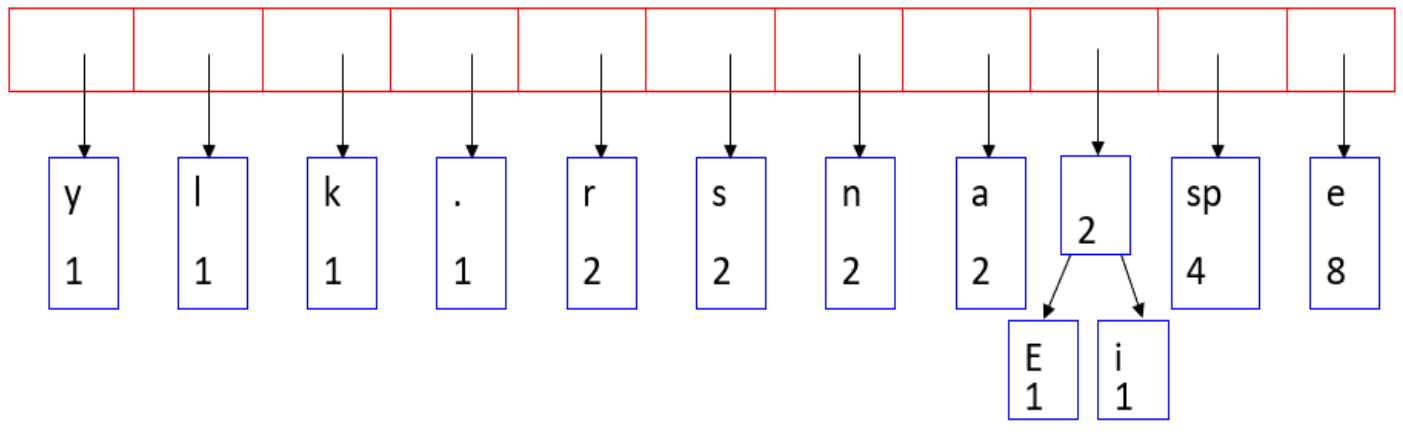


Step 2:

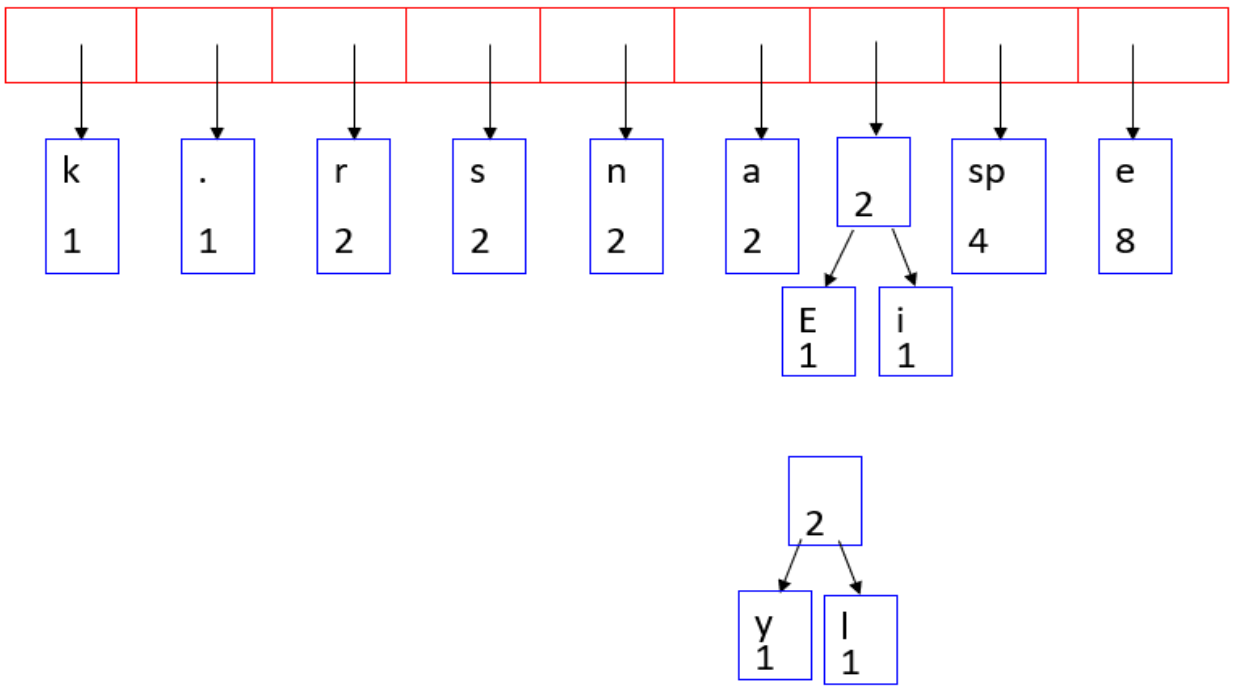




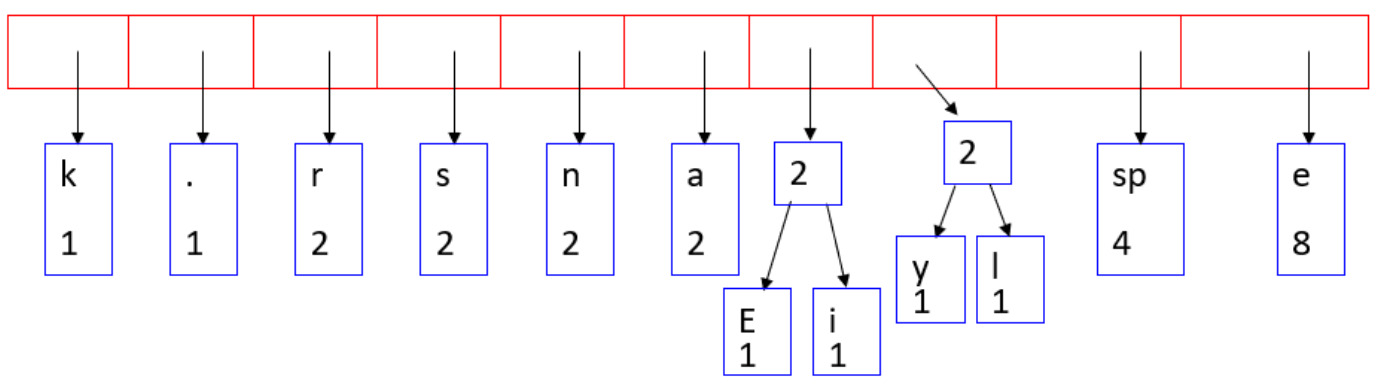
Step 3:



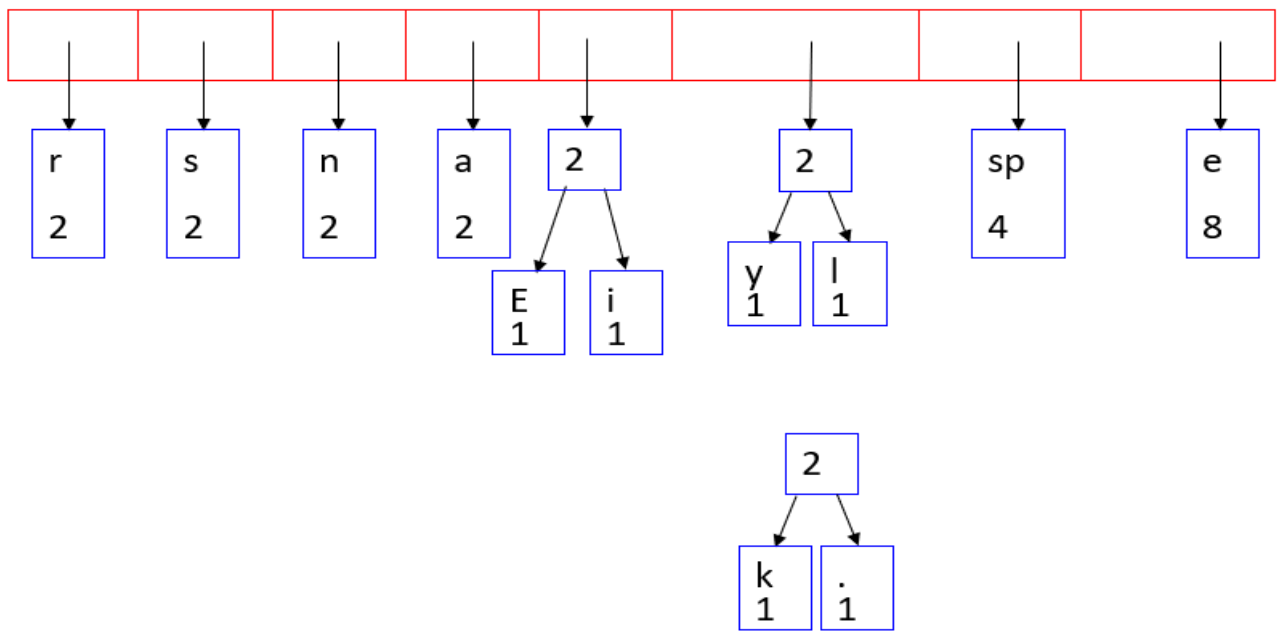
Step 4:



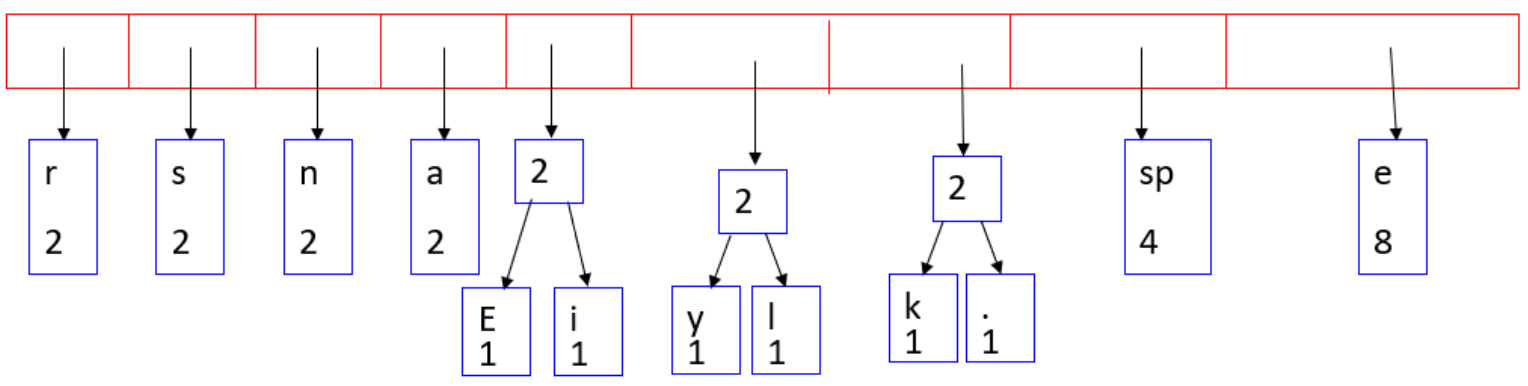
Step 5:



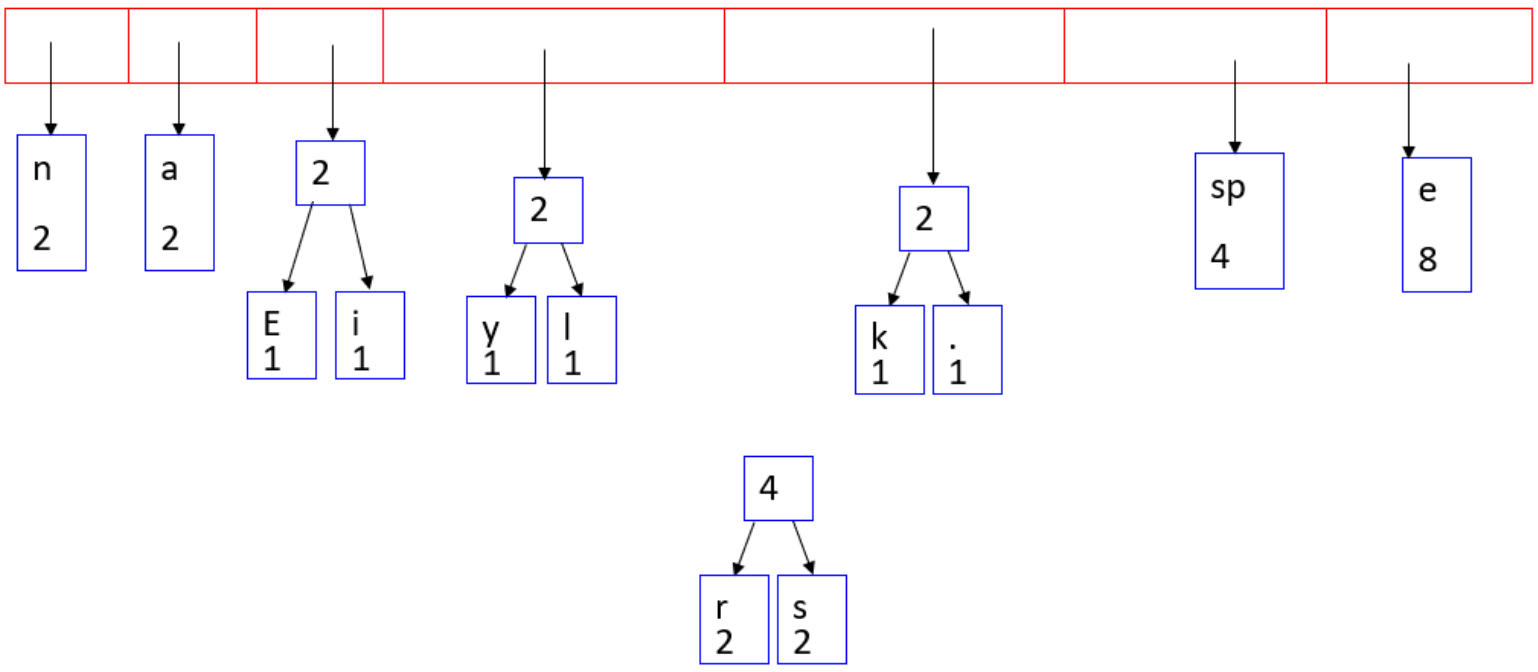
Step 6:



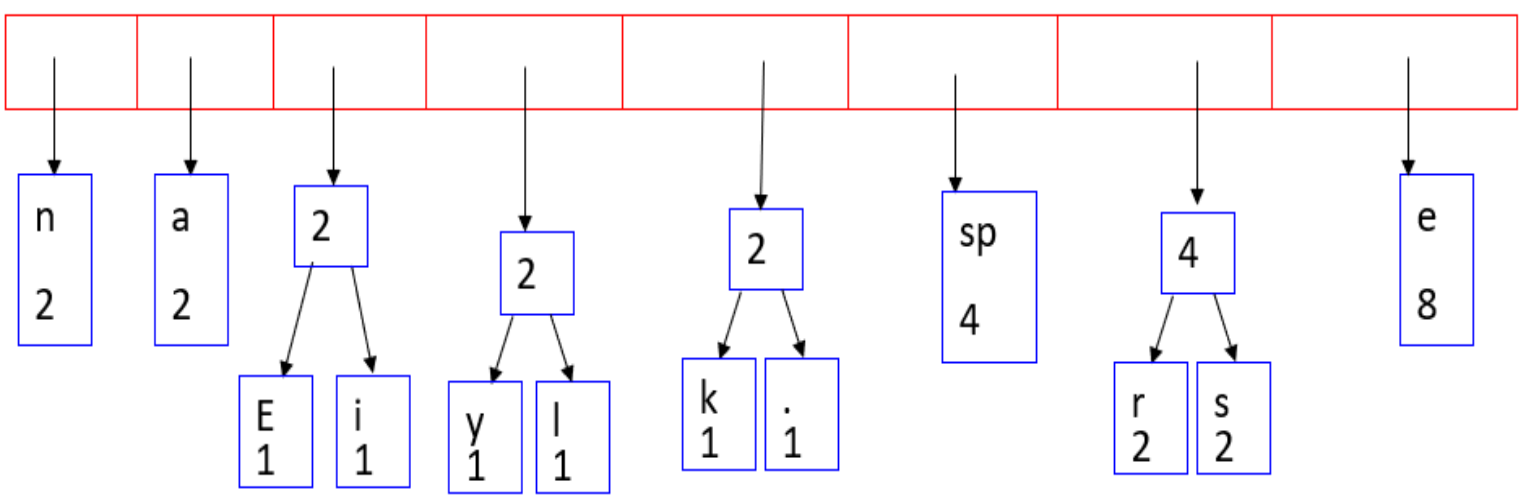
Step 7:



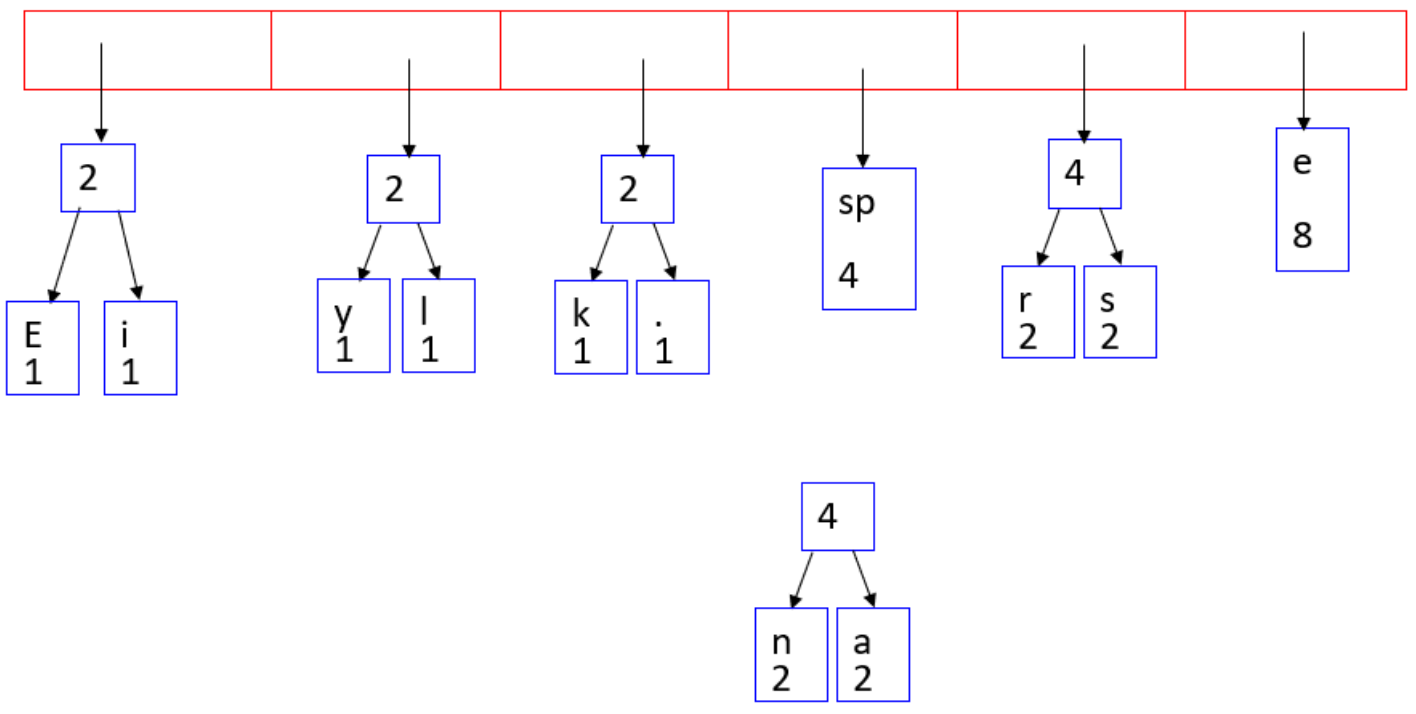
Step 8:



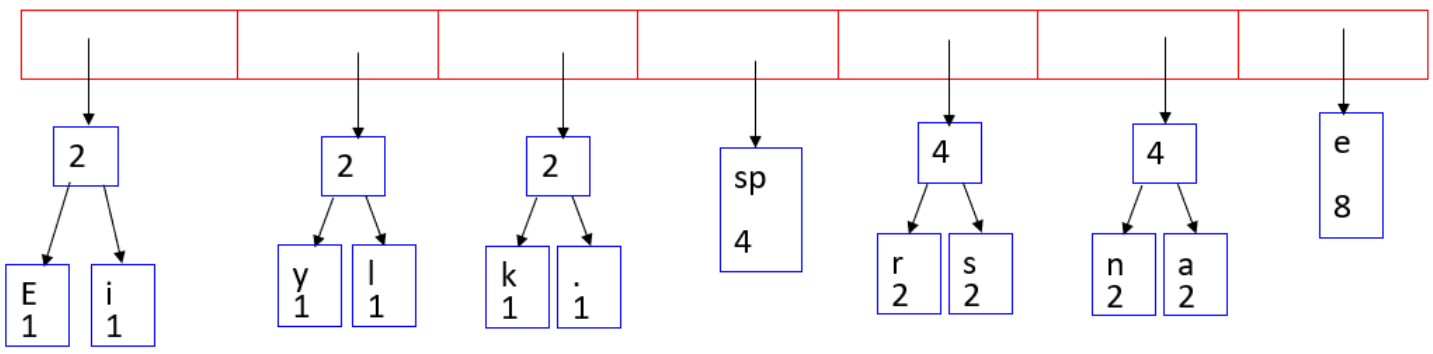
Step 9:



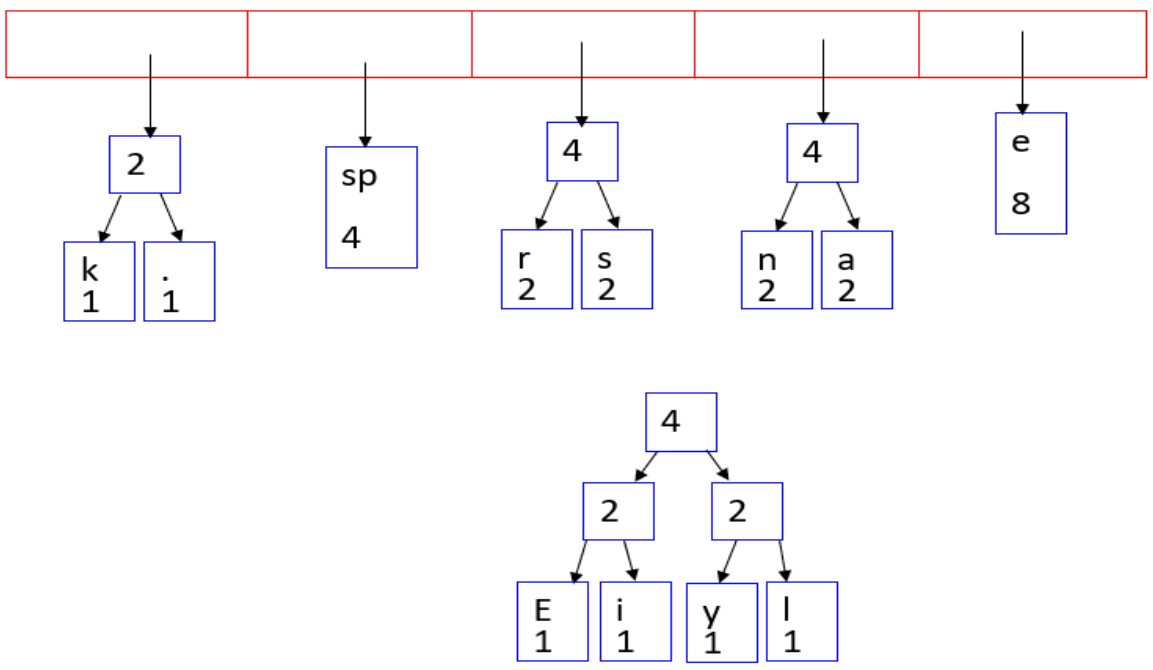
Step 10:



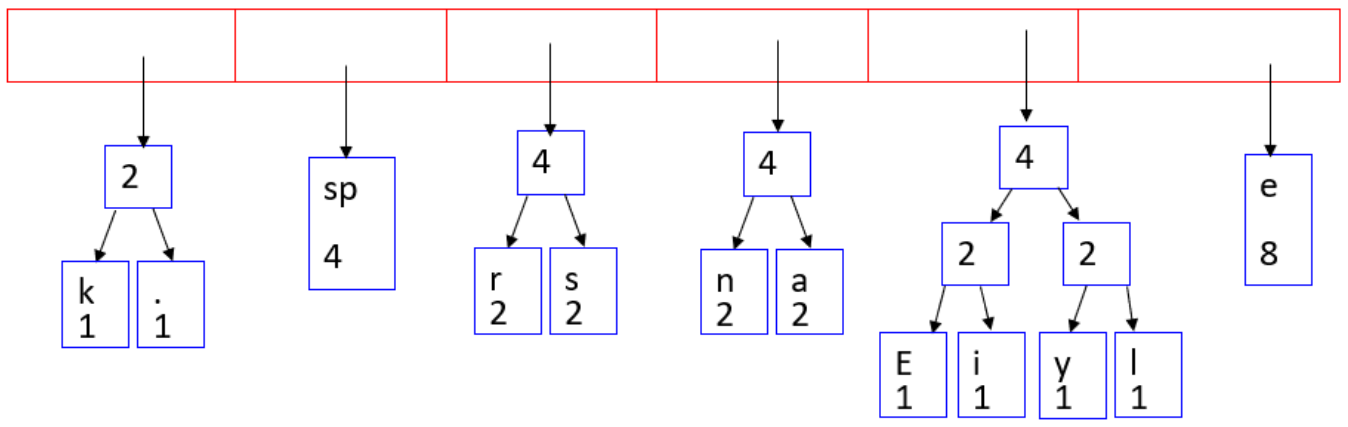
Step 11:



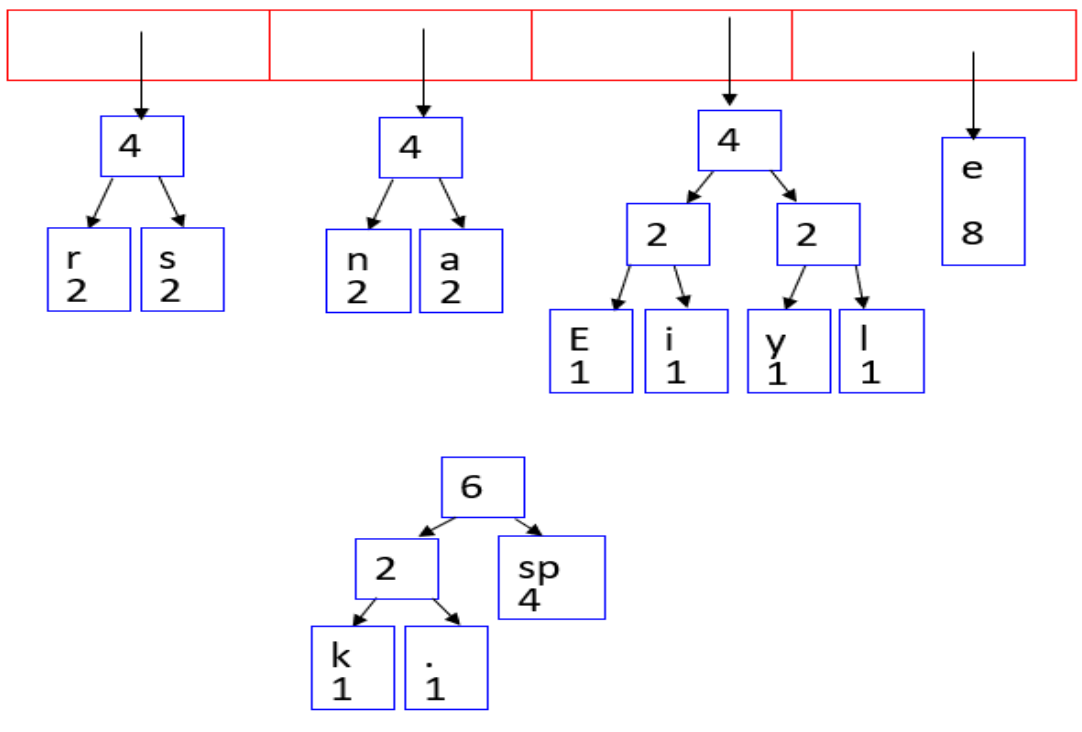
Step 12:



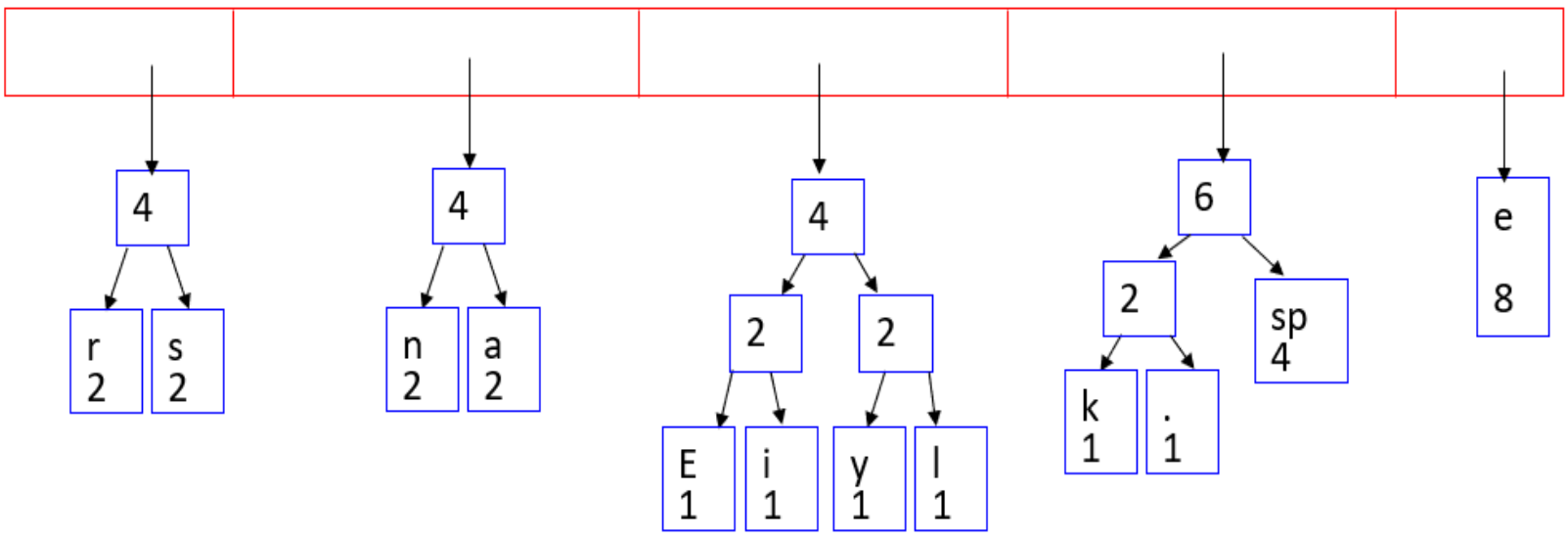
Step 13:



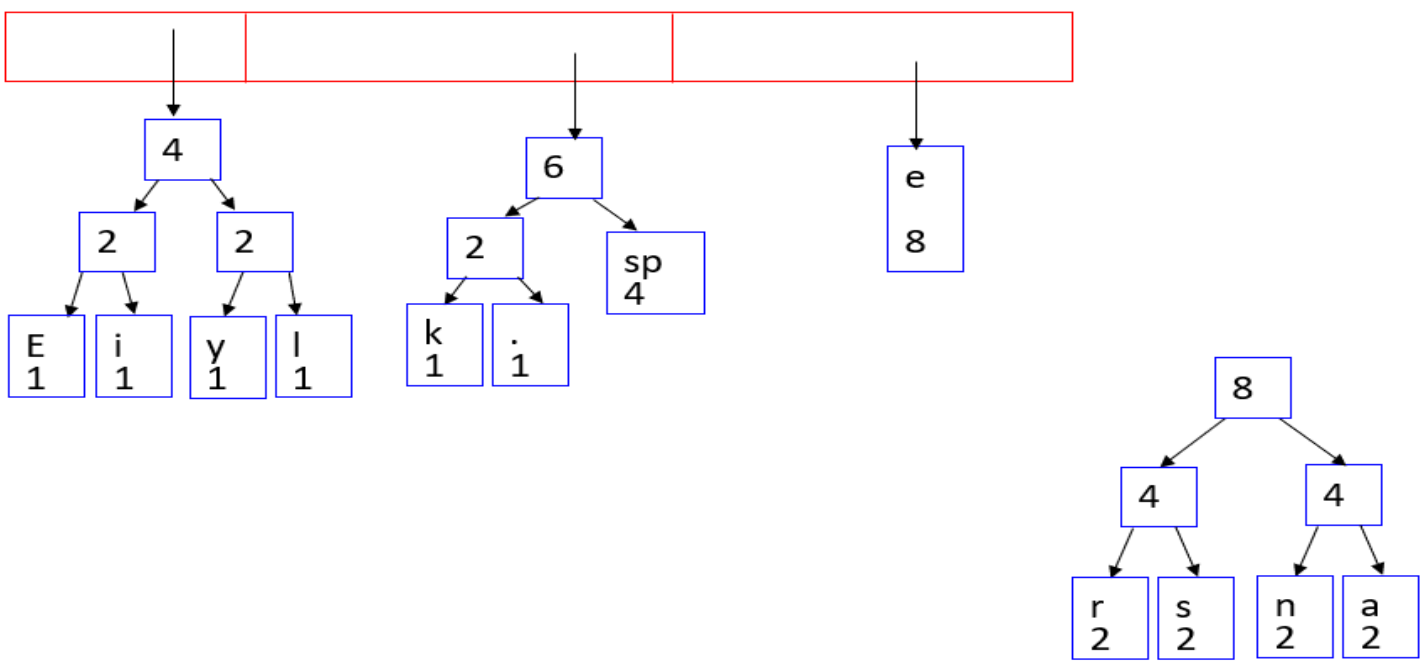
Step 14:



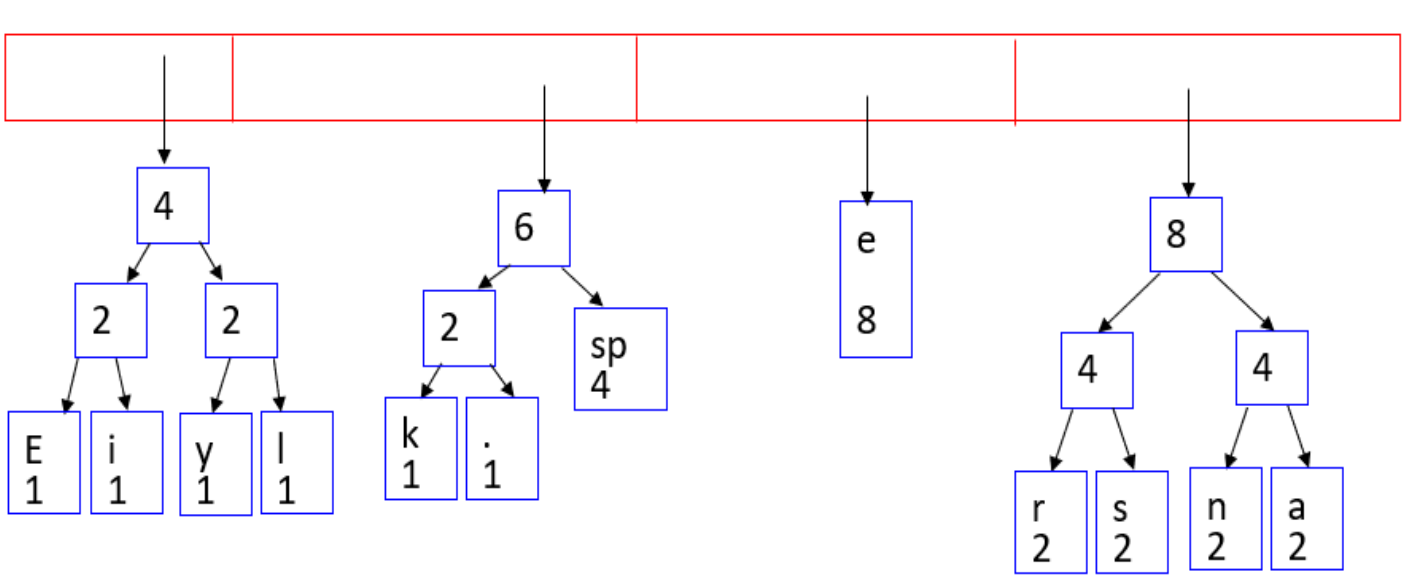
Step 15:



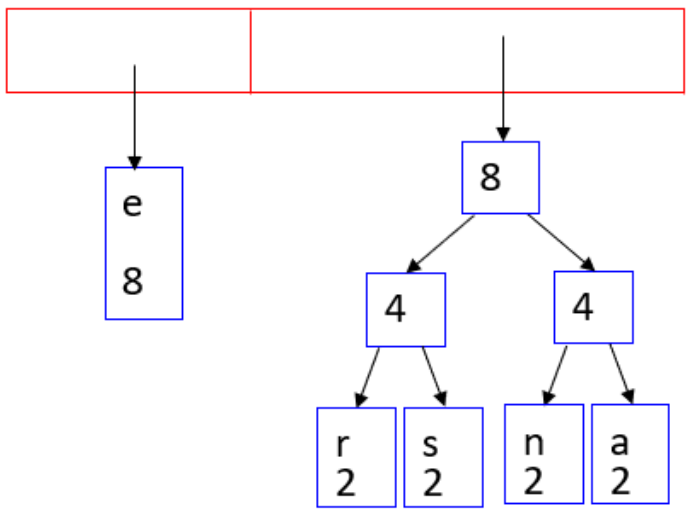
Step 16:

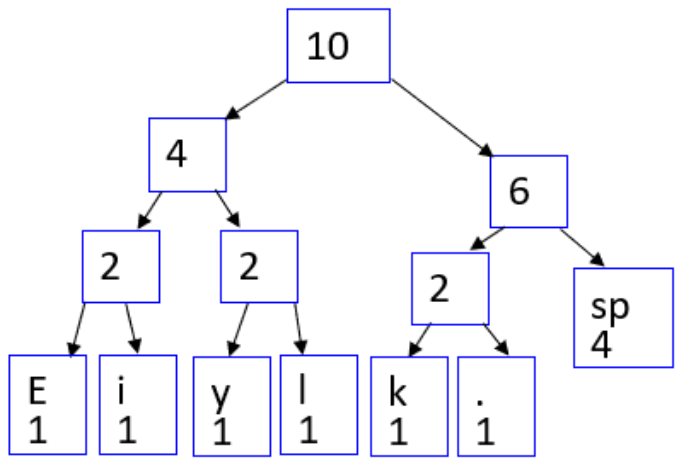


Step 17:

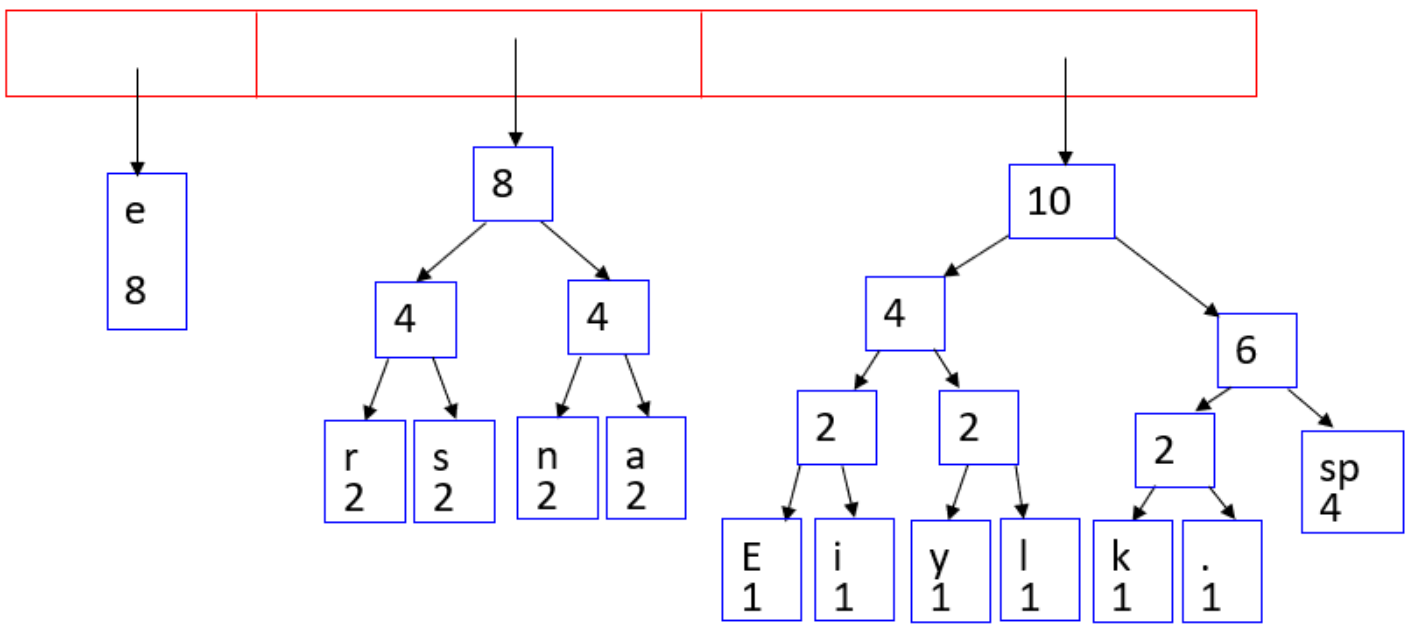


Step 18:

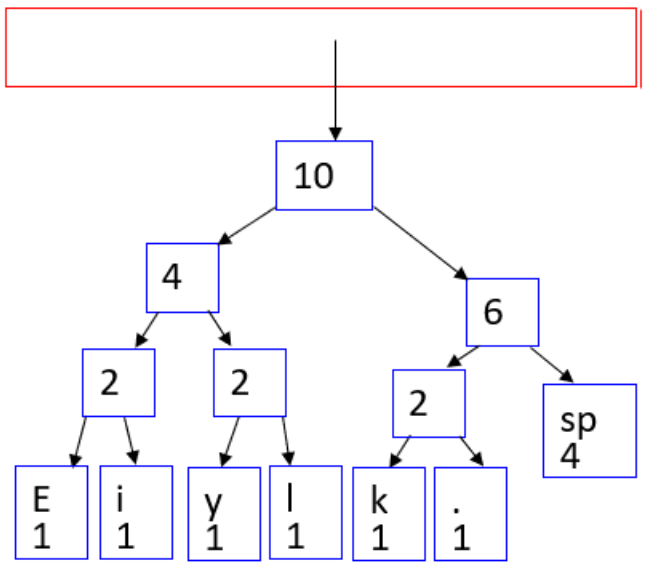


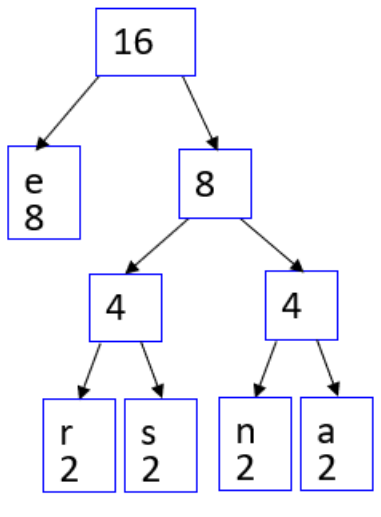


Step 19:

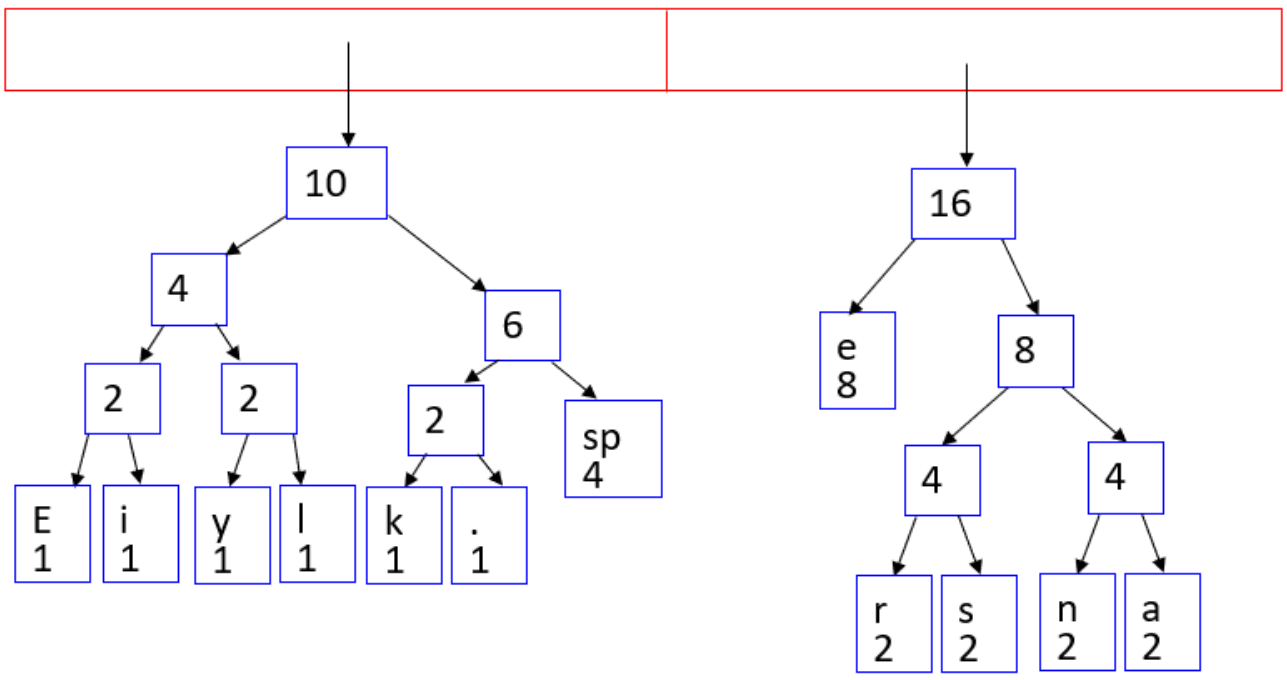


Step 20:

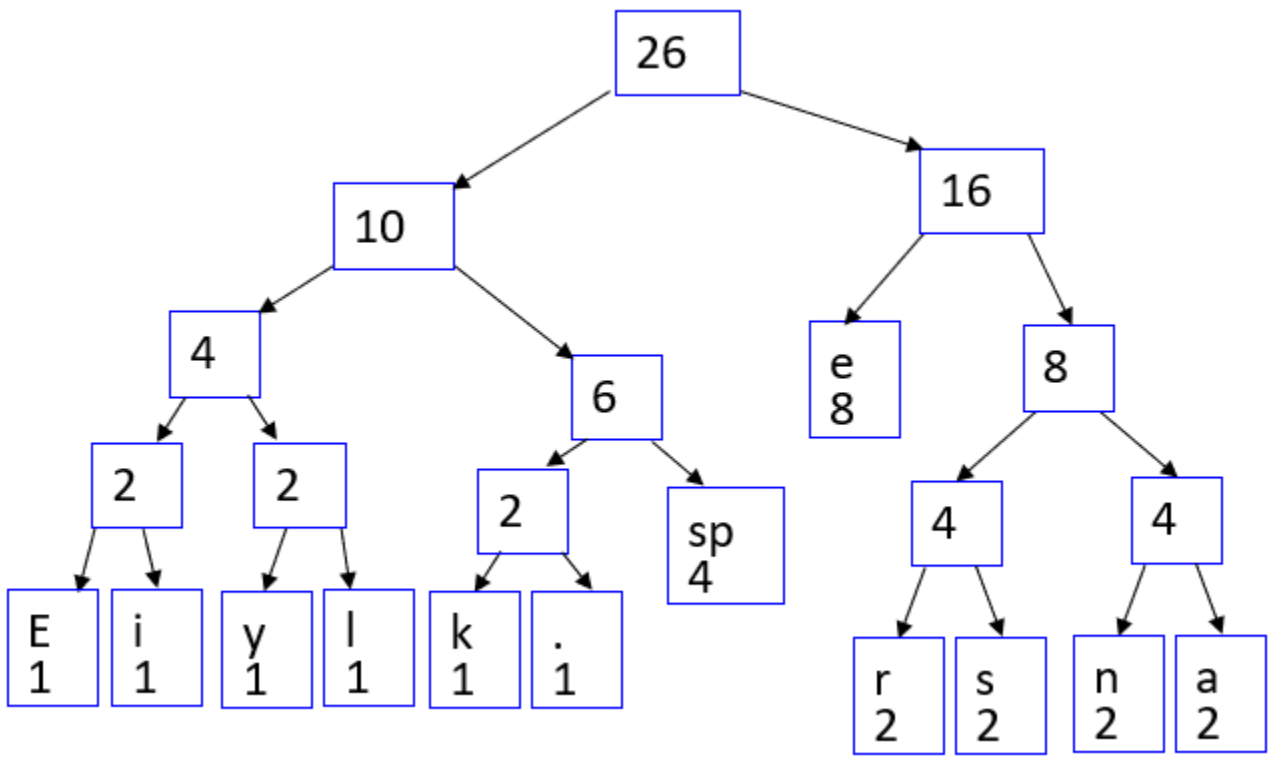




Step 21:



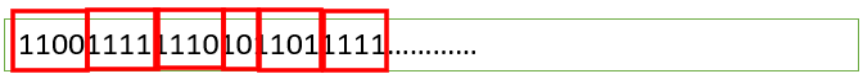
Step 22:



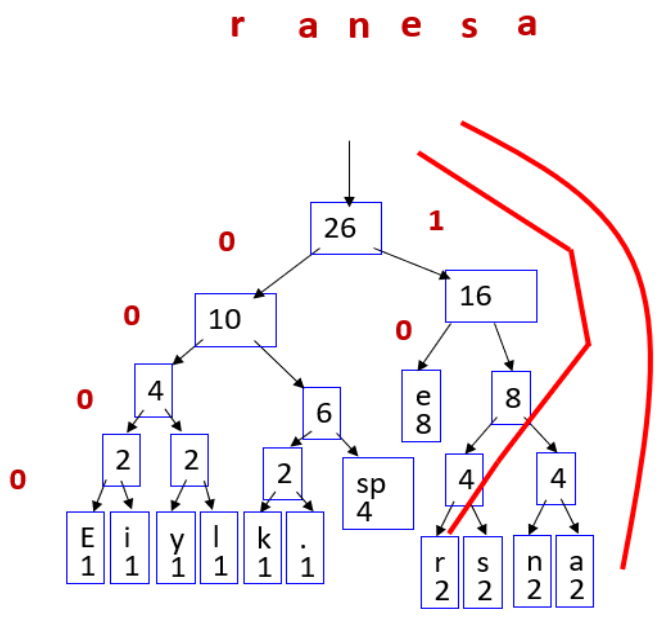
Step 23:

Encoding the File

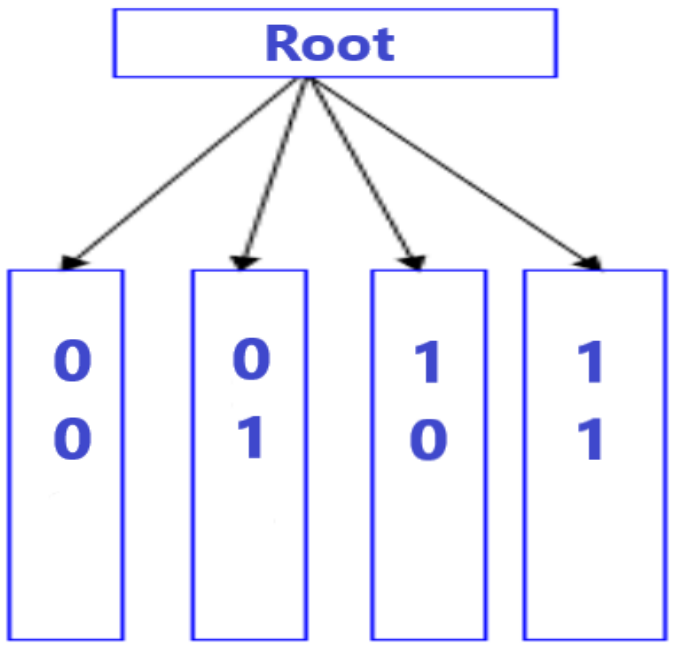
Traverse Tree for Codes



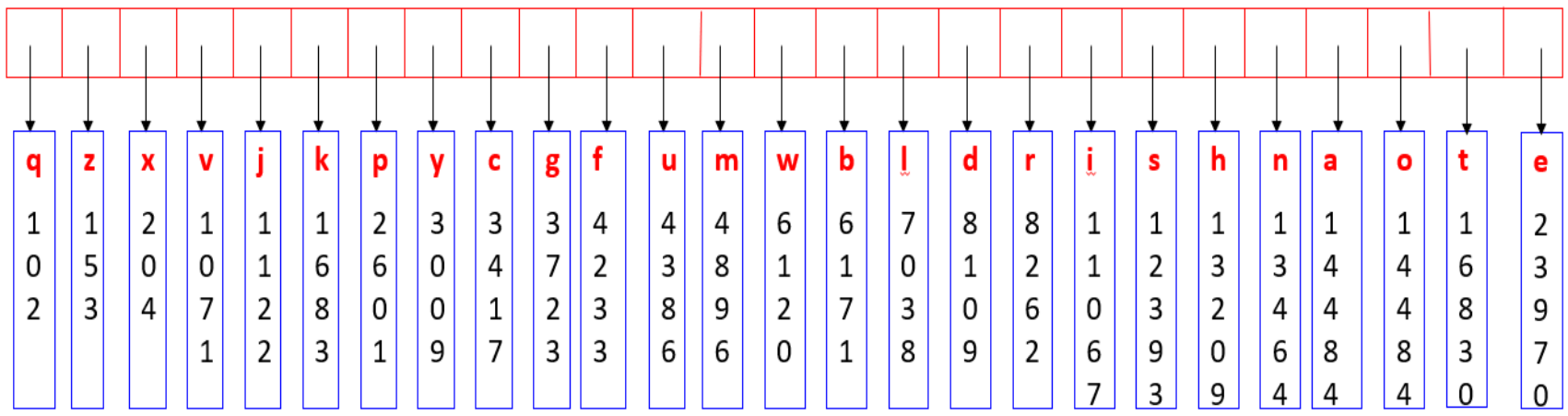
Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111



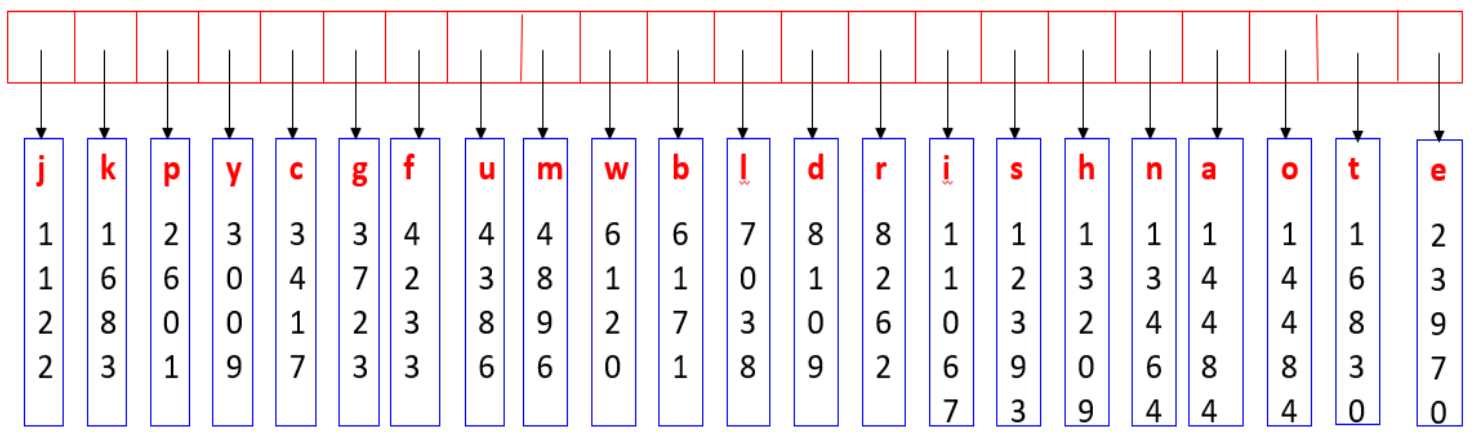
Building Quaternary Tree:

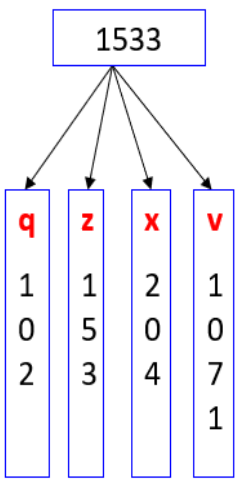


Step 1:

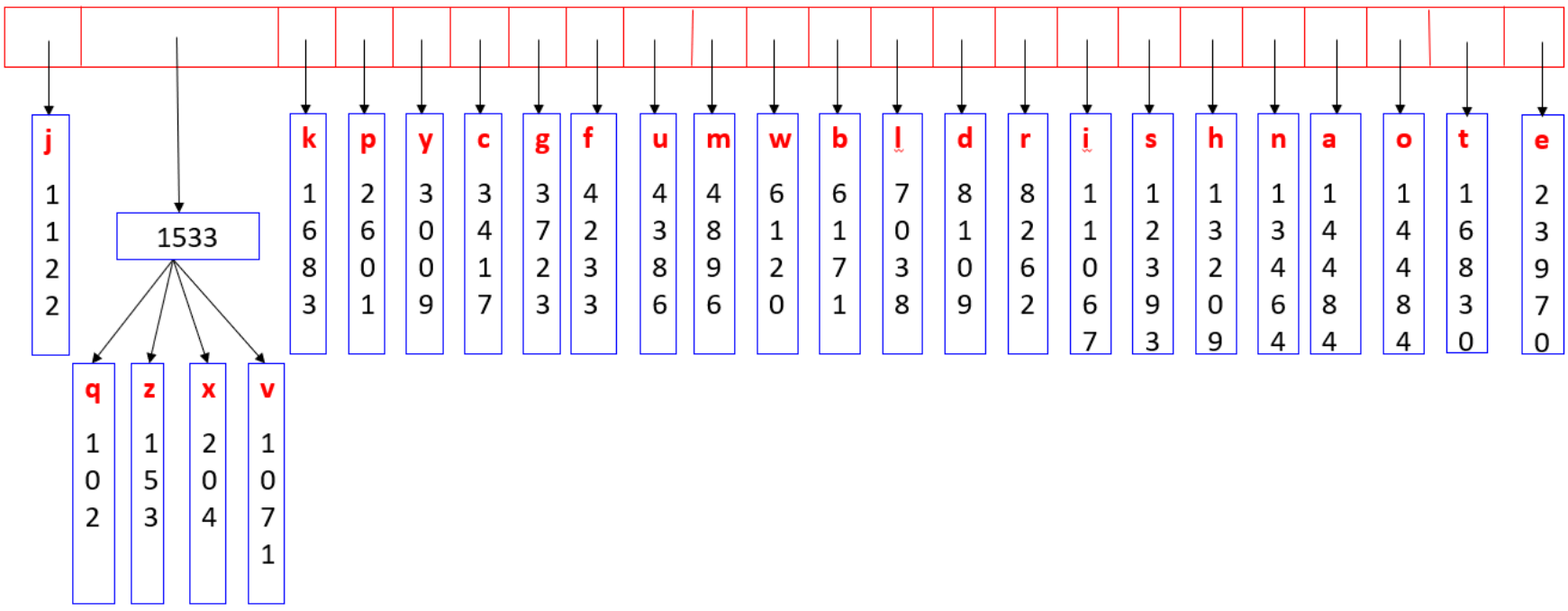


Step 2:

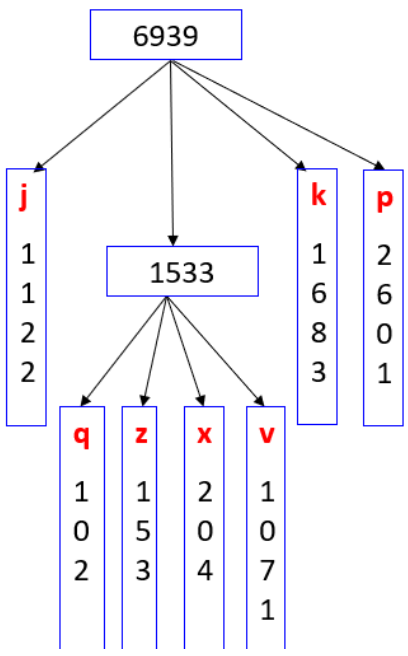


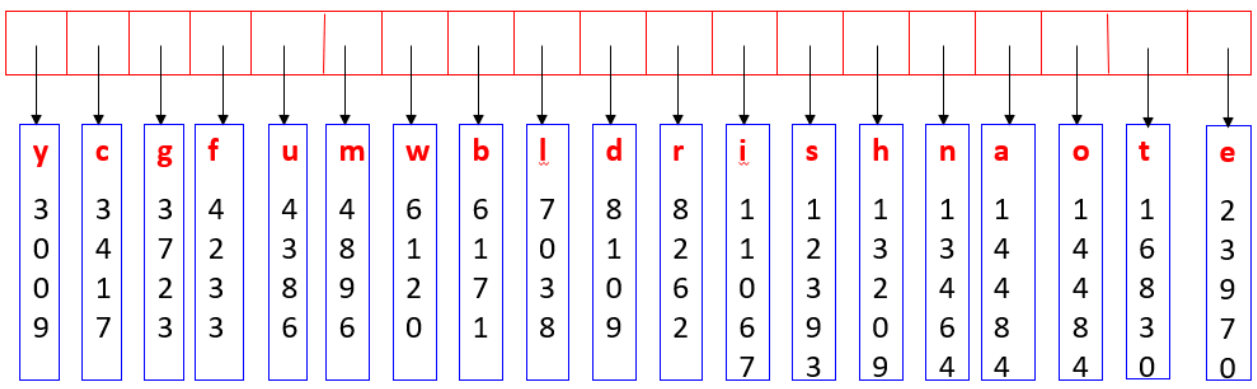


Step 3:

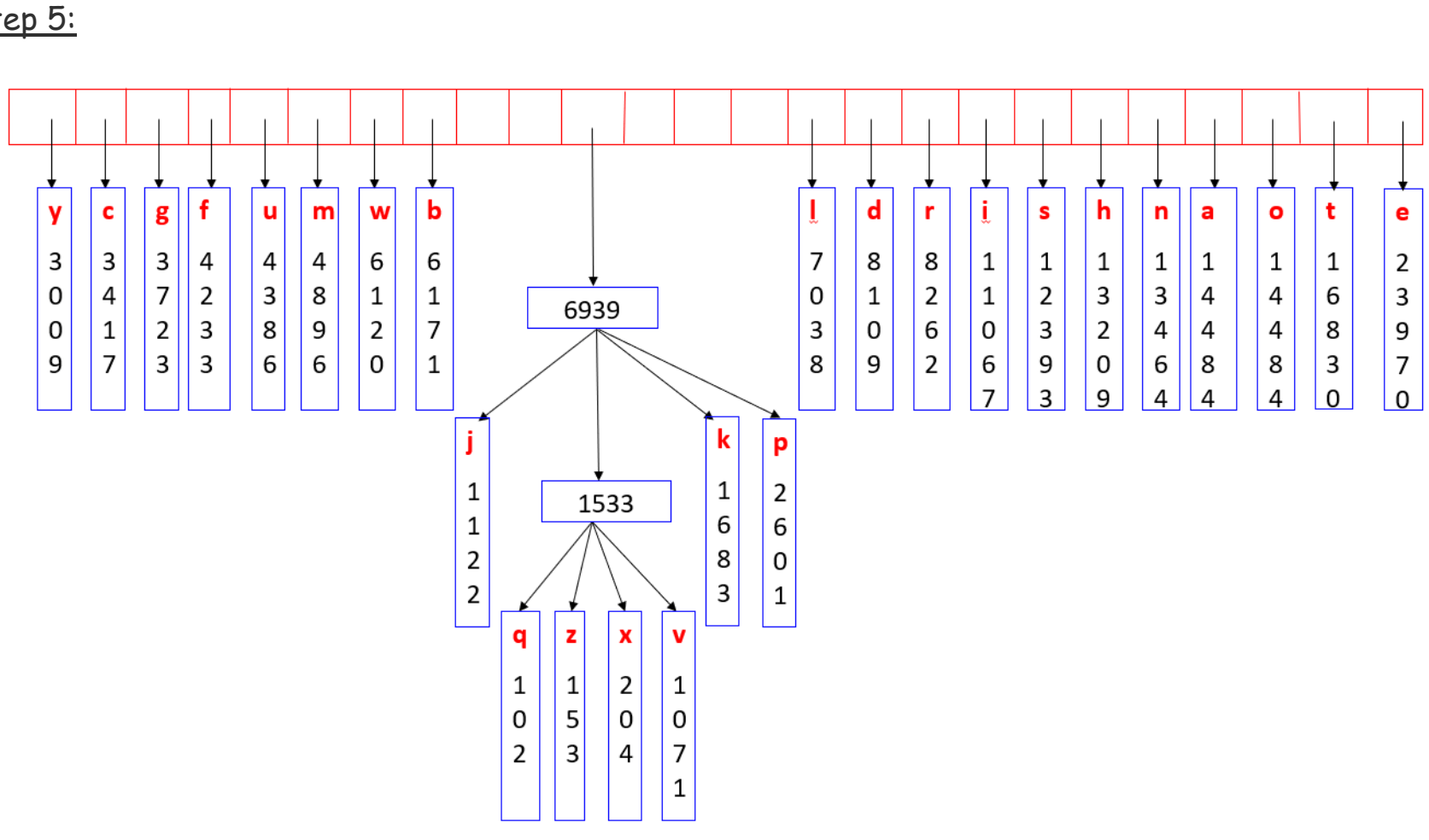


Step 4:

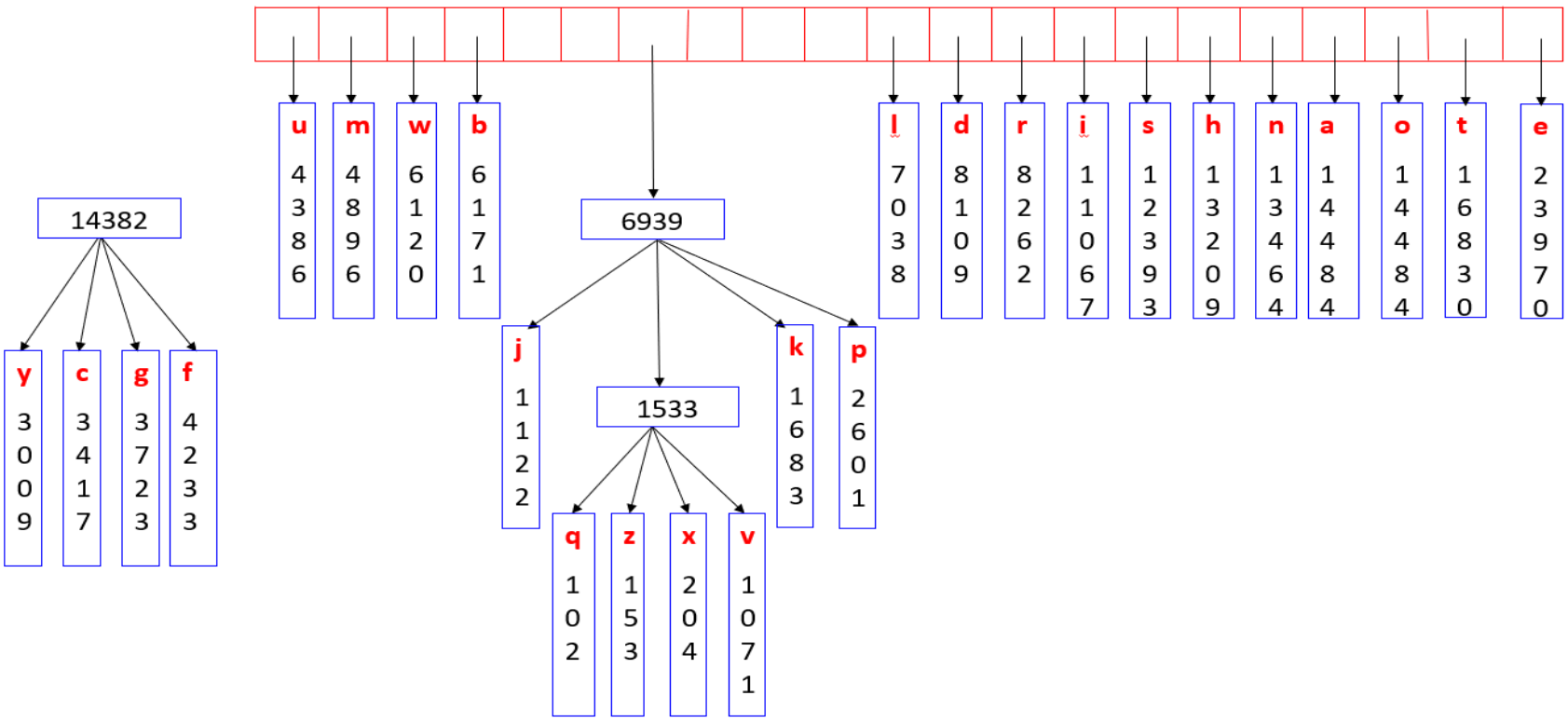




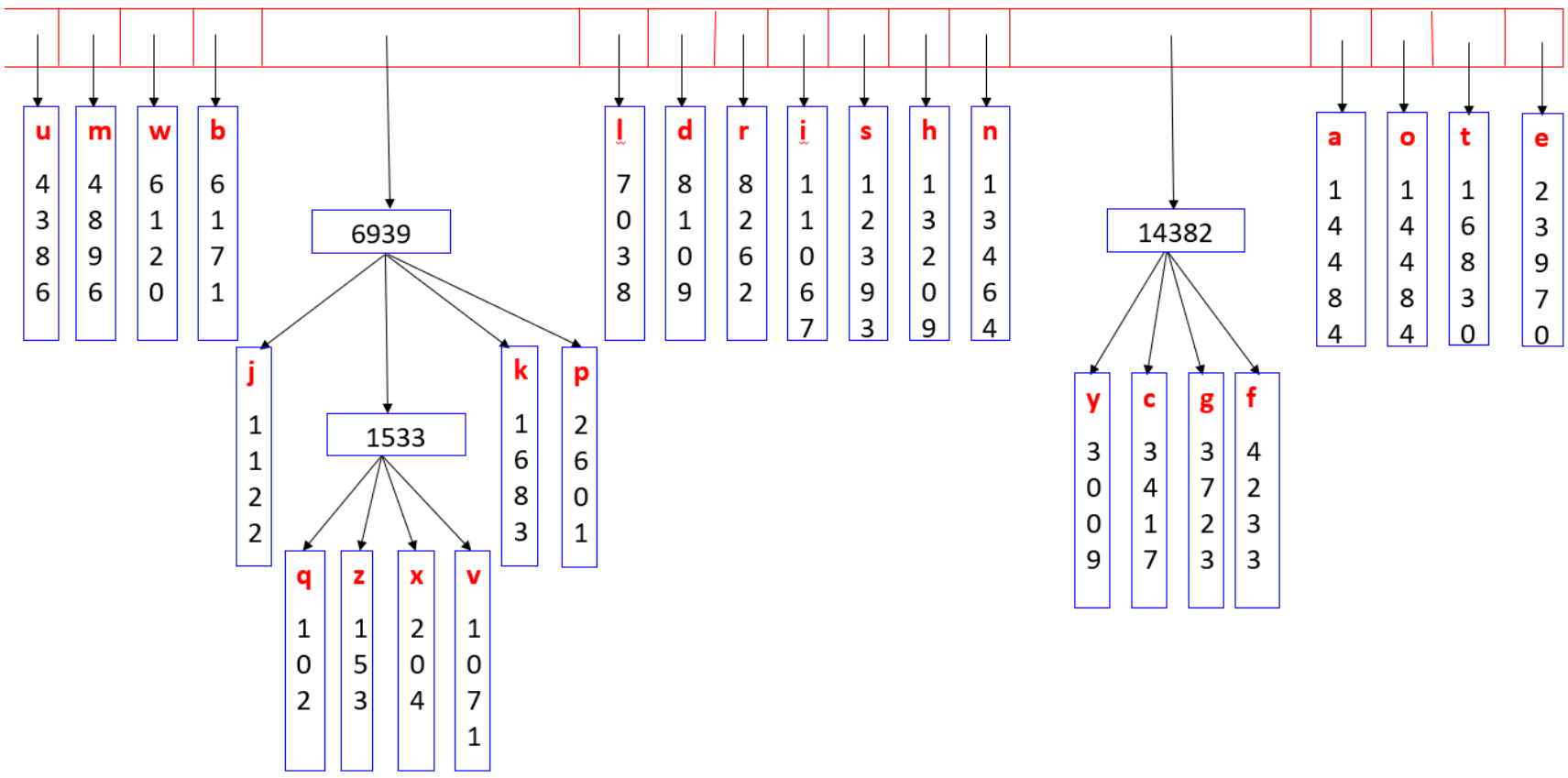
Step 5:



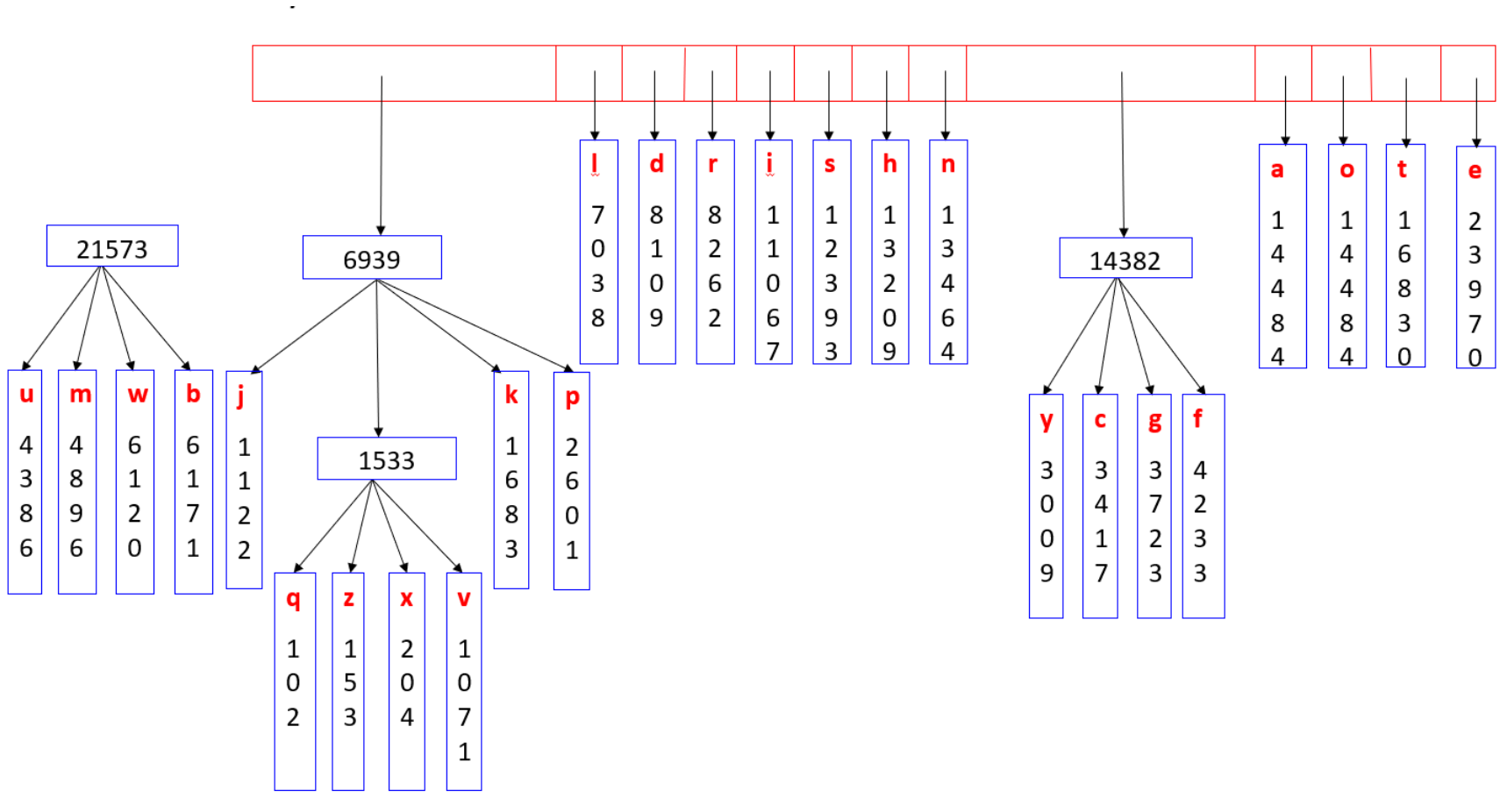
Step 6:



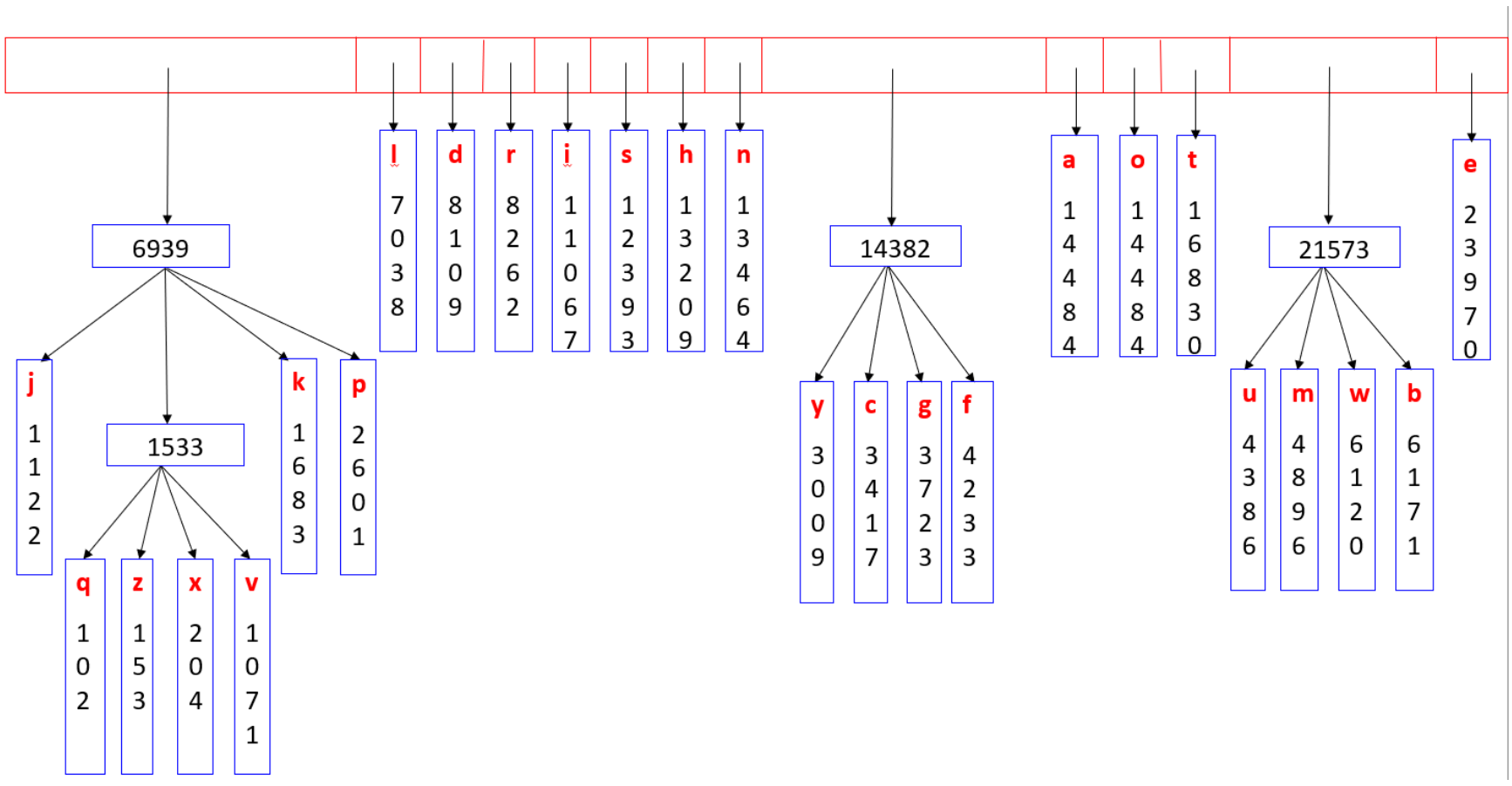
Step 7:



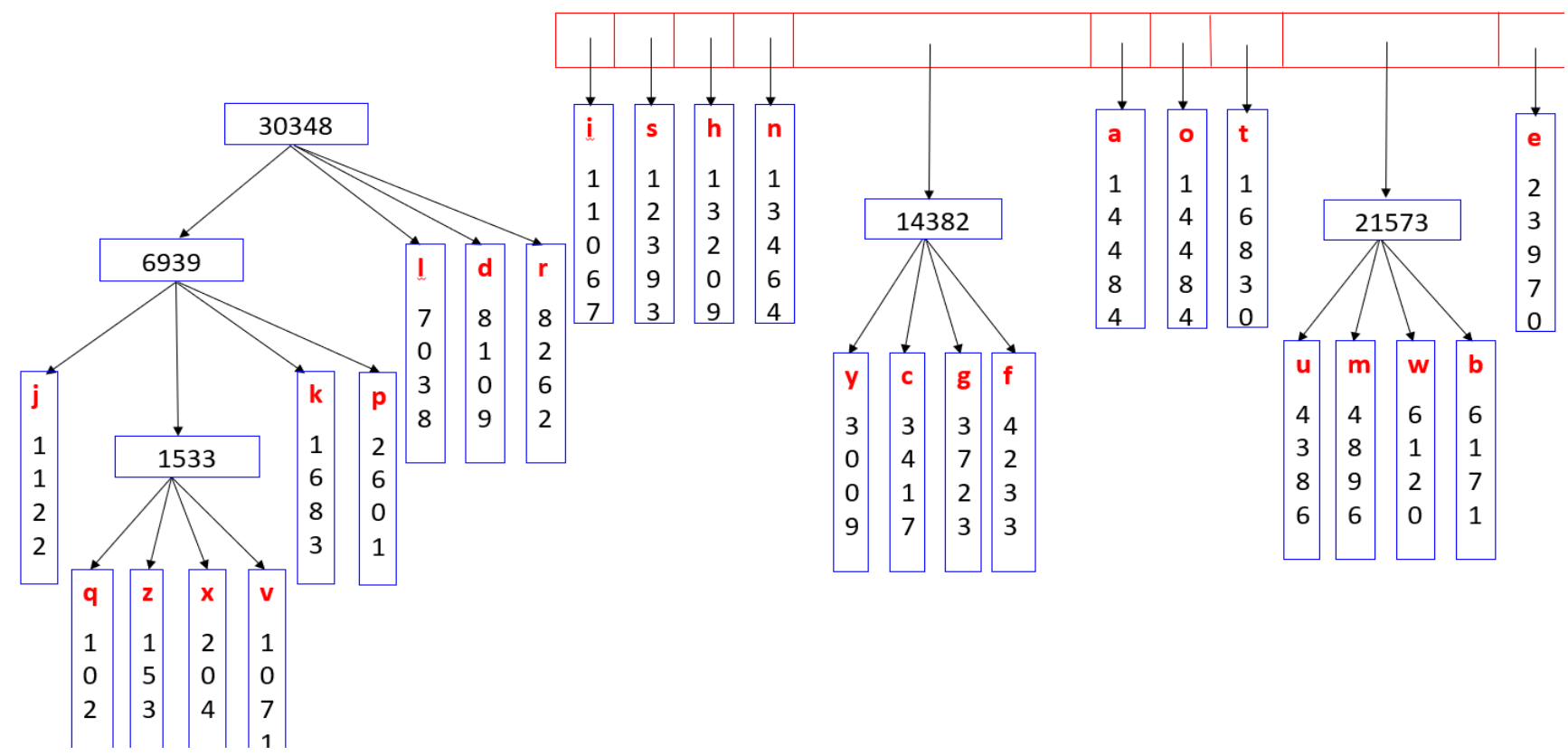
Step 8:



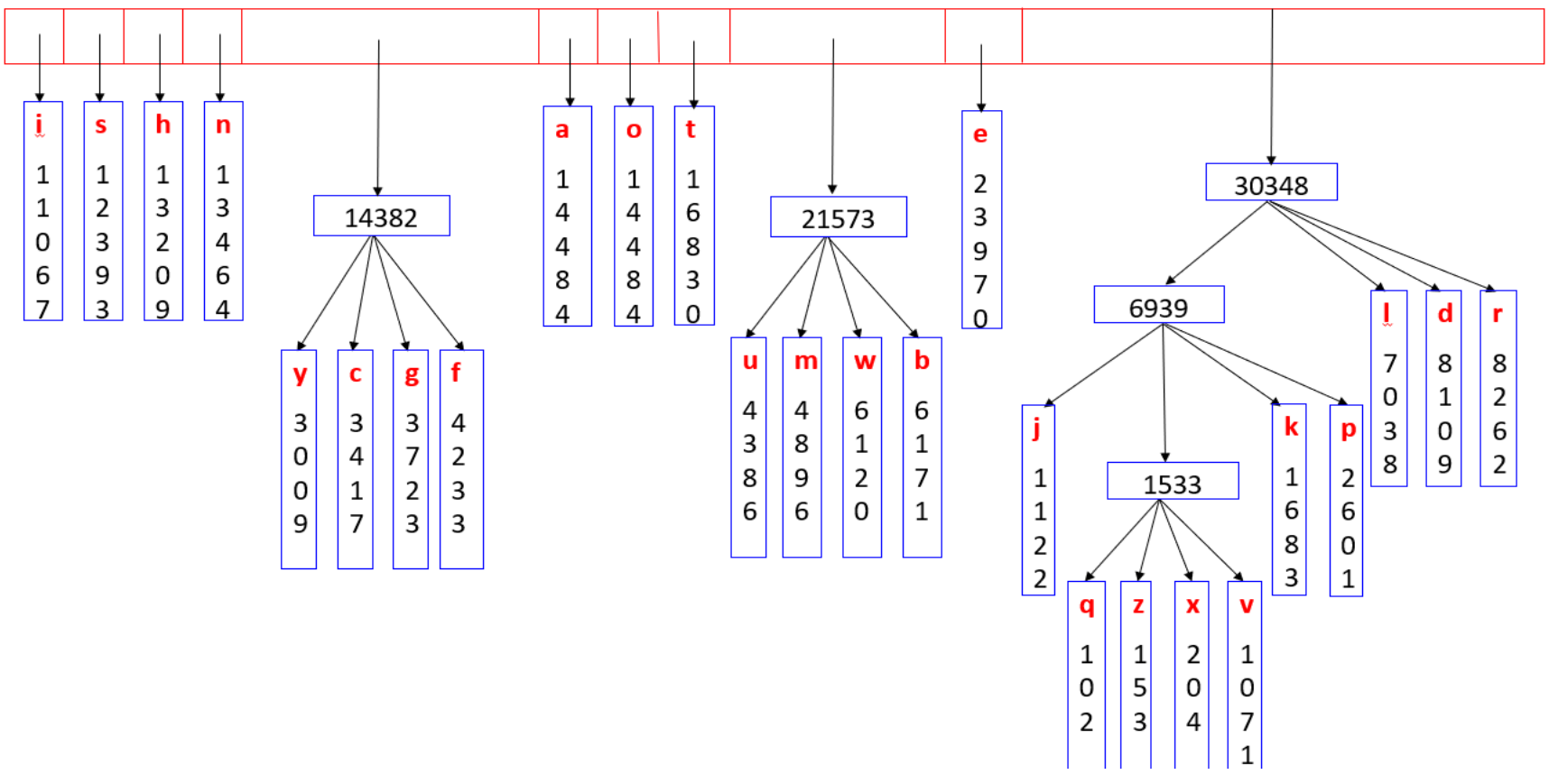
Step 9:



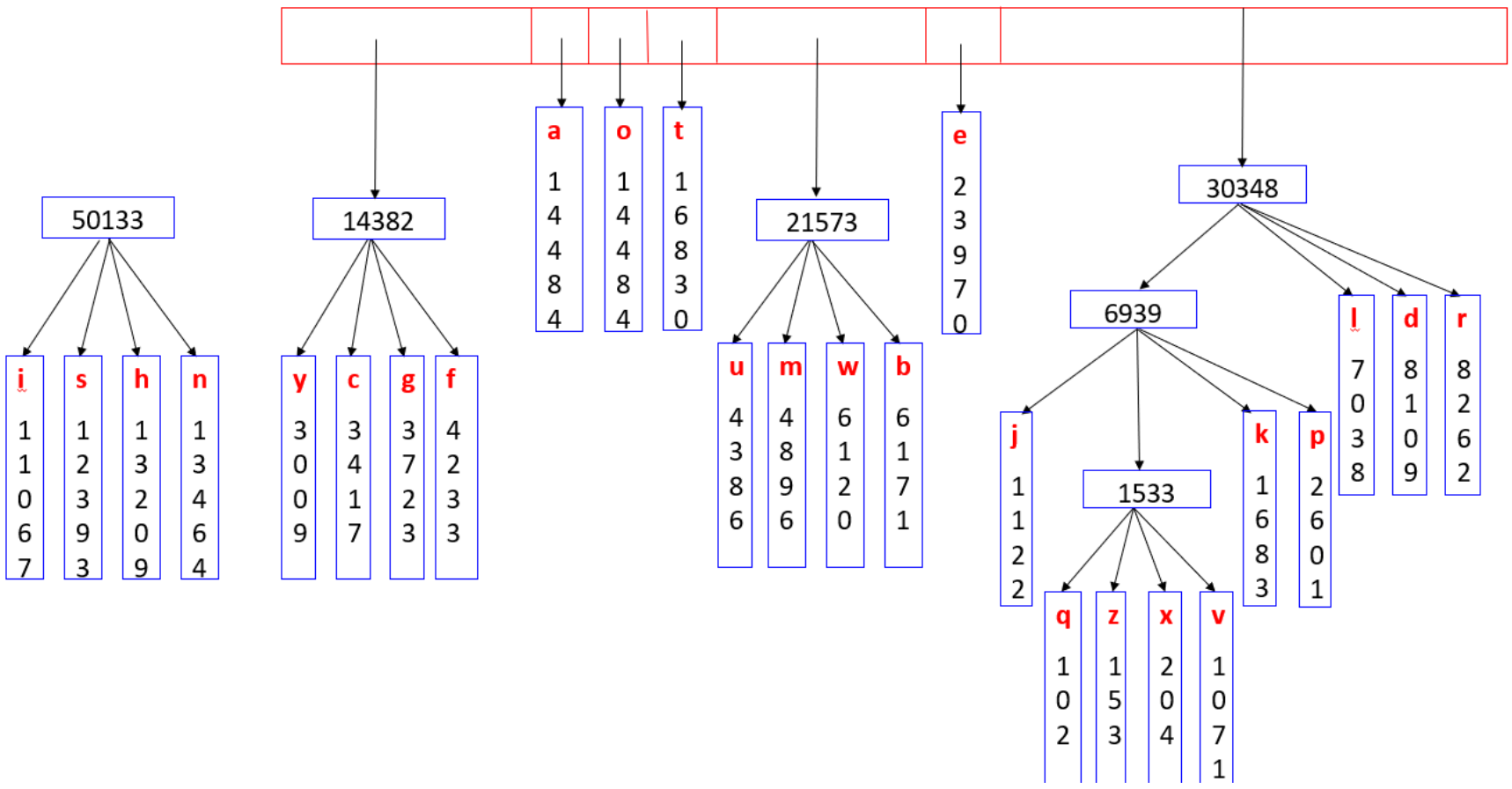
Step 10:



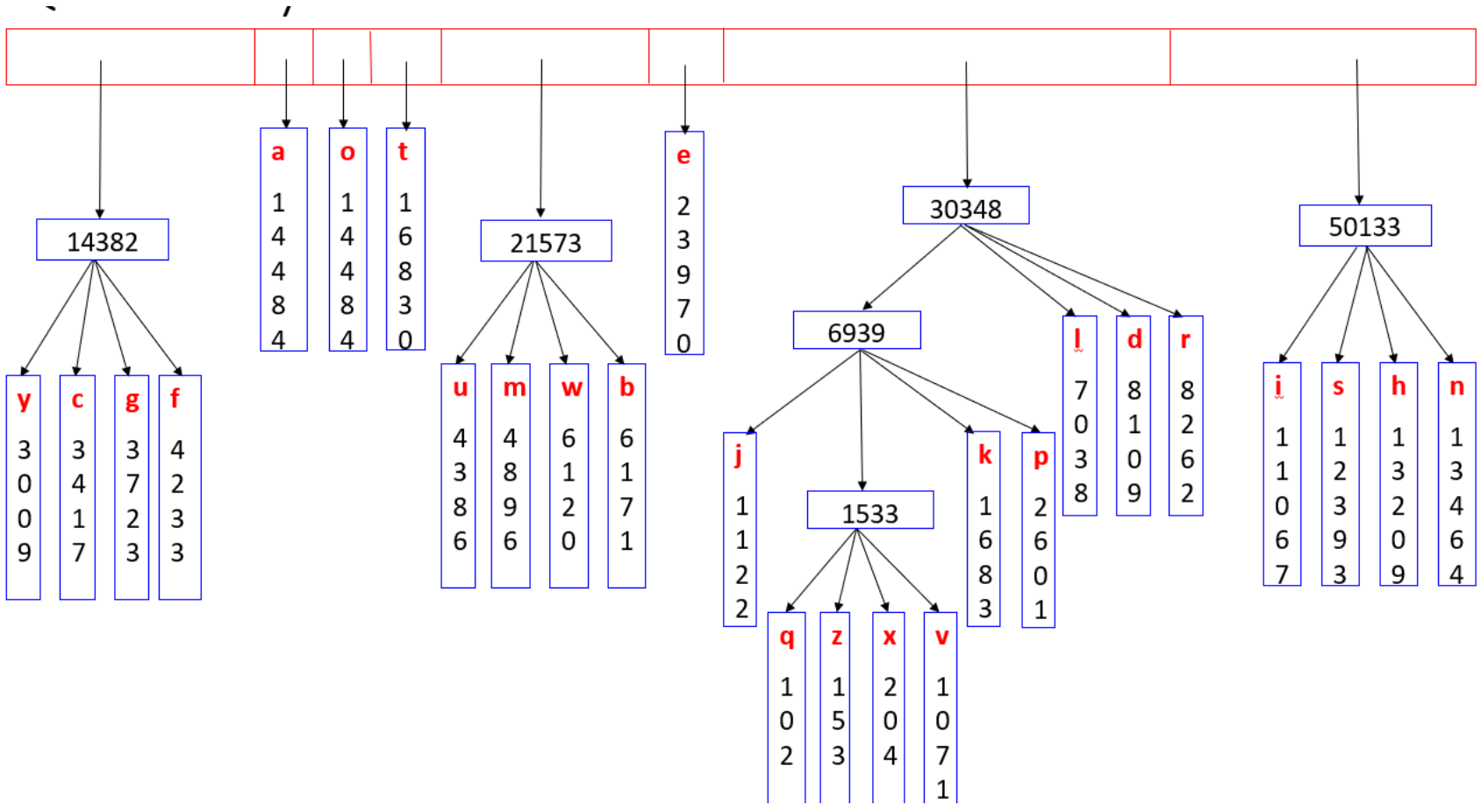
Step 11:



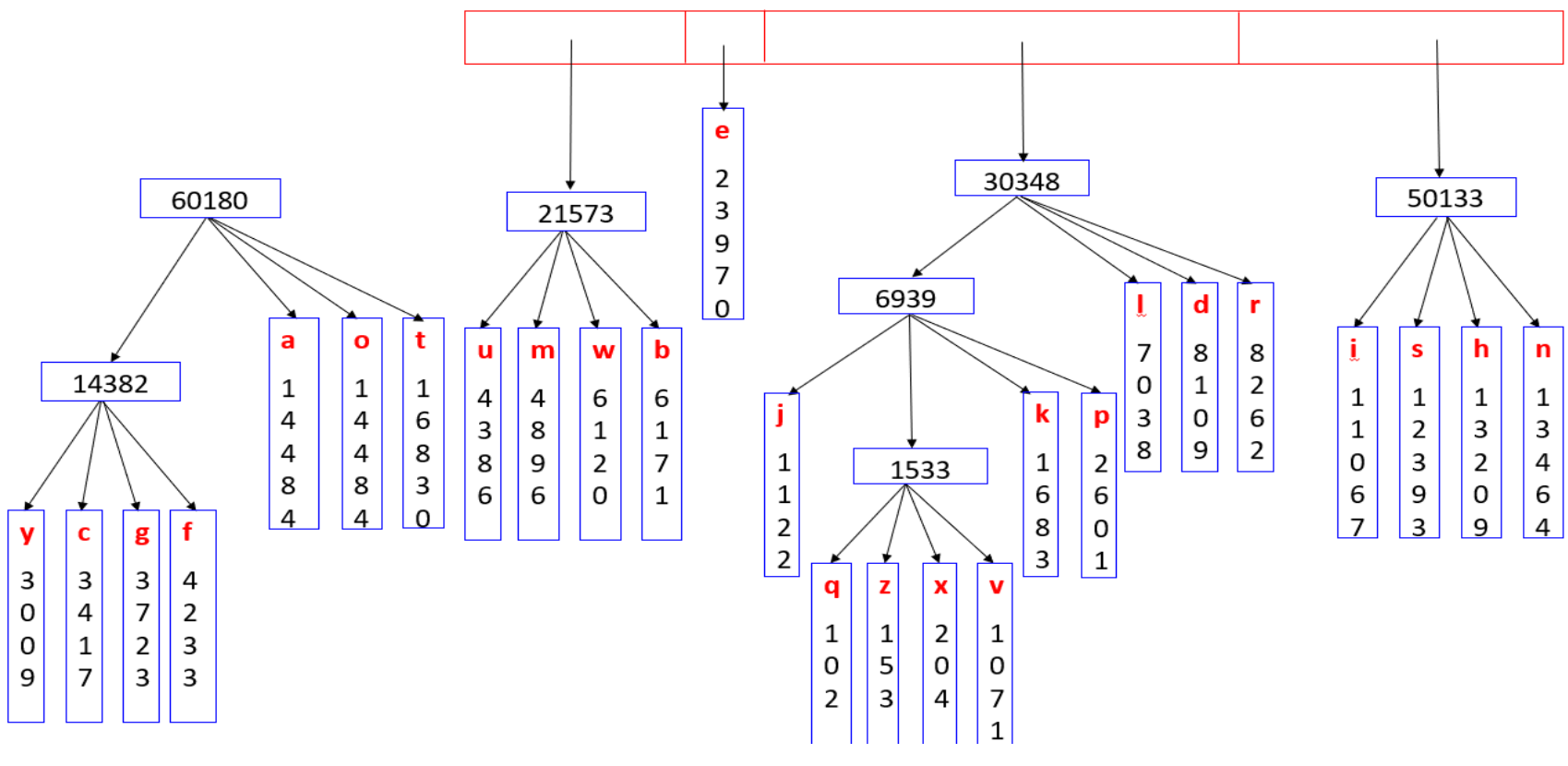
Step 12:



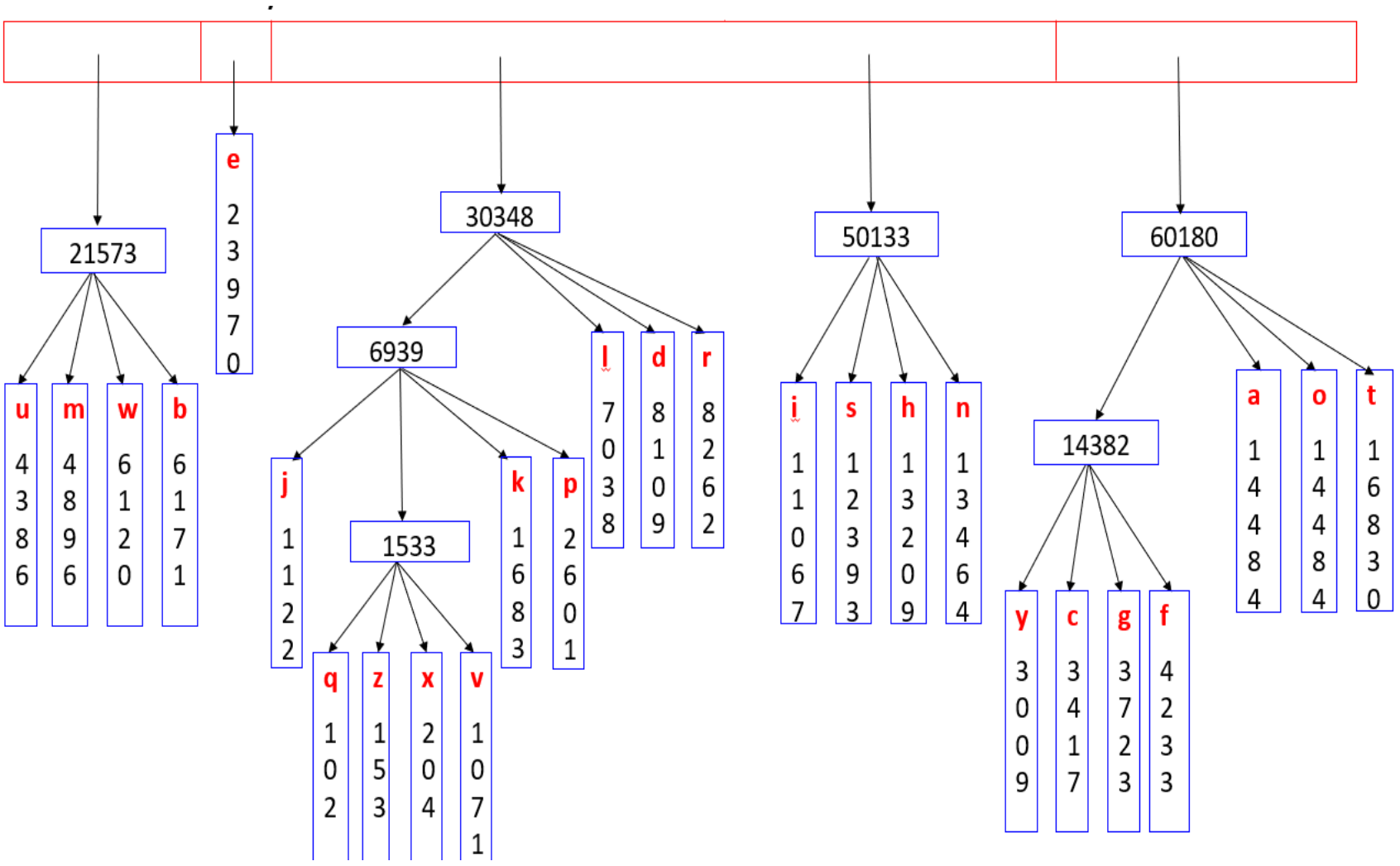
Step 13:



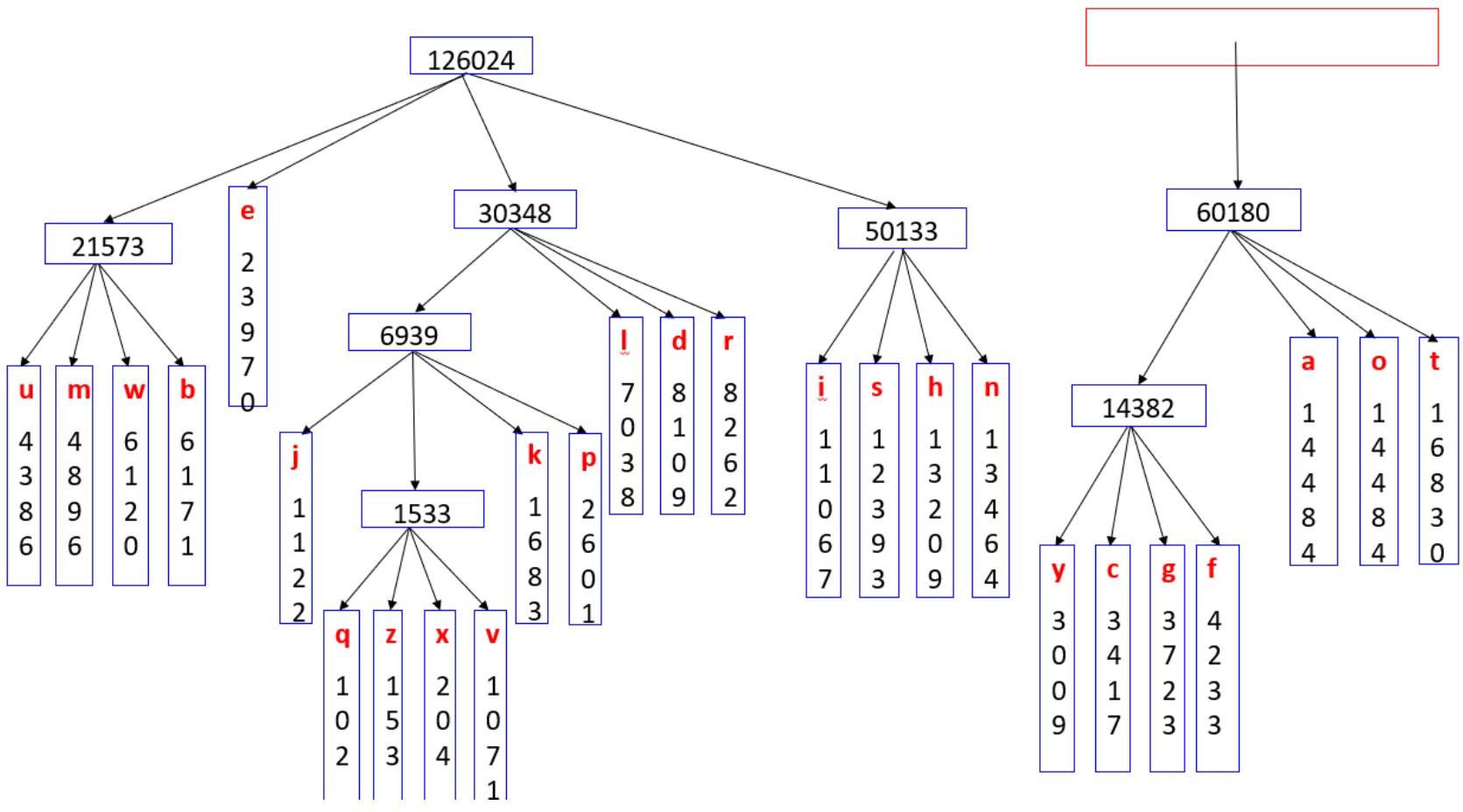
Step 14:



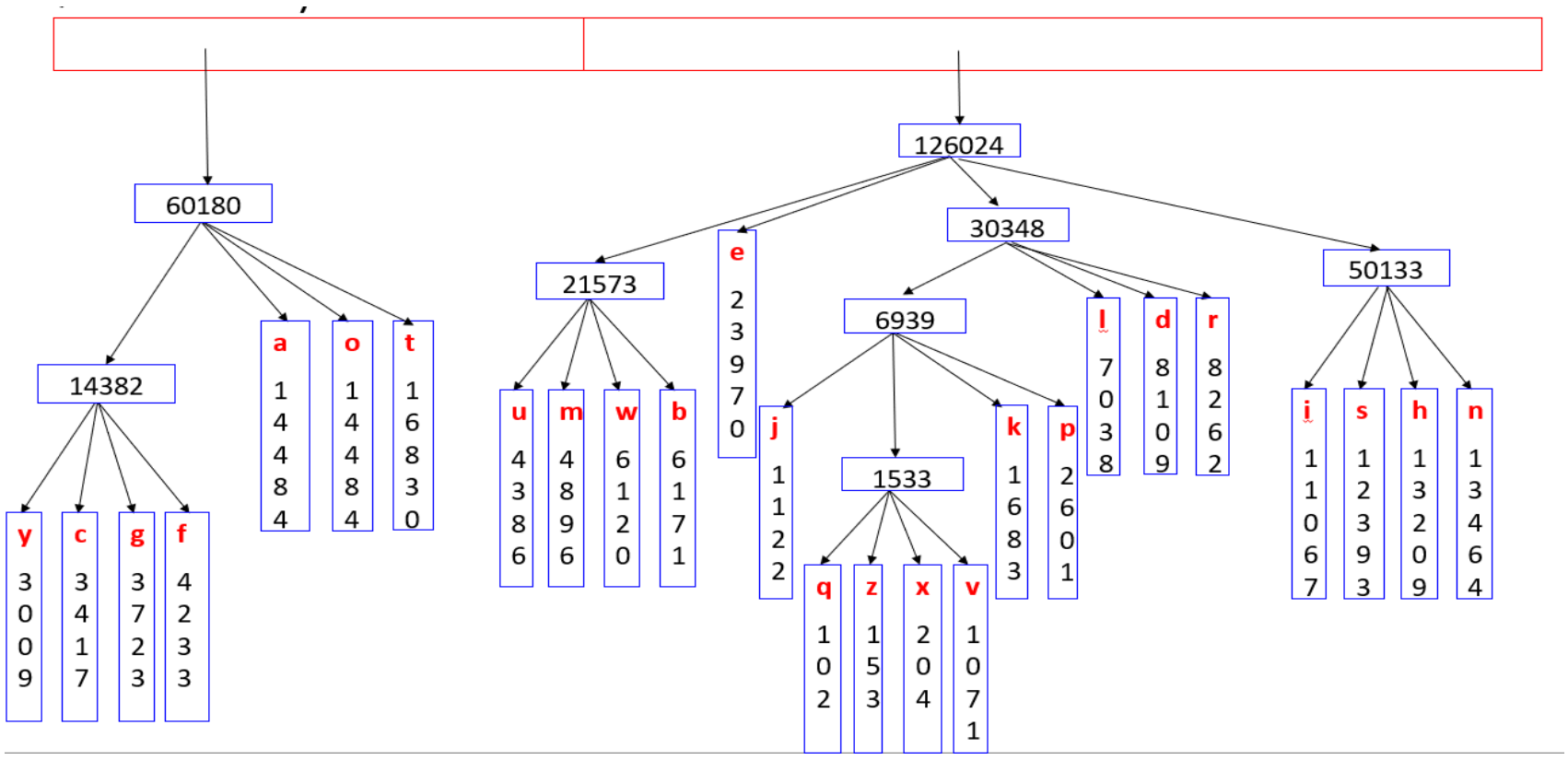
Step 15:



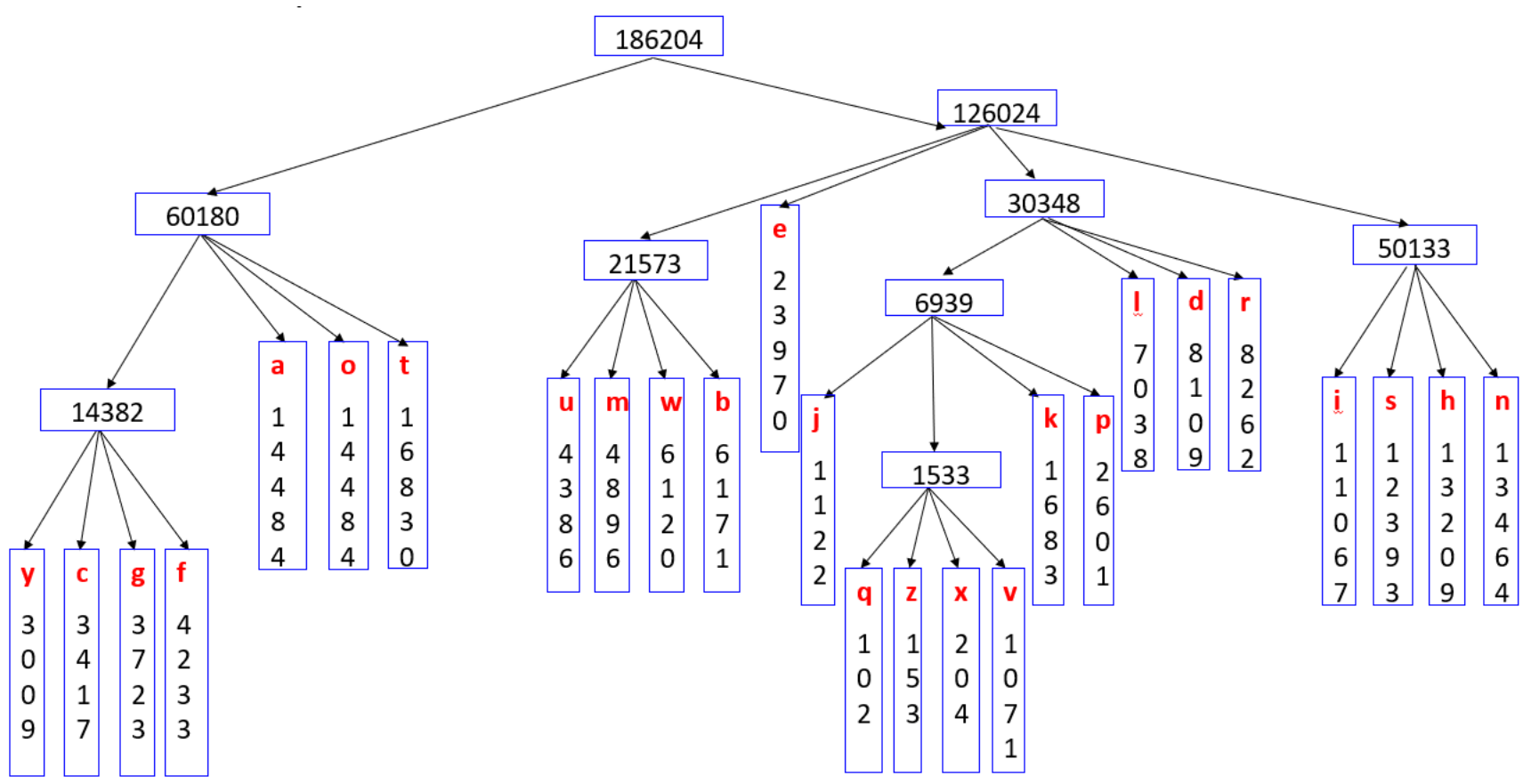
Step 16:



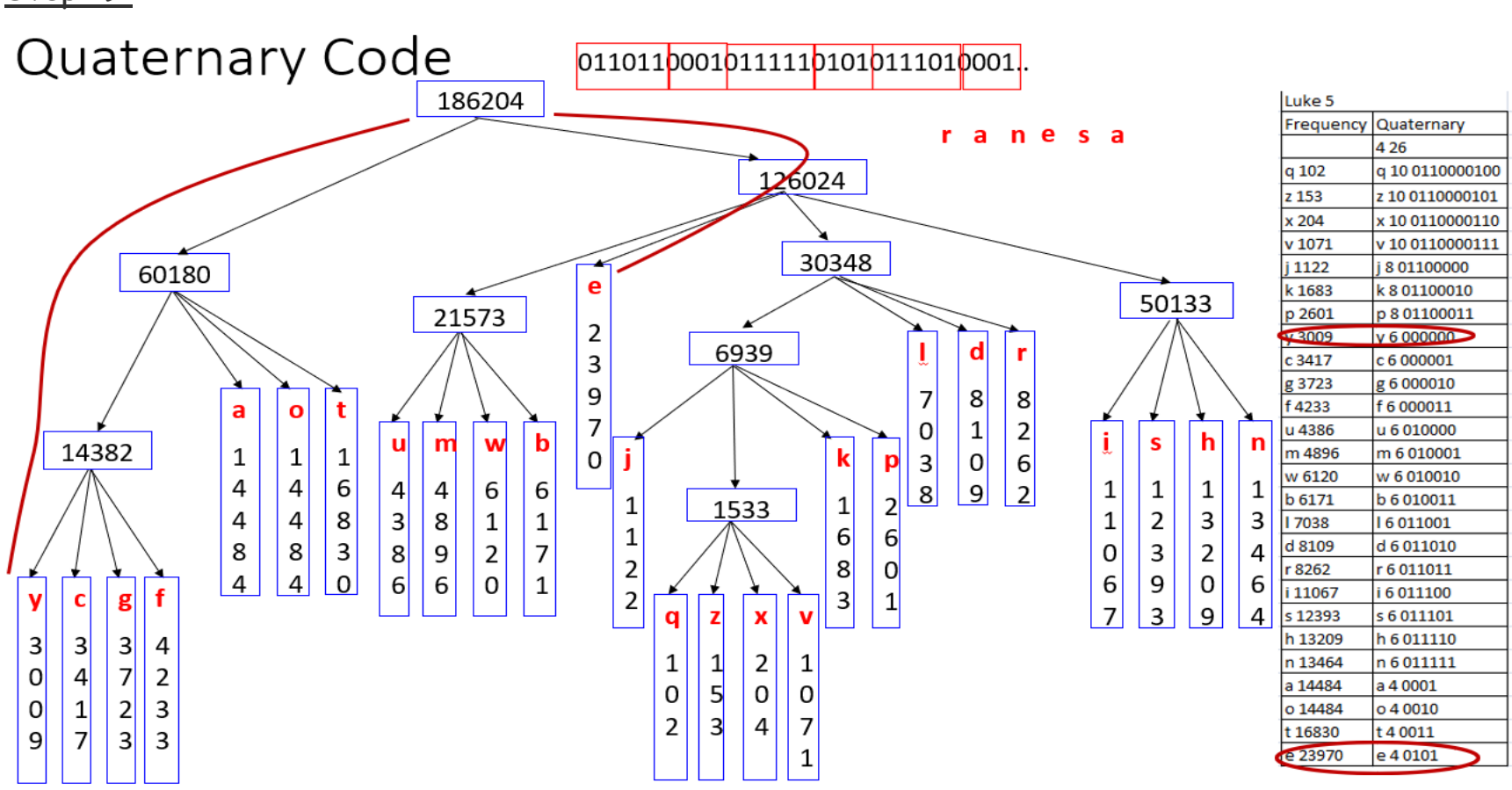
Step 17:



Step 18:

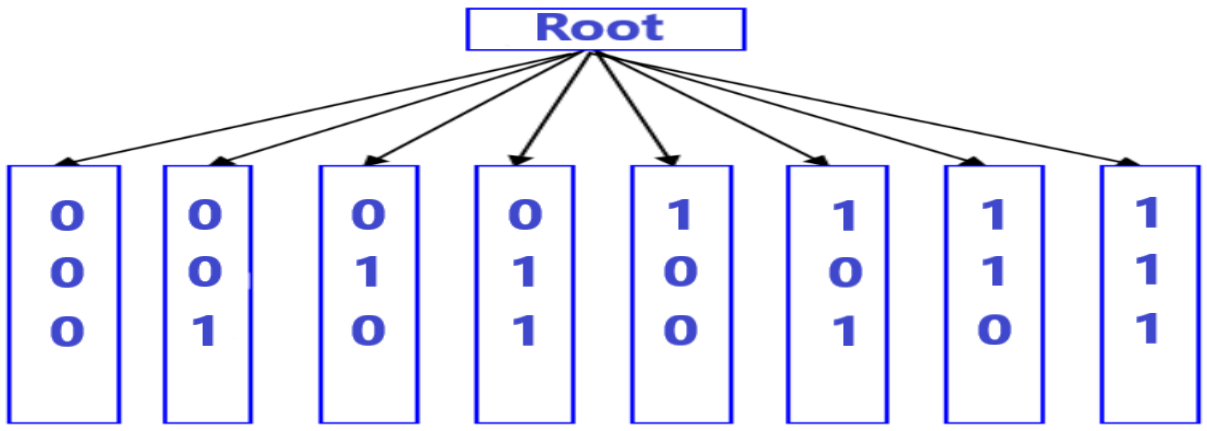


Step 19:

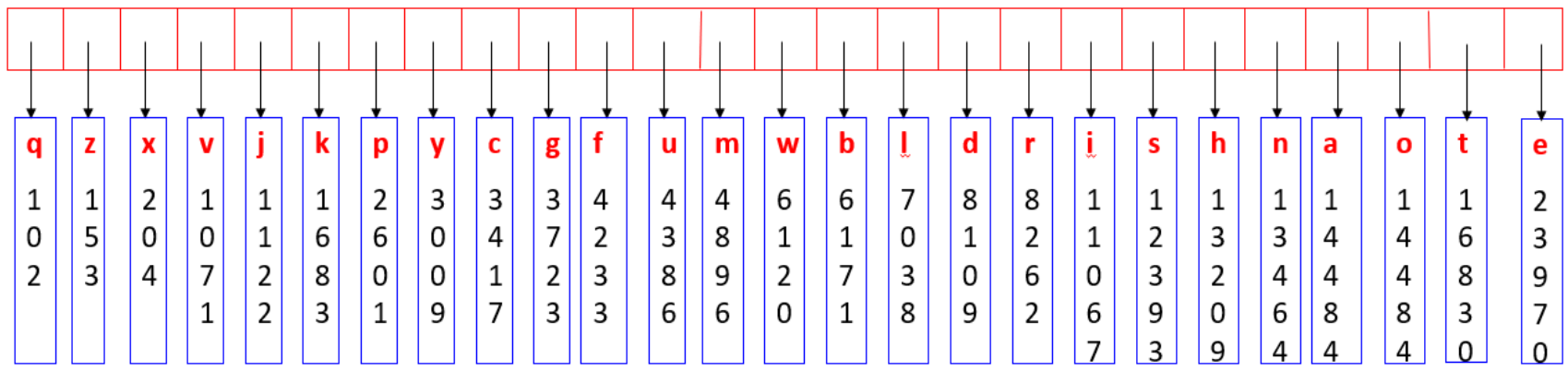


Luke 5	
Frequency	Quaternary
	4 26
q 102	q 10 0110000100
z 153	z 10 0110000101
x 204	x 10 0110000110
v 1071	v 10 0110000111
j 1122	j 8 01100000
k 1683	k 8 01100010
p 2601	p 8 01100011
y 3009	y 6 000000
c 3417	c 6 000001
g 3723	g 6 000010
f 4233	f 6 000011
u 4386	u 6 010000
m 4896	m 6 010001
w 6120	w 6 010010
b 6171	b 6 010011
l 7038	l 6 011001
d 8109	d 6 011010
r 8262	r 6 011011
i 11067	i 6 011100
s 12393	s 6 011101
h 13209	h 6 011110
n 13464	n 6 011111
a 14484	a 4 0001
o 14484	o 4 0010
t 16830	t 4 0011
e 23970	e 4 0101

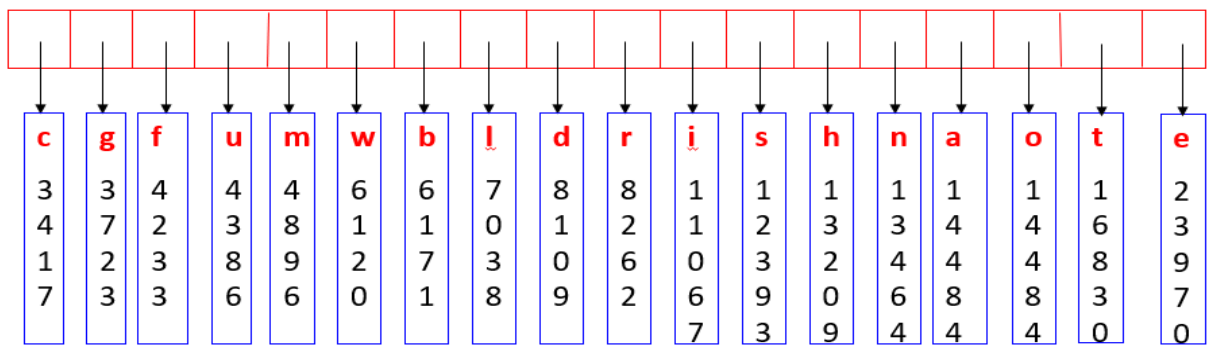
Building Octonary Tree

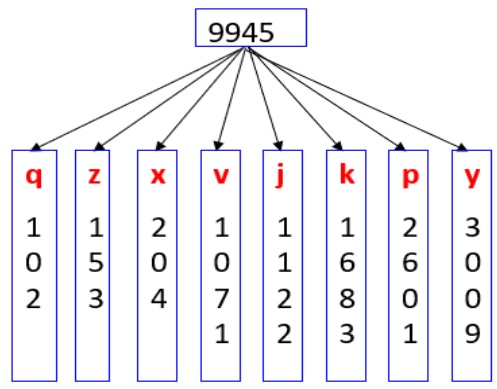


Step 1:

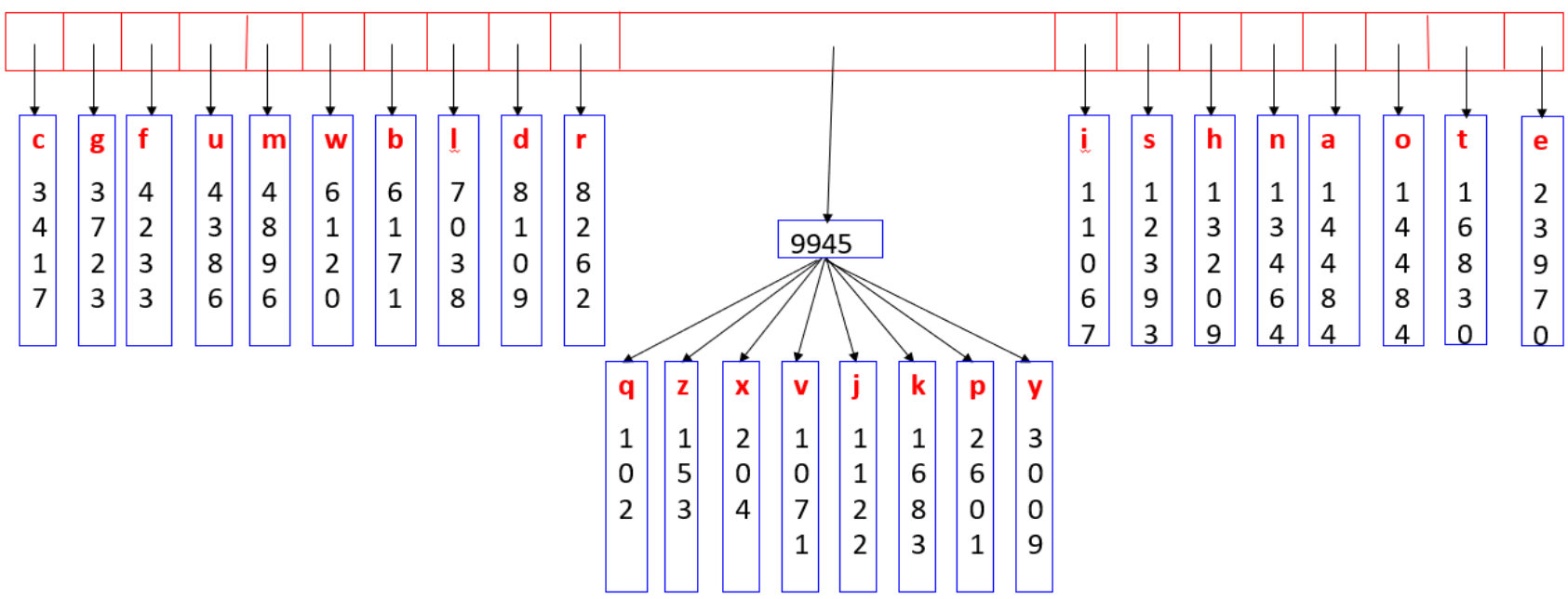


Step 2:

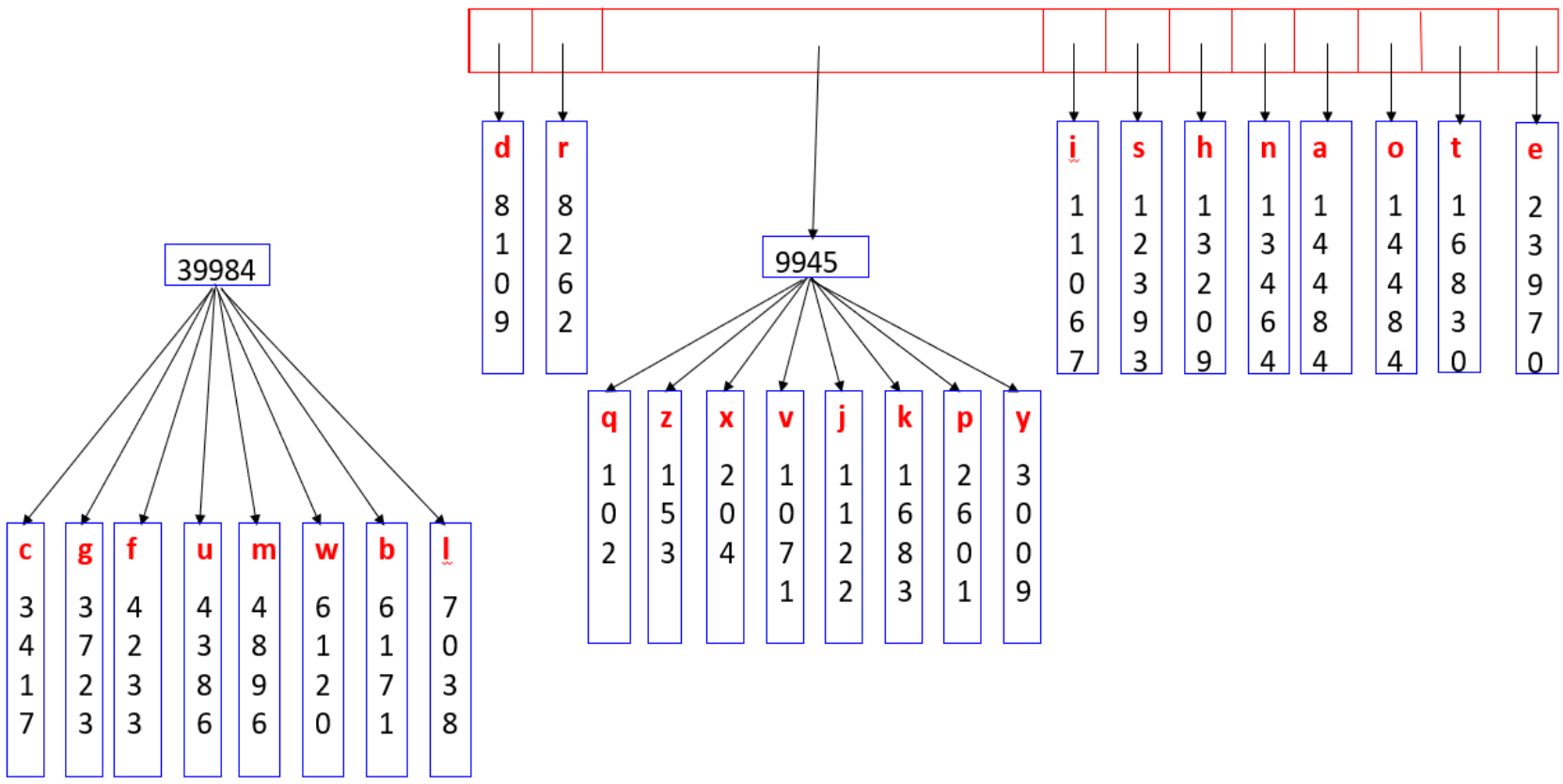




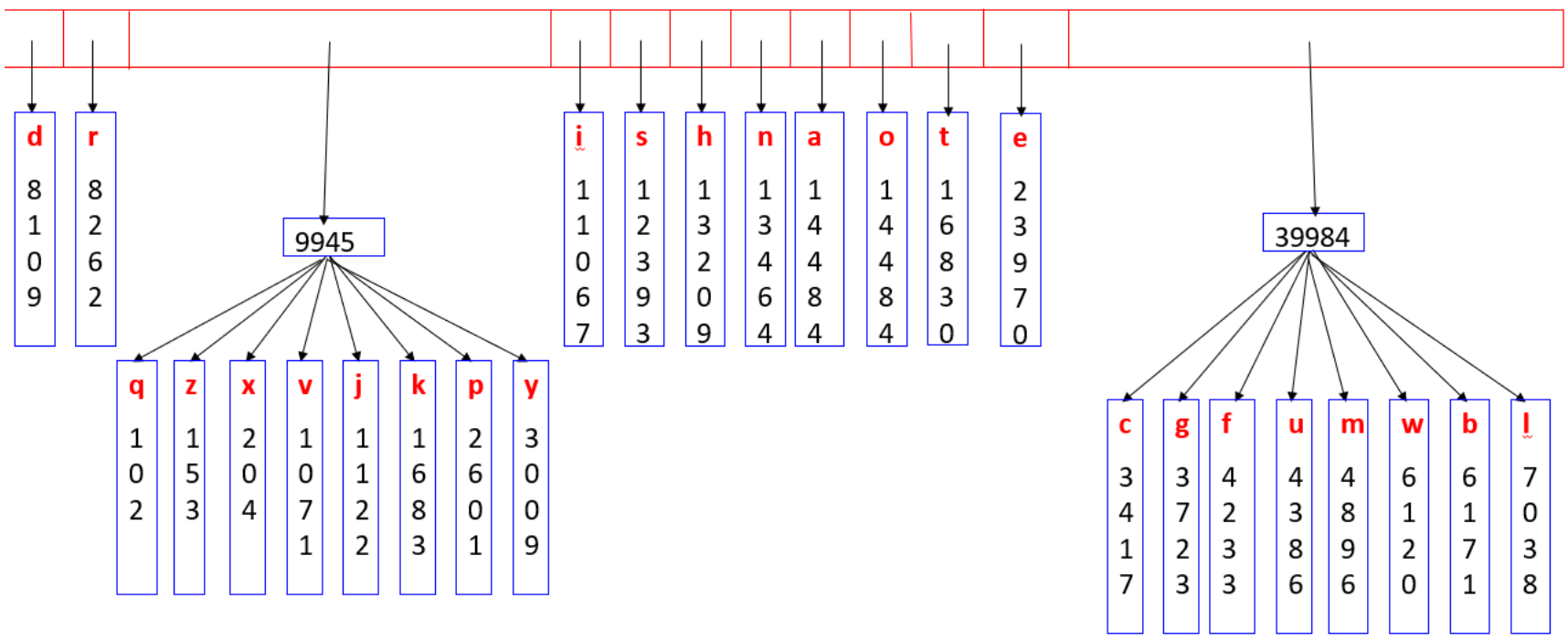
Step 3:



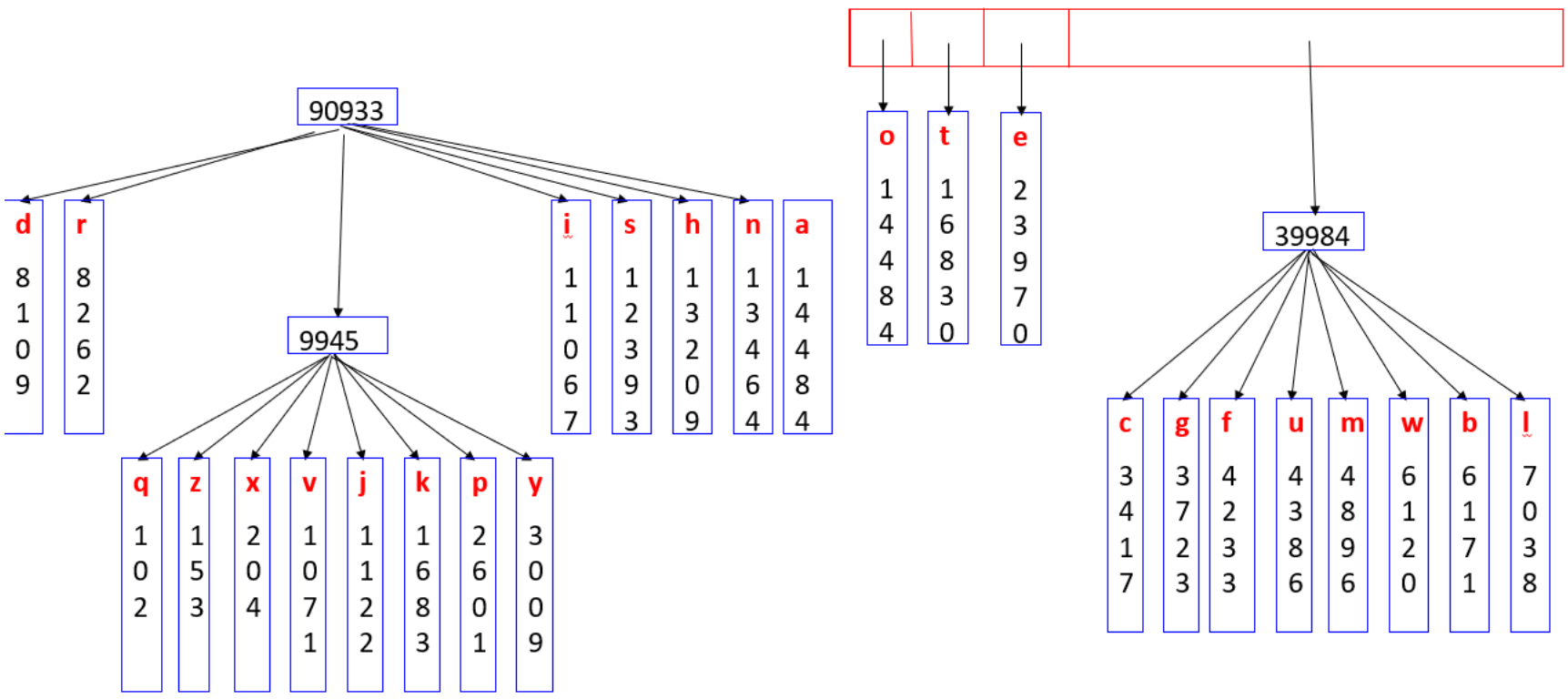
Step 4:



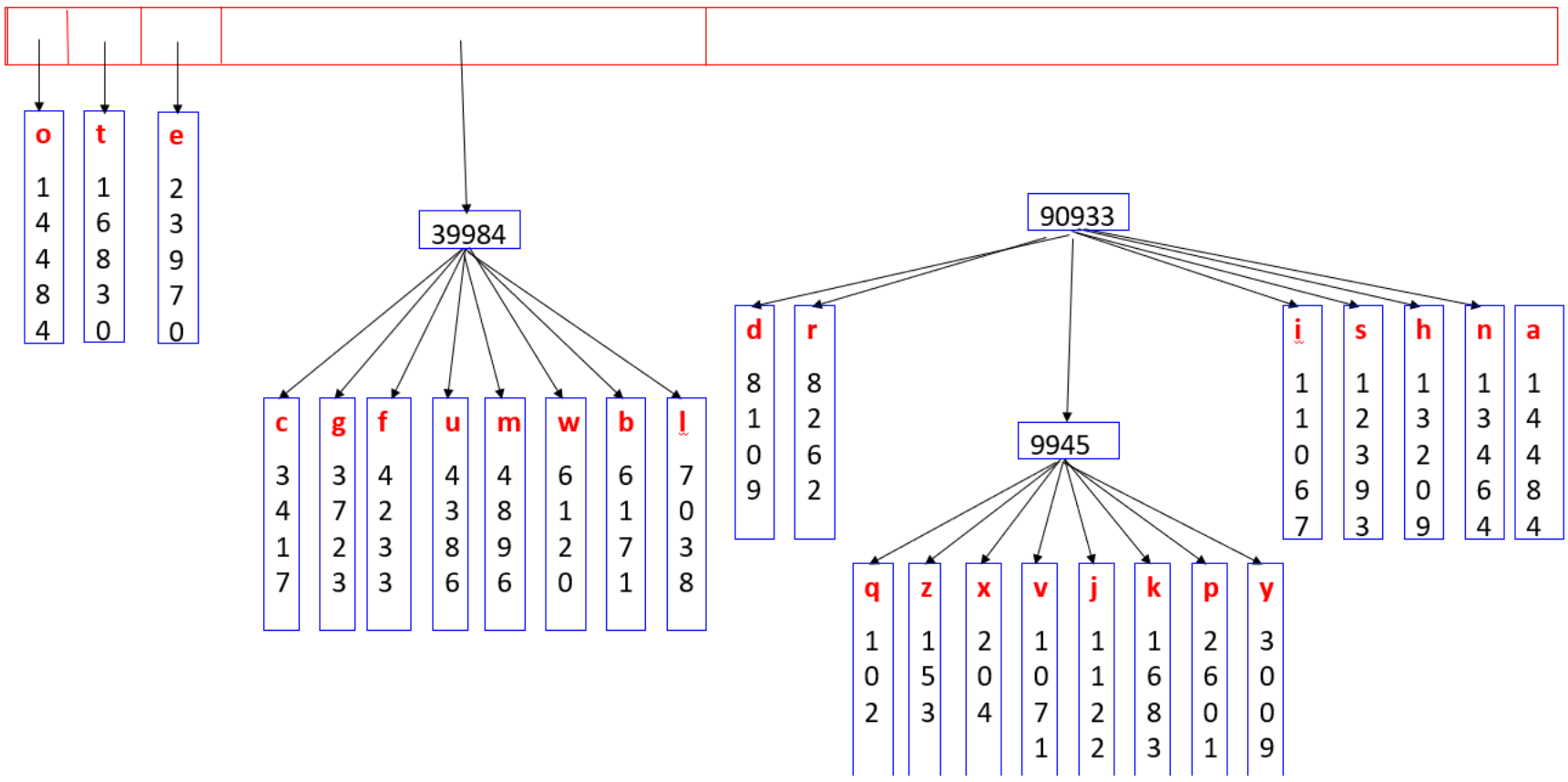
Step 5:



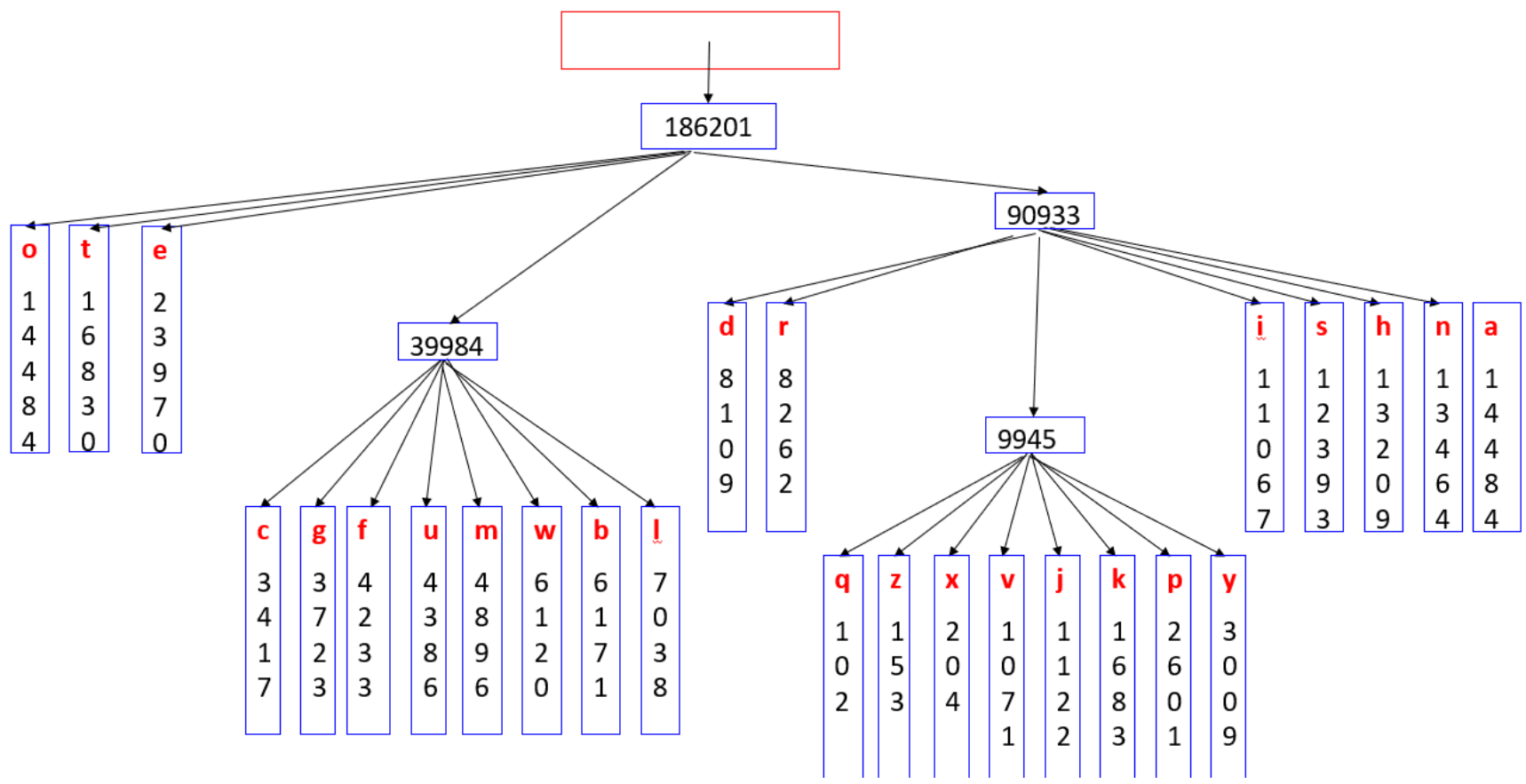
Step 6:



Step 7:

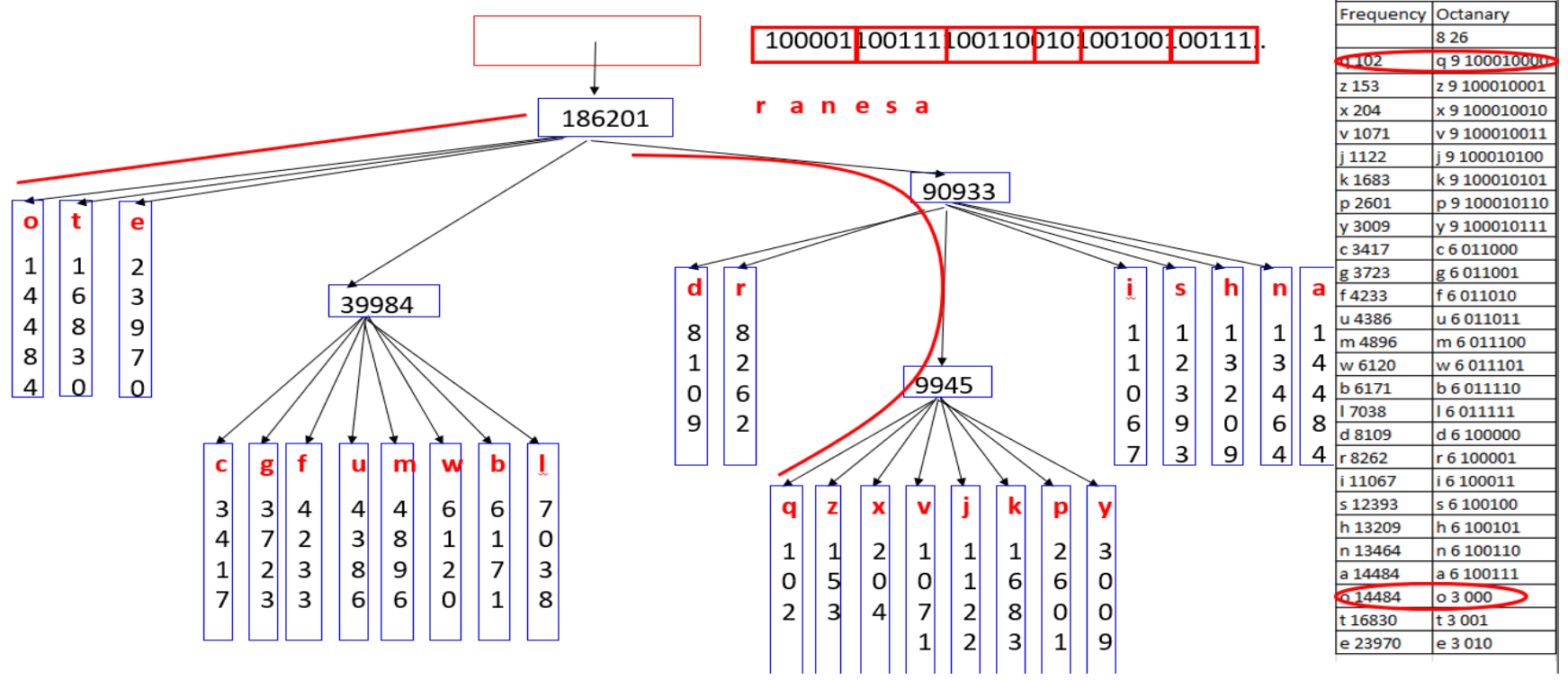


Step 8:



Step 9:

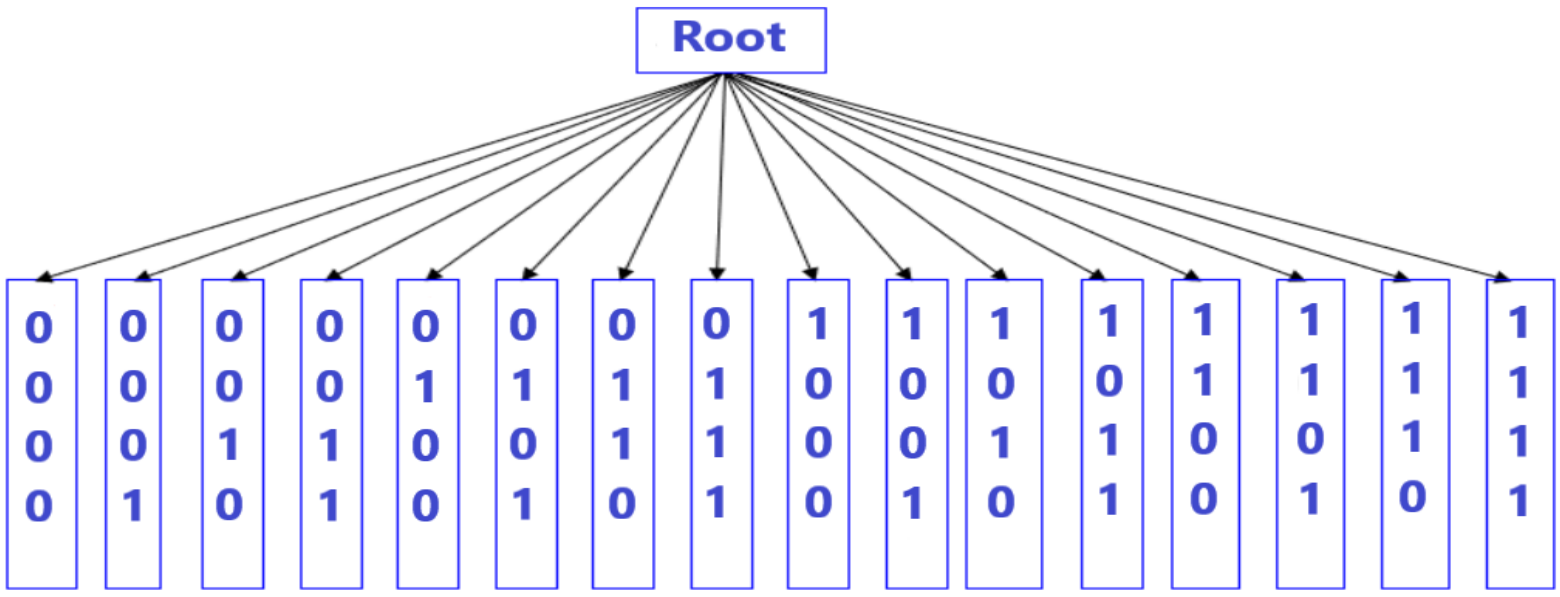
Octonary Code



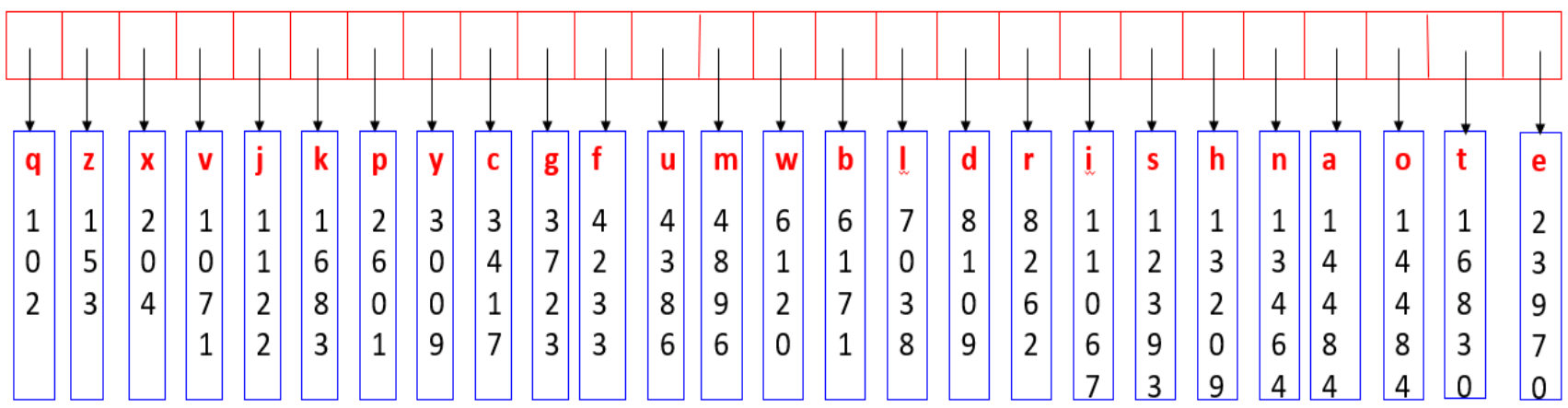
Luke 5	
Frequency	Octanary
	8 26
q 102	q 9 100010000
z 153	z 9 100010001
x 204	x 9 100010010
v 1071	v 9 100010011
j 1122	j 9 100010100
k 1683	k 9 100010101
p 2601	p 9 100010110
y 3009	y 9 100010111
c 3417	c 6 011000
g 3723	g 6 011001
f 4233	f 6 011010
u 4386	u 6 011011
m 4896	m 6 011100
w 6120	w 6 011101
b 6171	b 6 011110
l 7038	l 6 011111
d 8109	d 6 100000
r 8262	r 6 100001
i 11067	i 6 100011
s 12393	s 6 100100
h 13209	h 6 100101
n 13464	n 6 100110
a 14484	a 6 100111
o 14484	o 3 000
t 16830	t 3 001
e 23970	e 3 010

Horipriya Das Arpita - 2019831068

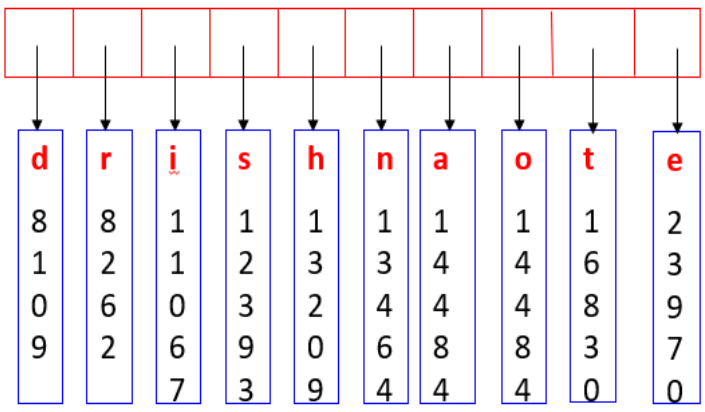
Building Hexanary Tree

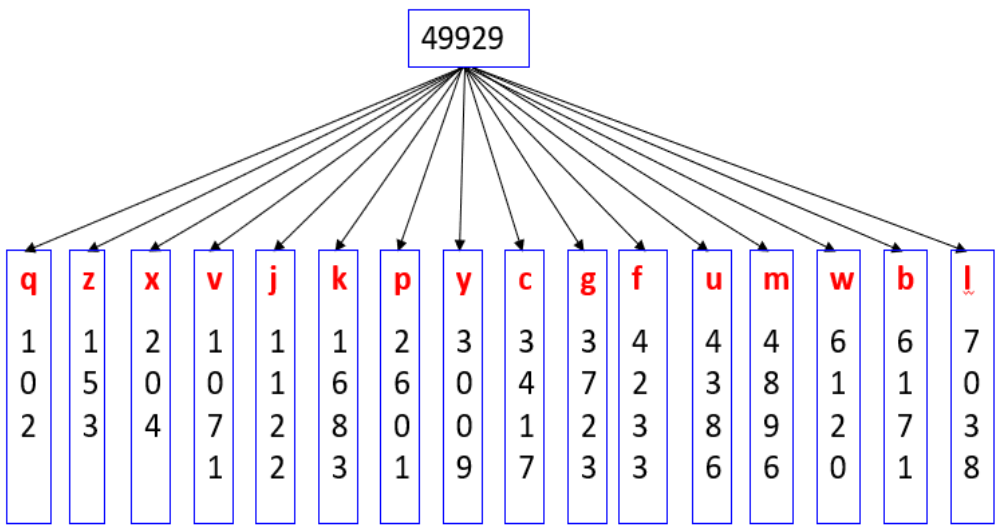


Step 1:

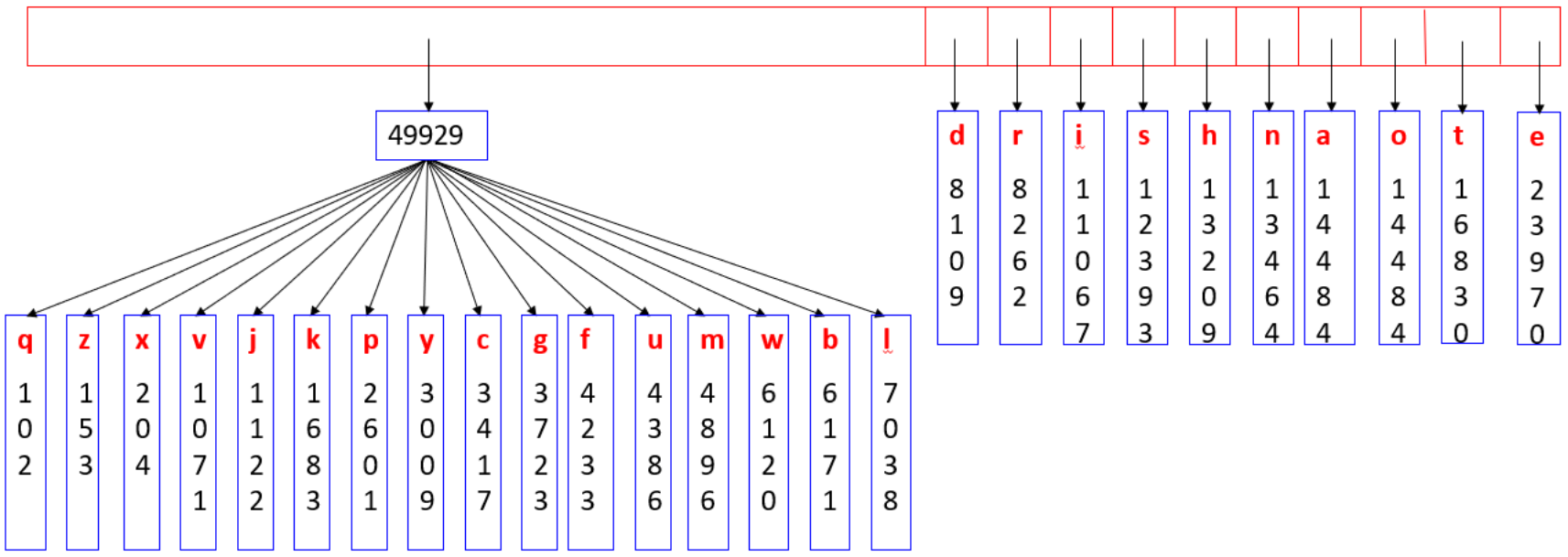


Step 2:

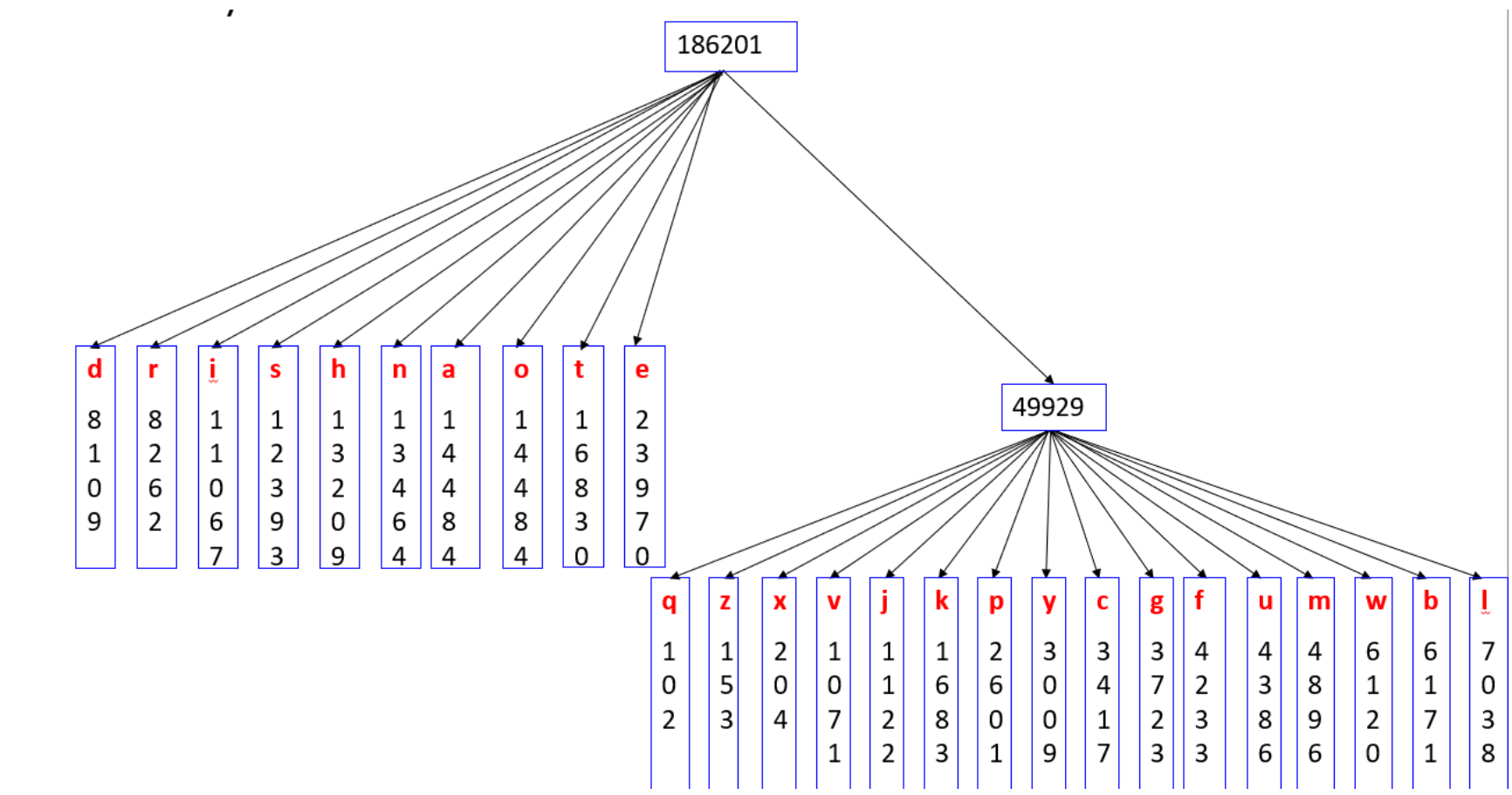




Step 3:

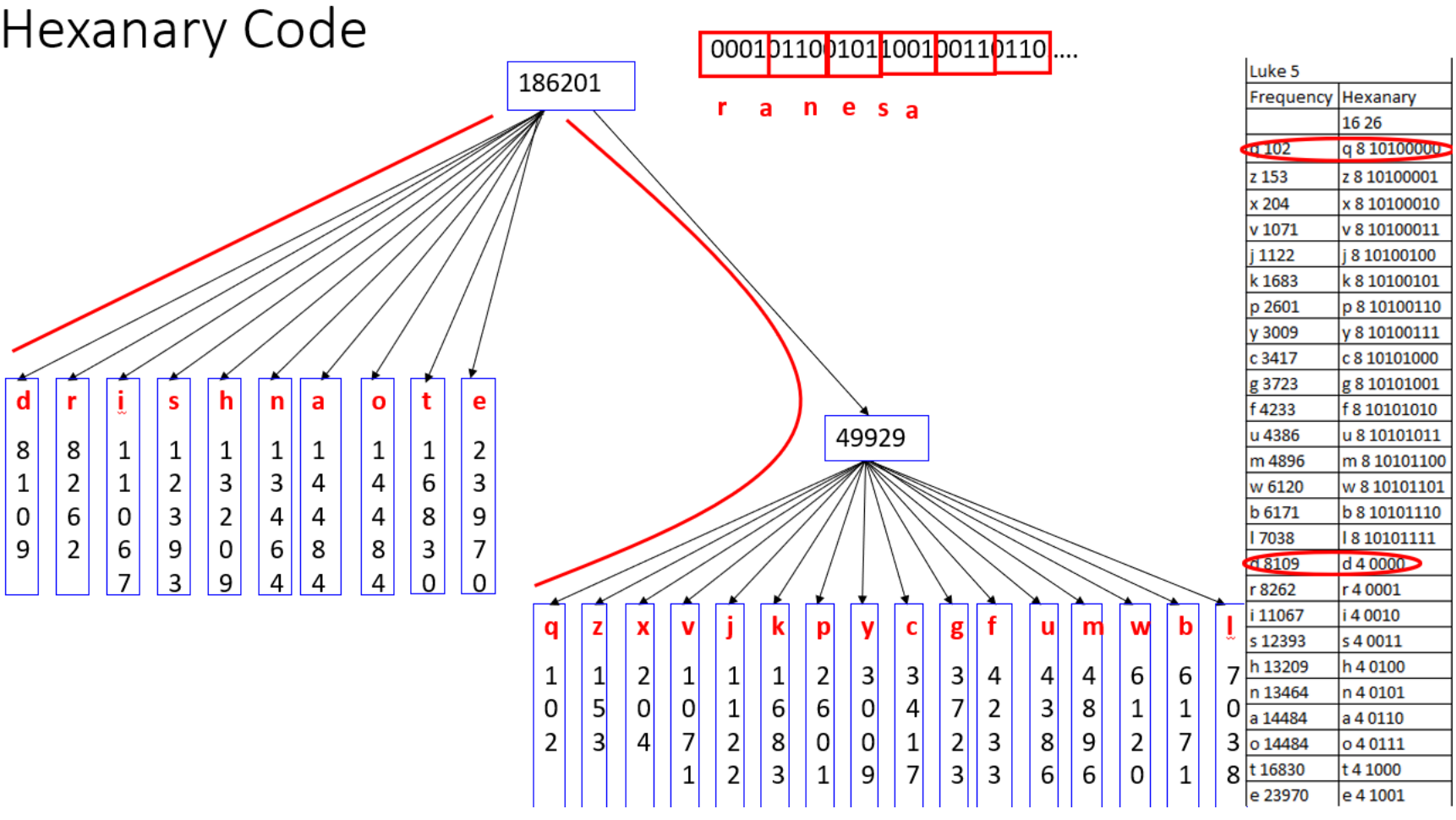


Step 4:



Step 5:

Hexanary Code



Luke 5	
Frequency	Hexanary
	16 26
q 102	q 8 10100000
z 153	z 8 10100001
x 204	x 8 10100010
v 1071	v 8 10100011
j 1122	j 8 10100100
k 1683	k 8 10100101
p 2601	p 8 10100110
y 3009	y 8 10100111
c 3417	c 8 10101000
g 3723	g 8 10101001
f 4233	f 8 10101010
u 4386	u 8 10101011
m 4896	m 8 10101100
w 6120	w 8 10101101
b 6171	b 8 10101110
l 7038	l 8 10101111
d 8109	d 4 0000
r 8262	r 4 0001
i 11067	i 4 0010
s 12393	s 4 0011
h 13209	h 4 0100
n 13464	n 4 0101
a 14484	a 4 0110
o 14484	o 4 0111
t 16830	t 4 1000
e 23970	e 4 1001