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

**Theorem**

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

**Proof** Given any 2 vertices \( x \) and \( y \) of a tree \( T \), there is a simple path between them in \( T \) because \( T \) is connected. The path must be unique, for if there were a second such path then there is a circuit in \( T \). Now assume there is a unique simple path between any 2 vertices \( x \) and \( y \) of \( T \), then \( T \) is connected. Furthermore, \( T \) can have no circuit, since otherwise there would be 2 simple paths between \( x \) and \( y \).
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.

If the directions of all edges are downward, the directions can be omitted.

For each edge $\langle u, v \rangle$, $u$ is the parent of $v$ and $v$ is the child of $u$. Vertices with the same parent are called siblings.

The ancestors of a vertex $v$ other than the root are the vertices in the path from the root to $v$, excluding $v$ and including the root.
The descendants of a vertex \( v \) are those vertices that have \( v \) as an ancestor.

A vertex of a tree is called the leaf if it has no children. Vertices that have children are called internal vertices.

The root is an internal vertex unless it is the only vertex in the tree, in which case it is a leaf. If \( v \) is a vertex in a tree, the subtree with \( v \) as its root is the subgraph of the tree containing \( v \) and its descendants and all edges incident to these descendants.

![a rooted tree]

Descendants of \( v \):
- \( a, b, c, d, e, f \)

Subtree rooted at \( v \)
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.

\[ T_1, \quad T_2, \quad T_3, \quad T_4 \]

\( T_1 \) is a full binary tree.
\( T_2 \) is a full \( 3 \)-ary tree.
\( T_3 \) is a full \( 4 \)-ary tree.
\( T_3 \) is not a full \( m \)-ary tree for \( m \geq 3 \).

An ordered rooted tree is a rooted tree where the children of each vertex are ordered from left to right.
In an ordered binary tree (or simply, binary tree), if an internal vertex has 2 children, the first child is called the **left child** and the second child is called the **right child**. The tree rooted at the left (right) child of a vertex is called the **left (right)** subtree of this vertex. For some applications, every vertex of a binary tree, other than the root, is designated as a left or right child of its parent.

![Binary Tree Diagram](image)

- **T**
- **Left Subtree of v**
- **Right Subtree of v**
Trees as Models

Representing organizations

FIGURE 10  An Organizational Tree for a Computer Company.

Computer file systems

The root is the root directory /. Internal vertices are directories. Leaves are files.

FIGURE 11  A Computer File System.

Tree-connected parallel processors

FIGURE 12  A Tree-Connected Network of Seven Processors.
Theorem 1. A tree with \( n \) vertices has \( n-1 \) edges.

Proof: By induction.

Basic step: A tree with 1 vertex has no edges. The theorem holds for \( n = 1 \).

Inductive step: Suppose that the theorem holds for \( n = k \) (hypothesis), and consider a tree \( T \) of \( n = k+1 \) vertices.

Let \( v \) be a leaf of \( T \) and \( w \) be the parent of \( v \). Removing \( v \) and the edge \((w,v)\) produces a tree \( T' \) with \( k \) vertices. By the induction hypothesis, \( T' \) has \( k-1 \) edges. It follows that \( T \) has \( k \) edges. This completes the inductive step.
Theorem 2. A full m-ary tree with \( i \) internal vertices contains \( n = mi + 1 \) vertices.

Proof: Because each of the \( i \) internal vertices has \( m \) children, there are \( mi \) vertices in the tree other than the root. Thus, the number of vertices in the tree is \( n = mi + 1 \).

Theorem 3. A full m-ary tree with

(i) \( n \) vertices has \( i = \frac{n-1}{m} \) internal vertices and \( l = \left(\frac{m-1}{m}\right)n + 1 \) leaves

(ii) \( i \) internal vertices has \( n = mi + 1 \) vertices and \( l = (m-1)i + 1 \) leaves

(iii) \( l \) leaves has \( n = \frac{ml-1}{m-1} \) vertices and \( i = \frac{l-1}{m-1} \) internal vertices.
Let \( n \) be the number of vertices, \( i \) the number of internal vertices, and \( l \) the number of leaves. From Theorem 2, we know that \( n = m + 1 \). We also know that \( n = l + i \).

(i) \[ n = m + 1 \quad \Rightarrow \quad i = \frac{n-1}{m} \]
\[
\begin{align*}
 n &= l + i \\
 i &= \frac{n-1}{m} \quad \Rightarrow \quad l &= n - i = n - \frac{n-1}{m} = \frac{(m-1)+1}{m}
\end{align*}
\]

(ii) \[ l = n - i \quad \Rightarrow \quad l = mi + 1 - i \\
 n = mi + 1 \quad \Rightarrow \quad = (m-1)i + 1
\]

(iii) \[ l = (m-1)i + 1 \quad \Rightarrow \quad i = \frac{l-1}{m-1} \]
\[
\begin{align*}
 n &= mi + 1 \\
 i &= \frac{l-1}{m-1} \quad \Rightarrow \quad n &= mi + 1 \\
 &= \frac{m(l-1)}{m-1} + 1 \\
 &= \frac{ml - m + m - 1}{m-1} \\
 &= \frac{ml - 1}{m-1}
\end{align*}
\]
Example: Someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other people. Some people do this, but others do not send any letter.

How many people have seen the letter, including the first person, if no one receives more than once, and if the chain letter ends after there have been 100 people who read it but did not send it out?

How many people sent out the letter?

Answer: The chain letter can be represented by a full 4-ary tree, with internal vertices representing the people who sent out the letter and leaves the people who did not send it out. Then, \( l = 100 \). By (iii) of Theorem 3, the number of people who have seen the letter is

\[
    n = \frac{ml - l}{m - 1} = \frac{4 \cdot 100 - l}{4 - 1} = 133.
\]

Since \( n = l + i \), we have

\[
    i = n - l = 133 - 100 = 33.
\]

So, 33 people sent out the letter.
The **level** of a vertex in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is 0. The **height** of a rooted tree is the maximum of the levels of the vertices.

A rooted m-ary tree of height $h$ is **balanced** if all leaves are at level $h$ or $h-1$.
**Theorem 4**  There are at most \( m^k \) leaves in an \( m \)-ary tree of height \( h \).

**Proof:**  Induction on height \( h \).

For \( h = 1 \), there are at most \( m' = m \) leaves.

Assume that the theorem holds for \( h = k \), and consider \( h = k + 1 \). The root has at most \( m \) subtrees of height \( k \), and by the hypothesis, each subtree can have at most \( m^k \) leaves. Therefore, an \( m \)-ary tree of height \( k + 1 \) has at most \( m \cdot m^k = m^{k+1} \) leaves. This completes the induction.
Corollary. If an m-ary tree of height h has l leaves, then \( h \geq \lceil \log_m l \rceil \).
If the m-ary tree is full and balanced, then \( h = \lceil \log_m l \rceil \).

Proof: By Theorem 4, \( l \leq m^h \). Therefore, \( \log_m l \leq \log_m (m^h) = h \). Because h is an integer, \( h \geq \lceil \log_m l \rceil \).

Suppose the tree is full and balanced. Then each leaf is at level h or h-1. This tree has more leaves than a tree \( T' \) of height h-1 by deleting all the vertices at level h. \( T' \) has \( m^{h-1} \) leaves. Because of \( T' \),

\( l \leq m^h \), we have \( m^{h-1} < l \leq m^h \).
That is, \( \log_m m^{h-1} < \log_m l \leq \log_m m^h \)
\( h-1 < \log_m l \leq h \)

Hence, \( h = \lceil \log_m l \rceil \).
Binary Search Trees (BSTs)

A BST is a binary tree in which

- each child is designated as a right or left child
- each vertex is labeled with a key.
  
  $\text{label}(v)$ is the key of vertex $v$

- $\text{label}(v) > \text{all keys in the left subtree of } v$

- $\text{label}(v) < \text{all keys in the right subtree of } v$

**Algorithm 1** Locating and Adding Items to a Binary Search Tree.

**procedure** insertion($T$ : binary search tree, $x$: item)

$v := \text{root of } T$

{a vertex not present in $T$ has the value null}

**while** $v \neq \text{null}$ and $\text{label}(v) \neq x$

**begin**

if $x < \text{label}(v)$ then

  if left child of $v \neq \text{null}$ then $v := \text{left child of } v$

  else add new vertex as a left child of $v$ and set $v := \text{null}$

else

  if right child of $v \neq \text{null}$ then $v := \text{right child of } v$

  else add new vertex as a right child of $v$ to $T$ and set $v := \text{null}$

**end**

if root of $T = \text{null}$ then add a vertex $v$ to the tree and label it with $x$

else if $v$ is null or $\text{label}(v) \neq x$ then label new vertex with $x$ and let $v$ be this new vertex

{v = location of x}
Given a BST $T$, we obtain full BST with leaves without keys.

![Diagram of BSTs $T$ and $U$](image)

**Figure 2** Adding Unlabeled Vertices to Make a Binary Search Tree Full.

Number of comparisons need to add a new key is the length of the longest path in $U$ from the root to a leaf.

The height of $U$ is $\lceil \log (n+1) \rceil$, by Corollary 1.

If $U$ is balanced, its height is $\lceil \log (n+1) \rceil$.

BSTs that are balanced give optimal worst-case complexity.
Decision Trees

Used to model problems in which a series of decisions leads to a solution.

Each internal vertex corresponds to a decision, with each subtree for a possible outcome of the decision.

Example

7 coins, all with the same weight
1 counterfeit coin, weighs less that

How many weighings are necessary using a balance scale to find the counterfeit?

Solution: \[ \lceil \log_3 8 \rceil = 2. \]

Each weighing has 3 possible outcomes.
The decision tree is a 3-ary tree. Then, the number of weighings is the height of this tree.

FIGURE 3 A Decision Tree for Locating a Counterfeit Coin. The counterfeit coin is shown in color below each final weighing.
Example: What is the minimum number of comparisons that are needed to sort n numbers?

Answer: Given n elements, there are n! possible orderings of these elements. The sorting problem based on binary comparisons can be represented by a binary decision tree in which each internal vertex represents a comparison of 2 elements. Each leaf represents one of the n! permutations of n elements.

The height of a binary tree with n! leaves is at least \( \lceil \log n! \rceil \) (by Corollary 1). Thus, at least \( \lceil \log n! \rceil \) comparisons are needed. Since \( \lceil \log n! \rceil = \Theta(n \log n) \), sorting n elements requires \( \Omega(n \log n) \) comparisons. Merge sort algorithm uses \( O(n \log n) \) comparisons, which is optimal.
Prefix Codes


Code tree: a full binary tree with the left edge labeled by 0 and the right edge labeled by 1 for each internal vertex. Each leaf corresponds to a letter. The bit string used to encode a letter is the sequence of labels of edges in the unique path from the root to the leaf that corresponds the letter.

Prefix codes: No bit string for a letter occurs as a prefix of the bit string for another letter.
Example: e = 0, a = 10, t = 110
n = 1110, s = 1111

This is a prefix coding.

Consider a string 1111101100
s a n e

Huffman Coding

A coding for data compression.

prefix

Each letter ai has a frequency wi.

---

**Algorithm 2** Huffman Coding.

**procedure** Huffman(C: symbols ai with frequencies wi, i = 1, ..., n)

F := forest of n rooted trees, each consisting of the single vertex ai and assigned weight wi

while F is not a tree

begin

Replace the rooted trees T and T' of least weights from F with w(T) ≥ w(T') with a tree having a new root that has T as its left subtree and T' as its right subtree. Label the new edge to T with 0 and the new edge to T' with 1.

Assign w(T) + w(T') as the weight of the new tree.

end

{the Huffman coding for the symbol ai is the concatenation of the labels of the edges in the unique path from the root to the vertex ai}
Example: Use Huffman coding to encode $A, B, C, D, E, \text{and } F$ with $w(A) = 0.08$, $w(B) = 0.10$, $w(C) = 0.12$, $w(D) = 0.15$, $w(E) = 0.20$, $w(F) = 0.35$.

![Huffman Coding Diagram]

FIGURE 6 Huffman Coding of Symbols in Example 4.

$A = 111, \ B = 110, \ C = 011, \ D = 010, \ E = 10, \ F = 00$

Average number of bits used to encode a letter is:

$3 \times 0.18 + 3 \times 0.10 + 3 \times 0.12 + 2 \times 0.15 + 2 \times 0.20 + 2 \times 0.35 = 2.45$
Game Trees

Used to characterizing certain, say 2-player, games.

Vertices: positions that a game can be in as it progresses.
All symmetric positions of a game are represented by the same vertex
Root: starting position
Edges: legal moves between positions.
Even levels (root at level 0): first player's move
Odd levels: second player's move.

Example: Nim. A legal move: remove 1 or more stones from 1 pile without removing all stones left.

FIGURE 7 The Game Tree for a Game of Nim.
The value of a vertex in a game tree is recursively defined as

(i) the value of a leaf is the payoff to
the first player when the game terminates
in the position represented by this leaf

(ii) the value of an internal vertex at an
even level is the maximum of the values
of its children, and the value of an
internal vertex at an odd level is the
minimum of the values of its children.
A strategy is a set of rules that tells a player how to select a move to win the game.

An optimal strategy for the first player is a strategy that maximizes the payoff of this player and for the second player is a strategy that minimizes this payoff.

The strategy where the first (second) player moves to a position represented by a child with maximum (minimum) value is called the minmax strategy.

If both players follow the minmax strategy, we can determine who will win the game by just looking at the value of the root.

The minmax strategy is the optimal strategy for both players.
Example: A Nim game that guarantees the first player always wins.

A Nim game that guarantees the second player always wins.

FIGURE 9  Showing the Values of Vertices in the Game of Nim.
Tree traversal

Ordered rooted trees are often used to store information. There are different ways to visit all vertices of an ordered rooted tree.

The universal address system is used to totally order the vertices of an ordered rooted tree:

1. Root is labeled 0. Its k children are labeled from left to right with 1, 2, 3, ..., k.

2. For each vertex v at level n with label A, label its k children from left to right with A.1, A.2, ..., A.k.

Then, vertices are totally ordered using the lexicographic ordering of their labels.
We display the labelings of the universal address system next to the vertices in the ordered rooted tree shown in Figure 1. The lexicographic ordering of the labelings is

\[0 < 1 < 1.1 < 1.2 < 1.3 < 2 < 3 < 3.1 < 3.1.1 < 3.1.2 < 3.1.2.1 < 3.1.2.2 < 3.1.2.3 < 3.1.2.4 < 3.1.3 < 3.2 < 4 < 4.1 < 5 < 5.1 < 5.1.1 < 5.2 < 5.3\]

\[\]

FIGURE 1 The Universal Address System of an Ordered Rooted Tree.

**Preorder traversal**

**FIGURE 2** Preorder Traversal.
Example

Preorder traversal: Visit root, visit subtrees left to right

FIGURE 4 The Preorder Traversal of $T$. 
Inorder traversal

Step 2: Visit r

\[ r \]

\[ T_1 \]

Step 1:
Visit \( T_1 \) in inorder

\[ T_2 \]

Step 3:
Visit \( T_2 \) in inorder

\[ \cdots \]

Step \( n+1 \):
Visit \( T_n \) in inorder

\[ T_n \]

FIGURE 5  Inorder Traversal.

Example

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

FIGURE 6  The Inorder Traversal of \( T \).
**Postorder traversal**

Step 1: Visit $T_1$ in postorder

Step 2: Visit $T_2$ in postorder

Step $n$: Visit $T_n$ in postorder

**FIGURE 7** Postorder Traversal.

**Example**

Postorder traversal: Visit subtrees left to right; visit root

**FIGURE 8** The Postorder Traversal of $T$. 
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
begin
$T(c) :=$ subtree with $c$ as its root
preorder($T(c)$)
end

ALGORITHM 2 Inorder Traversal.

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

ALGORITHM 3 Postorder Traversal.

**procedure** postorder($T$: ordered rooted tree)
$r :=$ root of $T$
for each child $c$ of $r$ from left to right
begin
$T(c) :=$ subtree with $c$ as its root
postorder($T(c)$)
end
list $r$
Used to represent complicated expressions such as compound propositions, combinations of sets, and arithmetic expressions.

**Arithmetic expressions**

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

An ordered rooted tree can be used to represent an arithmetic expression:

**Example**: \((x + y) \uparrow 2) + ((x - 4)/3)\)

**FIGURE 10** A Binary Tree Representing \(((x + y) \uparrow 2) + ((x - 4)/3)\).
**Infix form:** fully parenthesized expression obtained by inorder traversal of the rooted tree of the expression.

**Example:** \((x+y)^2 + (x-4)/3\)

**Prefix form (Polish notation):** obtained by preorder traversal of the rooted tree of the expression.

**Example:** (for \((x+y)^2 + (x-4)/3\))

\[ + \uparrow \uparrow \uparrow \ x \ y \ 2/ - \ x \ 4 \ 3 \]

**Postfix form (reverse Polish notation):** obtained by postorder traversal of the ordered tree.

**Example:** (for \((x+y)^2 + (x-4)/3\))

\[ x \ y \ + \ 2 \ \uparrow \ \times \ 4 \ - \ 3/ + \]
Prefix expression evaluation

\[
\begin{array}{clclcl}
+ & - & * & 2 & 3 & 5 & / & 1 & 2 & 3 & 4 \\
213 &= 8 \\
+ & - & * & 2 & 3 & 5 & / & 8 & 4 \\
8/4 &= 2 \\
+ & - & * & 2 & 3 & 5 & 2 \\
2+3 &= 6 \\
+ & - & 6 & 5 & 2 \\
6-5 &= 1 \\
+ & 1 & 2 \\
1+2 &= 3 \\
\end{array}
\]

Value of expression: 3

FIGURE 12 Evaluating a Prefix Expression.

Postfix expression evaluation

\[
\begin{array}{clclclclclcl}
7 & 2 & 3 & * & - & 4 & \uparrow & 9 & 3 & / & + \\
2+3 &= 6 \\
7 & 6 & - & 4 & \uparrow & 9 & 3 & / & + \\
7-6 &= 1 \\
1 & 4 & \uparrow & 9 & 3 & / & + \\
1^4 &= 1 \\
1 & 9 & 3 & / & + \\
9/3 &= 3 \\
1 & 3 & + \\
1+3 &= 4 \\
\end{array}
\]

Value of expression: 4

FIGURE 13 Evaluating a Postfix Expression.
Ordered tree representation of compound propositions

Example: \((\neg (p \land q)) \iff (\neg p \lor \neg q)\)

\[
\begin{array}{c}
\text{prefix:} \\
\iff \neg \land p q \lor \neg p \land q \\
\text{postfix:} \\
p q \land \neg p \land q \lor \neg \\
\text{infix:} \\
(\neg (p \land q)) \iff (\neg p \lor \neg q)
\end{array}
\]

FIGURE 14 Constructing the Rooted Tree for a Compound Proposition.
Let $G$ be a simple graph.

A **spanning tree** of $G$ is a subgraph of $G$ that is a tree containing every vertex of $G$.

A simple graph is connected if and only if it has a spanning tree.

A spanning tree of a connected simple graph can be built using **depth-first search**.

**Algorithm 1** Depth-First Search.

```plaintext
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 $T$  
begin  
   add vertex $w$ and edge $\{v, w\}$ to $T$  
   visit($w$)  
end
```

The edges selected by depth-first search are called **tree edges**. All other edges are called **back edges**.
A spanning tree of a graph can also be built by breadth-first search.

**Algorithm 2: Breadth-First Search.**

```plaintext
procedure BFS (G: connected graph with vertices v₁, v₂, ..., vₙ)
T := tree consisting only of vertex v₁
L := empty list
put v₁ in the list L of unprocessed vertices
while L is not empty
  begin
    remove the first vertex, v, from L
    for each neighbor w of v
      if w is not in L and not in T then
        begin
          add w to the end of the list L
          add w and edge {v, w} to T
        end
  end
end
```
**Example**

![Graph G](image)

**FIGURE 9** A Graph $G$.

![Tree Diagrams](image)

**FIGURE 10** Breadth-First Search of $G$.

---

**Backtracking**

A technique of searching a solution in a decision tree.

**Example:** The *n*-Queen problem

How $n$ queens can be placed on an $n \times n$ chessboard so that no 2 queens can attack one another?
Example: 4-queen problem solution

Example: Sums of Subsets

Given a set of positive integers \( x_1, x_2, ..., x_n \), find a subset of this set that has \( M \) as its sum.

Let \( S = \{31, 27, 15, 11, 7, 5\} \)

\[ M = 39 \]

\[
\emptyset \\
\text{Sum} = 0
\]

\[
\{31\} \\
\text{Sum} = 31
\]

\[
\{27\} \\
\text{Sum} = 27
\]

\[
\{31, 7\} \\
\text{Sum} = 38
\]

\[
\{31, 5\} \\
\text{Sum} = 36
\]

\[
\{27, 11\} \\
\text{Sum} = 38
\]

\[
\{27, 7\} \\
\text{Sum} = 34
\]

\[
\{27, 7, 5\} \\
\text{Sum} = 39
\]

FIGURE 12  A Backtracking Solution of the Four-Queens Problem.

FIGURE 13  Find a Sum Equal to 39 Using Backtracking.
A minimum spanning tree (MST) in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

**Prim's Algorithm**

Merge forests to form a tree in a greedy way.

**Algorithm 1** Prim's Algorithm.

procedure Prim(G: weighted connected undirected graph with \( n \) vertices)

\( T := \) a minimum-weight edge

for \( i := 1 \) to \( n - 2 \)

begin

\( e := \) 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

end \( \{ T \) is a minimum spanning tree of \( G \}\)

**Example**

![Graph](image)

**Figure 4** A Minimum Spanning Tree Produced Using Prim's Algorithm.
Algorithm 2: Kruskal’s Algorithm.

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

Example

![Diagram](attachment:image.png)

(a)

<table>
<thead>
<tr>
<th>Choice</th>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{c, d}</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>{k, l}</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>{b, f}</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>{c, g}</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>{a, b}</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>{f, j}</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>{b, c}</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>{j, k}</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>{g, h}</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>{i, j}</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>{a, e}</td>
<td>3</td>
</tr>
</tbody>
</table>

Total: 24

FIGURE 5  A Minimum Spanning Tree Produced by Kruskal’s Algorithm.