10.5 Trees

- Definitions: circuit-free, tree, trivial tree, forest
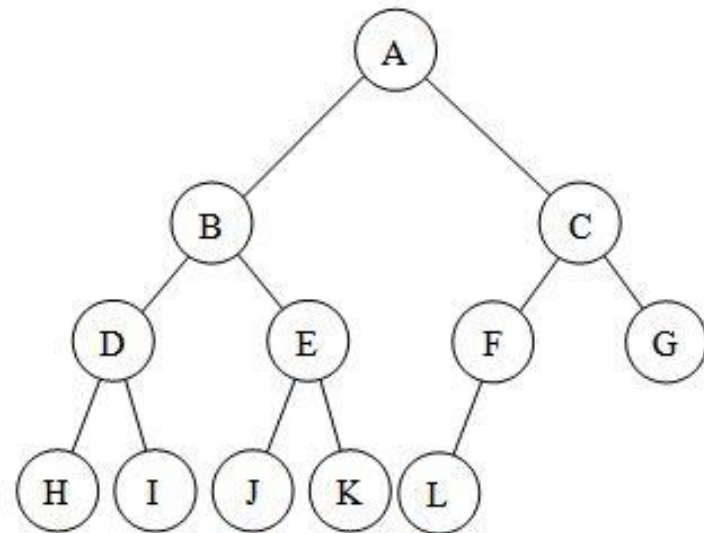- Characterizing trees: terminal vertex (leaf), internal vertex

10.6 Rooted Trees

- Definitions: rooted tree, root, level, height, child, parent, sibling, ancestor, descendant
- Definitions: binary tree, full binary tree, subtree
- Binary tree traversal: breadth-first-search (BFD), depth-first-search (DFS)

10.7 Spanning Trees and Shortest Paths

- Definitions: spanning tree, weighted graph, minimum spanning tree (MST)
- Kruskal's algorithm, Prim's algorithm
- Dijkstra's shortest path algorithm (non covered this semester)

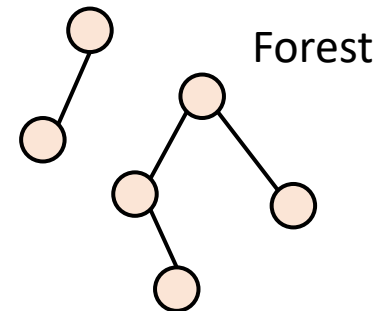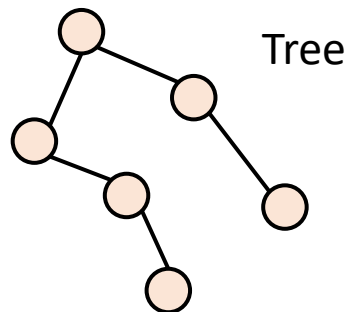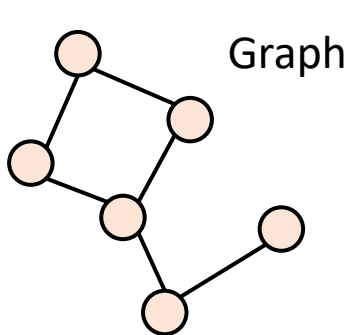# 10.5 Trees

Definition

Definition

Definition: Tree

(The graph is assumed to be undirected here.)

A **graph** is said to be **circuit-free** if and only if it has no circuits.

A simple graph is called a **tree** if and only if it is circuit-free and connected.

A **trivial tree** is a tree that consists of a single vertex.

A simple graph is called a **forest** if and only if it is circuit-free and not connected.

Graph

Tree

Forest

Example

Possibility Tree

As discussed in week 9, a possibility tree is used to keep systematic track of all possibilities in which events happen in order. For example:
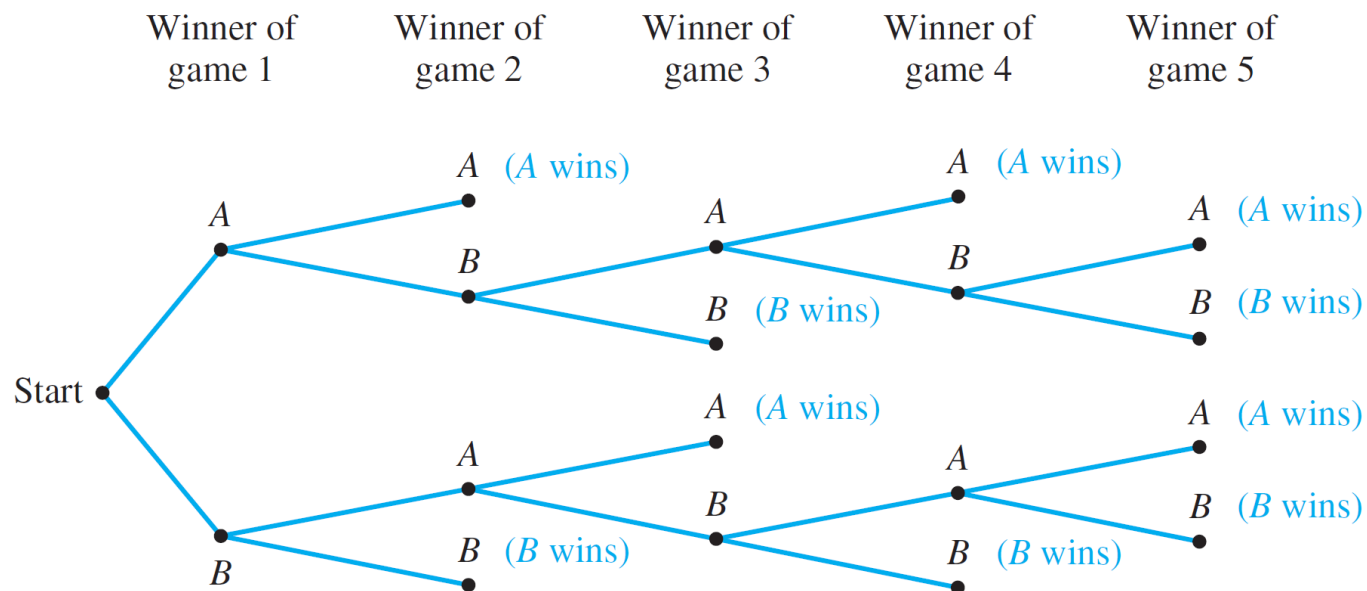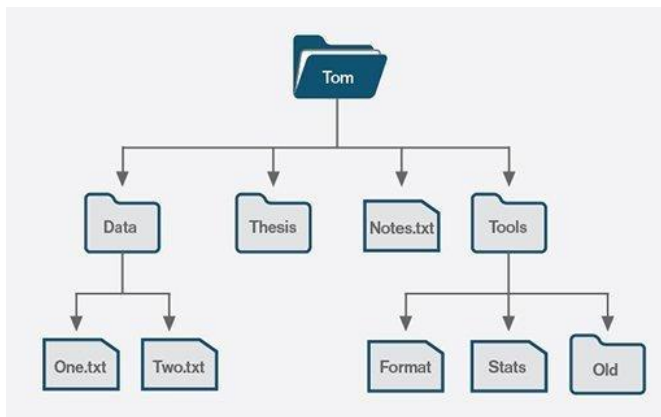

Figure 9.2.1 The Outcomes of a Tournament

Applications

Applications

Trees are used to store hierarchically ordered data.

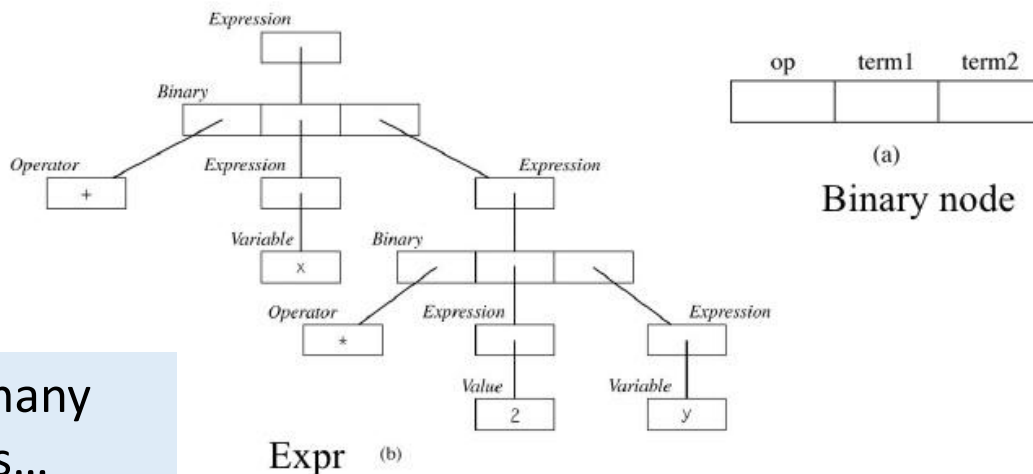Computer file systems

Binary Search Trees (BSTs)

And many others…

Abstract Syntax Tree

- Just as we can build a parse tree from a BNF grammar, we can build an abstract syntax tree from an abstract syntax

- Example for: x+2*y

```
Expression = Variable | Value | Binary
Binary = Operator op ; Expression term1, term2
```


(a)
Binary node

Expr   (b)

6

Lemma 10.5.1

Any non-trivial tree has at least one vertex of degree 1.

Proof: Let *T* be a particular but arbitrarily chosen non-trivial tree.

Step 1: Pick a vertex *v* of *T* and let *e* be an edge incident on *v*.

Step 2: While deg(*v*) > 1, repeat steps 2a, 2b and 2c:

> 2a: Choose *e'* to be an edge incident on *v* such that $e' \neq e$.

> 2b: Let *v'* be the vertex at the other end of *e'* from *v*, and *v'* has not be chosen before.

> 2c: Let *e = e' and v = v'*.
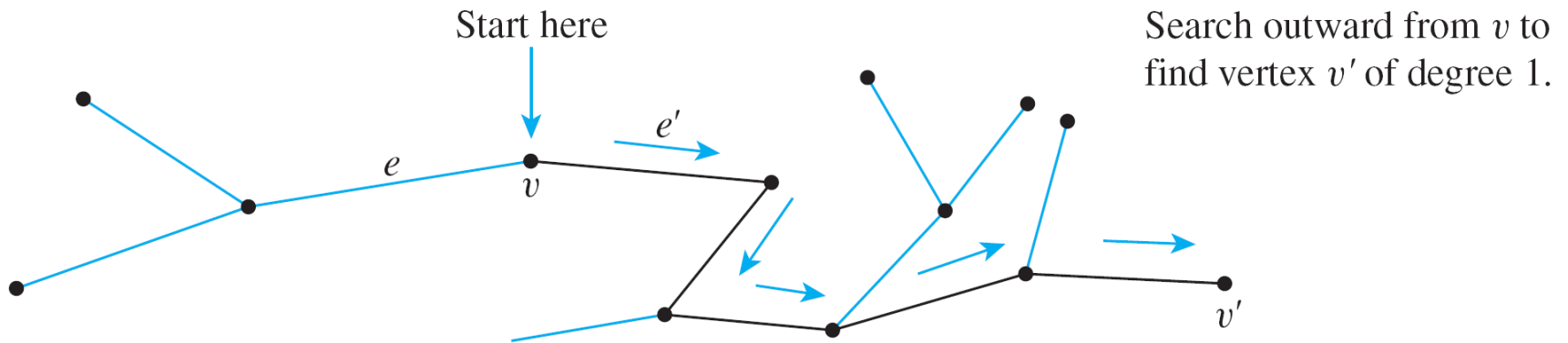
The algorithm must eventually terminate because the set of vertices of the tree *T* is finite and *T* is circuit-free. When it does, a vertex *v* of degree 1 will have been found.

Characterizing Trees

Lemma 10.5.1

Any non-trivial tree has at least one vertex of degree 1.

Note: We can use another theorem to prove that a non-trial tree actually has at least two vertices of degree 1.

Characterizing Trees

Definitions: Terminal vertex (leaf) and internal vertex

Let T be a tree. If T has only one or two vertices, then each is called a **terminal vertex** (or **leaf**). If T has at least three vertices, then a vertex of degree 1 in T is called a **terminal vertex** (or **leaf**), and a vertex of degree greater than 1 in T is called an **internal vertex**.

Example: Find all terminal vertices (leaves) and all internal vertices in the following tree:

Leaves: v_0, v_2, v_4, v_5, v_7 and v_8.

Internal vertices: v_6, v_1 and v_3.

Characterizing Trees

Theorem 10.5.2

Any tree with n vertices ($n > 0$) has $n - 1$ edges.

Proof: By mathematical induction.

Let the property $P(n)$ be "any tree with n vertices has $n - 1$ edges".

$P(1)$: Let T be a tree with one vertex. Then T has no edges.
So $P(1)$ is true.

Show that for all integers $k \geq 1$, if $P(k)$ is true then $P(k+1)$ is true.
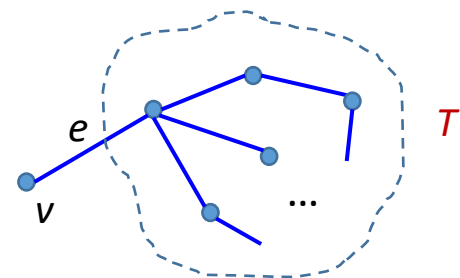
Suppose $P(k)$ is true.

1. Let T be a particular but arbitrarily chosen tree with $k + 1$ vertices.

2. Since k is positive, $(k + 1) \geq 2$, and so T has more than one vertex.

3. Hence, by Lemma 10.5.1, T has a vertex v of degree 1, and has at least another vertex in T besides v.

Characterizing Trees

Proof: (continued…)

3. Hence, by Lemma 10.5.1, T has a vertex v of degree 1, and has at least another vertex in T besides v.

4. Thus, there is an edge e connecting v to the rest of T.

5. Define a subgraph T' of T so that $V_{T'} = V_T - \{v\}$ and $E_{T'} = E_T - \{e\}$.

    5.1 The number of vertices of T' is $(k + 1) - 1 = k$.

    5.2 T' is circuit-free.

    5.3 T' is connected.

6. Hence by definition, T' is a tree.

7. Since T' has k vertices, by inductive hypothesis,
   number of edges of T' = (number of vertices of T') $- 1 = k - 1$.

8. Hence, number of edges of T = (number of edges of T') $+ 1 = k$.

9. Hence $P(k+1)$ is true.

Characterizing Trees

Exercise: Using Theorem 10.5.2, prove that a non-trivial tree has at least 2 vertices of degree 1.

Do this exercise on your own and discuss it on Canvas or QnA.

We will assume that this is proved and use it in our next problem.

Characterizing Trees

Example: Find all non-isomorphic trees with 4 vertices.

By Theorem 10.5.2, any tree with 4 vertices has 3 edges. And so by the Handshake Theorem, the tree has a total degree of 6.

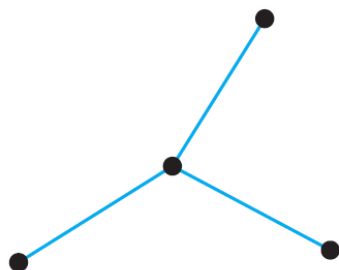| Theorem 10.1.1 The Handshake Theorem |
| --- |
| Given a graph $G=(V, E)$, the total degree of $G = 2|E|$. |

Also, every non-trivial tree has at least two vertices of degree 1.

The only possible combinations of degrees for the 4 vertices are:

1, 1, 1, 3 and 1, 1, 2, 2

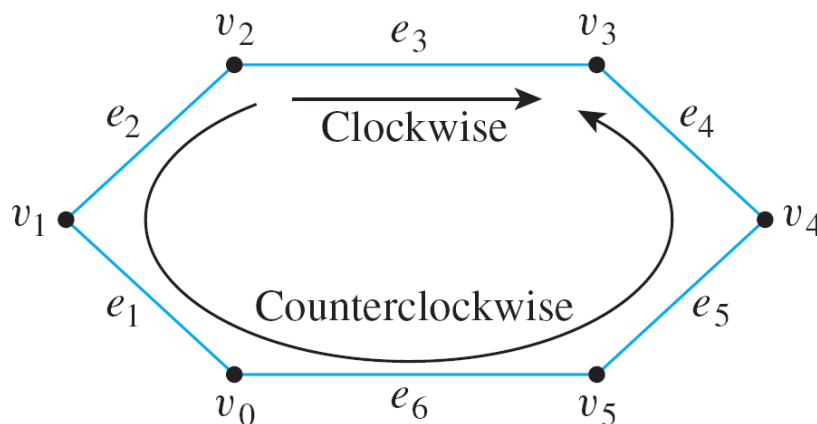Therefore, there are two non-isomorphic trees with 4 vertices.

Characterizing Trees

Lemma 10.5.3

If G is any connected graph, C is any circuit in G, and one of the edges of C is removed from G, then the graph that remains is still connected.

Essentially, the reason why Lemma 10.5.3 is true is that any two vertices in a circuit are connected by <u>2 distinct paths</u>. It is possible to draw the graph so that one of these goes "clockwise" and the other goes "counter-clockwise" around the circuit.

For example, in the circuit shown below:

The clockwise path from v_2 to v_3 is: $v_2\ e_3\ v_3$

The counter-clockwise path from v_2 to v_3 is:

$v_2\ e_2\ v_1\ e_1\ v_0\ e_6\ v_5\ e_5\ v_4\ e_4\ v_3$

Characterizing Trees

Theorem 10.5.4

If *G* is a connected graph with *n* vertices and *n* − 1 edges, then *G* is a tree.

Proof:

1. Suppose *G* is a particular but arbitrarily chosen graph that is connected and has *n* vertices and *n* − 1 edges.

2. Since *G* is connected, it suffices to show that *G* is circuit-free.

3. Suppose *G* is not circuit free

   3.1 Let *C* be the circuit in *G*.

   3.2 By Lemma 10.5.3, an edge of *C* can be removed from *G* to obtain a graph *G′* that is connected.

   3.3 If *G′* has a circuit, then repeat this process: Remove an edge of the circuit from *G′* to form a new connected graph.

   3.4 Continue the process of removing edges from the circuits until eventually a graph *G′′* is obtained that is connected and is circuit-free.

15

Characterizing Trees

Proof: (continued…)

3.4   Continue the process of removing edges from the circuits until eventually a graph G'' is obtained that is connected and is circuit-free.

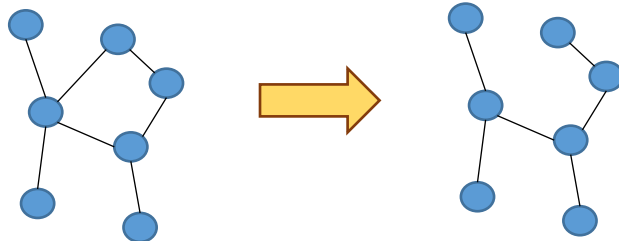3.5   By definition, G'' is a tree.

3.6   Since no vertices were removed from G to form G'', G'' has n vertices.

3.7   Thus, by Theorem 10.5.2, G'' has $n-1$ edges.

3.8   But the supposition that G has a circuit implies that at least one edge of G is removed to form G''.

3.9   Hence G'' has no more than $(n-1)-1 = n-2$ edges, which contradicts its having $n-1$ edges.

3.10   So the supposition is false.

4.   Hence G is circuit-free, and therefore G is a tree.

Assume G is not circuit-free.
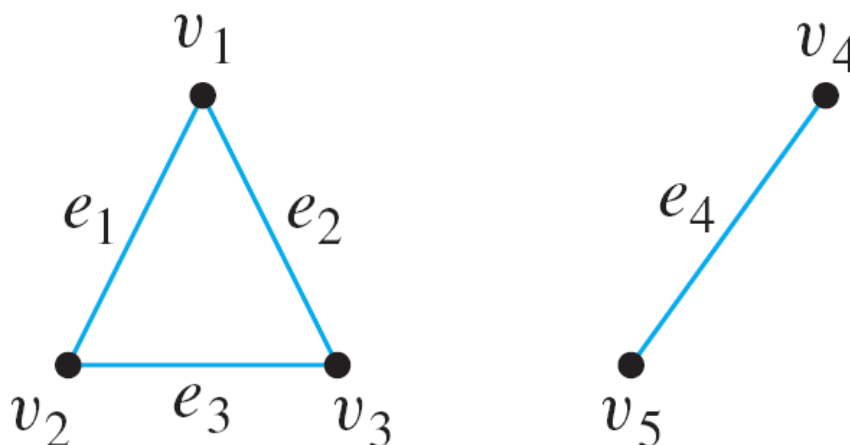G has n vertices and $n-1$ edges.

G'' is the result of removing edges from circuits in G. At least 1 edge removed from G.
G'' has n vertices and at most $n-2$ edges.

16

Characterizing Trees

Note that although it is true that every *connected* graph with *n* vertices and *n* − 1 edges is a tree, it is not true that *every* graph with *n* vertices and *n* − 1 edges is a tree.

Example: Give an example of a graph with five vertices and four edges that is not a tree.

By Theorem 10.5.4, such a graph cannot be connected. One example of such an unconnected graph is shown below.

10.6 Rooted Trees

Definitions

Definitions: Rooted Tree, Level, Height

A **rooted tree** is a tree in which there is one vertex that is distinguished from the others and is called the **root**.

The **level** of a vertex is the number of edges along the unique path between it and the root.

The **height** of a rooted tree is the maximum level of any vertex of the tree.
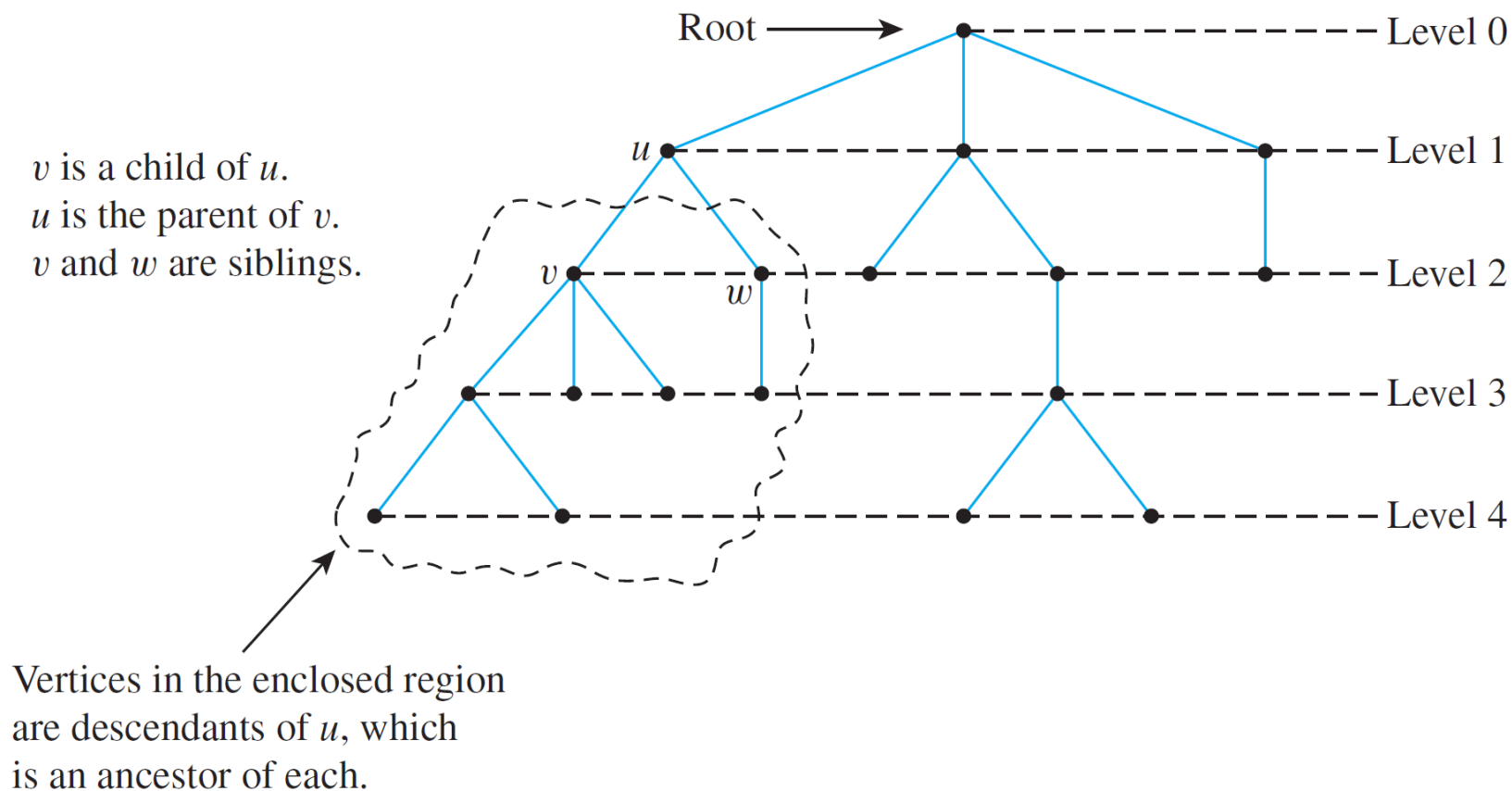
Definitions: Child, Parent, Sibling, Ancestor, Descendant

Given the root or any internal vertex v of a rooted tree, the **children** of v are all those vertices that are adjacent to v and are one level farther away from the root than v.

If w is a child of v, then v is called the **parent** of w, and two distinct vertices that are both children of the same parent are called **siblings**.

Given two distinct vertices v and w, if v lies on the unique path between w and the root, then v is an **ancestor** of w, and w is a **descendant** of v.

19

Definitions

v is a child of u.
u is the parent of v.
v and w are siblings.

Vertices in the enclosed region
are descendants of u, which
is an ancestor of each.

Figure 10.6.1 A Rooted Tree

Example

Example: Consider the tree with root v_0 shown below.

a. What is the level of v_5? 2

b. What is the level of v_0? 0

c. What is the height of this rooted tree? 3

d. What are the children of v_3? v_5 and v_6
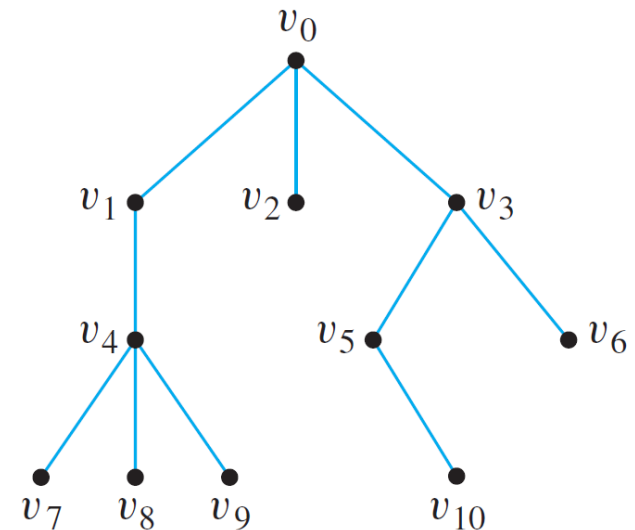
e. What is the parent of v_2? v_0

f. What are the siblings of v_8? v_7 and v_9

g. What are the descendants of v_3? v_5, v_6 and v_{10}

Binary Trees

Binary Trees

Definitions: Binary Tree, Full Binary Tree

A **binary tree** is a rooted tree in which every parent has at most two children. Each child is designated either a **left child** or a **right child** (but not both), and every parent has at most one left child and one right child.

A **full binary tree** is a binary tree in which each parent has exactly two children.

Definitions: Left Subtree, Right Subtree

Given any parent *v* in a binary tree *T*, if *v* has a left child, then the **left subtree** of *v* is the binary tree whose root is the left child of *v*, whose vertices consist of the left child of *v* and all its descendants, and whose edges consist of all those edges of *T* that connect the vertices of the left subtree.

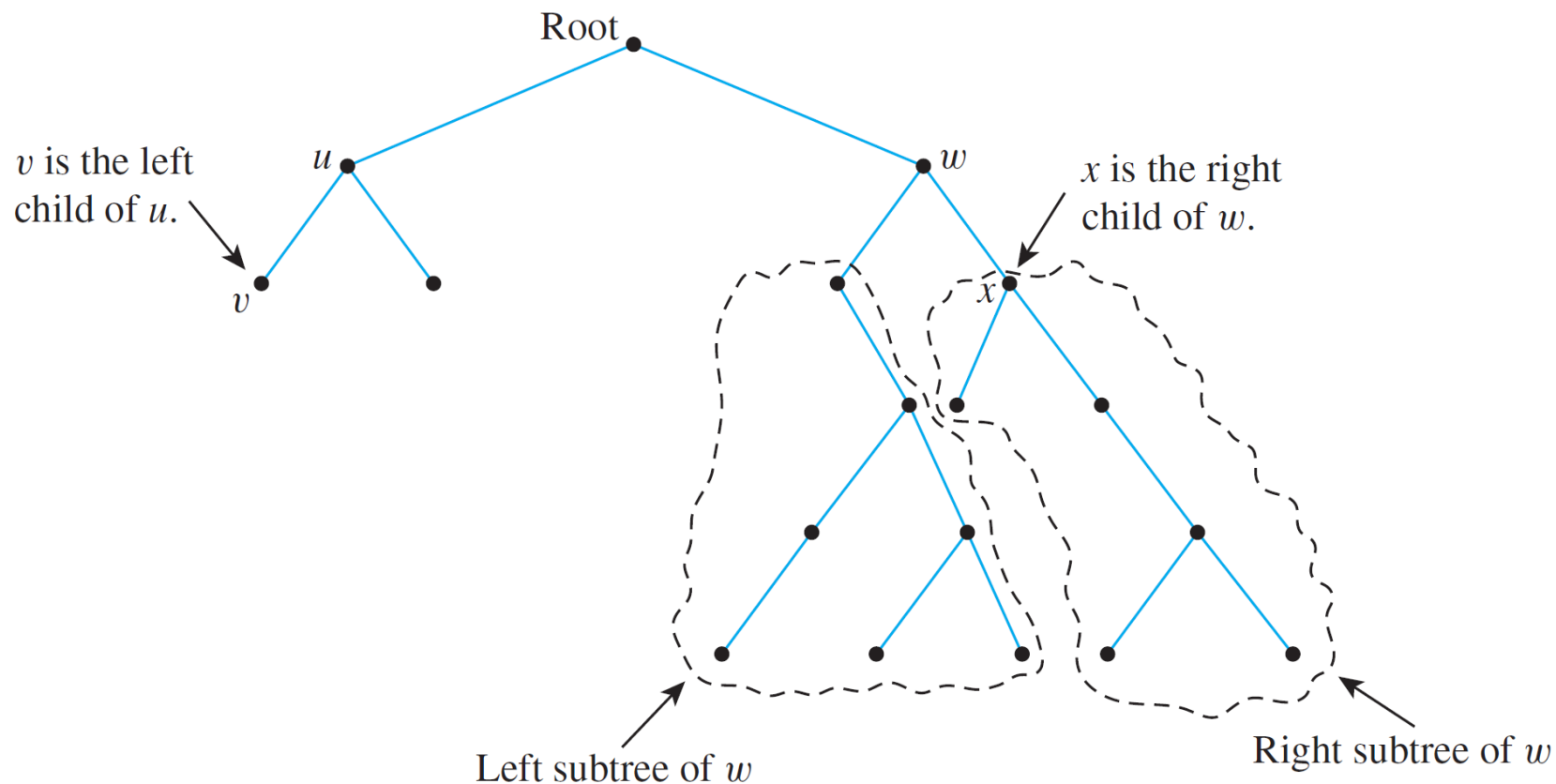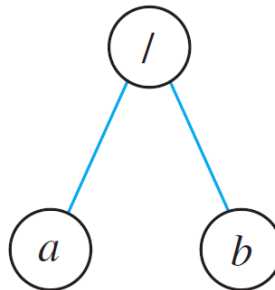The **right subtree** of *v* is defined analogously.

Binary Trees

Binary Trees

Figure 10.6.2 A Binary Tree
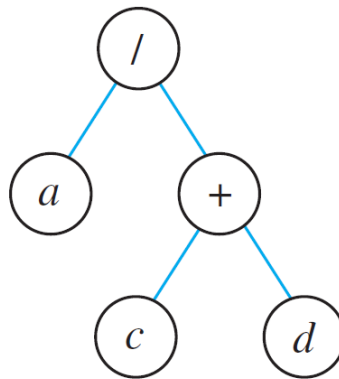
Example – Representation of Algebraic Expressions

Binary trees are used in many ways in computer science. One use is to represent algebraic expressions with arbitrary nesting of balanced parentheses.

For instance, the following (labeled) binary tree represents the expression *a/b*: The operator is at the root and acts on the left and right children of the root in left-right order.
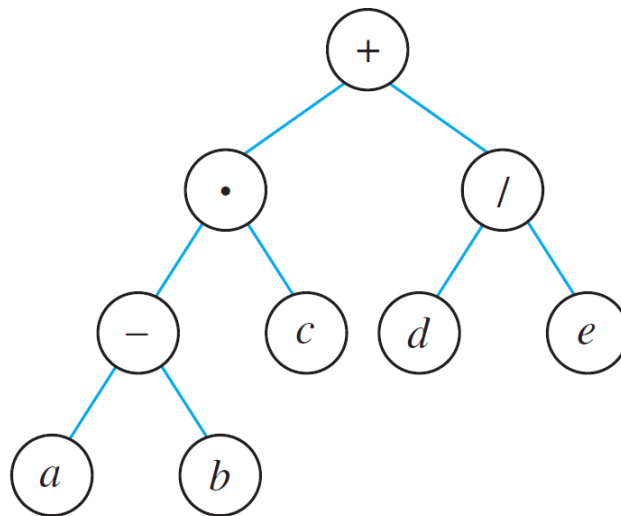
Example – Representation of Algebraic Expressions

More generally, the binary tree shown below represents the expression $a/(c + d)$. In such a representation, the internal vertices are arithmetic operators, the terminal vertices are variables, and the operator at each vertex acts on its left and right subtrees in left-right order.

Example – Representation of Algebraic Expressions

Draw a binary tree to represent the expression
$((a-b) \cdot c) + (d/e)$

Full Binary Tree

Theorem 10.6.1: Full Binary Tree Theorem

If *T* is a full binary tree with *k* internal vertices, then *T* has a total of $2k + 1$ vertices and has $k + 1$ terminal vertices (leaves).
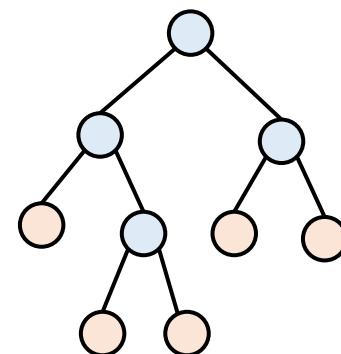
Proof:

1. Every vertex, except the root, has a parent.

2. Since every internal vertex of a full binary tree has exactly two children, the number of vertices that have a parent is twice the number of parents, or $2k$.

   #vertices of *T* = #vertices that have a parent +
            #vertices that do not have a parent
            = $2k + 1$

3. #terminal vertices = #vertices − #internal vertices
            = $2k + 1 - k = k + 1$

4. Therefore *T* has a total of **$2k + 1$ vertices** and has **$k + 1$ terminal vertices**.

Full Binary Tree

Q: Is there a full binary tree that has 10 internal vertices and 13 terminal vertices?

No, by Theorem 10.6.1, a full binary tree with 10 internal vertices has 10 + 1 = 11 terminal vertices.

Height and Terminal Vertices of a Binary Tree

Height and Terminal Vertices of a Binary Tree

Theorem 10.6.2

For non-negative integers h, if T is any binary tree with height h and t terminal vertices (leaves), then
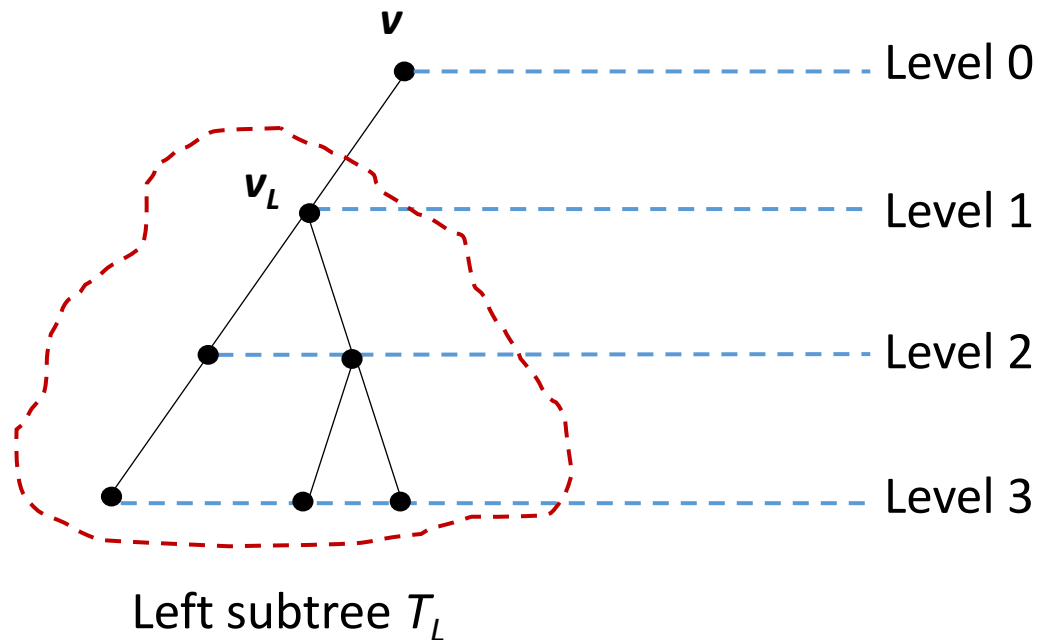
$$t \leq 2^h$$

Equivalently,

$$\log_2 t \leq h$$

This theorem says that the maximum number of terminal vertices (leaves) of a binary tree of height h is 2^h. Alternatively, a binary tree with t terminal vertices (leaves) has height of at least $\log_2 t$.

29

Height and Terminal Vertices of a Binary Tree

Proof: By strong mathematical induction

1. Let $P(h)$ be "If T is any binary tree of height h, then the number of leaves of T is at most 2^h."

2. $P(0)$: T consists of one vertex, which is a terminal vertex. Hence $t = 1 = 2^0$.

3. Show that for all integers $k \geq 0$, if $P(i)$ is true for all integers i from 0 through k, then $P(k+1)$ is true.

4. Let T be a binary tree of height $k + 1$, root v, and t leaves.

5. Since $k \geq 0$, hence $k + 1 \geq 1$ and so v has at least one child.

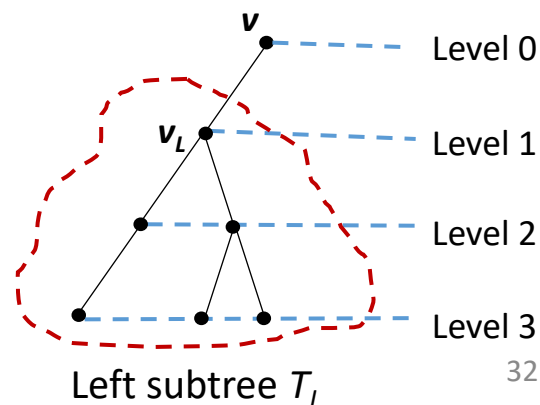6. We consider two cases: If v has only one child, or if v has two children.

Height and Terminal Vertices of a Binary Tree

Proof: (continued...)
Case 1 (v has only one child):

Left subtree T_L

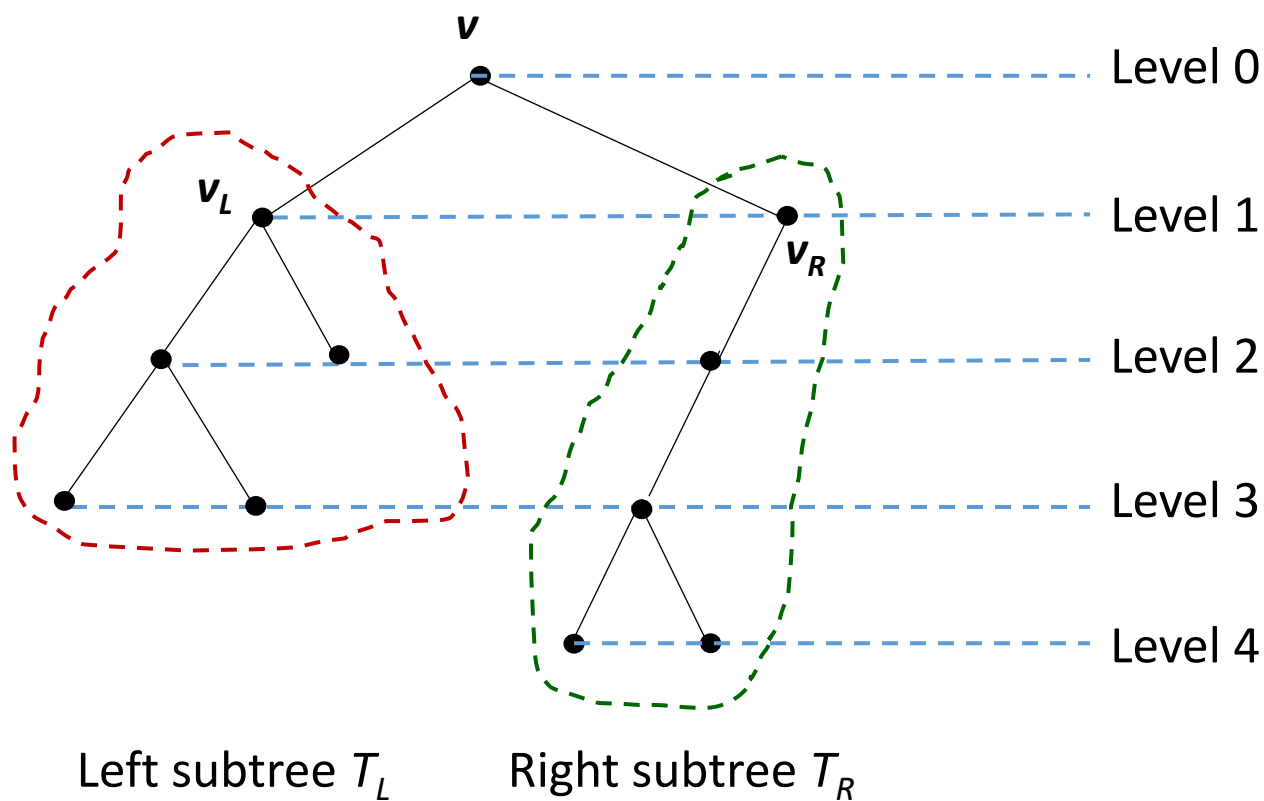## Height and Terminal Vertices of a Binary Tree

Proof: (continued…)

7. Case 1 (v has only one child):

    7.1 Without loss of generality, assume that v's child is a left child and denote it by v_L. Let T_L be the left subtree of v.

    7.2 Because v has only one child, v has degree 1 (leaf), so the total number of leaves in T equals the number of leaves in T_L + 1. Thus, if t_L is the number of leaves in T_L , then $t = t_L + 1$.

    7.3 By inductive hypothesis, $t_L \leq 2^k$ because the height of T_L is k, one less than the height of T.

    7.4 Also, because v has a child, $k+1 \geq 1$ and so $2^k \geq 2^0 = 1$.

    7.5 Therefore,

    $$t = t_L + 1 \leq 2^k + 1 \leq 2^k + 2^k = 2^{k+1}$$

Level 0

v_L Level 1

Level 2

Level 3

Left subtree T_L

32

Height and Terminal Vertices of a Binary Tree

Proof: (continued...)
Case 2 (*v* has two children):

Left subtree T_L Right subtree T_R

33

Height and Terminal Vertices of a Binary Tree

Proof: (continued...)

8.   Case 2 (*v* has two children):

8.1   Now *v* has a left child v_L and a right child v_R , and they are the roots of a left subtree T_L and a right subtree T_R respectively.

8.2   Let h_L and h_R be the heights of T_L and T_R respectively.

8.3   Then $h_L \leq k$ and $h_R \leq k$ since *T* is obtained by joining T_L and T_R and adding a level.

8.4   Let t_L and t_R be the number of leaves of T_L and T_R respectively.

8.5   Then, since both T_L and T_R have heights less than $k + 1$, by inductive hypothesis, $t_L \leq 2^{h_L}$ and $t_R \leq 2^{h_R}$.

8.6   Therefore,
$$t = t_L + t_R \leq 2^{h_L} + 2^{h_R} \leq 2^k + 2^k = 2^{k+1}$$

9.   In both cases, *P*(*k*+1) is true.

10. Hence if *T* is any binary tree with height *h* and *t* terminal vertices (leaves), then $t \leq 2^h$.

34

Height and Terminal Vertices of a Binary Tree

Q: Is there a binary tree that has height 5 and 38 terminal vertices?

No, by Theorem 10.6.2, any binary tree T with height 5 has at most $2^5 = 32$ terminal vertices, so such a tree cannot have 38 terminal vertices.

Binary Tree Traversal

Binary Tree Traversal

Tree traversal (also known as tree search) is the process of visiting each node in a tree data structure exactly once in a systematic manner.
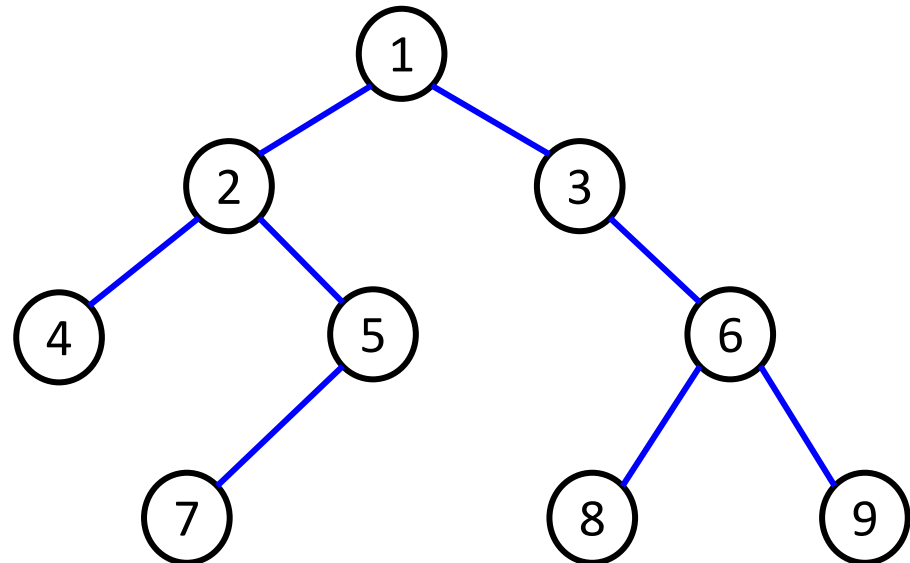
There are two types of traversal: breadth-first search (BFS) or depth-first search (DFS).

The following sections describe BFS and DFS on binary trees, but in general they can be applied on any type of trees, or even graphs.
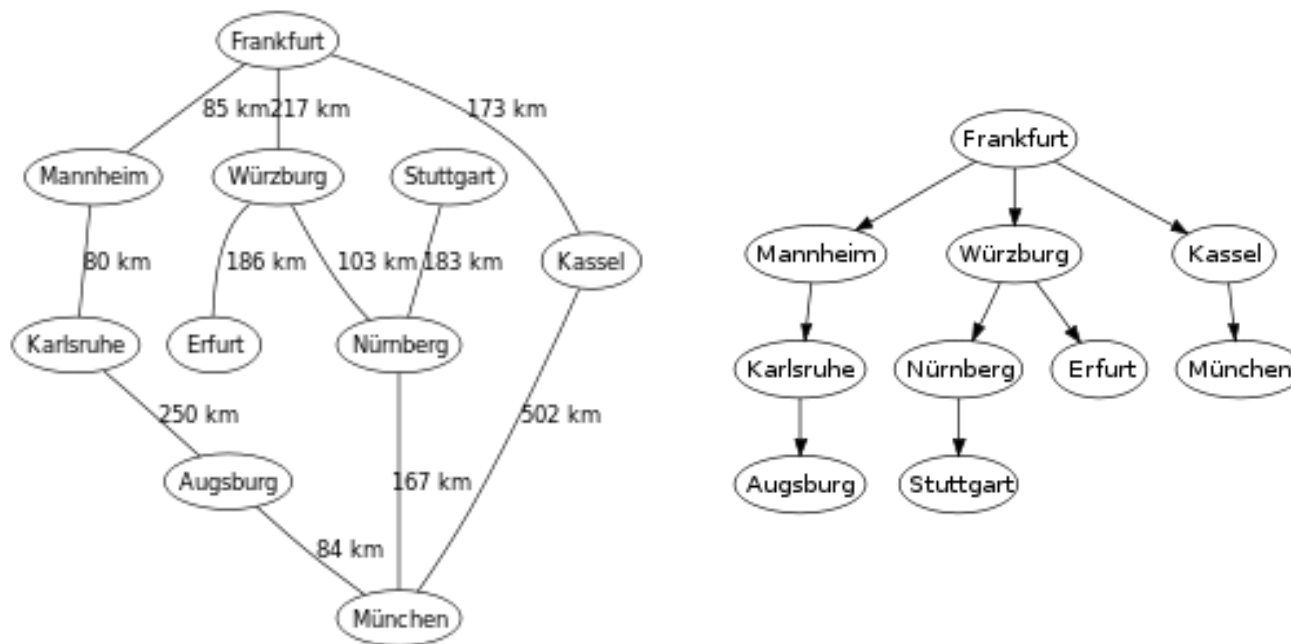
Binary Tree Traversal

Breadth-First Search

In breadth-first search (by E.F. Moore), it starts at the root and visits its adjacent vertices, and then moves to the next level.

The figure shows the order of the vertices visited.

Binary Tree Traversal

Breadth-First Search

The figure on the left shows a graph representing cities in Germany. The figure on the right shows the breadth-first traversal on the graph, starting with Frankfurt.

Acknowledgement: Wikipedia https://en.wikipedia.org/wiki/Breadth-first_search

Binary Tree Traversal

Depth-First Search

There are three types of depth-first traversal:

- **Pre-order**
  - Print the data of the root (or current vertex)
  - Traverse the left subtree by recursively calling the pre-order function
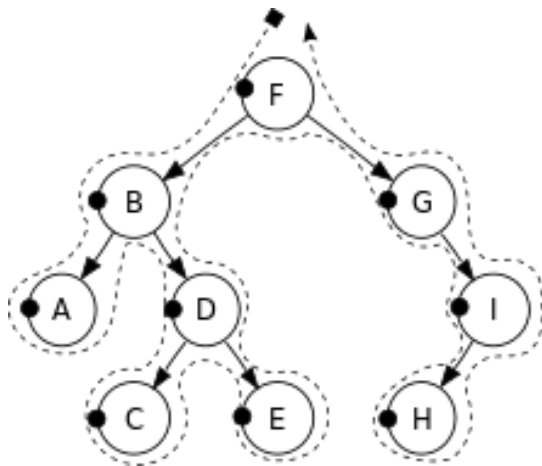 - Traverse the right subtree by recursively calling the pre-order function

- **In-order**
  - Traverse the left subtree by recursively calling the in-order function
  - Print the data of the root (or current vertex)
  - Traverse the right subtree by recursively calling the in-order function
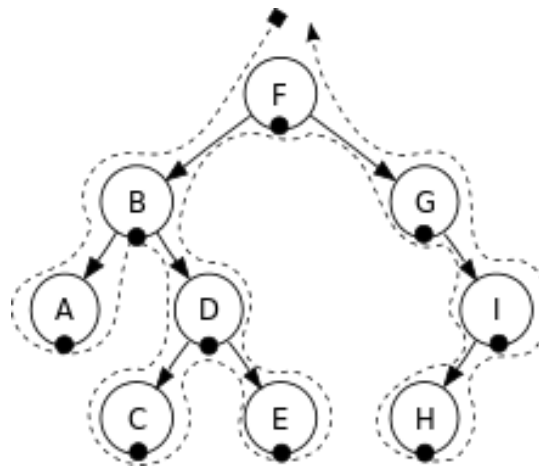
- **Post-order**
  - Traverse the left subtree by recursively calling the post-order function
  - Traverse the right subtree by recursively calling the post-order function
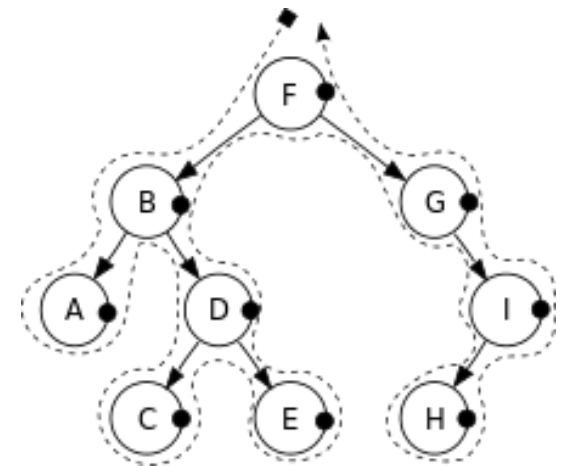 - Print the data of the root (or current vertex)

Binary Tree Traversal

Depth-First Search

Pre-order:

F, B, A, D, C, E, G, I, H

In-order:

A, B, C, D, E, F, G, H, I

Post-order:

A, C, E, D, B, H, I, G, F

Acknowledgement: Wikipedia https://en.wikipedia.org/wiki/Tree_traversal

10.7 Spanning Trees and Shortest Paths

Trees
○ ○

Rooted Trees
○ ○ ○

Spanning trees and Shortest Paths
● ○ ○ ○ ○

Definitions

An East Coast airline company wants to expand service to the Midwest and has received permission from the Federal Aviation Authority to fly any of the routes shown in Figure 10.7.1.
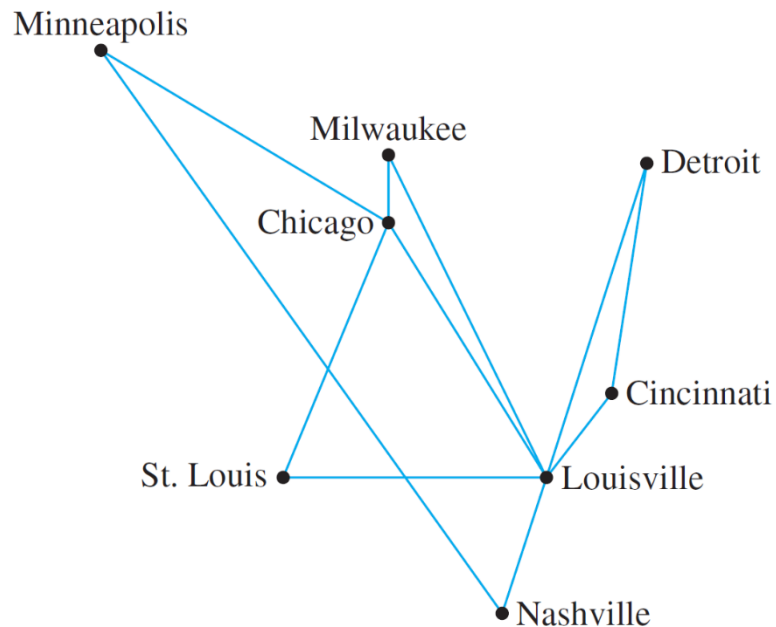

Figure 10.7.1

Definitions

The company wishes to legitimately advertise service to all the cities shown but, for reasons of economy, wants to use the least possible number of individual routes to connect them. One possible route system is given in Figure 10.7.2, where the chosen routes are in red.
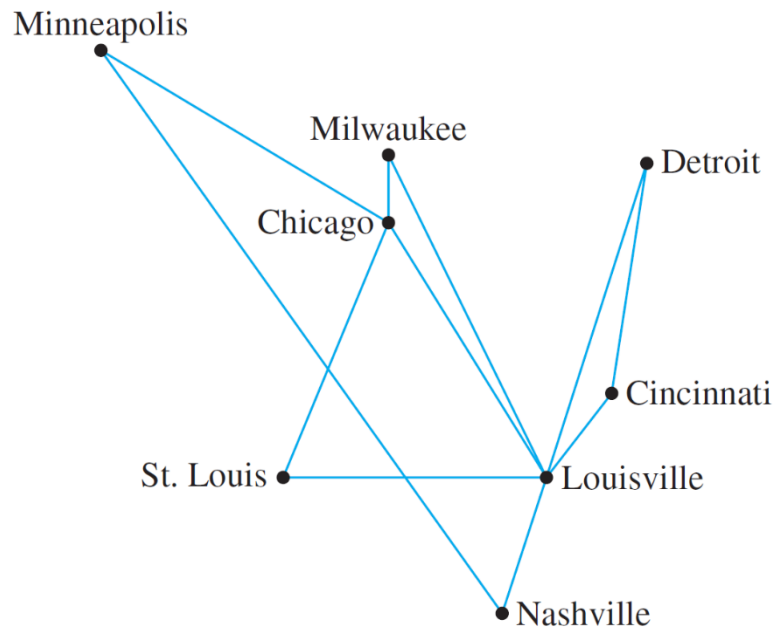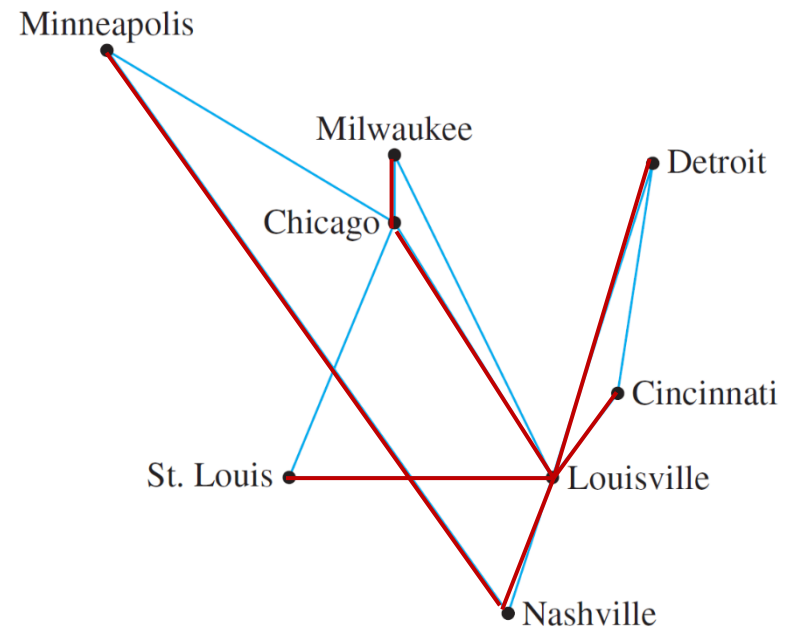

Figure 10.7.1

Figure 10.7.2

Definitions

Is the number of individual routes minimal?

The fact is that the graph of any system of routes that satisfies the company's wishes is a tree, because if the graph were to contain a circuit, then one of the routes in the circuit could be removed without disconnecting the graph (by Lemma 10.5.3), and that would give a smaller total number of routes.

Lemma 10.5.3

If G is any connected graph, C is any circuit in G, and one of the edges of C is removed from G, then the graph that remains is still connected.
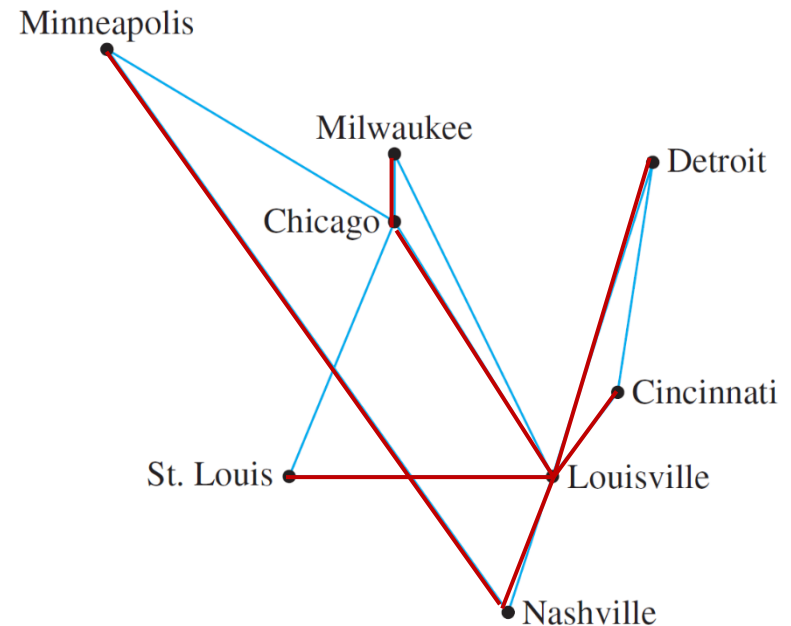
Minneapolis

Milwaukee

Detroit

Chicago

Cincinnati

St. Louis

Louisville

Nashville

Figure 10.7.2

Definitions

What you have seen is a spanning tree.
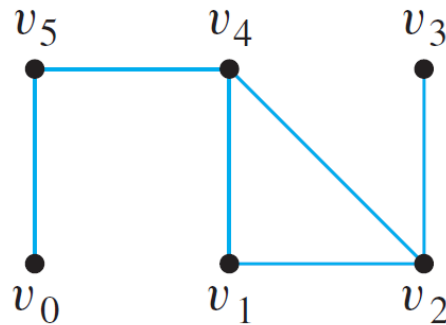
Definition: Spanning Tree

A **spanning tree** for a graph G is a subgraph of G that contains every vertex of G and is a tree.
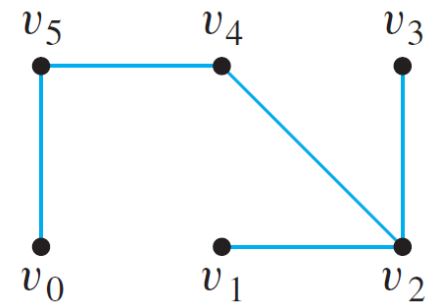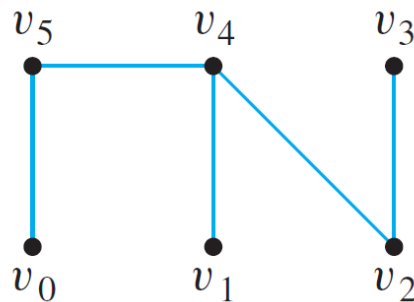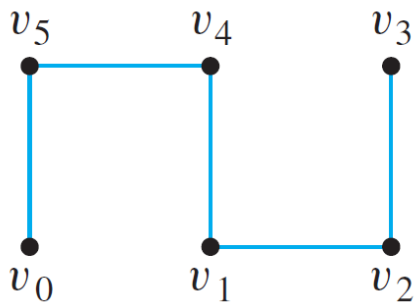
Proposition 10.7.1

1. Every connected graph has a spanning tree.
2. Any two spanning trees for a graph have the same number of edges.

Definitions

Example: Find all spanning trees for the graph *G* below.

The graph *G* has one circuit $v_2 v_1 v_4 v_2$ and removal of any edge of the circuit gives a tree. Hence there are three spanning trees for *G*.

Minimum Spanning Trees

Minimum Spanning Trees

The graph of the routes allowed by the Federal Aviation Authority shown in Figure 10.7.1 can be annotated by adding the distances (in miles) between each pair of cities. This is called a weighted graph.

Now, suppose the airline company wants to serve all the cities shown, but with a route system that minimizes the total mileage.

Figure 10.7.3

47

Minimum Spanning Trees

Definitions: Weighted Graph, Minimum Spanning Tree

A **weighted graph** is a graph for which each edge has an associated positive real number **weight** . The sum of the weights of all the edges is the **total weight** of the graph.

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.

If *G* is a weighted graph and *e* is an edge of *G*, then *w(e)* denotes the weight of *e* and *w(G)* denotes the total weight of *G*.

48

Kruskal's Algorithm

Kruskal's Algorithm (Joseph B. Kruskal, 1956)

Joseph B. Kruskal
(1928 – 2010)

In Kruskal's algorithm, the edges of a connected weighted graph are examined one by one in order of increasing weight.

At each stage the edge being examined is added to what will become the minimum spanning tree, provided that this addition does not create a circuit.

After $n – 1$ edges have been added (where n is the number of vertices of the graph), these edges, together with the vertices of the graph, form a minimum spanning tree for the graph.

49

Kruskal's Algorithm

Algorithm 10.7.1 Kruskal

Input: *G* [a connected weighted graph with *n* vertices]

Algorithm:

1. Initialize *T* to have all the vertices of *G* and no edges.
2. Let *E* be the set of all edges of *G*, and let *m* = 0.
3. While (*m* < *n* – 1)
   3a. Find an edge *e* in *E* of least weight.
   3b. Delete *e* from *E*.
   3c. If addition of *e* to the edge set of *T* does not produce a circuit, then add *e* to the edge set of *T* and set *m* = *m* + 1
   End while

Output: *T* [*T* is a minimum spanning tree for *G*]

Kruskal's Algorithm

Example: Describe the action of Kruskal's algorithm on the graph shown in Figure 10.7.4, where $n = 8$.

Figure 10.7.4

Kruskal's Algorithm

Using Kruskal's algorithm we can formulate the following table.

	Edge considered	Wt	Action taken
1	→ Chi – Mil	74	added
2	→ Lou – Cin	83	added
3	→ Lou – Nas	151	added
4	→ Cin – Det	230	added
5	→ StL – Lou	242	added
6	→ StL – Chi	262	added
7	→ Chi – Lou	269	not added
8	→ Lou – Det	306	not added
9	→ Lou – Mil	348	not added
10	→ Min – Chi	355	added

Figure 10.7.4

Total weight = 1397

52

Kruskal's Algorithm

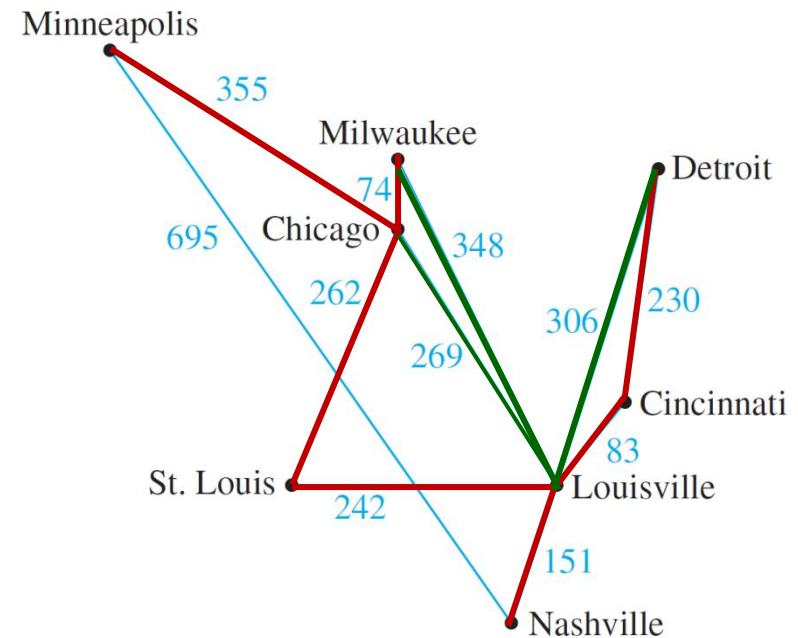When Kruskal's algorithm is used on a graph in which some edges have the same weight as others, more than one minimum spanning tree can occur as output.

To make the output unique, the edges of the graph can be placed in an array and edges having the same weight can be added in the order they appear in the array.

Prim's Algorithm

Prim's Algorithm (Robert C. Prim, 1957)

Robert C. Prim
(1921 - 2021)

Prim's algorithm works differently from Kruskal's. It builds a minimum spanning tree T by expanding outward in <u>connected links</u> from some vertex.

One edge and one vertex are added at each stage. The edge added is the one of least weight that connects the vertices already in T with those not in T, and the vertex is the endpoint of this edge that is not already in T.

Prim's Algorithm

Algorithm 10.7.2 Prim

Input: *G* [a connected weighted graph with *n* vertices]

Algorithm:
1. Pick a vertex *v* of *G* and let *T* be the graph with this vertex only.
2. Let *V* be the set of all vertices of *G* except *v*.
3. For *i* = 1 to *n* − 1

   3a. Find an edge *e* of *G* such that (1) *e* connects *T* to one of the vertices in *V*, and (2) *e* has the least weight of all edges connecting *T* to a vertex in *V*. Let *w* be the endpoint of *e* that is in *V*.

   3b. Add *e* and *w* to the edge and vertex sets of *T*, and delete *w* from *V*.

Output: *T* [*T* is a minimum spanning tree for *G*]

Prim's Algorithm

Example: Describe the action of Prim's algorithm on the graph shown in Figure 10.7.6, using the Minneapolis vertex as a starting point.
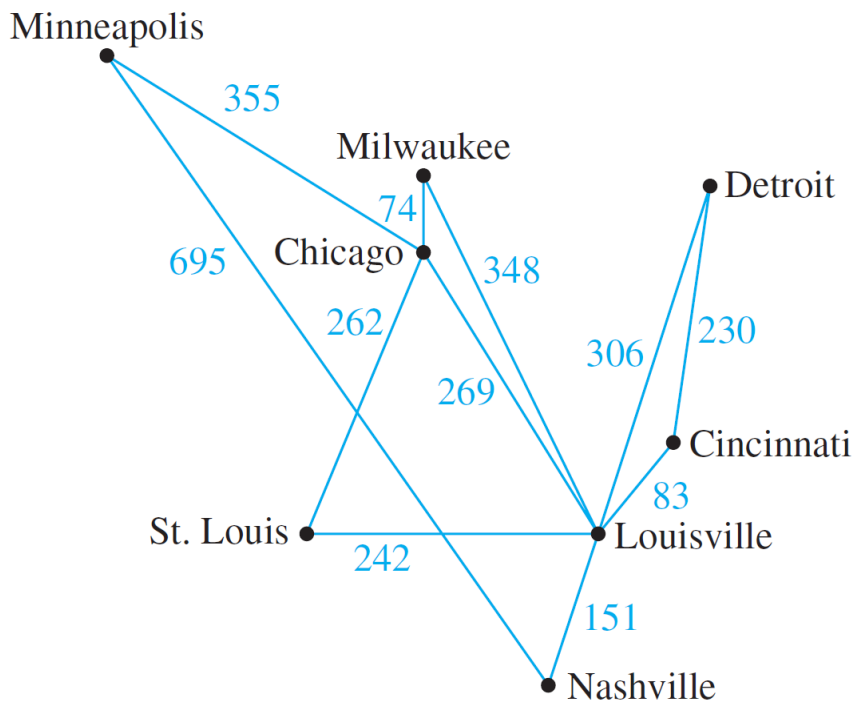

Figure 10.7.6

56

Prim's Algorithm

Using Prim's algorithm we can formulate the following table.

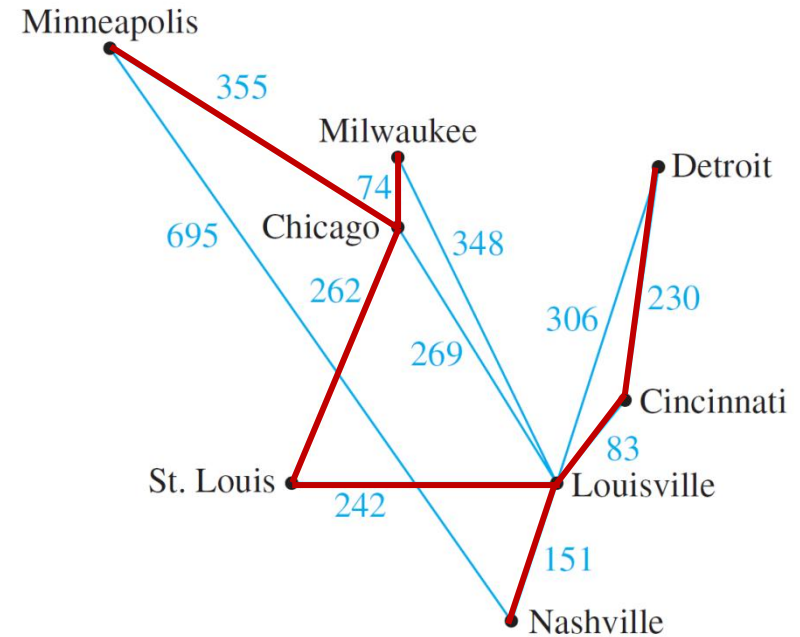|	Vertex added	Edge added	Weight
0	Minneapolis		
1	Chicago	Min – Chi	355
2	Milwaukee	Chi – Mil	74
3	St. Louis	Chi – StL	262
4	Louisville	StL – Lou	242
5	Cincinnati	Lou – Cin	83
6	Nashville	Lou – Nas	151
7	Detroit	Cin – Det	230

Total weight = 1397

Figure 10.7.6

57

Prim's Algorithm

Note that the tree obtained is the same as that obtained by Kruskal's algorithm, but the edges are added in a different order.

As with Kruskal's algorithm, in order to ensure a unique output, the edges of the graph could be placed in an array and those with the same weight could be added in the order they appear in the array.

Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm

Edsger W. Dijkstra
(1930 – 2002)

In 1959, Edsger Dijkstra developed an algorithm to find the shortest path between a starting vertex (source) and an ending vertex (destination) in a weighted graph in which all weights are positive.
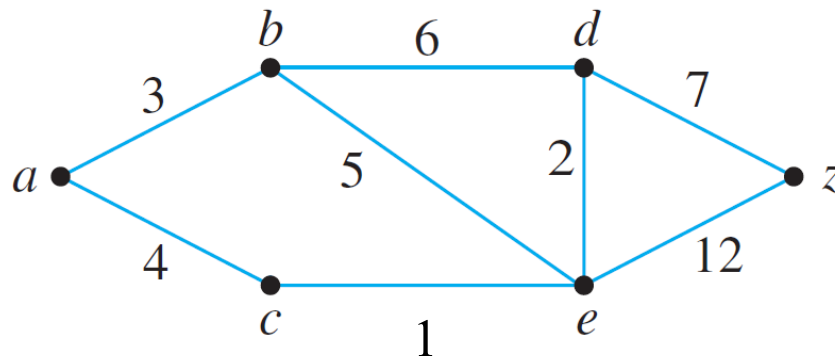
Somewhat similar to Prim's algorithms, it works outward from the source a, adding vertices and edges one by one to construct a shortest path tree T. It differs from Prim's algorithm in the way it chooses the next vertex to add, ensuring that for each added vertex v, the length of the shortest path from a to v has been identified.

Dijkstra's Shortest Path Algorithm

Intuition behind Dijkstra's algorithm:

- Report the vertices in increasing order of their distance from the source vertex.

- Construct the shortest path tree edge by edge; at each step adding one new edge, corresponding to construction of shortest path to the current new vertex.
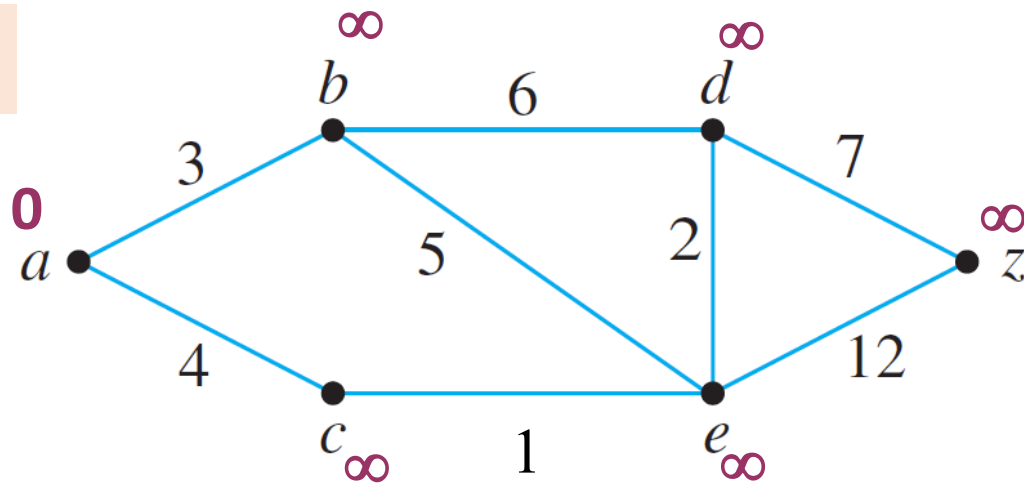
Example: To find shortest part from vertex *a* to vertex *z*.

Dijkstra's Shortest Path Algorithm

At the start, assign every vertex u a label $L(u)$, which is the current best estimate of the length of the shortest path from a to u.

$L(a)$ is set to 0.

$L(u)$ of each vertex u other than a is set to ∞.
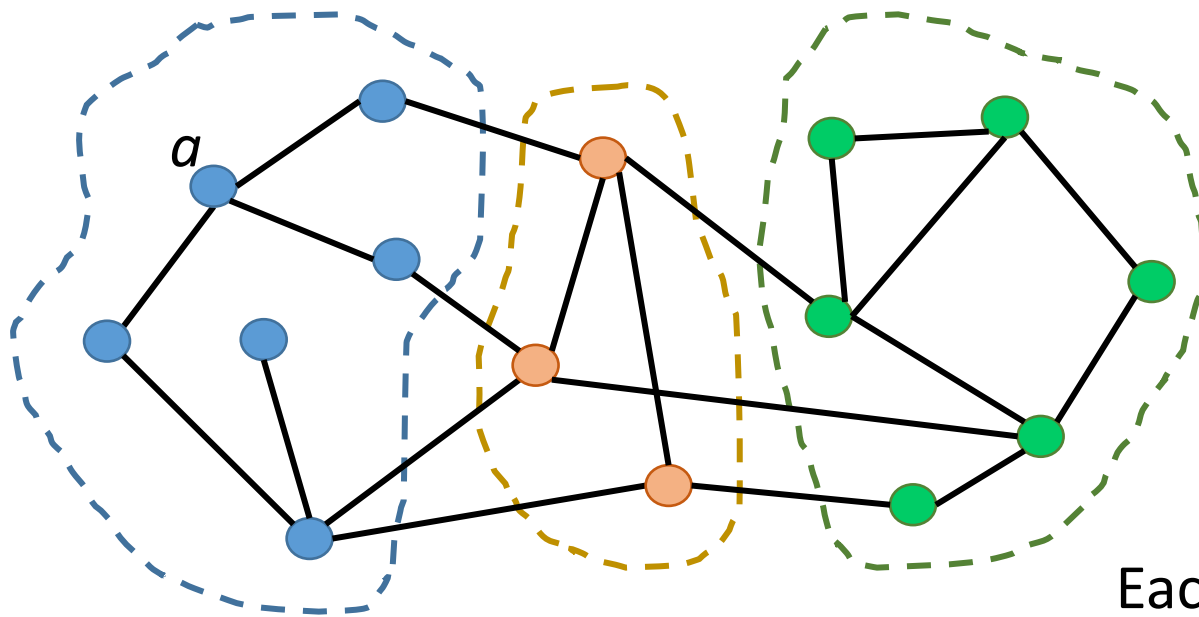
Dijkstra's Shortest Path Algorithm

As the algorithm progresses, the values of $L(u)$ are updated, eventually becoming the actual lengths of the shortest paths from a to u.

We construct the shortest path tree T outward from a.

At each stage of the algorithm, the only vertices that are candidates to join T are those that are adjacent to at least one vertex of T. We call these candidates the set of "fringe" vertices.

The graph G can be thought of as divided into 3 parts: the tree T that is being built up, the set of "fringe" vertices, and the rest of the vertices in G.

Dijkstra's Shortest Path Algorithm

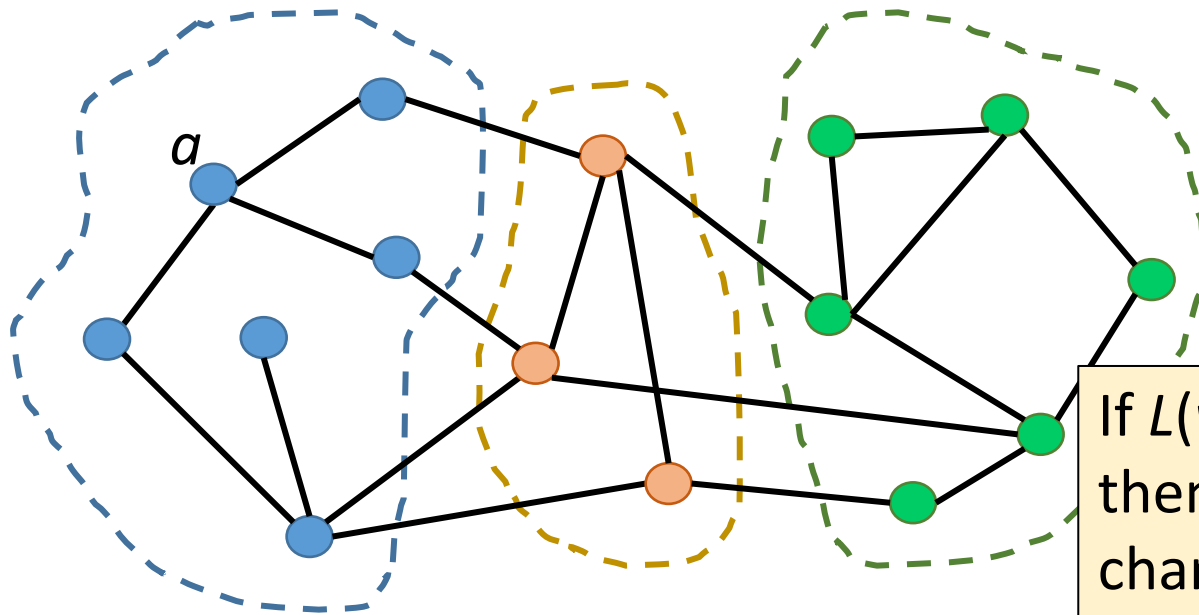**The rest of the vertices**

a

Shortest path tree *T*

"Fringe" vertices

Each fringe vertex is a candidate to be the next vertex added to *T*.

All vertices in *T* already have their final *L(u)* value computed.

The one that is chosen is the one for which the length of the shortest path to it from *a* through *T* is a minimum among all the vertices in the fringe.

Dijkstra's Shortest Path Algorithm

a

If $L(v) + w(v, u) < L(u)$, then the value of $L(u)$ is changed to $L(v) + w(v, u)$.

After each addition of a vertex *v* to *T*, each fringe vertex *u* adjacent to *v* is examined and two numbers are compared: the current value of $L(u)$ and the value of $L(v) + w(v, u)$, where $L(v)$ is the length of the shortest path to *v* (in *T*) and $w(v, u)$ is the weight of the edge joining *v* and *u*.

Dijkstra's Shortest Path Algorithm

Algorithm 10.7.3 Dijkstra

Inputs:

- *G* [a connected simple graph with positive weight for every edge]
- ∞ [a number greater than the sum of the weights of all the edges in *G*]
- *w*(*u*, *v*) [the weight of edge {*u*, *v*}]
- *a* [the source vertex]
- *z* [the destination vertex]

Algorithm:

1. Initialize *T* to be the graph with vertex *a* and no edges.
   Let *V*(*T*) be the set of vertices of *T*, and let *E*(*T*) be the set of edges of *T*.

2. *L*(*a*) ← 0, and for all vertices *u* in *G* except *a*, *L*(*u*) ← ∞ .
   [The number *L*(*u*) is called the label of *u*.]

3. Initialize *v* ← *a* and *F* ← {*a*}. [The symbol *v* is used to denote the vertex most recently added to *T*.]

Dijkstra's Shortest Path Algorithm
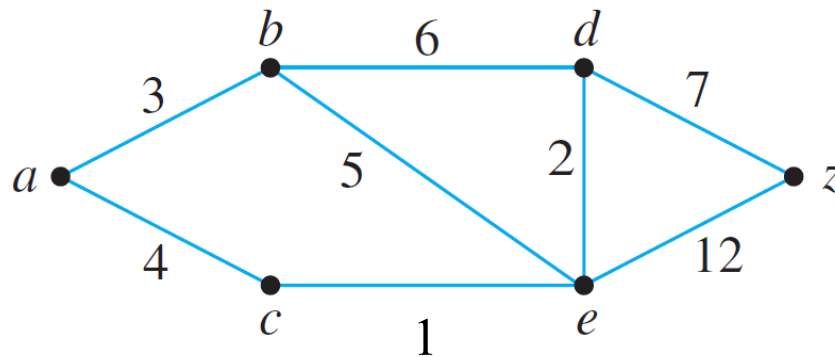
Algorithm 10.7.3 Dijkstra (continued...)

Let Adj(x) denote the set of vertices adjacent to vertex x.

4. while ($z \notin V(T)$)

    a. $F \leftarrow (F - \{v\}) \cup \{$vertices $\in$ Adj(v) and $\notin V(T)\}$

        [The set F is the set of fringe vertices.]

    b.   For each vertex $u \in$ Adj(v) and $\notin V(T)$,

            if $L(v) + w(v, u) < L(u)$ then

                 $L(u) \leftarrow L(v) + w(v, u)$

                 $D(u) \leftarrow v$

> [The notation $D(u)$ is introduced to keep track of which vertex in T gave rise to the smaller value.]

    c.   Find a vertex x in F with the smallest label.
         Add vertex x to $V(T)$, and add edge $\{D(x), x\}$ to $E(T)$.
         $v \leftarrow x$

Output: $L(z)$ [this is the length of the shortest path from a to z.]
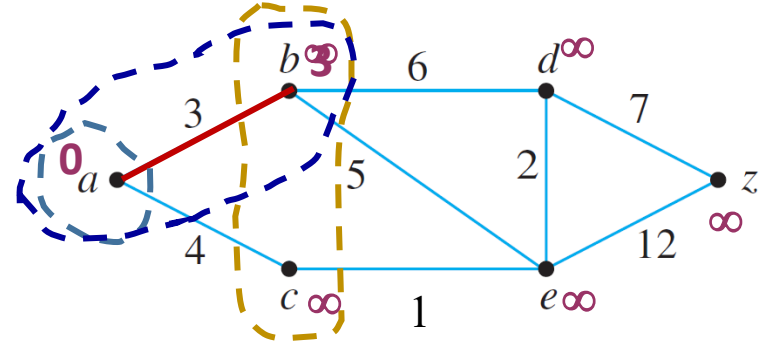
67

Dijkstra's Shortest Path Algorithm

Example: 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*.

Dijkstra's Shortest Path Algorithm

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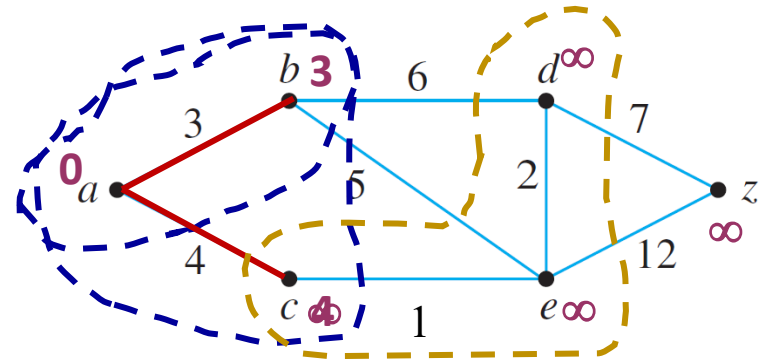$V(T) = \{a\}$, $E(T) = \varnothing$, and $F = \{a\}$

During iteration:

$F = \{b, c\}$, $L(b) = 3$, $L(c) = 4$.

Since $L(b) < L(c)$, b is added to $V(T)$ and $\{a, b\}$ is added to $E(T)$.

Dijkstra's Shortest Path Algorithm

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

During iteration:
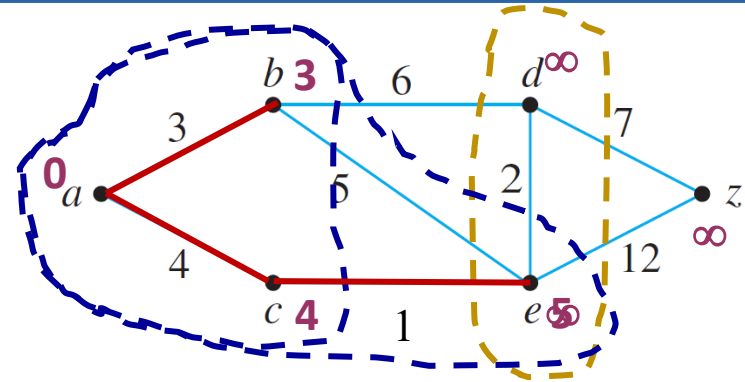
$F = \{c, d, e\}, L(c) = 4, L(d) = 9, L(e) = 8.$

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

Dijkstra's Shortest Path Algorithm

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

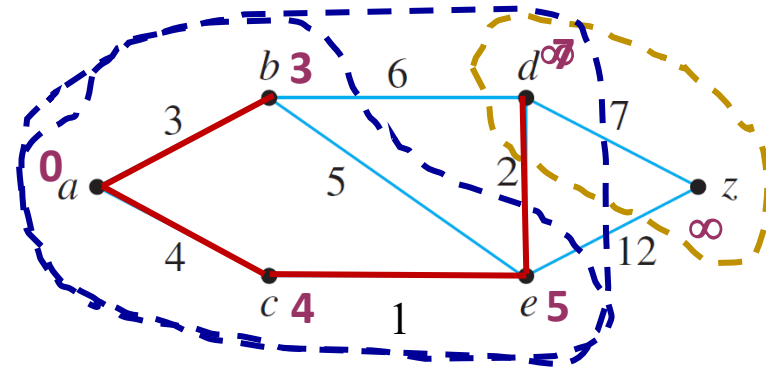**During iteration:**

$F = \{d, e\}$, $L(d) = 9$, $L(e) = 5$.

$L(e)$ 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 $V(T)$ and $\{c, e\}$ is added to $E(T)$.

71

Dijkstra's Shortest Path Algorithm

Step 4: Going into the **while** loop:

$V(T) = \{a, b, c, e\}$,

$E(T) = \{\{a, b\}, \{a, c\}, \{c, e\}\}$

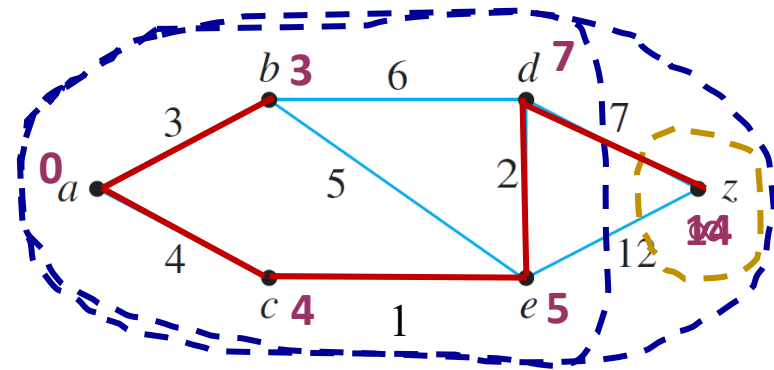**During iteration:**

$F = \{d, z\}$, $L(d) = 7$, $L(z) = 17$.

$L(d)$ 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 $V(T)$ and $\{e, d\}$ is added to $E(T)$.

Dijkstra's Shortest Path Algorithm

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 z than *acez*, which has length 17.

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

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

73

Dijkstra's Shortest Path Algorithm

Keeping track of the steps in a table is a convenient way to show the action of Dijkstra's algorithm. Table 10.7.1 does this for the graph in the previous example.

Step	$V(T)$	$E(T)$	F	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(z)$
0	$\{a\}$	Ø	$\{a\}$	**0**	∞	∞	∞	∞	∞
1	$\{a\}$	Ø	$\{b,\ c\}$	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	$\{a, b, c, e, d, z\}$	$\{\{a, b\}, \{a, c\}, \{c, e\}, \{e, d\}, \{d, z\}\}$							

Table 10.7.1