CHAPTER 10

THEORY OF GRAPHS AND TREES

10.5 Rooted Trees

Rooted Trees (1/3)

In mathematics, a rooted tree is a tree in which one vertex has been distinguished from the others and is designated the *root*. Given any other vertex *v* in the tree, there is a unique path from the root to *v*.

The number of edges in such a path is called the level of *v*, and the *height* of the tree is the length of the longest such path. It is traditional in drawing rooted trees to place the root at the top and show the branches descending from it.

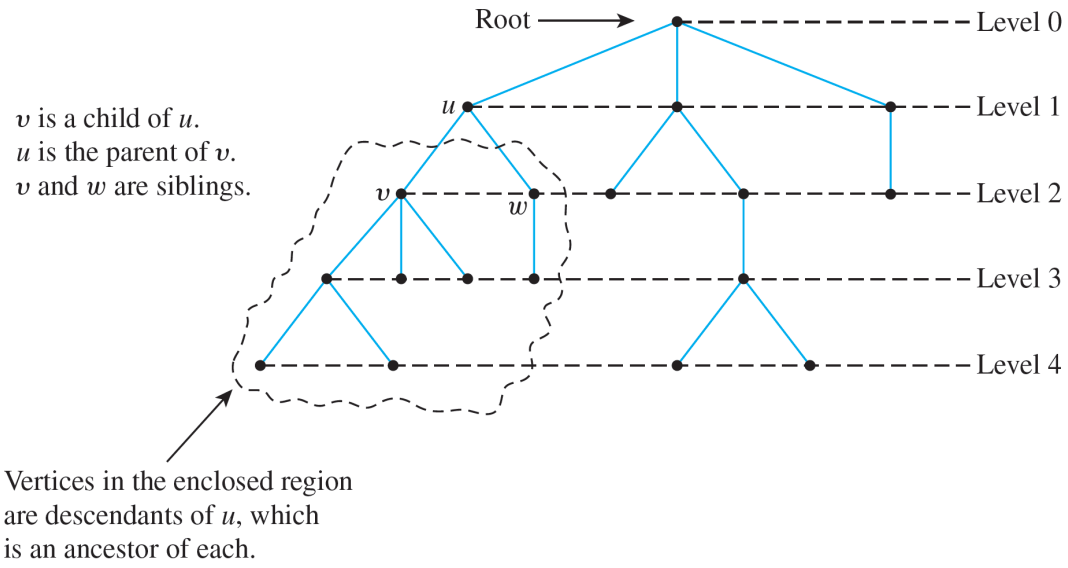
Rooted Trees (2/3)

Definition

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. 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.

Rooted Trees (3/3)

These terms are illustrated in Figure 10.5.1.



A Rooted Tree

Figure 10.5.1

Example 10.5.1 – *Rooted Trees (1/2)*

Consider the tree with root v_0 shown below.

a. What is the level of v_5?

b. What is the level of v_0?

c. What is the height of this rooted tree?

d. What are the children of v_3?

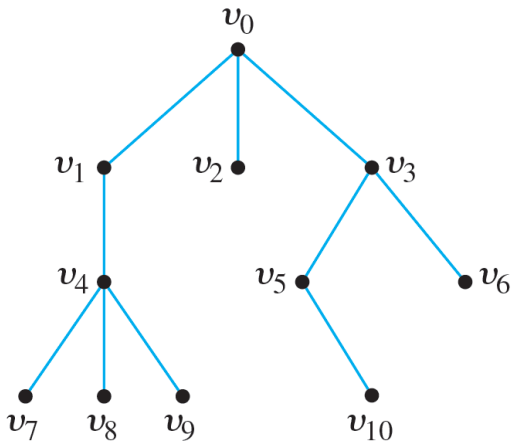
e. What is the parent of v_2?

f. What are the siblings of v_8?

g. What are the descendants of v_3?

Example 10.5.1 – *Rooted Trees (2/2)*

h. How many leaves (terminal vertices) are on the tree?



Example 10.5.1 – *Solution*

a. 2

b. 0

c. 3

d. v_5 and v_6

e. v_0

f. v_7 and v_9

g. v_5, v_6, v_{10}

h. 6

Binary Trees

Binary Trees (1/5)

When every vertex in a rooted tree has at most two children and each child is designated either the (unique) left child or the (unique) right child, the result is a *binary tree.*

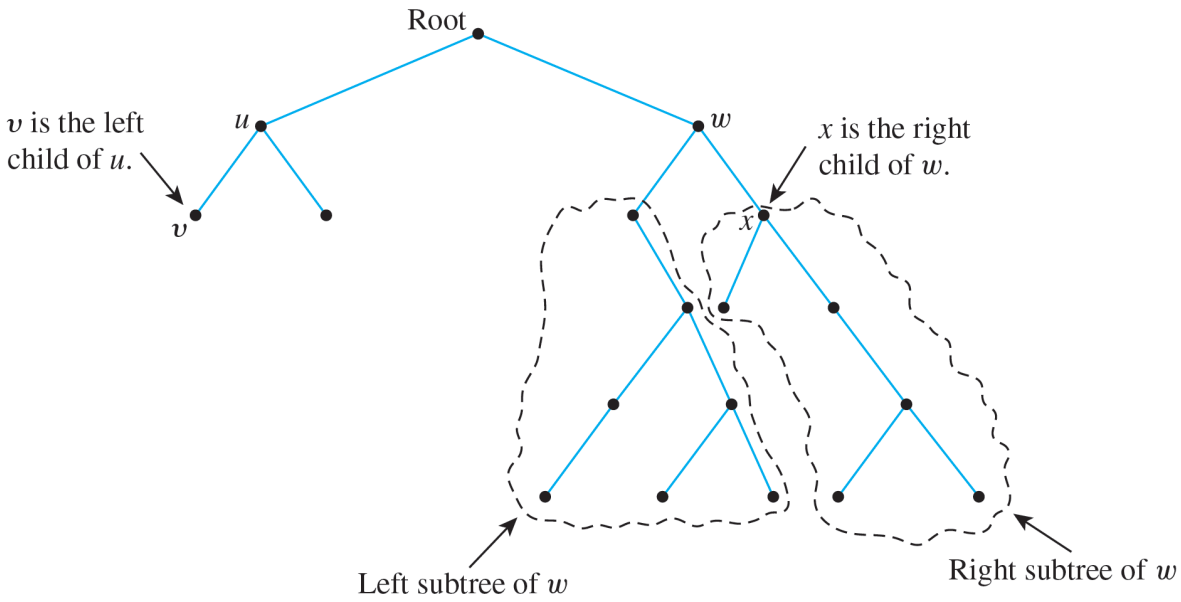
Definition

A **binary tree** is a rooted tree in which every parent has at most two children. Each child in a binary tree 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.

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 (2/5)

These terms are illustrated in Figure 10.5.2.

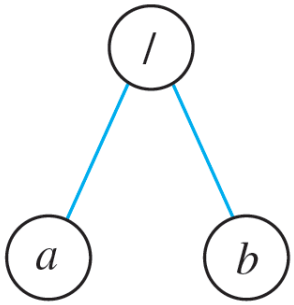


A Binary Tree

Figure 10.5.2

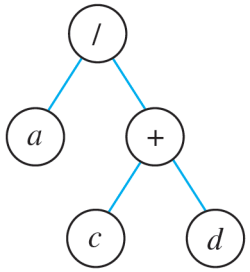
Example 10.5.2 – *Representation of Algebraic Expressions (1/2)*

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-to-right order.



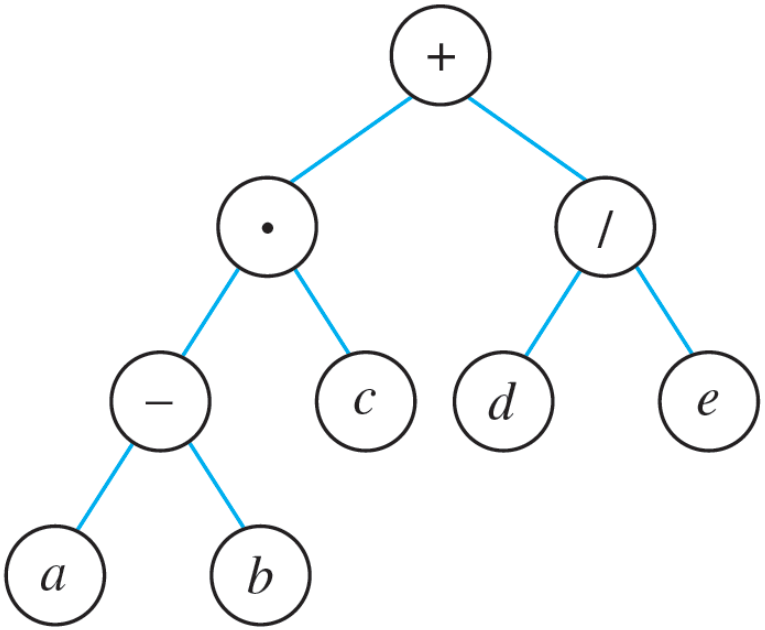
Example 10.5.2 – *Representation of Algebraic Expressions (2/2)*

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



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

Example 10.5.2 – *Solution*



Binary Trees (3/5)

An interesting theorem about binary trees says that if you know the number of internal vertices of a full binary tree, then you can calculate both the total number of vertices and the number of leaves, and conversely. More specifically, a full binary tree with k internal vertices has a total of $2k + 1$ vertices of which $k + 1$ are leaves.

Theorem 10.5.1

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

Example 10.5.3 – *Determining Whether a Certain Full Binary Tree Exists*

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

Example 10.5.3 – *Solution*

No. By Theorem 10.5.1, a full binary tree with 10 internal vertices has 10 + 1 = 11 leaves, not 13.

Theorem 10.5.1

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

Binary Trees (4/5)

Another interesting theorem about binary trees specifies the maximum number of leaves of a binary tree of a given height. Specifically, the maximum number of leaves of a binary tree of height *h* is 2^h. Another way to say this is that a binary tree with *t* leaves has height of at least $\log_2 t$.

Theorem 10.5.2

For every integer $h \geq 0$, if T is any binary tree with height h and t leaves, then

$$t \leq 2^h.$$

Equivalently:

$$\log_2 t \leq h.$$

Example 10.5.4 – *Determining Whether a Certain Binary Tree Exists*

Is there a binary tree that has height 5 and 38 leaves?

Example 10.5.4 – *Solution*

No. By Theorem 10.5.2, any binary tree T with height 5 has at most $2^5 = 32$ leaves, so such a tree cannot have 38 leaves.

Theorem 10.5.2

For every integer $h \geq 0$, if T is any binary tree with height h and t leaves, then

$$t \leq 2^h.$$

Equivalently: $\log_2 t \leq h.$

Binary Trees (5/5)

Corollary 10.5.3

A full binary tree of height h has 2^h leaves.

Binary Search Trees

Binary Search Trees (1/6)

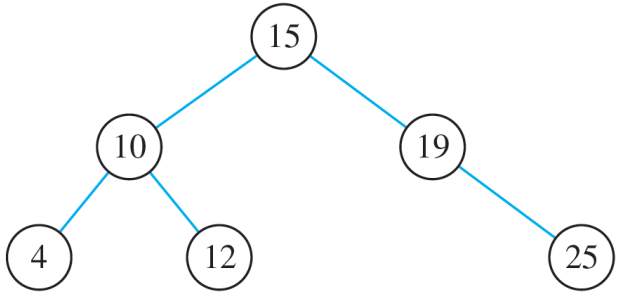
A binary search tree is a kind of binary tree in which data records, such as customer information, can be stored, searched, and processed very efficiently. To place records into a binary search tree, it must be possible to arrange them in a total order.

In case they do not have a natural total order of their own, an element of a totally ordered set, such as a number or a word and called a **key**, may be added to each record. The keys are inserted into the vertices of the tree and provide access to the records to which they are attached.

Binary Search Trees (2/6)

Once it is built, a binary search tree has the following property: for every internal vertex *v*, all the keys in the left subtree of *v* are less than the key in *v*, and all the keys in the right subtree of *v* are greater than the key in *v*.

For example, check that the following is a binary search tree for the set of records with the following keys: 15, 10, 19, 25, 12, 4.



Binary Search Trees (3/6)

To build a binary search tree, start by making a root and insert a key into it.

To add a new key, compare it to the key at the root. If the new key is less than the key at the root, give the root a left child and insert the new key into it.

If the key is greater than the key at the root, give the root a right child and insert the new key into it.

Binary Search Trees (4/6)

After the first couple of keys have been added, the root and other vertices may already have left and right children.

So to add a key at a subsequent stage, work down the tree to find a place to put the new key, starting at the root and either moving left or right depending on whether the new key is less or greater than the key at the vertex to which it is currently being compared.

Binary Search Trees (5/6)

Algorithm 10.5.1 Building a Binary Search Tree

Input: A totally ordered, nonempty set K of keys

Algorithm Body:

Initialize T to have one vertex, the root, and no edges. Choose a key from K to insert into the root.

while (there are still keys to be added)
Choose a key, *newkey*, from K to add. Let the root be called v, let $key(v)$ be the key at the root, and let $success = 0$.
while ($success = 0$)
if ($newkey < key(v)$)
then if (v has a left child), call the left child v_L and let $v := v_L$
else do 1. add a vertex v_L to T as the left child for v
2. add an edge to T to join v to v_L
3. insert *newkey* as the key for v_L
4. let $success := 1$ **end do**

Binary Search Trees (6/6)

continued

Algorithm 10.5.1 Building a Binary Search Tree

if ($newkey > key(v)$)
then if v has a right child
then call the right child v_R, and let $v := v_R$
else do 1. add a vertex v_R to T as the right child for v
2. add an edge to T to join v to v_R
3. insert *newkey* as the key for v_R
4. let *success* := 1 **end do**
end while
end while
Output: A binary search tree T for the set K of keys

Example 10.5.5 – *Steps for Building a Binary Search Tree*

Go through the steps to build a binary search tree for the keys 15, 10, 19, 25, 12, 4, and insert the keys in the order in which they are listed. For simplicity, use the same names for the vertices and their associated keys.

Example 10.5.5 – *Solution (1/2)*

Insert 15: Make 15 the root.

Insert 10: Compare 10 to 15.

Since 10 < 15 and 15 does not have a left child, make 10 the left child of 15 and add an edge joining 15 and 10.

Insert 19: Compare 19 to 15.

Since 19 > 15 and 15 does not have a right child, make 19 the right child of 15 and add an edge joining 15 and 19.

Example 10.5.5 – *Solution (2/3)* continued

Insert 25: Compare 25 to 15.

Since 25 > 15 and 15 has a right child, namely 19, compare 25 to 19.

Since 25 > 19 and 19 does not have a right child, make 25 the right child of 19 and add an edge joining 19 and 25.

Insert 12: Compare 12 to 15.

Since 12 < 15 and 15 has a left child, namely 10, compare 12 to 10.

Since 12 > 10 and 10 does not have a right child, make 12 the right child of 10 and add an edge joining 10 and 12.

Example 10.5.5 – *Solution (3/3)* continued

Insert 4: Compare 4 to 15.

Since 4 < 15 and 15 has a left child, namely 10, compare 4 to 10.

Since 4 < 10 and 10 does not have a left child, make 4 the left child of 10 and add an edge joining 10 and 4.

The sequence of steps is shown in the following diagrams.