8.2 Rooted Trees & Binary Search Trees-Reading

Notebook:	Discrete Mathematics [CM1020]		
Created:	2019-10-07 2:31 PM	**Updated:**	2020-01-03 6:36 PM
Author:	SUKHJIT MANN		

Cornell Notes	**Topic:** 8.2 Rooted Trees & Binary Search Trees-Reading	Course: BSc Computer Science
		Class: Discrete Mathematics-Reading
		Date: January 03, 2020

Essential Question:

What are rooted trees & binary search trees?

Questions/Cues:

- What is a rooted tree?
- What is some terminology used when describing trees?
- What is a m-ary tree?
- What are ordered rooted trees?
- What are some properties of trees?
- What are balanced m-ary trees?
- What are some properties of m-ary trees?
- What is Binary search tree and respective algorithm?

Notes

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.

The terminology for trees has botanical and genealogical origins. 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 (the reader should show that such a vertex is unique). When u is the parent of v, v is called a **child** of u. Vertices with the same parent are called **siblings**. 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 (that is, its parent, its parent's parent, and so on, until the root is reached). The **descendants** of a vertex v are those vertices that have v as an ancestor. A vertex of a rooted tree is called a **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 graph, in which case it is a leaf.

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.

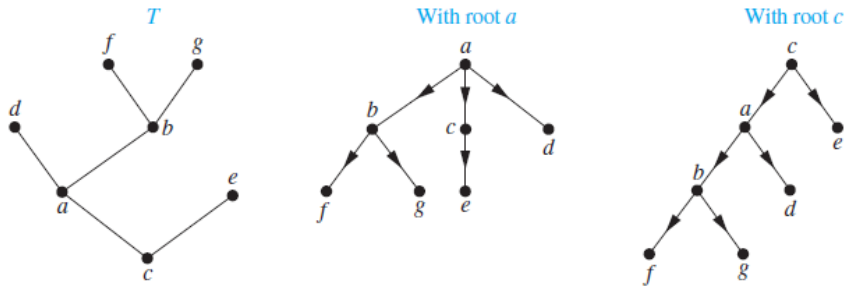

FIGURE 4 **A Tree and Rooted Trees Formed by Designating Two Different Roots.**

EXAMPLE 2 In the rooted tree T (with root a) shown in Figure 5, find the parent of c, the children of g, the siblings of h, all ancestors of e, all descendants of b, all internal vertices, and all leaves. What is the subtree rooted at g?

Extra Examples

Solution: The parent of c is b. The children of g are h, i, and j. The siblings of h are i and j. The ancestors of e are c, b, and a. The descendants of b are c, d, and e. The internal vertices are a, b, c, g, h, and j. The leaves are d, e, f, i, k, l, and m. The subtree rooted at g is shown in Figure 6. ◀

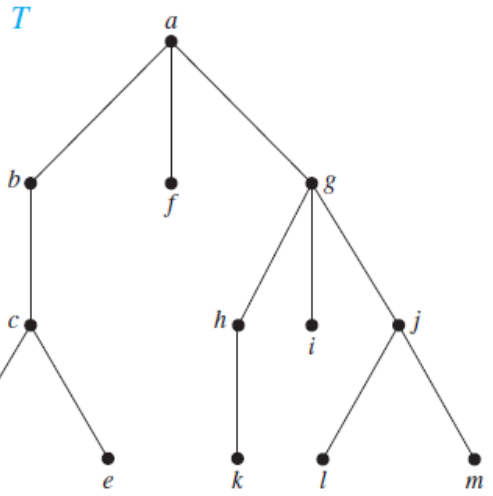

FIGURE 5 **A Rooted Tree T.**

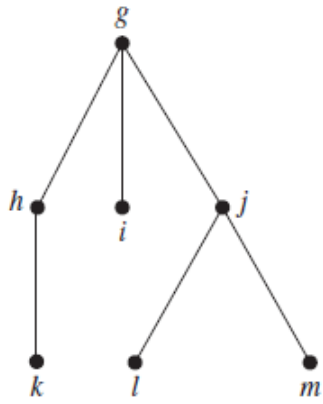

FIGURE 6 **The Subtree Rooted at g.**

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*.

EXAMPLE 3 Are the rooted trees in Figure 7 full m-ary trees for some positive integer m?

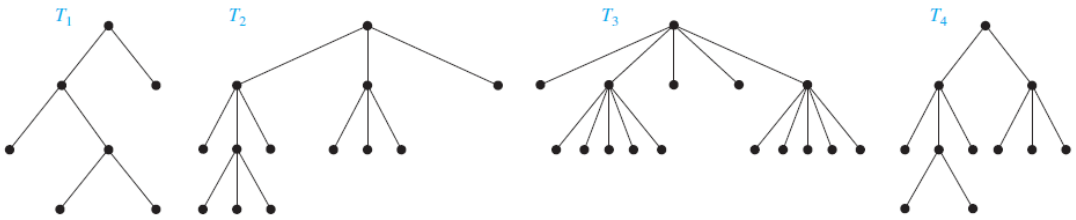

FIGURE 7 **Four Rooted Trees.**

Solution: T_1 is a full binary tree because each of its internal vertices has two children. T_2 is a full 3-ary tree because each of its internal vertices has three children. In T_3 each internal vertex has five children, so T_3 is a full 5-ary tree. T_4 is not a full m-ary tree for any m because some of its internal vertices have two children and others have three children. ◀

ORDERED ROOTED TREES An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right. Note that a representation of a rooted tree in the conventional way determines an ordering for its edges. We will use such orderings of edges in drawings without explicitly mentioning that we are considering a rooted tree to be ordered.

In an ordered binary tree (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child**. The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex. The reader should note that for some applications every vertex of a binary tree, other than the root, is designated as a right or a left child of its parent. This is done even when some vertices have only one child. We will make such designations whenever it is necessary, but not otherwise.

Ordered rooted trees can be defined recursively. Binary trees, a type of ordered rooted trees, were defined this way in Section 5.3.

EXAMPLE 4 What are the left and right children of d in the binary tree T shown in Figure 8(a) (where the order is that implied by the drawing)? What are the left and right subtrees of c?

Solution: The left child of d is f and the right child is g. We show the left and right subtrees of c in Figures 8(b) and 8(c), respectively. ◀

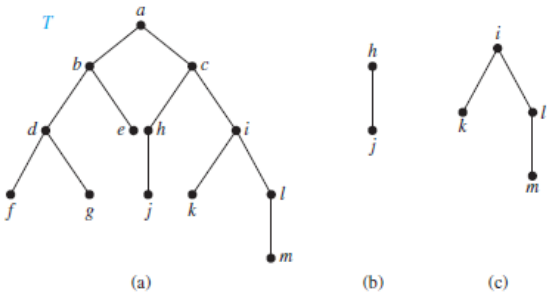

FIGURE 8 **A Binary Tree T and Left and Right Subtrees of the Vertex c.**

THEOREM 2 A tree with n vertices has $n - 1$ edges.

THEOREM 3 A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices.

THEOREM 4 A full m-ary tree with

(*i*) n vertices has $i = (n - 1)/m$ internal vertices and $l = [(m - 1)n + 1]/m$ leaves,
(*ii*) i internal vertices has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves,
(*iii*) l leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.

EXAMPLE 9 Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to four other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives more than one letter 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?

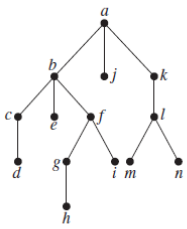

FIGURE 13 **A Rooted Tree.**

Solution: The chain letter can be represented using a 4-ary tree. The internal vertices correspond to people who sent out the letter, and the leaves correspond to people who did not send it out. Because 100 people did not send out the letter, the number of leaves in this rooted tree is $l = 100$. Hence, part (*iii*) of Theorem 4 shows that the number of people who have seen the letter is $n = (4 \cdot 100 - 1)/(4 - 1) = 133$. Also, the number of internal vertices is $133 - 100 = 33$, so 33 people sent out the letter. ◀

BALANCED m-ARY TREES It is often desirable to use rooted trees that are "balanced" so that the subtrees at each vertex contain paths of approximately the same length. Some definitions will make this concept clear. The **level** of a vertex v in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. 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.

EXAMPLE 10 Find the level of each vertex in the rooted tree shown in Figure 13. What is the height of this tree?

Solution: The root a is at level 0. Vertices b, j, and k are at level 1. Vertices c, e, f, and l are at level 2. Vertices d, g, i, m, and n are at level 3. Finally, vertex h is at level 4. Because the largest level of any vertex is 4, this tree has height 4. ◀

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

EXAMPLE 11 Which of the rooted trees shown in Figure 14 are balanced?

Solution: T_1 is balanced, because all its leaves are at levels 3 and 4. However, T_2 is not balanced, because it has leaves at levels 2, 3, and 4. Finally, T_3 is balanced, because all its leaves are at level 3. ◀

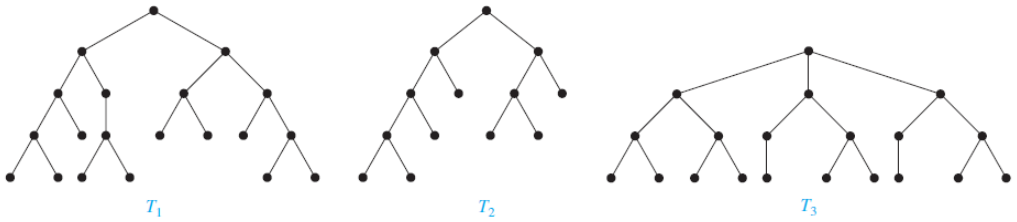

FIGURE 14 **Some Rooted Trees.**

A BOUND FOR THE NUMBER OF LEAVES IN AN m-ARY TREE It is often useful to have an upper bound for the number of leaves in an m-ary tree. Theorem 5 provides such a bound in terms of the height of the m-ary tree.

THEOREM 5 There are at most m^h leaves in an m-ary tree of height h.

COROLLARY 1 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 $h = \lceil \log_m l \rceil$. (We are using the ceiling function here. Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x.)

Binary Search Trees

Searching for items in a list is one of the most important tasks that arises in computer science. Our primary goal is to implement a searching algorithm that finds items efficiently when the items are totally ordered. This can be accomplished through the use of a **binary search tree**, which is a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child, and each vertex is labeled with a key, which is one of the items. Furthermore, vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

This recursive procedure is used to form the binary search tree for a list of items. Start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child, or moving to the right if the item is greater than the key of the respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child. We illustrate this procedure with Example 1.

EXAMPLE 1 Form a binary search tree for the words *mathematics, physics, geography, zoology, meteorology, geology, psychology*, and *chemistry* (using alphabetical order).

Solution: Figure 1 displays the steps used to construct this binary search tree. The word *mathematics* is the key of the root. Because *physics* comes after *mathematics* (in alphabetical order), add a right child of the root with key *physics*. Because *geography* comes before *mathematics*, add a left child of the root with key *geography*. Next, add a right child of the vertex with key *physics*, and assign it the key *zoology*, because *zoology* comes after *mathematics* and after *physics*. Similarly, add a left child of the vertex with key *physics* and assign this new vertex the key *meteorology*. Add a right child of the vertex with key *geography* and assign this new vertex the key *geology*. Add a left child of the vertex with key *zoology* and assign it the key *psychology*. Add a left child of the vertex with key *geography* and assign it the key *chemistry*. (The reader should work through all the comparisons needed at each step.) ◀

Once we have a binary search tree, we need a way to locate items in the binary search tree, as well as a way to add new items. Algorithm 1, an insertion algorithm, actually does both of these tasks, even though it may appear that it is only designed to add vertices to a binary search tree. That is, Algorithm 1 is a procedure that locates an item x in a binary search tree if it is present, and adds a new vertex with x as its key if x is not present. In the pseudocode, v is the vertex currently under examination and $label(v)$ represents the key of this vertex. The algorithm begins by examining the root. If x equals the key of v, then the algorithm has found the location of x and terminates; if x is less than the key of v, we move to the left child of v and repeat the procedure; and if x is greater than the key of v, we move to the right child of v and repeat the procedure. If at any step we attempt to move to a child that is not present, we know that x is not present in the tree, and we add a new vertex as this child with x as its key.

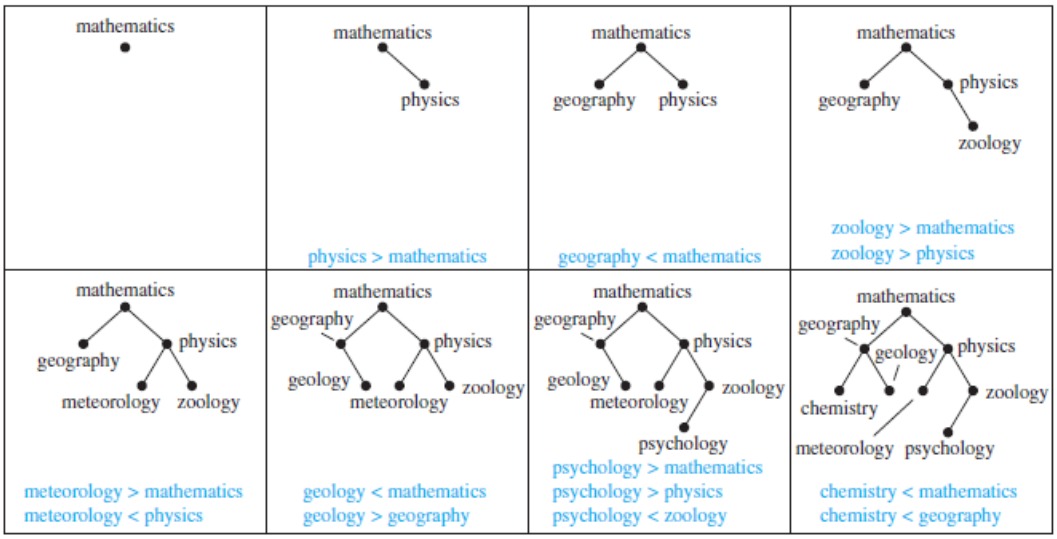

FIGURE 1 **Constructing a Binary Search Tree.**

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}

Summary

In this week, we learned what rooted tree is, special rooted trees, properties & terminology associated with rooted trees, what m-ary trees are & what a binary search tree is.