CMSC 420: Lecture 3
Rooted Trees and Binary Trees

Tree Definition and Notation: Trees and their variants are among the most fundamental data structures. A tree is a special class of graph. Recall from your previous courses that a *graph* $G = (V, E)$ consists of a finite set of *vertices* (or *nodes*) V and a finite set of edges E. Each *edge* is a pair of nodes. In an *undirected graph* (or simply "graph"), the edge pairs are unordered, and in a *directed graph* (or "digraph"), the edge pairs are ordered. An undirected graph is *connected* if there is path between any pair of nodes.

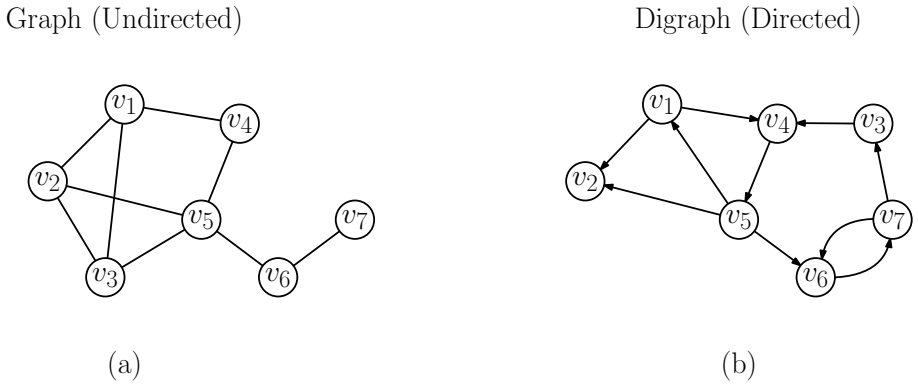


Fig. 1: Graphs: (a) undirected (b) directed.

The most general form of a tree, called a *free tree*, is simply a connected, undirected graph that has no cycles (see Fig. 2(a)). An example of a free tree is the minimum cost spanning tree (MST) of a graph.

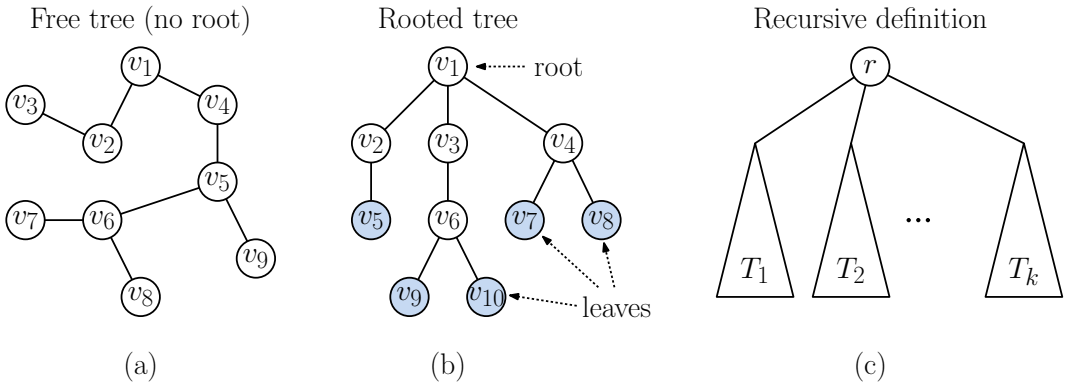


Fig. 2: Trees: (a) free tree, (b) rooted tree, (c) recursive definition.

Since we will want to use trees for applications in searching, it will be useful to assign some sense of order and direction to our trees. This will be done by designating a special node, called the *root*. In the same manner as a family tree, we think of the root as the *ancestor* of all the other nodes in the tree, or equivalently, the other nodes are *descendants* of the root. Nodes that have no descendants are called *leaves* (see Fig. 2(b)). All the others are called *internal nodes*. Each node of the tree can be viewed as the root of a *subtree*, consisting of this node an all of its descendants. Here is a formal (recursive) definition of a rooted tree:

Rooted tree: Is either:

- A single node, or
- A collection of one more more rooted trees $\{T_1, \ldots, T_k\}$ joined under a common root node (see Fig. 2(c)).

Since we will be dealing with rooted trees almost exclusively for the rest of the semester, when we say "tree" we will mean "rooted tree." We will use the term "free tree" otherwise.

Terminology: There is a lot of notation involving trees. Most terms are easily understood from the family-tree analogy. Every non-root node is descended from its *parent*, and the directed descendants of any node are called its *children* (see Fig. 3(a)). Two nodes that share a common parent are called *siblings*, and other relations (e.g., grandparent, grandchild) follow analogously.

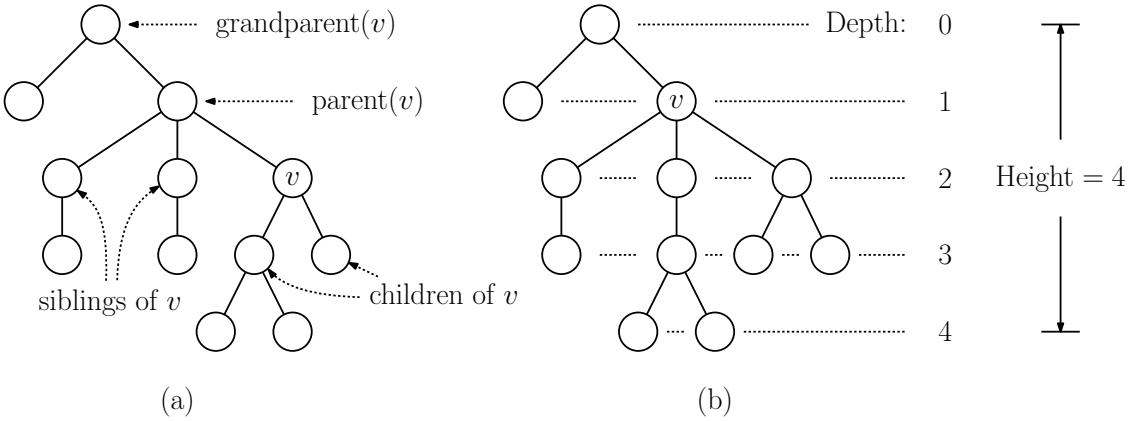


Fig. 3: (a) "Family" relations, (b) depth and height.

The *depth* of a node in the tree is the length (number of edges) of the (unique) path from the root to that node. Thus, the root is at depth 0. The *height* of a tree is the maximum depth of any of its nodes (see Fig. 1(b)). For example, the tree of Fig. 2(b) is of depth 3, as evidenced by nodes v_9 and v_{10}, which are at this depth. As we defined it, there is no special ordering among the children of a node. When the ordering among a node's is significant, it is called an *ordered tree*.

Representing Rooted Trees: Rooted trees arise in many applications in which hierarchies exist. Examples include computer file systems, hierarchically-based organizations (e.g., military and corporate), documents and reports (volume → chapter → section . . . paragraph). There are a number of ways of representing rooted trees. Later we will discuss specialized representations that are tailored for special classes of trees (e.g., binary search trees), but for now let's consider how to represent a "generic" rooted tree.

The idea is to think of each node of the tree as containing a pointer to the head of a linked list of its children, called its `firstChild`. Since a node may generally have siblings, in addition it will have a pointer to its next sibling, called nextSibling. Finally, each node will contain a *data* element (which we will also refer to as its *entry*), which will depend on the application involved. This is illustrated in Fig. 4(a). Fig. 4(b) illustrates how the tree in Fig. 1(b) would be represented using this technique.

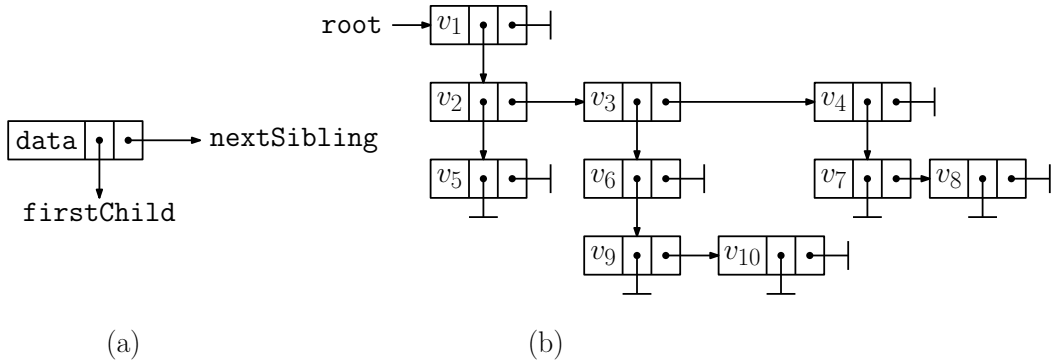


Fig. 4: The "binary" representation of a rooted tree from Fig. 2(b).

Because each node stores two pointers (references), we will often refer to this as the *binary representation* of a rooted tree. Note that this is minimal representation. In practice, we may wish to add additional information. (For example, we may also wish to include a reference to each node's parent.)

Binary Trees: Among rooted trees, by far the most popular in the context of data structures is the *binary tree*. A *binary tree* is a rooted, ordered tree in which every non-leaf node has two children, called *left* and *right* (see Fig. 5(a)). We allow for a binary tree to empty. (We will see that, like the empty string, it is convenient to allow for empty trees.)

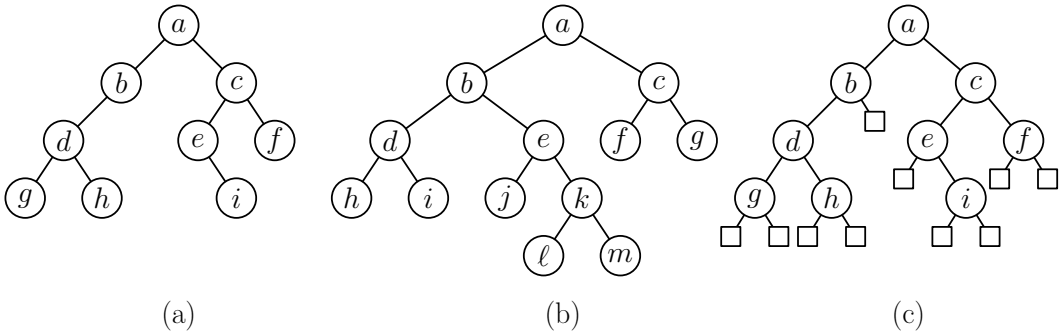


Fig. 5: Binary trees: (a) standard definition, (b) full binary tree, (c) extended binary tree.

Binary trees can be defined more formally as follows. First, an empty tree is a binary tree. Second, if T_L and T_R are two binary trees (possibly empty) then the structure formed by making T_L and T_R the left and right children of a node is also a binary tree. T_L and T_R are called the *subtrees* of the root. If both children are empty, then the resulting node is a *leaf*. Note that, unlike standard rooted trees, there is a difference between a node that has just one child on its left side as opposed to a node that has just one child on its right side. All the definitions from rooted trees (parent, sibling, depth, height) apply as well to binary trees.

Allowing for empty subtrees can make coding tricky. In some cases, we would like to forbid such binary trees. We say that a binary tree is *full* if every node has either zero children (a leaf) or exactly two (an internal node). An example is shown in Fig. 5(b).

Another approach to dealing with empty subtrees is through a process called *extension*. This is most easily understood in the context of the tree shown in Fig. 5(a). We *extend* the tree by adding a special *external node* to replace all the empty subtrees at the bottom of the tree.

The result is called a *extended tree*. (In Fig. 5(c) the external nodes are shown has squares.) This has the effect of converting an arbitrary binary tree to a full binary tree.

Java Representation: The typical Java representation of a tree as a data structure is given below. The `data` field contains the data for the node and is of some generic entry type `E`. The `left` field is a pointer to the left child (or `null` if this tree is empty) and the `right` field is analogous for the right child.

Binary Tree Node

```
class BTNode<E> {
    E           entry;          // this node's data
    BTNode<E>   left;           // left child
    BTNode<E>   right;          // right child
    // ... remaining details omitted
}
```

As with our rooted-tree representation, this is a minimal representation. Perhaps the most useful augmentation would be a parent link.

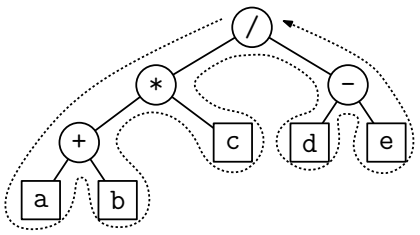
Binary trees come up in many applications. One that we will see a lot of this semester is for representing ordered sets of objects, a *binary search tree*. Another is an *expression tree*, which is used in compiler design in representing a parsed arithmetic exception (see Fig. 6).

Traversals: There are a number of natural ways of visiting or *enumerating* every node of a tree. For rooted trees, the three best known are *preorder*, *postorder*, and (for binary trees) *inorder*. Let T be a tree whose root is r and whose subtrees are $T_1, \ldots, T_k$ for $k \geq 0$. They are all most naturally defined recursively. (Fig. 6 illustrates these in the context of an *expression tree*.)

Preorder: Visit the root r, then recursively do a preorder traversal of $T_1, \ldots, T_k$.

Postorder: Recursively do a postorder traversal of $T_1, \ldots, T_k$ and then visit r. (Note that this is *not* the same as reversing the preorder traversal.)

Inorder: (for binary trees) Do an inorder traversal of T_L, visit r, do an inorder traversal of T_R.



Preorder: `/ * + a b c - d e`

Postorder: `a b + c * d e - /`

Inorder: `a + b * c / d - e`

Fig. 6: Expression tree for $((a + b) * c)/(d - e))$ and common traversals.

These traversals are most easily coded using recursion. The code block below shows a possible way of implementing the preorder traversal in Java. The procedure `visit` would depend on the specific application. The algorithm is quite efficient in that its running time is proportional to the size of the tree. That is, if the tree has n nodes then the running time of these traversal algorithms are all $O(n)$.

Preorder Traversal

```
void preorder(BTNode v)
{
    if (v == null) return;      // empty subtree - do nothing
    visit(v);                   // visit (depends on the application)
    preorder(v.left);           // recursively visit left subtree
    preorder(v.right);          // recursively visit right subtree
}
```

These are not the only ways of traversing a tree. For example, another option would be *breadth-first*, which visits the nodes level by level: "`/ * - + c d e a b`." An interesting question is whether a traversal uniquely determines the tree's shape. The short answer is no, but if you have an extended tree and you know which nodes are internal and which are leaves (as is the case in the expression tree example from Fig. 6), then such a reconstruction is possible. Think about this.

Extended Binary Trees: To motivate our next topic, let's consider the utilization of space in our trees. Recall the binary tree shown in Fig. 5(a). It has nine nodes, and each node has two child links, for a total of 18 links. We have eight actual links in the tree, which means that the remaining 10 links are `null`. Thus, nearly half of the child links are `null`! This is not unusual, as the following theorem states. (Take note the proof structure, since it is common for proofs involving binary trees.)

Claim: A binary tree with n nodes has $n+1$ `null` child links.

Proof: (by induction on the size of the tree) Let $x(n)$ denote the number of `null` child links in a binary tree of n nodes. We want to show that for all $n \geq 0$, $x(n) = n+1$. We'll make the convention of defnining $x(0) = 1$. (If you like, you can think of the `null` pointer to the root node as this link.)

For the basis case, $n = 0$, by our convention $x(0) = 1$, which satisfies the desired relation.

For the induction step, let's assume that $n \geq 1$. The induction hypothesis states that, for all $n' < n$, $x(n') = n' + 1$. A binary tree with at least one node consists of a root node and two (possibly empty) subtrees, T_L and T_R. Let n_L and n_R denote the numbers of nodes in the left and right subtrees, respectively. Together with the root, these must sum to n, so we have $n = 1+n_L+n_R$. By the induction hypothesis, the numbers of `null` links in the left and right subtrees are $x(n_L) = n_L + 1$ and $x(n_R) = n_R + 1$. Together, these constitute all the `null` links. Thus, the total number of `null` links is

$$x(n) \;=\; x(n_L) + x(n_R) \;=\; (n_L + 1) + (n_R + 1) \;=\; (1 + n_L + n_R) + 1 \;=\; n + 1,$$

as desired.

In large data structures, these `null` pointers represent quite a significant wastage of space. What can we do about this? One idea is to distinguish two different node types, one for internal nodes, which have (non-`null`) child links and another for leaves, which need no child links. One way to construct such a tree is called "extension." An *extended binary tree* is constructed by replacing each `null` link with a special leaf node, called an *external node*. (The other nodes are called *internal nodes*.) An example of an extended binary tree is shown in Fig. 5(c). Because we replace each `null` link with a leaf node, we have the following direct corollary from our previous theorem.

Corollary: An extended binary tree with n internal nodes has $n + 1$ external nodes, and hence $2n + 1$ nodes altogether.

The key "take-away" from this proof is that over half of the nodes in an extended binary tree are leaf nodes. In fact, it is generally true that if the degree of a tree is two or greater, leaves constitute the majority of the nodes.

Threaded Binary Trees: We have seen that extended binary trees provide one way to deal with the wasted space caused by `null` pointers in the nodes of a binary tree. In this section we will consider another rather cute use of these pointers.

Recall that binary tree traversals are naturally defined recursively. Therefore a straightforward implementation would require extra space to store the stack for the recursion. Is some way to traverse the tree without this additional storage? The answer is yes, and the trick is to employ each `null` pointer encode some additional information to aid in the traversal. Each left-child `null` pointer stores a reference to the node's inorder predecessor, and each right-child `null` pointer stores a reference to the node's inorder successor. The resulting representation is called a *threaded binary tree*. (For example, in Fig. 7(a), we show a threaded version of the tree in Fig. 5(b)).

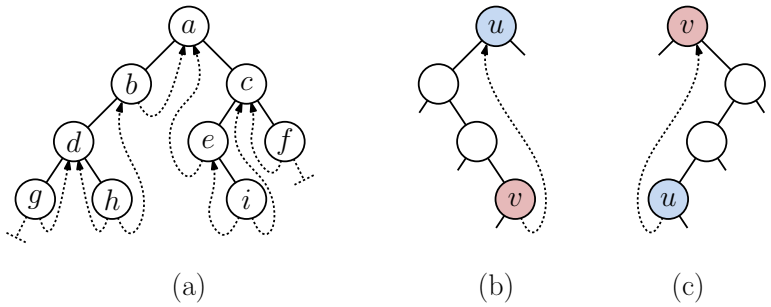


Fig. 7: A Threaded Tree.

We also need to add a special "mark bit" to each child link, which indicates whether the link is a thread or a standard parent-child link. The node structure is shown below.

Threaded Binary Tree Node

```
class ThreadBTNode<E> {
    E                entry;         // this node's data
    Boolean          leftIsThread;  // left child link is a thread
    Boolean          rightIsThread; // right child link is a thread
    ThreadBTNode<E>  left;          // left child
    ThreadBTNode<E>  right;         // right child
    // ... remaining details omitted
}
```

Let us consider how to do an inorder traversal in a threaded-tree representation. Suppose that we are currently visiting a node u. How do we find the inorder successor of u? First, if u's right-child link is a thread, then we just follow it (see Fig. 7(b)). Otherwise, we go the node's right child, and then traverse left-child links until reaching the bottom of the tree, that is a threaded link (see Fig. 7(c)).

Inorder Successor in a Threaded Tree

```
ThreadBTNode inorderSuccessor(ThreadBTNode v) {
    ThreadBTNode u = v.right;           // go to right child
    if (v.rightIsThread) return u;      // if thread, then done
    while (!u.leftIsThread) {           // else u is right child
        u = u.left;                     // go to left child
    }                                   // ...until hitting thread
    return u;
}
```

For example, in Fig. 7(b), if we start at d, the thread takes us directly to a, which is d's inorder successor. In Fig. 7(c), if we start at a, then we follow the right-child link to b, and then follow left-links until arriving at d, which is the inorder successor. Of course, to turn this into a complete traversal function, we would need to start by finding the first inorder node in the tree. We'll leave this as an exercise.

Threading is more of a "cute trick" than a common implementation technique with binary trees. Nonetheless, it is representative of the number of clever ideas that have been developed over the years for representing and processing binary trees.

Complete Binary Trees: We have discussed linked allocation strategies for rooted and binary trees. Is it possible to allocate trees using sequential (that is, array) allocation? In general it is not possible because of the somewhat unpredictable structure of trees (unless you are willing to waste a lot of space). However, there is a very important case where sequential allocation is possible.

Complete Binary Tree: Every level of the tree is completely filled, except possibly the bottom level, which is filled from left to right.

It is easy to verify that a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes, implying that a tree with n nodes has height $O(\log n)$ (see Fig. 8). (We leave these as exercises involving geometric series.)

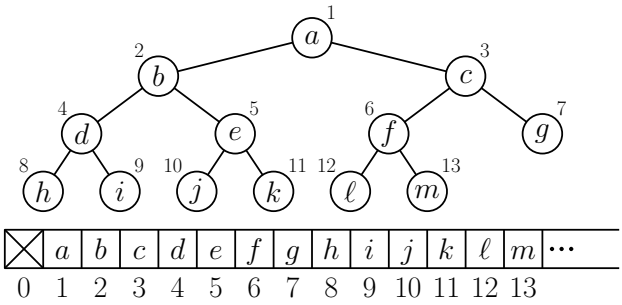


Fig. 8: A complete binary tree.

The extreme regularity of complete binary trees allows them to be stored in arrays, so no additional space is wasted on pointers. Consider an indexing of nodes of a complete tree from 1 to n in increasing level order (so that the root is numbered 1 and the last leaf is numbered n). Observe that there is a simple mathematical relationship between the index of a node and the indices of its children and parents. In particular:

leftChild(i)**:** if $(2i \leq n)$ then $2i$, else `null`.

rightChild(i)**:** if $(2i + 1 \leq n)$ then $2i + 1$, else `null`.

parent(i)**:** if $(i \geq 2)$ then $\lfloor i/2 \rfloor$, else `null`.

As an exercise, see if you can also compute the sibling of node i and the depth of node i.

Observe that the last leaf in the tree is at position n, so adding a new leaf simply means inserting a value at position $n+1$ in the list and updating n. Since arrays in Java are indexed from 0, omitting the 0th entry of the matrix is a bit of wastage. Of course, the above rules can be adapted to work even if we start indexing at zero, but they are not quite so simple.