Lecture 5: Trees

Read: Chapt. 4 in Weiss.

Trees: Trees and their variants are among the most common data structures. In its most general form, a *free tree* is a connected, undirected graph that has no cycles. Since we will want to use our trees for applications in searching, it will be more meaningful to assign some sense of order and direction to our trees.

Formally a *tree* (actually a *rooted tree*) is defined recursively as follows. It consists of one or more items called *nodes*. (Our textbook allows for the possibility of an empty tree with no nodes.) It consists of a distinguished node called the *root*, and a set of zero or more nonempty subsets of nodes, denoted $T_1, T_2, \ldots, T_k$, where each is itself a tree. These are called the *subtrees* of the root.

The root of each subtree $T_1, \ldots, T_k$ is said to be a *child* of r, and r is the *parent* of each root. The children of r are said to be *siblings* of one another. Trees are typically drawn like graphs, where there is an edge (sometimes directed) from a node to each of its children. See the figure below.

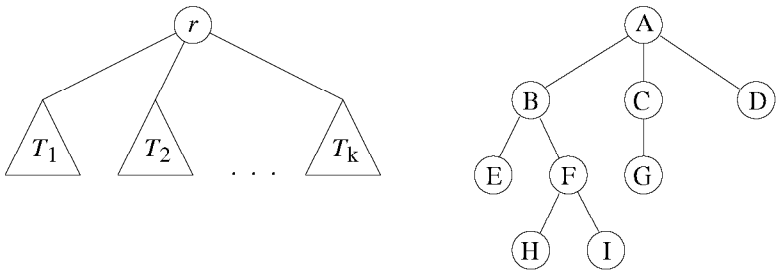

Figure 7: Trees.

If there is an order among the T_i's, then we say that the tree is an *ordered tree*. The *degree* of a node in a tree is the number of children it has. A *leaf* is a node of degree 0. A *path* between two nodes is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that u_i is a parent of u_{i+1}. The *length* of a path is the number of edges on the path (in this case $k - 1$). There is a path of length 0 from every node to itself.

The *depth* of a node in the tree is the length of the unique path from the root to that node. The root is at depth 0. The *height* of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0. If there is a path from u to v we say that v is a *descendants* of u. We say it is a *proper descendent* if $u \neq v$. Similarly, u is an *ancestor* of v.

Implementation of Trees: One difficulty with representing general trees is that since there is no bound on the number of children a node can have, there is no obvious bound on the size of a given node (assuming each node must store pointers to all its children). The more common representation of general trees is to store two pointers with each node: the `firstChild` and the `nextSibling`. The figure below illustrates how the above tree would be represented using this technique.

Trees arise in many applications in which hierarchies exist. Examples include the Unix file system, corporate managerial structures, and anything that can be described in "outline form" (like the chapters, sections, and subsections of a user's manual). One special case of trees will be very important for our purposes, and that is the notion of a binary tree.

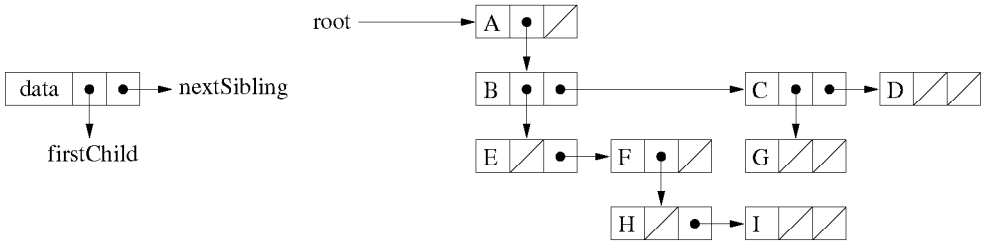

Figure 8: A binary representation of general trees.

Binary Trees: Our text defines a binary tree as a tree in which each node has no more than two children. However, this definition is subtly flawed. A *binary tree* is defined recursively as follows. A binary tree can be empty. Otherwise, a binary tree consists of a root node and two disjoint binary trees, called the *left* and *right* subtrees. The difference in the two definitions is important. There is a distinction between a tree with a single left child, and one with a single right child (whereas in our normal definition of tree we would not make any distinction between the two).

The typical Java representation of a tree as a data structure is given below. The `element` field contains the data for the node and is of some abstract type, which in Java might be `Object`. (In C++ the element type could be specified using a template.) When we need to be concrete we will often assume that element fields are just integers. 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.

```
class BinaryTreeNode {
    Object          element;        // data item
    BinaryTreeNode  left;           // left child
    BinaryTreeNode  right;          // right child
    ...
}
```

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 one that is used often in compiler design is *expression trees* which are used as an intermediate representation for expressions when a compiler is parsing a statement of some programming language. For example, in the figure below right, we show an expression tree for the expression $((a + b) * c)/(d - e))$.

Traversals: There are three natural ways of visiting or *traversing* every node of a tree, *preorder*, *postorder*, and (for binary trees) *inorder*. Let T be a tree whose root is r and whose subtrees are $T_1, T_2, \ldots, T_m$ for $m \geq 0$.

Preorder: Visit the root r, then recursively do a preorder traversal of $T_1, T_2, \ldots, T_k$. For example: $\langle /, *, +, a, b, c, -, d, e \rangle$ for the expression tree shown above.

Postorder: Recursively do a postorder traversal of $T_1, T_2, \ldots, T_k$ and then visit r. Example: $\langle a, b, +, c, *, d, e, -, / \rangle$. (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. Example: $\langle a, +, b, *, c, /, d, -, e \rangle$.

Note that theses traversal correspond to the familiar *prefix*, *postfix*, and *infix* notations for arithmetic expressions.

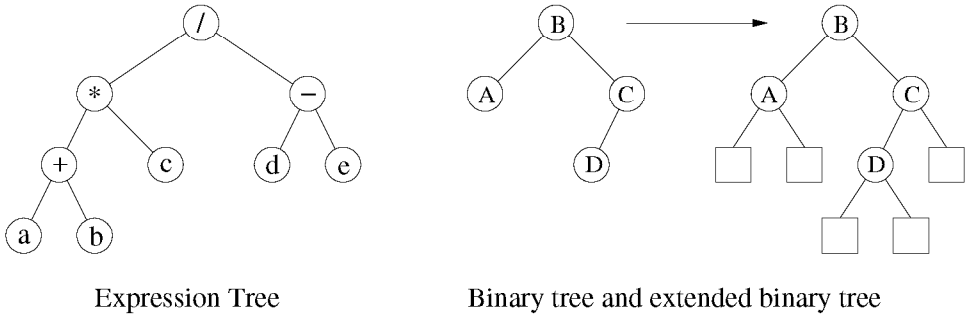

Figure 9: Expression tree.

Preorder arises in game-tree applications in AI, where one is searching a tree of possible strategies by *depth-first search*. Postorder arises naturally in code generation in compilers. Inorder arises naturally in binary search trees which we will see more of.

These traversals are most easily coded using recursion. If recursion is not desired (either for greater efficiency or for fear of using excessive system stack space) it is possible to use your own stack to implement the traversal. Either way 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 $O(n)$.

Extended Binary Trees: Binary trees are often used in search applications, where the tree is searched by starting at the root and then proceeding either to the left or right child, depending on some condition. In some instances the search stops at a node in the tree. However, in some cases it attempts to cross a null link to a nonexistent child. The search is said to "fall out" of the tree. The problem with falling out of a tree is that you have no information of where this happened, and often this knowledge is useful information. Given any binary tree, an *extended binary tree* is one which is formed by replacing each missing child (a null pointer) with a special leaf node, called an *external node*. The remaining nodes are called *internal nodes*. An example is shown in the figure above right, where the external nodes are shown as squares and internal nodes are shown as circles. Note that if the original tree is empty, the extended tree consists of a single external node. Also observe that each internal node has exactly two children and each external node has no children.

Let n denote the number of internal nodes in an extended binary tree. Can we predict how many external nodes there will be? It is a bit surprising but the answer is yes, and in fact the number of extended nodes is $n + 1$. The proof is by induction. This sort of induction is so common on binary trees, that it is worth going through this simple proof to see how such proofs work in general.

Claim: An extended binary tree with n internal nodes has $n + 1$ external nodes.

Proof: By (strong) induction on the size of the tree. Let $X(n)$ denote the number of external nodes in a binary tree of n nodes. We want to show that for all $n \geq 0$, $X(n) = n + 1$.

The basis case is for a binary tree with 0 nodes. In this case the extended tree consists of a single external node, and hence $X(0) = 1$.

Now let us consider the case of $n \geq 1$. By the induction hypothesis, for all $0 \leq n' < n$, we have $X(n') = n' + 1$. We want to show that it is true for n. Since $n \geq 1$ the tree contains a root node. Among the remaining $n - 1$ nodes, some number k are in the left subtree, and the other $(n - 1) - k$ are in the right subtree. Note that k and $(n - 1) - k$ are both

less than n and so we may apply the induction hypothesis. Thus, there are $X(k) = k + 1$ external nodes in the left subtree and $X((n-1)-k) = n-k$ external nodes in the right subtree. Summing these we have

$$X(n) = X(k) + X((n-1)-k) = (k+1) + (n-k) = n+1,$$

which is what we wanted to show.

Remember this general proof structure. When asked to prove any theorem on binary trees by induction, the same general structure applies.

Complete Binary Trees: We have discussed linked allocation strategies for general trees 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. However, there is a very important case where sequential allocation is possible.

Complete Binary Tree: is a binary tree in which 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 $O(\log n)$. (We leave these as exercises involving geometric series.) An example is provided in the figure below.

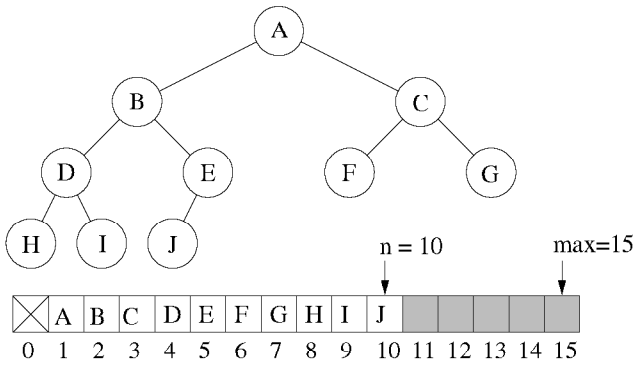

Figure 10: A complete binary tree.

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.

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.

Threaded Binary Trees: Note that binary tree traversals are defined recursively. Therefore a straightforward implementation would require extra space to store the stack for the recursion. The stack will save the contents of all the ancestors of the current node, and hence the additional space required is proportional to the height of the tree. (Either you do it explicitly or the system handles it for you.) When trees are balanced (meaning that a tree with n nodes has $O(\log n)$ height) this is not a big issue, because $\log n$ is so smaller compared to n. However, with arbitrary trees, the height of the tree can be as high as $n - 1$. Thus the required stack space can be considerable.

This raises the question of whether there is some way to traverse a tree without using additional storage. There are two tricks for doing this. The first one involves altering the links in the tree as we do the traversal. When we descend from parent to child, we reverse the parent-child link to point from parent to the grandparent. These reversed links provide a way to back up the tree when the recursion bottoms out. On backing out of the tree, we "unreverse" the links, thus restoring the original tree structure. We will leave the details as an exercise.

The second method involves a clever idea of using the space occupied for the null pointers to store information to aid in the traversal. In particular, each left-child pointer that would normally be null is set to the inorder predecessor of this node. Similarly each right-child pointer that would normally be null is set to the inorder successor of this node. The resulting links are called *threads*. This is illustrated in the figure below (where threads are shown as broken curves).

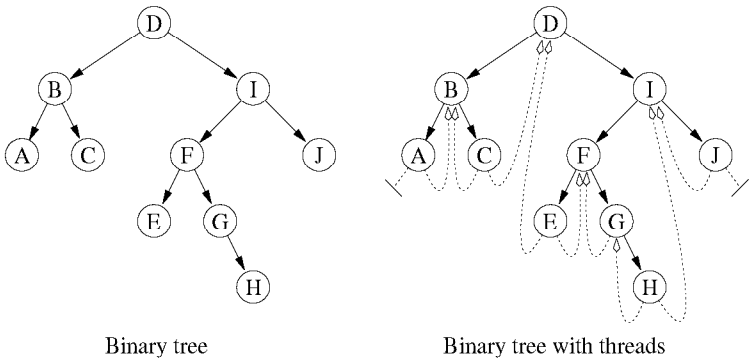

Figure 11: A Threaded Tree.

Each such pointer needs to have a special "mark bit" to indicate whether it used as a parent-child link or as a thread. So the additional cost is only two bits per node. Now, suppose that you are currently visiting a node u. How do we get to the inorder successor of u? If the right child pointer is a thread, then we just follow it. Otherwise, we go the right child, and then traverse left-child links until reaching the bottom of the tree (namely a threaded link).

```
BinaryTreeNode nextInOrder() {              // inorder successor of "this"
    BinaryTreeNode q = right;               // go to right child
    if (rightIsThread) return q;            // if thread, then done
    while (!q.leftIsThread) {               // else q is right child
        q = q.left;                         // go to left child
    }                                       // ...until hitting thread
    return q;
}
```

For example, in the figure below, suppose that we start with node A in each case. In the left case, we immediately hit a thread, and return B as the result. In the right case, the right pointer is not a thread, so we follow it to B, and then follow left links to C and D. But D's left child pointer is a thread, so we return D as the final successor. Note that the entire process starts at the first node in an inorder traversal, that is, the leftmost node in the tree.

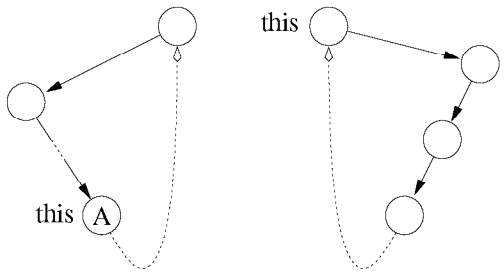

Figure 12: Inorder successor in a threaded tree.