Lecture Summary

In this lecture, we saw more about trees: the nomenclature, how they are represented, and how one traverses them.

1 The Basics

As we briefly previewed last lecture, trees are a special type of graph. We can think of them in two different ways, arriving at the same definition in the end.

- A tree is an undirected, connected graph without cycles.
- A tree is “like a” linked list, with each node having at most one prev pointer (exactly one node having none), and being allowed to have multiple next pointers.

An undirected graph without cycles that may or may not be connected is called a forest. In other words, a forest consists of one or more trees.

In the real world, we can model quite a few things using trees: family trees (either by going from one person to their parents, grandparents, etc., or from one ancestor to his/her children, grandchildren, and so forth), file system hierarchies, evolutionary trees among species (sort of like family trees), parsing of human language or formulas or programming languages, and a few others.

Even more importantly in this class, trees are very useful as the basis of many more complex data structures, and often offer much better tradeoffs between running times of different operations than linear structures like arrays or linked lists. When using trees in the design of data structures, we usually think of the trees as directed. There is a designated root node of the tree, and all edges point away from it.

The terminology for trees is a somewhat inconsistent mix of botany and family trees.

- If node $u$ has a directed edge to node $v$, then we say that $u$ is a parent of $v$, and $v$ is a child of $u$. If $u$ has a directed edge to nodes $v_1, v_2$, then $v_1$ and $v_2$ are siblings.
- The root node is the one node that does not have a parent. Since every other node can be reached from the root, if we have all the edges, it’s enough to have access to the root to find all other nodes.
- Nodes that can be reached from $u$ with a path of length 1 or more are called descendants of $u$. Nodes that can reach $v$ with a path of length 1 or more are called ancestors of $v$.
- A leaf is a node without children. A node with at least one child is called an internal node.
- When talking about rooted trees, the degree of $v$ refers to the number of edges out of a certain node. For example, if a node has one edge from its parent, and two to its children, its degree is 2 (even though if we thought of it as an undirected tree, we would call the degree 3).
- A subtree of a tree is a subset of nodes and edges that are (by themselves) connected. The most frequently used subtrees are those consisting of a node $v$ and all its descendants. Those are be called the subtree rooted at $v$. 
It is frequently useful to think about the levels of nodes. For any node \( v \), the level of \( v \) is the number of nodes in the path from the root to \( v \). Thus, the root itself is at level 1, its children are at level 2, its grandchildren at level 3, and so on. (Some sources may start with level 0 — make sure you use the same notation as whoever you are talking with.)

The height of a tree is the largest level of any of its nodes.

Much like other structures in this class, we can use recursion to say formally what a rooted tree is.

1. A root \( r \) by itself is a rooted tree. (Depending on how we feel about calling an empty tree “rooted”, we could instead use the empty tree as our base case.)

2. If \( T_1, T_2, \ldots, T_k \) are rooted trees, rooted at \( u_1, u_2, \ldots, u_k \), then by adding a node \( r \) and directed edges \((r, u_1), (r, u_2), \ldots, (r, u_k)\), we obtain a rooted tree.

This recursive definition of a tree often aligns very naturally with recursive algorithms for processing data on trees, as we will see soon enough.

There are specific subclasses of rooted trees that are frequently used in designing data structures, so we emphasize them here:

1. A tree is \( d \)-ary if each node has degree exactly 0 or \( d \). A binary tree is a 2-ary tree. We emphasize that we sometimes allow an exception for one node to have only one child, as we will see below.

2. A \( d \)-ary tree is full if all leaves are at the same level (so adding one more node would require a new level to be formed).

3. Being complete is slightly less restrictive than being full. A complete \( d \)-ary tree has all its leaves at level \( h \) or level \( h - 1 \). Usually, when talking about complete trees, we also assume that there is a natural “left-to-right” order among the subtrees, and we require that the leaves at level \( h \) are as far to the left as possible.

Many trees we will use for implementing data structures will be binary, but there will be an important exception of so-called 2-3-trees, in which each internal node has degree either 2 or 3.

## 2 Implementing trees

Trees are special types of graphs, so we could just use our general-purpose graph implementations (adjacency matrix or adjacency list) to implement them. Adjacency matrices are not very natural, for two reasons:

1. We rarely need to find out if two specific nodes are connected by an edge, but we frequently need to explore all children of a given node.

2. Trees are extremely sparse (\( n - 1 \) edges for \( n \) nodes), so adjacency matrices would be extremely wasteful with space.

Adjacency lists are a more natural fit. But we can simplify them a bit, since we know that each node has exactly one incoming edge (except the root, which has 0). So we won’t need a whole list of incoming edges, and can just store one pointer to the parent (which is NULL for the root). This gives us something like the following as the structure for a node.

```cpp
template <class T>
class Node {
    T data;
    Node<T>** parent;
    List<Node<T>**> children;
};
```
Depending on whether the order of the children matters, we may use a `Bag<Node<T>*> children` instead of the `List`. For a parse tree of an arithmetic expression, the order of children clearly matters, as $5 - 2$ is not the same as $2 - 5$. For some other trees, it doesn’t matter as much.

Many of our trees will be binary, in which case we don’t need a list of children, and can instead have two pointers for the children.

```cpp
template <class T>
class Node {
    T data;
    Node<T> *parent;
    Node<T> *leftChild, *rightChild;
};
```

In both cases (binary or more general), notice that we are storing pointers to other nodes, rather than something like `Node<T> parent`. This is important: ideally, we’d like to generate just one object for each node, and then have others point to it. If instead, we stored nodes themselves (rather than pointers), we would be creating a new copy of a node for each neighbor, and following around the structure would be hard.

An alternative to generating dynamic objects for nodes is to store all nodes in one array. Then, instead of storing pointers to the parent and children, we can just store their indices as integers. This would give us the following for a binary tree (and a similar idea for general trees):

```cpp
template <class T>
class Node {
    T data;
    int parent;
    int leftChild, rightChild;
};
```

If the tree is a complete binary tree, this can be simplified even further. If we start numbering the nodes at 1 (for the root node), then by a little bit of experimentation, we find that the parent of node $i$ is always node $\lfloor i/2 \rfloor$, and the two children are $2i$ and $2i + 1$. The formula looks a little different if we number the root with 0, but it’s not much more difficult. It also generalizes to $d$-ary trees, which you will explore on a homework.

### 3 Traversing a tree

Traversing a tree means visiting each node of the tree exactly once. This is naturally done recursively, following the recursive definition for a tree. Traversing a leaf is pretty easy — we just do whatever we want to do with it (such as print it or do some computation). Traversing an internal node $v$ involves visiting the node itself, and recursively visiting each of the subtrees rooted at its children $u_1, u_2, \ldots, u_k$ (for $k \geq 1$). The main distinction is then the relative timing: do we traverse the subtrees first, or do we visit the node $v$ itself first? This is exactly the difference between post-order traversal and pre-order traversal. If the tree is binary, it also makes sense to traverse the left subtree first, then visit the node $v$, and then traverse the right subtree. This order is called in-order traversal. We will illustrate all three with a traversal of the following tree:

In post-order traversal, we visit all the subtrees first, then process the root. Thus, the pseudo-code looks as follows:

```cpp
void traverse (Node<T> * r) {
    for each child u of r
        traverse(u);
    process r's data, e.g., output it or something;
}
```
Figure 1: A tree $T$ on which to illustrate different traversal orders. We would consider $T$ a complete binary tree, even though technically, node 4 has only one child, violating the definition. But the concept is useful enough that we’d like to not worry about this small detail.

If we apply it to our example tree, we get the following sequence: 7 8 3 9 4 1 5 6 2 0.

In pre-order traversal, we process the root first, then visit all of the subtrees. The code is obtained by just swapping the order of the two parts in the post-order traversal pseudo-code.

```c
void traverse (Node<T>* r) {
    process r’s data, e.g., output it or something;
    for each child u of r
        traverse(u);
}
```

For pre-order traversal, the visit order in our tree is: 0 1 3 7 8 4 9 2 5 6.

Finally, for binary trees, there is also in-order traversal, in which we visit all nodes of the left subtree, then the root, and then all nodes of the right subtree. We get the following pseudo-code:

```c
void traverse (Node<T>* r) {
    traverse (r->leftChild);
    process r’s data, e.g., output it or something;
    traverse (r->rightChild);
}
```

Here, the visit order for our example tree is: 7 3 8 1 9 4 0 5 2 6

In-order traversal is most natural for binary trees. If a node has more than 2 children, one would have to make a decision what is the right point to process its own data. If a node has $d$ children and $d - 1$ pieces of data (instead of just one), then it makes sense to alternate between children and pieces of data, and indeed, we will see that when talking about 2-3 trees later.