4.3 Tree Traversals

Stepping, or iterating, through the entries of a linearly ordered list has only two obvious orders: from front to back or from back to front. There is no obvious traversal of a general tree other than we are likely to start at the root.

When we develop a traversal of a tree, we are visiting each node. Consequently, we must perform at least n operations. Therefore, the run time of any traversal must be $\Omega(n)$; however, we will attempt to ensure the run time is $\Theta(n)$. We will try to avoid using as much memory as is currently used by the tree: that is, we want to use $o(n)$ memory.

4.3.1 Description

A tree traversal is an algorithm for iterating through the entries of a tree. We have already seen one in Section 3.3.5 (Queues) where we performed a breadth-first traversal of a directory structure, that is, a general tree. In this example, we visited all of the children of the root before we proceeded to the next lower depth, as shown in Figure 1.

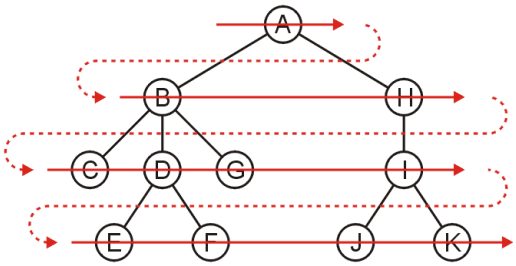

Figure 1. A breadth-first traversal.

As an alternative, we could instead search down first: continue following the first child of the first child and so on until we reach a leaf node, then go back to the most recent node that has a second child and follow that path.

Such a traversal is called a *depth-first traversal*.

4.3.2 Backtracking and Depth-first Traversals

Suppose we start at the root node and use the following algorithm to decide which node to visit next:

1. If there are unvisited children, we visit the next unvisted child (however *next* is defined), and
2. If we reach a node where we have visited all the children (including leaf nodes as a special case), we backtrack to the algorithm and repeat this decision-making process.

We are finished once all the children of the root are visited. Figure 2 shows this backtracking algorithm for visiting the nodes of the tree in Figure 1.

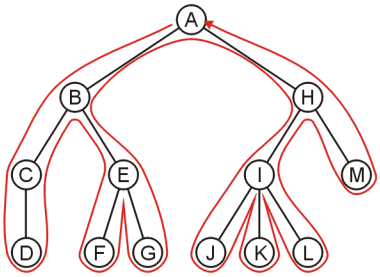

Figure 2. Visiting the nodes of a tree using a backtracking algorithm.

We will call a traversal that follows such a backtracking algorithm to be a *depth-first traversal*.

4.3.2.1 Pre-order and Post-order Depth-first Traversals

If you look at Figure 2, there is a point in the traversal where the node is visited for the first time, and a corresponding where the node is visited for a last time.

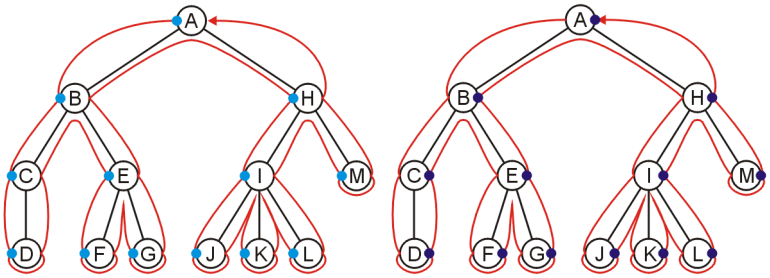

Figure 3. The First and last visits of a depth-first traversal.

If you list the nodes in the order in which they are visited for the first time (before any children are visited), you get the order

A, B, C, D, E, F, G, H, I, J, K, L, M

which is called the *pre-order* depth-first traversal. If you list the nodes in the order in which they are visited for the last time (after all children are visited), you get the order

D, C, F, G, E, B, J, K, L, I, M, H, A

and this is called the *post-order* depth-first traversal.

4.3.3 Implementation

A recursive implementation of a depth-first traversal is given here:

```
template <typename Type>
void Simple_tree<Type>::depth_first_traversal() const {
        // Perform any operations to be done before the children are visited

        for (
                ece250::Single_node<Simple_tree *> *ptr = children.head();
                ptr != 0;
                ptr = ptr->next()
        ) {
                ptr->retrieve()->depth_first_traversal();
        }

       // Perform any operations to be done after the children are visited
}
```

An alternate implementation would be similar to our breadth-first traversal which used a queue, but in this case, we would use a stack:

1. Create an empty stack and push the root node onto the stack, then
2. While the stack is not empty:
    a. Pop to the node on top of the stack, and
    b. Push all of its children onto the stack in *reverse* order.

4.3.4 Guidelines and Applications

When designing a depth-first traversal, it is necessary what information is required where:

1. The information that the children requires about the current node, and
2. The information that the current node requires from the children.

Therefore, the information should be collected as follows:

1. Before the children are traversed, what initializations, operations and calculations must be performed?
2. In recursively traversing the children:
    a. What information must be passed to the children during the recursive call?
    b. What information must the children pass back, and how must this information be collated?
3. Once all children have been traversed, what operations and calculations depend on information collated during the recursive traversals?
4. What information must be passed back to the parent?

We will look at three applications:

1. The height of a tree,
2. Printing a hierarchy, and
3. Determining memory usage.

4.3.4.1 Height

The `int height() const` function is recursive in nature:

1. Before the children are visited, the height of this sub-tree is assumed to be zero,
2. After each child is visited, we check whether we need to update the height,
3. No further calculations or operations are required after the traversals, and
4. Once all the children are visited, we return the height of the sub-tree.

If the node is the root of the tree, the returned height is the height of the tree. You can use Figure 4 to convince yourself about the truth of this statement.

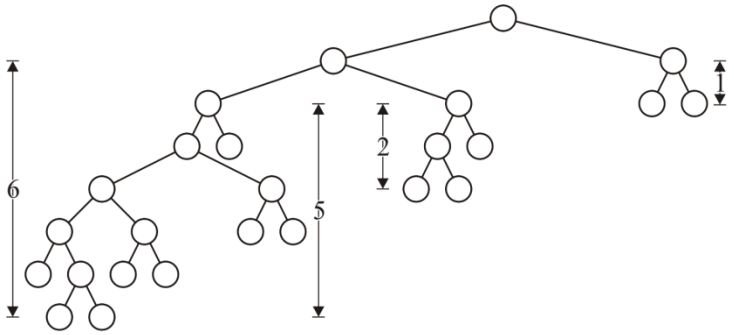

Figure 4. The heights of various trees in a sample.

4.3.4.2 Printing a Heirarchy

Consider the directory hierarchy shown in Figure 5.

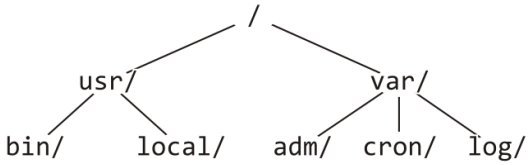

Figure 5. A directory hierarchy.

One means of visualizing such a hierarchy is to print it in a format similar to

```
/
   usr/
      bin/
      local/
   var/
      adm/
      cron/
      log/
```

Starting with the root at a tab level of 0, we recursively perform the operations:

1. Before the children are printed, print the name of the current directory at the current tab level,
2. For each child, recursively call this function (go to step 1) but at an incremented tab level,
3. Nothing is required once all the children are printed, and
4. Nothing need be returned to the parent.

Assuming that a member function `string Directory::name() const` returns a string containing the directory's name. A function that implements such a structure is

```
void Simple_tree<Directory>::print( int depth ) const {
    print_tabs( depth );
    std::cout << retrieve()->name() << '/' << std::endl;

    for (
        ece250::Single_node<Simple_tree *> *ptr = children.head();
        ptr != 0; ptr = ptr->next()
     ) {
        ptr->retrieve()->print( depth + 1 );
     }
}
```

4.3.4.3 Determining Memory Usage

Suppose that, in addition to the directory structure in Figure 5, we have the memory used by all the files in each directory as is shown in Figure 6.

Figure 6. A directory hierarchy with memory usage.

If we want to display each directory and the memory used by that directory and all of its subdirectories, we usually print it as follows:

```
      bin/ 12
      local/ 15
   usr/ 31
      adm/ 6
      cron/ 5
      log/ 9
   var/ 23
/ 61
```

Starting with the root at a tab level of 0, we recursively perform the operations:

1. Before the children are printed, initialize the memory usage to the files in the current directory,
2. For each child, recursively call this function (go to Step 1) but at an incremented tab level and add the returned memory usage onto the current directory's memory usage,
3. Print the current directory's name at the current tab level together with the memory usage and then return that memory usage, and
4. The memory usage must be returned to the parent.

Assuming that the member function `int Directory::memory() const` returns an integer equal to the memory occupied by the files in the current directory, a function that implements such a structure is

```
int Simple_tree<Type>::du( int depth ) const {
    int usage = retrieve()->memory();

    for (
        ece250::Single_node<Simple_tree *> * ptr = children.head();
        ptr != 0; ptr = ptr->next()
    ) {
        usage += ptr->retrieve()->du( depth + 1 );
    }

    print_tabs( depth );
    std::cout << retrieve()->name() << "/ " << usage << std::endl;

    return usage;
}
```