CS 228: Introduction to Data Structures
Lecture 34
Friday, April 17, 2015

Traversing Non-Binary Trees

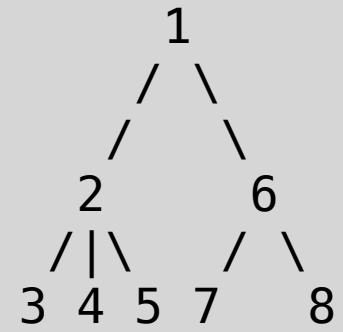
Here is the pseudocode for ***preorder*** traversal:

```
PREORDER(T):
    visit the root of T
    for each subtree T' of the root
        PREORDER(T')
```

The order in which we visit the subtrees of T is arbitrary; we can choose whichever order is best suited to our tree implementation. For the child-sibling representation, the simplest way is to start at the root's `firstChild` and follow the `nextSibling` links. This is what the implementation on the next page does.

```
public static void
  traversePreorder(CSNode<?> root)
{
  root.visit();
  if (root.getChild() != null)
  {
    CSNode<?> current = root.getChild();
    while (current != null)
    {
      traversePreorder(current);
      current = current.getSibling();
    }
  }
}
```

Example. Suppose `visit()` numbers the nodes in the order they are visited. Then, a preorder traversal would number the nodes like this:

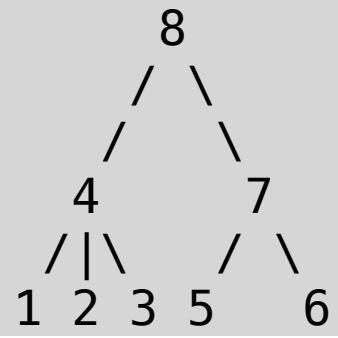


Each node is visited only once, so a preorder traversal takes O(n) time, where n is the number of nodes in the tree. In fact, all the traversals we will see take O(n) time.

Here is the pseudocode for ***postorder*** traversal:

```
POSTORDER(T):
    for each subtree T' of the root
        POSTORDER(T')
    visit the root of T
```

Example. A postorder traversal visits the nodes in this order.



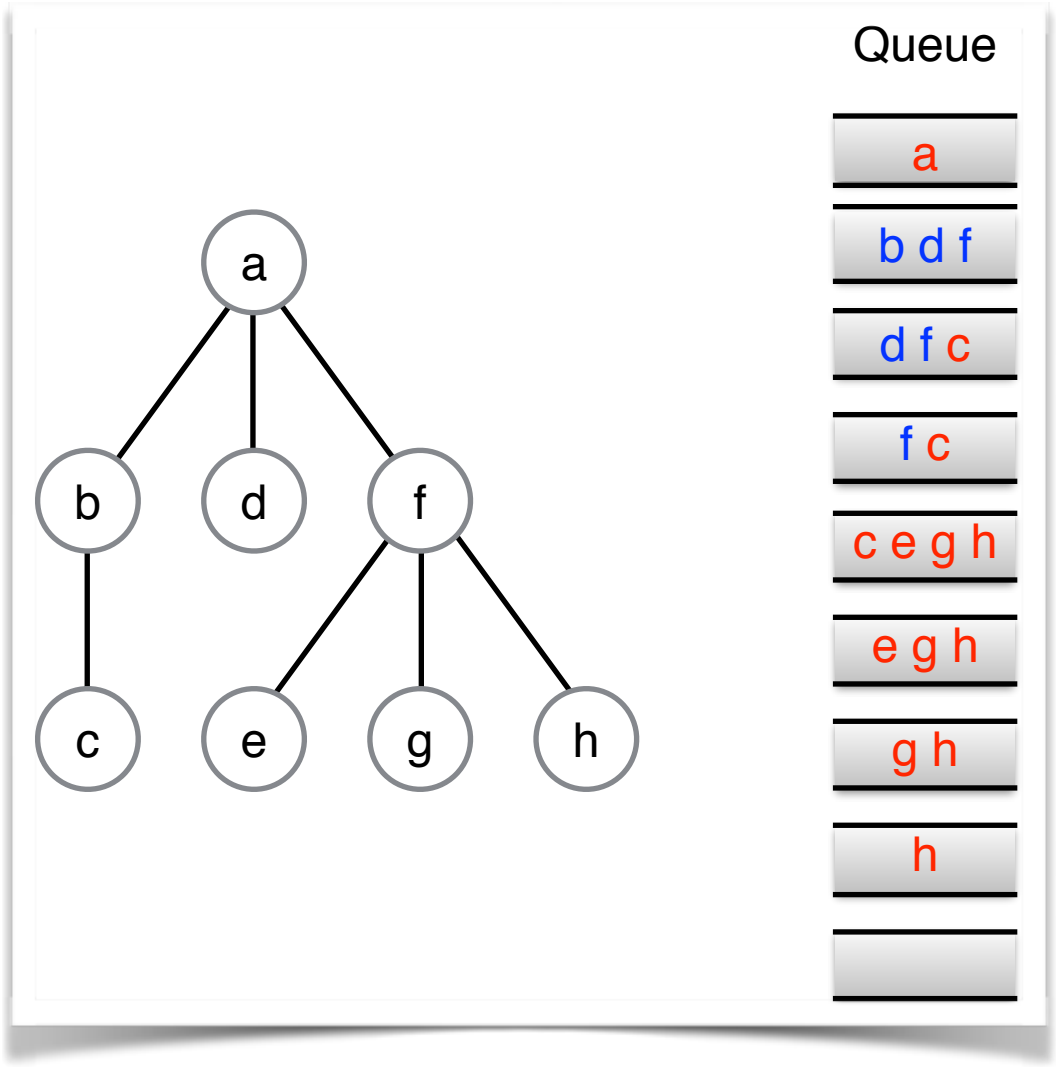
A postorder traversal is the natural way to sum the total disk space used in the root directory and its descendants. In the example above, to compute the total disk space at the root `~dfb/228/`, a postorder traversal would recursively compute the sizes of the files in `prj/`, `exm/` and `lec/`, and add these sizes to that of `syllabus`.

Level-order traversal. Unlike preorder and postorder traversals, it is easier to implement level order traversal iteratively, using a queue, than recursively. Initially, the queue contains only the root. We then repeat the following steps while the queue is nonempty:

- Dequeue a node.
- Visit it.
- Enqueue its children (in order, from left to right).

The running time of the algorithm is O(n): each node is added to the queue and removed from the queue exactly once. You'll find sample Java code on Blackboard.

The next figure illustrates level-order traversal. The colors of nodes in the queue alternate by level: nodes at levels 0 and 2 are red, nodes at level 1 are blue. Notice how using a queue ensures that the nodes in any level are processed before the nodes in the next level.



A final thought. If you use a stack instead of a queue, and push each node's children in reverse order — from right to left (so they pop off the stack in order from left to right) — you perform a preorder traversal. Think about why.

Graphs

A ***graph*** is an abstract representation of a set of objects, which are called ***nodes*** or ***vertices***, where some pairs of the objects are connected by links, which are called ***edges*** or ***arcs***. There are two types of graphs.

- In a ***directed graph*** (or ***digraph*** for short), every edge e is directed from some node v to some node w. We write e = (v,w), an ordered pair, and draw an arrow pointing from v to w. The vertex v is called the ***origin*** of e, and w is the ***destination*** of e.

- In an ***undirected graph***, edges have no favored direction, so we draw a curve without an arrowhead connecting v and w. We still write e = (v, w), but now e is an unordered pair, which means that (v, w) = (w, v).

Graphs can, for example, model street maps. For each intersection, define a node that represents it. If two intersections are connected by a length of street with no intervening intersection, define an edge connecting them. We might use an undirected graph, but if there are one-way streets, a directed graph is more appropriate. We can model a two-way street with two edges, one pointing in each direction.

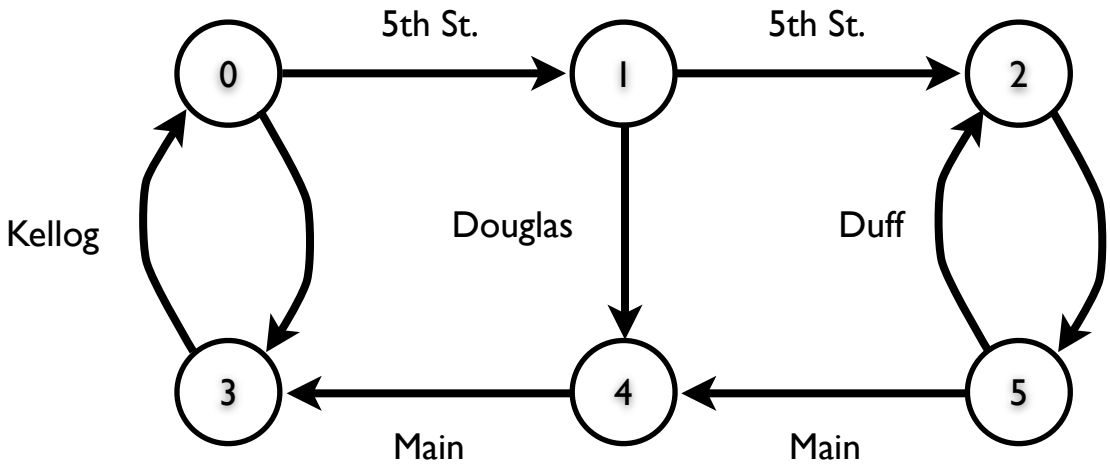


Figure 1. A street map.

On the other hand, if we want a graph that tells us which countries adjoin each other, an undirected graph makes sense.

Figure 2. Representing adjacencies between countries.

Multiple copies of an edge are (usually) forbidden, but a directed graph may contain both (v, w) and (w, v). Both types of graph can have ***self-edges*** of the form (v, v), which connect a vertex to itself. (Many applications, like the two illustrated above, don't use these.)

The preceding were just two out of the myriad of applications of graphs. In fact, graphs are everywhere:

- **Airline route maps:** Nodes are cities; *A* is related to *B* if there is a direct flight from *A* to *B*.

- **Food chains:** Nodes are species in an ecosystem, *A* is related to *B* if *A* is eaten by *B*.

- **Social networks:** Nodes are people; there is an edge between *A* and *B* if they are friends.

- **Prerequisite diagrams:** Nodes are courses in (say) the Software Engineering degree program; *A* is related to *B* if A is a prerequisite for *B*.

- **The World-Wide Web:** Nodes are web pages; *A* is related to *B* if *A* has a link to *B*.

- **Games:** Nodes are configurations of a chessboard, *A* is related to *B* if there is a legal move in *A* that takes you to configuration *B*.

- **State transition diagrams:** Nodes are states of a process, *A* is related to *B* if some step of the process takes you from state *A* to state *B* (e.g., telephone-answering finite-state machine, regular expression pattern matching).

Several of the above examples are illustrated on the slides posted on Blackboard.

Graph Terminology

We say that node u is a ***neighbor*** of node v if (v,u) is an edge.

- In an ***undirected*** graph, the neighbor relation is *symmetric* — that is, if u is a neighbor of v, then v is a neighbor u —, and we say that two nodes that are neighbors of each other are ***adjacent***. The number of neighbors of a node v is called the ***degree*** of v.

- In a ***directed*** graph, the number of neighbors of a node v is called the ***out-degree*** of v, and the number of nodes that have v as a neighbor is called the ***in-degree*** of v. Equivalently, the out-degree of v is the number of edges directed out of v and the in-degree is the number of edges directed into v.

A ***path*** is a sequence of vertices such that each successive pair of vertices is connected by an edge. For example, one path in the graph of Figure 1 is

France-Luxembourg-Germany-Denmark

If the graph is directed, the edges that form the path must all be aligned with the direction of the path. A ***simple path*** is one with no repeated nodes. A ***cycle*** is a path with at least one edge whose first and last nodes are the same. A

simple cycle is a cycle with no repeated edges or nodes (except the requisite repetition of the first and last nodes).

The ***length*** of a path or a cycle is its number of edges it traverses. E.g., in the street map graph above, 3−0−1−2−5 is a path of length 4. It is perfectly OK to talk about a path of length zero, such as (the single node) 2. The ***distance*** from one node to another is the length of the shortest path from one to the other.

An undirected graph is ***connected*** if there is a path from every node to every other node in the graph. A graph that is not connected consists of a set of ***connected components***, which are maximal connected subgraphs.

Intuitively, if the nodes were physical objects, such as knots or beads, and the edges were physical connections, such as strings or wires, a connected graph would stay in one piece if picked up by any node, and a graph that is not connected comprises two or more such pieces.

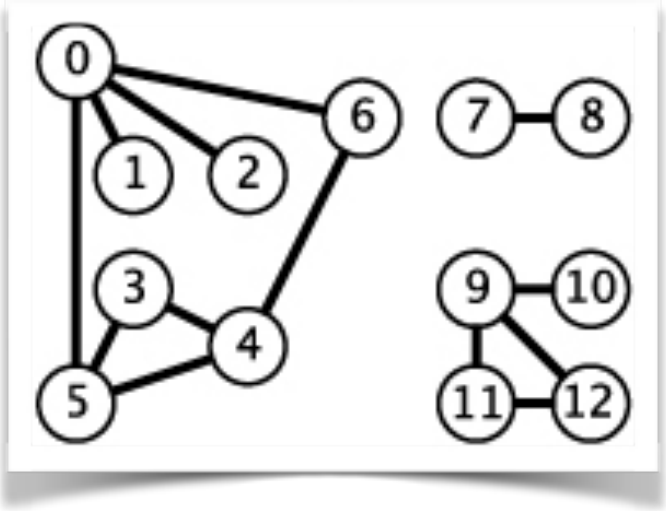


Figure 3. A disconnected graph. Its components are {0, 1, 2, 3, 4, 5, 6}, {7, 8}, and {9, 10, 11, 12}.

Generally, processing a graph necessitates processing the connected components one at a time.

In a directed graph, the existence of a path from u to v does not imply that there is a path from v to u. We say that a directed graph is ***strongly connected*** if for every pair of nodes u and v there is a path from u to v and a path from v to u. The graph in Figure 1 is strongly connected, but the one in the next figure is not. Notice, though that any directed graph can be decomposed into ***strongly connected components*** —subsets of nodes that are strongly connected, such that if you add any other node to the subset, it is no longer strongly connected.

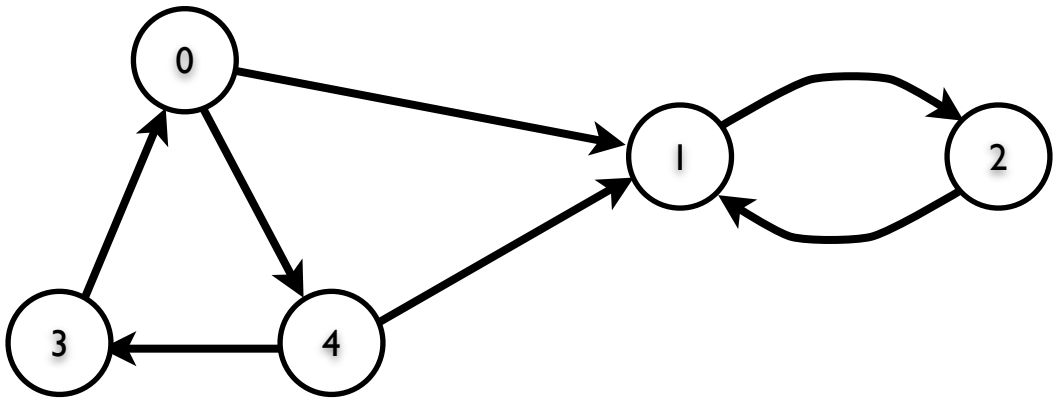


Figure 4. A graph that is not strongly connected. Its strongly connected components are {0, 3, 4} and {1, 2}.

We are now ready to define some common graph problems (there are many others):

- Given a starting node S, what other nodes are reachable by a path from S?

  Examples: web crawler, garbage collector

- Given nodes S and T, is there a path from S to T?

  Examples: path planning (robotics), model checking

- Given two nodes S and T, what is the shortest path from S to T? "Shortest" may mean fewest number of edges, or it may be based on an "edge cost" that measures something else.

Examples: network router (fewest number of hops), MapQuest (minimal driving time).

Representing Graphs

To manipulate graphs with a computer, we need data structures to represent them. Next week, we will study three approaches: adjacency matrices, adjacency lists, and a `HashMap`/`HashSet` structure. To give you a taste for what is to come, the next page shows the adjacency list representation for the graph of Figure 2. The representation consists of an array with one entry per node. The entry for each node v references the set of all neighbors of v.

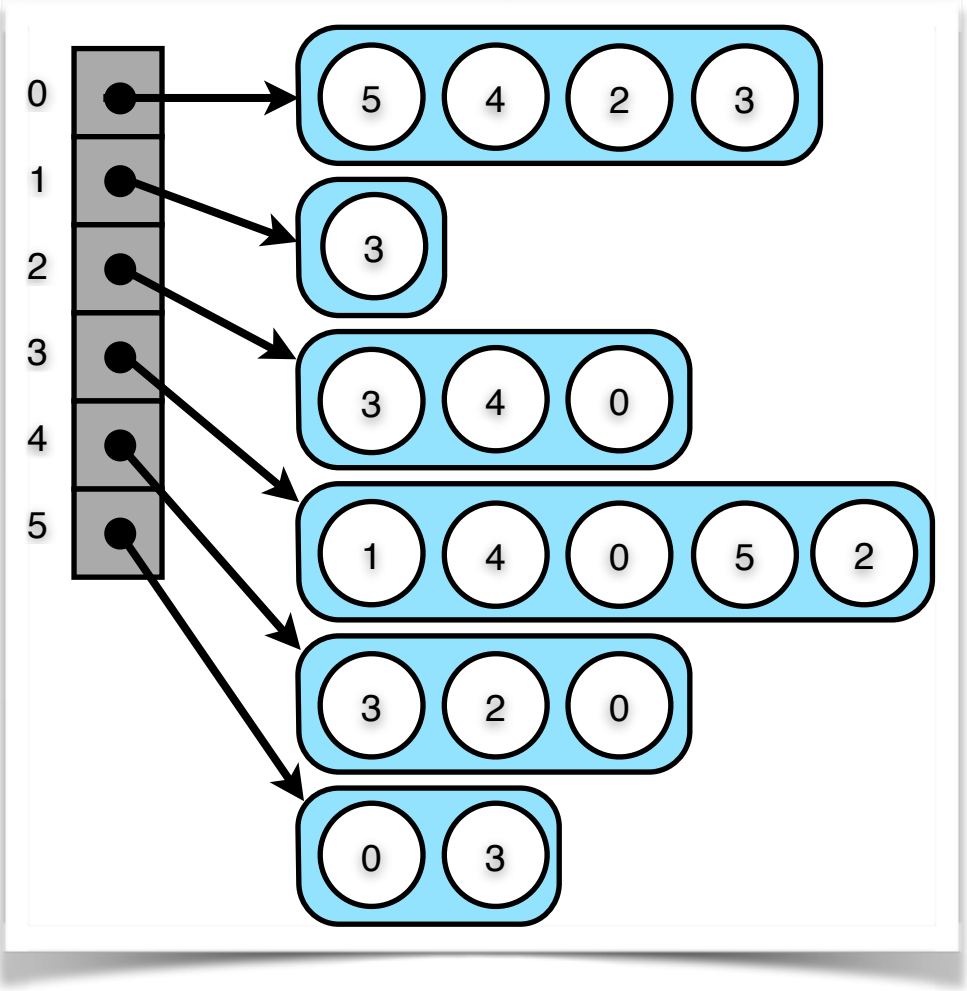


Figure 5. Adjacency list representation for the graph of Figure 2, where 0 ≡ Belgium, 1 ≡ Denmark, 2 ≡ France, 3 ≡ Germany, 4 ≡ Luxembourg, and 5 ≡ Netherlands.