CS 228: Introduction to Data Structures
Lecture 35
Monday, November 28, 2016

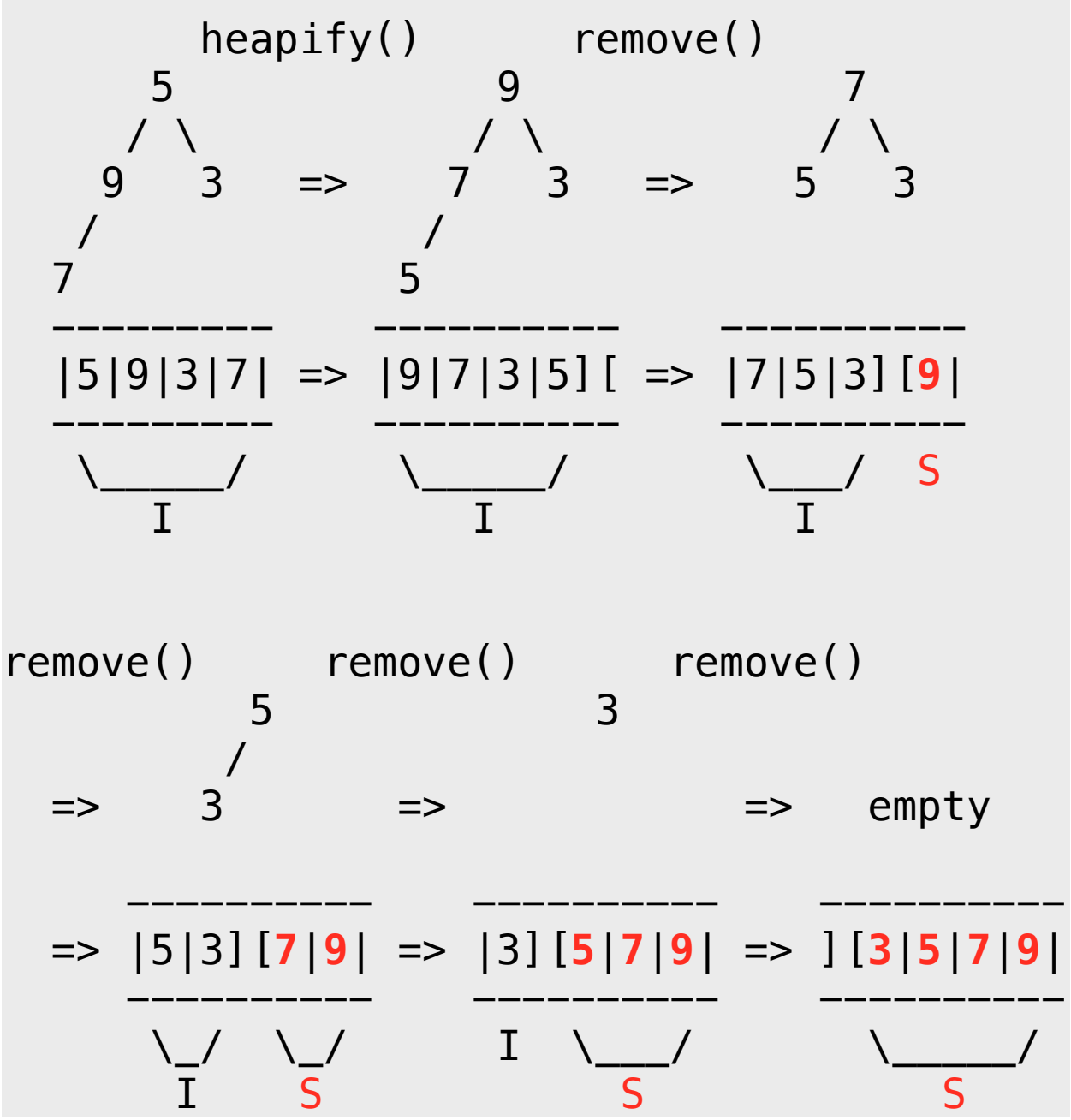
Heapsort

Binary heaps give us another O(n log n) sorting algorithm. We use a ***max heap***; i.e., one where the highest-priority item is the one with the largest key. Thus in a max heap, the largest item is at the root of the heap. A max heap is easily implemented by defining the appropriate comparator (see the code on Blackboard). We sort as follows.

```
HEAPSORT:
|   put the elements into a max heap H
|   create an empty list L
|   while !H.isEmpty()
|   |   x = H.remove()
|   |   Put x at the beginning of L
|   return L
```

In fact, if the input elements are in an array, we can do the sorting ***in place***. That is, we can use the same array to store the heap H and the list L. As items are removed, the heap shrinks toward the beginning of the array, making room for the elements removed.

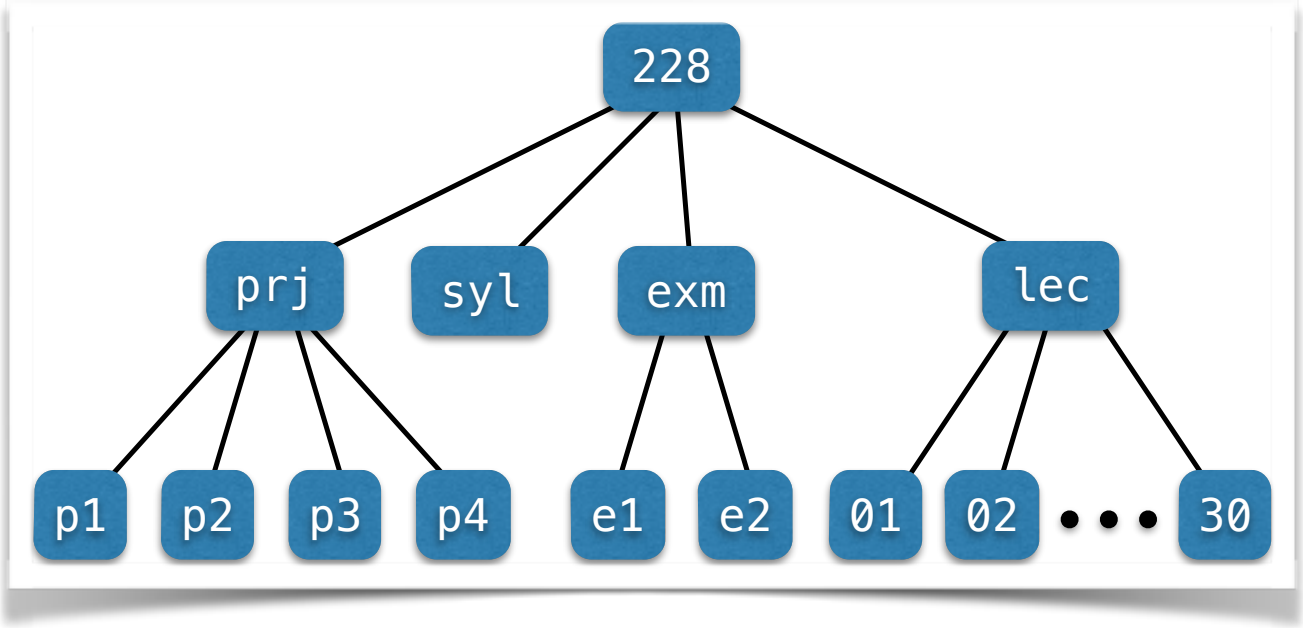
Example.



I = in heap S = sorted

Non-Binary Rooted Trees

All trees we have seen so far — expression trees, BSTs, heaps — are binary. ***Non***-binary trees also have several uses. For instance, file directories are typically tree-structured. Here is an early version of a directory for this class.



More generally, a rooted tree can represent all sorts of hierarchical information; e.g. organizational charts.

Representing Non-Binary Rooted Trees

We can represent a tree using a data structure where every node has one reference to the data stored at that node, one reference to that node's parent, and one

reference to a list of that node's children. The latter can be stored, for instance, as a linked list.

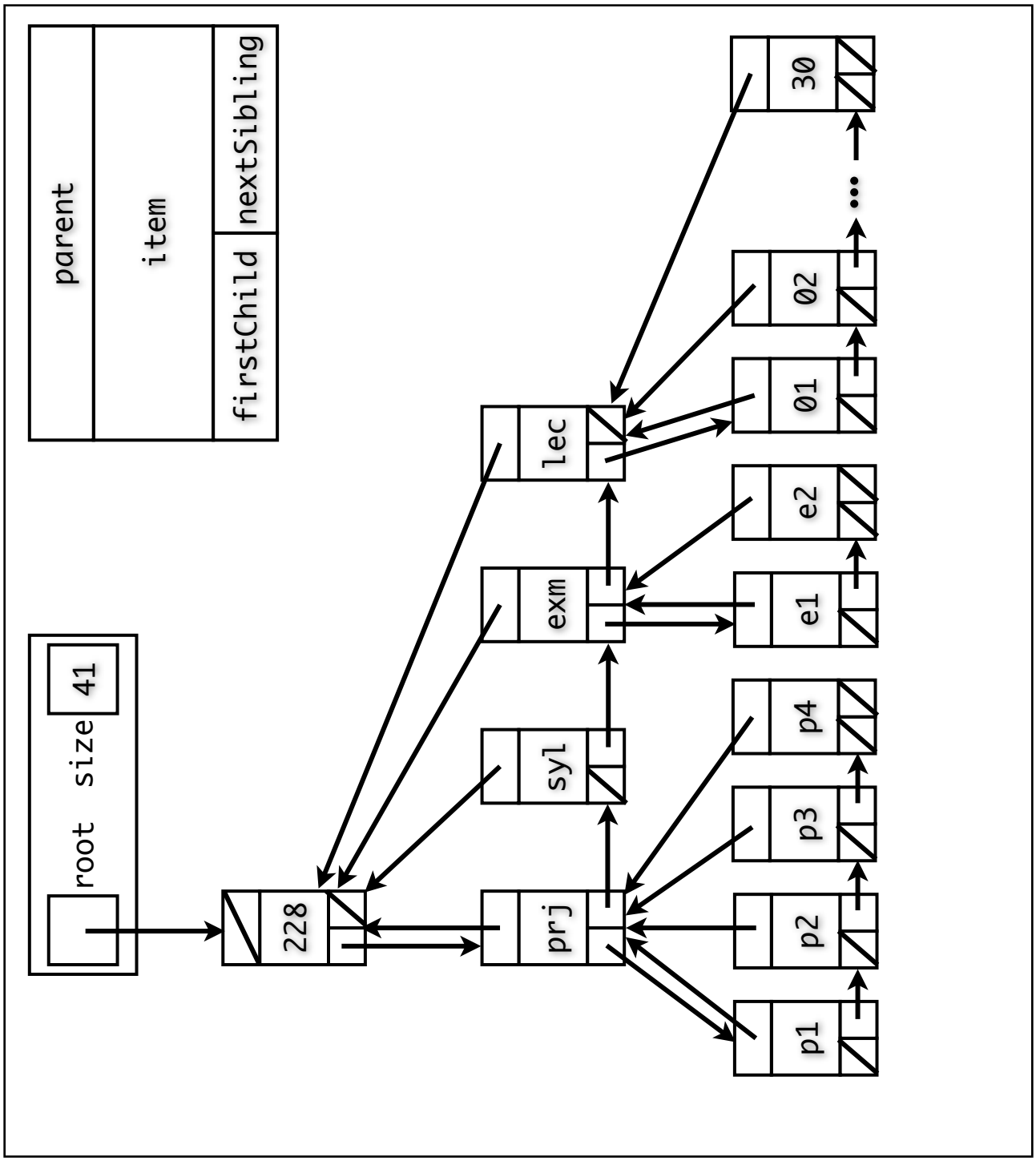
The ***child-sibling representation*** is another popular tree representation. As before, each node has `data` and `parent` references; however, instead of referencing a list of children, each node references just its leftmost child. Each node also references its next sibling to the right. These `nextSibling` references are used to join the children of a node in a singly-linked list, whose head is the node's `firstChild`. Here are the basic definitions; the full code is on Blackboard.

```
public class CSNode<E>
{
  protected CSNode<E> parent;
  protected CSNode<E> firstChild;
  protected CSNode<E> nextSibling;
  protected E data;

  public CSNode() {}

  public CSNode(E data)
  {
    this(data, null, null);
  }

  public CSNode(E data, CSNode<E> child,
                CSNode<E> sibling)
  {
    this.firstChild = child;
    this.nextSibling = sibling;
    this.data = data;
  }
```

A tree is represented by a reference to the root node, and the tree's size (number of nodes).

```
public class CSTree<E> {
  CSNode<E> root;
  int size;
}
```

Here is an example.



Traversing Non-Binary Trees

Preorder, postorder, and level-order traversal easily extend to non-binary trees (inorder traversal does not make sense for non-binary trees). For instance, a preorder traversal is a natural way to print a directory structure:

```
~dfb/228
   prj
      p1
      p2
      p3
      p4
   syllabus
   exm
      e1
      e2
   lec
      01
      02
      .
      .
      .
      30
```

We will see how to traverse non-binary trees next time.

Traversing Non-Binary Trees

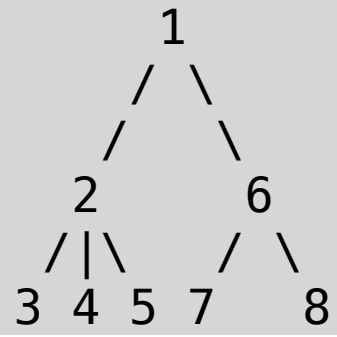
Here is the pseudocode for ***preorder*** traversal:

```
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, as shown below.

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

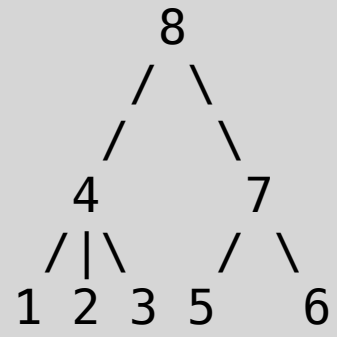
Example. Suppose `visit()` numbers the nodes in the order they are visited. Then, a preorder traversal would number the nodes like this:



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.



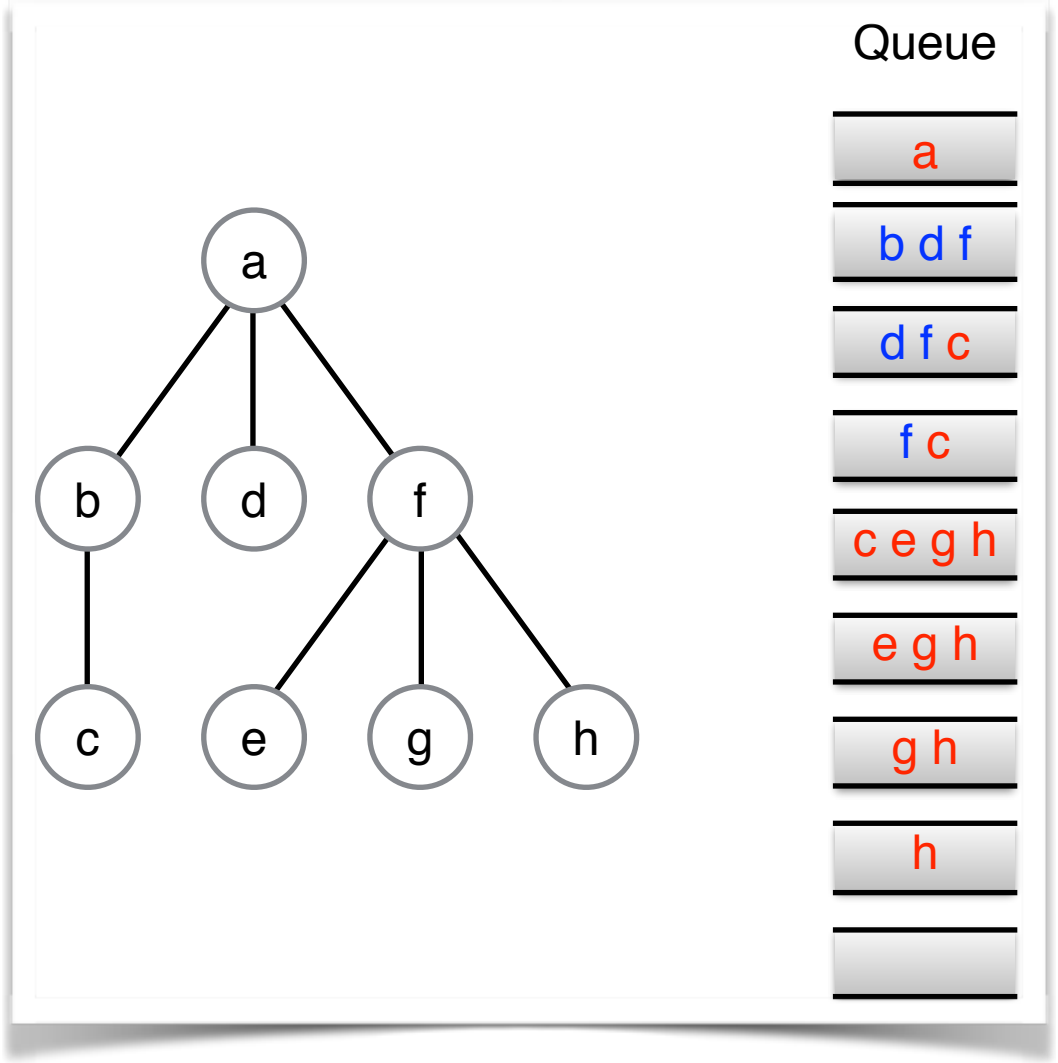
A postorder traversal is the natural way to sum the total disk space used in the root directory and its descendants. In the example above, to compute the total disk space at the root `~dfb/228/`, a postorder traversal would recursively compute the sizes of the files in `prj/`, `exm/` and `lec/`, and 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, called ***nodes*** or ***vertices***, where some pairs of the objects are connected by links, 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, so (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.

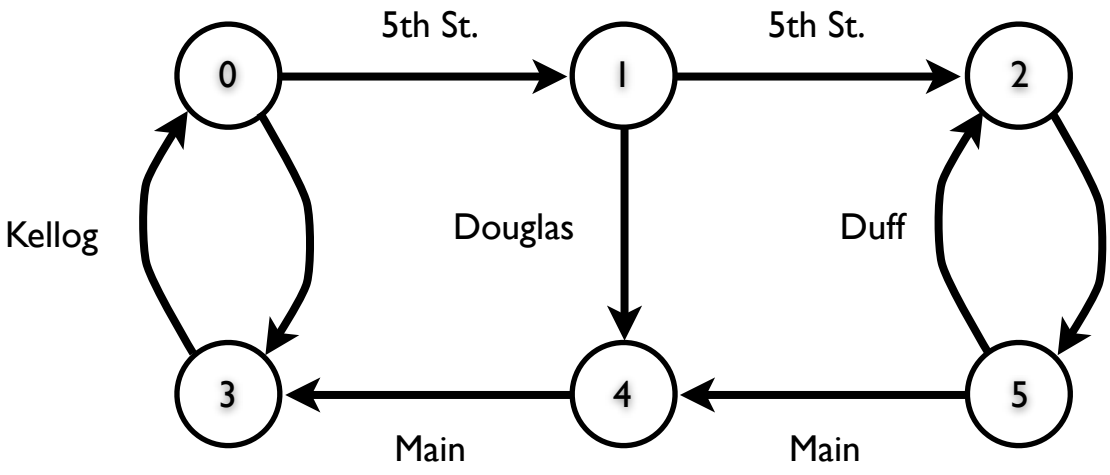


Figure 1. A street map.

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.