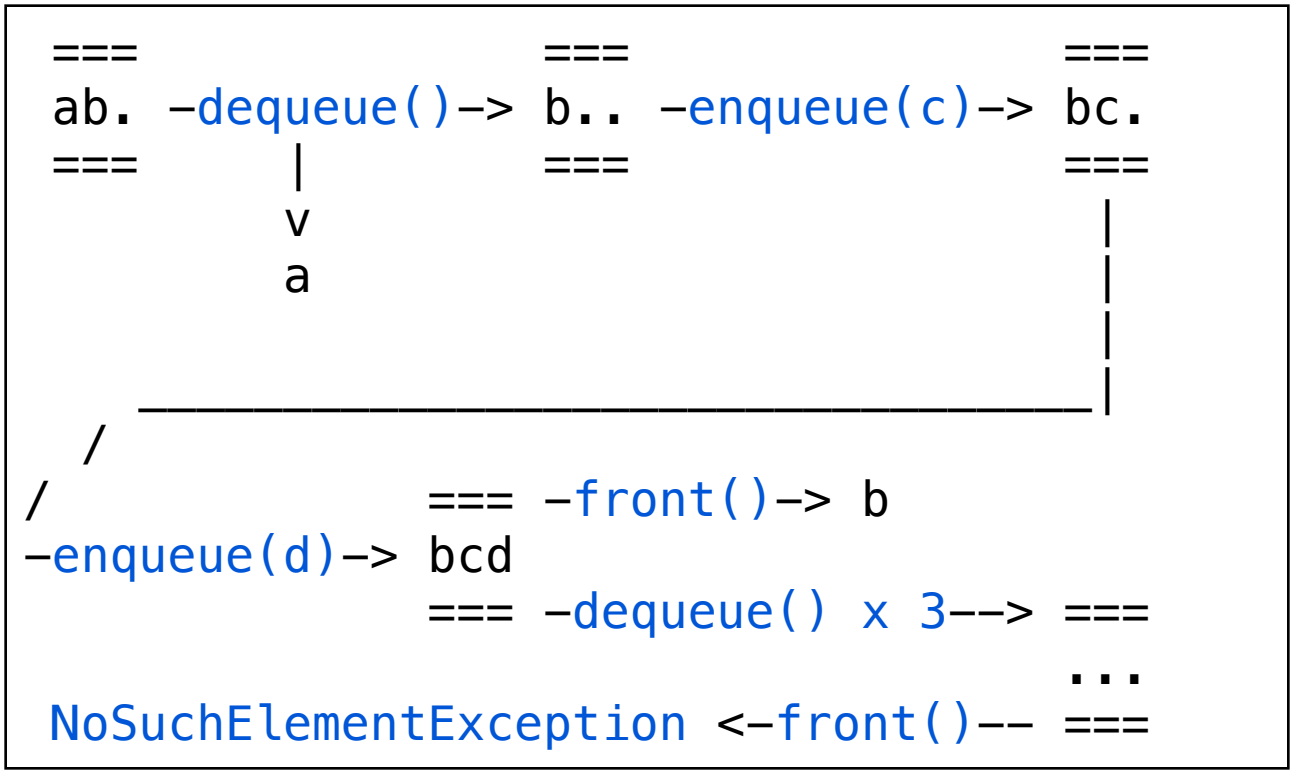
CS 228: Introduction to Data Structures
Lecture 24
Friday, March 13, 2015

Queues

A stack is a list that operates under the ***last-in first-out*** (***LIFO***) policy. A ***queue*** is a list that operates under the ***first-in first-out*** (***FIFO***) policy: You may read or remove only the item at the ***front*** (***head***) of the queue, and you may add an item only to the ***back*** (***tail***) of the queue. You may also examine the front item.



Application: Printer queues. Each job you submit to a printer goes into a queue. When the printer finishes printing a job, it dequeues the next job and prints it.

The Java Queue Interface

`java.util` contains a `Queue<E>` interface that contains all the methods you would expect from a FIFO queues, as well as other kinds of queues. Java offers several implementations, for example the `LinkedList` class.

The methods in this interface come in two forms: one throws an exception if the operation fails, the other returns a special value if the operation cannot be done.

front()

`E element()`. Retrieves, but does not remove, the head of this queue. Throws `NoSuchElementException` if this queue is empty.

`E` **`peek()`.** Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.

enqueue()

`boolean` **`add`**`(E e)`. Inserts e into this queue if it is possible to do so without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if out of space[1].

`boolean` **`offer`**`(E e)`. Inserts e into this queue if it is possible to do so without violating capacity restrictions.

dequeue()

`E` **`poll`**`()`. Retrieves and removes the head of this queue, or returns `null` if this queue is empty.

`E` **`remove`**`()`. Retrieves and removes the head of this queue. Throws `NoSuchElementException` if this queue is empty.

Since `Queue<E>` extends `Collection<E>`, it inherits all of the latter's methods, including `isEmpty()`, `size()`, and `iterator()`.

[1] This form of insertion is designed for use with capacity-restricted `Queue` implementations. In most implementations, insertion cannot fail.

Implementation

In any reasonable implementation, all queue methods run in O(1) time. Let's briefly examine two approaches.

Linked list implementation. It is easy to implement a queue as a singly-linked list with a tail pointer. It is perhaps even better to use a *circular* list, where a pointer to the last node also gives easy access to the first node, by following one link. Thus we can handle the structure by a single pointer, instead of two.

Circular array implementation. Here we use an array a to store the elements. Additionally, we have two indices:

- `first`: points to first element of queue (front)
- `last`: points to first available slot in the array (just before the back)

We initialize `first = last = 0`. The queue is empty when `first == last`.

To enqueue, put the new item in `a[last]` and increment `last`. To dequeue, return `a[first]` and increment `first`. A potential problem is that after a series of enqueues/dequeues, we could have something like this:

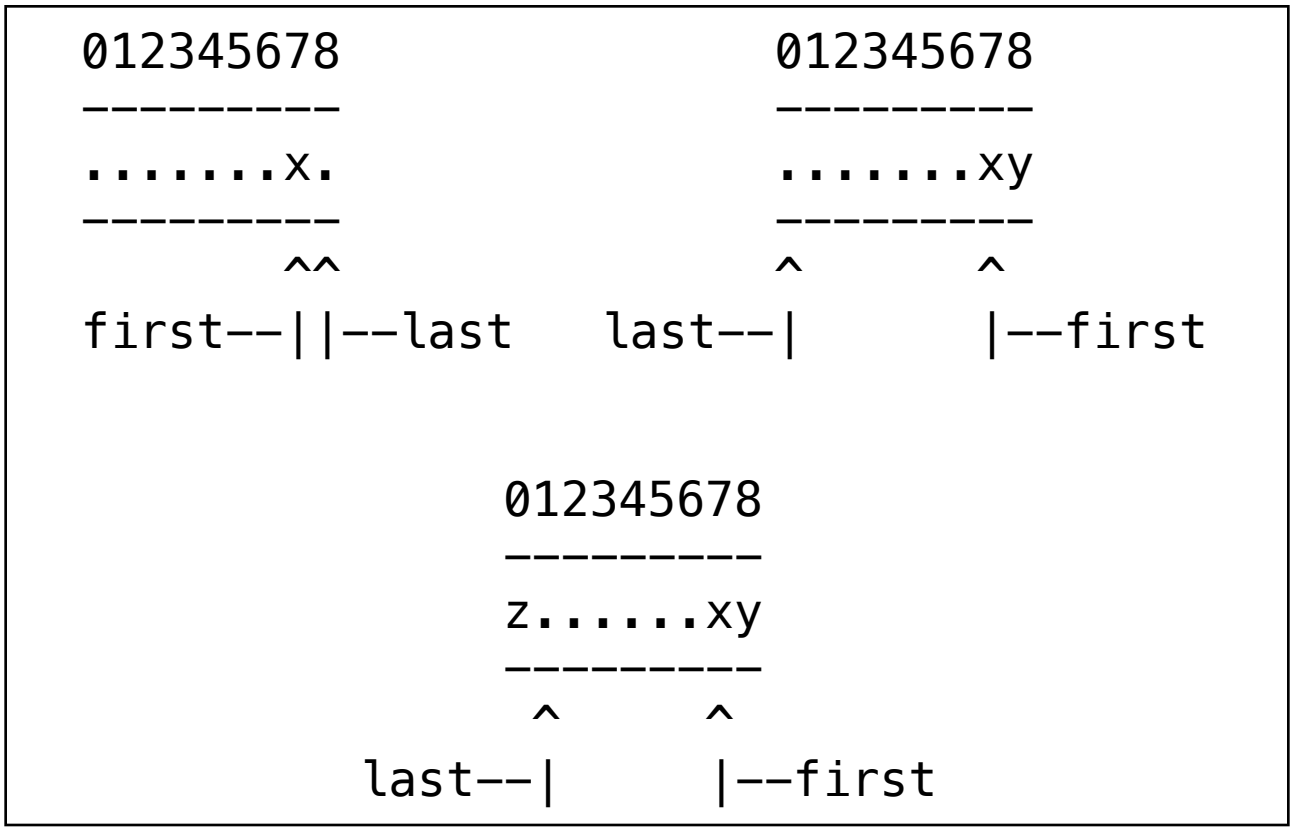
```
012345678
---------
.......x.
---------
        ^^
first--||--last
```

The array is not full, but we can only fit in one more element. Where do we put `last` after that?

An elegant solution is to treat the array as being ***circular***. That is, when `last == a.length` and we need to increment `last` to insert a new item, we just reset `last` to `0`. This is easy to do with ***modular arithmetic***:

```
last = (last + 1) % a.length;
```

Here "%" is the mod operator, and "A % B" is the remainder of A when divided by B.



Note that we still use the convention that the queue is empty when `first == last`. This means that at least one entry of array a will be left unused. Otherwise, we would not be able to distinguish between an empty queue and a full one.

Deques

A ***deque*** (pronounced "deck") is a **D**ouble-**E**nded **QUE**ue; it allows insertion and removal at both ends. You can easily build a fast deque using a doubly-linked list. I will not say much more on this topic for now other than to mention that Java offers a nice `Deque<E>` interface —

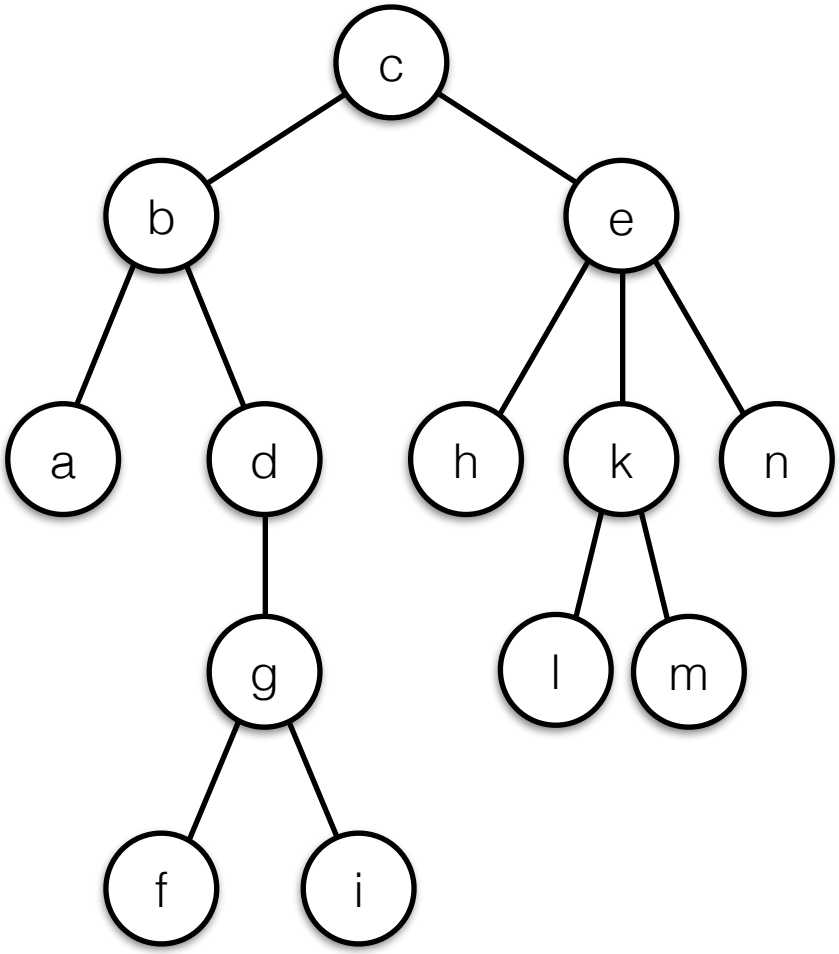
implemented by, e.g., `LinkedList<E>` —, which provides the necessary methods and, in fact gives you all the stack methods as well.

Rooted Trees

A ***tree*** consists of a set of ***nodes*** and a set of ***edges*** that connect pairs of nodes. A ***path*** is a connected sequence of zero or more edges. A tree has the property that there is exactly one path (no more, no less) between any two nodes of the tree.

In a ***rooted*** tree, one distinguished node is called the ***root***. Every node x, except the root, has exactly one ***parent*** node p, which is the first node traversed on the path from x to the root. x is p's ***child***. The root has no parent. A node can have any number of children.

To illustrate the notions we have just introduced, consider the tree T on the next page



The root of T is c, with children b and e. c-b-d-g-f is a path from c to f; b and g are the parents of d and f, respectively.

Let us use the tree T above to illustrate some additional tree terminology.

- A ***leaf*** is a node with no children. For example, the leaves of T are a, f, i, h, l, m, and n.

- ***Siblings*** are nodes with the same parent. For example, a and d are siblings in T; so are h, k, and n (there are

other examples of siblings in T).

- The ***ancestors*** of a node x are the nodes on the path from x to the root. These include x's parent, x's parent's parent, x's parent's parent's parent, and so forth up to the root. Note that the ancestors of x include x itself, and that the root is an ancestor of every node in the tree. For example, the ancestors of f in T are c, b, d, g, and f itself.

- If x is an ancestor of y, then y is a ***descendant*** of x. For example, the descendants of node e in T are e, h, k, l, m, and n.

- The ***length*** of a path is the number of edges in the path. For example, the length of the path c-b-d-g-f is 4; the length of path c-e-k-l is 3.

- The ***depth*** of a node x is the length of the path from x to the root. (The depth of the root is zero.) For example, the depth of k in T is 2; the depth of g is 3.

- The ***height of a node*** x is the length of the path from x to its deepest descendant. (The height of a leaf is zero.) For example, the height of b in T is 3, since its deepest descendant is f (or i) and the path from b to f, namely b-d-g-f, has 3 edges.

- The ***height of a tree*** is the depth of its deepest node; equivalently, it is the height of the root. The height of our example tree T is 4, since its deepest node, f (or i), is at depth 4.

- The ***subtree*** rooted at node x is the tree formed by x and its descendants. For example, the subtree at node e in T has nodes e, h, k, l, m, and n.

- The ***degree*** of a node x is the number of children of x. For example, the degree of c, d, e, and m in T are, respectively 2, 1, 3, and 0.

- A ***binary tree*** is a tree in which every node has degree at most two, and every child is either a ***left child*** or a ***right child***, even if it is the only child its parent has.

Rooted trees can be defined recursively. Here is a recursive definition of a binary tree.

Definition. A ***binary tree*** T is a structure defined on a finite set of nodes that either

- contains no nodes, or
- is composed of three disjoint sets of nodes:
  - a ***root*** node,
  - a binary tree called the ***left subtree*** of T, and
  - a binary tree called the ***right subtree*** of T.

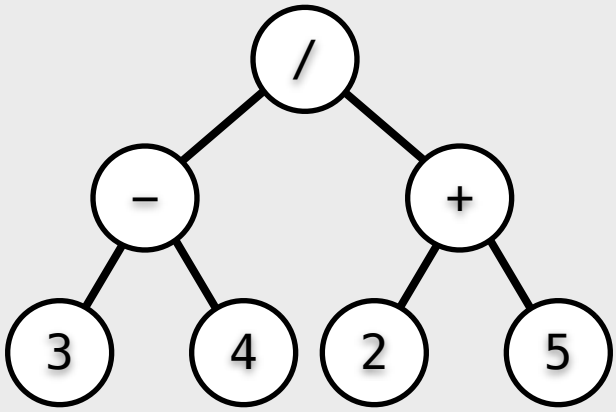
Note. This recursive definition will be useful when devising algorithms for binary trees. In fact, recursive definitions and recursive algorithms go hand in hand.

Binary trees have many uses. For instance, an ***expression tree*** is a binary tree that represents an arithmetic expression. Its leaves are operands; the remaining, internal, nodes are operators. An operator node labeled by an operator *op*, and is interpreted as "take the expressions represented by the left and right subtrees and combine them using *op*." Notice that left-to-right order is important; e.g., a−b is not the same as b−a.

Example. The expression tree for

```
(3−4)/(2+5)
```

is



Notice that the expression tree gives the order of evaluation of operators explicitly, so parentheses are not needed.

Expression trees are just one of the many, many applications of binary trees in Computer Science. In fact, even though binary trees are in a way just a special case of ordinary trees, they are important enough in their own right that we'll study them first, before considering more general trees and graphs.