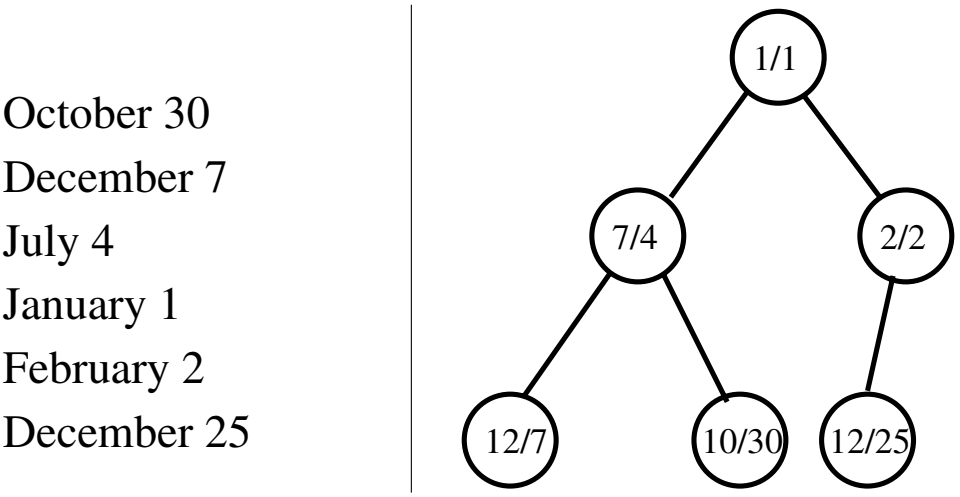

INPUT OUTPUT

12.2 Priority Queues

Input description: A set of records with numerically or otherwise totally-ordered keys.

Problem description: Build and maintain a data structure for providing quick access to the *smallest* or *largest* key in the set.

Discussion: Priority queues are useful data structures in simulations, particularly for maintaining a set of future events ordered by time. They are called "priority" queues because they enable you to retrieve items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but by which item has the highest priority of retrieval.

If your application will perform no insertions after the initial query, there is no need for an explicit priority queue. Simply sort the records by priority and proceed from top to bottom, maintaining a pointer to the last record retrieved. This situation occurs in Kruskal's minimum spanning tree algorithm, or when simulating a completely scripted set of events.

However, if you are mixing insertions, deletions, and queries, you will need a real priority queue. The following questions will help select the right one:

- *What other operations do you need?* – Will you be searching for arbitrary keys, or just searching for the smallest? Will you be deleting arbitrary elements from the data, or just repeatedly deleting the top or smallest element?

- *Do you know the maximum data structure size in advance?* – The issue here is whether you can preallocate space for the data structure.
- *Might you change the priority of elements already in the queue?* – Changing the priority of elements implies that we must be able to retrieve elements from the queue based on their key, in addition to being able to retrieve the largest element.

Your choices are between the following basic priority queue implementations:

- *Sorted array or list* – A sorted array is very efficient to both identify the smallest element and delete it by decrementing the top index. However, maintaining the total order makes inserting new elements slow. Sorted arrays are only suitable when there will be few insertions into the priority queue. Basic priority queue implementations are reviewed in Section 3.5 (page 83).
- *Binary heaps* – This simple, elegant data structure supports both insertion and extract-min in $O(\lg n)$ time each. Heaps maintain an implicit binary tree structure in an array, such that the key of the root of any subtree is less than that of all its descendents. Thus, the minimum key always sits at the top of the heap. New keys can be inserted by placing them at an open leaf and percolating the element upwards until it sits at its proper place in the partial order. An implementation of binary heap construction and retrieval in C appears in Section 4.3.1 (page 109)

  Binary heaps are the right answer when you know an upper bound on the number of items in your priority queue, since you must specify array size at creation time. Even this constraint can be mitigated by using dynamic arrays (see Section 3.1.1 (page 66)).
- *Bounded height priority queue* – This array-based data structure permits constant-time insertion and find-min operations whenever the range of possible key values is limited. Suppose we know that all key values will be integers between 1 and n. We can set up an array of n linked lists, such that the ith list serves as a bucket containing all items with key i. We will maintain a *top* pointer to the smallest nonempty list. To insert an item with key k into the priority queue, add it to the kth bucket and set $top = \min(top, k)$. To extract the minimum, report the first item from bucket *top*, delete it, and move *top* down if the bucket has become empty.

  Bounded height priority queues are very useful in maintaining the vertices of a graph sorted by degree, which is a fundamental operation in graph algorithms. Still, they are not as widely known as they should be. They are usually the right priority queue for any small, discrete range of keys.
- *Binary search trees* – Binary search trees make effective priority queues, since the smallest element is always the leftmost leaf, while the largest element is

always the rightmost leaf. The min (max) is found by simply tracing down left (right) pointers until the next pointer is nil. Binary tree heaps prove most appropriate when you need other dictionary operations, or if you have an unbounded key range and do not know the maximum priority queue size in advance.

- *Fibonacci and pairing heaps* – These complicated priority queues are designed to speed up *decrease-key* operations, where the priority of an item already in the priority queue is reduced. This arises, for example, in shortest path computations when we discover a shorter route to a vertex v than previously established.

  Properly implemented and used, they lead to better performance on very large computations.

Implementations: Modern programming languages provide libraries offering complete and efficient priority queue implementations. Member functions `push`, `top`, and `pop` of the C++ *Standard Template Library* (STL) `priority_queue` template mirror heap operations `insert`, `findmax`, and `deletemax`. STL is available with documentation at *http://www.sgi.com/tech/stl/*. See Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL.

LEDA (see Section 19.1.1 (page 658)) provides a complete collection of priority queues in C++, including Fibonacci heaps, pairing heaps, Emde-Boas trees, and bounded height priority queues. Experiments reported in [MN99] identified simple binary heaps as quite competitive in most applications, with pairing heaps beating Fibonacci heaps in head-to-head tests.

The *Java Collections* `PriorityQueue` class is included in the java.util package of Java standard edition (*http://java.sun.com/javase/*). The *Data Structures Library in Java* (JDSL) provides an alternate implementation, and is available for non-commercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL.

Sanders [San00] did extensive experiments demonstrating that his sequence heap, based on k-way merging, was roughly twice as fast as a well-implemented binary heap. See *http://www.mpi-inf.mpg.de/∼sanders/programs/spq/* for his implementations in C++.

Notes: *The Handbook of Data Structures and Applications* [MS05] provides several up-to-date surveys on all aspects of priority queues. Empirical comparisons between priority queue data structures include [CGS99, GBY91, Jon86, LL96, San00].

Double-ended priority queues extend the basic heap operations to simultaneously support both find-min and find-max. See [Sah05] for a survey of four different implementations of double-ended priority queues.

Bounded-height priority queues are useful data structures in practice, but do not promise good worst-case performance when arbitrary insertions and deletions are permitted. However, von Emde Boas priority queues [vEBKZ77] support $O(\lg \lg n)$ insertion, deletion, search, max, and min operations where each key is an element from 1 to n.

Fibonacci heaps [FT87] support insert and decrease-key operations in constant amortized time, with $O(\lg n)$ amortized time extract-min and delete operations. The constant-time decrease-key operation leads to faster implementations of classical algorithms for shortest-paths, weighted bipartite-matching, and minimum spanning tree. In practice, Fibonacci heaps are difficult to implement and have large constant factors associated with them. However, pairing heaps appear to realize the same bounds with less overhead. Experiments with pairing heaps are reported in [SV87].

Heaps define a partial order that can be built using a linear number of comparisons. The familiar linear-time merging algorithm for heap construction is due to Floyd [Flo64]. In the worst case, $1.625n$ comparisons suffice [GM86] and $1.5n - O(\lg n)$ comparisons are necessary [CC92].

Related Problems: Dictionaries (see page 367), sorting (see page 436), shortest path (see page 489).