The Priority Queue

Both the stack and the queue are data structures whose elements are ordered based on the sequence in which they have been inserted. The *pop* operation retrieves the last element inserted, and the *remove* operation retrieves the first element inserted. If there is an intrinsic order among the elements themselves (for example, numeric order or alphabetic order), it is ignored in the stack or queue operations.

The ***priority queue*** is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations. There are two types of priority queues: an ascending priority queue and a descending priority queue. An ***ascending priority queue*** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If *apq* is an ascending priority queue, the operation *pqinsert*(*apq*,*x*) inserts element *x* into *apq* and *pqmindelete*(*apq*) removes the minimum element from *apq* and returns its value.

A ***descending priority queue*** is similar but allows deletion of only the *largest* item. The operations applicable to a descending priority queue, *dpq*, are *pqinsert*(*dpq*,*x*) and *pqmaxdelete*(*dpq*). *pqinsert*(*dpq*,*x*) inserts element *x* into *dpq* and is logically identical to *pqinsert* for an ascending priority queue. *pqmaxdelete*(*dpq*) removes the maximum element from *dpq* and returns its value.

The operation *empty*(*pq*) applies to both types of priority queue and determines whether a priority queue is empty. *pqmindelete* or *pqmaxdelete* can only be applied to a nonempty priority queue [that is, if *empty*(*pq*) is *FALSE*].

Once *pqmindelete* has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest, and so on. Thus the operation successively retrieves elements of a priority queue in ascending order. (However, if a small element is inserted after several deletions, the next retrieval will return that small element, which may be smaller than a previously retrieved element.) Similarly, *pqmaxdelete* retrieves elements of a descending priority queue in descending order. This explains the designation of a priority queue as either ascending or descending.

The elements of a priority queue need not be numbers or characters that can be compared directly. They may be complex structures that are ordered on one or several fields. For example, telephone-book listings consist of last names, first names, addresses, and phone numbers and are ordered by last name.

Sometimes the field on which the elements of a priority queue is ordered is not even part of the elements themselves; it may be a special, external value used specifically for the purpose of ordering the priority queue. For example, a stack may be viewed as a descending priority queue whose elements are ordered by time of insertion. The element that was inserted last has the greatest insertion-time value and is the only item that can be retrieved. A queue may similarly be viewed as an ascending priority queue whose elements are ordered by time of insertion. In both cases the time of insertion is not part of the elements themselves but is used to order the priority queue.

We leave as an exercise for the reader the development of an ADT specification for a priority queue. We now look at implementation considerations.

Array Implementation of a Priority Queue

As we have seen, a stack and a queue can be implemented in an array so that each insertion or deletion involves accessing only a single element of the array. Unfortunately, this is not possible for a priority queue.

Suppose that the n elements of a priority queue *pq* are maintained in positions 0 to $n - 1$ of an array *pq.items* of size *maxpq*, and suppose that *pq.rear* equals the first empty array position, n. Then *pqinsert*(*pq*, *x*) would seem to be a fairly straightforward operation:

```
if (pq.rear >= maxpq)  {
   printf("priority queue overflow");
   exit(1);
}  /* end if */
pq.items[pq.rear] = x;
pq.rear++;
```

Note that under this insertion method the elements of the priority queue are not kept ordered in the array.

As long as only insertions take place, this implementation works well. Suppose, however, that we attempt the operation *pqmindelete*(*pq*) on an ascending priority queue. This raises two issues. First, to locate the smallest element, every element of the array from *pq.items*[0] through *pq.items*[*pq.rear* − 1] must be examined. Therefore a deletion requires accessing every element of the priority queue.

Second, how can an element in the middle of the array be deleted? Stack and queue deletions involve removal of an item from one of the two ends and do not require any searching. Priority queue deletion under this implementation requires both searching for the element to be deleted and removal of an element in the middle of an array.

There are several solutions to this problem, none of them entirely satisfactory:

- A special "empty" indicator can be placed into a deleted position. This indicator can be a value that is invalid as an element (for example, -1 in a priority queue of nonnegative numbers), or a separate field can be contained in each array element to indicate whether it is empty. Insertion proceeds as before, but when *pq.rear* reaches *maxpq* the array elements are compacted into the front of the array and *pq.rear* is reset to one more than the number of elements. There are several disadvantages to this approach. First, the search process to locate the maximum or minimum element must examine all the deleted array positions in addition to the actual priority queue elements. If many items have been deleted but no compaction has yet taken place, the deletion operation accesses many more array elements than exist in the priority queue. Second, once in a while insertion requires accessing every single position of the array, as it runs out of room and begins compaction.
- The deletion operation labels a position empty as in the previous solution, but insertion is modified to insert a new item in the first "empty" position. Insertion then involves accessing every array element up to the first one that has been deleted. This decreased efficieny of insertion is a major drawback to this solution.
- Each deletion can compact the array by shifting all elements past the deleted element by one position and then decrementing *pq.rear* by 1. Insertion remains unchanged. On the average, half of all priority queue elements are shifted for each deletion, so that deletion becomes quite inefficient. A slightly better alternative is to shift either all preceding elements forward or all succeeding elements backward, depending on which group is smaller. This would require

maintaining both *front* and *rear* indicators and treating the array as a circular structure, as we did for the queue.

- Instead of maintaining the priority queue as an unordered array, maintain it as an ordered, circular array as follows:

```
#define MAXPQ
struct pqueue {
        int items[MAXPQ];
        int front, rear;
}
struct pqueue pq;
```

pq.front is the position of the smallest element, *pq.rear* is 1 greater than the position of the largest. Deletion involves merely increasing *pq.front* (for the ascending queue) or decreasing *pq.rear* (for a descending queue). However, insertion requires locating the proper position of the new element and shifting the preceding or succeeding elements (again, the technique of shifting whichever group is smaller is helpful). This method moves the work of searching and shifting from the deletion operation to the insertion operation. However, since the array is ordered, the search for the position of the new element in an ordered array is only half as expensive on the average as finding the maximum or minimum of the unordered array, and a binary search might be used to reduce the cost even more. Other techniques that involve leaving gaps in the array between elements of the priority queue to allow for subsequent insertions are also possible.

4.1.9. A ***deque*** is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends of a deque *left* and *right*. How can a deque be represented as a C array? Write four C routines,

```
remvleft, remvright, insrtleft, insrtright
```

to remove and insert elements at the left and right ends of a deque. Make sure that the routines work properly for the empty deque and that they detect overflow and underflow.

4.1.10. Define an ***input-restricted deque*** as a deque (see the previous exercise) for which only the operations *remvleft*, *remvright*, and *insrtleft* are valid, and an ***output-restricted deque*** as a deque for which only the operations *remvleft*, *insrtleft*, and *insrtright* are valid. Show how each of these can be used to represent both a stack and a queue.