**Priority Queue**

A very common abstract data type (ADT) in computer science is the **priority queue**.

Each item in a priority queue has an associated **priority value** that determines the order in which the items leave the queue (not first-in first-out, as in the queues we have seen previously.)

The priority values of

A **Stack** and **Queue** are based on **insertion order**.

A **Priority Queue** are based on **completeness** and **heap-order**.

The primary operations that are necessary for a priority queue are:

1. insert
2. findMin (or findMax)
3. deleteMin (or deleteMax)

Of course, we want to be able to implement these three operations efficiently in terms of *n* (the number of items in the queue.)

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | insert | deleteMin | findMin |
| Unsorted collection | O(1) | O(n) | O(n) |
| Sorted Collection | O(n) | O(1) | O(1) |
| Binary search tree | O(n) worst-case | O(n) worst-case | O(n) worst-case |

**Priority Queue Implementations**

There are several obvious ways to implement a priority queue.

1. **Unsorted Collection**
   1. **Unsorted ArrayList**

**insert():** Perform insertions at the back in amortized *O*(1)

**deleteMin():** Delete the minimum by traversing then fixing the list in *O*(*N*) time

**findMin():** traversing the list in *O*(*N*) time

* 1. **Unsorted Linked List**

**insert():** Perform insertions at the front or back in *O*(1)

**deleteMin():** Delete the minimum by traversing then fixing the list in *O*(*N*) time

**findMin():** traversing the list in *O*(*N*) time

Unsorted linked lists are probably the better idea of the two, based on the fact that there are never more deleteMins than insertions.

1. **Sorted Collection**
   1. **Sorted Circular Array**

**insert():** Must sort entire array for every insert making insertions expensive (*O*(*N*))

**deleteMin():** deleteMins are cheap because they are always at front/back (*O*(*1*))

**findMin():** return the pointer to front/back of the list in *O*(*1*) time

* 1. **Sorted Linked List**

**insert():** Must sort entire list for every insert making insertions expensive (*O*(*N*))

**deleteMin():** deleteMins are cheap because they are always at front/back (*O*(*1*))

**findMin():** return the pointer to front/back of the list in *O*(*1*) time

1. **Binary Search Tree**
   1. **Unbalanced Binary Search Tree**

**insert():** *O*(log *N*) average but *O*(*N*) worst

**deleteMin():** *O*(log *N*) average but *O*(*N*) worst

**findMin():** *O*(log *N*) average but *O*(*N*) worst

Recall that the only element we ever delete is the minimum.

Since in a binary search tree, the **left tree always contains values less than the current node**.

Therefore, anytime we performed a deleteMin operation, we would be repeatedly hurting the balance of the tree by making the right subtree heavy.

In the worst case, where the deleteMins have depleted the left subtree, the right subtree would have at most twice as many elements as it should.

* 1. **Balanced Binary Search Tree**

**insert():** *O*(log *N*) average and worst

**deleteMin():** *O*(log *N*) average and worst

**findMin():** *O*(log *N*) average and worst

We can guarantee O(log n) time for these operations in the worst-case.

* 1. **Array Binary Heap Implementation**

**insert():** *O*(*1*) averageand*O*(log *N*) worst

**deleteMin():** *O*(log *N*) worst

**findMin():** *O*(*1*) average and worst

Using a binary search tree could be overkill because it supports a host of operations that are not required.

The basic data structure we will use will not require links and will support the insert() and deleteMin() operations in *O*(log *N*) worst-case time.

Insertion will actually take constant time on average, and our implementation will allow building a priority queue of *N* items in linear time, if no deletions intervene.

We will then discuss how to implement priority queues to support efficient merging.

This additional operation seems to complicate matters a bit and apparently requires the use of a linked structure.

**Comparison of Implementations**

None of the above methods is efficient (in the worst-case), unless the number of insertions is very small compared to the number of findMin/deleteMin operations.

There are two types of data structures that can guarantee O(log n) time for these operations in the worst-case.

1. One is a balanced binary search tree, which we will discuss in a few lectures.
2. The other (simpler) data structure is a ***heap***. (Note that this is not related to the usage of heap to describe free memory.)