5.1 Abstract Priority Queues

Recall that for Abstract Lists, the linear ordering is explicitly defined by the programmer. Abstract Data Types (ADTs) that allowed such data types include strings. There are data structures that allow some of the operations to be performed quickly; however, at least some operations will be, on average $O(n)$.

We then saw three specific ADTs (stacks, queues, and deques) that restricted the number of allowed operations and this allowed us to implement those operations in $\Theta(1)$ time.

The Abstract Sorted List describes implicitly ordered objects and we saw that two data structures for implementing this ADT were AVL trees and B+-trees. Most operations on these data structures can be executed in $O(\ln(n))$ time.

We will now look at a restriction of the Abstract Sorted List that most operations to run in $\Theta(1)$ time: the abstract priority queue.

5.1.1 Definition

A priority queue is an ADT where each object placed into the priority queue is associated with a *priority* where the priority is linearly or weakly ordered. Any object with a priority may be pushed onto a priority queue; however, pop and top operate to that object in the priority queue that has the *highest* priority.

We will use a slightly unintuitive, but common ordering of priorities:

1. The value 0 has the highest priority, and
2. If $x > y$, x has lower priority than y.

Thus, 5 has lower priority than 3 which has a lower priority than 1.

5.1.2 Operations

The operations on a priority queue are similar to that of a queue: top, push, and pop. Figure 1 demonstrates these operations under the assumption that the entries of the priority queue are stored in the linear order of the priority.

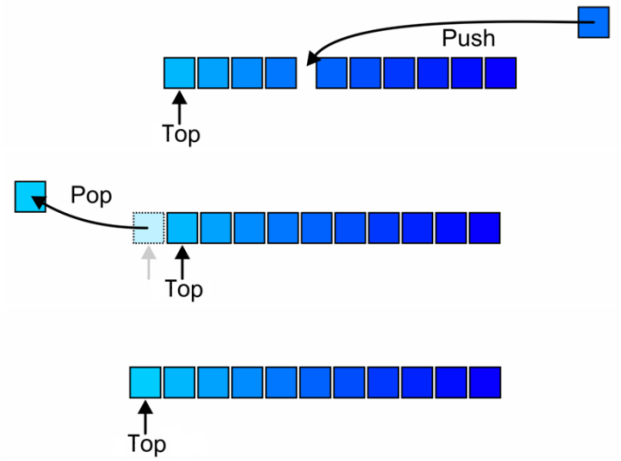

Figure 1. The operations on a priority queue.

5.1.3 Lexicographical Priority

Recall that with a lexicographical ordering (§ 2.1.6.1.2), we could say that (x_1, y_1) has higher priority than (x_2, y_2) if either $x_1 < x_2$, or $x_1 = x_2$ and $y_1 < y_2$.

5.1.4 Examples

The most common application of priority and priority queues for electrical and computer engineering is the queuing of processes for a processor. In any flavour of Unix, the highest priority for any user process is 0 and when you execute any Unix command at the prompt, it automatically runs at priority-level 0. It is possible, however, to lower the propriety of a process. Instead of running the command `% ./a.out`, it is possible to use the command nice, as in

```
% nice + 15 ./a.out
```

This reduces the priority of the process `./a.out` by 15.

Now, why would anyone *reduce* the priority of a process? Suppose one user is using an interactive process such as an editor—this is a program that has relatively fast responses without significant use of the processor: the user strikes a key, an interrupt is handled, the character is placed into the appropriate location, and the screen is updated.

A CPU-bound process is any process that will use the CPU at every single opportunity. Examples include circuit simulations, randomized testing for determining the stability of a system, or any other long-running process the main concern of which is to perform calculations. Suppose there are numerous CPU-bound processes running at the same time as an interactive process. Most people will be familiar with the results: press a key and wait half a second before it appears on the screen, or type a sentence and a second later, the entire sentence finally appears on the screen. By reducing the priority of the CPU-bound processes, those using interactive processes will not notice a negligible effect. Oddly enough, because the interactive processes use so little CPU time, the total run time of the CPU-bound processes is not significantly affected, either.

The Windows Task Manager under the Processes tab allows the user to change the priority of processes, as well, but these are given names:

Realtime, High, Above Normal, Normal, Below Normal, and Low.

5.1.5 Implementations

We will look at three different implementations of priority queues:

1. Multiple queues: an array of queues,
2. An AVL tree, and
3. Heaps.

The first two are based on data structures that we have already implemented.

5.1.5.1 Multiple Queues

Suppose that there are M priorities 0, 1, 2, ..., $M - 1$. In this case, we could keep an array of M queues and when an object comes in at priority k, we would place that object into the k^{th} queue. To pop an object or to access the top, we would search through the queues from 0 to $M - 1$ until we find the first non-empty queue.

```
#include <cassert>

template <typename Object, int M>
class Multiqueue {
    private:
        Queue<Object> queues[M];
        int count;

    public:
        Multiqueue();
        bool empty() const;
        Object top() const;
        void push( Object const &, int );
        Object pop();
};

template <typename Object, int M>
Multiqueue<Object>::Multiqueue():
count( 0 ) {
    // Empty constructor--the constructors for the queues are called by the compiler
}

template <typename Object, int M>
bool Multiqueue<Object>::empty() {
    return ( count == 0 );
}

template <typename Object, int M>
void Multiqueue<Object>::push( Object const &obj, int k ) {
    if ( k < 0 || k >= M ) {
        throw illegal_argument();
    }

    queues[k].push( obj );
    ++count;
}
```

```
template <typename Object, int M>
Object Multiqueue<Object>::top() const {
    if empty() {
        throw underflow();
    }

    for ( int k = 0; k < M; ++k ) {
        if ( !queues[k].empty() ) {
            return queues[k].front();
        }
    }

    assert( false );  // This should never happen
}

template <typename Object, int M>
Object Multiqueue<Object>::pop() {
    if empty() {
        throw underflow();
    }

    for ( int k = 0; k < M; ++k ) {
        if ( !queues[k].empty() ) {
            return queues[k].pop();
            --count;
        }
    }

    assert( false );  // This should never happen
}
```

The run time of push is $\Theta(1)$ while the run times of top and pop are $O(M)$. The memory requirements are $O(M + n)$. The most significant drawback, however, is that this data structure will not work if there are not a fixed and finite number of priorities.

5.1.5.2 AVL Trees

We could implement a priority queue using an associative AVL tree where the AVL tree is sorted based on the priority.

Suppose

```
template <typename K, typename R>
class AVL_tree;
```

represents an associative AVL tree where the entries are sorted on the key of type `K` and associated with that key is a record of type `R`. In this case, we could implement a priority queue allowing arbitrary integers as priorities as follows:

```
template <typename Object>
class AVL_priority_queue {
    private:
        AVL_tree<int, Object> tree;

    public:
        bool empty() const;
        Object top() const;
        void push( Object const &, int );
        Object pop();
};

template <typename Object>
bool AVL_priority_queue<Object>::empty() const {
    return tree.empty();
}

template <typename Object>
Object AVL_priority_queue<Object>::top() const {
    return tree.retrieve( tree.front() );
}

template <typename Object>
void AVL_priority_queue<Object>::push( Object const &obj, int k ) const {
    tree.insert( k, obj );
}

template <typename Object>
Object AVL_priority_queue<Object>::pop( Object const &obj, int k ) const {
    Object tp = tree.top();

    tree.erase( tree.front() );

    return tp;
}
```

5.1.5.3 Heaps

Unfortunately, now all operations are $\Theta(\ln(n))$ (why Θ and not O?). Because we are restricting the operations to one general insertion, a specific access (top) and a specific removal (pop), one may suspect we should be able to do better than this. Fortunately, there is an entire class of data structures collectively referred to as *heaps*. We will look at one of the more basic forms: the binary heap; however, there are numerous different flavours of heaps beyond the scope of this course, including

1. *d*-ary heaps,
2. leftist heaps,
3. binomial heaps,
4. Fibonacci heaps, and
5. Brodal heaps.