CS 2420: Priority Queue

Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Outline

- This topic will:
  - Review queues
  - Discuss the concept of priority and priority queues
  - Look at two simple implementations:
    - Arrays of queues
    - AVL trees
  - Introduce heaps, an alternative tree structure which has better run-time characteristics

Background

We have discussed Abstract Lists with explicit linear orders

- Arrays, linked lists, strings

We saw three cases which restricted the operations:

- Stacks, queues, deques

Following this, we looked at search trees for storing implicit linear orders: Abstract Sorted Lists

- Run times were generally $\Theta(\ln(n))$

We will now look at a restriction on an implicit linear ordering:

- Priority queues

Definition

With queues

- The order may be summarized by *first in, first out*

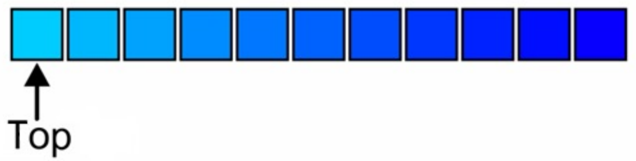
If each object is associated with a priority, we may wish to pop that object which has highest priority

With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

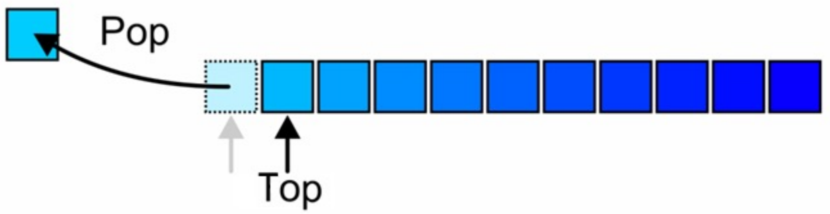
- The value 0 has the *highest* priority, and
- The higher the number, the lower the priority

Operations

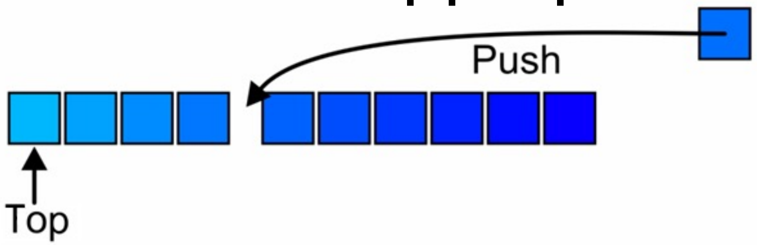
The top of a priority queue is the object with highest priority



Popping from a priority queue removes the current highest priority object:



Push places a new object into the appropriate place



Lexicographical Priority

Priority may also depend on multiple variables:

- Two values specify a priority: (a, b)
- A pair (a, b) has higher priority than (c, d) if:
  - $a < c$, or
  - $a = c$ and $b < d$

For example,

- (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)

Process Priority in Unix

This is the scheme used by Unix, *e.g.*,

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

reduces the priority of the execution of the routine `a.out` by 15

This allows the processor to be used by interactive programs

- This does not significantly affect the run-time of CPU-bound processes

Implementations

Our goal is to make the run time of each operation as close to $\Theta(1)$ as possible

We will look at two implementations using data structures we already know:

- Multiple queues—one for each priority
- An AVL tree

The next topic will be a more appropriate data structure: the heap

Multiple Queues

Assume there is a fixed number of priorities, say M

- Create an array of M queues
- Push a new object onto the queue corresponding to the priority
- Top and pop find the first empty queue with highest priority

Multiple Queues

```
template <typename Type, int M>
class Multiqueue {
    private:
        queue<Type> queue_array[M];
        int queue_size;
    public:
        Multiqueue();
        bool empty() const;
        Type top() const;
        void push( Type const &, int );
        Type pop();
};
```

```
template <typename Type, int M>
Multiqueue<Type>::Multiqueue():
queue_size( 0 ) {
    // The compiler allocates memory for the M queues
    // and calls the constructor on each of them
}

template <typename Type, int M>
bool Multiqueue<Type>::empty()  const{
    return ( queue_size == 0 );
}
```

Multiple Queues

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

    queue_array[pri].push( obj );
    ++queue_size;
}

template <typename Type, int M>
Type Multiqueue<Type>::top() const {
    for ( int pri = 0; pri < M; ++pri ) {
        if ( !queue_array[pri].empty() ) {
            return queue_array[pri].front();
        }
    }

    // The priority queue is empty
    throw underflow();
}
```

```
template <typename Type, int M>
Type Multiqueue<Type>::pop() {
    for ( int pri = 0; pri < M; ++pri ) {
        if ( !queue_array[pri].empty() ) {
            --queue_size;
            return queue_array[pri].pop();
        }
    }

    // The priority queue is empty
    throw underflow();
}
```

Multiple Queues

The run times are reasonable:

- Push is $\Theta(1)$
- Top and pop are both $\mathbf{O}(M)$

Unfortunately:

- It restricts the range of priorities
- The memory requirement is $\Theta(M + n)$

AVL Trees

We could simply insert the objects into an AVL tree where the order is given by the stated priority:

- Insertion is $\Theta(\ln(n))$ `void insert( Type const & );`
- Top is $\Theta(\ln(n))$ `Type front();`
- Remove is $\Theta(\ln(n))$ `bool remove( front() );`

There is significant overhead for maintaining both the tree and the corresponding balance

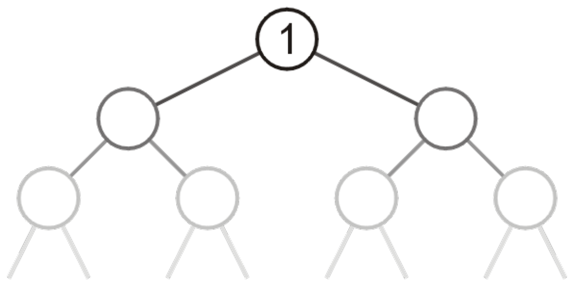
Heaps

Can we do better?

- That is, can we reduce some (or all) of the operations down to $\Theta(1)$?

The next topic defines a *heap*

- A tree with the top object at the root
- We will look at binary heaps
- Numerous other heaps exists:
  - d-ary heaps
  - Leftist heaps
  - Skew heaps
  - Binomial heaps
  - Fibonacci heaps
  - Bi-parental heaps



Summary

This topic:

- Introduced priority queues
- Considered two obvious implementations:
  - Arrays of queues
  - AVL trees
- Discussed the run times and claimed that a variation of a tree, a heap, can do better