Earlier, we discussed an ADT that supports enqueue and dequeue operations. "FIFO" stands for first in, first out.

Priority queue: A queue that must support
- insert
- deleteMin: return element of smallest value and remove from P.Q.
Could support delete Max - instead -.

Have to pick one in advance.

Use balanced binary search tree if you only care about big-Oh bounds.

$O(\log n)$ per op.
Binary heaps also called heap or \textit{min heap}.

Complete binary tree that has the heap-order property.

Complete: Full except last level which is filled left-to-right.
Heap-order: For any node \( v \), all children of \( v \) have greater value elements than \( v \) has.

Not a binary search tree
Implemented as an array.

Length = \( a + 1 \)

Root element at position 1.

Children for element at positions \( 2i \) and \( 2i + 1 \)

\( \Rightarrow \) Parent at \( \lfloor \frac{i}{2} \rfloor \)
Height is $\log n$. Want ops to take time proportional to height.
1) Make new node at next position for a complete tree, want to fill it with $x$, but heap-order property...
Call new node the hole.
Percolate up:

While hole has a parent > x, put parent element in hole & make the parent be the hole.

$O(\log n)$ time worst-case

$O(1)$ time "on average"

```java
public class BinaryHeap<E extends Comparable<? super E>> {
    public void insert(E x) {
        if (size == array.length - 1) {
            // enlarge array...
        }
        // percolate up
        int hole = ++size; // ++size increments and then evaluates to the new size
        // place x at position 0 so loop automatically terminates at root
        for (array[0] = x; x.compareTo(array[hole / 2]) < 0; hole /= 2) {
            array[hole] = array[hole / 2]; // move element down/hole up
        }
        array[hole] = x;
    }

    private int size;
    private E[] array;
}
```
delete Min:

Remove root element, leaving a hole.

Want to put the last element in the hole so we can remove its node, but element may be too large...

Percolate down.

While there is a smaller child of hole than x, move
public E deleteMin() {
    if (size == 0) {
        throw new UnderflowException();
    }

    E minElement = array[1];
    array[1] = array[size--]; // size-- evaluates to size and then decrements
    percolateDown(1);

    return minElement;
}

private void percolateDown(int hole) {
    int child;
    E tmp = array[hole];

    for (; hole * 2 <= size; hole = child) { // while there is a child
        child = hole * 2;
        if (child != size && array[child + 1].compareTo(array[child]) < 0) {
            child++; // set to smaller of the children
        }
        if (array[child].compareTo(tmp) < 0) { // child element is smaller
            array[hole] = array[child];
        } else {
            break;
        }
    }
    array[hole] = tmp;
}

O(log n) time worst-case & on average
Other operations

decrease key \( (p, \Delta) \):

\[ \begin{align*}
\uparrow & \quad \leftarrow \geq 0 \\
\text{position}
\end{align*} \]

//decrease value at position \( p \)

by \( \Delta \)

change value & percolate up

\( O(\log n) \) worst-case

increase key \( (p, \Delta) \):

change value & percolate down

\( O(\log n) \) worst-case
delete(e) :
  decrease key (p, \infty)
  delete Min()  \leftarrow \text{unsorted array}

build heap (elements) :
  // build a heap containing every member of elements
  could do n inserts in worst-case \( O(n \log n) \) time
Copy elements to heap array. Call percolateDown on each position from n to 1.

```java
public BinaryHeap(E[] elements) {
    size = elements.length;
    array = (E[]) new Comparable[(size + 1) * 2];

    int i = 1;
    for (E element : elements) { // a "for all" loop
        array[i++] = element;
    }
    makeHeap();
}

private void makeHeap() {
    for (int i = size / 2; i > 0; i--) { // skip childless nodes
        percolateDown(i);
    }
}
```
Thm: A perfect/full binary tree of height $h$ has $2^{h+1} - 1$ nodes, and the sum of heights is $2^{h+1} - 1 - (h-1)$.

Proof: $2^i$ nodes at height $h-i$.

Sum of heights is

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$= h + 2(h-1) + 4(h-2) + \ldots + 2^h$$

$$\Rightarrow 2S = 2h + 4(h-1) + 8(h-2) + \ldots + 2^h(h)$$

$$\Rightarrow S = h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h$$
\[ = (2^{h+1} - 1) - (h + 1) \]

\[ \Rightarrow \text{sum of heights for } n \text{ nodes is } \leq 2n. \]

\[ \Rightarrow \text{buildHeap takes } O(n) \text{ time} \]