Motivating examples

we are a bank and we want to manage a collection of our customers' information
- we'll be performing many adds and removes
- we'd like to be able to ask the collection to remove or get the information about the customer with the minimum bank account balance

another example: A shared Unix server has a list of print jobs to print. It wants to print them in chronological order, but each print job also has a priority, and higher-priority jobs always print before lower-priority jobs

another example: We are writing a parking search algorithm. It needs to search for the best parking spot near a location; it will enqueue all possible paths to potential open parking spots with priorities (based on various parameters), and try them in order

Some bad implementations

list: store all customers/jobs in an unordered list, remove min/max one by searching for it
- problem: expensive to search
sorted list: store all in a sorted list, then search it in $O(\log n)$ time with binary search
- problem: expensive to add/remove
binary search tree: store in a BST, search it in $O(\log n)$ time for the min (leftmost) element
- problem: tree could be unbalanced on nonrandom input
balanced BST
- problem: in practice, if the program has many adds/removes, it performs slowly on AVL trees and other balanced BSTs
- problem: removal always occurs from the left side, which unbalances the tree

Priority queue ADT (Weiss 21.1)

priority queue: an collection of ordered elements that provides fast access to the minimum (or maximum) element
- a mix between a queue and a BST
- usually implemented using a structure called a heap

priority queue operations:
- add
  - $O(1)$ average, $O(\log n)$ worst-case
- peek
  - returns the minimum element
  - $O(1)$
- removeMin - removes/returns minimum element
  - $O(\log n)$ worst-case
- isEmpty, clear, size, iterator
  - $O(1)$
Java's PriorityQueue class

```java
public class PriorityQueue<E> implements Queue<E>
    public void clear()
    public Iterator<E> iterator()
    public boolean offer(E o)
    public E peek()
    public E poll()
    public E remove()
```

Implementing a priority queue using a heap

Heap properties (Weiss 21.1.)

- 1. completeness (Weiss 6.3.1)
  - complete tree: every level is full except possibly the lowest level, which must be filled from left to right with no leaves to the right of a missing node (i.e., a node may not have any children until all of its possible siblings exist)

Heap shape:

Heap properties 2

- 2. heap ordering (Weiss 6.3.2)
  - a tree has heap ordering if \( P < X \) for every element \( X \) with parent \( P \)
  - in other words, in heaps, parents' element values are always smaller than those of their children
  - implies that minimum element is always the root
  - is every a heap a BST? Are any heaps BSTs? Are any BSTs heaps?
Which are min-heaps?

Which are max-heaps?

Heap height and runtime

Adding to a heap (Weiss 21.2)
Adding to a heap, cont’d.

to restore heap ordering property, the newly added element must be
shifted upward ("bubbled up") until it reaches its proper place

- bubble up (book: “percolate up”) by swapping with parent
- how many bubble-ups could be necessary, at most?

Adding to a max-heap

same operations, but must bubble up larger values to top

Heap practice problem

Draw the state of the heap tree after adding the following elements to it:

6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88

The peek operation

peek on a min-heap is trivial; because of the heap properties, the
minimum element is always the root

- peek is O(1)
- peek on a max-heap would be O(1) as well, but would return you the
maximum element and not the minimum one
Removing from a min-heap

- **min-heaps only support** `remove` **of the min element (the root)**
  - must remove the root while maintaining heap completeness and ordering properties
  - intuitively, the last leaf must disappear to keep it a heap
  - initially, just swap root with last leaf (we’ll fix it)

```
10
/  \
20   15
/  \
40   60
/ \  / \ \
50 700 65 99
```

Removing from heap, cont’d.

- must fix heap-ordering property: root is out of order
  - shift the root downward (“bubble down”) until it’s in place
  - swap it with its smaller child each time

```
10
/  \
20   15
/  \
40   60
/ \  / \ \
50 700 65 99
```

Turning any input into a heap (Weiss 21.3)

- we can quickly turn any complete tree of comparable elements into a heap with a `buildHeap` algorithm
- simply perform a “bubble down” operation on every node that is not a leaf, starting from the rightmost internal node and working back to the root
  - why does this `buildHeap` operation work?
  - how long does it take to finish? (big-Oh)

```
45
/  \
21   18
/  \
14   60 32  6
```

Array tree implementation

**corollary:** any complete binary tree can be implemented using an array (the example tree shown is not a heap)

<table>
<thead>
<tr>
<th>Item</th>
<th>Left Child</th>
<th>2*i</th>
<th>Right Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Sue</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Ann</td>
<td>-1</td>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>Jane</td>
<td>-1</td>
<td>17</td>
<td>-1</td>
</tr>
<tr>
<td>Mary</td>
<td>-1</td>
<td>19</td>
<td>-1</td>
</tr>
</tbody>
</table>
Array binary tree - parent

<table>
<thead>
<tr>
<th>Item</th>
<th>Parent</th>
<th>(i/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Joe</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sue</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Mike</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Sam</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Tom</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Ann</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Jane</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Mary</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor \]

Implementation of a heap

when implementing a complete binary tree, we actually can "cheat" and just use an array

- index of root = 1 (leave 0 empty for simplicity)
- for any node \( n \) at index \( i \),
  - index of \( n\text{.left} = 2i \)
  - index of \( n\text{.right} = 2i + 1 \)

Code for add (insert) method

```java
public void insert(AnyType x) {
    // grow array if needed
    if (currentSize >= array.length - 1)
        enlargeArray(array.length*2 + 1);

    // place element into heap at bottom
    int hole = ++currentSize;
    for (;
        hole > 1 & x.compareTo(array[hole/2]) < 0;
        hole /= 2 )
        array[hole] = array[hole/2];
}
```

Code for peek (findMin) method

```java
public AnyType peek() {
    if (isEmpty()) {
        throw new UnderflowException();
    }

    return array[1];
}
```
Code for remove (deleteMin) method

```java
public AnyType remove() {
    AnyType result = peek();

    // move last element of array up to root
    array[1] = array[currentSize--];

    percolateDown(1);
    return result;
}
```

The bubbleUp helper

```java
private void bubbleUp() {
    int index = this.size;

    while (hasParent(index) &&
           elements[index].compareTo(elements[parent(index)]) < 0) {
        // parent/child are out of order; swap them
        int parentIndex = parentIndex(index);
        swap(elements, index, parentIndex);
        index = parentIndex;
    }
}
```

// helpers
```java
private boolean hasParent(int i) { return i > 1; }
private int parentIndex(int i) { return i / 2; }
private AnyType parent(int i) { return elements[parentIndex(i)]; }
```

The percolateDown helper

```java
private void percolateDown(int hole) {
    int index = 1;
    while (hasLeftChild(index)) {
        int childIndex = leftIndex(index);
        if (hasRightChild(index) &&
            right(index).compareTo(left(index)) < 0) {
            childIndex = rightIndex(index);
        }

        if (elements[index].compareTo(elements[childIndex]) > 0) {
            swap(elements, index, childIndex);  // out of order
            index = childIndex;
        } else {
            break;
        }
    }
}
```

// helpers
```java
private int leftIndex(int i) { return i * 2; }
private int rightIndex(int i) { return i * 2 + 1; }
private boolean hasLeftChild(int i) { return leftIndex(i) <= size(); }
private boolean hasRightChild(int i) { return rightIndex(i) <= size(); }
```
Advantages of array heap

the "implicit representation" of a heap in an array makes several operations very fast
- add a new node at the end (O(1))
- from a node, find its parent (O(1))
- swap parent and child (O(1))
- a lot of dynamic memory allocation of tree nodes is avoided
- the algorithms shown usually have elegant solutions

```java
private void buildHeap() {
    for (int i = array.currentSize / 2; i > 0; i--) {
        bubbleDown(i);
    }
}
```

Heap sort

Heap sort: an algorithm to sort an array of N elements by turning the array into a heap, then doing a `removeN` N times
- the elements will come out in sorted order!
- we can put them into a new sorted array
- what is the runtime?

A max-heap

the heaps shown have been minimum heaps because the elements come out in ascending order
a max-heap is the same thing, but with the elements in descending order
Improved heap sort

the heap sort shown requires a second array

we can use a max-heap to implement an improved version of heap sort
that needs no extra storage

- Big(O(n log n)) runtime
- no external storage required!
- useful on low-memory devices
- elegant

Improved heap sort 1

use an array heap, but with 0 as the root index
max-heap state after buildHeap operation:

```
    97
   /\  \
  53 59
 /\  /\  \
26 41 58 31
 \
16 21 36
```

```
97 53 59 26 41 58 31 16 21 36
0  1  2  3  4  5  6  7  8  9 10 11 12 13
```

Improved heap sort 2

state after one remove operation:

- modified remove that moves element to end

```
    59
   /\  \
  53 58
 /\  /\  \
26 41 36 31
 \
16 21 97
```

```
59 53 58 26 41 36 31 16 21 97
0  1  2  3  4  5  6  7  8  9 10 11 12 13
```

Improved heap sort 3

state after two remove operations:

- notice that the largest elements are at the end
  (becoming sorted?)

```
    58
   /\  \
  53 36
 /\  /\  \
26 41 21 31
 \
16 59 97
```

```
58 53 36 26 41 21 31 16 59 97
0  1  2  3  4  5  6  7  8  9 10 11 12 13
```
Sorting algorithms review

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Average case (†)</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$n$</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$n$</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n^2$</td>
</tr>
</tbody>
</table>

† According to Knuth, the average growth rate of Insertion sort is about 0.9 times that of Selection sort and about 0.4 times that of Bubble Sort. Also, the average growth rate of Quicksort is about 0.74 times that of Mergesort and about 0.5 times that of Heapsort.