CSE 373: Data Structures and Algorithms

Lecture 13: Priority Queues (Heaps)
Motivating examples

- **Bandwidth management**: A router is connected to a line with limited bandwidth. If there is insufficient bandwidth, the router maintains a queue for incoming data such that the most important data will get forwarded first as bandwidth becomes available.

- **Printing**: A shared 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.

- **Algorithms**: We are writing a ghost AI algorithm for Pac-Man. It needs to search for the best path to find Pac-Man; it will enqueue all possible paths with priorities (based on guesses about which one will succeed), and try them in order.
Priority Queue ADT

• **priority queue**: A collection of elements that provides fast access to the minimum (or maximum) element
  – a mix between a queue and a BST

• basic priority queue operations:
  – **insert**: Add an element to the priority queue (priority matters)
  – **remove (i.e. deleteMin)**: Removes/returns minimum element
# Using `PriorityQueue`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PriorityQueue&lt;E&gt;()</code></td>
<td>constructs a <code>PriorityQueue</code> that orders the elements according to their <code>compareTo</code> (element type must implement <code>Comparable</code>)</td>
</tr>
<tr>
<td><code>add(element)</code></td>
<td>inserts the element into the <code>PriorityQueue</code></td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>removes and returns the element at the head of the queue</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>returns, but does not remove, the element at the head of the queue</td>
</tr>
</tbody>
</table>

```java
Queue<String> pq = new PriorityQueue<String>();
pq.add("Kona");
pq.add("Daisy");
```

— implements `Queue` interface
# Potential Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>delete-min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL Trees</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Potential Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)^*$</td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$ worst</td>
<td>$\Theta(n)$ worst</td>
</tr>
<tr>
<td>AVL Trees</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>
Heap properties

- **heap**: a tree with the following two properties:
  - 1. *completeness*
    - **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*

A tree has *heap ordering* if $P \leq X$ 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?
Which are min-heaps?
Which are max-heaps?

1. 30
   - 10
   - 20

2. 48
   - 21
   - 10
   - 14

3. 30
   - 33
   - 10
   - 17
   - 7
   - 3

4. 30
   - 10
   - 40
   - 11

5. 80
   - 25
   - 30
   - 24

6. 50
   - 30
   - 35
   - 28
   - 18
   - 9

Wrong!

Wrong!
Heap height and runtime

- height of a complete tree is always $\log n$, because it is always balanced
  - because of this, if we implement a priority queue using a heap, we can provide the $O(\log n)$ runtime required for the add and remove operations

$n$-node complete tree of height $h$:
$2^h \leq n \leq 2^{h+1} - 1$
$h = \lceil \log n \rceil$
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.left = 2i
    • index of n.right = 2i + 1
  – parent index?
public interface IntPriorityQueue {
    public void add(int value);
    public boolean isEmpty();
    public int peek();
    public int remove();
}

public class IntBinaryHeap implements IntPriorityQueue {
    private static final int DEFAULT_CAPACITY = 10;
    private int[] array;
    private int size;

    public IntBinaryHeap () {
        array = new int[DEFAULT_CAPACITY];
        size = 0;
    }
    ...
}
Adding to a heap

- when an element is added to a heap, it should be initially placed as the rightmost leaf (to maintain the completeness property)
  - heap ordering property becomes broken!
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 \textit{larger} values to top

```
5
/   \
3 18
```

```
16
/   \n3 18
```

```
16
/   \n3 5
```

```
18
/   \n16 11
```

```
18
/   \n3 11
```
Heap practice problem

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

  6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2
public void add(int value) {
    // grow array if needed
    if (size >= array.length - 1) {
        array = this.resize();
    }

    // place element into heap at bottom
    size++;
    int index = size;
    array[index] = value;

    bubbleUp();
}
The `bubbleUp` helper

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

    while (hasParent(index)
            && (parent(index) > array[index])) {
        // parent/child are out of order; swap them
        swap(index, parentIndex(index));
        index = parentIndex(index);
    }
}

// helpers
private boolean hasParent(int i) { return i > 1; }
private int parentIndex(int i) { return i/2; }
private int parent(int i) { return array[parentIndex(i)]; }
```
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
public int peek() {
    if (this.isEmpty()) {
        throw new IllegalStateException();
    }

    return array[1];
}