# Computer Science 331
## Priority Queues

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #27



## Outline







## Priority Queues

**Definition:** A *priority queue* is a data structure for maintaining a multiset $S$ of elements, each with an associated value called a *key*.

A *max-priority queue* supports the following operations:

- Insert($S$, *key*): Insert element with key *key* into $S$
- Maximum($S$): Report the largest key in $S$ without changing $S$
- Extract-Max($S$): Remove and return the element of $S$ with largest key
- Increase-Key($S$, $i$, *key*): Increase the key of the value indicated by $i$ to *key*

Reference: textbook Section 8.5 (offer, peek/element, remove/poll, no Increase-Key)





## Priority Queues

**Priority Queues in Java:**

- Class “PriorityQueue” in the Java Collections framework implements a “min-priority queue.”
  - implements the “Queue” interface, so calls to “Insert,” “Minimum,” and “Extract-Min” are implemented using calls to operations “add,” “element,” and “remove,” respectively.
  - There is no operation corresponding to “Increase-Key.”

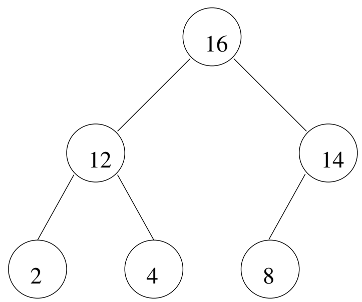
**Applications:**

- Scheduling: keys represent “priorities” used to determine order in which requests should be served

## Implementation

*Binary Heaps* are often used to implement priority queues.

*Example:* One representation of a max-priority queue including keys $S = \{2, 4, 8, 12, 14, 16\}$ is as follows:




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 12 | 14 | 2 | 4 | 8 | 9 | 3 |

length$(A) = 8$; heap-size$(A) = 6$





## Increase-Key

**Precondition:**

- $A$: Max-heap representing a max-priority queue $S$, containing elements of some ordered type
- $i$: Integer such that $0 \le i <$ heap-size$(A)$
- *key* : A value with the same type as elements of $S$

Let $\ell$ be the value originally stored at location $i$ of $A$.

**Postcondition:** If $key \ge \ell$ then $A$ represents the max-priority queue obtained by removing $\ell$ from $S$ and inserting *key*. $A$ is unchanged, otherwise.

**Exception:**

- `SmallValueException`, thrown if $key < \ell$
- `IndexOutofBoundsException`, thrown if $i < 0$ or $i \ge$ heap-size$(A)$





## Idea and Pseudocode

**Idea:** "Bubble" the new key up until it is in place.

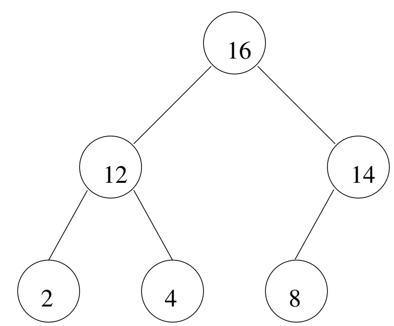
**Increase-Key**($A$, $i$, *key*)
  **if** (($i < 0$) **or** ($i \ge$ heap-size$(A)$)) **then**
    Throw `IndexOutOfBoundsException`
  **else if** $key < A[i]$ **then**
    Throw `SmallValueException`
  **else**
    $A[i] = key$; $j = i$
    **while** ($j > 0$) **and** ($A[\text{parent}(j)] < A[j]$) **do**
      Swap:
        $tmp = A[j]$; $A[j] = A[\text{parent}(j)]$; $A[\text{parent}(j)] = tmp$
      $i = \text{parent}(j)$
    **end while**
  **end if**





## Example

Consider the application of **Increase-Key**($A$, 4, 20) for $A$ as follows.

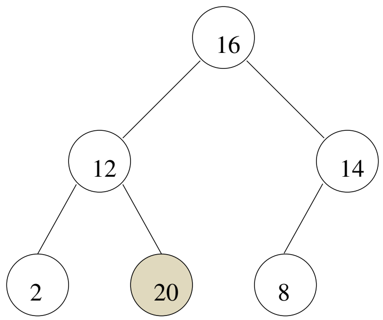



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 12 | 14 | 2 | 4 | 8 | | |

heap-size$(A) = 6$

# Example: First Step

$A$ is as follows after the initial replacement of $A[i]$.




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 12 | 14 | 2 | 20 | 8 | | |

heap-size$(A) = 6$

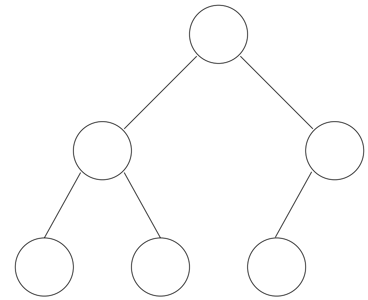




# Example: First Execution of Loop Body

$A$ is as follows after the first execution of the loop body.

- 



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

heap-size$(A) =$

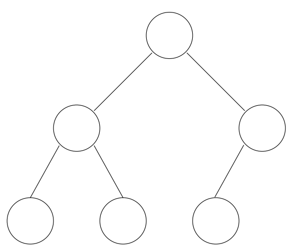




# Example: Second Execution of Loop Body

$A$ is as follows after the second execution of the loop body.

- 



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

heap-size$(A) =$

$i = 0$ : loop and function terminate





# Partial Correctness: Loop Invariant

Let $\ell$ be the value stored at location $i$ of the array $A$.

If the loop body is executed $k$ or more times then the following set $I(k)$ of properties is satisfied immediately after the $k$th execution of the loop body.

- $A$ represents the multiset obtained from the original multiset $S$ by removing $\ell$ and by inserting *key*
- $0 \leq j <$ heap-size$(A)$ and $j \leq \lfloor i/2^k \rfloor$
- For every integer $h$ such that $1 \leq h <$ heap-size$(A)$, if $h \neq j$ then $A[h] \leq A[\text{parent}(h)]$
- If $j > 0$ and left$(j) <$ heap-size$(A)$ then $A[\text{left}(j)] \leq A[\text{parent}(j)]$
- If $j > 0$ and right$(j) <$ heap-size$(A)$ then $A[\text{right}(j)] \leq A[\text{parent}(j)]$

## Partial Correctness: Application of Loop Invariant

**Exercises:**

1. Prove that this really is a loop invariant for the loop in this program (using induction on $k$).
2. Use the loop invariant to establish partial correctness of this program.





## Termination and Efficiency

**Loop Variant:** $f(n, i, j) = \lfloor \log_2(j + 1) \rfloor$

**Justification:**

-
  -
  -
-

**Application of Loop Variant:**

-
-
-





## Insert

**Precondition:**

$A$: Max-heap representing a max-priority queue $S$, containing elements from some ordered type

$key$ : A value with the same type as the elements of $S$

**Postcondition:** $A$ is a max-heap representing a max-priority queue $S \cup \{key\}$.

**Exception:** `QueueFullException`, thrown if there is no room left in $A$ (so $A$ is unchanged)





## Idea and Pseudocode

**Idea:**

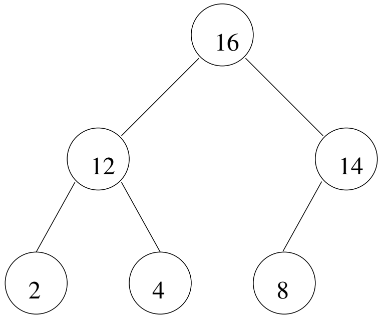
- Add the new key to the next available leaf on the last level.
- Use **Increase-Key** to reorganize the priority queue.

**Insert**($A$, $key$)

```
if heap-size(A) < length(A) then
   heap-size(A) = heap-size(A) + 1
   A[heap-size(A) − 1] = −∞
   Increase-Key(A, heap-size(A) − 1, key)
else
   Throw QueueFullException
end if
```

## Example

Consider the application of Insert($A$, 20) for $A$ as follows.




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 16 | 12 | 14 | 2 | 4 | 8 | | |

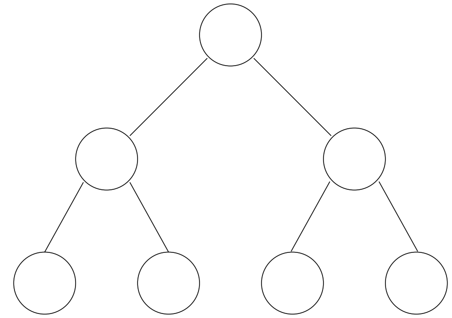
heap-size($A$) $= 6$





## Example: First Step

$A$ is as follows before the call to **Increase-Key**.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

heap-size($A$) $=$

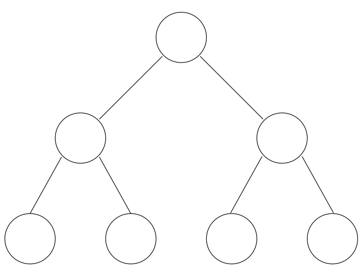




## Example: Completion

**Increase-Key**($A$, 6, 20):

- 
- 
- 



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

heap-size($A$) $=$





## Maximum and Extract-Max

**Idea:** The largest element of a max-heap is always located at the root.

**Maximum**($A$)
  **if** heap-size($A$) $> 0$ **then**
    **return** $A[0]$
  **else**
    Throw `EmptyQueueException`
  **end if**

The “Extract-Max” operation is the same as the “Delete-Max” operation used as part of Heap Sort.

- The operation can be implemented in the same way.

# Binomial and Fibonacci Heaps

*Introduction to Algorithms*, Chapter 19 and 20

Better than binary heaps if **Union** operation must be supported:

- creates a new heap consisting of all nodes in two input heaps

| Function | Binary Heap (worst-case) | Binomial Heap (worst-case) | Fib. Heap (amortized) |
|---|---|---|---|
| Insert | $\Theta(\log n)$ | $O(\log n)$ | $\Theta(1)$ |
| Maximum | $\Theta(1)$ | $O(\log n)$ | $\Theta(1)$ |
| Extract-Max | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| Increase-Key | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| Union | $\Theta(n)$ | $O(\log n)$ | $\Theta(1)$ |