Priority Queues and Heaps

September 9, 2014
Motivation: Sort a List of Numbers

Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$. 

Possible algorithm:

1. Store all the numbers in a data structure $D$.
2. Repeatedly find the smallest number in $D$, output it, and remove it.

To get $O(n \log n)$ running time, each find minimum step and each remove step must take $O(\log n)$ time.
Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
Motivation: Sort a List of Numbers

Sort

**INSTANCE:** Nonempty list \( x_1, x_2, \ldots, x_n \) of integers.

**SOLUTION:** A permutation \( y_1, y_2, \ldots, y_n \) of \( x_1, x_2, \ldots, x_n \) such that \( y_i \leq y_{i+1} \), for all \( 1 \leq i < n \).

- Possible algorithm:
  - Store all the numbers in a data structure \( D \).
  - Repeatedly find the smallest number in \( D \), output it, and remove it.

- To get \( O(n \log n) \) running time, each “find minimum” step and each “remove” step must take \( O(\log n) \) time.
Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
- Data structure must support insertion of a number, finding minimum, and deleting minimum.
Possible algorithm:

- Store all the numbers in a data structure $D$.
- Repeatedly find the smallest number in $D$, output it, and remove it.

Data structure must support insertion of a number, finding minimum, and deleting minimum.
Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
- Data structure must support insertion of a number, finding minimum, and deleting minimum.

  **List** Insertion and deletion take $O(1)$ time but finding minimum requires scanning the list and takes $\Omega(n)$ time.
Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure $D$.
  - Repeatedly find the smallest number in $D$, output it, and remove it.
- Data structure must support insertion of a number, finding minimum, and deleting minimum.

  **List** Insertion and deletion take $O(1)$ time but finding minimum requires scanning the list and takes $\Omega(n)$ time.

  **Sorted array** Finding minimum takes $O(1)$ time but insertion and deletion can take $\Omega(n)$ time in the worst case.
Priority Queue

- Store a set $S$ of elements, where each element $v$ has a priority value $\text{key}(v)$.
- Smaller key values $\equiv$ higher priorities.
- Operations supported:
  - find the element with smallest key
  - remove the smallest element
  - insert an element
  - delete an element
  - update the key of an element
- Element deletion and key update require knowledge of the position of the element in the priority queue.
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order**: For every element $v$ at a node $i$, the element $w$ at $i$’s parent satisfies $\text{key}(w) \leq \text{key}(v)$. 

W e can implement a heap in a pointer-based data structure. Alternatively, assume maximum number $N$ of elements is known in advance. Store nodes of the heap in an array. Node at index $i$ has children at indices $2i$ and $2i+1$ and parent at index $\lfloor i/2 \rfloor$. Index 1 is the root. How do you know that a node at index $i$ is a leaf? If $2i > n$, where $n$ is the current number of elements in the heap.
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order.** For every element $v$ at a node $i$, the element $w$ at $i$’s parent satisfies $\text{key}(w) \leq \text{key}(v)$.
- We can implement a heap in a pointer-based data structure.
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order.** For every element $v$ at a node $i$, the element $w$ at $i$’s parent satisfies $\text{key}(w) \leq \text{key}(v)$.
- We can implement a heap in a pointer-based data structure.
- Alternatively, assume maximum number $N$ of elements is known in advance.
- Store nodes of the heap in an array.
  - Node at index $i$ has children at indices $2i$ and $2i + 1$ and parent at index $\lfloor i/2 \rfloor$.
  - Index 1 is the root.
  - How do you know that a node at index $i$ is a leaf?
Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- **Heap order**: For every element $v$ at a node $i$, the element $w$ at $i$’s parent satisfies $\text{key}(w) \leq \text{key}(v)$.
- We can implement a heap in a pointer-based data structure.
- Alternatively, assume maximum number $N$ of elements is known in advance.
- Store nodes of the heap in an array.
  - Node at index $i$ has children at indices $2i$ and $2i + 1$ and parent at index $\lfloor i/2 \rfloor$.
  - Index 1 is the root.
  - How do you know that a node at index $i$ is a leaf? If $2i > n$, where $n$ is the current number of elements in the heap.
Example of a Heap

Each node’s key is at least as large as its parent’s.

Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.
Inserting an Element: Heapify-up

1. Insert new element at index $n + 1$.
2. Fix heap order using Heapify-up($H, n + 1$).

Heapify-up($H, i$):

If $i > 1$ then
  let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
  If $\text{key}[H[i]] < \text{key}[H[j]]$ then
    swap the array entries $H[i]$ and $H[j]$
    Heapify-up($H, j$)
  Endif
Endif
Endif
Example of Heapify-up

The Heapify-up process is moving element $v$ toward the root.

Figure 2.4 The Heapify-up process. Key 3 (at position 16) is too small (on the left). After swapping keys 3 and 11, the heap violation moves one step closer to the root of the tree (on the right).
Correctness of Heapify-up: Setup

To the root

\[
\begin{array}{c}
\left\lfloor \frac{i}{4} \right\rfloor \\
\left\lfloor \frac{i}{2} \right\rfloor \\
\end{array}
\]

\[x\]

\[y\]

Key

Index

September 9, 2014
Correctness of Heapify-up: Strategy

- Heapify-up($H, n+1$) invokes Heapify-up($H, \lfloor (n+1)/2 \rfloor$), which invokes Heapify-up($H, \lfloor (n+1)/4 \rfloor$), ... which invokes Heapify-up($H, 1$).
- It is possible that the heap property may be violated at any invocation and at more than one invocation.

- Two elements to prove strategy:
  1. Heap violation can occur at most one index.
  2. For every $i$, execution of Heapify-up($H, i$) pushes heap violation from index $i$ to index $\lfloor i/2 \rfloor$.
Correctness of Heapify-up: Strategy

- Heapify-up($H, n+1$) invokes Heapify-up($H, [(n+1)/2]$), which invokes Heapify-up($H, [(n+1)/4]$), ... which invokes Heapify-up($H, 1$).
- It is possible that the heap property may be violated at any invocation and at more than one invocation.

- Let us be precise and make a formal definition: a heap violation occurs at index $i$ of $H$ if $\text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor])$.
Correctness of Heapify-up: Strategy

- Heapify-up($H, n + 1$) invokes Heapify-up($H, [(n + 1)/2]$), which invokes Heapify-up($H, [(n + 1)/4]$), ... which invokes Heapify-up($H, 1$).
- It is possible that the heap property may be violated at any invocation and at more than one invocation.
- Let us be precise and make a formal definition: a heap violation occurs at index $i$ of $H$ if key($H[i]$) < key($H[\lfloor i/2 \rfloor]$).
- What is the precise statement we want to prove?
  - After Heapify-up($H, n + 1$) returns, $H$ is a heap.
  - For every $1 \leq i \leq n + 1$, after Heapify-up($H, i$) returns, $H$ is a heap.
Correctness of Heapify-up: Strategy

- Heapify-up\((H, n + 1)\) invokes Heapify-up\((H, [(n + 1)/2])\), which invokes Heapify-up\((H, [(n + 1)/4])\), \ldots which invokes Heapify-up\((H, 1)\).
- It is possible that the heap property may be violated at any invocation and at more than one invocation.
- Let us be precise and make a formal definition: a *heap violation occurs at index* \(i\) *of* \(H\) *if* \(\text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor])\).*
- What is the precise statement we want to prove?
  - After Heapify-up\((H, n + 1)\) returns, \(H\) is a heap.
  - For every \(1 \leq i \leq n + 1\), after Heapify-up\((H, i)\) returns, \(H\) is a heap.
  - Both statements don’t say anything about where heap violations occur!
Correctness of Heapify-up: Strategy

- Heapify-up($H, n + 1$) invokes Heapify-up($H, [(n + 1)/2]$), which invokes Heapify-up($H, [(n + 1)/4]$), ... which invokes Heapify-up($H, 1$).

- It is possible that the heap property may be violated at any invocation and at more than one invocation.

- Let us be precise and make a formal definition: a heap violation occurs at index $i$ of $H$ if $\text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor])$.

- What is the precise statement we want to prove?
  - After Heapify-up($H, n + 1$) returns, $H$ is a heap.
  - For every $1 \leq i \leq n + 1$, after Heapify-up($H, i$) returns, $H$ is a heap.
  - Both statements don’t say anything about where heap violations occur!
  - A better statement to prove: for every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.
Correctness of Heapify-up: Strategy

- Heapify-up($H, n+1$) invokes Heapify-up($H, [(n+1)/2]$), which invokes Heapify-up($H, [(n+1)/4]$), ... which invokes Heapify-up($H, 1$).
- It is possible that the heap property may be violated at any invocation and at more than one invocation.

- Let us be precise and make a formal definition: a heap violation occurs at index $i$ of $H$ if $\text{key}(H[i]) < \text{key}(H[(i/2)])$.

- What is the precise statement we want to prove?
  - After Heapify-up($H, n+1$) returns, $H$ is a heap.
  - For every $1 \leq i \leq n+1$, after Heapify-up($H, i$) returns, $H$ is a heap.
  - Both statements don’t say anything about where heap violations occur!
  - A better statement to prove: for every $1 \leq i \leq n+1$, if the heap violation occurs only at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.

- Two elements to proof strategy:
  1. Heap violation can occur at most one index.
  2. For every $i$, execution of Heapify-up($H, i$) pushes heap violation from index $i$ to index $[i/2]$. 
Correctness of Heapify-up: Strategy

- Heapify-up\((H, n + 1)\) invokes Heapify-up\((H, \lfloor (n + 1)/2 \rfloor)\), which invokes Heapify-up\((H, \lfloor (n + 1)/4 \rfloor)\), ... which invokes Heapify-up\((H, 1)\).

- It is possible that the heap property may be violated at any invocation and at more than one invocation.

- Let us be precise and make a formal definition: a heap violation occurs at index \(i\) of \(H\) if \(\text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor])\).

- What is the precise statement we want to prove?
  - After Heapify-up\((H, n + 1)\) returns, \(H\) is a heap.
  - For every \(1 \leq i \leq n + 1\), after Heapify-up\((H, i)\) returns, \(H\) is a heap.
  - Both statements don’t say anything about where heap violations occur!
  - A better statement to prove: for every \(1 \leq i \leq n + 1\), if the heap violation occurs only at \(i\), then after Heapify-up\((H, i)\) returns, \(H\) is a heap.

- Two elements to proof strategy:
  1. Heap violation can occur at most one index.
  2. For every \(i\), execution of Heapify-up\((H, i)\) pushes heap violation from index \(i\) to index \(\lfloor i/2 \rfloor\).
Correctness of Heapify-up: Proof by Induction

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.
Correctness of Heapify-up: Proof by Induction

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after \text{Heapify-up}(H, i)$ returns, $H$ is a heap.

- Base case: $i = 1$. 

Correctness of Heapify-up: Proof by Induction

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.

- Base case: $i = 1$. Statement is trivially true, since node with index 1 has no parent; hence, no heap violation can occur at this index.
Correctness of Heapify-up: Proof by Induction

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.

- Base case: $i = 1$. Statement is trivially true, since node with index 1 has no parent; hence, no heap violation can occur at this index.
- Inductive hypothesis: (suggested by statement we need to prove)
Correctness of Heapify-up: Proof by Induction

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after Heapify-up$(H, i)$ returns, $H$ is a heap.

- **Base case:** $i = 1$. Statement is trivially true, since node with index 1 has no parent; hence, no heap violation can occur at this index.
- **Inductive hypothesis:** (suggested by statement we need to prove) If the heap violation occurs only at $\lfloor i/2 \rfloor$, then after Heapify-up$(H, \lfloor i/2 \rfloor)$ returns, $H$ is a heap.
**Correctness of Heapify-up: Proof by Induction**

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after $\text{Heapify-up}(H, i)$ returns, $H$ is a heap.

- **Base case:** $i = 1$. Statement is trivially true, since node with index 1 has no parent; hence, no heap violation can occur at this index.
- **Inductive hypothesis:** (suggested by statement we need to prove) If the heap violation occurs only at $\lfloor i/2 \rfloor$, then after $\text{Heapify-up}(H, \lfloor i/2 \rfloor)$ returns, $H$ is a heap.
- **Inductive step:** What do we need to prove?
Correctness of Heapify-up: Proof by Induction

To prove:

For every $1 \leq i \leq n + 1$, if the heap violation occurs only at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.

- **Base case:** $i = 1$. Statement is trivially true, since node with index 1 has no parent; hence, no heap violation can occur at this index.
- **Inductive hypothesis:** (suggested by statement we need to prove) If the heap violation occurs only at $\lfloor i/2 \rfloor$, then after Heapify-up($H, \lfloor i/2 \rfloor$) returns, $H$ is a heap.
- **Inductive step:** What do we need to prove? If the heap violation occurs only at $i$, then after the swap statement in Heapify-up($H, i$), $H$ is a heap or the heap violation occurs only at $\lfloor i/2 \rfloor$. 
Correctness of Heapify-up: Inductive Step

Starting point: Heap violation occurs only at $i$.
Goal: Before Heapify-up($H, \lfloor i/2 \rfloor$): $H$ is a heap or the heap violation occurs only at $\lfloor i/2 \rfloor$.

Heapify-up($H, i$):
If $i > 1$ then
  let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
  If $\text{key}[H[i]] < \text{key}[H[j]]$ then
    swap the array entries $H[i]$ and $H[j]$
    Heapify-up($H, j$)
  Endif
Endif
Starting point: Heap violation occurs only at \( i \).

Goal: Before \( \text{Heapify-up}(H, \lfloor i/2 \rfloor) \): \( H \) is a heap or the heap violation occurs only at \( \lfloor i/2 \rfloor \).

What is the situation after the swap statement in \( \text{Heapify-up}(H, i) \)?
Correctness of Heapify-up: Inductive Step

Starting point: Heap violation occurs only at $i$.

Goal: Before Heapify-up($H, \lfloor i/2 \rfloor$): $H$ is a heap or the heap violation occurs only at $\lfloor i/2 \rfloor$.

What is the situation after the swap statement in Heapify-up($H, i$)?

Heapify-up($H, i$):

If $i > 1$ then
  let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
  If key[$H[i]$] < key[$H[j]$] then
    swap the array entries $H[i]$ and $H[j]$
    Heapify-up($H, j$)
  Endif
Endif
Correctness of Heapify-up: Inductive Step

- Starting point: Heap violation occurs only at $i$.
- Goal: Before Heapify-up($H, \lfloor i/2 \rfloor$): $H$ is a heap or the heap violation occurs only at $\lfloor i/2 \rfloor$.
- What is the situation after the swap statement in Heapify-up($H, i$)?
- How do we show that $x < u, v$? Difficult since we do not know anything about relationship of $x$ with respect to $u$ and $v$ from the proof so far.

Heapify-up($H, i$):

If $i > 1$ then

let $j = \text{parent}(i) = \lfloor i/2 \rfloor$

If $\text{key}[H[i]] < \text{key}[H[j]]$ then

swap the array entries $H[i]$ and $H[j]$

Heapify-up($H, j$)

Endif

Endif
Correctness of Heapify-up: Inductive Step

Starting point: Heap violation occurs only at $i$.

Goal: Before `Heapify-up(H, ⌊i/2⌋)`: $H$ is a heap or the heap violation occurs only at $⌊i/2⌋$.

What is the situation after the swap statement in `Heapify-up(H, i)`?

How do we show that $x < u, v$? Difficult since we do not know anything about relationship of $x$ with respect to $u$ and $v$ from the proof so far.

Let us try definition from the textbook (slightly modified).
Correctness of Heapify-up: Strategy

- Modified definition from textbook: \( H \text{ is too small at index } i \) if
  1. \( \text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor]) \)

Correctness of Heapify-up: Strategy

- Modified definition from textbook: *H is too small at index i* if
  
  1. $\text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor])$ and
  2. there is a value $\alpha$ such that increasing $\text{key}(H[i])$ to $\alpha$ makes $H$ a heap.
Correctness of Heapify-up: Strategy

- Modified definition from textbook: *H is too small at index i* if
  1. \( \text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor]) \) and
  2. there is a value \( \alpha \) such that increasing \( \text{key}(H[i]) \) to \( \alpha \) makes \( H \) a heap.

- We want to prove for every \( 1 \leq i \leq n + 1 \), if \( H \) is too small at \( i \), then after Heapify-up\((H, i)\) returns, \( H \) is a heap.
Correctness of Heapify-up: Strategy

- Modified definition from textbook: $H$ is too small at index $i$ if
  1. $\text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor])$ and
  2. there is a value $\alpha$ such that increasing $\text{key}(H[i])$ to $\alpha$ makes $H$ a heap.

- We want to prove for every $1 \leq i \leq n + 1$, if $H$ is too small at $i$, then after Heapify-up($H$, $i$) returns, $H$ is a heap.

- Two elements to proof strategy:
  1. Heap violation can occur at most one index and we know how to fix it.
  2. For every $i$, execution of Heapify-up($H$, $i$) pushes heap violation from index $i$ to index $\lfloor i/2 \rfloor$. 
Correctness of Heapify-up: Strategy

- Modified definition from textbook: \( H \) is too small at index \( i \)
  - 1. \( \text{key}(H[i]) < \text{key}(H[\lfloor i/2 \rfloor]) \) and
  - 2. there is a value \( \alpha \) such that increasing \( \text{key}(H[i]) \) to \( \alpha \) makes \( H \) a heap.

- We want to prove for every \( 1 \leq i \leq n + 1 \), if \( H \) is too small at \( i \), then after Heapify-up\((H, i)\) returns, \( H \) is a heap.

- Two elements to proof strategy:
  - 1. Heap violation can occur at most one index and we know how to fix it.
  - 2. For every \( i \), execution of Heapify-up\((H, i)\) pushes heap violation from index \( i \) to index \( \lfloor i/2 \rfloor \).

To prove:

For every \( 1 \leq i \leq n + 1 \), if \( H \) is too small at \( i \), then after Heapify-up\((H, i)\) returns, \( H \) is a heap.
Correctness of Heapify-up: Base Case

Base case: $i = 1$.

Heapify-up($H, i$):

If $i > 1$ then

let $j = \text{parent}(i) = \lfloor i/2 \rfloor$

If $\text{key}[H[i]] < \text{key}[H[j]]$ then

swap the array entries $H[i]$ and $H[j]$

Heapify-up($H, j$)

Endif

Endif
Correctness of Heapify-up: Base Case

Base case: $i = 1$.
  - $H$ is too small at 1.

Heapify-up($H, i$):
  If $i > 1$ then
    let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
    If $\text{key}[H[i]] < \text{key}[H[j]]$ then
      swap the array entries $H[i]$ and $H[j]$
      Heapify-up($H, j$)
    Endif
  Endif
Correctness of Heapify-up: Base Case

Base case: \( i = 1 \).

- \( H \) is too small at 1.
- There is a value \( \alpha \geq \text{key}(H[1]) \) such that increasing \( \text{key}(H[1]) \) to \( \alpha \) makes \( H \) a heap.

Heapify-up(\( H, i \)):

If \( i > 1 \) then

let \( j = \text{parent}(i) = \lfloor i/2 \rfloor \)
If \( \text{key}[H[i]] < \text{key}[H[j]] \) then

swap the array entries \( H[i] \) and \( H[j] \)

Heapify-up(\( H, j \))

Endif

Endif

\( \alpha \geq y \)

Is \( y < u,v? \)
Correctness of \textit{Heapify-up}: Base Case

- **Base case:** \( i = 1 \).
  - \( H \) is too small at 1.
  - There is a value \( \alpha \geq \text{key}(H[1]) \) such that increasing \( \text{key}(H[1]) \) to \( \alpha \) makes \( H \) a heap.
  - \( \text{key}(H[1]) \leq \alpha \leq \text{key}(H[2]), \text{key}(H[3]) \) \( \implies \) \( H \) is a heap.

\texttt{Heapify-up}(H,i):
- If \( i > 1 \) then
  - let \( j = \text{parent}(i) = \lfloor i/2 \rfloor \)
  - If \( \text{key}[H[i]] < \text{key}[H[j]] \) then
    - swap the array entries \( H[i] \) and \( H[j] \)
    - \texttt{Heapify-up}(H,j)
  - Endif
- Endif
Correctness of Heapify-up: Inductive Hypothesis

To prove:

For every $1 \leq i \leq n+1$, if $H$ is too small at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.
Correctness of Heapify-up: Inductive Hypothesis

To prove:
For every $1 \leq i \leq n+1$, if $H$ is too small at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.

- Inductive hypothesis: (suggested by statement we need to prove)
Correctness of Heapify-up: Inductive Hypothesis

To prove:

For every $1 \leq i \leq n+1$, if $H$ is too small at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.

- Inductive hypothesis: (suggested by statement we need to prove) If $H$ is too small at $\lfloor i/2 \rfloor$, then after Heapify-up($H, \lfloor i/2 \rfloor$) returns, $H$ is a heap.
Correctness of Heapify-up: Inductive Hypothesis

To prove:

For every $1 \leq i \leq n+1$, if $H$ is too small at $i$, then after $\text{Heapify-up}(H, i)$ returns, $H$ is a heap.

- Inductive hypothesis: (suggested by statement we need to prove) If $H$ is too small at $\lfloor i/2 \rfloor$, then after $\text{Heapify-up}(H, \lfloor i/2 \rfloor)$ returns, $H$ is a heap.
- Inductive step: What do we need to prove?
Correctness of Heapify-up: Inductive Hypothesis

To prove:

For every $1 \leq i \leq n+1$, if $H$ is too small at $i$, then after $\text{Heapify-up}(H, i)$ returns, $H$ is a heap.

▶ Inductive hypothesis: (suggested by statement we need to prove) If $H$ is too small at $\lfloor i/2 \rfloor$, then after $\text{Heapify-up}(H, \lfloor i/2 \rfloor)$ returns, $H$ is a heap.

▶ Inductive step: What do we need to prove? If $H$ is too small at $i$, then after the swap statement in $\text{Heapify-up}(H, i)$, $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$. 

**Correctness of Heapify-up: Inductive Step**

Starting point: $H$ is too small at $i$.

Goal: Before `Heapify-up(H, \lfloor i/2 \rfloor)$: $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$.

---

**Heapify-up(H,i):**

If $i > 1$ then

let $j = \text{parent}(i) = \lfloor i/2 \rfloor$

If $\text{key}[H[i]] < \text{key}[H[j]]$ then

swap the array entries $H[i]$ and $H[j]$

Heapify-up(H,j)

Endif

Endif
Correctness of Heapify-up: Inductive Step

Starting point: $H$ is too small at $i$.
Goal: Before Heapify-up$(H, \lfloor i/2 \rfloor)$: $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$.
What is the situation after the swap statement in Heapify-up$(H, i)$?

Heapify-up$(H, i)$:
If $i > 1$ then
let $j =$ parent$(i) = \lfloor i/2 \rfloor$
If key$[H[i]] < $key$[H[j]]$ then
swap the array entries $H[i]$ and $H[j]$
Heapify-up$(H, j)$
Endif
Endif
Correctness of Heapify-up: Inductive Step

Starting point: $H$ is too small at $i$.

Goal: Before Heapify-up($H, \lfloor i/2 \rfloor$): $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$.

What is the situation after the swap statement in Heapify-up($H, i$)?

Heapify-up($H, i$):

If $i > 1$ then

let $j = \text{parent}(i) = \lfloor i/2 \rfloor$

If $\text{key}[H[i]] < \text{key}[H[j]]$ then

swap the array entries $H[i]$ and $H[j]$

Heapify-up($H, j$)

Endif

Endif
Correctness of Heapify-up: Inductive Step

Starting point: \( H \) is too small at \( i \).

Goal: Before Heapify-up\((H, \lfloor i/2 \rfloor)\): \( H \) is a heap or \( H \) is too small at \( \lfloor i/2 \rfloor \).

What is the situation after the swap statement in Heapify-up\((H, i)\)?

How do we show that \( x < u, v \)?
Correctness of Heapify-up: Inductive Step

Starting point: $H$ is too small at $i$.

Goal: Before Heapify-up($H$, $[i/2]$): $H$ is a heap or $H$ is too small at $[i/2]$.

What is the situation after the swap statement in Heapify-up($H$, $i$)?

How do we show that $x < u, v$?

- Use the fact that $H$ is too small at $i$: there is an $\alpha \geq x$ such that increasing $x$ to $\alpha$ makes $H$ a heap.
Correctness of Heapify-up: Inductive Step

- Starting point: $H$ is too small at $i$.
- Goal: Before Heapify-up($H$, $\lfloor i/2 \rfloor$): $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$.
- What is the situation after the swap statement in Heapify-up($H$, $i$)?
- How do we show that $x < u, v$?
  - Use the fact that $H$ is too small at $i$: there is an $\alpha \geq x$ such that increasing $x$ to $\alpha$ makes $H$ a heap.
  - Therefore, $x \leq \alpha \leq u, v$.

Heapify-up($H$, $i$):

If $i > 1$ then

let $j = \text{parent}(i) = \lfloor i/2 \rfloor$

If $\text{key}[H[i]] < \text{key}[H[j]]$ then

swap the array entries $H[i]$ and $H[j]$

Heapify-up($H$, $j$)

Endif

Endif
**Correctness of Heapify-up: Inductive Step**

Starting point: $H$ is too small at $i$.

Goal: Before Heapify-up($H$, $\lfloor i/2 \rfloor$): $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$.

What is the situation after the swap statement in Heapify-up($H$, $i$)?

How do we show that $x < u, v$?

- Use the fact that $H$ is too small at $i$: there is an $\alpha \geq x$ such that increasing $x$ to $\alpha$ makes $H$ a heap.
- Therefore, $x \leq \alpha \leq u, v$.

Now if $H$ is not a heap, why is it too small at $\lfloor i/2 \rfloor$?

---

Heapify-up($H$, $i$):

If $i > 1$ then

let $j = \text{parent}(i) = \lfloor i/2 \rfloor$

If key[$H[i]$] < key[$H[j]$] then

swap the array entries $H[i]$ and $H[j]$

Heapify-up($H$, $j$)

Endif

Endif
Correctness of Heapify-up: Inductive Step

- **Starting point:** $H$ is too small at $i$.
- **Goal:** Before Heapify-up($H, \lfloor i/2 \rfloor$): $H$ is a heap or $H$ is too small at $\lfloor i/2 \rfloor$.
- **What is the situation after the swap statement in Heapify-up($H, i$)?**
- **How do we show that $x < u, v$?**
  - Use the fact that $H$ is too small at $i$: there is an $\alpha \geq x$ such that increasing $x$ to $\alpha$ makes $H$ a heap.
  - Therefore, $x \leq \alpha \leq u, v$.
- **Now if $H$ is not a heap, why is it too small at $\lfloor i/2 \rfloor$?** Increasing $y$ to $x$ makes $H$ a heap!

Heapify-up($H, i$):

If $i > 1$ then

- let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
- If $\text{key}[H[i]] < \text{key}[H[j]]$ then
  - swap the array entries $H[i]$ and $H[j]$
  - Heapify-up($H, j$)
- Endif
- Endif
For every $1 \leq i \leq n + 1$, we have shown if $H$ is too small at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.
Correctness of Heapify-up: Completing the Proof

- For every $1 \leq i \leq n + 1$, we have shown if $H$ is too small at $i$, then after Heapify-up($H$, $i$) returns, $H$ is a heap.
- We know that before Heapify-up($H$, $n + 1$), $H$ is too small at $n + 1$. Why?
Correctness of Heapify-up: Completing the Proof

- For every $1 \leq i \leq n + 1$, we have shown if $H$ is too small at $i$, then after Heapify-up($H, i$) returns, $H$ is a heap.
- We know that before Heapify-up($H, n + 1$), $H$ is too small at $n + 1$. Why?
- Therefore, setting $i = n + 1$, we have that Heapify-up($H, n + 1$) creates a heap on all $n + 1$ elements.
Running time of Heapify-up

Heapify-up(H, i):
   If \( i > 1 \) then
      let \( j = \text{parent}(i) = \lfloor i/2 \rfloor \)
      If key[H[i]] < key[H[j]] then
         swap the array entries H[i] and H[j]
         Heapify-up(H, j)
      Endif
   Endif

Running time of Heapify-up(i)
Running time of Heapify-up

Heapify-up(H,i):
   If $i > 1$ then
      let $j = \text{parent}(i) = \lfloor i/2 \rfloor$
      If key[H[i]] < key[H[j]] then
         swap the array entries H[i] and H[j]
         Heapify-up(H,j)
      Endif
   Endif

Running time of Heapify-up(i) is $O(\log i)$.

$$T(i) \leq \begin{cases} 
   T(\lfloor i/2 \rfloor) + O(1) & \text{if } i > 1 \\
   O(1) & \text{if } i = 1 
\end{cases}$$
Deleting an Element: Heapify-down

- Suppose $H$ has $n + 1$ elements.
1. Delete element at $H[i]$ by moving element at $H[n+1]$ to $H[i]$.
2. If element at $H[i]$ is too small, fix heap order using Heapify-up($H, i$).
3. If element at $H[i]$ is too large, fix heap order using Heapify-down($H, i$).

---

Heapify-down($H, i$):

Let $n = \text{length}(H)$

If $2i > n$ then

Terminate with $H$ unchanged

Else if $2i < n$ then

Let left $= 2i$, and right $= 2i + 1$

Let $j$ be the index that minimizes key[$H[\text{left}]$] and key[$H[\text{right}]$]

Else if $2i = n$ then

Let $j = 2i$

Endif

If key[$H[j]$] < key[$H[i]$] then

swap the array entries $H[i]$ and $H[j]$

Heapify-down($H, j$)

Endif
The Heapify-down process is moving element $w$ down, toward the leaves.

**Figure 2.5** The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down

Statement to prove: for every $i \leq j \leq n$, if $H$ is too big at $j$ then Heapify-down($H$, $j$) creates a heap.

Proof by reverse induction on $j$ from $n$ down to $i$.

Base case: $2j > n$. If decreasing key($H[j]$) to $\alpha$ makes $H$ a heap, then key($H[j]$) $\leq$ key($H[\lfloor j/2 \rfloor]$).

Figure 2.5 The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down

- $H$ is *too big at $j$* if there is a value $\alpha \leq \text{key}(H[j])$ such that decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap. (Note: at start, $H$ is indeed too big at $i$.)

Figure 2.5 The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down

- **H is too big at j** if there is a value $\alpha \leq \text{key}(H[j])$ such that decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap. (Note: at start, $H$ is indeed too big at $i$.)

- **Statement to prove:** for every $i \leq j \leq n$, if $H$ is too big at $j$ then Heapify-down($H, j$) creates a heap.

- **Proof by reverse induction** on $j$ from $n$ down to $i$. 

---

Figure 2.5 The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down

- $H$ is too big at $j$ if there is a value $\alpha \leq \text{key}(H[j])$ such that decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap. (Note: at start, $H$ is indeed too big at $i$.)
- Statement to prove: for every $i \leq j \leq n$, if $H$ is too big at $j$ then Heapify-down($H, j$) creates a heap.
- Proof by reverse induction on $j$ from $n$ down to $i$.
- Base case:
Correctness of Heapify-down

- $H$ is *too big at $j$* if there is a value $\alpha \leq \text{key}(H[j])$ such that decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap. (Note: at start, $H$ is indeed too big at $i$.)

- Statement to prove: for every $i \leq j \leq n$, if $H$ is too big at $j$ then Heapify-down$(H, j)$ creates a heap.

- Proof by reverse induction on $j$ from $n$ down to $i$.

- Base case: $2j > n$. If decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap, then $\text{key}(H[j]) \leq \text{key}(H[\lfloor j/2 \rfloor])$. 

*Figure 2.5* The Heapify-down process. Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down: Inductive Step

The Heapify-down process is moving element $w$ down, toward the leaves.

Figure 2.5 The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down: Inductive Step

- $H$ is too big at $j$ if there is a value $\alpha \leq \text{key}(H[j])$ such that decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap.

![Diagram showing the heapify-down process](Image)

Figure 2.5 The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Correctness of Heapify-down: Inductive Step

- **H is too big at** $j$ **if there is a value** $\alpha \leq \text{key}(H[j])$ **such that decreasing** $\text{key}(H[j])$ **to** $\alpha$ **makes** $H$ **a heap.**
- **Inductive hypothesis (two parts):**
  - If $H$ too big at $2j$, then Heapify-down($H, 2j$) creates a heap.
  - If $H$ is too big at $2j + 1$, then Heapify-down($H, 2j + 1$) creates a heap.
Correctness of Heapify-down: Inductive Step

- **H is too big at j** if there is a value $\alpha \leq \text{key}(H[j])$ such that decreasing $\text{key}(H[j])$ to $\alpha$ makes $H$ a heap.
- **Inductive hypothesis (two parts):**
  - If $H$ too big at $2j$, then $\text{Heapify-down}(H, 2j)$ creates a heap.
  - If $H$ is too big at $2j + 1$, then $\text{Heapify-down}(H, 2j + 1)$ creates a heap.
- **Start of inductive step:** $H$ is too big at $j$.
- **Inductive step:**

---

*Figure 2.5* The Heapify-down process: Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
**Correctness of Heapify-down: Inductive Step**

- **H is too big at** \( j \) **if there is a value** \( \alpha \leq \text{key}(H[j]) \) **such that decreasing** \( \text{key}(H[j]) \) **to** \( \alpha \) **makes** \( H \) **a heap.**
- **Inductive hypothesis (two parts):**
  - If \( H \) **too big at** \( 2j \), then **Heapify-down** \((H, 2j)\) **creates a heap.**
  - If \( H \) **is too big at** \( 2j + 1 \), then **Heapify-down** \((H, 2j + 1)\) **creates a heap.**
- **Start of inductive step:** \( H \) **is too big at** \( j \).
- **Inductive step:** After the swap statement in **Heapify-down** \((H, j)\), (a) \( H \) **is a heap,** (b) \( H \) **is too big at** \( 2j \), **or (c) \( H \) **is too big at** \( 2j + 1 \).
Correctness of Heapify-down: Inductive Step

- **H is too big at** $j$ **if there is a value** $\alpha \leq \text{key}(H[j])$ **such that decreasing** $\text{key}(H[j])$ **to** $\alpha$ **makes** $H$ **a heap.**
- **Inductive hypothesis (two parts):**
  - If $H$ too big at $2j$, then $\text{Heapify-down}(H,2j)$ creates a heap.
  - If $H$ is too big at $2j + 1$, then $\text{Heapify-down}(H,2j + 1)$ creates a heap.
- **Start of inductive step:** $H$ is too big at $j$.
- **Inductive step:** After the swap statement in $\text{Heapify-down}(H,j)$, (a) $H$ is a heap, (b) $H$ is too big at $2j$, or (c) $H$ is too big at $2j + 1$. Proof on board.
Running time of Heapify-down

Heapify-down(H,i):
  Let $n = \text{length}(H)$
  If $2i > n$ then
    Terminate with $H$ unchanged
  Else if $2i < n$ then
    Let left = $2i$, and right = $2i + 1$
    Let $j$ be the index that minimizes key[H[left]] and key[H[right]]
  Else if $2i = n$ then
    Let $j = 2i$
  Endif
  If key[H[j]] < key[H[i]] then
    swap the array entries $H[i]$ and $H[j]$
    Heapify-down(H,j)
  Endif

Recurrence for running time of Heapify-down($H, i$)
Running time of Heapify-down

Heapify-down(H,i):
    Let \( n = \text{length}(H) \)
    If \( 2i > n \) then
        Terminate with \( H \) unchanged
    Else if \( 2i < n \) then
        Let left = \( 2i \), and right = \( 2i + 1 \)
        Let \( j \) be the index that minimizes \( \text{key}[H[\text{left}]] \) and \( \text{key}[H[\text{right}]] \)
    Else if \( 2i = n \) then
        Let \( j = 2i \)
    Endif
    If \( \text{key}[H[j]] < \text{key}[H[i]] \) then
        swap the array entries \( H[i] \) and \( H[j] \)
        Heapify-down(H,j)
    Endif

\[ T(i) = \begin{cases} 
\max \left( T(2i), T(2i+1) \right) + 1 & \text{if } i > 1 \\
O(1) & \text{if } 2i > n 
\end{cases} \]
Running time of Heapify-down

Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
    swap the array entries H[i] and H[j]
    Heapify-down(H,j)
  Endif

- Recurrence for running time of Heapify-down(H, i)
  \[ T(i) = \begin{cases} 
  \max(T(2i), T(2i + 1)) + 1 & \text{if } i > 1 \\
  O(1) & \text{if } 2i > n 
  \end{cases} \]

- Alternative proof since the recurrence is ugly.
Running time of Heapify-down

Heapify-down(H, i):
   Let n = length(H)
   If 2i > n then
      Terminate with H unchanged
   Else if 2i < n then
      Let left = 2i, and right = 2i + 1
      Let j be the index that minimizes key[H[left]] and key[H[right]]
   Else if 2i = n then
      Let j = 2i
   Endif
   If key[H[j]] < key[H[i]] then
      swap the array entries H[i] and H[j]
      Heapify-down(H, j)
   Endif

Recurrence for running time of Heapify-down(H, i)

\[ T(i) = \begin{cases} 
   \max(T(2i), T(2i + 1)) + 1 & \text{if } i > 1 \\
   O(1) & \text{if } 2i > n 
\end{cases} \]

Alternative proof since the recurrence is ugly.
Every invocation of Heapify-down increases its second argument by a factor of at least two.
Running time of Heapify-down

Heapify-down(H,i):
Let n = length(H)
If 2i > n then
    Terminate with H unchanged
Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
Else if 2i = n then
    Let j = 2i
Endif
If key[H[j]] < key[H[i]] then
    swap the array entries H[i] and H[j]
    Heapify-down(H,j)
Endif

Recurrence for running time of Heapify-down(H, i)

\[
T(i) = \begin{cases} 
\max(T(2i), T(2i + 1)) + 1 & \text{if } i > 1 \\
O(1) & \text{if } 2i > n
\end{cases}
\]

Alternative proof since the recurrence is ugly.

Every invocation of Heapify-down increases its second argument by a factor of at least two.

After k invocations argument must be at least
Running time of Heapify-down

Heapify-down(H, i):
  Let $n = \text{length}(H)$
  If $2i > n$ then
    Terminate with $H$ unchanged
  Else if $2i < n$ then
    Let $\text{left} = 2i$, and $\text{right} = 2i + 1$
    Let $j$ be the index that minimizes $\text{key}[H[\text{left}]]$ and $\text{key}[H[\text{right}]]$
  Else if $2i = n$ then
    Let $j = 2i$
  Endif
  If $\text{key}[H[j]] < \text{key}[H[i]]$ then
    swap the array entries $H[i]$ and $H[j]$
    Heapify-down(H, j)
  Endif

Recurrence for running time of Heapify-down($H, i$)

$$T(i) = \begin{cases} 
\max(T(2i), T(2i + 1)) + 1 & \text{if } i > 1 \\
O(1) & \text{if } 2i > n
\end{cases}$$

Alternative proof since the recurrence is ugly.
Every invocation of Heapify-down increases its second argument by a factor of at least two.
After $k$ invocations argument must be at least $i2^k \leq n$, which implies that $k \leq \log_2 n/i$. Therefore running time is $O(\log_2 n/i)$. 

Priority Queues and Heaps

September 9, 2014
Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$. 
Sorting Numbers with the Priority Queue

Sort

**INSTANCE:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

Final algorithm:
- Insert each number in a priority queue $H$.
- Repeatedly find the smallest number in $H$, output it, and delete it from $H$. 

September 9, 2014 Priority Queues and Heaps
Sorting Numbers with the Priority Queue

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

▶ Final algorithm:
  ▶ Insert each number in a priority queue $H$.
  ▶ Repeatedly find the smallest number in $H$, output it, and delete it from $H$.

▶ Each insertion and deletion takes $O(\log n)$ time for a total running time of $O(n \log n)$.