Priority Queues

T. M. Murali

August 31, 2009
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 $1 \leq i < n$. 
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.
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 must take \( O(\log n) \) time.
Candidate Data Structures for Sorting

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

Data structure must support insertion, finding minimum, and deleting minimum.

List
Candidate Data Structures for Sorting

Data structure must support insertion, 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

Data structure must support insertion, 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**
Candidate Data Structures for Sorting

Data structure must support insertion, 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, update the key of an element, insert an element, delete an element.
- Key update and element deletion 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 \( \text{key}(w) \leq \text{key}(v) \).
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 \( 2i > n \), the 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 \( \text{Heapify-up}(H, n + 1) \).

---

\( \text{Heapify-up}(H, i) : \)

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
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

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

H is *almost a heap with key of H*[i] *too small* if there is a value \( \alpha \geq \text{key}(H[i]) \) such that increasing \( \text{key}(H[i]) \) to \( \alpha \) makes \( H \) a heap.

Prove by induction on \( i \).
Correctness of Heapify-up

- **$H$ is almost a heap with key of $H[i]$ too small** if there is a value $\alpha \geq \text{key}(H[i])$ such that increasing $\text{key}(H[i])$ to $\alpha$ makes $H$ a heap.
- Prove by induction on $i$.
- Proof base case: $i = 1$.
- Proof inductive step: If $H$ is almost a heap with key of $H[i]$ too small, after Heapify-up($H$, $i$), $H$ is a heap or almost a heap with the key of $H[j]$ too small.

---

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

- **$H$** is *almost a heap with key of $H[i]$ too small* if there is a value $\alpha \geq \text{key}(H[i])$ such that increasing $\text{key}(H[i])$ to $\alpha$ makes $H$ a heap.
- Prove by induction on $i$.
- Proof base case: $i = 1$.
- Proof inductive step: If $H$ is almost a heap with key of $H[i]$ too small, after Heapify-up($H, i$), $H$ is a heap or almost a heap with the key of $H[j]$ too small.
- Running time is $O(\log i)$.
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 $n =$ 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
Example of **Heapify-down**

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

- **$H$ is almost a heap with key of $H[i]$ too big** if there is a value $\alpha \leq \text{key}(H[i])$ such that decreasing $\text{key}(H[i])$ to $\alpha$ makes $H$ a heap.
- Proof by reverse induction on $i$.
- Proof base case:

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 *almost a heap with key of** \( H[i] \) *too big* if there is a value \( \alpha \leq \text{key}(H[i]) \) such that decreasing \( \text{key}(H[i]) \) to \( \alpha \) makes \( H \) a heap.
- **Proof by reverse induction** on \( i \).
- **Proof base case:** \( 2i > n \).
- **Proof 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

- \( H \) is *almost a heap with key of \( H[i] \) too big* if there is a value \( \alpha \leq \text{key}(H[i]) \) such that decreasing \( \text{key}(H[i]) \) to \( \alpha \) makes \( H \) a heap.

- Proof by reverse induction on \( i \).

- Proof base case: \( 2i > n \).

- Proof inductive step: after Heapify-down(\( H, i \)), \( H \) is a heap or almost a heap with the key of \( H[j] \) too big.

**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 *almost a heap with key of H[i] too big* if there is a value \( \alpha \leq \text{key}(H[i]) \) such that decreasing \( \text{key}(H[i]) \) to \( \alpha \) makes \( H \) a heap.
- Proof by reverse induction on \( i \).
- Proof base case: \( 2i > n \).
- Proof inductive step: after Heapify-down(\( H, i \)), \( H \) is a heap or almost a heap with the key of \( H[j] \) too big.
- Running time of Heapify-down(\( H, i \)) is \( O(\log n) \).

---

**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).
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 \( 1 \leq i < n \).
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 $H$. 

T. M. Murali August 31, 2009 CS 4104: Priority Queues
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 $O(n \log n)$. 