Lecture 8
Heapsort

View in slide-show mode
Heapsort

- Worst-case runtime: $O(n \log n)$

- Sorts in-place

- Uses a special data structure (heap) to manage information during execution of the algorithm

  ➔ Another design paradigm
Heap Data Structure

Nearly complete binary tree

⇒ Completely filled on all levels except possibly the lowest level
Heap Data Structures

Height of node $i$: Length of the longest simple downward path from $i$ to a leaf

Height of the tree: height of the root
Heap Data Structures

Depth of node $i$: Length of the simple downward path from the root to node $i$
Heap Property: Min-Heap

Min heap: For every node $i$ other than root, $A[parent(i)] \leq A[i]$

⇒ Parent node is always smaller than the child nodes
Heap Property: Max-Heap

Max heap: For every node $i$ other than root, $A[parent(i)] \geq A[i]$

$\Rightarrow$ Parent node is always larger than the child nodes
Heap Property: Max-Heap

**Max heap**: For every node \( i \) other than root, \( A[\text{parent}(i)] \geq A[i] \)

\( \Rightarrow \) Parent node is always larger than the child nodes

The largest element in any subtree is the root element in a max-heap
Heap Data Structure

Heap can be stored in a linear array

Storage

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>14</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Heap Data Structure

The links in the heap are implicit:

- $left(i) = 2i$
- $right(i) = 2i + 1$
- $parent(i) = \lfloor i / 2 \rfloor$
Heap Data Structure

- **left(i)** = 2i
  - e.g. Left child of node 4 has index 8

- **right(i)** = 2i + 1
  - e.g. Right child of node 2 has index 5

- **parent(i)** = $\left\lfloor \frac{i}{2} \right\rfloor$
  - e.g. Parent of node 7 has index 3
Heap Data Structures

- Computing left child, right child, and parent indices very fast
  - left(i) = 2i \(\Rightarrow\) binary left shift
  - right(i) = 2i+1 \(\Rightarrow\) binary left shift, then set the lowest bit to 1
  - parent(i) = floor(i/2) \(\Rightarrow\) right shift in binary

- A[1] is always the root element

- Array A has two attributes:
  - length(A): The number of elements in A
  - n = heap-size(A): The number of elements in heap

\[
n \leq \text{length}(A)
\]
Heap Operations: Extract-Max

**EXTRACT-MAX(A, n)**

max ← A[1]
n ← n − 1
**HEAPIFY(A, 1, n)**
return max

Return the max element, and reorganize the heap to maintain heap property
Heap Operations: HEAPIFY

Maintaining heap property:

Subtrees rooted at left[i] and right[i] are already heaps.

But, A[i] may violate the heap property (i.e., may be smaller than its children)

Idea: Float down the value at A[i] in the heap so that subtree rooted at i becomes a heap.
Heap Operations: HEAPIFY

**HEAPIFY**(A, i, n)

1. largest ← i
2. if \( 2i \leq n \) and \( A[2i] > A[i] \) then largest ← 2i
3. if \( 2i + 1 \leq n \) and \( A[2i+1] > A[\text{largest}] \) then largest ← 2i + 1
4. if largest ≠ i then exchange \( A[i] \leftarrow A[\text{largest}] \)
5. \( \text{HEAPIFY}(A, \text{largest}, n) \)

- Initialize largest to be the node \( i \)
- Check the left child of node \( i \)
- Check the right child of node \( i \)
- Exchange the largest of the 3 with node \( i \)
- Recursive call on the subtree
Heap Operations: HEAPIFY

**HEAPIFY(A, i, n)**

largest ← i

if $2i \leq n$ and $A[2i] > A[i]$ then largest ← 2i

if $2i + 1 \leq n$ and $A[2i+1] > A[\text{largest}]$ then largest ← 2i + 1

if largest ≠ i then
  exchange $A[i] \leftrightarrow A[\text{largest}]

HEAPIFY(A, largest, n)

recursive call
HEAPIFY($A$, $i$, $n$)

largest $\leftarrow i$

if $2i \leq n$ and $A[2i] > A[i]$ then largest $\leftarrow 2i$

if $2i + 1 \leq n$ and $A[2i+1] > A[\text{largest}]$ then largest $\leftarrow 2i + 1$

if largest $\neq i$ then

exchange $A[i] \leftrightarrow A[\text{largest}]$

HEAPIFY($A$, largest, $n$)

recursive call:

HEAPIFY($A$, 2, 9)
Heap Operations: HEAPIFY

**HEAPIFY(A, i, n)**

largest $\leftarrow i$

if $2i \leq n$ and $A[2i] > A[i]$

then largest $\leftarrow 2i$

if $2i + 1 \leq n$ and $A[2i+1] > A[\text{largest}]$

then largest $\leftarrow 2i + 1$

if largest $\neq i$ then

exchange $A[i] \leftrightarrow A[\text{largest}]$

**HEAPIFY(A, largest, n)**

recursive call: **HEAPIFY(A, 4, 9)**

recursive call (base case)
HEAPIFY: Summary (Floating Down the Value)

**HEAPIFY**(A, i, n)

largest ← i

if 2i ≤ n and A[2i] > A[i] then largest ← 2i

if 2i +1 ≤ n and A[2i+1] > A[largest] then largest ← 2i +1

if largest ≠ i then
   HEAPIFY(A, largest, n)
Heap Operations: HEAPIFY

**HEAPIFY**(A, i, n)

largest ← i

if \(2i \leq n\) and \(A[2i] > A[i]\) then largest ← 2i

if \(2i + 1 \leq n\) and \(A[2i+1] > A[\text{largest}]\) then largest ← 2i +1

if largest ≠ i then exchange \(A[i] \leftrightarrow A[\text{largest}]\)

**HEAPIFY**(A, largest, n)

**after HEAPIFY:**

```
14  
10  
9   
8   
7   
6   
5   
4   
3   
2   
1   
```
Intuitive Analysis of HEAPIFY

• Consider $\text{HEAPIFY}(A, i, n)$
  – let $h(i)$ be the height of node $i$
  – at most $h(i)$ recursion levels
    • Constant work at each level: $\Theta(1)$
  – Therefore $T(i) = O(h(i))$

• Heap is almost-complete binary tree
  $\triangleright h(n) = O(\lg n)$

• Thus $T(n) = O(\lg n)$
What is the recurrence?

- Depends on the size of the subtree on which recursive call is made
- In the next couple of slides, we try to compute an upper bound for this subtree.
Reminder: Binary trees

For a complete binary tree:

- # of nodes at depth $d$: $2^d$
- # of nodes with depths less than $d$: $2^d - 1$

Example:

$\begin{align*}
d &= 2 \\
\text{# of nodes at depth } d=2: &\ 4 \\
\text{# of nodes with depths } d<2: &\ 3
\end{align*}$
Formal Analysis of HEAPIFY

- Worst case occurs when last row of the subtree $S_i$ rooted at node $i$ is half full

- $T(n) \leq T(|S_{L(i)}|) + \Theta(1)$
- $S_{L(i)}$ and $S_{R(i)}$ are complete binary trees of heights $h(i) - 1$ and $h(i) - 2$, respectively
Formal Analysis of HEAPIFY

- Let \( m \) be the number of leaf nodes in \( S_{L(i)} \)
- \(| S_{L(i)} | = m + (m - 1) = 2m - 1 \); outer + inner leaf nodes
- \(| S_{R(i)} | = m/2 + (m/2 - 1) = m - 1 \)
- \(| S_{L(i)} | + | S_{R(i)} | + 1 = n \)

\[(2m - 1) + (m - 1) + 1 = n \Rightarrow m = (n+1)/3\]

\(| S_{L(i)} | = 2m - 1 = 2(n+1)/3 - 1 = (2n/3 + 2/3) - 1 = 2n/3 - 1/3 \leq 2n/3\]

- \( T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n) \)

By case 2 of Master Thm
HEAPIFY: Efficiency Issues

- Recursion vs iteration:
  - In the absence of tail recursion, iterative version is in general more efficient
    - because of the pop/push operations to/from stack at each level of recursion.
Heap Operations: HEAPIFY

**Recursive:**

\[
\text{HEAPIFY}(A, i, n)
\]

1. \( \text{largest} \leftarrow i \)
2. \( \text{if } 2i \leq n \text{ and } A[2i] > A[i] \) \( \text{then } \text{largest} \leftarrow 2i \)

3. \( \text{if } 2i +1 \leq n \text{ and } A[2i+1] > A[\text{largest}] \) \( \text{then } \text{largest} \leftarrow 2i +1 \)

4. \( \text{if } \text{largest} \neq i \text{ then} \)
   - exchange \( A[i] \leftrightarrow A[\text{largest}] \)
   - \( \text{HEAPIFY}(A, \text{largest}, n) \)

**Iterative:**

\[
\text{HEAPIFY}(A, i, n)
\]

1. \( j \leftarrow i \)
2. \( \text{while (true) do} \)
   - \( \text{largest} \leftarrow j \)
   - \( \text{if } 2j \leq n \text{ and } A[2j] > A[j] \) \( \text{then } \text{largest} \leftarrow 2j \)
   - \( \text{if } 2j +1 \leq n \text{ and } A[2j+1] > A[\text{largest}] \) \( \text{then } \text{largest} \leftarrow 2j +1 \)
   - \( \text{if } \text{largest} \neq j \text{ then} \)
     - exchange \( A[j] \leftrightarrow A[\text{largest}] \)
     - \( j \leftarrow \text{largest} \)
3. \( \text{else return} \)
Given an arbitrary array, how to build a heap from scratch?

Basic idea: Call HEAPIFY on each node bottom up

- Start from the leaves (which trivially satisfy the heap property)
- Process nodes in bottom up order.
- When HEAPIFY is called on node i, the subtrees connected to the left and right subtrees already satisfy the heap property.
Where are the leaves stored?

Lemma: The last $\lceil n/2 \rceil$ nodes of a heap are all leaves
Proof of Lemma

**Lemma**: last $\lceil n/2 \rceil$ nodes of a heap are all leaves

**Proof**: 

- $m = 2^{d-1}$: # nodes at level $d-1$
- $f$: # nodes at level $d$ (last level)

# of nodes with depth $d-1$: $m$
# of nodes with depth $< d-1$: $m-1$
# of nodes with depth $d$: $f$
Total # of nodes: $n = f + 2m - 1$
Proof of Lemma (cont’d)

\[ f = n - 2m + 1 \]

\# of leaves: \[ f + m - \left\lfloor \frac{f}{2} \right\rfloor \]

\[ = m + \left\lfloor \frac{f}{2} \right\rfloor \]

\[ = m + \left\lfloor \frac{n-2m+1}{2} \right\rfloor \]

\[ = \left\lfloor \frac{(n+1)}{2} \right\rfloor \]

\[ = \left\lfloor \frac{n}{2} \right\rfloor \]

**Proof complete**
Heap Operations: Building Heap

\[
\text{BUILD-HEAP} (A, n) \\
\text{for } i = \lfloor n/2 \rfloor \text{ downto } 1 \text{ do} \\
\text{HEAPIFY} (A, i, n)
\]

**Reminder**: The last \( \lfloor n/2 \rfloor \) nodes of a heap are *all leaves*, which trivially satisfy the heap property.
Build-Heap: Example

HEAPIFY(A, 5, 10)

A

| 4 | 1 | 3 | 2 | 7 | 9 | 10 | 14 | 8 | 16 |

i=5
Build-Heap: Example

HEAPIFY(A, 4, 10)

A

1 2 3 4 5 6 7 8 9 10
Build-Heap: Example

HEAPIFY(A, 3, 10)
Build-Heap: Example

HEAPIFY(A, 2, 10)

i=2
Build-Heap: Example

\[
\text{HEAPIFY}(A, 2, 10) \quad i=2 \ (cont'd)
\]
Build-Heap: Example

HEAPIFY(A, 1, 10)

A  

\[
\begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 \\
4 & \quad 16 & \quad 10 & \quad 14 & \quad 7 & \quad 9 & \quad 3 & \quad 2 & \quad 8 & \quad 1
\end{align*}
\]
Build-Heap: Example

HEAPIFY(A, 1, 10)

i=1 (cont’d)
Build-Heap: Example

$\text{HEAPIFY}(A, 1, 10)$

$i = 1$ (cont’d)
Build-Heap: Example

After Build-Heap

A

1  2  3  4  5  6  7  8  9  10

16  14  10  8  7  9  3  2  4  1
Build-Heap: Runtime Analysis

- Simple analysis:
  - $O(n)$ calls to `HEAPIFY`, each of which takes $O(lgn)$ time
  - $O(nlgn)$ loose bound

- In general, a good approach:
  - Start by proving an easy bound
  - Then, try to tighten it

- Is there a tighter bound?
Build-Heap: tighter running time analysis

If the heap is complete binary tree then $h_\ell = d - \ell$

Otherwise, nodes at a given level do not all have the same height

But we have $d - \ell - 1 \leq h_\ell \leq d - \ell$
Build-Heap: **tighter running time analysis**

Assume that all nodes at level $\ell = d - 1$ are processed

$$\sum_{\ell=0}^{d-1} n_{\ell} O(h_{\ell}) = O\left(\sum_{\ell=0}^{d-1} n_{\ell} h_{\ell}\right)$$

Let $h = d - \ell \Rightarrow \ell = d - h$ (change of variables)

$$\therefore T(n) = O\left(\sum_{\ell=0}^{d-1} 2^\ell (d - \ell)\right)$$

$$T(n) = O\left(\sum_{h=1}^{d} h 2^{d-h}\right) = O\left(\sum_{h=1}^{d} h 2^{d/2^h}\right) = O\left(2^d \sum_{h=1}^{d} h (1/2)^h\right)$$

but $2^d = \Theta(n) \Rightarrow T(n) = O\left(n \sum_{h=1}^{d} h (1/2)^h\right)$
Build-Heap: tighter running time analysis

\[
\sum_{h=1}^{d} h(1/2)^h \leq \sum_{h=0}^{d} h(1/2)^h \leq \sum_{h=0}^{\infty} h(1/2)^h
\]

recall infinite decreasing geometric series

\[
\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \text{ where } |x| < 1
\]

differentiate both sides

\[
\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}
\]
Build-Heap: tighter running time analysis

\[ \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2} \]

then, multiply both sides by \( x \)

\[ \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \]

in our case: \( x = 1/2 \) and \( k = h \)

\[ \therefore \sum_{h=0}^{\infty} h(1/2)^h = \frac{1/2}{(1-1/2)^2} = 2 = O(1) \]

\[ \therefore T(n) = O(n \sum_{h=1}^{d} h(1/2)^h) = O(n) \]
Heapsort Algorithm

The HEAPSORT algorithm

1. Build a heap on array \( A[1…n] \) by calling BUILD-HEAP\((A, n)\)
2. The largest element is stored at the root \( A[1] \)
3. Discard node \( n \) from the heap
4. Subtrees (\( S_2 \) & \( S_3 \)) rooted at children of root remain as heaps but the new root element may violate the heap property
   Make \( A[1…n – 1] \) a heap by calling HEAPIFY\((A, 1, n – 1)\)
5. \( n \leftarrow n – 1 \)
6. Repeat steps 2–4 until \( n = 2 \)
**Heapsort Algorithm**

**HEAPSORT**(\(A, n\))

**BUILD-HEAP**\((A, n)\)

for \(i \leftarrow n\) downto 2 do

exchange \(A[1] \leftrightarrow A[i]\)

**HEAPIFY**\((A, 1, i - 1)\)

---

**A**

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
16 & 14 & 10 & 8 & 7 & 9 & 3 & 2 & 4 & 1
\end{array}
\]
Heapsort Algorithm

HEAPSORT(A, n)

BUILD-HEAP(A, n)

for i ← n downto 2 do


HEAPIFY(A, 1, i -1)
HEAPSORT($A, n$)

BUILD-HEAP($A, n$)

for $i \leftarrow n$ downto 2 do
    exchange $A[1] \leftrightarrow A[i]$
    HEAPIFY($A, 1, i-1$)
Heapsort Algorithm

**HEAPSORT**(*A, n*)

**BUILD-HEAP**(*A, n*)

for *i* ← *n* downto 2 do

exchange *A*[1] ↔ *A*[i]

**HEAPIFY**(*A, 1, i − 1*)

```
A
1 2 3 4 5 6 7 8 9 10
1 8 10 4 7 9 3 2 14 16
```
Heapsort Algorithm

HEAPSORT(A, n)

BUILD-HEAP(A, n)

for i ← n downto 2 do
  HEAPIFY(A, 1, i − 1)
Heapsort Algorithm

**HEAPSORT**(A, n)

BUILD-HEAP(A, n)

for \( i \leftarrow n \) downto 2 do

exchange A[1] \( \leftrightarrow \) A[i]

HEAPIFY(A, 1, \( i - 1 \))
Heapsort Algorithm

**HEAPSORT**(A, n)

BUILD-HEAP(A, n)

for i ← n downto 2 do


HEAPIFY(A, 1, i − 1)
Heapsort Algorithm

**HEAPSORT(A, n)**

BUILD-HEAP(A, n)

for $i \leftarrow n \text{ downto } 2$ do

exchange $A[1] \leftrightarrow A[i]$

HEAPIFY(A, 1, $i - 1$)
Heapsort Algorithm

\[ \text{HEAPSORT}(A, n) \]

1. BUILD-HEAP(A, n)
2. for \( i \leftarrow n \) downto 2 do
   - exchange \( A[1] \leftrightarrow A[i] \)
   - HEAPIFY(A, 1, i - 1)
HEAPSORT($A, n$)

BUILD-HEAP($A, n$)

for $i \leftarrow n$ downto $2$ do

exchange $A[1] \leftrightarrow A[i]$

HEAPIFY($A, 1, i - 1$)
Heapsort Algorithm

**HEAPSORT**(*A, n*)

BUILD-HEAP(*A, n*)

for *i* ← *n* downto 2 do

exchange *A*[1] ↔ *A*[i]

HEAPIFY(*A, 1, i – 1*)

---

A

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
Heapsort Algorithm

HEAPSORT($A, n$)

BUILD-HEAP($A, n$)

for $i \leftarrow n$ downto 2 do

exchange $A[1] \leftrightarrow A[i]$

HEAPIFY($A, 1, i - 1$)
Heapsort Algorithm

HEAPSORT\((A, n)\)

BUILD-HEAP\((A, n)\)

for \(i \leftarrow n\) downto 2 do

exchange \(A[1] \leftrightarrow A[i]\)

HEAPIFY\((A, 1, i - 1)\)
Heapsort Algorithm

HEAPSORT(A, n)

BUILD-HEAP(A, n)

for i ← n downto 2 do

HEAPIFY(A, 1, i - 1)

A

1 2 3 4 5 6 7 8 9 10

1 2 3 4 7 8 9 10 14 16
HEAPSORT (A, n)

BUILD-HEAP(A, n)

for i ← n downto 2 do
HEAPIFY(A, 1, i - 1)
Heapsort Algorithm

\[ \text{HEAPSORT}(A, n) \]

\[ \text{BUILD-HEAP}(A, n) \]

\[ \text{for } i \leftarrow n \text{ downto } 2 \text{ do} \]

\[ \text{exchange } A[1] \leftrightarrow A[i] \]

\[ \text{HEAPIFY}(A, 1, i - 1) \]
Heapsort Algorithm

\[ \text{HEAPSORT}(A, n) \]
\[ \text{BUILD-HEAP}(A, n) \]
\[ \text{for } i \leftarrow n \text{ downto } 2 \text{ do} \]
\[ \text{exchange } A[1] \leftrightarrow A[i] \]
\[ \text{HEAPIFY}(A, 1, i-1) \]
Heapsort Algorithm

HEAPSORT\( (A, n) \)

BUILD-HEAP\( (A, n) \)

for \( i \leftarrow n \) downto 2 do

exchange \( A[1] \leftrightarrow A[i] \)

HEAPIFY\( (A, 1, i-1) \)
Heapsort Algorithm

\[ \text{HEAPSORT}(A, n) \]

\[ \text{BUILD-HEAP}(A, n) \]

\text{for } i \leftarrow n \text{ downto } 2 \text{ do}

\text{exchange } A[1] \leftrightarrow A[i]

\text{HEAPIFY}(A, 1, i - 1)
Heapsort Algorithm: Runtime Analysis

\[
\text{HEAPSORT}(A, n) \\
\text{BUILD-HEAP}(A, n) \quad \Theta(n) \\
\text{for } i \leftarrow n \text{ downto } 2 \text{ do} \\
\quad \text{exchange } A[1] \leftrightarrow A[i] \quad \Theta(1) \\
\quad \text{HEAPIFY}(A, 1, i - 1) \quad O(\lg(i-1))
\]

\[
T(n) = \Theta(n) + \sum_{i=2}^{n} O(\lg i) = \Theta(n) + O\left(\sum_{i=2}^{n} O(\lg n)\right) = O(n \lg n)
\]
Heapsort - Notes

- **Heapsort** is a very good algorithm but, a good implementation of **quicksort** always **beats** heapsort in practice.

- However, **heap data structure** has many popular applications, and it can be efficiently used for implementing **priority queues**.
Data structures for Dynamic Sets

• Consider sets of records having *key* and *satellite* data
Operations on Dynamic Sets

• Queries: Simply return info; Modifying operations: Change the set

  – INSERT($S$, $x$): (Modifying) $S ← S \cup \{x\}$
  – DELETE($S$, $x$): (Modifying) $S ← S - \{x\}$
  – $\text{MAX}(S) / \text{MIN}(S)$: (Query) return $x \in S$ with the largest/smallest key
  – $\text{EXTRACT-MAX}(S) / \text{EXTRACT-MIN}(S)$: (Modifying) return and delete $x \in S$ with the largest/smallest key
  – SEARCH($S$, $k$): (Query) return $x \in S$ with $\text{key}[x] = k$
  – $\text{SUCCESSOR}(S, x) / \text{PREDECESSOR}(S, x)$: (Query) return $y \in S$ which is the next larger/smaller element after $x$

• Different data structures support/optimize different operations
Priority Queues ($PQ$)

- **Supports**
  - INSERT
  - MAX / MIN
  - EXTRACT-MAX / EXTRACT-MIN

- **One application**: Schedule jobs on a shared resource
  - PQ keeps track of jobs and their relative priorities
  - When a job is finished or interrupted, highest priority job is selected from those pending using EXTRACT-MAX
  - A new job can be added at any time using INSERT
Priority Queues

- **Another application**: Event-driven simulation
  - Events to be simulated are the items in the PQ
  - Each event is associated with a time of occurrence which serves as a *key*
  - Simulation of an event can cause other events to be simulated in the future
  - Use **EXTRACT-MIN** at each step to choose the next event to simulate
  - As new events are produced insert them into the PQ using **INSERT**
Implementation of Priority Queue

- **Sorted linked list**: Simplest implementation
  - **INSERT**
    - \( O(n) \) time
    - Scan the list to find place and splice in the new item
  - **EXTRACT-MAX**
    - \( O(1) \) time
    - Take the first element

▷ Fast extraction but **slow** insertion.
Implementation of Priority Queue

- **Unsorted linked list**: Simplest implementation
  - **INSERT**
    - $O(1)$ time
    - Put the new item at front
  - **EXTRACT-MAX**
    - $O(n)$ time
    - Scan the whole list

> Fast insertion but slow extraction

Sorted linked list is better on the average
- **Sorted list**: on the average, scans $n/2$ elem. per insertion
- **Unsorted list**: always scans $n$ elem. at each extraction
Heap Implementation of PQ

- **INSERT** and **EXTRACT-MAX** are both \( O(\lg n) \)
  - good compromise between fast insertion but slow extraction and vice versa
- **EXTRACT-MAX**: already discussed **HEAP-EXTRACT-MAX**

**INSERT**: Insertion is like that of Insertion-Sort.

Traverses \( O(\lg n) \) nodes, as **HEAPIFY** does but makes fewer comparisons and assignments

- **HEAPIFY**: compares parent with both children
- **HEAP-INSERT**: with only one

**HEAP-INSERT**(A, \( key \), \( n \))

\[
\begin{align*}
n & \leftarrow n + 1 \\
i & \leftarrow n \\
\textbf{while } i > 1 \textbf{ and } A[\lfloor i/2 \rfloor] < key \textbf{ do} \\
A[i] & \leftarrow A[\lfloor i/2 \rfloor] \\
i & \leftarrow \lfloor i/2 \rfloor \\
A[i] & \leftarrow key
\end{align*}
\]
Example: \textbf{HEAP-INSERT}(A, 15)

\begin{align*}
\text{HEAP-INSERT}(A, \text{key}, n) & \\
n & \leftarrow n + 1 \\
i & \leftarrow n \\
\text{while } i > 1 \text{ and } A[\lfloor i/2 \rfloor] < \text{key} \text{ do} & \\
A[i] & \leftarrow A[\lfloor i/2 \rfloor] \\
i & \leftarrow \lfloor i/2 \rfloor \\
A[i] & \leftarrow \text{key}
\end{align*}

\begin{itemize}
\item[A\[4\]]{8} \\
\item[A\[2\]]{2} \\
\text{key = 15}
\end{itemize}
Example: **HEAP-INSERT(A, 15)**

**HEAP-INSERT(A, key, n)**

\[ n \leftarrow n + 1 \]

\[ i \leftarrow n \]

**while** \( i > 1 \) **and** \( A\lfloor i/2 \rfloor < key \) **do**

\[ A[i] \leftarrow A\lfloor i/2 \rfloor \]

\[ i \leftarrow \lfloor i/2 \rfloor \]

\[ A[i] \leftarrow key \]

**Diagram:**

- **16**
- **14**
- **10**
- **7**
- **9**
- **3**
- **2**
- **4**
- **1**
- **8**
- **10**
- **11**

**Key:** 15
Example: **HEAP-INSERT**(*A, 15*)

**HEAP-INSERT**(*A, key, n*)

\[
\begin{align*}
n & \leftarrow n + 1 \\
i & \leftarrow n \\
\text{while } i > 1 \text{ and } A[\lfloor i/2 \rfloor] < key & \text{ do} \\
A[i] & \leftarrow A[\lfloor i/2 \rfloor] \\
i & \leftarrow \lfloor i/2 \rfloor \\
A[i] & \leftarrow key
\end{align*}
\]
Example: **HEAP-INSERT**(A, 15)

**HEAP-INSERT**(A, key, n)

\[ n \leftarrow n + 1 \]
\[ i \leftarrow n \]
\[ \text{while } i > 1 \text{ and } A[\lfloor i/2 \rfloor] < key \text{ do} \]
\[ A[i] \leftarrow A[\lfloor i/2 \rfloor] \]
\[ i \leftarrow \lfloor i/2 \rfloor \]
\[ A[i] \leftarrow key \]

key = 15
Example: **HEAP-INSERT**(A, 15)

**HEAP-INSERT**(A, key, n)

\[ n \leftarrow n + 1 \]

\[ i \leftarrow n \]

while \( i > 1 \) and A\[\lfloor i/2 \rfloor\] < key do

\[ A[i] \leftarrow A\lfloor i/2 \rfloor \]

\[ i \leftarrow \lfloor i/2 \rfloor \]

\[ A[i] \leftarrow \text{key} \]
Heap Increase Key

- Key value of \( i \)-th element of heap is increased from \( A[i] \) to \( key \)

\[
\text{HEAP-INCREASE-KEY}(A, i, key)
\]

\[
\begin{align*}
\text{if} \quad &key < A[i] \text{ then} \\
\quad &\text{return error} \\
\text{while} \quad &i > 1 \text{ and } A[\lfloor i/2 \rfloor] < key \text{ do} \\
\quad &A[i] \leftarrow A[\lfloor i/2 \rfloor] \\
\quad &i \leftarrow \lfloor i/2 \rfloor \\
\quad &A[i] \leftarrow key
\end{align*}
\]
Example: **HEAP-INCREASE-KEY**(A, 9, 15)

**HEAP-INCREASE-KEY**(A, i, key)

if key < A[i] then
  return error

while i > 1 and A[\lfloor i/2 \rfloor] < key do
  A[i] ← A[\lfloor i/2 \rfloor]
  i ← \lfloor i/2 \rfloor
A[i] ← key

key = 15
Example: **HEAP-INCREASE-KEY**(A, 9, 15)

**HEAP-INCREASE-KEY**(A, i, key)

if key < A[i] then
  return error
while i > 1 and A[⌊i/2⌋] < key do
  A[i] ← A[⌊i/2⌋]
  i ← ⌊i/2⌋
A[i] ← key

key = 15
Example: **HEAP-INCREASE-KEY**(A, 9, 15)

**HEAP-INCREASE-KEY**(A, i, key)

if key < A[i] then
  return error

while i > 1 and A[\lfloor i/2 \rfloor] < key do
  A[i] ← A[\lfloor i/2 \rfloor]
  i ← \lfloor i/2 \rfloor
A[i] ← key

key = 15
Example: **HEAP-INCREASE-KEY**(A, 9, 15)

**HEAP-INCREASE-KEY**(A, i, key)

if key < A[i] then
    return error

while i > 1 and A\[\left\lfloor i/2 \right\rfloor\] < key do
    A[i] ← A\[\left\lfloor i/2 \right\rfloor\]
    i ← \[i/2\]
A[i] ← key

Key = 15
Example: **HEAP-INCREASE-KEY(A, 9, 15)**

```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i] then
        return error
    while i > 1 and A[floor(i/2)] < key do
        A[i] ← A[floor(i/2)]
        i ← floor(i/2)
    A[i] ← key
```
Heap Implementation of PQ

### Storage in Application

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
<th>H-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>*</td>
<td>--</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>g</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>h</td>
<td>*</td>
<td>--</td>
</tr>
<tr>
<td>i</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>j</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

### Heap Storage

<table>
<thead>
<tr>
<th>handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>k</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

### Abstract Heap Representation

```
16 10 3
  14
  8
  2
  4
  7
  9
  3
```
Summary: Max Heap

**Heapify**(A, i)
- Works when both child subtrees of node i are heaps
- “*Floats down*” node i to satisfy the heap property
- Runtime: $O(\log n)$

**Max (A, n)**
- Returns the max element of the heap (no modification)
- Runtime: $O(1)$

**Extract-Max (A, n)**
- Returns and removes the max element of the heap
- Fills the gap in A[1] with A[n], then calls Heapify(A,1)
- Runtime: $O(\log n)$
Summary: Max Heap

**Build-Heap(A, n)**
Given an arbitrary array, builds a heap from scratch
Runtime: $O(n)$

**Min(A, n)**
How to return the min element in a max-heap?
Worst case runtime: $O(n)$
because ~half of the heap elements are leaf nodes
Instead, use a min-heap for efficient min operations

**Search(A, x)**
For an arbitrary x value, the worst-case runtime: $O(n)$
Use a sorted array instead for efficient search operations
Summary: Max Heap

**Increase-Key(A, i, x)**

Increase the key of node i (from A[i] to x)
“Float up” x until heap property is satisfied
Runtime: $O(\log n)$

**Decrease-Key(A, i, x)**

Decrease the key of node i (from A[i] to x)
Call Heapify(A, i)
Runtime: $O(\log n)$
Example Problem: Phone Operator

A phone operator answering $n$ phones

Each phone $i$ has $x_i$ people waiting in line for their calls to be answered.

Phone operator needs to answer the phone with the largest number of people waiting in line.

New calls come continuously, and some people hang up after waiting.
Solution

**Step 1**: Define the following array:

\[
\begin{array}{c|c}
\text{key} & \text{id} \\
1 & \\
& \\
& \\
n & \\
\end{array}
\]

- \text{A}[i]: \text{the } i^{\text{th}} \text{ element in heap}
- \text{A}[i].\text{id}: \text{the index of the corresponding phone}
- \text{A}[i].\text{key}: \# \text{ of people waiting in line for phone with index } \text{A}[i].\text{id}
Solution

Step 2: Build-Max-Heap \((A, n)\)

Execution:

When the operator wants to answer a phone:

\[ id = A[1].id \]

\[ \text{Decrease-Key}(A, 1, A[1].\text{key}-1) \]

answer phone with index id

When a new call comes in to phone \(i\):

\[ \text{Increase-Key}(A, i, A[i].\text{key}+1) \]

When a call drops from phone \(i\):

\[ \text{Decrease-Key}(A, i, A[i].\text{key}-1) \]