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, \( A[\text{parent}(i)] \leq A[i] \)

\( \Rightarrow \) Parent node is always smaller than the child nodes
Heap Property: Max-Heap

**Max heap**: For every node $i$ other than root, $A[\text{parent}(i)] \geq A[i]$ → Parent node is always larger than the child nodes
Heap Property: Max-Heap

The largest element in any subtree is the root element in a 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
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($i$) = $i / 2$
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
- $parent(i) = \lfloor i / 2 \rfloor$
  - 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 elements in heap

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

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

1. `max ← A[1]`
3. `n ← n - 1`
4. **HEAPIFY**(A, 1, n)
5. **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)

- 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 exchange A[i] ↔ A[largest]

**HEAPIFY**(A, 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

compute the largest of:

1) node i
2) left child of node i
3) right child of node i
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
Heap Operations: HEAPIFY

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

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

---

**Diagram:**

Recursive call:

**HEAPIFY**(A, 4, 9)

Recursive call:

**HEAPIFY**(A, 1, 4)

Recursive call (base case)
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 $A[i] \leftrightarrow A[\text{largest}]
HEAPIFY(A, \text{largest}, n)$
Heap Operations: HEAPIFY

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

![Diagram](image.png)

after HEAPIFY:

![Heap Diagram](image.png)
Intuitive Analysis of HEAPIFY

• Consider 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 \( T(n) = O(\lg n) \)
Formal Analysis of HEAPIFY

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:

$\begin{array}{c}
\text{d = 2} \\
\hline
\end{array}$

- # of nodes at depth $d=2$: 4
- # of nodes with depths $d<2$: 3
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 $S_{L(i)}$
- $|S_{L(i)}| = m + (m - 1) = 2m - 1$
- $|S_{R(i)}| = m/2 + (m/2 - 1) = m - 1$
- $|S_{L(i)}| + |S_{R(i)}| + 1 = n$
- $(2m - 1) + (m - 1) + 1 = n \Rightarrow m = (n+1)/3$
- $|S_{L(i)}| = 2m - 1 = 2(n+1)/3 - 1 = (2n/3 + 2/3) - 1 = 2n/3 - 1/3 \leq 2n/3$
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log n)$

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:**

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
    HEAPIFY(A, largest, n)

**Iterative:**

HEAPIFY(A, i, n)

j ← i

while (true) do

    largest ← j

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

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

    if largest ≠ j then
        j ← largest
    else return
Heap Operations: Building Heap

- 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 \( \left\lfloor \frac{n}{2} \right\rfloor \) nodes of a heap are all leaves.
Proof of Lemma

**Lemma**: last \( \left\lfloor n/2 \right\rfloor \) nodes of a heap are all leaves

**Proof**: 

\[
m = 2^{d-1}: \text{# nodes at level } d - 1
\]

\[
f: \text{# nodes at level } d \text{ (last level)}
\]

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

\[ = m + \left\lfloor \frac{f}{2} \right\rfloor \]
\[ = m + \left\lfloor \frac{(n-2m+1)}{2} \right\rfloor \]
\[ = \left\lfloor \frac{(n+1)}{2} \right\rfloor \]
\[ = \left\lceil \frac{n}{2} \right\rceil \]

Proof complete
Heap Operations: Building Heap

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

```latex
\text{for } i = \lceil n/2 \rceil \text{ downto } 1 \text{ do }
\text{HEAPIFY}(A, i, n)
```

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

HEAPIFY(A, 5, 10)

i=5

A

1 2 3 4 5 6 7 8 9 10

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

A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

i = 4

HEAPIFY(A, 4, 10)
Build-Heap: Example

\[ \text{HEAPIFY}(A, 3, 10) \]
Build-Heap: Example

$\text{HEAPIFY}(A, 2, 10)$
Build-Heap: Example

HEAPIFY(A, 2, 10)

i=2 (cont’d)
Build-Heap: Example

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

16
  /   \
 14   10
  /    /   \ 
 8  7  9   3
  /  /  \
 2 4 1

16 [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 $d - \ell - 1 \leq h_\ell \leq d - \ell$
Assume that all nodes at level $\ell = d - 1$ are processed

$$T(n) = \sum_{\ell=0}^{d-1} n_\ell \Theta(h_\ell) = \Theta(\sum_{\ell=0}^{d-1} n_\ell h_\ell) \quad \begin{cases} n_\ell = 2^\ell = \# \text{ of nodes at level } \ell \\ h_\ell = \text{height of nodes at level } \ell \end{cases}$$

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

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

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

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

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

recall infinite decreasing geometric series

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

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 \(A[1]\)

   Put it into its correct final position \(A[n]\) by \(A[1] \leftrightarrow A[n]\)

(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(\textbf{A}, n)

BUILD-HEAP(A, n)

\textbf{for} i \leftarrow n \textbf{ downto } 2 \textbf{ 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 Algorithm:

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

**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

**HEAPSORT**(A, n)

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

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


**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 ← n downto 2 do


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

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

\begin{align*}
\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)
\end{align*}
Heapsort Algorithm

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 ← n downto 2 do


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

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

1. BUILD-HEAP(*A, n*)
2. for *i* ← *n* downto 2 do
   - exchange *A*[1] ↔ *A*[i]
   - **HEAPIFY**(*A, 1, i − 1*)

**Algorithm Code**

```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        HEAPIFY(A, 1, i − 1)
```

**Example**

- **A** = [4, 2, 3, 1, 7, 8, 9, 10, 14, 16]

**Diagram**

[Diagram showing the heap structure and the process of heap sort]

- **BUILD-HEAP** is used to create a max heap.
- **HEAPIFY** is used to maintain the heap property.
- The algorithm iterates over the array from the second last element to the first, exchanging the root (maximum) with the last element and then calling **HEAPIFY** on the reduced heap size.
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 ← n downto 2 do


HEAPIFY(A, 1, i – 1)

---

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

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

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: Runtime Analysis

**HEAPSORT(A, n)**

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

\[
T(n) = \sum_{i=2}^{n} O(\log i) = \sum_{i=2}^{n} O(1) = O(n \log n)
\]
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

```
x → key

record

key

satellite data
```
Operations on Dynamic Sets

- **Queries:** Simply return info; **Modifying operations:** Change the set

  - \textbf{INSERT}(S, x): (Modifying) \( S \leftarrow S \cup \{x\} \)
  
  - \textbf{DELETE}(S, x): (Modifying) \( S \leftarrow S - \{x\} \)
  
  - \textbf{MAX}(S) / \textbf{MIN}(S): (Query) return \( x \in S \) with the largest/smallest \( key \)
  
  - \textbf{EXTRACT-MAX}(S) / \textbf{EXTRACT-MIN}(S): (Modifying) return and delete \( x \in S \) with the largest/smallest \( key \)
  
  - \textbf{SEARCH}(S, k): (Query) return \( x \in S \) with \( key[x] = k \)
  
  - \textbf{SUCCESSOR}(S, x) / \textbf{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**($A$, $key$, $n$)

- $n \leftarrow n + 1$
- $i \leftarrow n$
- **while** $i > 1$ **and** $A[\lfloor i/2 \rfloor] < key$ **do**
  - $A[i] \leftarrow A[\lfloor i/2 \rfloor]$
  - $i \leftarrow \lfloor i/2 \rfloor$
- $A[i] \leftarrow key$
Example: **HEAP-INSERT(A, 15)**

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

1. \( n \leftarrow n + 1 \)
2. \( i \leftarrow n \)
3. **while** \( i > 1 \) **and** \( A\lfloor i/2 \rfloor < key \) **do**
   1. \( A[i] \leftarrow A\lfloor i/2 \rfloor \)
   2. \( i \leftarrow \lfloor i/2 \rfloor \)
4. \( A[i] \leftarrow key \)

---

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

- **Key = 15**
- **Heap Structure**
  - 16
  - 14
  - 10
  - 7
  - 9
  - 3
  - 2
  - 4
  - 8
  - 1
  - 5
  - 6
  - 11

---

CS 473 – Lecture 8
Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University
Example: \textbf{HEAP-INSERT}(A, 15)

\textbf{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*}

key = 15
Example: \textsc{HEAP-INSERT}(A, 15)

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

**HEAP-INSERT**\( (A, \text{key}, n) \)

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

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

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

\[
\begin{align*}
n &\leftarrow n + 1 \\
i &\leftarrow n \\
\text{while } i > 1 \text{ and } A\lfloor i/2 \rfloor < \text{key} \text{ do} \\
&\quad A[i] \leftarrow A\lfloor i/2 \rfloor \\
&\quad i \leftarrow \lfloor i/2 \rfloor \\
&\quad A[i] \leftarrow \text{key}
\end{align*}
\]
Heap Increase Key

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

```plaintext
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
```
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)

1. if key < A[i] then
   
   return error

2. while i > 1 and A[⌊i/2⌋] < key do

   A[i] ← A[⌊i/2⌋]

   i ← ⌊i/2⌋

3. A[i] ← key

key = 15
Example: \textbf{HEAP-INCREASE-KEY}(A, 9, 15)

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

\begin{algorithm}
\textbf{HEAP-INCREASE-KEY}(A, i, key)
\begin{algorithmic}
\If{key < A[i]}
\State \textbf{return} error
\EndIf
\While{i > 1 \And A\lfloor i/2 \rfloor < key}
\State A[i] \gets A\lfloor i/2 \rfloor
\State i \gets \lfloor i/2 \rfloor
\EndWhile
\State A[i] \gets key
\end{algorithmic}
\end{algorithm}

\begin{figure}
\centering
\begin{tikzpicture}
\node (1) at (0,0) {1};
\node (2) at (-1,-1) {2};
\node (4) at (-2,-2) {4};
\node (14) at (-3,-3) {14};
\node (16) at (-4,-4) {16};
\node (3) at (1,-1) {3};
\node (5) at (2,-2) {7};
\node (6) at (3,-3) {9};
\node (7) at (4,-4) {10};
\node (8) at (-1.5,-5) {8};
\node (2) at (-0.5,-5) {8};
\node (1) at (0.5,-5) {1};
\node (9) at (2.5,-5) {9};
\node (10) at (3.5,-5) {10};
\node (14) at (-2.5,-5) {14};
\node (1) at (-3.5,-5) {16};
\node (16) at (-4.5,-5) {3}
\draw[->] (1) -- (2);
\draw[->] (2) -- (4);
\draw[->] (4) -- (14);
\draw[->] (14) -- (16);
\draw[->] (3) -- (5);
\draw[->] (5) -- (6);
\draw[->] (6) -- (7);
\draw[->] (7) -- (10);
\draw[->] (8) -- (2);
\draw[->] (2) -- (14);
\draw[->] (14) -- (1);
\draw[->] (9) -- (10);
\draw[->] (10) -- (16);
\node (key) at (0,-6) {key = 15};
\end{tikzpicture}
\end{figure}
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
Heap Implementation of PQ

<table>
<thead>
<tr>
<th>Storage in Application</th>
<th>Heap Storage</th>
<th>Abstract Heap Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>data</td>
<td>H-index</td>
</tr>
<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>

Abstract Heap Representation:

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 14
- 8
- 7
- 3
- 16
- 2
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(lgn)

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(lgn)
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:

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

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

**Execution:**

- **When the operator wants to answer a phone:**
  
  \[
  \text{id} = A[1].\text{id} \\
  \text{Decrease-Key}(A, 1, A[1].\text{key}-1) \\
  \text{answer phone with index } \text{id}
  \]

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