Introduction to Algorithms and Data Structures

Heapsort
Sorting algorithms so far
Sorting algorithms so far

- **Insertsort** (or **Insertionsort**)  
  - Worst case: $\Theta(n^2)$  
  - Best case: $\Theta(n)$
Sorting algorithms so far

- **Insertsort** (or **Insertionsort**)
  - Worst case: $\Theta(n^2)$
  - Best case: $\Theta(n)$

- **Mergesort**
  - Worst case: $\Theta(n \lg n)$
  - Best case: $\Theta(n \lg n)$
A simple sorting algorithm
A simple sorting algorithm

- Selectsort (or Selectionsort)
A simple sorting algorithm

- **Selectsort** (or **Selectionsort**)
  - “Scan” the array to find the maximum element.
A simple sorting algorithm

- Selectsort (or Selectionsort)
  - “Scan” the array to find the maximum element.
  - Put the maximum element at the end of the array.
A simple sorting algorithm

- **Selectsort** (or **Selectionsort**)
  - “Scan” the array to find the maximum element.
  - Put the maximum element at the end of the array.
  - Repeat for the part of the array that has not been sorted.
Selectsort Example

1 2 3 4 5 6 7 8 9 10

4 1 3 2 16 9 10 14 8 7
Selectsort Example
Selectsort Example
Selectsort Example

1 2 3 4 5 6 7 8 9 10

4 1 3 2 16 9 10 14 8 7
Selectsort Example
Select sort Example
Selectsort Example

4 1 3 2 16 9 10 14 8 7
Selectsort Example
Selectsort Example
Selectsort Example

1  2  3  4  5  6  7  8  9  10
4  1  3  2  16  9  10  14  8  7
Selectsort Example
Selectsort Example

1  2  3  4  5  6  7  8  9  10

4 1 3 2 16 9 10 14 8 7
Selectsort Example

1 2 3 4 5 6 7 8 9 10

4 1 3 2 16 9 10 14 8 7
Selectsort Example
Selectsort Example

4 1 3 2 16 9 10 14 8 7
Selectsort Example

1 2 3 4 5 6 7 8 9 10

4 1 3 2 16 9 10 14 8 7
Selectsort Example
Selectsort Example
Selectsort Example
Selectsort Example
Selectsort Example

1 2 3 4 5 6 7 8 9 10

4 1 3 2 7 9 10 14 8 16
Selectsort Example
Selectsort Example
Selectsort Example
## Selectsort Example

<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>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>8</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Selectsort Example

4 1 3 2 7 9 10 14 8 16
Selectsort Example

4 1 3 2 7 9 10 14 8 16
Selectsort Example
Selectsort Example
Selectsort Example
Selectsort Example

4 1 3 2 7 9 10 14 8 16
Selectsort Example
Selectsort Example

4 1 3 2 7 9 10 8 14 16
Selectsort Example

• What is the worst-case running time of Selectsort?
What is the worst-case running time of Selectsort?

What is the best case-running time of Selectsort?
Selectsort Example

- What is the worst-case running time of Selectsort?
- What is the best case-running time of Selectsort?
- Let’s try it on wooclap!
Selectsort Running time

• Worst-case: $\Theta(n^2)$

• Average-case: $\Theta(n^2)$

• Best-case: $\Theta(n^2)$

• Why is this happening?
Selectsort Running time

- Worst-case: $\Theta(n^2)$
- Average-case: $\Theta(n^2)$
- Best-case: $\Theta(n^2)$
- Why is this happening?
  - For $\Omega(n)$ iterations, we need $\Omega(n)$ comparisons to find the maximum element.
What if we could...
What if we could...

• …find the maximum element in $O(1)$ time?
What if we could…

• …find the maximum element in $O(1)$ time?

• That would give us a sorting algorithm with $O(n)$ running time…
What if we could...

• ...find the maximum element in $O(1)$ time?

• That would give us a sorting algorithm with $O(n)$ running time...

• ...which is in general not possible!
What if we could...
What if we could…

• …find the maximum element in $O(1)$ time…
What if we could...

- ...find the maximum element in $O(1)$ time...
- ... after we preprocess the array a little bit...
What if we could...

• …find the maximum element in $O(1)$ time...

• … after we preprocess the array a little bit…

• … and after we process it again after each iteration …
What if we could...

• …find the maximum element in $O(1)$ time...

• … after we preprocess the array a little bit…

• … and after we process it again after each iteration …

• … without using too many comparisons/operations?
Heapsort (very informally)
Heapsort (very informally)

1. Preprocess the array to become a special array.
Heapsort (very informally)

1. Preprocess the array to become a special array.

2. Find the maximum element of the special array in $O(1)$ time and move it to the last position.
Heapsort (very informally)

1. Preprocess the array to become a *special array*.

2. Find the maximum element of the special array in $O(1)$ time and move it to the last position.

3. Consider the remaining array and process it to become a special array again.
Heapsort (very informally)

1. Preprocess the array to become a special array.

2. Find the maximum element of the special array in $O(1)$ time and move it to the last position.

3. Consider the remaining array and process it to become a special array again.

4. Repeat Steps 2-4 for the remaining special array, until the remaining array has size 0.
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.


Attribute $A$\text{.heap-size}, so we can access $A[1 : A$.heap-size]
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.


Attribute $A$.heap-size, so we can access $A[1:A$.heap-size$]$

Parent($i$) return $\lfloor i/2 \rfloor$

Left($i$) return $2i$

Right($i$) return $2i + 1$
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.


Attribute $A.\text{heap-size}$, so we can access $A[1 : A.\text{heap-size}]$.

- Parent$(i)$
  \[ \text{return } \lfloor i/2 \rfloor \]
- Left$(i)$
  \[ \text{return } 2i \]
- Right$(i)$
  \[ \text{return } 2i + 1 \]
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.


Attribute $A$.heap-size, so we can access $A[1 : A$.heap-size$]$

```
Parent(i) return ⌊i/2⌋
Left(i) return 2i
Right(i) return 2i + 1
```
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.


Attribute $A$.heap-size, so we can access $A[1 : A$.heap-size$]$

Parents:

- $A[1]$ = return $i/2$

Lefts:

- return $2i$

Rights:

- return $2i + 1$
The Heap Data Structure

An almost complete binary tree.

All levels completely filled, except possibly the last one, which is filled from the left to the right.


Parent($i$)
return $\lfloor i/2 \rfloor$

Left($i$)
return $2i$

Right($i$)
return $2i + 1$

These can be done in $O(1)$ time.
Height of a binary tree

\[ \text{height}(i) = \text{# edges on the longest simple path to a leaf} \]

\[ \text{height}(T) = \text{height(root)} \]
height of a heap

height(i) = # edges on the longest simple path to a leaf

height(Tree) = height(root)
The height of a heap on $n$ nodes is $\Theta(\lg n)$ (in particular, $\lfloor \lg n \rfloor$).

\[
\text{height}(i) = \# \text{ edges on the longest simple path to a leaf}
\]

\[
\text{height}(\text{Tree}) = \text{height}(\text{root})
\]
(Max) Heap Property

The value of a node is at most the value of its parent, i.e.,

$$A[\text{Parent}(i)] \geq A[i].$$
(Max) Heap Property

The value of a node is at most the value of its parent, i.e.,

$$A[\text{Parent}(i)] \geq A[i].$$
(Max) Heap Property

The value of a node is at most the value of its parent, i.e.,

\[ A[\text{Parent}(i)] \geq A[i]. \]

This will be our special array.
Heapsort (a bit less informally)
Heapsort (a bit less informally)

1. Preprocess the array to become a heap.
Heapsort (a bit less informally)

1. Preprocess the array to become a heap.

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position (how?)
Heapsort (a bit less informally)

1. **Preprocess** the array to become a *heap*.

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position (*how?*)

3. Consider the remaining array and **process** it to become a heap again.
Heapsort (a bit less informally)

1. **Preprocess** the array to become a *heap*.

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position (*how?*)

3. Consider the remaining array and **process** it to become a heap again.

4. Repeat Steps 2-4 for the remaining special array, until the remaining array has size 0.
Max-Heapify
3. Consider the remaining array and process it to become a heap again.
Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).
Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).

Precondition: Trees rooted at Left(i) and Right(i) are heaps.
Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify(A, i).

Precondition: Trees rooted at Left(i) and Right(i) are heaps.

Postcondition: The tree routed at i is a heap.
Max-Heapify

3. Consider the remaining array and process it to become a heap again.

Max-Heapify($A, i$).

**Precondition:** Trees rooted at $\text{Left}(i)$ and $\text{Right}(i)$ are heaps.

**Postcondition:** The tree rooted at $i$ is a heap.
Max-Heapify

```
MAX-HEAPIFY(A, i)
1    l = LEFT(i)
2    r = RIGHT(i)
4        largest = l
5    else largest = i
7        largest = r
8    if largest ≠ i
9        exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```
Max-Heapify

Max-Heapify$(A, i) \quad i = 2, \quad A[i] = 4$

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$ 
   largest = l
4. else largest = i
5. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$ 
   largest = r
6. if largest $\neq i$
   exchange $A[i]$ with $A[\text{largest}]$
7. Max-Heapify$(A, \text{largest})$
Max-Heapify

Max-Heapify($A, i$)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$
   4. $\text{largest} = l$
5. else $\text{largest} = i$
6. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
   7. $\text{largest} = r$
8. if $\text{largest} \neq i$
   9. exchange $A[i]$ with $A[\text{largest}]$
10. $\text{MAX-HEAPIFY}(A, \text{largest})$
Max-Heapify

Max-Heapify \((A, i)\)  
\[i = 2, \ A[i] = 4\]

1. \(l = \text{LEFT}(i)\)  
2. \(r = \text{RIGHT}(i)\)
3. \(\text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i]\)
   \(\quad \text{largest} = l\)
4. \(\text{else } \text{largest} = i\)
5. \(\text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}]\)
   \(\quad \text{largest} = r\)
6. \(\text{if } \text{largest} \neq i\)
   \(\quad \text{exchange } A[i] \text{ with } A[\text{largest}]\)
7. \(\text{MAX-HEAPIFY}(A, \text{largest})\)
Max-Heapify

Max-Heapify($A, i$)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. If $l \leq A.\text{heap-size}$ and $A[l] > A[i]$, set $\text{true}$ and $\text{largest} = l$
4. Else, set $\text{largest} = i$
5. If $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$, set $\text{true}$ and $\text{largest} = r$
6. If $\text{largest} \neq i$, exchange $A[i]$ with $A[\text{largest}]$
7. Max-Heapify($A, \text{largest}$)
Max-Heapify

Max-Heapify\((A, i)\)  
\(i = 2, \ A[i] = 4\)

1. \(l = \text{LEFT}(i)\) →
2. \(r = \text{RIGHT}(i)\) →
3. if \(l \leq A.\text{heap-size}\) and \(A[l] > A[i]\) true
   largest = \(l\)
   largest = 4
4. else largest = \(i\)
5. if \(r \leq A.\text{heap-size}\) and \(A[r] > A[\text{largest}]\)
   largest = \(r\)
6. if largest ≠ \(i\)
   exchange \(A[i]\) with \(A[\text{largest}]\)
7. MAX-HEAPIFY\((A, \text{largest})\)
Max-Heapify

**Algorithm:**

1. Let $l =$ LEFT$(i)$
2. Let $r =$ RIGHT$(i)$
3. If $l \leq A\.heapsize$ and $A[l] > A[i]$ then $true$
   4. Otherwise, $largest = l$
   5. Else $largest = i$
6. If $r \leq A\.heapsize$ and $A[r] > A[largest]$ then $false$
   7. Otherwise, $largest = r$
8. If $largest \neq i$ then exchange $A[i]$ with $A[largest]$
9. MAX-HEAPIFY($A, largest$)
Max-Heapify

Max-Heapify(A, i)

1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. if \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \) then true
   largest = l
   largest = 4
4. else largest = i
5. if \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \) then false
   largest = r
6. if \( \text{largest} \neq i \) then true
7. exchange \( A[i] \) with \( A[\text{largest}] \)
8. MAX-HEAPIFY(A, largest)
Max-Heapify

Max-Heapify \((A, i)\)  
\(i = 2, \ A[i] = 4\)

1.  
2.  
3. \(\text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \text{ true}\)
   
4. \(\text{largest = } l\)
   
5. \(\text{else largest = } i\)

6. \(\text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \text{ false}\)
   
7. \(\text{largest = } r\)

8. \(\text{if largest } \neq i \text{ true}\)
   
9. \(\text{exchange } A[i] \text{ with } A[\text{largest}] \rightarrow\)

10. \(\text{MAX-HEAPIFY}(A, \text{largest})\)
Max-Heapify

Max-Heapify(A, i)

1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. if \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \) true
   largest = l
   largest = 4
4. else largest = i
5. if \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \) false
   largest = r
6. if largest \( \neq i \) true
7. exchange \( A[i] \) with \( A[\text{largest}] \)
8. \( \text{MAX-HEAPIFY}(A, \text{largest}) \)
Max-Heapify

```
Max-Heapify(A,i)  i = 2,  A[i] = 4
1  l = LEFT(i)  →
2  r = RIGHT(i)  →
   largest = l  largest = 4
4  else largest = i
6     largest = r
7  if largest ≠ i  true
8     exchange A[i] with A[largest]  →
9  MAX-HEAPIFY(A, largest)
10```

```
Max-Heapify (cont)

```
MAX-HEAPIFY(A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
   largest = l
4 else largest = i
6   largest = r
7 if largest ≠ i
8   exchange A[i] with A[largest]
9  MAX-HEAPIFY(A, largest)
```
Max-Heapify (cont)

```
MAX-HEAPIFY(A, i)  \( i = 4 \),  \( A[i] = 4 \)
1  \( l = \text{LEFT}(i) \)
2  \( r = \text{RIGHT}(i) \)
3  \text{if} \ l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \text{ then}
4    \( \text{largest} = l \)
5  \text{else} \ \( \text{largest} = i \)
6  \text{if} \ r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \text{ then}
7    \( \text{largest} = r \)
8  \text{if} \ \text{largest} \neq i \text{ then}
9    \text{exchange } A[i] \text{ with } A[\text{largest}] \)
10  \text{MAX-HEAPIFY}(A, \text{largest})
```
Max-Heapify (cont)

\[
\begin{array}{c}
\text{Max-Heapify}(A, i) \quad i = 4, \quad A[i] = 4 \\
1 \quad l = \text{LEFT}(i) \quad \rightarrow \\
2 \quad r = \text{RIGHT}(i) \\
3 \quad \text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \\
4 \quad \quad \text{largest} = l \\
5 \quad \text{else } \quad \text{largest} = i \\
6 \quad \text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \\
7 \quad \quad \text{largest} = r \\
8 \quad \text{if } \text{largest} \neq i \\
9 \quad \quad \text{exchange } A[i] \text{ with } A[\text{largest}] \\
10 \quad \text{Max-Heapify}(A, \text{largest})
\end{array}
\]
Max-Heapify (cont)

MAX-HEAPIFY$(A, i)$  
\[ i = 4, \ A[i] = 4 \]

1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. If \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \)
4. \( \text{largest} = l \)
5. Else \( \text{largest} = i \)
6. If \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \)
7. \( \text{largest} = r \)
8. If \( \text{largest} \neq i \)
9. Exchange \( A[i] \) with \( A[\text{largest}] \)
10. \( \text{MAX-HEAPIFY}(A, \text{largest}) \)
Max-Heapify (cont)

Max-Heapify \((A, i)\)  
\(i = 4, \quad A[i] = 4\)

1. \(l = \text{LEFT}(i)\)
2. \(r = \text{RIGHT}(i)\)
3. \(\text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \text{ false} \)
4. \(\quad \text{largest} = l\)
5. \(\text{else } \text{largest} = i\)
6. \(\text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}]\)
7. \(\quad \text{largest} = r\)
8. \(\text{if } \text{largest} \neq i\)
9. \(\quad \text{exchange } A[i] \text{ with } A[\text{largest}]\)
10. \(\text{MAX-HEAPIFY}(A, \text{largest})\)
**Max-Heapify (cont)**

Max-Heapify algorithm:

1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. if \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \) false
   
   largest = \( l \)
4. else largest = \( i \)
5. if \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \) true
   
   largest = \( r \)
6. if largest \( \neq i \)
   
   exchange \( A[i] \) with \( A[\text{largest}] \)
7. MAX-HEAPIFY(\( A, \text{largest} \) )
Max-Heapify (cont)

```
MAX-HEAPIFY(A, i)  i = 4,  A[i] = 4
1  l = LEFT(i)  →
2  r = RIGHT(i)  →
   largest = l
4  else largest = i
6      largest = r
7      largest = 9
8  if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```
Max-Heapify (cont)

```
MAX-HEAPIFY(A, i)  i = 4, A[i] = 4
1  l = LEFT(i)  →
2  r = RIGHT(i)  →
4    largest = l
5  else largest = i
7    largest = r    largest = 9
8  if largest ≠ i  true
9  exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Max-Heapify (cont)

MAX-HEAPIFY \(A, i\)  \(i = 4, \ A[i] = 4\)

1. \(l = \text{LEFT}(i)\)  
2. \(r = \text{RIGHT}(i)\)  
3. if \(l \leq A.\text{heap-size} \text{ and } A[l] > A[i]\) false
   largest = \(l\)  
4. else largest = \(i\)  
5. if \(r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}]\) true
   largest = \(r\)  
6. if largest \(\neq i\) true
7. exchange \(A[i]\) with \(A[\text{largest}]\)  
8. MAX-HEAPIFY \((A, \text{largest})\)
Max-Heapify (cont)

Max-Heapify($A, i$)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$ then false
   largest = $l$
4. else largest = $i$
5. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$ then true
   largest = $r$
   largest = $9$
6. if $\text{largest} \neq i$ then true
7. exchange $A[i]$ with $A[\text{largest}]$
8. MAX-HEAPIFY($A, \text{largest}$)

Given: $i = 4, A[i] = 4$

Diagram:

- Node 4 is the node to be heapified with children 8 and 9.
- The largest child of 4 is 9.
- Call MAX-HEAPIFY($A, 9$).
Max-Heapify (cont)

Our tree is now a max-heap!
Max-Heapify running time

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
4     largest = l
5  else largest = i
7     largest = r
8  if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
4       largest = l
5   else largest = i
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)  \( O(1) \)
2  r = RIGHT(i)
3  if \( l \leq A.heap-size \) and \( A[l] > A[i] \)
4      largest = l
5  else largest = i
6  if \( r \leq A.heap-size \) and \( A[r] > A[largest] \)
7      largest = r
8  if largest \( \neq i \)
9      exchange \( A[i] \) with \( A[largest] \)
10     MAX-HEAPIFY(A, largest)
```
What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY (A, i)
1 l = LEFT(i)  O(1)
2 r = RIGHT(i) O(1)
   largest = l
4 else largest = i
6   largest = r
7 if largest ≠ i
8   exchange A[i] with A[largest]
9 MAX-HEAPIFY (A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)  \(O(1)\)
2  r = RIGHT(i) \(O(1)\)
3  if \(l \leq A.\text{heap-size} \) and \(A[l] > A[i]\) \(O(1)\)
4     largest = l
5  else largest = i
6  if \(r \leq A.\text{heap-size}\) and \(A[r] > A[\text{largest}]\)
7     largest = r
8  if largest \(\neq i\)
9     exchange \(A[i]\) with \(A[\text{largest}]\)
10    MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)   O(1)
2   r = RIGHT(i) O(1)
4       largest = l O(1)
5   else largest = i
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)    O(1)
2  r = RIGHT(i)  O(1)
4     largest = l    O(1)
5   else largest = i    O(1)
7     largest = r
8   if largest ≠ i
9     exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)   O(1)
2  r = RIGHT(i) O(1)
4      largest = l O(1)
5  else largest = i O(1)
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)  O(1)
2  r = RIGHT(i)  O(1)
4    largest = l  O(1)
5  else largest = i  O(1)
7    largest = r  O(1)
8  if largest ≠ i
9    exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

Max-Heapify(A, i)
1  l = LEFT(i)  $O(1)$
2  r = RIGHT(i)  $O(1)$
3  if $l \leq A$.heap-size and $A[l] > A[i]$  $O(1)$
4      largest = l  $O(1)$
5  else largest = i  $O(1)$
6  if $r \leq A$.heap-size and $A[r] > A[largest]$  $O(1)$
7      largest = r  $O(1)$
8  if largest $\neq i$  $O(1)$
9      exchange $A[i]$ with $A[largest]$
10     MAX-HEAPIFY(A, largest)
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

```plaintext
MAX-HEAPIFY(A, i)
1   l = LEFT(i)  O(1)
2   r = RIGHT(i) O(1)
4     largest = l O(1)
5 else largest = i O(1)
7     largest = r O(1)
8 if largest ≠ i O(1)
10 MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)    O(1)
2  r = RIGHT(i)  O(1)
4     largest = l  O(1)
5  else largest = i  O(1)
7     largest = r  O(1)
8  if largest ≠ i  O(1)
10    MAX-HEAPIFY(A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node $i$. 

```
MAX-HEAPIFY (A, i)
1  l = LEFT(i)  $O(1)$
2  r = RIGHT(i) $O(1)$
4      largest = l  $O(1)$
5  else largest = i  $O(1)$
6  if $r \leq A.heap-size$ and $A[r] > A[largest]$  $O(1)$
7      largest = r  $O(1)$
8  if largest $\neq i$  $O(1)$
10 MAX-HEAPIFY (A, largest)
```
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node $i$. 

Max-Heapify($A, i$) $T(h)$

1. $l = \text{LEFT}(i)$ $O(1)$
2. $r = \text{RIGHT}(i)$ $O(1)$
   \hspace{1cm} \text{largest} = l$ $O(1)$
4. \textbf{else} $\text{largest} = i$ $O(1)$
5. \textbf{if} $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$ $O(1)$
   \hspace{1cm} \text{largest} = r$ $O(1)$
6. \textbf{if} $\text{largest} \neq i$ $O(1)$
   \hspace{1cm} \text{exchange $A[i]$ with $A[\text{largest}]$}$ $O(1)$
7. \hspace{1cm} \text{MAX-HEAPIFY($A, \text{largest}$)}
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node $i$. 

```
MAX-HEAPIFY(A, i)  T(h)
1  l = LEFT(i)  O(1)
2  r = RIGHT(i)  O(1)
    largest = l  O(1)
4  else largest = i  O(1)
6    largest = r  O(1)
7  if largest ≠ i  O(1)
8    exchange A[i] with A[largest]  O(1)
9  MAX-HEAPIFY(A, largest)  T(h − 1)
```
What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in \( O(1) \) time.

Plus the time needed for the recursive call of Max-Heapify on the child of node \( i \).

\[
T(h) \leq \begin{cases} 
T(h - 1) + O(1), & \text{if } h \geq 1 \\
O(1) & \text{if } h = 0 
\end{cases}
\]
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node $i$.

$$T(h) \leq (h + 1) \cdot O(1) = O(h) = O(\lg n)$$

```
MAX-HEAPIFY(A, i)  T(h)
1  l = LEFT(i)  O(1)
2  r = RIGHT(i)  O(1)
4      largest = l  O(1)
5  else largest = i  O(1)
7      largest = r  O(1)
8  if largest ≠ i  O(1)
10     MAX-HEAPIFY(A, largest)  T(h - 1)
```
Build-Max-Heap

1. Preprocess the array to become a heap.
Build-Max-Heap

1. *Preprocess the array to become a heap.*

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.
**Build-Max-Heap**

**Idea:** Apply Max-Heapify repeatedly until the tree becomes a heap.
Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Where do we start?
Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Where do we start?

Can we start here?

Can we start here?
Max-Heapify

Max-Heapify\((A, i)\).

**Precondition:** Trees rooted at \text{Left}(i) and \text{Right}(i) are heaps.

**Postcondition:** The tree rooted at \(i\) is a heap.

\[
\text{MAX-HEAPIFY}(A, i) \\
1 \quad l = \text{LEFT}(i) \\
2 \quad r = \text{RIGHT}(i) \\
3 \quad \textbf{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \\
4 \quad \quad \text{largest} = l \\
5 \quad \textbf{else } \text{largest} = i \\
6 \quad \textbf{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \\
7 \quad \quad \text{largest} = r \\
8 \quad \textbf{if } \text{largest} \neq i \\
9 \quad \quad \text{exchange } A[i] \text{ with } A[\text{largest}] \\
10 \quad \text{MAX-HEAPIFY}(A, \text{largest})
\]

CLRS pp 165
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Where do we start?
Can we start here?
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.
Build-Max-Heap

Idea: Apply Max-Heapify repeatedly until the tree becomes a heap.

Leaves are (trivially) heaps!
Observation: The elements of the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are leaves.
Build-Max-Heap

Observation: The elements of the subarray $A[[\lfloor n/2 \rfloor + 1 : n]$ are leaves.
Build-Max-Heap

Observation: The elements of the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are leaves.
Build-Max-Heap

BUILD-MAX-HEAP(A, n)
1. A.heap-size = n
2. for i = ⌊n/2⌋ downto 1
3. MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1. l = LEFT(i)
2. r = RIGHT(i)
   4. largest = l
5. else largest = i
   7. largest = r
8. if largest ≠ i
10. MAX-HEAPIFY(A, largest)
Build-Max-Heap

```plaintext
BUILD-MAX-HEAP(A, n)
1 A.heap-size = n
2 for i = ⌊n/2⌋ downto 1
3    MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
4    largest = l
5 else largest = i
7    largest = r
8 if largest ≠ i
9    exchange A[i] with A[largest]
10   MAX-HEAPIFY(A, largest)
```
Build-Max-Heap

**Build-Max-Heap**

```plaintext
A.heap-size = n
for i = \lfloor n/2 \rfloor \text{ downto 1}
    MAX-HEAPIFY(A, i)
```

**MAX-HEAPIFY**

```plaintext
l = LEFT(i)
r = RIGHT(i)
    largest = l
else largest = i
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```
Build-Max-Heap

BUILD-MAX-HEAP(A, n)
1. A.heap-size = n
2. for i = [n/2] downto 1
3. MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1. l = LEFT(i)
2. r = RIGHT(i)
4. largest = l
5. else largest = i
7. largest = r
8. if largest ≠ i
10. MAX-HEAPIFY(A, largest)
Build-Max-Heap

CLRS pp 167

BUILD-MAX-HEAP(A, n)
1. A.heap-size = n
2. for i = ⌊n/2⌋ downto 1
   3. MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1. l = LEFT(i)
2. r = RIGHT(i)
   4. largest = l
5. else largest = i
   7. largest = r
8. if largest ≠ i
10. MAX-HEAPIFY(A, largest)
Build-Max-Heap

CLRS pp 167

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

1. \(A\.heap\-size = n\)
2. for \(i = \lceil n/2 \rceil \) down to 1
3. \(MAX\-HEAPIFY(A, i)\)

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

1. \(l = \text{LEFT}(i)\)
2. \(r = \text{RIGHT}(i)\)
3. if \(l \leq A\.heap\-size\) and \(A[l] > A[i]\)
4. \(\text{largest} = l\)
5. else \(\text{largest} = i\)
6. if \(r \leq A\.heap\-size\) and \(A[r] > A[\text{largest}]\)
7. \(\text{largest} = r\)
8. if \(\text{largest} \neq i\)
9. exchange \(A[i]\) with \(A[\text{largest}]\)
10. \(MAX\-HEAPIFY(A, \text{largest})\)
Build-Max-Heap

\[
\begin{align*}
A.\text{heap-size} &= n \\
\text{for } i &= \lfloor n/2 \rfloor \text{ downto } 1 \\
\text{MAX-HEAPIFY}(A, i)
\end{align*}
\]

\[
\begin{align*}
\text{MAX-HEAPIFY}(A, i) \\
1 & l = \text{LEFT}(i) \\
2 & r = \text{RIGHT}(i) \\
3 & \text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \\
4 & \text{largest} = l \\
5 & \text{else } \text{largest} = i \\
6 & \text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \\
7 & \text{largest} = r \\
8 & \text{if } \text{largest} \neq i \\
9 & \text{exchange } A[i] \text{ with } A[\text{largest}] \\
10 & \text{MAX-HEAPIFY}(A, \text{largest})
\end{align*}
\]
Build-Max-Heap

CLRS pp 167

BUILD-MAX-HEAP(A, n)
1   A.heap-size = n
2   for i = [n/2] downto 1
3       MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
4       largest = l
5   else largest = i
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
Build-Max-Heap

\[
\begin{align*}
\text{BUILD-MAX-HEAP}(A, n) \\
&\quad A.\text{heap-size} = n \\
&\quad \text{for } i = \lceil n/2 \rceil \text{ downto } 1 \\
&\quad \text{MAX-HEAPIFY}(A, i)
\end{align*}
\]

\[
\begin{align*}
\text{MAX-HEAPIFY}(A, i) \\
&\quad l = \text{LEFT}(i) \\
&\quad r = \text{RIGHT}(i) \\
&\quad \text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \\
&\quad \quad \text{largest} = l \\
&\quad \text{else largest} = i \\
&\quad \text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \\
&\quad \quad \text{largest} = r \\
&\quad \text{if } \text{largest} \neq i \\
&\quad \quad \text{exchange } A[i] \text{ with } A[\text{largest}] \\
&\quad \text{MAX-HEAPIFY}(A, \text{largest})
\end{align*}
\]
**Build-Max-Heap**

```
BUILD-MAX-HEAP(A, n)
1 A.heap-size = n
2 for i = ⌊n/2⌋ downto 1
3   MAX-HEAPIFY(A, i)
```

```
MAX-HEAPIFY(A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
4   largest = l
5 else largest = i
7   largest = r
8 if largest ≠ i
9   exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Build-Max-Heap

CLRS pp 167
Build-Max-Heap

**Algorithm:**

1. **Build-Max-Heap**($A, n$)
   1. $A.heap-size = n$
   2. for $i = [n/2]$ downto 1
   3. **Max-Heapify**($A, i$)

**Max-Heapify**($A, i$)

1. $l = \text{Left}(i)$
2. $r = \text{Right}(i)$
4.     largest = $l$
5. else largest = $i$
6. if $r \leq A.heap-size$ and $A[r] > A[\text{largest}]$
7.     largest = $r$
8. if $\text{largest} \neq i$
9.     exchange $A[i]$ with $A[\text{largest}]$
10. **Max-Heapify**($A, \text{largest}$)

**Figure:**

A max-heap with values 4, 16, 10, 14, 7, 9, 3, 2, 8, 1. The algorithm is illustrated with the tree and the steps of heapifying nodes.
Build-Max-Heap

CLRS pp 167

BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = [n/2] downto 1
3  MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
4     largest = l
5  else largest = i
7     largest = r
8  if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
Build-Max-Heap

CLRS pp 167

BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
4     largest = l
5  else largest = i
7     largest = r
8  if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
## Build-Max-Heap

### Algorithm

BUILD-MAX-HEAP($A$, $n$)
1. $A$.heap-size = $n$
2. for $i = \lfloor n/2 \rfloor$ downto 1
3. MAX-HEAPIFY($A$, $i$)

### MAX-HEAPIFY($A$, $i$)
1. $l$ = LEFT($i$)
2. $r$ = RIGHT($i$)
   4. largest = $l$
   5. else largest = $i$
4. if $r \leq A$.heap-size and $A[r] > A[largest]$
   7. largest = $r$
5. if largest $\neq i$
7. MAX-HEAPIFY($A$, largest)

---

CLRS pp 167
Build-Max-Heap

CLRS pp 167

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

\[
\text{MAX-HEAPIFY}(A, i)
\]
1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. if \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \)
4. \( \text{largest} = l \)
5. else \( \text{largest} = i \)
6. if \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \)
7. \( \text{largest} = r \)
8. if \( \text{largest} \neq i \)
9. exchange \( A[i] \) with \( A[\text{largest}] \)
10. \( \text{MAX-HEAPIFY}(A, \text{largest}) \)
Build-Max-Heap

CLRS pp 167

Build-Max-Heap(A, n)
1. A.heap-size = n
2. for i = [n/2] downto 1
3. Max-Heapify(A, i)

Max-Heapify(A, i)
1. l = LEFT(i)
2. r = RIGHT(i)
   4. largest = l
5. else largest = i
   7. largest = r
8. if largest ≠ i
10. Max-Heapify(A, largest)
Build-Max-Heap
Correctness
Build-Max-Heap
Correctness

We will argue via a loop invariant:
Build-Max-Heap

Correctness

We will argue via a loop invariant:

*At the start of each iteration of the for loop (lines 2-3), each node $i + 1$, $i + 2$, …, $n$ is the root of a max-heap.*
Build-Max-Heap
Correctness

We will argue via a loop invariant:

*At the start of each iteration of the for loop (lines 2-3), each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.*

**Initialisation:** Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and hence trivially max-heaps.
Build-Max-Heap  
Correctness  

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node \( i + 1, i + 2, \ldots, n \) is the root of a max-heap.

**Initialisation:** Prior to the first iteration of the loop \( i = \lfloor n/2 \rfloor \). In that case the nodes \( \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n \) are leaves, and hence trivially max-heaps.

**Maintenance:**
Build-Max-Heap

Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node \( i + 1, i + 2, \ldots, n \) is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop \( i = \lfloor n/2 \rfloor \). In that case the nodes \( \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n \) are leaves, and hence trivially max-heaps.

Maintenance:

Left(\( i \)) and Right(\( i \)) have higher indices than \( i \). By the loop invariant, they are roots of max-heaps.
Build-Max-Heap

Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node $i+1$, $i+2$, ..., $n$ is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop $i = \lfloor n/2 \rfloor$. In that case the nodes $\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor + 1$, ..., $n$ are leaves, and hence trivially max-heaps.

Maintenance:

Left($i$) and Right($i$) have higher indices than $i$. By the loop invariant, they are roots of max-heaps.

The precondition of Max-Heapify($i$) is thus satisfied. By the postcondition, $i$ will be the root of a max-heap. Furthermore, nodes $i + 1$, ..., $n$ are still roots of max-heaps.
Build-Max-Heap

Correctness

We will argue via a loop invariant:

At the start of each iteration of the for loop (lines 2-3), each node \( i + 1, \ i + 2, \ldots, \ n \) is the root of a max-heap.

Initialisation: Prior to the first iteration of the loop \( i = \lfloor n/2 \rfloor \). In that case the nodes \( \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n \) are leaves, and hence trivially max-heaps.

Maintenance:

\( \text{Left}(i) \) and \( \text{Right}(i) \) have higher indices than \( i \). By the loop invariant, they are roots of max-heaps.

The precondition of Max-Heapify(\( i \)) is thus satisfied. By the postcondition, \( i \) will be the root of a max-heap. Furthermore, nodes \( i + 1, \ldots, n \) are still roots of max-heaps.

Termination: The loop obviously terminates, when \( i = 0 \). By the loop invariant, each node is the root of a max-heap, and so is the root.
Build-Max-Heap Running Time
Build-Max-Heap Running Time

Easy bound:
Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$. 
Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time \( O(\lg n) \).

Max-Heapify is called \( O(n) \) times.
Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$. 
Heapsort
Heapsort

1. Preprocess the array to become a heap.
   Build-Max-Heap$(A, n)$
Heapsort

1. Preprocess the array to become a heap.
   Build-Max-Heap(A, n)

2. Find the maximum element of the heap in \(O(1)\) time and move it to the last position.
   exchange \(A[1]\) with \(A[i]\) (where initially \(i = n\)).
Heapsort

1. **Preprocess** the array to become a *heap*.
   
   \[ \text{Build-Max-Heap}(A, n) \]

2. Find the maximum element of the heap in \( O(1) \) time and move it to the last position.
   
   exchange \( A[1] \) with \( A[i] \) (where initially \( i = n \)).

3. Consider the remaining array and **process** it to become a heap again.
   
   \[ A.\text{heap-size} = A.\text{heapsize} - 1 \]
   
   \[ \text{Max-Heapify}(A, 1) \]
Heapsort

1. Preprocess the array to become a heap.
   Build-Max-Heap($A, n$)

2. Find the maximum element of the heap in $O(1)$ time and move it to the last position.

3. Consider the remaining array and process it to become a heap again.
   $A$.heap-size = $A$.heapsize -1
   Max-Heapify($A, 1$)

4. Repeat Steps 2-4 for the remaining special array, until the remaining array has size 0.
Heapsort

**Heapsort**

```plaintext
HEAPSORT(A, n)
1   BUILD-MAX-HEAP(A, n)
2   for i = n downto 2
4   A.heap-size = A.heap-size − 1
5   MAX-HEAPIFY(A, 1)
```

CLRS pp 170
HEAPSORT($A, n$)
1 BUILD-MAX-HEAP($A, n$)
2 for $i = n$ downto 2
4 $A$.heap-size = $A$.heap-size - 1
5 MAX-HEAPIFY($A, 1$)

CLRS pp 170
**Heapsort**

**HEAPSORT**(A, n)
1 BUILD-MAX-HEAP(A, n)
2 for i = n downto 2
4 A.heap-size = A.heap-size - 1
5 MAX-HEAPIFY(A, i)

CLRS pp 170
Heapsort

HEAPSORT(A, n)
1. BUILD-MAX-HEAP(A, n)
2. for i = n downto 2
   4. A.heap-size = A.heap-size - 1
5. MAX-HEAPIFY(A, 1)

CLRS pp 170
Heapsort

Heapsort($A, n$)
1. BUILD-MAX-HEAP($A, n$)
2. for $i = n$ downto 2
4. $A.heap-size = A.heap-size - 1$
5. MAX-HEAPIFY($A, 1$)

CLRS pp 170
Heapsort

Heapsort($A, n$)
1. BUILD-MAX-HEAP($A, n$)
2. for $i = n$ downto 2
   4. $A.heap-size = A.heap-size - 1$
5. MAX-HEAPIFY($A, 1$)
Heapsort

Heapsort($A, n$)
1 BUILD-MAX-HEAP($A, n$)
2 for $i = n$ downto 2
4 $A$.heap-size = $A$.heap-size − 1
5 MAX-HEAPIFY($A, 1$)

CLRS pp 170
Heapsort

HEAPSORT($A, n$)
1 \textbf{BUILD-MAX-HEAP($A, n$)}
2 \textbf{for} $i = n$ \textbf{downto} 2
3 \hspace{1em} exchange $A[1]$ with $A[i]$
4 \hspace{1em} $A$.heap-size = $A$.heap-size - 1
5 \hspace{1em} \textbf{MAX-HEAPIFY($A, i$)}

CLRS pp 170
Heapsort

\textbf{HEAPSORT}(A, n)
1 \textbf{BUILD-MAX-HEAP}(A, n)
2 \textbf{for} i = n \textbf{down to} 2
3 \quad \text{exchange } A[1] \text{ with } A[i]
4 \quad A.\text{heap-size} = A.\text{heap-size} - 1
5 \quad \text{MAX-HEAPIFY}(A, 1)

\text{CLRS pp 170}
Heapsort

**Heapsort**

Heapsort \((A, n)\)
1. **Build-Max-Heap** \((A, n)\)
2. for \(i = n\) downto 2
   3. exchange \(A[1]\) with \(A[i]\)
   4. \(A.heap-size = A.heap-size - 1\)
5. **Max-Heapify** \((A, 1)\)

CLRS pp 170
Heapsort

**Heapsort**

1. **Build-Max-Heap**($A, n$)
2. **for** $i = n$ **down to** 2
4. $A.heap-size = A.heap-size - 1$
5. **Max-Heapify**($A, 1$)

CLRS pp 170
Heapsort

Heapsort($A, n$)
1. BUILD-MAX-HEAP($A, n$)
2. for $i = n$ downto 2
   4. $A$.heap-size = $A$.heap-size – 1
5. MAX-HEAPIFY($A, 1$)
Heapsort

Heapsort\((A, n)\)
1. BUILD-MAX-HEAP\((A, n)\)
2. for \(i = n\) downto 2
   3. exchange \(A[1]\) with \(A[i]\)
   4. \(A.\text{heap-size} = A.\text{heap-size} - 1\)
   5. MAX-HEAPIFY\((A, 1)\)

Full execution: CLRS pp 171
Heapsort Running Time

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
4  A.heap-size = A.heap-size - 1
5  MAX-HEAPIFY(A, 1)
```

CLRS pp 170
Heapsort Running Time

Heapsort ($A, n$)
1. Build-Max-Heap ($A, n$)
2. for $i = n$ downto 2
4. $A.\text{heap-size} = A.\text{heap-size} - 1$
5. Max-Heapify ($A, 1$)

CLRS pp 170

Easy bound:

Max-Heapify has running time $O(\log n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \log n)$. 
Heapsort Running Time

Heapsort

```plaintext
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
4     A.heap-size = A.heap-size - 1
5  MAX-HEAPIFY(A, 1)
```

CLRS pp 170

Running time of Heapsort:

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$. 
Heapsort Running Time

Heapsort \( A, n \)

1. **Build-Max-Heap** \( A, n \)
2. for \( i = n \ to 2 \) do
4. \( A.heap-size = A.heap-size - 1 \)
5. **Max-Heapify** \( A, 1 \)

**Running time of Heapsort:** \( \Theta(n \lg n) \)

Easy bound:

Max-Heapify has running time \( O(\lg n) \).

Max-Heapify is called \( O(n) \) times.

Build-Max-Heap has running time \( O(n \lg n) \).
Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n\lg n)$. 
Build-Max-Heap Running Time

Easy bound:

Max-Heapify has running time $O(\lg n)$.

Max-Heapify is called $O(n)$ times.

Build-Max-Heap has running time $O(n \lg n)$.

Is this really tight?
Build-Max-Heap Running Time, better analysis

Refined bound:
Build-Max-Heap Running Time, better analysis

Refined bound:

For Max-Heapify, we actually proved that
Refined bound:

For Max-Heapify, we actually proved that

\[ T(h) \leq (h + 1) \cdot O(1) = O(h) = O(\lg n) \]
Build-Max-Heap Running Time, better analysis

Refined bound:

For Max-Heapify, we actually proved that

\[ T(h) \leq (h + 1) \cdot O(1) \]

\[ = O(h) = O(\log n) \]

In other words, there is a constant \( c \) such that \( T(h) \leq ch \) for sufficiently large \( h \).
Refined bound:

For Max-Heapify, we actually proved that

\[ T(h) \leq (h + 1) \cdot O(1) \]

\[ = O(h) = O(\log n) \]

In other words, there is a constant \( c \) such that \( T(h) \leq ch \) for sufficiently large \( h \).

Our cost depends on the height of the subtree we are considering!
Build-Max-Heap Running Time, better analysis

Refined bound:

\[
T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil c h
\]

\[
\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} c h
\]

\[
\leq c n \sum_{h=0}^{\infty} \frac{h}{2^h}
\]

\[
\leq c n \cdot \frac{1/2}{(1 - 1/2)^2}
\]

\[
= O(n)
\]
Build-Max-Heap Running Time, better analysis

Refined bound:

\[ T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \]

\[ \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \]

\[ \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \]

\[ \leq cn \cdot \frac{1/2}{(1 - 1/2)^2} \]

\[ = O(n) \]
Build-Max-Heap Running Time, better analysis

Refined bound:

\[ T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{[\lg n]} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \]

\[ \left\lceil \frac{n}{2^{h+1}} \right\rceil \leq \frac{n}{2^h} \]

straightforward

\[ \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \]

\[ \leq cn \cdot \frac{1/2}{(1 - 1/2)^2} = O(n) \]
Build-Max-Heap Running Time, better analysis

Refined bound:

\[ T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil c h \]

\[ \left\lceil \frac{n}{2^{h+1}} \right\rceil \leq \frac{n}{2^h} \]

straightforward

\[ \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \]

\[ \leq cn \cdot \frac{1/2}{(1 - 1/2)^2} \]

\[ = O(n) \]
Build-Max-Heap Running Time, better analysis

Refined bound:

\[ T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lceil \lg n \rceil} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor c^h \]

height(heap) = \lfloor \lg n \rfloor

\[ \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \leq \frac{n}{2^h} \]

straightforward

\[ \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \]

\[ \leq cn \cdot \frac{1/2}{(1 - 1/2)^2} \]

= \(O(n)\)
Build-Max-Heap Running Time, better analysis

Refined bound:

\[ T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor c h \]

There are at most \( \left\lceil \frac{n}{2^{h+1}} \right\rceil \) nodes of any height \( h \)

\[ \left\lceil \frac{n}{2^{h+1}} \right\rceil \leq \frac{n}{2^h} \]

straightforward

\[ \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \]

\[ \leq cn \cdot \frac{1/2}{(1 - 1/2)^2} = O(n) \]
Build-Max-Heap Running Time, better analysis

Refined bound:

\[ T(\text{Build-Max-Heap}(A, n)) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \]

There are at most \( \left\lfloor n/2^{h+1} \right\rfloor \) nodes of any height \( h \)

straightforward

\[ \sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2}, \quad \text{for } |x| < 1, \text{ with } x = 1/2. \]

\[ \leq cn \cdot \frac{1/2}{(1 - 1/2)^2} = O(n) \]

height(heap) = \( \lfloor \lg n \rfloor \)
Heapsort Properties

• Invented by J.W.J. Williams in 1964 (the Heap too!)

• It is an *in-place* algorithm (no auxiliary array).

• It is not a *stable* algorithm (i.e., it does not maintain the relative order between equal keys).

• Whether equal keys are allowed or not influences the best-case running time of the algorithm (think about it!).

• **Next time:** More about the heap - what else is it good for?