CSE 548 / AMS 542: Analysis of Algorithms

Prerequisites Review 5
(Heaps and Heapsort)

Rezaul Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2023
Selection Sort


SELECTION-SORT ( \( A \) )

1. \( \text{for } j = A.\text{length} \text{ downto } 2 \)
2. \( \text{// find the index of an entry with the largest value in } A[1..j] \)
3. \( \text{max} = 1 \)
4. \( \text{for } i = 2 \text{ to } j \)
5. \( \text{if } A[i] > A[\text{max}] \)
6. \( \text{max} = i \)
7. \( \text{// swap } A[j] \text{ and } A[\text{max}] \)
8. \( A[j] \leftrightarrow A[\text{max}] \)

This way of finding the index of an entry with the largest value in a subarray of length \( m \) takes \( \Theta(m) \) time, which is bad!
**Selection Sort**

**SELECTION-SORT** (A)

1. for \( j = A\. \)length downto 2
2. // find the index of an entry with the largest value in \( A[1..j] \)
3. \( \text{max} = 1 \)
4. for \( i = 2 \) to \( j \)
5. \( \text{if } A[i] > A[\text{max}] \)
6. \( \text{max} = i \)
7. // swap \( A[j] \) and \( A[\text{max}] \)
8. \( A[j] \leftrightarrow A[\text{max}] \)

This way of finding the index of an entry with the largest value in a subarray of length \( m \) takes \( \Theta(m) \) time, which is bad!

Let \( L(m) \) be the time needed to find the index of an entry with the largest value in a subarray of length \( m \).

Then running time of **SELECTION-SORT**, \( T(n) = \sum_{2 \leq j \leq n} L(j) \)

\[
= \sum_{2 \leq j \leq n} \Theta(j) = \Theta\left( \sum_{2 \leq j \leq n} j \right) = \Theta(n^2)
\]
Selection Sort

**SELECTION-SORT (A)**

1. *for j = A.length downto 2*
2. // find the index of an entry with the largest value in A[1..j]
3. \( \text{max} = 1 \)
4. *for i = 2 to j*
6. \( \text{max} = i \)
7. // swap A[j] and A[max]
8. \( A[j] \leftrightarrow A[\text{max}] \)

This way of finding the index of an entry with the largest value in a subarray of length \( m \) takes \( \Theta(m) \) time, which is bad!

If we can decrease \( L(m) \), then the running time of SELECTION-SORT will also decrease. For example, if we have \( L(m) = O(\log m) \),

running time of SELECTION-SORT will be,

\[
T(n) = \sum_{2 \leq j \leq n} L(j)
\]

\[
= \sum_{2 \leq j \leq n} O(\log(j)) = O\left(\sum_{2 \leq j \leq n} \log(j)\right) = O(n \log n)
\]

How can you decrease \( L(m) \) to \( O(\log m) \)?
Heap (Binary Heap)

A (binary) heap data structure is an array object that can be viewed as a complete binary tree.

Each node of the tree corresponds to an element of the array.

The tree is completely filled on all levels except possibly the last, which is filled from the left up to a point.

An array $A$ that represents a heap is an object with two attributes:

- $A.length$, which gives the number of elements in the array.
- $A.heapsize$, which represents how many elements in the heap are stored within array $A$.

Though $A[1..A.length]$ may contain numbers, only $A[1..A.heapsize]$ contain valid elements of the heap, where $1 \leq A.heapsize \leq A.length$. 
Parent and Children


A node has 0, 1 or 2 children.

A node with no child is called a leaf.

A node with at least one child is called an internal node.

Given the index $i$ of a node, we can easily compute the indices of its parent, left child and right child.

**PARENT** ($i$)

1. \text{return} \left\lfloor \frac{i}{2} \right\rfloor

**LEFT** ($i$)

1. \text{return} $2i$

**RIGHT** ($i$)

1. \text{return} $2i + 1$
Max-Heap and Min-Heap


**Max-heap.** Each node $i > 1$ satisfies the max-heap property: $A[\text{PARENT}(i)] \geq A[i]$.

Hence, the largest element in a max-heap is stored at the root.

We will use max-heaps in the heapsort algorithm which can be viewed as improved selection sort.

**Min-heap.** Each node $i > 1$ satisfies the min-heap property: $A[\text{PARENT}(i)] \leq A[i]$.

Hence, the smallest element in a min-heap is stored at the root.

Min-heaps commonly implement priority queues which have many applications, e.g., in shortest paths computation.
Height and Levels of a Heap


**Height of a node** = Number of edges on the longest simple downward path from that node to a leaf.

**Height of a heap** = height of its root.

Levels:
- Level of the root, $\text{LEVEL}(1) = 0$
- Level of node $i > 1$, $\text{LEVEL}(i) = \text{LEVEL}(\text{PARENT}(i)) + 1$

A heap of height $h$ has exactly $h + 1$ levels, numbered from 0 to $h$. 
Height of an \( n \)-node Binary Heap

Let \( h \) be the height of a heap containing \( n > 0 \) elements.

So, the heap will have exactly \( h + 1 \) levels.

Let \( n_l \) be the number of nodes at level \( l \), where \( 0 \leq l \leq h \).

Clearly, \( n_l = 2^l \) for \( 0 \leq l \leq h - 1 \),
and \( 1 \leq n_l \leq 2^l \) for \( l = h \).

Also \( n = n_0 + n_1 + \cdots + n_h = \sum_{l=0}^{h} n_l \).
Height of an \( n \)-node Binary Heap

We have, \( n = \sum_{l=0}^{h} n_l = n_h + \sum_{l=0}^{h-1} n_l = n_h + \sum_{l=0}^{h-1} 2^l = n_h + (2^h - 1) \).

But \( 1 \leq n_h \leq 2^h \)

\[ \Rightarrow 1 + (2^h - 1) \leq n_h + (2^h - 1) \leq 2^h + (2^h - 1) \]

\[ \Rightarrow 2^h \leq n \leq 2^{h+1} - 1 \]

\[ \Rightarrow 2^h \leq n < 2^{h+1} \]

\[ \Rightarrow \log_2 n - 1 < h \leq \log_2 n \]

Since \( h \) is an integer, and the only integer \( > \log_2 n - 1 \) and \( \leq \log_2 n \) is \( \lfloor \log_2 n \rfloor \), we have \( h = \lfloor \log_2 n \rfloor \).
Maintaining Heap Property

This is a valid max-heap:
Maintaining Heap Property

But this is not:

Example 1
To fix the max-heap property at $A[1]$:


Example 1
Maintaining Heap Property

To fix the max-heap property at $A[1]$:


Example 1

![Heap Diagram]

$A$

<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>34</td>
<td>41</td>
<td>41</td>
<td>47</td>
<td>38</td>
<td>35</td>
<td>33</td>
<td>19</td>
<td>25</td>
<td>44</td>
<td>31</td>
<td>36</td>
<td>30</td>
<td>18</td>
<td>21</td>
<td>10</td>
<td>5</td>
<td>15</td>
<td>17</td>
<td>8</td>
<td>26</td>
<td>42</td>
<td>27</td>
<td>12</td>
<td>19</td>
<td>30</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Maintaining Heap Property

Maintaining Heap Property

To fix the max-heap property at $A[2]$:


Example 1
To fix the max-heap property at $A[2]$:


### Example 1

The heap tree is shown with the nodes highlighted to indicate the process of maintaining the max-heap property.
Maintaining Heap Property


Example 1
Maintaining Heap Property

To fix the max-heap property at \( A[5] \):


Example 1
Maintaining Heap Property

To fix the max-heap property at $A[5]$:


Example 1

Example 1
Maintaining Heap Property

To fix the max-heap property at $A[11]$:

Maintaining Heap Property

To fix the max-heap property at $A[11]$:

Maintaining Heap Property

This is now a valid max-heap:

Example 1
Maintaining Heap Property

This is another valid max-heap:
Maintaining Heap Property

But this is not:
Maintaining Heap Property

Maintaining Heap Property

To fix the max-heap property at $A[1]:$


**Example 2**
Maintaining Heap Property

To fix the max-heap property at $A[1]$:


Example 2
Maintaining Heap Property

To fix the max-heap property at $A[3]$


Example 2
Maintaining Heap Property

To fix the max-heap property at $A[3]$:


Example 2

![Heap Diagram]

The heap diagram shows a max heap with the values 1 through 47. The process of maintaining the heap property is illustrated by highlighting the elements that need to be swapped to ensure the property is satisfied at the specified index.
Maintaining Heap Property


Example 2

![Example 2 diagram](image-url)
Maintaining Heap Property

This is now a valid max-heap:
Maintaining Heap Property

**Input:** An array $A$ and an index $i$ into the array with the subtrees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps, but $A[i]$ might be smaller than its children and thus violating the max-heap property.

**Output:** Array $A$ with its elements rearranged so that the subtree rooted at index $i$ is a max-heap.

$$\text{MAX-HEAPIFY}(A, i)$$

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. **if** $l \leq A.\text{heapsize}$ and $A[l] > A[i]$
   4. $\text{largest} = l$
5. **else** $\text{largest} = i$
6. **if** $r \leq A.\text{heapsize}$ 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. $\text{MAX-HEAPIFY}(A, \text{largest})$
Building a Max-Heap

The items in $A[1..28]$ are not correctly ordered to form a valid max-heap. We want to reorder them so that they form a valid max-heap.
We reorder the items by calling \texttt{MAX-HEAPIFY( }\texttt{A, i )} on each \texttt{A[i]} starting from \( i = \left\lfloor \frac{A.length}{2} \right\rfloor = \left\lfloor \frac{28}{2} \right\rfloor = 14 \) down to \( i = 1 \).
Building a Max-Heap

\texttt{MAX-HEAPIFY}( A, 14 ):
Building a Max-Heap

MAX-HEAPIFY(A, 14):
Building a Max-Heap

MAX-HEAPIFY( A, 13 ):
Building a Max-Heap

MAX-HEAPIFY( A, 13 ):
Building a Max-Heap

**MAX-HEAPIFY**(\texttt{A}, 12):
Building a Max-Heap

**MAX-HEAPIFY** (A, 12):
Building a Max-Heap

**MAX-HEAPIFY( A, 11 ):**
Building a Max-Heap

**Max-Heapify(A, 11):**
Building a Max-Heap

MAX-HEAPIFY( A, 10):

A

10 31 26 18 47 11 41 12 36 5 49 42 52 38 19 17 44 25 13 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Building a Max-Heap

```
MAX-HEAPIFY(A, 10):
```
**Building a Max-Heap**

MAX-HEAPIFY (A, 9):

```
A = [10, 31, 26, 18, 47, 11, 41, 12, 36, 30, 49, 52, 38, 19, 17, 44, 25, 35, 19, 5, 27, 41, 21, 33, 15, 8, 36]
```
Building a Max-Heap

**Max-Heapify (A, 9):**
Building a Max-Heap

MAX-HEAPIFY(A, 8):
Building a Max-Heap

Max-Heapify(A, 8):

A = [10, 31, 26, 18, 47, 11, 41, 44, 36, 30, 49, 42, 52, 38, 19, 17, 12, 25, 33, 15, 8, 36]
Building a Max-Heap

**MAX-HEAPIFY** (*A*, 7):

![Image of a Max-Heap tree with nodes labeled 1 to 10, 11 to 15, 16 to 20, 21 to 28, and 29 to 41.]
Building a Max-Heap

**MAX-HEAPIFY** \((A, 7)\):
Building a Max-Heap

`MAX-HEAPIFY(A, 6):`
Building a Max-Heap

**Max-Heapify (A, 6):**

```
A = [10, 31, 26, 18, 47, 52, 41, 44, 36, 30, 49, 15, 42, 21, 33, 11, 8, 36]
```
For building a Max-Heap, consider the array $A$ and the operation `MAX-HEAPIFY(A, 5)` as shown in the diagram.
Building a Max-Heap

MAX-HEAPIFY( A, 5 ):
Building a Max-Heap

MAX-HEAPIFY( A, 4 ):
Building a Max-Heap

MAX-HEAPIFY( A, 4 ):
Building a Max-Heap

MAX-HEAPIFY( A, 3 ):
Building a Max-Heap

**MAX-HEAPIFY**( `A`, `3` ):
Building a Max-Heap

**MAX-HEAPIFY** (A, 2):

1. Start with the root node (index 1).
2. Compare the root with its children. If the root has a smaller value, swap the root with the larger child.
3. Recursively apply MAX-HEAPIFY to the subtree rooted at the swapped child.
4. Continue this process until all nodes are correctly ordered to maintain the max-heap property.

The diagram illustrates the process with a max-heap array `A` and the steps to build the heap.

```
A = [10, 31, 52, 44, 49, 42, 41, 18, 36, 30, 47, 33, 15, 38, 19, 17, 12, 25, 35, 19, 5, 27, 41, 21, 26, 11, 8, 36]
```
Building a Max-Heap

**MAX-HEAPIFY\( (A, 2) \):**
### Building a Max-Heap

**Max-Heapify** ($A, 1$):
Building a Max-Heap

**Max-Heapify** (A, 1):
Building a Max-Heap

The items in $A[1..28]$ are now correctly ordered to form a valid max-heap.
Building a Max-Heap


Output: Array $A$ with its elements rearranged so that the entire array is now a max-heap.

**BUILD-MAX-HEAP** ($A$)

1. $A\.heapsize = A\.length$
2. \textbf{for} $i = \lfloor A\.length/2 \rfloor$ \textbf{downto} 1
3. \textbf{Max-Heapify} ($A, i$)
The Heapsort Algorithm

Suppose we are given an array $A[1..28]$ of $n = 28$ numbers.

We want to sort $A[1..28]$ in nondecreasing order of value.
The Heapsort Algorithm

First, we make $A[1..28]$ a valid max-heap by calling `BUILD-MAX-HEAP (A)`.
The Heapsort Algorithm

Here is what we get after \texttt{BUILD-MAX-HEAP (A)} finishes execution (A is now a valid max-heap):
The Heapsort Algorithm

\( A[1] = 52 \) is the largest number in the max-heap \( A[1..28] \).
The Heapsort Algorithm


The Heapsort Algorithm

But $A[1..27]$ is not a valid max-heap!
The Heapsort Algorithm

But $A[1..27]$ is not a valid max-heap!

We make $A[1..27]$ a valid max-heap by calling $\text{MAX-HEAPIFY}(A, 1)$. 

$A$

| 49 | 47 | 42 | 44 | 41 | 33 | 18 | 36 | 30 | 36 | 26 | 15 | 38 | 19 | 17 | 12 | 25 | 35 | 19 | 5 | 27 | 31 | 21 | 10 | 11 | 8 | 52 |

$A[1..27]$ is not a valid max-heap!
The Heapsort Algorithm

A[1] = 49 is the largest number in the max-heap A[1..27].
The Heapsort Algorithm


We swap $A[1]$ with $A[27]$ and treat $A[1..26]$ as the max-heap from now on.
The Heapsort Algorithm

But $A[1..26]$ is not a valid max-heap!
The Heapsort Algorithm

But $A[1..26]$ is not a valid max-heap!

We make $A[1..26]$ a valid max-heap by calling $\text{MAX-HEAPIFY}(A, 1)$. 

![Diagram of a max-heap with nodes labeled from 1 to 26]
The Heapsort Algorithm


---

Diagram of a max heap with the root at the top and the leaves at the bottom. The numbers in the heap are arranged in a binary tree structure, with each parent node having a value greater than or equal to its children. The root node, labeled 47, is at the top of the tree, and the remaining nodes are arranged in a hierarchical manner, with each level of the tree representing a layer of the heap.
The Heapsort Algorithm


The Heapsort Algorithm

But $A[1..25]$ is not a valid max-heap!
The Heapsort Algorithm

But $A[1..25]$ is not a valid max-heap!

We make $A[1..25]$ a valid max-heap by calling $\text{MAX-HEAPIFY}(A, 1)$.
The Heapsort Algorithm

The Heapsort Algorithm


The Heapsort Algorithm

But $A[1..24]$ is not a valid max-heap!
The Heapsort Algorithm

But $A[1..24]$ is not a valid max-heap!

We make $A[1..24]$ a valid max-heap by calling $\text{MAX-HEAPIFY}(A, 1)$. 
The Heapsort Algorithm

The Heapsort Algorithm


The Heapsort Algorithm

But $A[1..23]$ is not a valid max-heap!
The Heapsort Algorithm

But $A[1..23]$ is not a valid max-heap!

We make $A[1..23]$ a valid max-heap by calling $\text{MAX-HEAPIFY}(A, 1)$. 
The Heapsort Algorithm


\[
A = \begin{bmatrix}
\end{bmatrix}
\]
The Heapsort Algorithm


The Heapsort Algorithm

But $A[1..22]$ is not a valid max-heap!
The Heapsort Algorithm

But $A[1..22]$ is not a valid max-heap!

We make $A[1..22]$ a valid max-heap by calling $\text{MAX-HEAPIFY}(A, 1)$. 
The Heapsort Algorithm

Keep going!

Finally, \( A[1..28] \) will have its items sorted in nondecreasing order of value.
The Heapsort Algorithm


\begin{itemize}
  \item \textbf{HEAPSORT (A)}
  \begin{enumerate}
    \item \textbf{BUILD-MAX-HEAP (A)}
    \item \textit{for} $i = A\.\text{length} \ \textbf{downto} \ 2$
    \item exchange $A[1]$ with $A[i]$
    \item $A\.\text{heapsize} = A\.\text{heapsize} - 1$
    \item \textbf{MAX-HEAPIFY (A, 1)}
  \end{enumerate}
\end{itemize}
Priority Queues

A priority queue is a data structure for maintaining a set \( S \) of elements, each with an associated value called a key.

A max-priority queue supports the following operations:

**Insert** \((S, x)\) inserts the element \( x \) into the set \( S \), which is equivalent to the operation \( S = S \cup \{x\} \).

**Maximum** \((S)\) returns the element of \( S \) with the largest key.

**Extract-Max** \((S)\) removes and returns the element of \( S \) with the largest key.

**Increase-Key** \((S, x, k)\) increases the value of element \( x \)’s key to the new value \( k \), which is assumed to be at least as large as \( x \)’s current key value.
A Max-Heap as a Max-Priority Queue

**Heap-Maximum (A)**

1. return $A[1]$

**Heap-Extract-Max (A)**

1. if $A$.heapsize < 1
2. error “heap underflow”
3. max = $A[1]$
5. $A$.heapsize = $A$.heapsize − 1
6. Max-Heapify (A, 1)
7. return max
Extracting the Maximum (Extract-Max)

We want to extract the maximum item from the following max-heap.
Extracting the Maximum (Extract-Max)

The maximum item is at $A[1]$. So, we remove that item and return.
Extracting the Maximum (Extract-Max)

The maximum item is at $A[1]$. So, we remove that item and return.
Extracting the Maximum (Extract-Max)

But the remaining items do not form a valid max-heap.
Extracting the Maximum (Extract-Max)

To fix the data structure we move the last item at $A[28]$ to $A[1]$. 
Extracting the Maximum (Extract-Max)

To fix the data structure we move the last item at $A[28]$ to $A[1]$. 
Extracting the Maximum (Extract-Max)

To fix the data structure we move the last item at $A[28]$ to $A[1]$. 
Extracting the Maximum (Extract-Max)

To fix the data structure we move the last item at $A[28]$ to $A[1]$. 
Extracting the Maximum (Extract-Max)

But the data structure is still not a valid max-heap.

To fix it we simply call $\text{MAX-HEAPIFY}( A, 1 )$. 

```
A
36 49 42 44 47 33 41 18 36 30 41 26 15 38 19 17 12 25 35 19 5 27 31 21 10 11 8
```
### Extracting the Maximum (Extract-Max)


To fix it we simply call $\text{MAX-HEAPIFY}( A, 1 )$. 

![Max-Heap Diagram]
Extracting the Maximum (Extract-Max)


To fix it we simply call $\text{MAX-HEAPIFY}(A, 1)$.
Extracting the Maximum (Extract-Max)

This is now a valid max-heap:
A Max-Heap as a Max-Priority Queue

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

1.  \textbf{if} key < A[i]
2.  \textbf{error} “new key is smaller than current key”
3.  A[i] = key
4.  \textbf{while} i > 1 and A[PARENT(i)] < A[i]
5.  exchange A[i] with A[PARENT(i)]
6.  i = PARENT(i)
Increasing a Key

Suppose we want to increase $A[26]$ from 19 to 47.

Example 1
Increasing a Key

Suppose we want to increase $A[26]$ from 19 to 47.

So, we set $A[26] = 47$. 
Increasing a Key

Increasing a Key

But \( A[26] = 47 \) violates the max-heap property because it is larger than its parent \( A[13] = 36 \).

We fix it by swapping \( A[13] \) and \( A[26] \).
Increasing a Key


Example 1
Increasing a Key


Example 1
Increasing a Key


Example 1
Increasing a Key


Example 1
Increasing a Key


Example 1
Increasing a Key


So, we now have a valid max-heap.
Increasing a Key

Now, suppose we want to increase $A[10]$ from 25 to 55.
Increasing a Key

Now, suppose we want to increase $A[10]$ from 25 to 55.

So, we set $A[10] = 55$. 

Example 2
**Increasing a Key**


**Example 2**


\[ A = \begin{bmatrix} 
\end{bmatrix} \]
Increasing a Key


Example 2
Increasing a Key


Example 2
Increasing a Key


Increasing a Key


Example 2


Example 2
**Increasing a Key**


---

**Example 2**


Increasing a Key

Increasing a Key


So, we now have a valid max-heap.
A Max-Heap as a Max-Priority Queue

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

1. A.heapsize = A.heapsize + 1
3. **HEAP-INCREASE-KEY** (A, A.heapsize, key)
Suppose we want to insert a new key 50 into the following max-heap.
Suppose we want to insert a new key 50 into the following max-heap.

So, we create a new location $A[29]$ and set $A[29] = -\infty$. 
Suppose we want to insert a new key 50 into the following max-heap.

So, we create a new location $A[29]$ and set $A[29] = -\infty$.

Then we call $\text{HEAP-INCREASE-KEY}(A, 29, 50)$
Inserting a New Key

Suppose we want to insert a new key 50 into the following max-heap.

So, we create a new location $A[29]$ and set $A[29] = -\infty$.

Then we call `HEAP-INCREASE-KEY(A, 29, 50)`
Suppose we want to insert a new key 50 into the following max-heap.

So, we create a new location $A[29]$ and set $A[29] = -\infty$.

Then we call `HEAP-INCREASE-KEY( A, 29, 50 )`.
## A Max-Heap as a Max-Priority Queue

<table>
<thead>
<tr>
<th>Heap Operation</th>
<th>Max-Heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BUILD-HEAP</strong></td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>INSERT</strong></td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>MAXIMUM</strong></td>
<td>Θ(1)</td>
</tr>
<tr>
<td><strong>EXTRACT-MAX</strong></td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>INCREASE-KEY</strong></td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>O(log n)</td>
</tr>
</tbody>
</table>