Problem Solving
2-11 Heap & Heapsort

MA Jun

Institute of Computer Software

May 4, 2022
Contents

1. Heaps
2. Heapsort
3. Priority Queue
Contents

1 Heaps
2 Heapsort
3 Priority Queue
Heap
The (binary) heap data structure is an array object that we can view as a nearly complete binary tree.
Heaps

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree.

- The tree is completely filled on all levels except possibly the lowest.
Heaps: Max-heap VS Min-heap

Max-heap property

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

Min-heap property

\[ A[\text{Parent}(i)] \leq A[i] \]
Q 1: Why do we implement a heap with an array?
Q 1: Why do we implement a heap with an array?

- Easy to index

```
\text{PARENT}(i)
1 \quad \text{return } [i/2]

\text{LEFT}(i)
1 \quad \text{return } 2i

\text{RIGHT}(i)
1 \quad \text{return } 2i + 1
```
Q 1: Why do we implement a heap with an array?

- Easy to index
- Save memory

```
# Parent function
1  return [i/2]

# Left function
1  return 2i

# Right function
1  return 2i + 1
```
Q 1: Why do we implement a heap with an array?

- Easy to index
- Save memory
- Better cache locality

```plaintext
parent(i)
1 return [i/2]

left(i)
1 return 2i

right(i)
1 return 2i + 1
```
Heaps: Height

The height of a node

- The number of edges on the *longest simple downward path* from the node to a leaf.
Heaps: Height

The height of a node
- The number of edges on the longest simple downward path from the node to a leaf.

The height of a heap
- The height of its root, $\Theta(\lg n)$.
- A heap of $n$ elements is based on a nearly complete binary tree.
Heaps: basic operations

- The `MAX-HEAPIFY` procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The `BUILD-MAX-HEAP` procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The `HEAPSORT` procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The `MAX-HEAP-INSERT`, `HEAP-EXTRACT-MAX`, `HEAP-INCREASE-KEY`, and `HEAP-MAXIMUM` procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.
Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.
Heaps: basic operations

- The **Max-Heapify** procedure, which runs in $O(lg n)$ time, is the key to maintaining the max-heap property.
- The **Build-Max-Heap** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **Heapsort** procedure, which runs in $O(n lg n)$ time, sorts an array in place.
- The **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key**, and **Heap-Maximum** procedures, which run in $O(lg n)$ time, allow the heap data structure to implement a priority queue.
Heaps: basic operations

- The **Max-Heapify** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **Build-Max-Heap** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **Heapsort** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key**, and **Heap-Maximum** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.
Heaps: basic operations

- The **Max-Heapify** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **Build-Max-Heap** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **Heapsort** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key**, and **Heap-Maximum** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.
Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.
Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.
Q 2: Can you explain the process of **MAX-HEAPIFY**

**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. \(\text{MAX-HEAPIFY}(A, \text{largest})\)

![Diagram](image-url)
Maintaining the heap property: **MAX-HEAPIFY**

**Question 2:** Can you explain the process of **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 \( A[l] > A[i] \)
   - 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 ≠ \( i \)
   - exchange \( A[i] \) with \( A[\text{largest}] \)
7. **MAX-HEAPIFY**\( (A, \text{largest}) \)

**Pre-condition:** \( \text{Left}(i) \) and \( \text{Right}(i) \) are max-heaps.

**Post-condition:** the tree rooted at \( i \) is a maxheap.
Q 2: Can you explain the process of **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** $\text{largest} = l$
4. **else** $\text{largest} = i$
5. **if** $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} = r$
6. **if** $\text{largest} \neq i$ **then** exchange $A[i]$ with $A[\text{largest}]
7. **Max-Heapify**$(A, \text{largest})$

**Pre-condition:** $\text{Left}(i)$ and $\text{Right}(i)$ are max-heaps.

**Post-condition:** the tree rooted at $i$ is a maxheap.

Index of the largest element in the tree rooted at $i$.
Q 2: Can you explain the process of **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 $A[l] > A[i]$
   
   \[ \text{largest} = l \]
4. else $\text{largest} = i$
5. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
   
   \[ \text{largest} = r \]
6. if $\text{largest} \neq i$
   
   exchange $A[i]$ with $A[\text{largest}]$
7. \[ \text{Max-Heapify}(A, \text{largest}) \]

**Pre-condition:** Left($i$) and Right($i$) are max-heaps.

**Post-condition:** the tree rooted at $i$ is a maxheap.

Index of the largest element in the tree rooted at $i$
Maintaining the heap property: **Max-Heapify**

**Worst-case** for **Max-Heapify**

```python
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)
```

The running time of **Max-Heapify** on a subtree of size $n$ rooted at a given node $i$ is the sum of:

- $\Theta(1)$ time to find the largest,
- $\Theta(1)$ time to run **Max-Heapify** recursively,
- $\leq T(2n/3)$

Thus, $T(n) = O(lg n)$. 

MA Jun (Institute of Computer Software)

Problem Solving

May 4, 2022 8 / 30
Maintaining the heap property: **Max-Heapify**

**Worst-case** for **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)
```

The running time of **Max-Heapify** on a subtree of size $n$ rooted at a given node $i$ is the sum of:

- Time to find the largest, $\Theta(1)$
Maintaining the heap property: **Max-Heapify**

**Worst-case for Max-Heapify**

\[
\text{Max-Heapify}(A, i)
\]

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

The running time of **Max-Heapify** on a subtree of size \( n \) rooted at a given node \( i \) is the sum of:

- Time to find the largest, \( \Theta(1) \)
- Time to run **Max-Heapify** recursively, \( \leq T(2n/3) \)
Maintaining the heap property: **Max-Heapify**

**Worst-case for Max-Heapify**

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

The running time of **Max-Heapify** on a subtree of size \(n\) rooted at a given node \(i\) is the sum of:

- Time to find the largest, \(\Theta(1)\)
- Time to run **Max-Heapify** recursively, \(\leq T(2n/3)\)

\[
T(n) \leq T(2n/3) + \Theta(1) = O(\lg n) = O(h)
\]
Maintaining the heap property: **Max-Heapify**

**Worst-case** for **Max-Heapify**

Time to run **Max-Heapify** recursively, $\leq T(2n/3)$
Maintaining the heap property: \textbf{MAX-HEAPIFY}

**Worst-case** for \textbf{MAX-HEAPIFY}

Time to run \textbf{MAX-HEAPIFY} recursively, $\leq T \left( \frac{2n}{3} \right)$
Maintaining the heap property: **Max-Heapify**

**Worst-case for Max-Heapify**

Time to run Max-Heapify recursively, $\leq T(2n/3)$

The subtree rooted at $l$ is full
Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, \( \leq T(2n/3) \)

The subtree rooted at \( l \) is full
Maintaining the heap property: **Max-Heapify**

**Worst-case** for **Max-Heapify**

Time to run **Max-Heapify** recursively, \( \leq T(2n/3) \)

![Diagram of subtree]

The subtree rooted at \( l \) is full

- Total number of nodes \( n = 3X + 2 \)
Maintaining the heap property: Max-Heapify

Worst-case for Max-Heapify

Time to run Max-Heapify recursively, \( \leq T(2n/3) \)

The subtree rooted at \( l \) is full

- Total number of nodes \( n = 3X + 2 \)
- Total number of nodes in the left subtree \( 2X + 1 \)
Maintaining the heap property: **Max-Heapify**

**Worst-case for Max-Heapify**

Time to run Max-Heapify recursively, \( \leq T(2n/3) \)

- The subtree rooted at \( l \) is full

- Total number of nodes \( n = 3X + 2 \)
- Total number of nodes in the left subtree \( 2X + 1 \)
- \( \frac{2X+1}{3X+2} < \frac{2}{3} \)
Q 3: Can you explain the process of **Build-Max-Heap**

**Build-Max-Heap**\( (A) \)

1. \( A.heap-size = A.length \)
2. for \( i = \lfloor A.length / 2 \rfloor \) downto 1
3. **Max-Heapify** \( (A, i) \)
Q 3: Can you explain the process of **BUILD-MAX-HEAP**

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

1. \( A.heap-size = A.length \)
2. \textbf{for} \( i = \lceil A.length/2 \rceil \) \textbf{down to} 1
3. \textbf{MAX-HEAPIFY} \( (A, i) \)
**Q 3:** Can you explain the process of **BUILD-MAX-HEAP**

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

1. \( A \text{-heap-size} = A \text{-length} \)
2. \( \text{for } i = \lfloor A \text{-length}/2 \rfloor \text{ downto } 1 \)
3. \( \text{MAX-HEAPIFY}(A, i) \)
Building a heap: **Build-Max-Heap**

**Q 3:** Can you explain the process of **Build-Max-Heap**

```
Build-Max-Heap(A)
1   A.heap-size = A.length
2    for i = ⌊A.length/2⌋ downto 1
3       Max-Heapify(A, i)
```

**Q 4:** Can you prove the correctness of **Build-Max-Heap**?
Correctness of Build-Max-Heap

Invariant

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

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

**Proof.**

**Initialization:** Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

**Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call \textsc{Max-Heapify}$(A, i)$ to make node $i$ a max-heap root. Moreover, the \textsc{Max-Heapify} call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the \textbf{for} loop update reestablishes the loop invariant for the next iteration.

**Termination:** At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.
Correctness of **Build-Max-Heap**

**Invariant**

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

**Proof.**

**Initialization:** Prior to the first iteration of the loop, $i = \lceil n/2 \rceil$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

**Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call `MAX-HEAPIFY(A, i)` to make node $i$ a max-heap root. Moreover, the `MAX-HEAPIFY` call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the `for` loop update reestablishes the loop invariant for the next iteration.

**Termination:** At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.
Correctness of **Build-Max-Heap**

**Invariant**

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

**Proof.**

**Initialization:** Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

**Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call `MAX-HEAPIFY(A, i)` to make node $i$ a max-heap root. Moreover, the `MAX-HEAPIFY` call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the `for` loop update reestablishes the loop invariant for the next iteration.

**Termination:** At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.
Correctness of **Build-Max-Heap**

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

**Proof.**

**Initialization:** Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

**Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call **MAX-HEAPIFY**$(A, i)$ to make node $i$ a max-heap root. Moreover, the **MAX-HEAPIFY** call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the **for** loop update reestablishes the loop invariant for the next iteration.

**Termination:** At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.
Running time of **BUILD-MAX-HEAP**

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

1. \(A.\text{heap-size} = A.\text{length} \)
2. **for** \(i = \lfloor A.\text{length}/2 \rfloor \text{ **downto** } 1\)
3. **MAX-HEAPIFY**(A, i)
Running time of \textbf{Build-Max-Heap}

\textbf{Build-Max-Heap}(A)

\begin{itemize}
  \item 1 \hspace{1em} A.heap-size = A.length
  \item 2 \hspace{1em} for \hspace{0.5em} i = \lfloor A.length/2 \rfloor \hspace{0.5em} \textbf{downto} \hspace{0.5em} 1
  \item 3 \hspace{1em} Max-Heapify(A, i)
\end{itemize}

A poor upper bound

- Each call to \textbf{Max-Heapify} costs $O(\lg n)$
- At most $O(n)$ calls
- Thus, $O(n \lg n)$
Running time of **Build-Max-Heap**

**Build-Max-Heap**($A$)

1. $A.heap-size = A.length$
2. for $i = \lceil A.length/2 \rceil$ downto 1
3. Max-Heapify($A, i$)

A poor upper bound

- Each call to **Max-Heapify** costs $O(lg\ n)$
- At most $O(n)$ calls
- Thus, $O(n \cdot lg\ n)$

**Q 5:** Can you give a better one?
A tighter linear upper bound
Running time of \textbf{Build-Max-Heap}

A tighter linear upper bound

- An $n$-element heap has height $\lceil \lg n \rceil$. 
Running time of **Build-Max-Heap**

A tighter linear upper bound

- An \( n \)-element heap has height \( \lceil \log n \rceil \).
- At most \( \lceil n/2^h + 1 \rceil \) nodes of any height \( h \).
A tighter linear upper bound

- An $n$-element heap has height $\lfloor \log n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height $h$.
- Thus,

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$
A tighter linear upper bound

- An $n$-element heap has height $\lfloor \log n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height $h$.
- Thus,

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$
Running time of **Build-Max-Heap**

A tighter linear upper bound

- An $n$-element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height $h$.
- Thus,

$$
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)
$$
Running time of **BUILD-MAX-HEAP**

A tighter linear upper bound

- An $n$-element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height $h$.
- Thus,

\[
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^{h}} \right) = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^{h}} \right) = O(2n) = O(n)
\]

\[
\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^{h}} \leq \sum_{h=0}^{\infty} \frac{h}{2^{h}} = \frac{1/2}{(1-1/2)^2} = 2
\]
Inserting an element into a Heap

- **Step-1:** Add the new element to the end of the heap
- **Step-2:** Compare the new element to its parent, if it is greater than its parent, swap the two elements
- **Step-3:** Repeat step-2 until the new element is smaller than its parent or it is the root.
Inserting an element into a Heap

**Step-1:** Add the new element to the end of the heap

**Step-2:** Compare the new element to its parent, if it is greater than its parent, swap the two elements

**Step-3:** Repeat step-2 until the new element is smaller than its parent or it is the root.

\[ O(\lg n) \]
Build heap with insertion?

- Insert $A[1, .., n]$ to a heap one by one.
- Complexity?
Deleting an element from a Heap

Assume that we try to delete a node $i$
Deleting an element from a Heap

Assume that we try to delete a node $i$

- **Step-1**: Copy the value of the last node to node $i$
- **Step-2**: Remove the last node
- **Step-3**: Call **Max-Heapify** on node $i$
Deleting an element from a Heap

Assume that we try to delete a node $i$

- **Step-1:** Copy the value of the last node to node $i$
- **Step-2:** Remove the last node
- **Step-3:** Call `MAX-HEAPIFY` on node $i$
Deleting an element from a Heap

Assume that we try to delete a node $i$

- **Step-1**: Copy the value of the last node to node $i$
- **Step-2**: Remove the last node
- **Step-3**: Call MAX-HEAPIFY on node $i$

![Diagram showing the process of deleting an element from a heap.]

Delete 12

Copy 6 to the node to be deleted
Deleting an element from a Heap

Assume that we try to delete a node $i$

- **Step-1:** Copy the value of the last node to node $i$
- **Step-2:** Remove the last node
- **Step-3:** Call `Max-Heapify` on node $i$

Delete 12

Copy 6 to the node to be deleted

Delete the last node
Heapsort

**Heapsort**

**Heapsort**

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

\[
\begin{align*}
\text{a) } & \quad 16 \quad 14 \quad 10 \quad 8 \quad 7 \quad 9 \quad 3 \quad 2 \quad 4 \quad 1 \\
\text{b) } & \quad 14 \quad 8 \quad 10 \quad 4 \quad 7 \quad 9 \quad 3 \quad 2 \quad 1 \quad \text{16} \\
\text{c) } & \quad 10 \quad 8 \quad 9 \quad 4 \quad 7 \quad 1 \quad 3 \quad 2 \quad \text{14} \quad \text{16} \\
\text{d) } & \quad \text{i} \quad 9 \quad 10 \quad 14 \quad 16 \quad 8 \quad 3 \quad 7 \quad 2 \quad 4 \\
\text{e) } & \quad 10 \quad 14 \quad 16 \quad 4 \quad 2 \quad 1 \quad 7 \quad 3 \quad \text{i} \quad 9 \\
\text{f) } & \quad 10 \quad 14 \quad 16 \quad 4 \quad 8 \quad \text{i} \quad 7 \quad 3 \quad 9
\end{align*}
\]
Heapsort: Correctness

**HEAPSORT**($A$)

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

**Q 6:** How to prove the correctness of **HEAPSORT**?
Heapsort: Correctness

**HEAPSORT(A)**

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

**Q 6:** How to prove the correctness of **HEAPSORT**?

**Loop Invariant (Exercise 6.4-2)**

At the start of each iteration of the for loop of lines 2-5,

- the subarray $A[1..i]$ is a max-heap containing the $i$ smallest elements of $A[1..n]$,
- the subarray $A[i + 1..n]$ contains the $n - i$ largest elements of $A[1..n]$, sorted.
In-place sorting

In-place sorting algorithms

Algorithms require $O(1)$ extra space and sorting is said to be happened in-place, or for example, within the array itself.
In-place sorting

In-place sorting algorithms

Algorithms require $O(1)$ extra space and sorting is said to be happened in-place, or for example, within the array itself.

- **Bubble-sort ✓**
- **Insertion-sort ✓**
- **Heapsort ✓**
- **Mergesort ✓**
- **Quicksort ✗
Review Quicksort

Q 7: Why is Quicksort more efficient in practice?
Review Quicksort

Q 7: Why is Quicksort more efficient in practice?

Based on a fix computer model!

- Quicksort: $11.667(n + 1) \ln n - 1.74n - 18.74$
- Mergesort: $12.5n \ln n$
- Heapsort: $16n \ln n + 0.01n$
- Insertion sort: $2.25n^2 + 7.75n - 3 \ln n$

Donald Knuth

https://www-cs-faculty.stanford.edu/~knuth/taocp.html
Q 7: Why is Quicksort more efficient in practice?

Analyze abstract basic operations! \#swap & \#comparison

- **Quicksort**: $2n \ln n$ comparisons and $\frac{1}{3}n \ln n$ swaps on average
- **Mergesort**: $1.44n \ln n$ comparisons, but up to $8.66n \ln(n)$ array accesses (mergesort is not swap based, so we cannot count that).
- **InsertionSort**: $\frac{1}{4}n^2$ comparisons and $\frac{1}{4}n^2$ swaps on average.
Contents

1. Heaps
2. Heapsort
3. Priority Queue
Priority Queue: ADT

Priority queue
A data structure for maintaining a set $S$ of elements, each with an associated value called a key.
Priority queue
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:

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

$\text{MAXIMUM}(S)$ returns the element of $S$ with the largest key.

$\text{EXTRACT-MAX}(S)$ removes and returns the element of $S$ with the largest key.

$\text{INCREASE-KEY}(S, x, k)$ 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.
Q 8: What is key difference between a Queue and a Priority-queue?
Q 8: What is key difference between a Queue and a Priority-queue?

Queue

**FIFO**: First-In-First-Out

Priority Queue

- **Order** does not matter
- **Priority** matters
Priority Queue: Implementation

Heap $\rightarrow$ Priority Queue

**Heap-Maximum** $(A)$

1. return $A[1]$

**Heap-Maximum** $(A)$

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

**Heap-Increase-Key** $(A, i, key)$

1. if key < $A[i]$
2. error “new key is smaller than current key”
3. $A[i] = key$
4. while $i > 1$ and $A[PARENT(i)] < A[i]$
5. exchange $A[i]$ with $A[PARENT(i)]$
6. $i = PARENT(i)$

**Max-Heap-Insert** $(A, key)$

1. $A$.heap-size = $A$.heap-size + 1
3. Heap-Increase-Key $(A, A$.heap-size$, key)$
## V.S. Fibonacci Heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Fibonacci heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
</tbody>
</table>

Message Queue

In the cloud, a message queue is typically used to delegate tasks to background processing.
Message Queue (Without Priority Queue)

Application sends messages to the queue that handles messages of the designated priority. All messages in a queue have the same priority.

Message queue for priority 1 messages

1

Consumer

Consumer

Message queue for priority 2 messages

2

Consumer

Consumer

Message queue for priority 3 messages

3

Consumer

Consumer

Priority Queue: Applications

Dijkstra’s Shortest Path Algorithm

Edsger W. Dijkstra.
1972 ACM A.M. Turing
Award winner

“I designed in about twenty minutes”.
In 1956, “One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path”

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
Priority Queue: Applications

Prim Algorithm for MST

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
Thank You!
Questions?
Office 541
majun@nju.edu.cn