Problem Solving
2-11 Heap & Heapsort

MA Jun
Institute of Computer Software

May 7, 2020
Contents

1 Heaps

2 Heapsort

3 Priority Queue
Heap
Heaps

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

(a) Diagram of a heap.

(b) Array representation of the heap.
Q-1: Why do we implement a heap with an array?

- Easy to index

```
// PARENT(i)
1    return [i/2]

// LEFT(i)
1    return 2i

// RIGHT(i)
1    return 2i + 1
```
Heaps: Storage

**Q-1**: Why do we implement a heap with an array?

- Easy to index
- Save memory

**Parent(i)**
1. `return [i/2]`

**Left(i)**
1. `return 2i`

**Right(i)**
1. `return 2i + 1`
Heaps: Storage

Q-1: Why do we implement a heap with an array?

- Easy to index
- Save memory
- Better cache locality

\[ \text{Parent}(i) \]
1. `return \lfloor i/2 \rfloor`

\[ \text{Left}(i) \]
1. `return 2i`

\[ \text{Right}(i) \]
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.
Maintaining the heap property: **Max-Heapify**

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

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

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

\textbf{Pre-condition:} $\text{Left}(i)$ and $\text{Right}(i)$ are maxheaps.

\textbf{Post-condition:} the tree rooted at $i$ is a maxheap.
Maintaining the heap property: **Max-Heapify**

**Q-2:** Can you explain the process of **Max-Heapify**

```
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)
```

**Pre-condition:** Left(i) and Right(i) are maxheaps.

**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**

**Q-2:** Can you explain the process 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)
```

**Pre-condition:** Left(i) and Right(i) are maxheaps.

**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**

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

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

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

\[
\begin{align*}
\text{Time to find the largest} & : \Theta(1) \\
\text{Time to run Max-Heapify recursively} & : \leq T(2n/3) \\
\text{Total running time} & : O(lg n) = O(h)
\end{align*}
\]
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, $\Theta(1)$

---

Heaps
Maintaining the heap property
Maintaining the heap property: **Max-Heapify**

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

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

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

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. 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, \(\leq T(2n/3)\)

\[
T(n) \leq T(2n/3) + \Theta(1) = O(\log n) = O(h)
\]
Maintaining the heap property: \texttt{MAX-HEAPIFY}

\textbf{Worst-case for MAX-HEAPIFY}

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

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

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

![Diagram of a tree with nodes \( i \), \( l \), and \( r \). The root node \( i \) has two children, \( l \) and \( r \). The subtree rooted at \( l \) is full. The total number of nodes is \( n = 3 + 2 \times 2 \), the total number of nodes in the left subtree is \( 3 \times 2 + 2 \), and the total number of nodes in the right subtree is \( 2/3 \).]
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
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
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 \( n = 3X + 2 \)
Maintaining the heap property: **Max-Heapify**

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

Time to run **Max-Heapify** recursively, \( \leq T \left( \frac{2n}{3} \right) \)

The subtree rooted at \( i \) 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 \( 2X + 1 \)
- \( \frac{2X+1}{3X+2} < 2/3 \)
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)
Building a heap: **BUILD-MAX-HEAP**

**Q-3:** Can you explain the process of **BUILD-MAX-HEAP**

**BUILD-MAX-HEAP**

1. \( A.\text{heap-size} = A.\text{length} \)
2. for \( i = \lfloor A.\text{length}/2 \rfloor \) downto 1
3. **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)
```

One-element heaps:

- (a)
- (b)
- (c)
- (d)
- (e)
- (f)
Building a heap: \texttt{BUILD-MAX-HEAP}

**Q-3:** Can you explain the process of \texttt{BUILD-MAX-HEAP}

\texttt{BUILD-MAX-HEAP}(A)

1. \(A\text{.heap-size} = A\text{.length}\)
2. \textbf{for} \(i = \lfloor A\text{.length}/2 \rfloor \textbf{ downto } 1\)
3. \texttt{MAX-HEAPIFY}(A, i)

**Q-4:** Can you prove the correctness of \texttt{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 \textbf{Build-Max-Heap}

\textbf{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.

\textbf{Proof.}

\textbf{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.

\textbf{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 \texttt{MAX-HEAPIFY}(A, i) to make node $i$ a max-heap root. Moreover, the \texttt{MAX-HEAPIFY} call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the \texttt{for} loop update reestablishes the loop invariant for the next iteration.

\textbf{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 \textbf{Build-Max-Heap}

\textbf{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.

\textbf{Proof.}

\textbf{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.

\textbf{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.

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

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

1. $A.\text{heap-size} = A.\text{length}$
2. `for i = ⌈A.length/2⌉ 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, $O(n \cdot lg n)$
Running time of **Build-Max-Heap**

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

1. $\text{A.heap-size} = \text{A.length}$
2. for $i = \lfloor\text{A.length}/2\rfloor$ downto 1 
3. \textbf{Max-Heapify}($A, i$)

A poor upper bound

- Each call to \textbf{Max-Heapify} costs $O(\lg n)$
- At most $O(n)$ calls
- Thus, $O(n \lg n)$

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

A tighter linear upper bound
Running time of **Build-Max-Heap**

A tighter linear upper bound

- An $n$-element heap has height $\lfloor \lg n \rfloor$. 
Running time of BUILD-MAX-HEAP

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 $h$. 
Running time of **BUILD-MAX-HEAP**

A tighter linear upper bound

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

$$
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)
$$
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,

$$
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^{h}} \right)
$$
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.

$O(\log n)$
Build heap with insertion?

- Insert $A[1, \ldots, n]$ 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 $i$

```
13
 /   \
12   10
 /   /   \
8  7  9  6
```
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 $\text{Max-Heapify}$ on node $i$

Delete 12
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 $\text{MAX-HEAPIFY}$ on node $i$

![Diagram of a heap with nodes 13, 12, 10, 6, 8, 7, 9]
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)

![Heapsort Diagram](image-url)
**Heapsort: Correctness**

**HEAPSORT**(*A*)

1. **BUILD-MAX-HEAP**(*A*)
2. for *i* = *A*.length downto 2
3. exchange *A*[1] with *A*[i]
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: Correctness

**Heapsort**

1. **BUILD-MAX-HEAP**
2. for \( i = A.\text{length} \) downto 2
4. \( A.\text{heap-size} = A.\text{heap-size} - 1 \)
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: \(16n \ln n + 0.01n\)
- InsertionSort: \(2.25n^2 + 7.75n - 3 \ln n\)

Donald Knuth

https://www-cs-faculty.stanford.edu/~knuth/taocp.html
Review **QuickSort**

**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.

Robert Sedgewick

https://algs4.cs.princeton.edu/home/
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: ADT

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:

- **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.
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 → Priority Queue

**Heap-Maximum(A)**
1. return $A[1]$

**Heap-Extract-Max(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 $A[\text{Parent}(i)] < A[i]$
5. exchange $A[i]$ with $A[\text{Parent}(i)]$
6. $i = \text{Parent}(i)$

**Max-Heap-Insert(A, key)**
1. $A.heap-size = A.heap-size + 1$
3. Heap-Increase-Key(A, $A.heap-size$, key)
Message Queue

In the cloud, a message queue is typically used to delegate tasks to background processing.

Priority Queue: Applications

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.

Dijkstra’s Shortest Path Algorithm

“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”

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

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 819
majun@nju.edu.cn