Analysis of Algorithms

Sorting

Andres Mendez-Vazquez

September 16, 2018

Outline

Outline

Sorting Problem

Input

A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$.

Output

A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n \rangle$ such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

Sorting Problem

Input

A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$.

Output

A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n \rangle$ such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

Outline

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(k + n))$	$\Theta(d(k + n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Some Sorting Algorithms

Table of Sorting Algorithms

Algorithm	Worst-case running time	Expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$(expected)
Countingsort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(k + n))$	$\Theta(d(k + n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$(average-case)

Outline

Imagine 1964

The System/360 family was introduced by IBM

The slowest System/360, the Model 30, could perform up to 34,500 instructions per second, with memory from 8 to 64 KB.

Its main programming language was Basic Assembly Language (BAL)

You were basically EIGHT years from the first Fortran compiler (Also IBM).

Additionally, POINTERS were invented this year barely

Therefore... back to first principles my dear Clarice....

Imagine 1964

The System/360 family was introduced by IBM

The slowest System/360, the Model 30, could perform up to 34,500 instructions per second, with memory from 8 to 64 KB.

Its main programming language was Basic Assembly Language (BAL)

You were basically EIGHT years from the first Fortran compiler (Also IBM).

Additionally, POINTERS were invented this year barely

Therefore... back to first principles my dear Clarice....

Imagine 1964

The System/360 family was introduced by IBM

The slowest System/360, the Model 30, could perform up to 34,500 instructions per second, with memory from 8 to 64 KB.

Its main programming language was Basic Assembly Language (BAL)

You were basically EIGHT years from the first Fortran compiler (Also IBM).

Additionally, POINTERS were invented this year barely

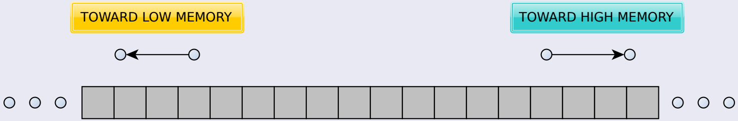
Therefore... back to first principles my dear Clarice....

Yepi

Yes... my dear Clarice...

Then, if you put all together

We have a memory structure like

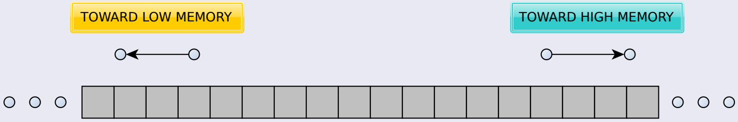


Let us to think about it

How? We can assume a series of constraints....

Then, if you put all together

We have a memory structure like



Let us to think about it

How? We can assume a series of constraints....

Constraints

We want a system that allows for priorities such that

- We do not want to scan the entire memory for that.
- We want to avoid doing to a lot of shifting in the main memory.

Therefore

We want that allows the following ADT operations:

- Insertion
- Deletion
- Search

In a Time LESS than

$$O(n)$$

Constraints

We want a system that allows for priorities such that

- We do not want to scan the entire memory for that.
- We want to avoid doing to a lot of shifting in the main memory.

Therefore

We want that allows the following ADT operations:

- Insertion
- Deletion
- Search

In a Time LESS than

$$O(n)$$

Constraints

We want a system that allows for priorities such that

- We do not want to scan the entire memory for that.
- We want to avoid doing to a lot of shifting in the main memory.

Therefore

We want that allows the following ADT operations:

- Insertion
- Deletion
- Search

In a Time LESS than

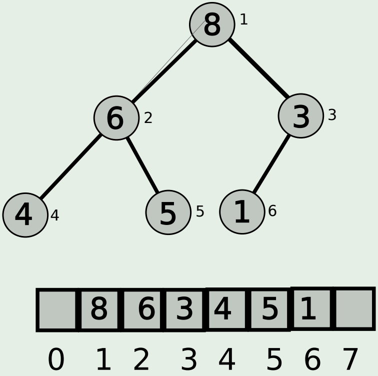
$$O(n)$$

Outline

Definitions

Definition

A heap is an array object that can be viewed as a nearly complete binary tree.



Basic Attributes

Given an array A, we have that $length[A]$

It is the size of the storing array.

$heap\text{-}size[A]$

Tell us how many elements in the heap are stored in the array.

Thus, we have

$$0 \leq heap\text{-}size[A] \leq length[A] \quad (1)$$

Basic Attributes

Given an array A, we have that $length[A]$

It is the size of the storing array.

$heap\text{-}size[A]$

Tell us how many elements in the heap are stored in the array.

Thus, we have

$$0 \leq heap\text{-}size[A] \leq length[A] \quad (1)$$

Basic Attributes

Given an array A, we have that $length[A]$

It is the size of the storing array.

$heap\text{-}size[A]$

Tell us how many elements in the heap are stored in the array.

Thus, we have

$$0 \leq heap\text{-}size[A] \leq length[A] \tag{1}$$

Outline

Heap Sort: Calculations given a Node i in the heap

$Parent(i)$ - Parent Node

$$Parent(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

Left Node Child: $Left(i)$

$$Left(i) = 2i$$

Right Node Child: $Right(i)$

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

Heap Sort: Calculations given a Node i in the heap

$Parent(i)$ - Parent Node

$$Parent(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

Left Node Child: $Left(i)$

$$Left(i) = 2i$$

Right Node Child: $Right(i)$

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

Heap Sort: Calculations given a Node i in the heap

$Parent(i)$ - Parent Node

$$Parent(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

Left Node Child: $Left(i)$

$$Left(i) = 2i$$

Right Node Child: $Right(i)$

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

Max/Min Heap Properties

Given that

$A\left[i\right]$ returns the value of the key, we have that

Max Heap property

$A\left[Parent(i)\right] \geq A[i]$

Min Heap property

$A\left[Parent(i)\right] \leq A\left[i\right]$

Max/Min Heap Properties

Given that

$A[i]$ returns the value of the key, we have that

Max Heap property

$A[Parent(i)] \geq A[i]$

Min Heap property

$A[Parent(i)] \leq A[i]$

Max/Min Heap Properties

Given that

$A\left[i\right]$ returns the value of the key, we have that

Max Heap property

$A\left[Parent(i)\right] \geq A[i]$

Min Heap property

$A\left[Parent(i)\right] \leq A\left[i\right]$

Outline

What we want!!!

A function to keep the property of max or min heap

After all, remembering Kolmogorov, we are acting in a part of the array trying to keep certain properties

- Which ONE?

Remember

What we want!!!

A function to keep the property of max or min heap

After all, remembering Kolmogorov, we are acting in a part of the array trying to keep certain properties

- Which ONE?

Remember

Single nodes are always min heaps or max heaps

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

$l = Left(i)$
$r = Right$(i)
If $l \leq heap - size\left[A\right]$ and $A\left[l\right] > A\left[i\right]$
    $largest = l$
else $largest = i$
If $r \leq heap - size\left[A\right]$ and $A\left[r\right] > A\left[largest\right]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    Max-Heapify$(A, largest)$

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

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

Figure: A trickle down algorithm

Heap Sort: Max-Heapify

Algorithm (preserving the heap property) when somebody violates the max/min property

Max-Heapify(A, i)

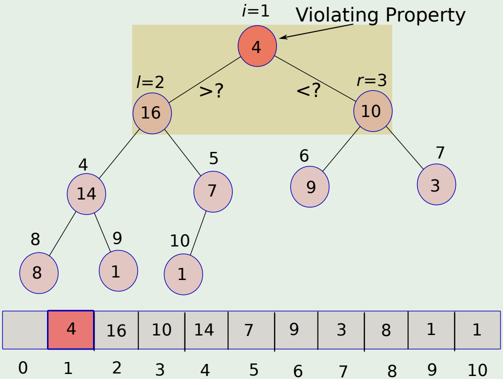
1. $l = Left(i)$
2. $r = Right$(i)
3. If $l \leq heap - size\left[A\right]$ and $A\left[l\right] > A\left[i\right]$
4. $\quad largest = l$
5. else $largest = i$
6. If $r \leq heap - size\left[A\right]$ and $A\left[r\right] > A\left[largest\right]$
7. $\quad largest = r$
8. if $largest \neq i$
9. $\quad$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

Figure: A trickle down algorithm

Example keeping the heap property starting at $i = 1$

Here, you could imagine that somebody inserted a node at $i = 1$

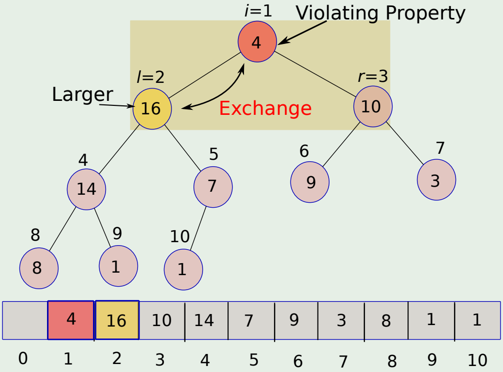
3. If $l \leq heap - size\,[A]$ and $A\,[l] > A\,[i]$
4. $\quad largest = l$
5. else $largest = i$
6. If $r \leq heap - size\,[A]$ and $A\,[r] > A\,[largest]$
7. $\quad largest = r$



Example keeping the heap property starting at $i = 1$

One of the children is chosen to be exchanged

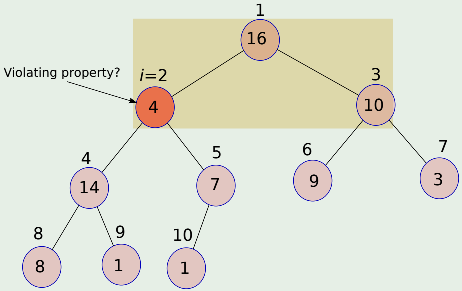
8. if $largest \neq i$
9. exchange $A[i]$ with $A[largest]$



Example: Now $i = largest$

Make the excahnge and call the **Max-Heapify**

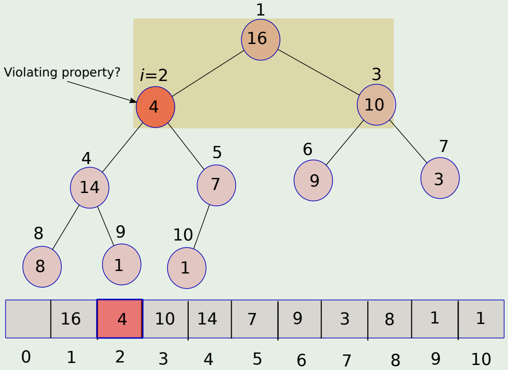
10. **Max-Heapify**$(A, largest)$



	16	4	10	14	7	9	3	8	1	1
0	1	2	3	4	5	6	7	8	9	10

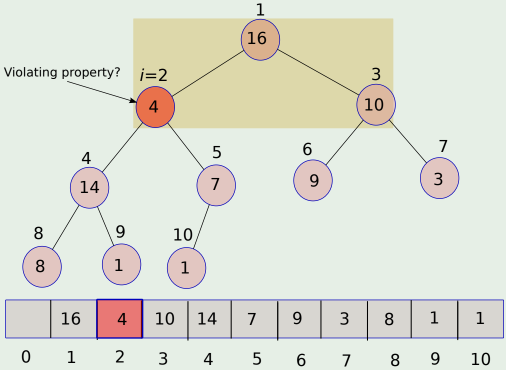
Example: Now $i = largest$

Keep going



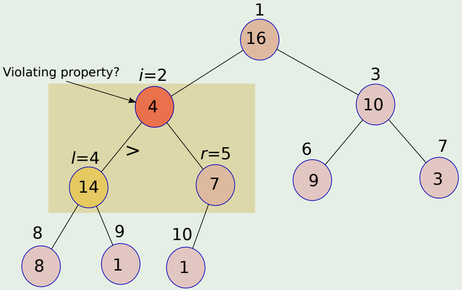
Example: Now $i = largest$

Keep going



Example: Now $i = largest$

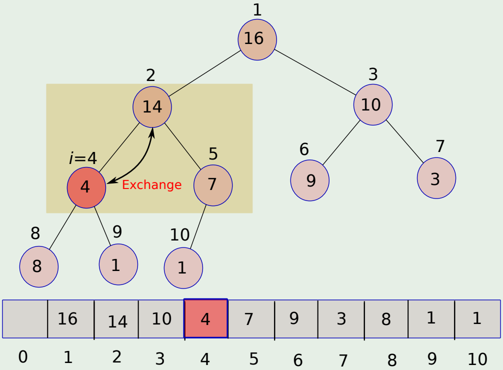
Keep going



	16	4	10	14	7	9	3	8	1	1
0	1	2	3	4	5	6	7	8	9	10

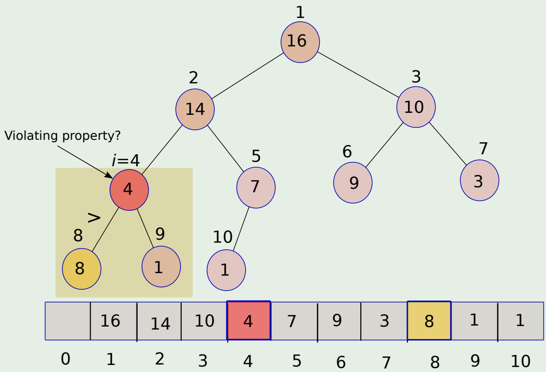
Example: Now $i = largest$

Keep going



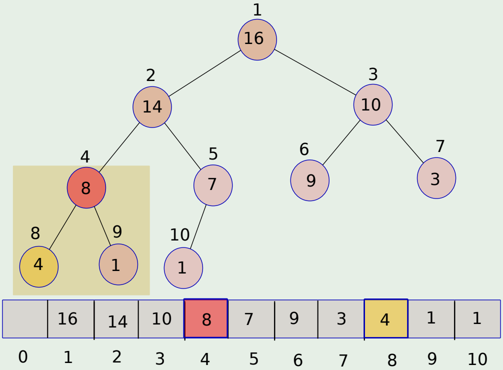
Example: Now $i = largest$

Keep going



Example: Now $i = largest$

Keep going



Complexity of Max-Heapify

For this

It is possible to prove that the upper bound on the size of each children's subtrees is $\frac{2n}{3}$ starting at the root (First Recursive Call!!!).

Thus

In addition, we use the idea of height from the root node $(h = 0)$ to leaves $(h = \log n - 1)$.

Complexity of Max-Heapify

For this

It is possible to prove that the upper bound on the size of each children's subtrees is $\frac{2n}{3}$ starting at the root (First Recursive Call!!!).

Thus

In addition, we use the idea of height from the root node $(h = 0)$ to leaves $(h = \log n - 1)$.

Then

We have that by using the nearly complete structure

1. First for $n = 1$, we have that the size of children's subtrees is $0 < \frac{2}{3}$.
2. For $n = 2$, we have that the size of children's subtrees is at most $1 < \frac{4}{3}$.
3. For $n = 3$, we have that the size of children's subtrees is at most $1 < \frac{6}{3} = 2$.
4. For $n = 4$, we have that the size of children's subtrees is at most $2 < \frac{8}{3}$.
5. etc...

Then

We have that by using the nearly complete structure

1. First for $n = 1$, we have that the size of children's subtrees is $0 < \frac{2}{3}$.
2. For $n = 2$, we have that the size of children's subtrees is at most $1 < \frac{4}{3}$.
3. For $n = 3$, we have that the size of children's subtrees is at most $1 < \frac{6}{3} = 2$.
4. For $n = 4$, we have that the size of children's subtrees is at most $2 < \frac{8}{3}$.
5. etc...

Then

We have that by using the nearly complete structure

1. First for $n = 1$, we have that the size of children's subtrees is $0 < \frac{2}{3}$.
2. For $n = 2$, we have that the size of children's subtrees is at most $1 < \frac{4}{3}$.
3. For $n = 3$, we have that the size of children's subtrees is at most $1 < \frac{6}{3} = 2$.
4. For $n = 4$, we have that the size of children's subtrees is at most $2 < \frac{8}{3}$.
5. etc...

Then

We have that by using the nearly complete structure

1. First for $n = 1$, we have that the size of children's subtrees is $0 < \frac{2}{3}$.
2. For $n = 2$, we have that the size of children's subtrees is at most $1 < \frac{4}{3}$.
3. For $n = 3$, we have that the size of children's subtrees is at most $1 < \frac{6}{3} = 2$.
4. For $n = 4$, we have that the size of children's subtrees is at most $2 < \frac{8}{3}$.
5. etc...

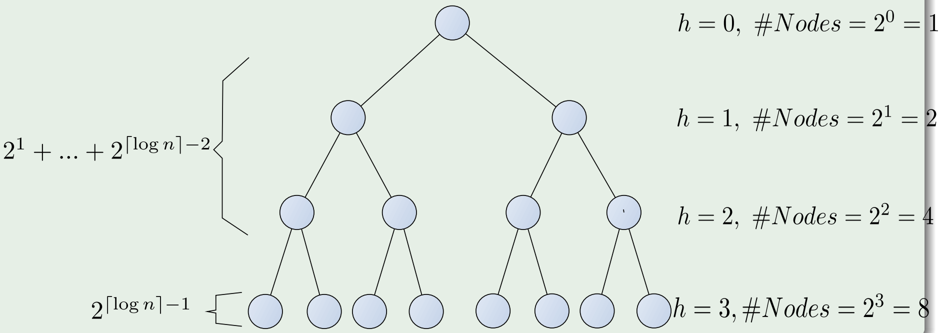
Then

We have that by using the nearly complete structure

1. First for $n = 1$, we have that the size of children's subtrees is $0 < \frac{2}{3}$.
2. For $n = 2$, we have that the size of children's subtrees is at most $1 < \frac{4}{3}$.
3. For $n = 3$, we have that the size of children's subtrees is at most $1 < \frac{6}{3} = 2$.
4. For $n = 4$, we have that the size of children's subtrees is at most $2 < \frac{8}{3}$.
5. etc...

Do you notice the following?

Imagine the following case

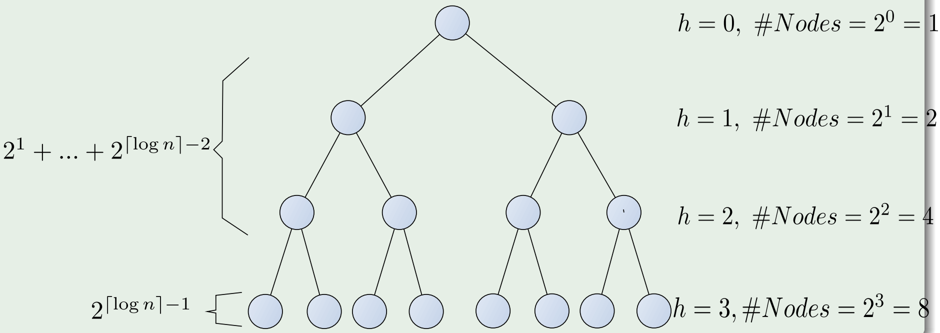


The maximum number of nodes in both children assuming a full tree with n nodes

$$2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2} + 2^{\lceil \log n \rceil - 1} \quad (2)$$

Do you notice the following?

Imagine the following case

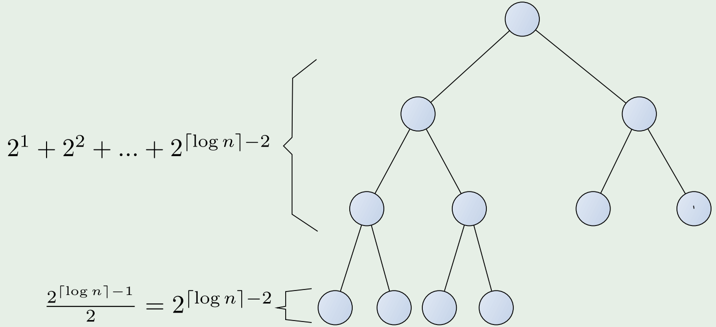


The maximum number of nodes in both children assuming a full tree with n nodes

$$2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2} + 2^{\lceil \log n \rceil - 1} \tag{2}$$

Now

Imagine the following special case



The maximum number of nodes in one child is equal to

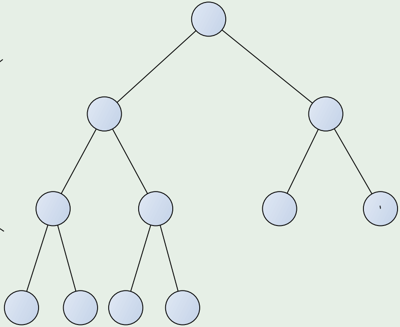
$$\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + \frac{2^{\lceil \log n \rceil - 1}}{2} \quad (3)$$

Now

Imagine the following special case

$$2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}$$

$$\frac{2^{\lceil \log n \rceil - 1}}{2} = 2^{\lceil \log n \rceil - 2}$$



The maximum number of nodes in one child is equal to

$$\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + \frac{2^{\lceil \log n \rceil - 1}}{2} \tag{3}$$

The total number of elements in a child's subtree

The total number of nodes in a CHILD is bounded

$$
\begin{aligned}
\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &= 1 + 2^2 + ... + 2^{\lceil \log n \rceil - 3} + 2^{\lceil \log n \rceil - 2}\\
&= \frac{1 - 2^{\lceil \log n \rceil - 2}}{1 - 2} + 2^{\lceil \log n \rceil - 2}\\
&= 2^{\lceil \log n \rceil - 2} - 1 + 2^{\lceil \log n \rceil - 2}\\
&= 2 \times 2^{\lceil \log n \rceil - 2} - 1\\
&= 2^{\lceil \log n \rceil - 1} - \frac{2}{3} - \frac{1}{3}\\
&< 2^{\lceil \log n \rceil - 1} - \frac{2}{3}
\end{aligned}
$$

The total number of elements in a child's subtree

The total number of nodes in a CHILD is bounded

$$
\begin{aligned}
\frac{2^1 + 2^2 + \ldots + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &= 1 + 2^2 + \ldots + 2^{\lceil \log n \rceil - 3} + 2^{\lceil \log n \rceil - 2} \\
&= \frac{1 - 2^{\lceil \log n \rceil - 2}}{1 - 2} + 2^{\lceil \log n \rceil - 2} \\
&= 2^{\lceil \log n \rceil - 2} - 1 + 2^{\lceil \log n \rceil - 2} \\
&= 2 \times 2^{\lceil \log n \rceil - 2} - 1 \\
&= 2^{\lceil \log n \rceil - 1} - \frac{2}{3} - \frac{1}{3} \\
&< 2^{\lceil \log n \rceil - 1} - \frac{2}{3}
\end{aligned}
$$

The total number of elements in a child's subtree

The total number of nodes in a CHILD is bounded

$$
\begin{aligned}
\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &= 1 + 2^2 + ... + 2^{\lceil \log n \rceil - 3} + 2^{\lceil \log n \rceil - 2} \\
&= \frac{1 - 2^{\lceil \log n \rceil - 2}}{1 - 2} + 2^{\lceil \log n \rceil - 2} \\
&= 2^{\lceil \log n \rceil - 2} - 1 + 2^{\lceil \log n \rceil - 2} \\
&= 2 \times 2^{\lceil \log n \rceil - 2} - 1 \\
&= 2^{\lceil \log n \rceil - 1} - \frac{2}{3} - \frac{1}{3} \\
&< 2^{\lceil \log n \rceil - 1} - \frac{2}{3}
\end{aligned}
$$

The total number of elements in a child's subtree

The total number of nodes in a CHILD is bounded

$$
\begin{aligned}
\frac{2^1+2^2+...+2^{\lceil \log n \rceil -2}}{2}+2^{\lceil \log n \rceil -2} &= 1+2^2+...+2^{\lceil \log n \rceil -3}+2^{\lceil \log n \rceil -2}\\
&= \frac{1-2^{\lceil \log n \rceil -2}}{1-2}+2^{\lceil \log n \rceil -2}\\
&= 2^{\lceil \log n \rceil -2}-1+2^{\lceil \log n \rceil -2}\\
&= 2\times 2^{\lfloor \log n \rfloor -2}-1\\
&= 2^{\lceil \log n \rceil -1}-\frac{2}{3}-\frac{1}{3}\\
&< 2^{\lceil \log n \rceil -1}-\frac{2}{3}
\end{aligned}
$$

The total number of elements in a child's subtree

The total number of nodes in a CHILD is bounded

$$
\begin{aligned}
\frac{2^1 + 2^2 + \ldots + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &= 1 + 2^2 + \ldots + 2^{\lceil \log n \rceil - 3} + 2^{\lceil \log n \rceil - 2} \\
&= \frac{1 - 2^{\lceil \log n \rceil - 2}}{1 - 2} + 2^{\lceil \log n \rceil - 2} \\
&= 2^{\lceil \log n \rceil - 2} - 1 + 2^{\lceil \log n \rceil - 2} \\
&= 2 \times 2^{\lfloor \log n \rfloor - 2} - 1 \\
&= 2^{\lceil \log n \rceil - 1} - \frac{2}{3} - \frac{1}{3} \\
&< 2^{\lceil \log n \rceil - 1} - \frac{2}{3}
\end{aligned}
$$

The total number of elements in a child's subtree

The total number of nodes in a CHILD is bounded

$$
\begin{aligned}
\frac{2^1+2^2+...+2^{\lceil \log n \rceil -2}}{2} + 2^{\lceil \log n \rceil -2} &= 1 + 2^2 + ... + 2^{\lceil \log n \rceil -3} + 2^{\lceil \log n \rceil -2} \\
&= \frac{1-2^{\lceil \log n \rceil -2}}{1-2} + 2^{\lceil \log n \rceil -2} \\
&= 2^{\lceil \log n \rceil -2} - 1 + 2^{\lceil \log n \rceil -2} \\
&= 2 \times 2^{\lfloor \log n \rfloor -2} - 1 \\
&= 2^{\lceil \log n \rceil -1} - \frac{2}{3} - \frac{1}{3} \\
&< 2^{\lceil \log n \rceil -1} - \frac{2}{3}
\end{aligned}
$$

The total number of elements in a child's subtree

Notice the following

$$2^{\lceil \log n \rceil} < \frac{4}{3}\left[2^{\log n}\right] \quad (4)$$

- When $n = 2^p - 1$
- For the case that that $n \leq 2^p - 1$ we can use the fact that $\lceil \log n \rceil = p$ for some power of 2.

Induction to prove the previous statement

Step $n = 1$

$$2^{\lceil \log 1 \rceil} = 2^0 = 1 < \frac{4}{3} \times 2^{\log 1} \tag{5}$$

Assume is true for n

$$2^{\lceil \log n \rceil} < \frac{4}{3}\left[2^{\log n}\right] \tag{6}$$

Induction to prove the previous statement

Step $n = 1$

$$2^{\lceil \log 1 \rceil} = 2^0 = 1 < \frac{4}{3} \times 2^{\log 1} \tag{5}$$

Assume is true for n

$$2^{\lceil \log n \rceil} < \frac{4}{3} \left[2^{\log n} \right] \tag{6}$$

Induction to prove the previous statement

Now prove for $n+1$

$$
\begin{aligned}
2^{\lceil \log(n+1) \rceil} &= 2^{\lceil \log(2^p - 1 + 1) \rceil} \\
&= 2^{\lceil p \rceil} \\
&= 2^{p} \\
&= 2^{\log 2^p} \\
&< \frac{4}{3}\left[2^{\log 2^p}\right] \\
&= \frac{4}{3}\left[2^{\log(n+1)}\right]
\end{aligned}
$$

Induction to prove the previous statement

Now prove for $n+1$

$$
\begin{aligned}
2^{\lceil \log(n+1) \rceil} &= 2^{\lceil \log(2^p - 1 + 1) \rceil} \\
&= 2^{\lceil p \rceil} \\
&= 2^p \\
&= 2^{\log 2^p} \\
&< \frac{4}{3}\left[2^{\log 2^p}\right] \\
&= \frac{4}{3}\left[2^{\log(n+1)}\right]
\end{aligned}
$$

cinvestav

Induction to prove the previous statement

Now prove for $n+1$

$$
\begin{aligned}
2^{\lceil \log(n+1) \rceil} &= 2^{\lceil \log(2^p - 1 + 1) \rceil} \\
&= 2^{\lceil p \rceil} \\
&= 2^p \\
&= 2^{\log 2^p} \\
&< \frac{4}{3}\left[2^{\log 2^p}\right] \\
&= \frac{4}{3}\left[2^{\log(n+1)}\right]
\end{aligned}
$$

Induction to prove the previous statement

Now prove for $n+1$

$$
\begin{aligned}
2^{\lceil \log(n+1) \rceil} &= 2^{\lceil \log(2^p - 1 + 1) \rceil} \\
&= 2^{\lceil p \rceil} \\
&= 2^{p} \\
&= 2^{\log 2^p} \\
&< \frac{4}{3}\left[2^{\log 2^p}\right] \\
&= \frac{4}{3}\left[2^{\log (n+1)}\right]
\end{aligned}
$$

Induction to prove the previous statement

Now prove for $n+1$

$$
\begin{aligned}
2^{\lceil \log(n+1) \rceil} &= 2^{\lceil \log(2^p - 1 + 1) \rceil} \\
&= 2^{\lceil p \rceil} \\
&= 2^{p} \\
&= 2^{\log 2^p} \\
&< \frac{4}{3}\left[2^{\log 2^p}\right] \\
&= \frac{4}{3}\left[2^{\log(n+1)}\right]
\end{aligned}
$$

Induction to prove the previous statement

Now prove for $n+1$

$$
\begin{aligned}
2^{\lceil \log(n+1) \rceil} &= 2^{\lceil \log(2^p - 1 + 1) \rceil} \\
&= 2^{\lceil p \rceil} \\
&= 2^{p} \\
&= 2^{\log 2^p} \\
&< \frac{4}{3}\left[2^{\log 2^p}\right] \\
&= \frac{4}{3}\left[2^{\log (n+1)}\right]
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &< 2^{\lceil \log n \rceil - 1} - \frac{2}{3} \\
&= \frac{2^{\lceil \log n \rceil}}{2} - \frac{2}{3} \\
&< \frac{4}{3}\left[2^{\log n - 1}\right] - \frac{2}{3} \\
&= \frac{2}{3}\left[2 \times 2^{\log n - 1} - 1\right] \\
&= \frac{2}{3}\left[2^{\log n} - 1\right] \\
&= \frac{2}{3}\left[n - 1\right] \\
&< \frac{2n}{3}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
\frac{2^1 + 2^2 + \ldots + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &< 2^{\lceil \log n \rceil - 1} - \frac{2}{3} \\
&= \frac{2^{\lceil \log n \rceil}}{2} - \frac{2}{3} \\
&< \frac{4}{3}\left[2^{\log n - 1}\right] - \frac{2}{3} \\
&= \frac{2}{3}\left[2 \times 2^{\log n - 1} - 1\right] \\
&= \frac{2}{3}\left[2^{\log n} - 1\right] \\
&= \frac{2}{3}\left[n - 1\right] \\
&< \frac{2n}{3}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
\frac{2^1+2^2+\ldots+2^{\lceil \log n\rceil-2}}{2}+2^{\lceil \log n\rceil-2} &< 2^{\lceil \log n\rceil-1}-\frac{2}{3}\\
&= \frac{2^{\lceil \log n\rceil}}{2}-\frac{2}{3}\\
&< \frac{4}{3}\left[2^{\log n-1}\right]-\frac{2}{3}\\
&= \frac{2}{3}\left[2\times 2^{\log n-1}-1\right]\\
&= \frac{2}{3}\left[2^{\log n}-1\right]\\
&= \frac{2}{3}\left[n-1\right]\\
&< \frac{2n}{3}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &< 2^{\lceil \log n \rceil - 1} - \frac{2}{3} \\
&= \frac{2^{\lceil \log n \rceil}}{2} - \frac{2}{3} \\
&< \frac{4}{3}\left[2^{\log n - 1}\right] - \frac{2}{3} \\
&= \frac{2}{3}\left[2 \times 2^{\log n - 1} - 1\right] \\
&= \frac{2}{3}\left[2^{\log n} - 1\right] \\
&= \frac{2}{3}\left[n - 1\right] \\
&< \frac{2n}{3}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
\frac{2^1 + 2^2 + \ldots + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &< 2^{\lceil \log n \rceil - 1} - \frac{2}{3} \\
&= \frac{2^{\lceil \log n \rceil}}{2} - \frac{2}{3} \\
&< \frac{4}{3}\left[2^{\log n - 1}\right] - \frac{2}{3} \\
&= \frac{2}{3}\left[2 \times 2^{\log n - 1} - 1\right] \\
&= \frac{2}{3}\left[2^{\log n} - 1\right] \\
&= \frac{2}{3}\left[n - 1\right] \\
&< \frac{2n}{3}
\end{aligned}
$$

Therefore

We have that

$$
\begin{aligned}
\frac{2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2}}{2} + 2^{\lceil \log n \rceil - 2} &< 2^{\lceil \log n \rceil - 1} - \frac{2}{3} \\
&= \frac{2^{\lceil \log n \rceil}}{2} - \frac{2}{3} \\
&< \frac{4}{3}\left[2^{\log n - 1}\right] - \frac{2}{3} \\
&= \frac{2}{3}\left[2 \times 2^{\log n - 1} - 1\right] \\
&= \frac{2}{3}\left[2^{\log n} - 1\right] \\
&= \frac{2}{3}\left[n - 1\right] \\
&< \frac{2n}{3}
\end{aligned}
$$

Therefore

We have that

$$\begin{aligned}
\frac{2^1+2^2+\dots+2^{\lceil \log n\rceil-2}}{2}+2^{\lceil \log n\rceil-2} &< 2^{\lceil \log n\rceil-1}-\frac{2}{3}\\
&=\frac{2^{\lceil \log n\rceil}}{2}-\frac{2}{3}\\
&<\frac{4}{3}\left[2^{\log n-1}\right]-\frac{2}{3}\\
&=\frac{2}{3}\left[2\times 2^{\log n-1}-1\right]\\
&=\frac{2}{3}\left[2^{\log n}-1\right]\\
&=\frac{2}{3}\left[n-1\right]\\
&<\frac{2n}{3}
\end{aligned}$$

Outline

Complexity of Max-Heapify

Knowing that the number of nodes in any child is bounded by

$$\frac{2n}{3} \tag{7}$$

Complexity of Max-Heapify

Knowing that the number of nodes in any child is bounded by

$$\frac{2n}{3} \tag{7}$$

Thus, given that $T(n)$ represent the complexity of the Max-Heapify

$$T(n) = T(\text{How many nodes will be touched by the recusrsion}) + \Theta(1) \tag{8}$$

Complexity of Max-Heapify

Knowing that the number of nodes in any child is bounded by

$$\frac{2n}{3} \tag{7}$$

Thus, given that $T(n)$ represent the complexity of the Max-Heapify

$$T(n) = T(\text{How many nodes will be touched by the recusrsion}) + \Theta(1) \tag{8}$$

Here

- $\Theta(1)$ is the constant part of the algorithm before recursion.
- $T(\text{How many nodes will be touched by the recusrsion}) = T\left(\sum_{i=1}^{\frac{\log_2 n}{2}-1} 3\right)$.
- How?

Complexity of Max-Heapify

Knowing that the number of nodes in any child is bounded by

$$\frac{2n}{3} \tag{7}$$

Thus, given that $T(n)$ represent the complexity of the Max-Heapify

$$T(n) = T(\text{How many nodes will be touched by the recusrsion}) + \Theta(1) \tag{8}$$

Here

- $\Theta(1)$ is the constant part of the algorithm before recursion.
- $T(\text{How many nodes will be touched by the recusrsion}) = T\left(\sum_{i=1}^{\frac{\log_2 n}{2}-1} 3\right)$.
- How?

Complexity of Max-Heapify

Knowing that the number of nodes in any child is bounded by

$$\frac{2n}{3} \tag{7}$$

Thus, given that $T(n)$ represent the complexity of the Max-Heapify

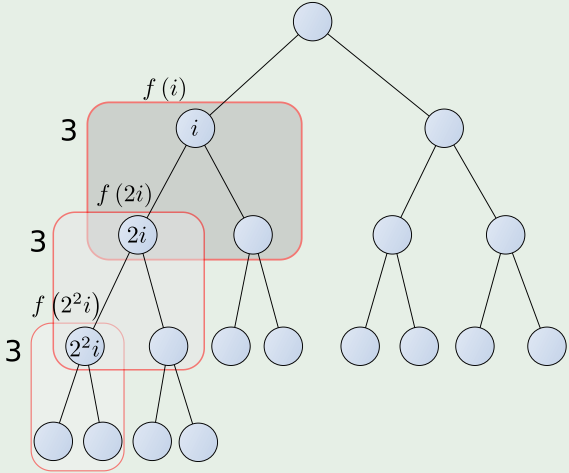
$$T(n) = T\left(\text{How many nodes will be touched by the recusrsion}\right) + \Theta(1) \tag{8}$$

Here

- $\Theta(1)$ is the constant part of the algorithm before recursion.
- $T\left(\text{How many nodes will be touched by the recusrsion}\right) = T\left(\sum_{i=1}^{\frac{\log_2 n}{2}-1} 3\right)$.
- How?

Complexity of Max-Heapify

The Recursion Idea



Complexity of Max-Heapify

Thus

$$
\begin{aligned}
\sum_{i=1}^{\frac{\log_2 n-1}{2}-1} 3 &= \frac{3^{\frac{\log_2 n}{2}}-1}{3-1} - 3 \\
&= \frac{\left(3^{\frac{1}{2}}\right)^{\log_2 n}}{2} - 3 \\
&= \frac{n^{\log_2\left(3^{\frac{1}{2}}\right)}}{2} - 3 \\
&\leq \frac{n^{0.8}}{2} \\
&\leq \frac{2n^{0.8}}{3} \\
&\leq \frac{2n}{3}
\end{aligned}
$$

Complexity of Max-Heapify

Thus

$$
\begin{aligned}
\sum_{i=1}^{\frac{\log_2 n-1}{2}-1} 3 &= \frac{3^{\frac{\log_2 n}{2}}-1}{3-1}-3\\
&= \frac{\left(3^{\frac{1}{2}}\right)^{\log_2 n}}{2}-3\\
&= \frac{n^{\log_2\left(3^{\frac{1}{2}}\right)}}{2}-3\\
&\leq \frac{n^{0.8}}{2}\\
&\leq \frac{2n^{0.8}}{3}\\
&\leq \frac{2n}{3}
\end{aligned}
$$

Complexity of Max-Heapify

Thus

$$
\begin{aligned}
\sum_{i=1}^{\frac{\log_2 n - 1}{2} - 1} 3 &= \frac{3^{\frac{\log_2 n}{2}} - 1}{3 - 1} - 3 \\
&= \frac{\left(3^{\frac{1}{2}}\right)^{\log_2 n}}{2} - 3 \\
&= \frac{n^{\log_2\left(3^{\frac{1}{2}}\right)}}{2} - 3 \\
&\leq \frac{n^{0.8}}{2} \\
&\leq \frac{2n^{0.8}}{3} \\
&\leq \frac{2n}{3}
\end{aligned}
$$

Complexity of Max-Heapify

Thus

$$
\begin{aligned}
\sum_{i=1}^{\frac{\log_2 n-1}{2}-1} 3 &= \frac{3^{\frac{\log_2 n}{2}}-1}{3-1}-3 \\
&= \frac{\left(3^{\frac{1}{2}}\right)^{\log_2 n}}{2}-3 \\
&= \frac{n^{\log_2\left(3^{\frac{1}{2}}\right)}}{2}-3 \\
&\leq \frac{n^{0.8}}{2} \\
&\leq \frac{2n^{0.8}}{3} \\
&\leq \frac{2n}{3}
\end{aligned}
$$

Complexity of Max-Heapify

Thus

$$
\begin{aligned}
\sum_{i=1}^{\frac{\log_2 n-1}{2}-1} 3 &= \frac{3^{\frac{\log_2 n}{2}}-1}{3-1}-3\\
&= \frac{\left(3^{\frac{1}{2}}\right)^{\log_2 n}}{2}-3\\
&= \frac{n^{\log_2\left(3^{\frac{1}{2}}\right)}}{2}-3\\
&\leq \frac{n^{0.8}}{2}\\
&\leq \frac{2n^{0.8}}{3}\\
&\leq \frac{2n}{3}
\end{aligned}
$$

Complexity of Max-Heapify

Thus

$$
\begin{aligned}
\sum_{i=1}^{\frac{\log_2 n-1}{2}-1} 3 &= \frac{3^{\frac{\log_2 n}{2}}-1}{3-1}-3\\
&= \frac{\left(3^{\frac{1}{2}}\right)^{\log_2 n}}{2}-3\\
&= \frac{n^{\log_2\left(3^{\frac{1}{2}}\right)}}{2}-3\\
&\leq \frac{n^{0.8}}{2}\\
&\leq \frac{2n^{0.8}}{3}\\
&\leq \frac{2n}{3}
\end{aligned}
$$

Complexity of Max-Heapify

Thus, if we assume that T is an increasing monotone function

$$T(n) = T\left(\sum_{i=1}^{\frac{\log_2 n}{2}-1} 3\right) + \Theta(1)$$

$$\leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

Complexity of Max-Heapify

Thus, if we assume that T is an increasing monotone function

$$T(n) = T\left(\sum_{i=1}^{\frac{\log_2 n}{2}-1} 3\right) + \Theta(1)$$
$$\leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

Algorithm Complexity

This is by the master the master theorem $O(\log_2 n)$.

Complexity of Max-Heapify

Thus, if we assume that T is an increasing monotone function

$$
\begin{aligned}
T(n) &= T\left(\sum_{i=1}^{\frac{\log_2 n}{2}-1} 3\right) + \Theta(1) \\
&\leq T\left(\frac{2n}{3}\right) + \Theta(1)
\end{aligned}
$$

Algorithm Complexity

This is by the master the master theorem $O(\log_2 n)$.

Outline

Heap Sort: Using Max-Heapify

Algorithm Build-Max-Heap

Build-Max-Heap(A)

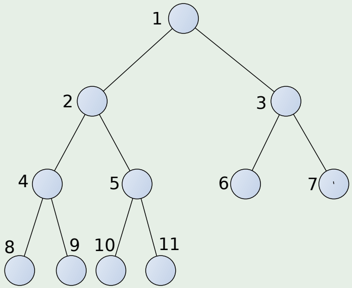
1. $heap - size[A] = length[A]$
2. for $i = \lfloor length[A]/2 \rfloor$ downto 1
3. **Max-Heapify**(A, i)

Figure: Building a Heap

Question?

Why from $\left\lfloor {}^{length[A]}/_{2} \right\rfloor$?

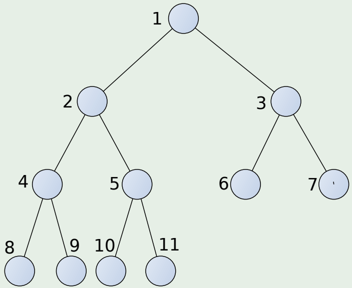
Look at this



Question?

Why from $\lfloor length[A]/2 \rfloor$?

Look at this



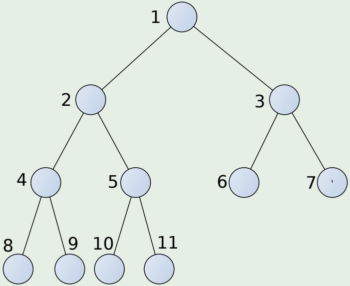
Thus, the nodes $\lfloor length[A]/2 \rfloor + 1, \lfloor length[A]/2 \rfloor + 2, ..., n$

- They are actually leaves.
- This can be proved by induction on n!!!
  - I leave this to you.

Question?

Why from $\lfloor {length[A]}/{2} \rfloor$?

Look at this



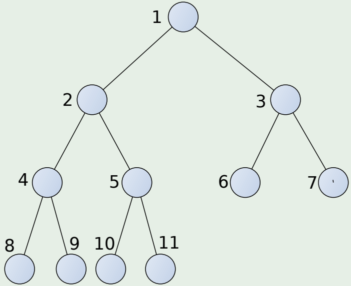
Thus, the nodes $\lfloor {length[A]}/{2} \rfloor + 1, \lfloor {length[A]}/{2} \rfloor + 2, ..., n$

- They are actually leaves.
- This can be proved by induction on n!!!
  - I leave this to you.

Question?

Why from $\left\lfloor {}^{length[A]}/_{2} \right\rfloor$?

Look at this



Thus, the nodes $\left\lfloor {}^{length[A]}/_{2} \right\rfloor + 1, \left\lfloor {}^{length[A]}/_{2} \right\rfloor + 2, ..., n$

- They are actually leaves.
- This can be proved by induction on n!!!
  - I leave this to you.

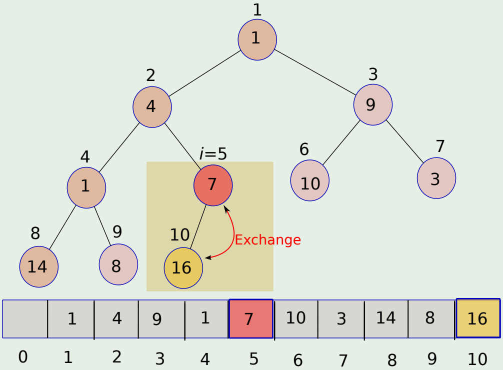
Question?

What about the loop invariance?

Look at the Board!!!

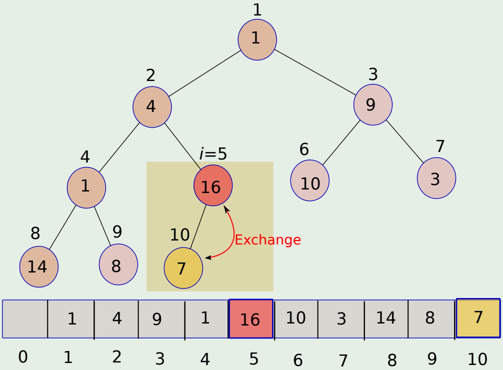
Build Max Heap: Using Max-Heapify

Example



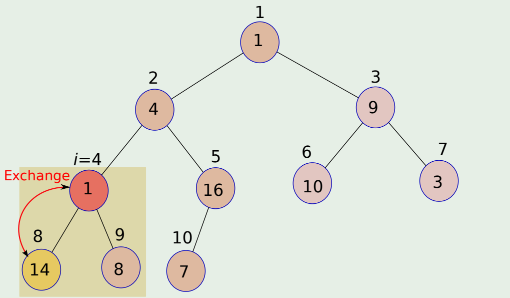
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

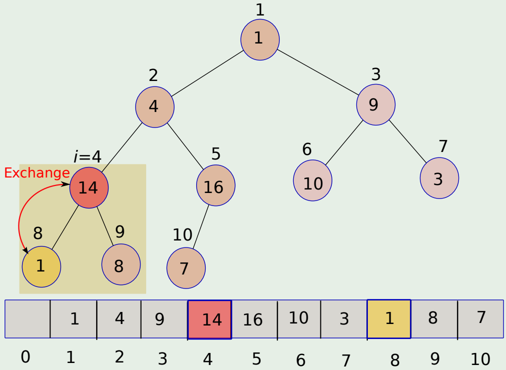
Example



0	1	2	3	4	5	6	7	8	9	10
	1	4	9	1	16	10	3	14	8	7

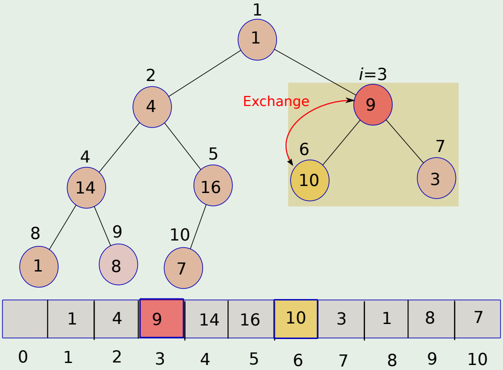
Build Max Heap: Using Max-Heapify

Example



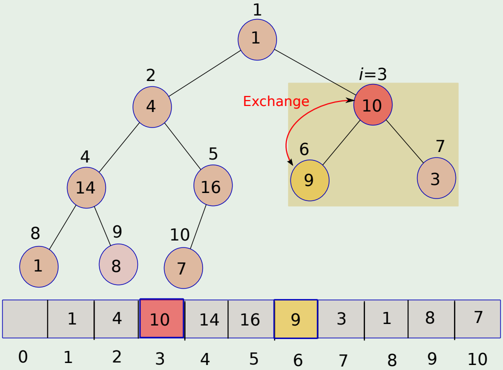
Build Max Heap: Using Max-Heapify

Example



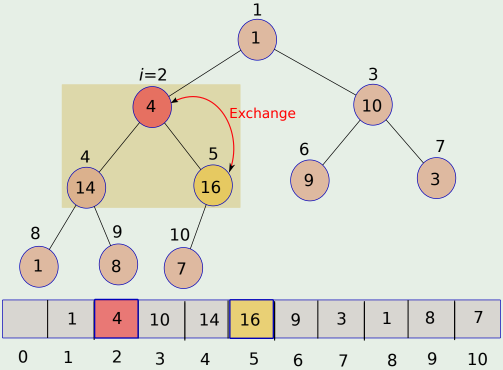
Build Max Heap: Using Max-Heapify

Example



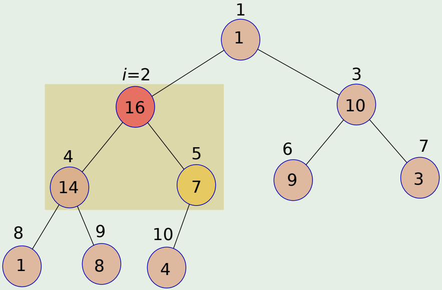
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

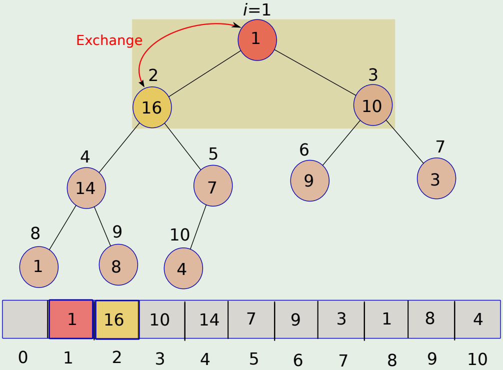
Example



	1	16	10	14	7	9	3	1	8	4
0	1	2	3	4	5	6	7	8	9	10

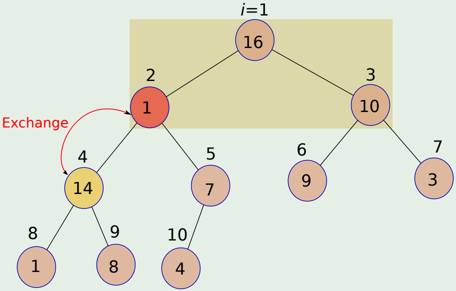
Build Max Heap: Using Max-Heapify

Example



Build Max Heap: Using Max-Heapify

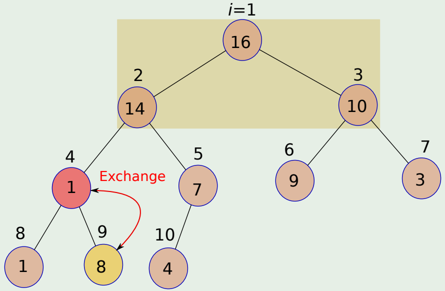
Example



	16	1	10	14	7	9	3	1	8	4
0	1	2	3	4	5	6	7	8	9	10

Build Max Heap: Using Max-Heapify

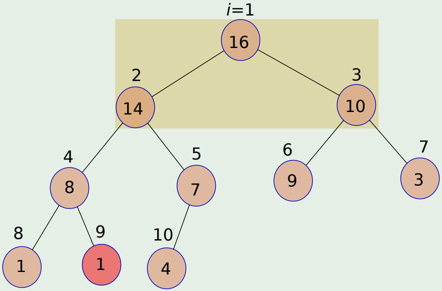
Example



	16	14	10	1	7	9	3	1	8	4
0	1	2	3	4	5	6	7	8	9	10

Build Max Heap: Using Max-Heapify

Example



	16	14	10	8	7	9	3	1	1	4
0	1	2	3	4	5	6	7	8	9	10

Height h of the Heap for Complexity of Build-Max-Heap

We can use the height of a three to derive a tight bound

- The height h is the number of edges on the longest simple downward path from the node to a leaf.
- You have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h, where n is the total number of nodes.

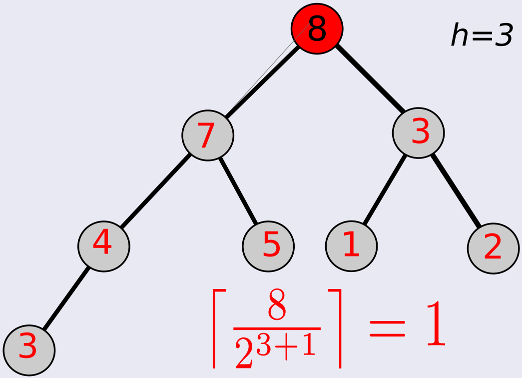
Height h of the Heap for Complexity of Build-Max-Heap

We can use the height of a three to derive a tight bound

- The height h is the number of edges on the longest simple downward path from the node to a leaf.
- You have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height h, where n is the total number of nodes.

Example

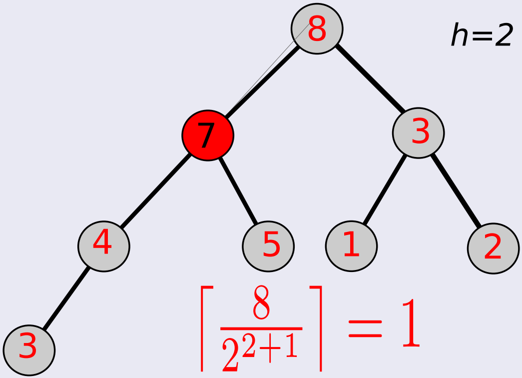
$h = 1$



$$\left\lceil \frac{8}{2^{3+1}} \right\rceil = 1$$

Furthermore

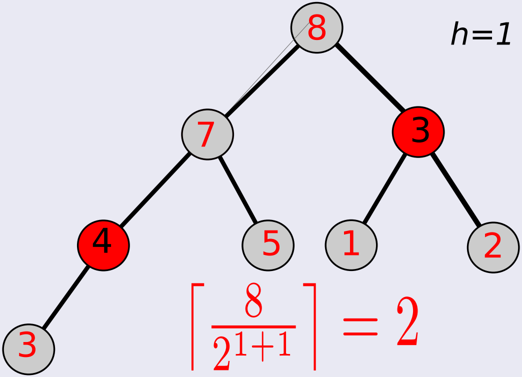
$h = 3$



$$\left\lceil \frac{8}{2^{2+1}} \right\rceil = 1$$

Furthermore

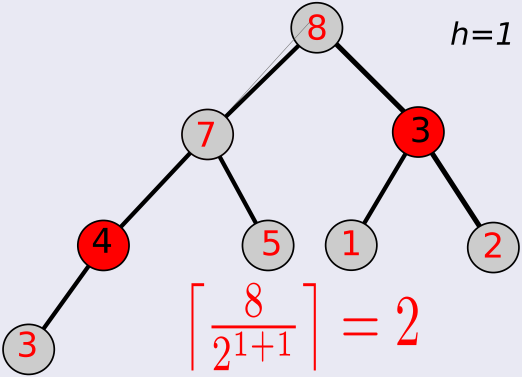
$h = 1$



$$\left\lceil \frac{8}{2^{1+1}} \right\rceil = 2$$

Furthermore

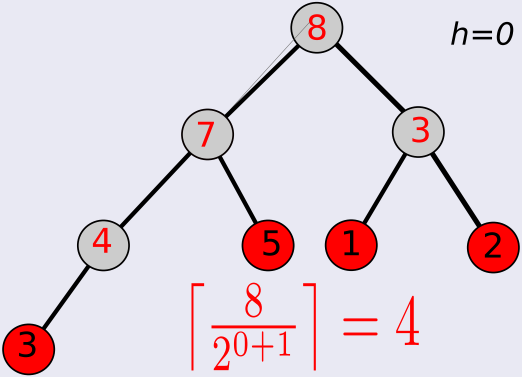
$h = 1$



$$\left\lceil \frac{8}{2^{1+1}} \right\rceil = 2$$

Furthermore

$h = 0$



$$\left\lceil \frac{8}{2^{0+1}} \right\rceil = 4$$

Cost of Building the Build-Max-Heap

Possible cost

$$O\left(n \log_2 n\right) \tag{9}$$

Cost of Building the Build-Max-Heap

We have that you have

- The number of nodes **explored horizontally** by the “for” loop can be bounded by

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \tag{10}$$

- The depth of the **Max-Heapify** is

$$O\left(h\right) \tag{11}$$

Cost of Building the Build-Max-Heap

We have that you have

- The number of nodes **explored horizontally** by the “for” loop can be bounded by

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \tag{10}$$

- The depth of the **Max-Heapify** is

$$O(h) \tag{11}$$

Therefore we have the following total tighter cost

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

Cost of Building the Build-Max-Heap

We have that you have

- The number of nodes **explored horizontally** by the "for" loop can be bounded by

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \tag{10}$$

- The depth of the **Max-Heapify** is

$$O(h) \tag{11}$$

Therefore we have the following total tighter cost

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

Cost of Building the Build-Max-Heap

We have that you have

- The number of nodes **explored horizontally** by the "for" loop can be bounded by

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \tag{10}$$

- The depth of the **Max-Heapify** is

$$O\left(h\right) \tag{11}$$

Therefore we have the following total tighter cost

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

Cost of Building the Build-Max-Heap

We have that you have

- The number of nodes **explored horizontally** by the "for" loop can be bounded by

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \tag{10}$$

- The depth of the **Max-Heapify** is

$$O\left(h\right) \tag{11}$$

Therefore we have the following total tighter cost

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

Thus

From (A.8) at Cormen's

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Thus, we have that

Thus

From (A.8) at Cormen's

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Thus, we have that

$$O\left(n\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$= O\left(n\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right)$$

$$= O\left(n\left[\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right]\right)$$

$$= O(n)$$

Thus

From (A.8) at Cormen's

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Thus, we have that

$$\begin{aligned} O\left(n\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) &= O\left(n\sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O\left(n\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right) \\ &= O\left(n\left[\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right]\right) \\ &= O(n) \end{aligned}$$

Thus

From (A.8) at Cormen's

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Thus, we have that

$$\begin{aligned}
O\left(n\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) &= O\left(n\sum_{h=0}^{\infty} \frac{h}{2^h}\right)\\
&= O\left(n\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right)\\
&= O\left(n\left[\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right]\right)\\
&= O(n)
\end{aligned}$$

Thus

From (A.8) at Cormen's

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Thus, we have that

$$\begin{aligned}
O\left(n\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) &= O\left(n\sum_{h=0}^{\infty} \frac{h}{2^h}\right)\\
&= O\left(n\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right)\\
&= O\left(n\left[\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right]\right)\\
&= O\left(n\right)
\end{aligned}$$

Outline

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

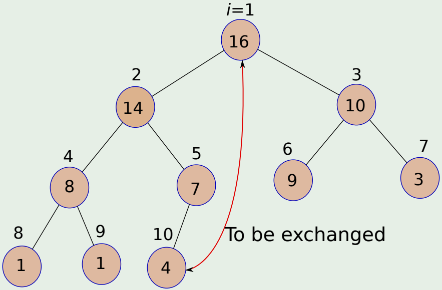
Heapsort(A)

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

Figure: Heapsort

Heap Sort: Using Max-Heapify

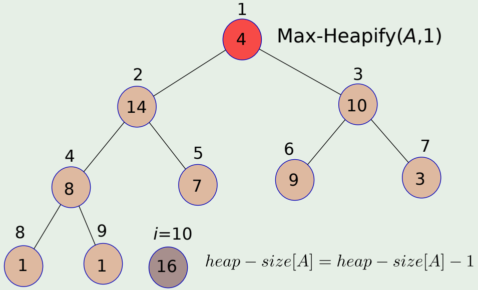
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	16	14	10	8	7	9	3	1	1	4
0	1	2	3	4	5	6	7	8	9	10

Heap Sort: Using Max-Heapify

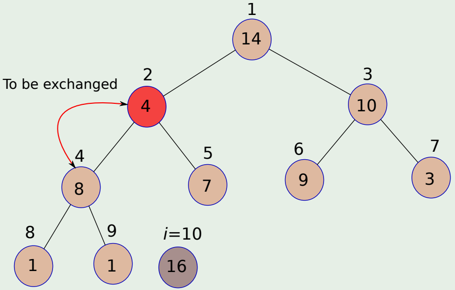
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	4	14	10	8	7	9	3	1	1	16
0	1	2	3	4	5	6	7	8	9	10

Heap Sort: Using Max-Heapify

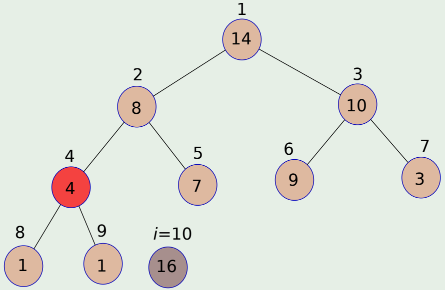
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	14	4	10	8	7	9	3	1	1	16
0	1	2	3	4	5	6	7	8	9	10

Heap Sort: Using Max-Heapify

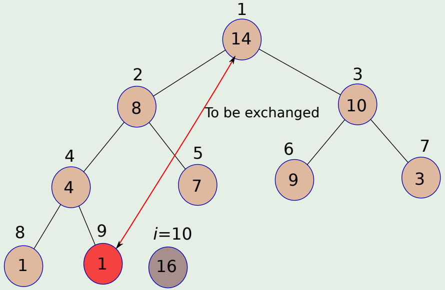
Example: Heapsort in action! By Moving the top element to the bottom position!!!



0	1	2	3	4	5	6	7	8	9	10
	14	8	10	4	7	9	3	1	1	16

Heap Sort: Using Max-Heapify

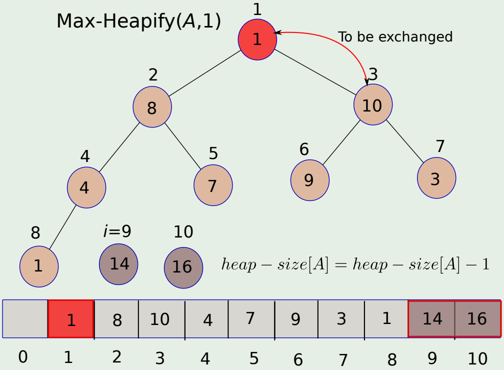
Example: Heapsort in action! By Moving the top element to the bottom position!!!



0	1	2	3	4	5	6	7	8	9	10
	14	8	10	4	7	9	3	1	1	16

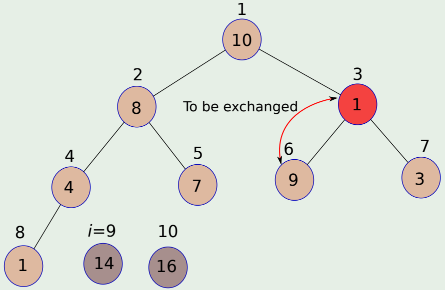
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Heap Sort: Using Max-Heapify

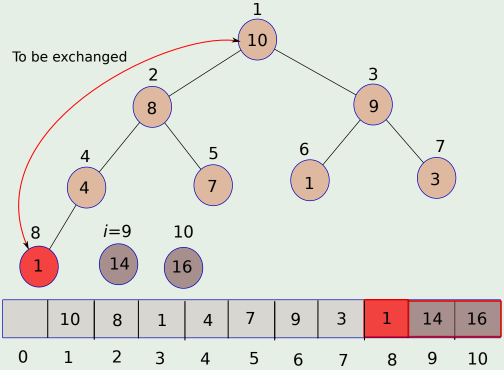
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	10	8	1	4	7	9	3	1	14	16
0	1	2	3	4	5	6	7	8	9	10

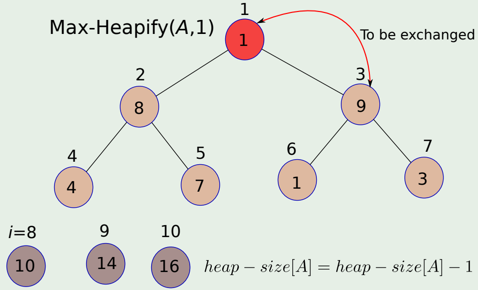
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Heap Sort: Using Max-Heapify

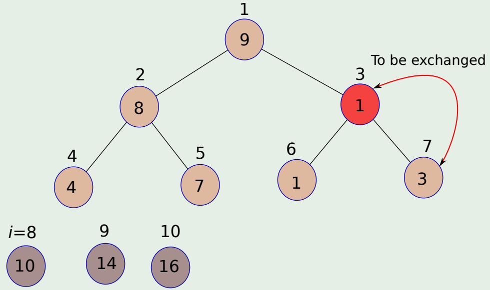
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	1	8	1	4	7	9	3	10	14	16
0	1	2	3	4	5	6	7	8	9	10

Heap Sort: Using Max-Heapify

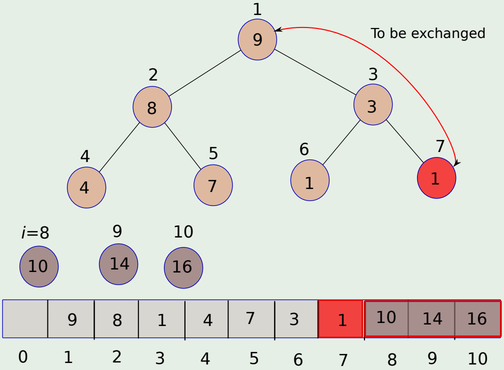
Example: Heapsort in action! By Moving the top element to the bottom position!!!



0	1	2	3	4	5	6	7	8	9	10
	9	8	1	4	7	1	3	10	14	16

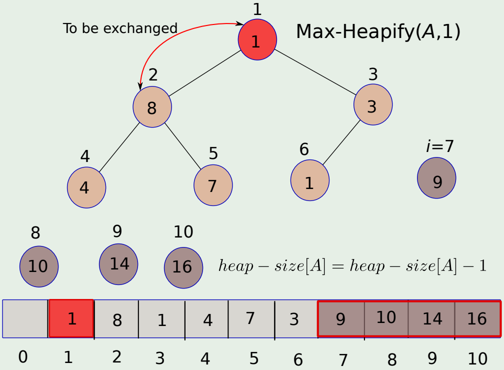
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



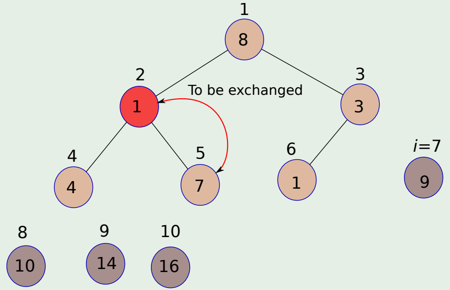
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Heap Sort: Using Max-Heapify

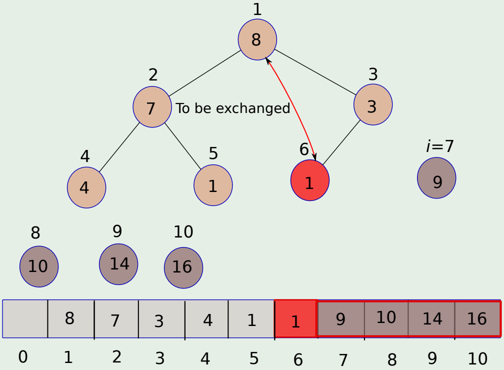
Example: Heapsort in action! By Moving the top element to the bottom position!!!



	8	1	3	4	7	1	9	10	14	16
0	1	2	3	4	5	6	7	8	9	10

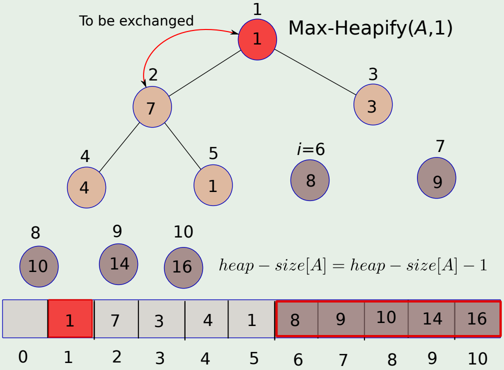
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



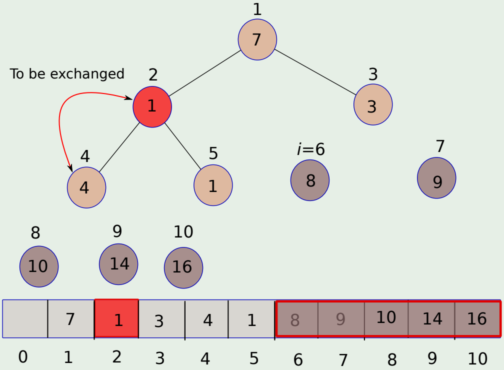
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



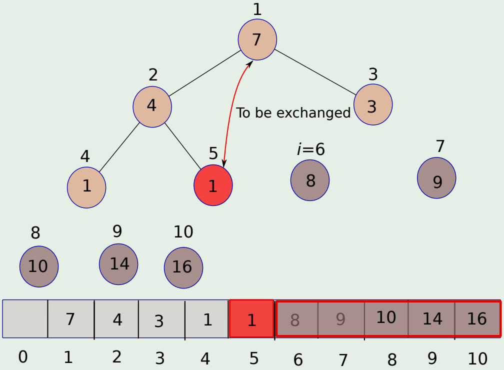
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



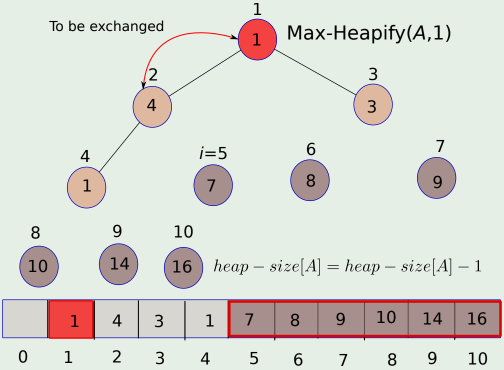
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



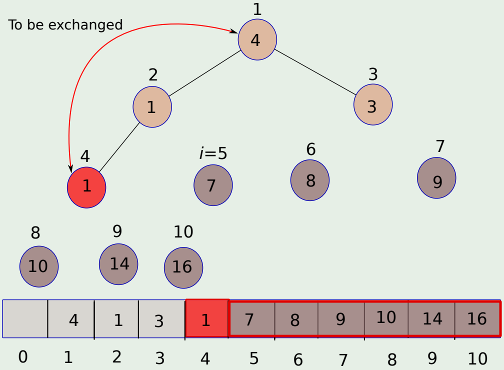
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



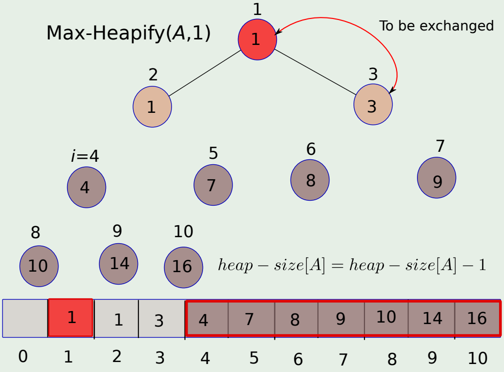
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



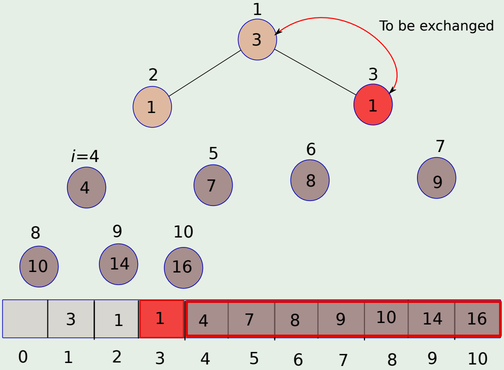
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



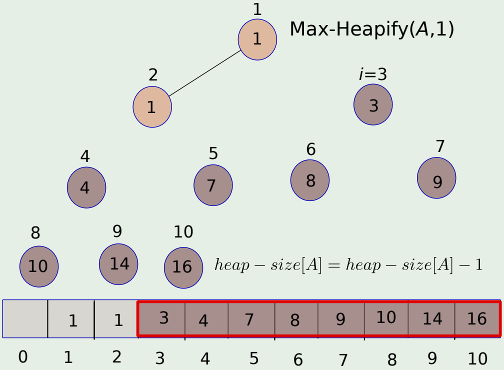
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



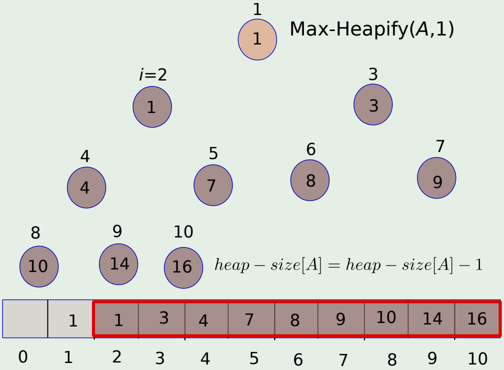
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



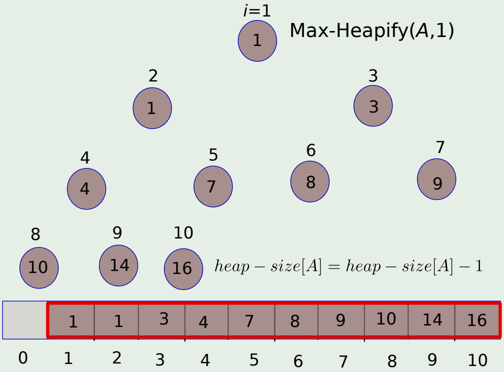
Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Heap Sort: Using Max-Heapify

Example: Heapsort in action! By Moving the top element to the bottom position!!!



Heap Sort: Using Max-Heapify

Cost

$O(n \log n)$

Outline

Applications of Heap Data Structure

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

Applications of Heap Data Structure

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
   1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
   1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
    1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
    1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
    1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
    1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
    1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
    1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Priority Queues

Here, Heaps can be modified to support insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time

This has direct applications

1. Bandwidth management:
    1. Many modern protocols for Local Area Networks include the concept of Priority Queues at the Media Access Control (MAC).
2. Discrete Event Simulations
3. Schedulers
4. Huffman coding
5. The Real-time Optimally Adapting Meshes (ROAM)
    1. It computes a dynamically changing triangulation of a terrain using two priority queues.

Applications of Heap Data Structure

Heap Sort of Arrays

Clearly, if the list of numbers is stored in an array!!!

Outline

Heap Sort: Exercices

From Cormen's book

- 6.1-1
- 6.1-4
- 6.1-7
- 6.2-5
- 6.2-6
- 6.3-3
- 6.4-2
- 6.4-3
- 6.4-4

Outline

Who invented Quicksort?

Imagine this

The Quicksort algorithm was developed in 1960 by Tony Hoare (He has a postgraduate certificate in Statistics) while in the Soviet Union, as a visiting student at Moscow State University.

Why?

At that time, Hoare worked in a project on machine translation for the National Physical Laboratory.

To do

He developed the algorithm in order to sort the words to be translated.

Who invented Quicksort?

Imagine this

The Quicksort algorithm was developed in 1960 by Tony Hoare (He has a postgraduate certificate in Statistics) while in the Soviet Union, as a visiting student at Moscow State University.

Why?

At that time, Hoare worked in a project on machine translation for the National Physical Laboratory.

To do

He developed the algorithm in order to sort the words to be translated.

Who invented Quicksort?

Imagine this

The Quicksort algorithm was developed in 1960 by Tony Hoare (He has a postgraduate certificate in Statistics) while in the Soviet Union, as a visiting student at Moscow State University.

Why?

At that time, Hoare worked in a project on machine translation for the National Physical Laboratory.

To do

He developed the algorithm in order to sort the words to be translated.

Outline

Imagine the following...

First Attempt

We want an algorithm that can sort by using the Divide and Conquer method

Now, we have the following constraint

We need to use the same array to do the sorting!!! Sorting in place!!!

What if we use the following strategy

Given a number in the array!!!

- Move some elements to the left of the number!!!
- Move some other elements to the right of the number!!!

Imagine the following...

First Attempt

We want an algorithm that can sort by using the Divide and Conquer method

Now, we have the following constraint

We need to use the same array to do the sorting!!! Sorting in place!!!

What if we use the following strategy

Given a number in the array!!!

- Move some elements to the left of the number!!!
- Move some other elements to the right of the number!!!

Imagine the following...

First Attempt

We want an algorithm that can sort by using the Divide and Conquer method

Now, we have the following constraint

We need to use the same array to do the sorting!!! Sorting in place!!!

What if we use the following strategy

Given a number in the array!!!

- Move some elements to the left of the number!!!
- Move some other elements to the right of the number!!!

Something like

We have...

9	8	2	4	5	10	6	3	7

2	4	3	5	9	8	10	6	7

Now What?

- Any Ideas?
- What about our old friend? Recursion!!!

Something like

We have...

9	8	2	4	5	10	6	3	7

2	4	3	5	9	8	10	6	7

Now What?

- Any Ideas?
- What about our old friend? Recursion!!!

The Divide and Conquer Quicksort

Divide Process

1. Compute the index q as part of this partitioning procedure.
2. Partition (rearrange) the array $A[p, ..., r]$ into two (possibly empty) sub-arrays $A[p, ..., q-1]$ and $A[q+1, ..., r]$
   1. each element of $A[p, ..., q-1]$ is less than or equal to $A[q]$.
   2. $A[q]$ is less than or equal to each element of $A[q+1, ..., r]$.

The Divide and Conquer Quicksort

Divide Process

1. Compute the index q as part of this partitioning procedure.
2. Partition (rearrange) the array $A[p, ..., r]$ into two (possibly empty) sub-arrays $A[p, ..., q-1]$ and $A[q+1, ..., r]$
   1. each element of $A[p, ..., q-1]$ is less than or equal to $A[q]$.
   2. $A[q]$ is less than or equal to each element of $A[q+1, ..., r]$.

The Divide and Conquer Quicksort

Divide Process

1. Compute the index q as part of this partitioning procedure.
2. Partition (rearrange) the array $A[p, ..., r]$ into two (possibly empty) sub-arrays $A[p, ..., q-1]$ and $A[q+1, ..., r]$
   1. each element of $A[p, ..., q-1]$ is less than or equal to $A[q]$.
   2. $A[q]$ is less than or equal to each element of $A[q+1, ..., r]$.

The Divide and Conquer Quicksort

Divide Process

1. Compute the index q as part of this partitioning procedure.
2. Partition (rearrange) the array $A[p, ..., r]$ into two (possibly empty) sub-arrays $A[p, ..., q-1]$ and $A[q+1, ..., r]$
   1. each element of $A[p, ..., q-1]$ is less than or equal to $A[q]$.
   2. $A[q]$ is less than or equal to each element of $A[q+1, ..., r]$.

The Divide and Conquer Quicksort

Conquer

Sort the two sub-arrays $A[p, ..., q-1]$ and $A[q+1, ..., r]$ by recursive calls to quicksort.

Combine

Since the sub-arrays are sorted in place, no work is needed to combine them: the entire array $A[p, ..., r]$ is now sorted.

The Divide and Conquer Quicksort

Conquer

Sort the two sub-arrays $A[p, ..., q-1]$ and $A[q+1, ..., r]$ by recursive calls to quicksort.

Combine

Since the sub-arrays are sorted in place, no work is needed to combine them: the entire array $A[p, ..., r]$ is now sorted.

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q-1)$
4. $\quad$ **Quicksort**$(A, q+1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q-1)$
4. $\quad$ **Quicksort**$(A, q+1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q-1)$
4. $\quad$ **Quicksort**$(A, q+1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q - 1)$
4. $\quad$ **Quicksort**$(A, q + 1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$**Quicksort**$(A, p, q-1)$
4. $\quad$**Quicksort**$(A, q+1, r)$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

- $x = A[r]$
- $i = p - 1$
- for $j = p$ to $r - 1$
    - if $A[j] \leq x$
        - $i = i + 1$
        - exchange $A[i]$ with $A[j]$
- exchange $A[i+1]$ with $A[r]$
- return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.   if $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.   if $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i + 1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

$x = A[r]$

$i = p - 1$

for $j = p$ to $r - 1$

    if $A[j] \leq x$

        $i = i + 1$

        exchange $A[i]$ with $A[j]$

exchange $A[i+1]$ with $A[r]$

return $i + 1$

Quicksort: What is the Invariance?

Loop Invariance

1. **If** $p \leq k \leq$ **i, then** $A[k] \leq x$**.**
2. **If** $i+1 \leq k \leq j-1$**, then** $A[k] > x$**.**
3. **If** $k = r$ **, then** $A[k] = x$**.**
4. **UNKNOWN**

Quicksort: What is the Invariance?

Loop Invariance

1. **If** $p \leq k \leq$ **i, then** $A[k] \leq x$.
2. **If** $i+1 \leq k \leq j-1$**, then** $A[k] > x$.
3. **If** $k = r$ **, then** $A[k] = x$.
4. **UNKNOWN**

Quicksort: What is the Invariance?

Loop Invariance

1. **If** $p \leq k \leq$ **i, then** $A[k] \leq x$**.**
2. **If** $i + 1 \leq k \leq j - 1$**, then** $A[k] > x$**.**
3. **If** $k = r$ **, then** $A[k] = x$**.**
4. UNKNOWN

Quicksort: What is the Invariance?

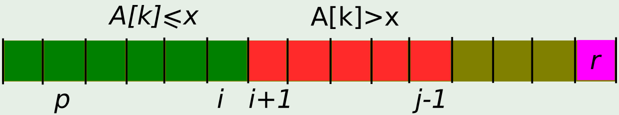
Loop Invariance

1. **If** $p \leq k \leq$ **i, then** $A[k] \leq x$**.**
2. **If** $i + 1 \leq k \leq j - 1$**, then** $A[k] > x$**.**
3. **If** $k = r$ **, then** $A[k] = x$**.**
4. **UNKNOWN**

Quicksort: What is the Invariance?

Loop Invariance

1. **If** $p \leq k \leq$ **i, then** $A[k] \leq x$**.**
2. **If** $i + 1 \leq k \leq j - 1$**, then** $A[k] > x$**.**
3. **If** $k = r$ **, then** $A[k] = x$**.**
4. **UNKNOWN**



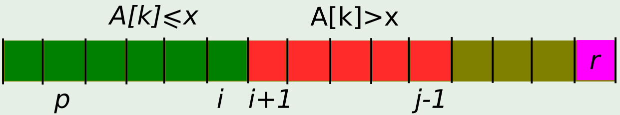
Proof of the Loop Invariance

Look at the Board.

Quicksort: What is the Invariance?

Loop Invariance

1. **If** $p \leq k \leq$ **i, then** $A[k] \leq x$**.**
2. **If** $i + 1 \leq k \leq j - 1$**, then** $A[k] > x$**.**
3. **If** $k = r$ **, then** $A[k] = x$**.**
4. **UNKNOWN**



Proof of the Loop Invariance

Look at the Board.

Outline

Complexity Analysis

Best-case Analysis

- Partition returns two arrays size $\frac{n}{2}$ and $\frac{n}{2} - 1$.
- Then, we have the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

What about the Worst-Case?

- Partition returns two arrays, one of size 0 and one of size $n - 1$.
- Then, we have the recurrence:

$$T(n) = T(n-1) + \Theta(n) = O(n^2). \tag{12}$$

Complexity Analysis

Best-case Analysis

- Partition returns two arrays size $\frac{n}{2}$ and $\frac{n}{2} - 1$.
- Then, we have the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

What about the Worst-Case?

- Partition returns two arrays, one of size 0 and one of size $n-1$.
- Then, we have the recurrence:

$$T(n) = T(n-1) + \Theta(n) = O(n^2). \tag{12}$$

Complexity Analysis

Best-case Analysis

- Partition returns two arrays size $\frac{n}{2}$ and $\frac{n}{2} - 1$.
- Then, we have the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

What about the Worst-Case?

- Partition returns two arrays, one of size 0 and one of size $n-1$.
- Then, we have the recurrence:

$$T(n) = T(n-1) + \Theta(n) = O(n^2). \quad (12)$$

Complexity Analysis

Best-case Analysis

- Partition returns two arrays size $\frac{n}{2}$ and $\frac{n}{2} - 1$.
- Then, we have the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.

What about the Worst-Case?

- Partition returns two arrays, one of size 0 and one of size $n - 1$.
- Then, we have the recurrence:

$$T(n) = T(n-1) + \Theta(n) = O(n^2). \tag{12}$$

Complexity Analysis

Best-case Analysis

- Partition returns two arrays size $\frac{n}{2}$ and $\frac{n}{2}-1$.
- Then, we have the recurrence $T(n)=2T\left(\frac{n}{2}\right)+\Theta(n)$.

What about the Worst-Case?

- Partition returns two arrays, one of size 0 and one of size $n-1$.
- Then, we have the recurrence:

$$T(n)=T(n-1)+\Theta(n)=O(n^2). \tag{12}$$

Outline

What about a No So Unbalanced Partition?

What are you talking about?

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) \qquad (13)$$

This can happen when

The pivot split the array in two sub-array...

x_1	pivot	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Even when this happen

Using the tree method!!! We notice something weird!!!

What about a No So Unbalanced Partition?

What are you talking about?

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) \tag{13}$$

This can happen when

The pivot split the array in two sub-array...

x_1	pivot	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Even when this happen

Using the tree method!!! We notice something weird!!!

What about a No So Unbalanced Partition?

What are you talking about?

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) \quad (13)$$

This can happen when

The pivot split the array in two sub-array...

x_1	pivot	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

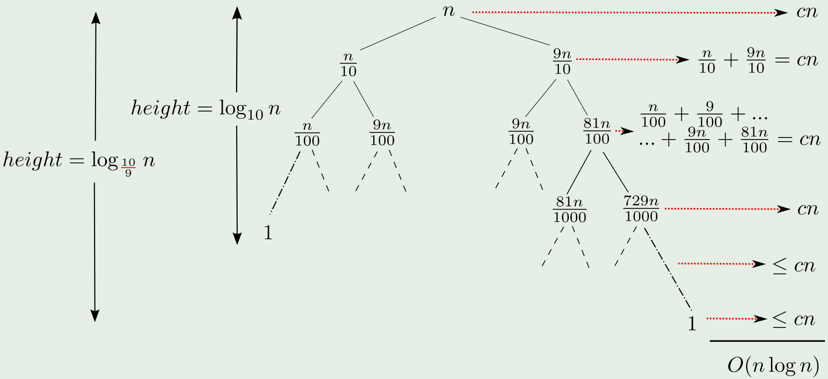
Even when this happen

Using the tree method!!! We notice something weird!!!

Unbalanced Partition Tree Method Analysis

Unbalanced partitioning returns a $O\left(n \log n\right)$

After certain level, the total steps are $\leq$ than cn!!!



Outline

Worst Case Complexity Analysis

We do not know which pivot gets the worst case

Thus, Why do not ask the recursion each possible pivot?

We can get the worst case asking

Worst Case Complexity Analysis

We do not know which pivot gets the worst case

Thus, Why do not ask the recursion each possible pivot?

Remember!!!

After all, we can split the sub-arrays in any way we want!!!

$$T(q) + T(n-q-1) \tag{14}$$

We can get the worst case asking

$$\max_{0 \leq q \leq n-1} \left(T(q) + T(n-q-1) \right) \tag{15}$$

cinvestav

Worst Case Complexity Analysis

We do not know which pivot gets the worst case

Thus, Why do not ask the recursion each possible pivot?

Remember!!!

After all, we can split the sub-arrays in any way we want!!!

$$T\left(q\right)+T\left(n-q-1\right) \tag{14}$$

We can get the worst case asking

$$\max_{0\leq q\leq n-1}\left(T(q)+T(n-q-1)\right) \tag{15}$$

Worst Case Complexity Analysis

We do not know which pivot gets the worst case

Thus, Why do not ask the recursion each possible pivot?

Remember!!!

After all, we can split the sub-arrays in any way we want!!!

$$T(q) + T(n - q - 1) \tag{14}$$

We can get the worst case asking

$$\max_{0 \leq q \leq n-1} \left(T(q) + T(n - q - 1) \right) \tag{15}$$

cinvestav

Worst Case Complexity Analysis

We do not know which pivot gets the worst case

Thus, Why do not ask the recursion each possible pivot?

Remember!!!

After all, we can split the sub-arrays in any way we want!!!

$$T(q) + T(n-q-1) \tag{14}$$

We can get the worst case asking

$$\max_{0 \leq q \leq n-1} \left(T(q) + T(n-q-1)\right) \tag{15}$$

Worst Case Complexity Analysis

Worst-case Recursion

$$T(n) = \max_{0 \leq q \leq n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n) \qquad (16)$$

By substitution, we can prove

Complexity $O(n^2)$.

This can be proved as follows

BLACKBOARD!

Worst Case Complexity Analysis

Worst-case Recursion

$$T(n) = \max_{0 \leq q \leq n-1} \left(T(q) + T(n - q - 1) \right) + \Theta(n) \tag{16}$$

By substitution, we can prove

Complexity $O(n^2)$.

This can be proved as follows

BLACKBOARD!

Worst Case Complexity Analysis

Worst-case Recursion

$$T(n) = \max_{0 \leq q \leq n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n) \tag{16}$$

By substitution, we can prove

Complexity $O(n^2)$.

This can be proved as follows

BLACKBOARD!

Outline

Remember?

The use of uniform distribution

To get the average behavior!!!

In many cases

It is better than the worst case scenario....

Thus

We can introduce randomization in the Quicksort.

Remember?

The use of uniform distribution

To get the average behavior!!!

In many cases

It is better than the worst case scenario....

Thus

We can introduce randomization in the Quicksort.

Remember?

The use of uniform distribution

To get the average behavior!!!

In many cases

It is better than the worst case scenario....

Thus

We can introduce randomization in the Quicksort.

Randomized Quicksort

RANDOMIZED-QUICKSORT(A,p,r)

Randomized-Quicksort(A, p, r)

1. if $p < r$
2. $\quad q =$Randomized-Partition(A, p, r)
3. $\quad$ **Randomized-Quicksort**$(A, p, q-1)$
4. $\quad$ **Randomized-Quicksort**$(A, q+1, r)$

Randomized Quicksort

RANDOMIZED-QUICKSORT(A,p,r)

Randomized-Quicksort(A, p, r)

```
if p < r
    q = Randomized-Partition(A, p, r)
    Randomized-Quicksort(A, p, q − 1)
    Randomized-Quicksort(A, q + 1, r)
```

RANDOMIZED-PARTITION(A,p,r)

Randomized Quicksort

RANDOMIZED-QUICKSORT(A,p,r)

Randomized-Quicksort(A, p, r)

1. if $p < r$
2. $\quad q =$Randomized-Partition(A, p, r)
3. $\quad$**Randomized-Quicksort**$(A, p, q-1)$
4. $\quad$**Randomized-Quicksort**$(A, q+1, r)$

RANDOMIZED-PARTITION(A,p,r)

Randomized Quicksort

RANDOMIZED-QUICKSORT(A,p,r)

Randomized-Quicksort(A, p, r)

1. if $p < r$
2. $\quad q =$Randomized-Partition(A, p, r)
3. $\quad$ **Randomized-Quicksort**$(A, p, q-1)$
4. $\quad$ **Randomized-Quicksort**$(A, q+1, r)$

RANDOMIZED-PARTITION(A,p,r)

Randomized Quicksort

RANDOMIZED-QUICKSORT(A,p,r)

Randomized-Quicksort(A, p, r)

1. if $p < r$
2. $q =$Randomized-Partition(A, p, r)
3. **Randomized-Quicksort**$(A, p, q-1)$
4. **Randomized-Quicksort**$(A, q+1, r)$

RANDOMIZED-PARTITION(A,p,r)

Randomized-Partition(A, p, r)

1. $i =$Random(p, r)
2. exchange $A[r]$ with $A[i]$
3. **return** Partition(A, p, r)

Outline

Expected Running Time of Randomized Quicksort

Expected running time

The expected running time for the Randomized Quicksort algorithm arises from the following lemma.

Lemma 7.1 (Cormen's book)

- Let X be the number of comparisons performed in line 4 of PARTITION algorithm over the entire execution of QUICKSORT on an n-element array.
- Then, the running time of QUICKSORT is $O(n + X)$.

Now the proof of the expected running time.

BLACKBOARD!

Expected Running Time of Randomized Quicksort

Expected running time

The expected running time for the Randomized Quicksort algorithm arises from the following lemma.

Lemma 7.1 (Cormen's book)

- Let X be the number of comparisons performed in line 4 of PARTITION algorithm over the entire execution of QUICKSORT on an n-element array.
- Then, the running time of QUICKSORT is $O(n + X)$.

Now the proof of the expected running time.

BLACKBOARD!

Expected Running Time of Randomized Quicksort

Expected running time

The expected running time for the Randomized Quicksort algorithm arises from the following lemma.

Lemma 7.1 (Cormen's book)

- Let X be the number of comparisons performed in line 4 of PARTITION algorithm over the entire execution of QUICKSORT on an n-element array.
- Then, the running time of QUICKSORT is $O(n+X)$.

Now the proof of the expected running time.

BLACKBOARD!

Therefore

It is possible to conclude that

The Average Time Complexity of the Quicksort is $O(n \log n)$

Applications

Sorting in Special Environments

Example: Using Massive Parallel Stream Processors.

Multi-Objective Optimization

Yes!!! Numerical Analysis using the Quick Sort!!!

Real-Time Visualization of Large Time-Varying Molecules

Use the distance of the atoms to the viewers - the Painters Algorithms!!!

Applications

Sorting in Special Environments

Example: Using Massive Parallel Stream Processors.

Multi-Objective Optimization

Yes!!! Numerical Analysis using the Quick Sort!!!

Real-Time Visualization of Large Time-Varying Molecules

Use the distance of the atoms to the viewers - the Painters Algorithms!!!

Applications

Sorting in Special Environments

Example: Using Massive Parallel Stream Processors.

Multi-Objective Optimization

Yes!!! Numerical Analysis using the Quick Sort!!!

Real-Time Visualization of Large Time-Varying Molecules

Use the distance of the atoms to the viewers - the Painters Algorithms!!!

Outline

Basic Concepts

Mergesort and Heapsort

- They are algorithms that sort in $O(n \log n)$.
- It is more we can give a sequence such that $\Omega(n \log n)$.

Basic Concepts

Mergesort and Heapsort

- They are algorithms that sort in $O(n \log n)$.
- It is more we can give a sequence such that $\Omega(n \log n)$.

Property

- The sorted order they determine is based only on comparisons between the input elements.
- We call such sorting algorithms comparison sorts.

Basic Concepts

Mergesort and Heapsort

- They are algorithms that sort in $O(n \log n)$.
- It is more we can give a sequence such that $\Omega(n \log n)$.

Property

- The sorted order they determine is based only on comparisons between the input elements.
- We call such sorting algorithms comparison sorts.

Basic Concepts

Mergesort and Heapsort

- They are algorithms that sort in $O(n \log n)$.
- It is more we can give a sequence such that $\Omega(n \log n)$.

Property

- The sorted order they determine is based only on comparisons between the input elements.
- We call such sorting algorithms comparison sorts.

Theorem and Corollary

Theorem

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Corollary

Heapsort and Mergesort are asymptotically optimal comparison sorts.

Theorem and Corollary

Theorem

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Corollary

Heapsort and Mergesort are asymptotically optimal comparison sorts.

Outline

Exercises

Cormen's Chapter 7

- 7.1-4
- 7.2-3
- 7.2-5
- 7.4-1