More on Sorting: Quick Sort and Heap Sort

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

March 14, 2024

Outline

- Another divide-and-conquer sorting algorithm
- The *heap*
- Heap sort

Sorting Algorithms Seen So Far

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort				

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort				

Algorithm	Complexity			In place?
	worst	*average*	*best*	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort				

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
??		$\Theta(n \log n)$		***yes***
??	$\Theta(n \log n)$			***yes***

Using the Partitioning Algorithm

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

Using the Partitioning Algorithm

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1\rangle$

we pick a splitting value, say $v = 5$

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle$$

Using the Partitioning Algorithm

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle$$

Using the Partitioning Algorithm

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle \quad A_R = \langle 36, 21, 8, 13, 11, 20 \rangle$$

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1\rangle$

we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1\rangle \quad A_v = \langle 5, 5\rangle \quad A_R = \langle 36, 21, 8, 13, 11, 20\rangle$$

- *Can we use the same idea for sorting A?*

Using the Partitioning Algorithm

- *Basic step:* partition A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater than* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle \quad A_R = \langle 36, 21, 8, 13, 11, 20 \rangle$$

- *Can we use the same idea for sorting A?*

- *Can we partition A* ***in place****?*

Another Strategy for Sorting

- *Problem:* sorting

Another Strategy for Sorting

- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \dots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

- *Problem:* sorting

- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

2	36	4	21	8	13	11	20	5	5	1

Another Strategy for Sorting

- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \dots n]$ in three parts based on a chosen “pivot” value $v \in A$
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \dots n]$ contain elements that are *greater than* v

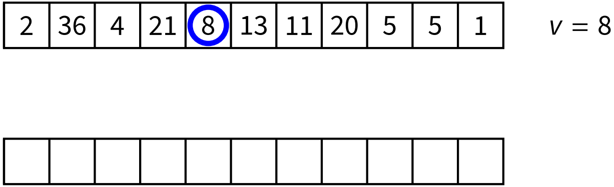
2	36	4	21	8	13	11	20	5	5	1

$v = 8$

Another Strategy for Sorting

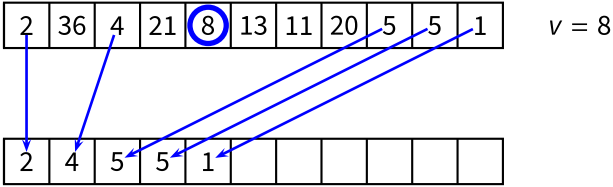
- *Problem:* sorting

- *Idea:* rearrange the sequence $A[1 \dots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \dots n]$ contain elements that are *greater than* v



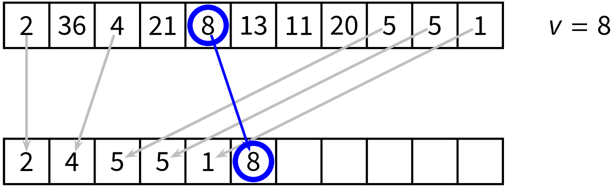
Another Strategy for Sorting

- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to v*
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than v*



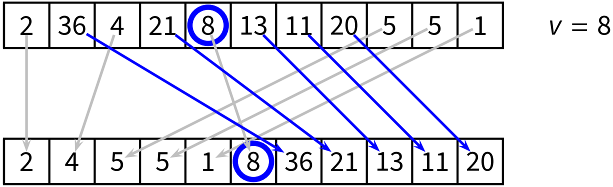
Another Strategy for Sorting

- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

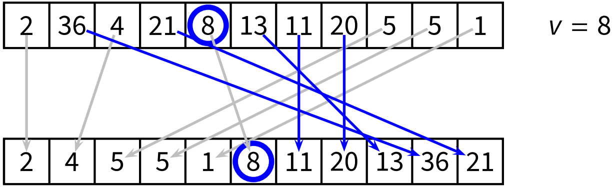


- *Problem:* sorting

- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen “pivot” value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

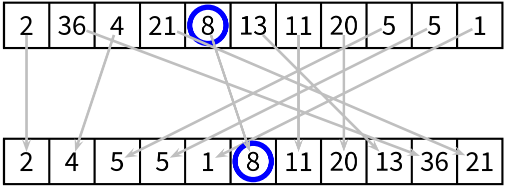


- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v



- *Problem:* sorting

- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

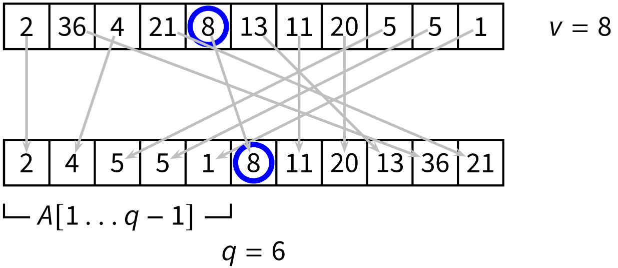


$v = 8$

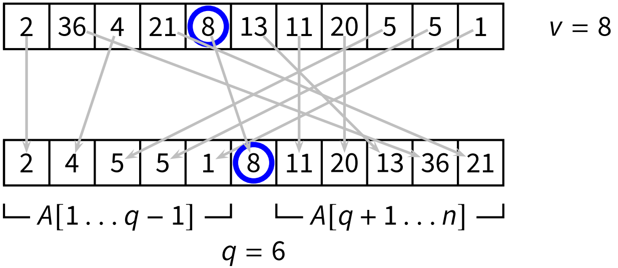
$q = 6$

Another Strategy for Sorting

- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \dots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \dots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \dots n]$ contain elements that are *greater than* v



- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to v*
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than v*



Another Divide-and-Conquer for Sorting

- ***Divide:***

Another Divide-and-Conquer for Sorting

- ***Divide:*** partition A in $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$ such that

$$1 \leq i < q < j \leq n \Rightarrow A[i] \leq A[q] \leq A[j]$$

Another Divide-and-Conquer for Sorting

- ***Divide:*** partition A in $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$ such that

$$1 \leq i < q < j \leq n \Rightarrow A[i] \leq A[q] \leq A[j]$$

- ***Conquer:***

Another Divide-and-Conquer for Sorting

- ***Divide:*** partition A in $A[1 \dots q-1]$ and $A[q+1 \dots n]$ such that

$$1 \le i < q < j \le n \Rightarrow A[i] \le A[q] \le A[j]$$

- ***Conquer:*** sort $A[1 \dots q-1]$ and $A[q+1 \dots n]$

Another Divide-and-Conquer for Sorting

- ***Divide:*** partition A in $A[1 \dots q-1]$ and $A[q+1 \dots n]$ such that

$$1 \leq i < q < j \leq n \Rightarrow A[i] \leq A[q] \leq A[j]$$

- ***Conquer:*** sort $A[1 \dots q-1]$ and $A[q+1 \dots n]$

- ***Combine:***

Another Divide-and-Conquer for Sorting

- ***Divide:*** partition A in $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$ such that

$$1 \le i < q < j \le n \Rightarrow A[i] \le A[q] \le A[j]$$

- ***Conquer:*** sort $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$
- ***Combine:*** nothing to do here
  - notice the difference with **MergeSort**

Another Divide-and-Conquer for Sorting

- ***Divide:*** partition A in $A[1 \dots q-1]$ and $A[q+1 \dots n]$ such that

$$1 \le i < q < j \le n \Rightarrow A[i] \le A[q] \le A[j]$$

- ***Conquer:*** sort $A[1 \dots q-1]$ and $A[q+1 \dots n]$
- ***Combine:*** nothing to do here
  - notice the difference with **MergeSort**

```
QuickSort(A, begin, end)
if begin < end
        q = Partition(A, begin, end)
        QuickSort(A, begin, q − 1)
        QuickSort(A, q + 1, end)
```

Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

36	11	5	21	1	13	2	20	5	4	8

Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

36	11	5	21	1	13	2	20	5	4	8

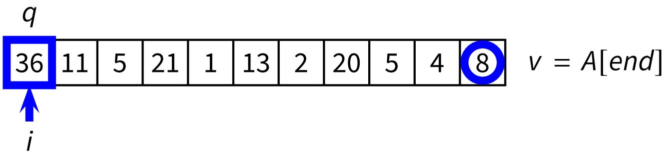
$v = A[end]$

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



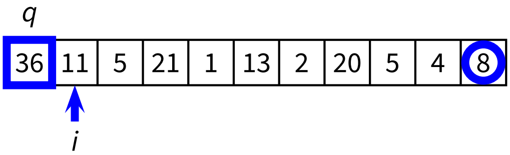
Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



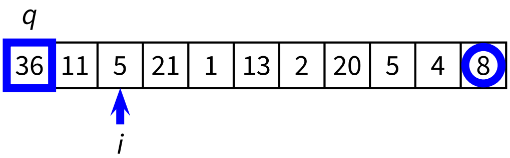
Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

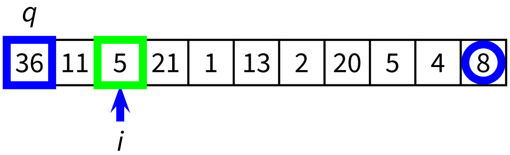
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

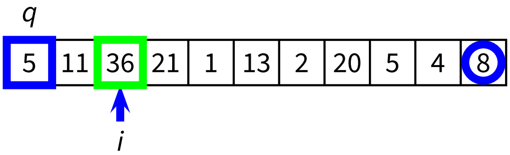


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

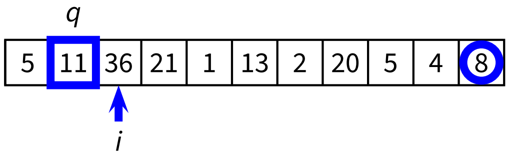
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

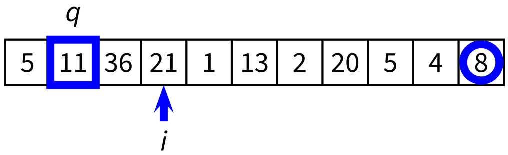


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

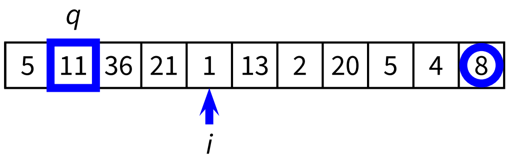


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

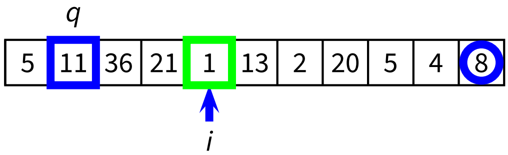
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

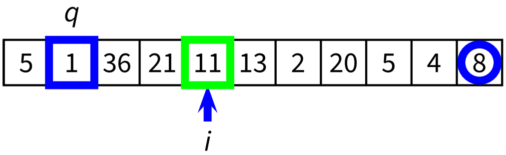


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



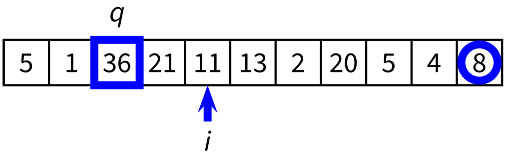
Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

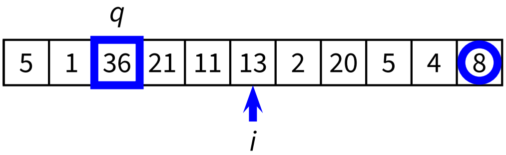
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

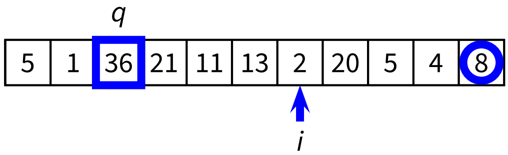


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

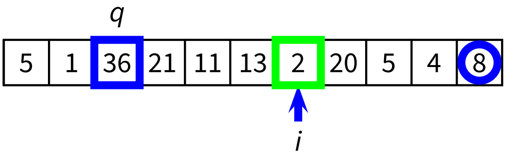
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

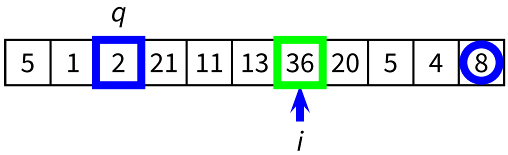


Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

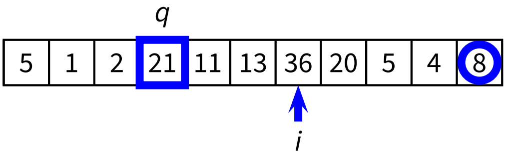


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

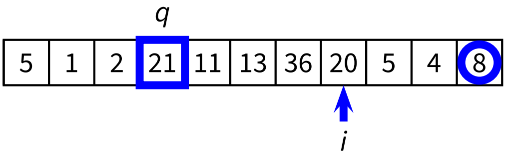


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

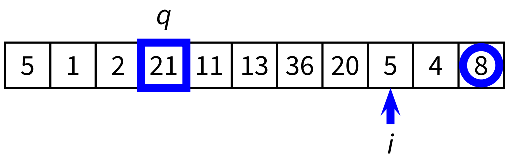
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

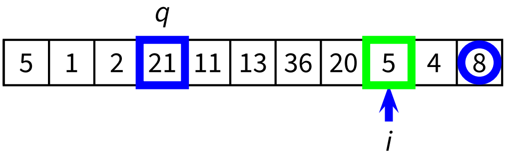
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

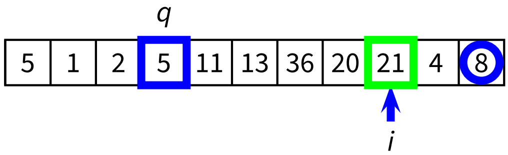


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



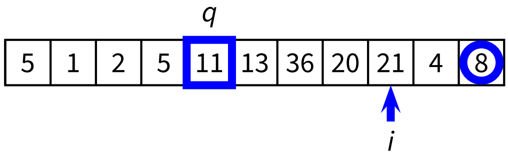
Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

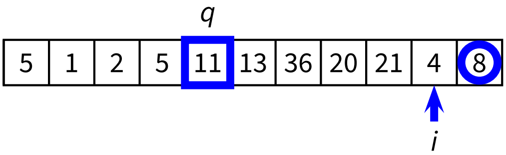
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

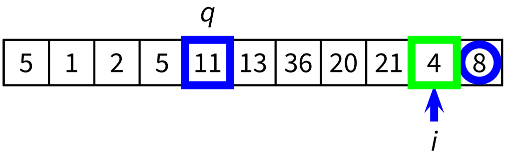


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

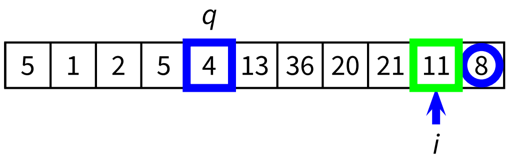
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

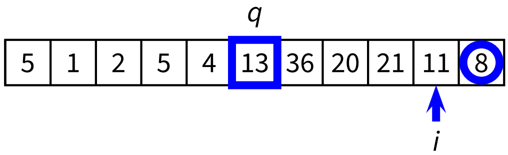


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



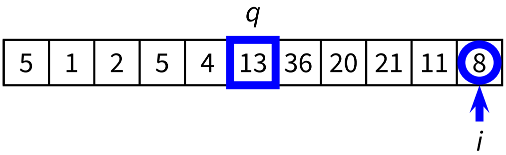
Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*

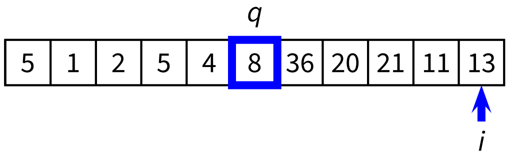
- Scan the array left-to-right, starting at position 2

- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$

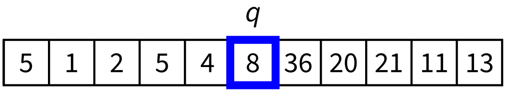


- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



Partition

- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow A[k] \leq v$
  - $q < k < i \Rightarrow A[k] > v$



Complete QUICKSORT Algorithm

```
PARTITION(A, begin, end)
q = begin
v = A[end]
for i = begin to end
    if A[i] ≤ v
        swap A[i] and A[q]
        q = q + 1
return q − 1
```

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

```
PARTITION(A, begin, end)
q = begin
v = A[end]
for i = begin to end
        if A[i] ≤ v
                swap A[i] and A[q]
                q = q + 1
return q − 1
```

```
PARTITION(A, begin, end)
  q = begin
  v = A[end]
  for i = begin to end
      if A[i] ≤ v
          swap A[i] and A[q]
          q = q + 1
  return q − 1
```

$$T(n) = \Theta(n)$$

Complexity of QuickSort

```
QuickSort(A, begin, end)
  if begin < end
      q = Partition(A, begin, end)
      QuickSort(A, begin, q − 1)
      QuickSort(A, q + 1, end)
```

Complexity of QuickSort

```
QuickSort(A, begin, end)
if begin < end
    q = Partition(A, begin, end)
    QuickSort(A, begin, q − 1)
    QuickSort(A, q + 1, end)
```

- Worst case

Complexity of QuickSort

```
QuickSort(A, begin, end)
if begin < end
        q = Partition(A, begin, end)
        QuickSort(A, begin, q − 1)
        QuickSort(A, q + 1, end)
```

- Worst case
  - $q = begin$ or $q = end$

```
QuickSort(A, begin, end)
  if begin < end
        q = Partition(A, begin, end)
        QuickSort(A, begin, q − 1)
        QuickSort(A, q + 1, end)
```

- Worst case
  - $q = begin$ or $q = end$
  - the partition transforms P of size n in P of size $n - 1$

Complexity of QuickSort

```
QuickSort(A, begin, end)
if begin < end
        q = Partition(A, begin, end)
        QuickSort(A, begin, q − 1)
        QuickSort(A, q + 1, end)
```

- Worst case
  - $q = begin$ or $q = end$
  - the partition transforms P of size n in P of size $n - 1$

$$T(n) = T(n-1) + \Theta(n)$$

```
QuickSort(A, begin, end)
if begin < end
    q = Partition(A, begin, end)
    QuickSort(A, begin, q − 1)
    QuickSort(A, q + 1, end)
```

- Worst case
  - $q = begin$ or $q = end$
  - the partition transforms P of size n in P of size $n-1$

$$T(n) = T(n-1) + \Theta(n)$$

$$T(n) = \Theta(n^2)$$

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

Complexity of QuickSort (2)

```
QuickSort(A, begin, end)
if begin < end
    q = Partition(A, begin, end)
    QuickSort(A, begin, q − 1)
    QuickSort(A, q + 1, end)
```

- Best case

Complexity of QuickSort (2)

```
QUICKSORT(A, begin, end)
  if begin < end
      q = PARTITION(A, begin, end)
      QUICKSORT(A, begin, q − 1)
      QUICKSORT(A, q + 1, end)
```

- Best case
  - $q = \lceil n/2 \rceil$

Complexity of QUICKSORT (2)

```
QUICKSORT(A, begin, end)
if begin < end
        q = PARTITION(A, begin, end)
        QUICKSORT(A, begin, q − 1)
        QUICKSORT(A, q + 1, end)
```

- Best case
  - $q = \lceil n/2 \rceil$
  - the partition transforms P of size n into *two* problems P of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, respectively

Complexity of QUICKSORT (2)

```
QUICKSORT(A, begin, end)
if begin < end
    q = PARTITION(A, begin, end)
    QUICKSORT(A, begin, q − 1)
    QUICKSORT(A, q + 1, end)
```

- Best case
  - $q = \lceil n/2 \rceil$
  - the partition transforms P of size n into *two* problems P of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, respectively

$$T(n) = 2T(n/2) + \Theta(n)$$

```
QuickSort(A, begin, end)
if begin < end
    q = Partition(A, begin, end)
    QuickSort(A, begin, q − 1)
    QuickSort(A, q + 1, end)
```

- Best case
  - $q = \lceil n/2 \rceil$
  - the partition transforms P of size n into *two* problems P of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, respectively

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \log n)$$

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QuickSort				

Sorting Algorithms Seen So Far

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes

Sorting Algorithms Seen So Far

Algorithm	**Complexity**			**In place?**
	worst	*average*	*best*	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QUICKSORT	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes
??	$\Theta(n \log n)$			***yes***

- Our first real ***data structure***

- Our first real ***data structure***
- Interface

Binary Heap

- Our first real ***data structure***

- Interface
  - **BUILD-MAX-HEAP**(A) rearranges A into a max-heap
  - **HEAP-INSERT**(H, key) inserts *key* in the heap
  - **HEAP-EXTRACT-MAX**(H) extracts the maximum key
  - $H.heap\text{-}size$ is the number of keys in H

Binary Heap

- Our first real ***data structure***

- Interface
  - **Build-Max-Heap**(A) rearranges A into a max-heap
  - **Heap-Insert**(H, key) inserts *key* in the heap
  - **Heap-Extract-Max**(H) extracts the maximum key
  - $H.heap\text{-}size$ is the number of keys in H

- Two kinds of binary heaps

- Our first real ***data structure***

- Interface
  - **Build-Max-Heap**(A) rearranges A into a max-heap
  - **Heap-Insert**(H, key) inserts *key* in the heap
  - **Heap-Extract-Max**(H) extracts the maximum key
  - $H.heap\text{-}size$ is the number of keys in H

- Two kinds of binary heaps
  - max-heaps

- Our first real ***data structure***

- Interface
  - **Build-Max-Heap**(A) rearranges A into a max-heap
  - **Heap-Insert**(H, *key*) inserts *key* in the heap
  - **Heap-Extract-Max**(H) extracts the maximum key
  - $H.\mathit{heap\text{-}size}$ is the number of keys in H

- Two kinds of binary heaps
  - max-heaps
  - min-heaps

Binary Heap

- Our first real ***data structure***
- Interface
  - **BUILD-MAX-HEAP**(A) rearranges A into a max-heap
  - **HEAP-INSERT**(H, key) inserts *key* in the heap
  - **HEAP-EXTRACT-MAX**(H) extracts the maximum key
  - $H.\mathit{heap\text{-}size}$ is the number of keys in H
- Two kinds of binary heaps
  - max-heaps
  - min-heaps
- Useful applications

- Our first real ***data structure***

- Interface
  - **BUILD-MAX-HEAP**(A) rearranges A into a max-heap
  - **HEAP-INSERT**(H, key) inserts *key* in the heap
  - **HEAP-EXTRACT-MAX**(H) extracts the maximum key
  - $H.heap\text{-}size$ is the number of keys in H

- Two kinds of binary heaps
  - max-heaps
  - min-heaps

- Useful applications
  - sorting

Binary Heap

- Our first real ***data structure***

- Interface
  - **Build-Max-Heap**(A) rearranges A into a max-heap
  - **Heap-Insert**(H, key) inserts key in the heap
  - **Heap-Extract-Max**(H) extracts the maximum key
  - $H.heap\text{-}size$ is the number of keys in H

- Two kinds of binary heaps
  - max-heaps
  - min-heaps

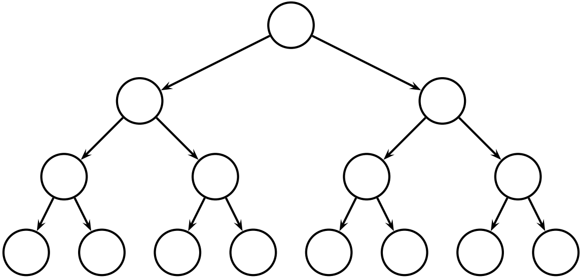
- Useful applications
  - sorting
  - priority queue

Binary Heap: Structure

Binary Heap: Structure

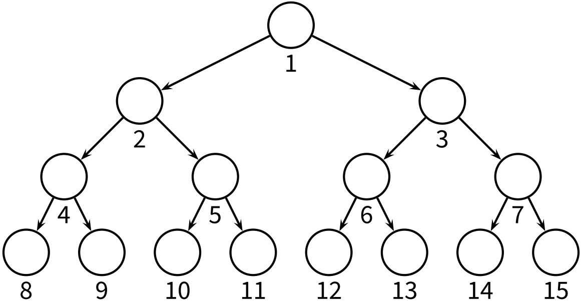
- Conceptually a full binary tree

- Conceptually a full binary tree



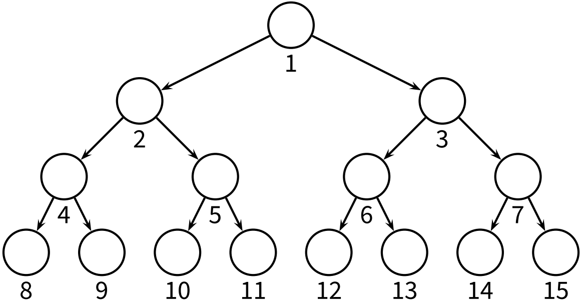
Binary Heap: Structure

- Conceptually a full binary tree



Binary Heap: Structure

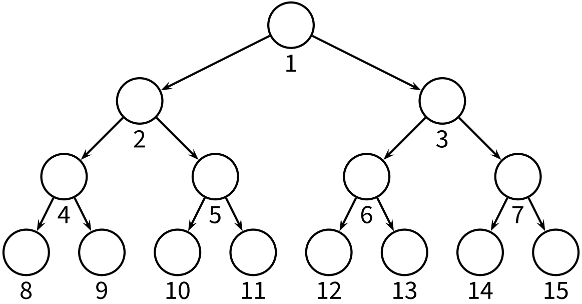
- Conceptually a full binary tree



- Implemented as an array

Binary Heap: Structure

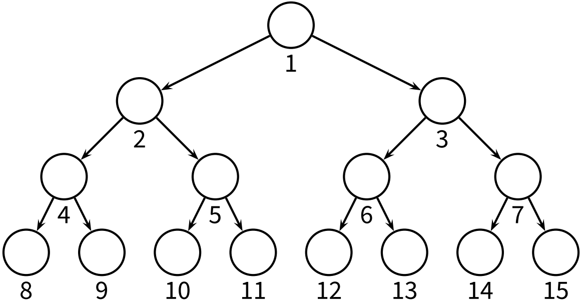
- Conceptually a full binary tree



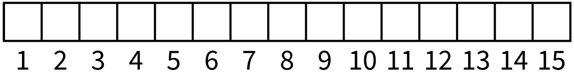
- Implemented as an array

Binary Heap: Structure

- Conceptually a full binary tree

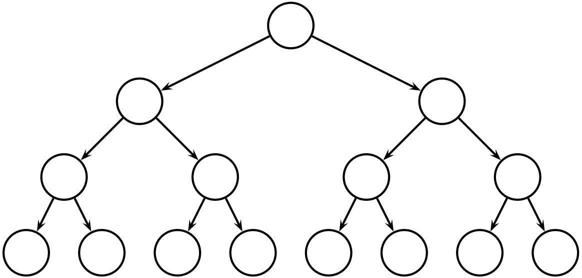


- Implemented as an array

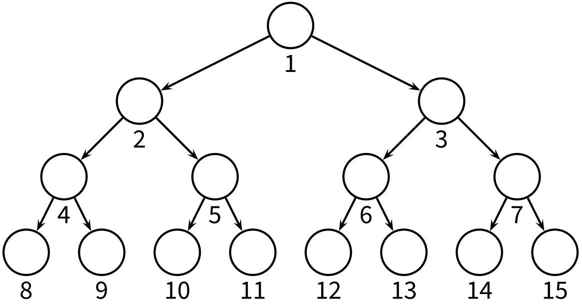


Binary Heap: Properties

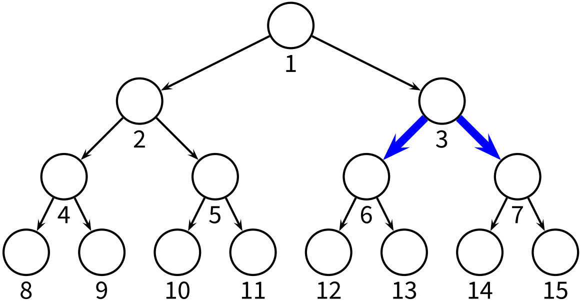
Binary Heap: Properties



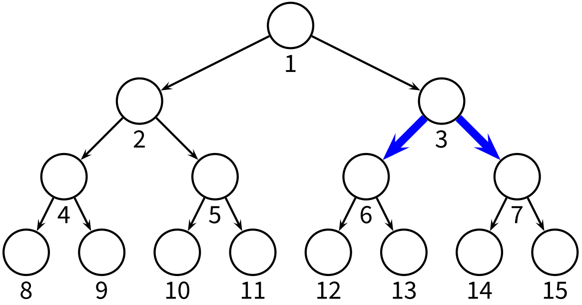
Binary Heap: Properties



Binary Heap: Properties



Binary Heap: Properties



Parent(i)

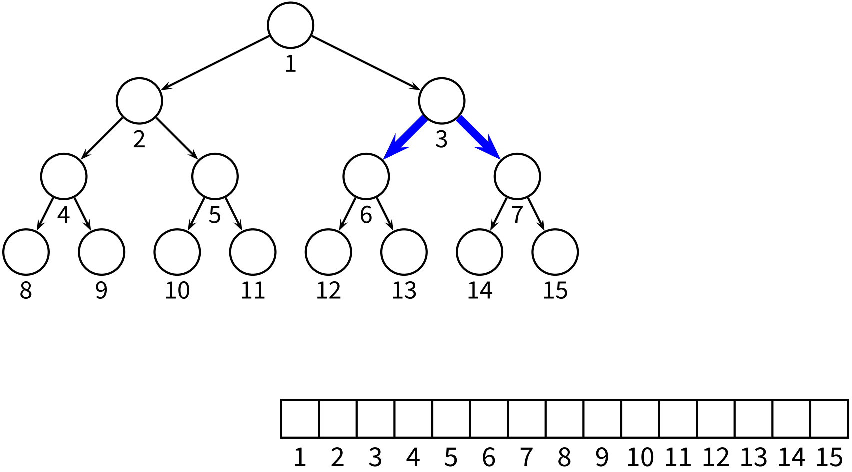
return $\lfloor i/2 \rfloor$

Left(i)

return $2i$

Right(i)

return $2i + 1$



Parent(i)

return $\lfloor i/2 \rfloor$

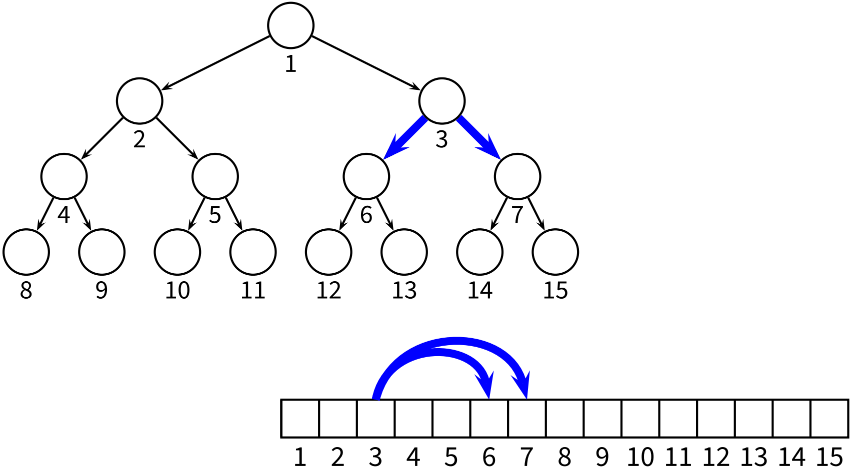
Left(i)

return $2i$

Right(i)

return $2i + 1$

Binary Heap: Properties



Parent(i)

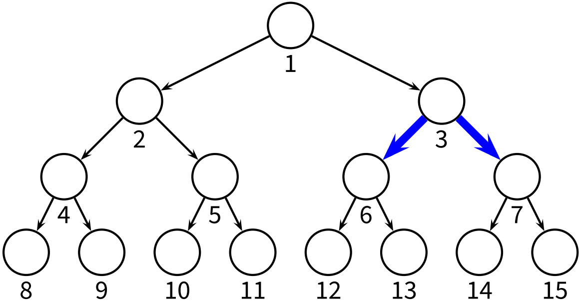
  return $\lfloor i/2 \rfloor$

Left(i)

  return $2i$

Right(i)

  return $2i + 1$



PARENT(i)

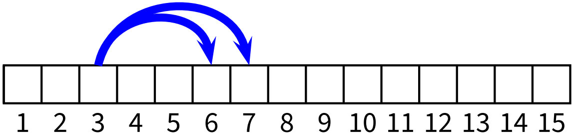
return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$



- ***Max-heap property:*** for all $i > 1$ $A[\text{PARENT}(i)] \geq A[i]$

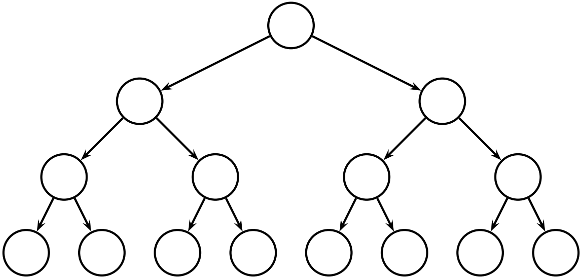
Example

- *Max-heap property:* for all $i > 1$ $A[\textbf{Parent}(i)] \geq A[i]$

Example

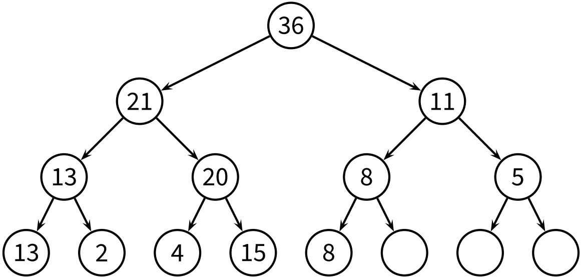
- *Max-heap property:* for all $i > 1$ $A[\textsc{Parent}(i)] \geq A[i]$

  E.g.,



- *Max-heap property:* for all $i > 1$ $A[\textbf{Parent}(i)] \geq A[i]$

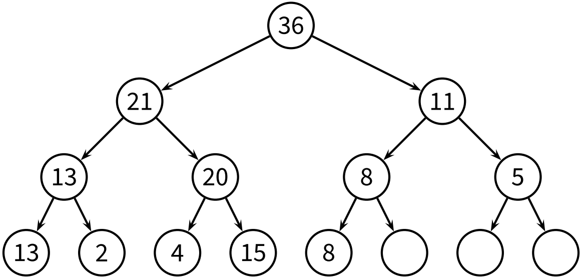
  E.g.,



Example

- *Max-heap property:* for all $i > 1$ $A[\textsc{Parent}(i)] \geq A[i]$

  E.g.,

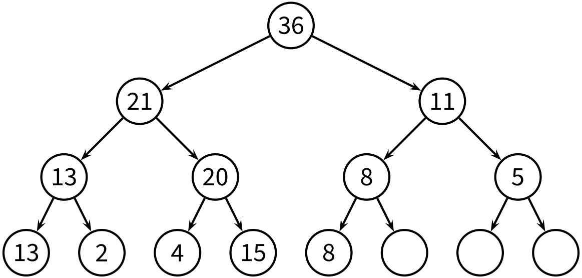


- Where is the max element?

Example

- *Max-heap property:* for all $i > 1$ $A[\textsc{Parent}(i)] \geq A[i]$

  E.g.,



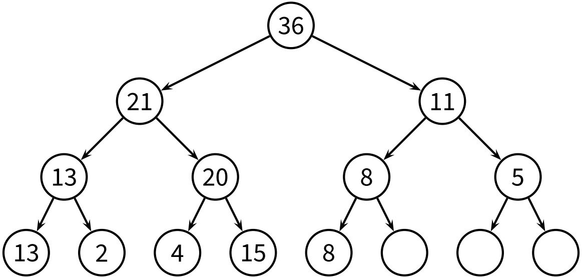
- Where is the max element?

- How can we implement **Heap-Extract-Max**?

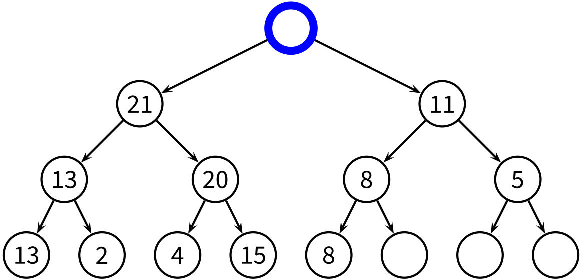
Heap-Extract-Max

- **Heap-Extract-Max** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*

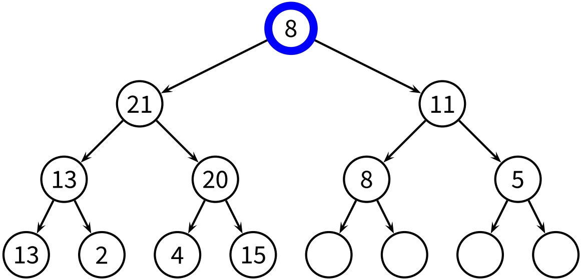
- **Heap-Extract-Max** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



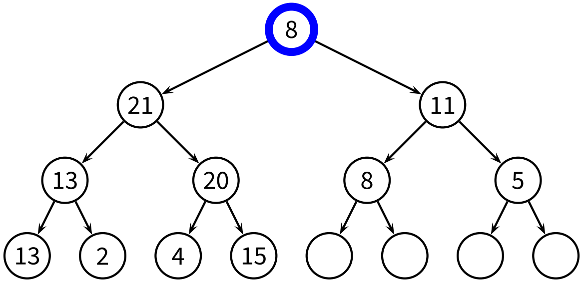
- **Heap-Extract-Max** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



- **Heap-Extract-Max** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



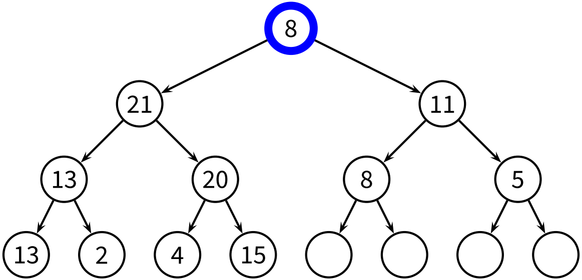
- **Heap-Extract-Max** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



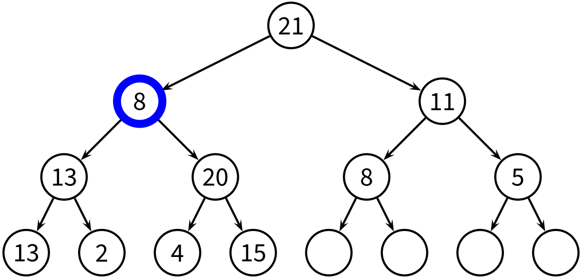
- Now we have two subtrees where the *max-heap property* holds

- **MAX-HEAPIFY**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node i
  - *goal:* rearrange the heap to maintain the *max-heap property*

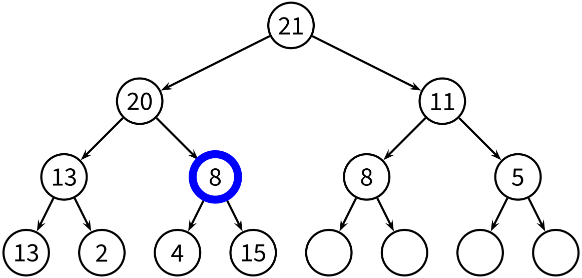
- **Max-Heapify**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node i
  - *goal:* rearrange the heap to maintain the *max-heap property*



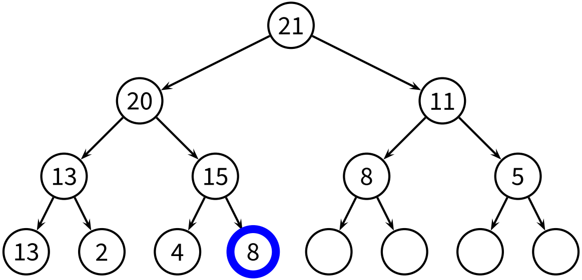
- **Max-Heapify**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node *i*
  - *goal:* rearrange the heap to maintain the *max-heap property*



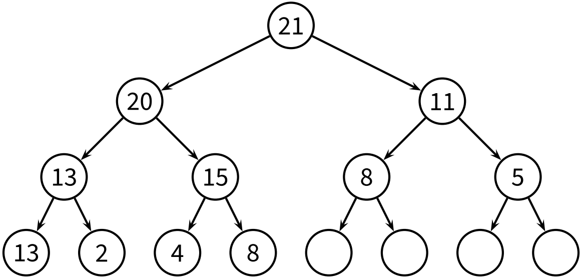
- **MAX-HEAPIFY**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node i
  - *goal:* rearrange the heap to maintain the *max-heap property*



- **Max-Heapify**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node *i*
  - *goal:* rearrange the heap to maintain the *max-heap property*



- **MAX-HEAPIFY**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node *i*
  - *goal:* rearrange the heap to maintain the *max-heap property*



Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        swap A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
```

Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        swap A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
```

- Complexity of **MAX-HEAPIFY**?

Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        swap A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
```

- Complexity of **MAX-HEAPIFY**? The height of the tree!

Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        swap A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
```

- Complexity of **MAX-HEAPIFY**? The height of the tree!

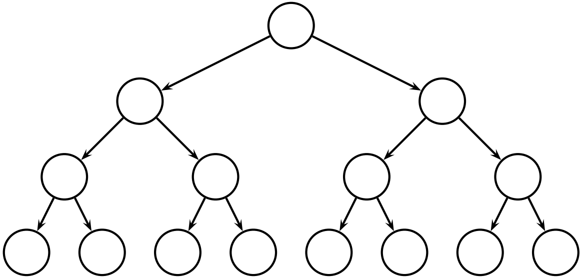
$$T(n) = \Theta(\log n)$$

Building a Heap

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

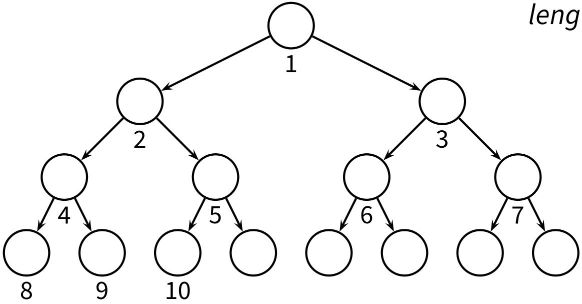
Building a Heap

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



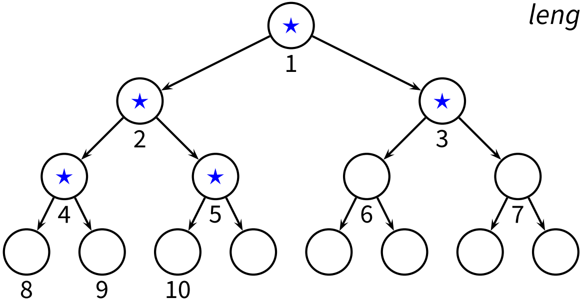
Building a Heap

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



$length(A) = 10$

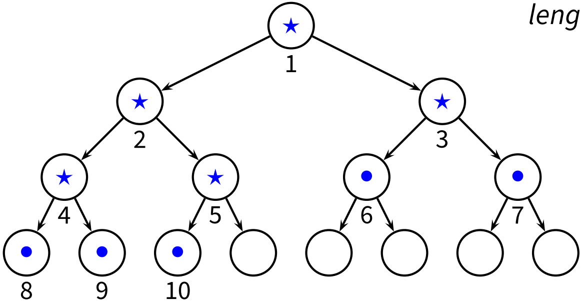
```
BUILD-MAX-HEAP(A)
1  A.heap-size = length(A)
2  for i = ⌊length(A)/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



$length(A) = 10$

Building a Heap

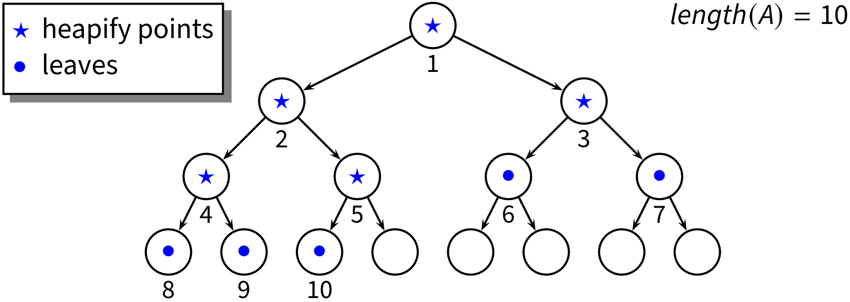
```
BUILD-MAX-HEAP(A)
  A.heap-size = length(A)
  for i = ⌊length(A)/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```



$length(A) = 10$

BUILD-MAX-HEAP(A)

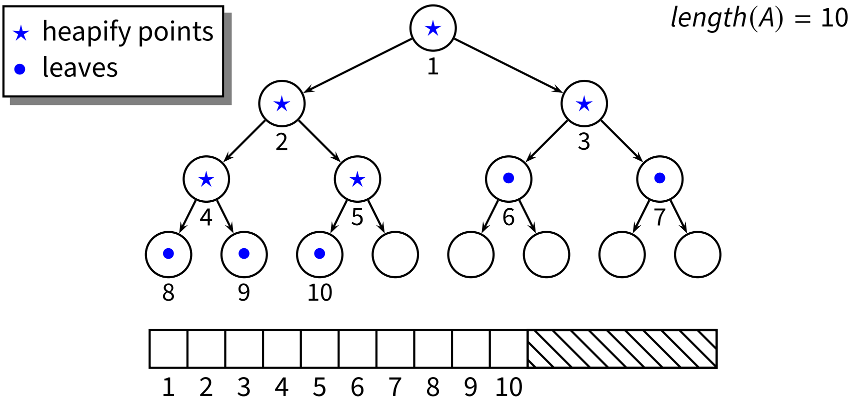
```
1  A.heap-size = length(A)
2  for i = ⌊length(A)/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Building a Heap

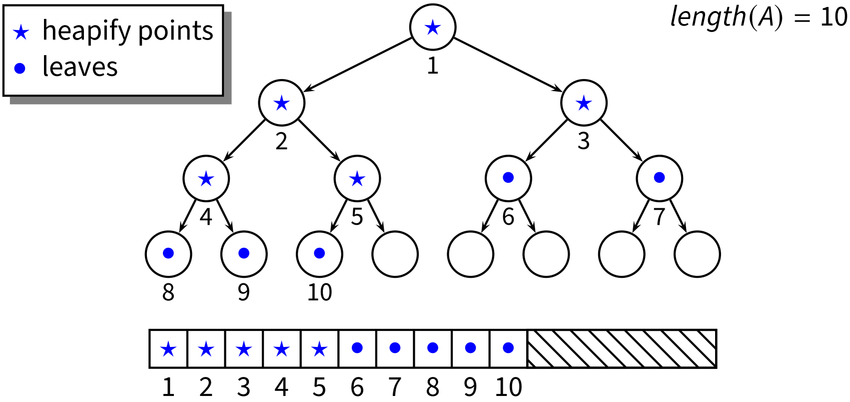
BUILD-MAX-HEAP(A)

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



Building a Heap

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



- Idea: we can use a heap to sort an array

- Idea: we can use a heap to sort an array

```
HEAP-SORT(A)
BUILD-MAX-HEAP(A)
for i = length(A) downto 1
    swap A[i] and A[1]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Heap Sort

- Idea: we can use a heap to sort an array

```
HEAP-SORT(A)
    BUILD-MAX-HEAP(A)
    for i = length(A) downto 1
        swap A[i] and A[1]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

- What is the complexity of **HEAP-SORT**?

Heap Sort

- Idea: we can use a heap to sort an array

```
HEAP-SORT(A)
BUILD-MAX-HEAP(A)
for i = length(A) downto 1
    swap A[i] and A[1]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

- What is the complexity of **HEAP-SORT**?

$$T(n) = \Theta(n \log n)$$

Heap Sort

- Idea: we can use a heap to sort an array

HEAP-SORT(A)

```
BUILD-MAX-HEAP(A)
for i = length(A) downto 1
        swap A[i] and A[1]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

- What is the complexity of **HEAP-SORT**?

$$T(n) = \Theta(n \log n)$$

- Benefits
  - in-place sorting; worst-case is $\Theta(n \log n)$

Summary of Sorting Algorithms

Algorithm	Complexity			In place?
	worst	*average*	*best*	
INSERTION-SORT				

Summary of Sorting Algorithms

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort				

Summary of Sorting Algorithms

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort				

Summary of Sorting Algorithms

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no

Algorithm	Complexity			In place?
	worst	*average*	*best*	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QUICK-SORT				

Algorithm	Complexity			In place?
	worst	*average*	*best*	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QUICK-SORT	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes

Summary of Sorting Algorithms

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
Quick-Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes
Heap-Sort				

Summary of Sorting Algorithms

Algorithm	Complexity			In place?
	worst	*average*	*best*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
Quick-Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes
Heap-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes