More on Sorting: Quick Sort and Heap Sort

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

March 14, 2017

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				

Sorting Algorithms Seen So Far

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				

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

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
BUBBLE-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
BUBBLE-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
BUBBLE-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
BUBBLE-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
BUBBLE-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 then* 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 then* v

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

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 then* v

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

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 then* 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 then* 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 then* v

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

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

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 then* 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 then* 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 $A[1 \ldots n]$ 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

2	36	5	21	8	13	11	20	5	4	1

Another Strategy for Sorting

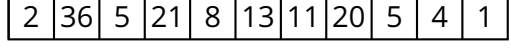
- *Problem:* sorting
- *Idea:* rearrange the sequence $A[1 \ldots n]$ in three parts based on a chosen "pivot" value $v \in A$
  - $A[1 \ldots q-1]$ contain elements that are *less than or equal to* v
  - $A[q] = v$
  - $A[q+1 \ldots n]$ contain elements that are *greater than* v

2	36	5	21	8	13	11	20	5	4	1

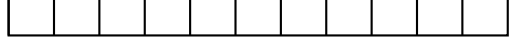
$v = 8$

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



$v = 8$



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

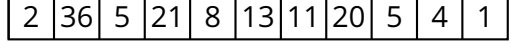
2	36	5	21	8	13	11	20	5	4	1

$v = 8$

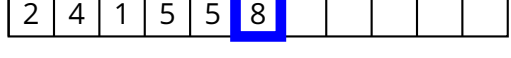
2	4	1	5	5						

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

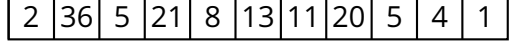


$v = 8$

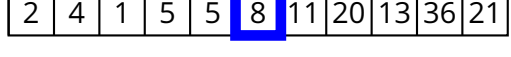


Another Strategy for Sorting

- *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

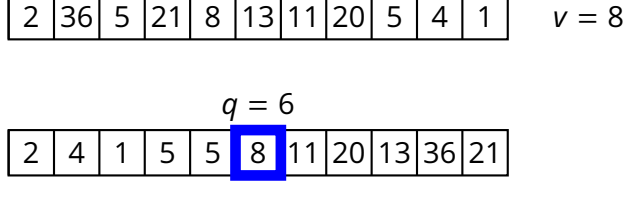


$v = 8$



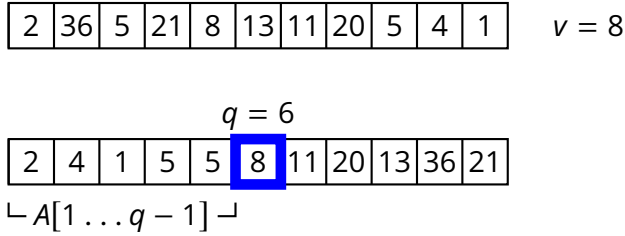
Another Strategy for Sorting

- *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



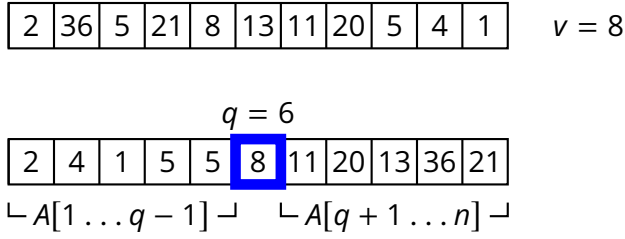
Another Strategy for Sorting

- *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



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

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:*** sort $A[1 \ldots q-1]$ and $A[q+1 \ldots n]$
- ***Combine:***

Another Divide-and-Conquer for Sorting

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

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

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

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

Partition

Partition

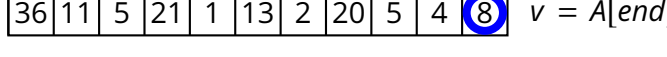
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right

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

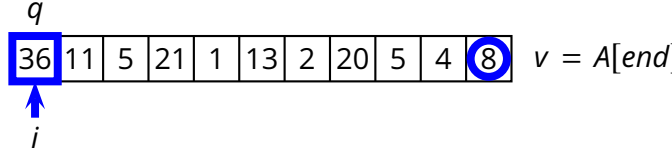
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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

36	11	5	21	1	13	2	20	5	4	8

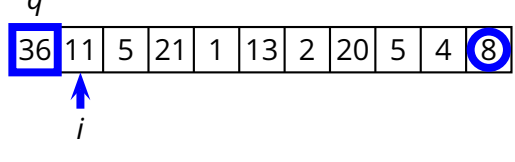
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



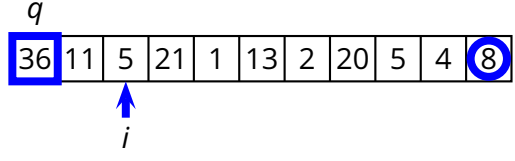
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



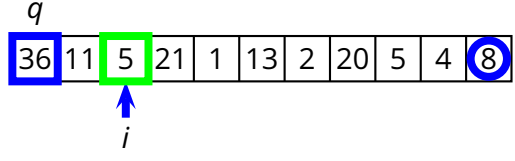
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



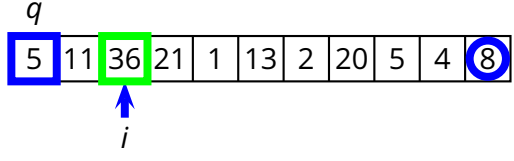
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



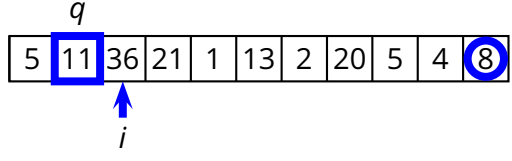
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



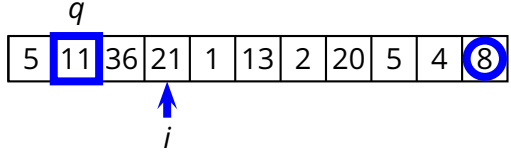
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



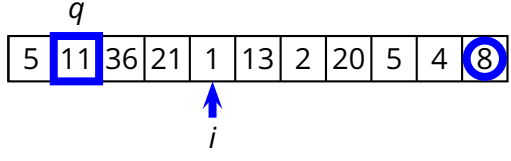
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



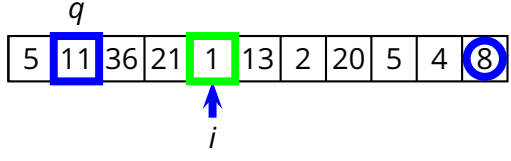
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



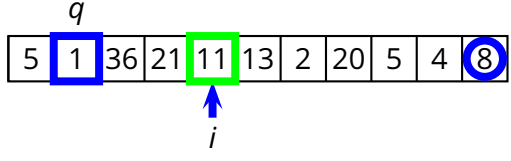
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



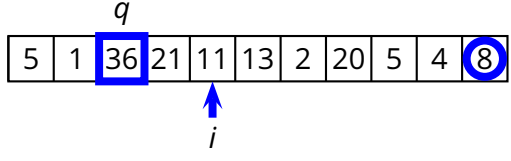
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



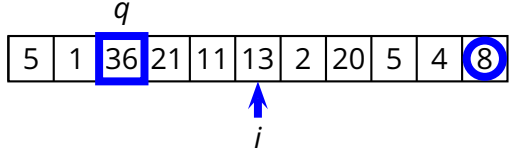
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



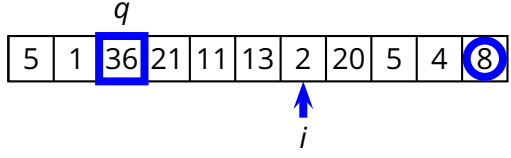
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



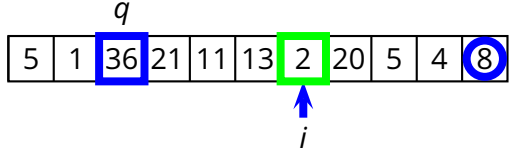
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



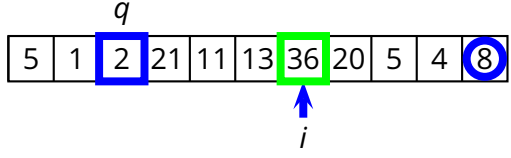
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



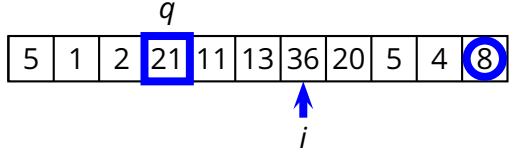
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



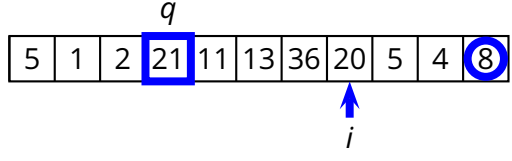
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



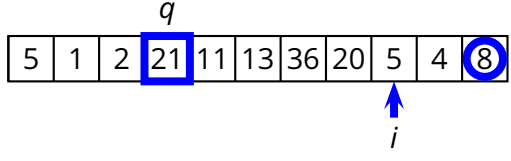
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



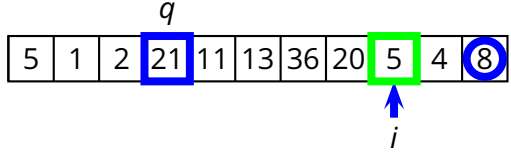
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



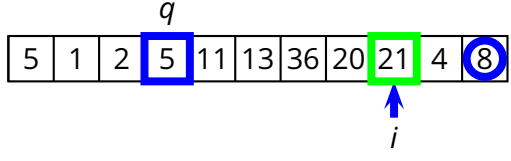
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



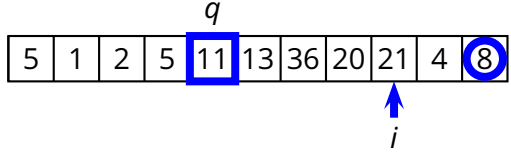
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



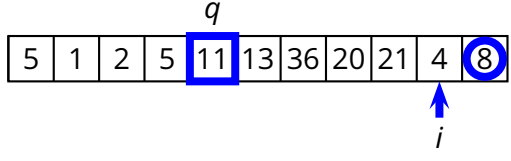
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



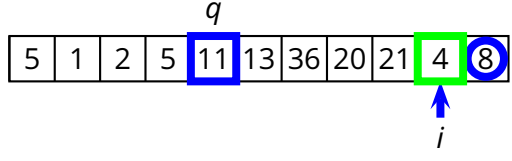
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



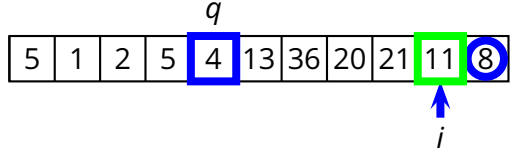
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



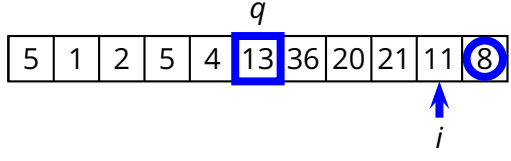
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



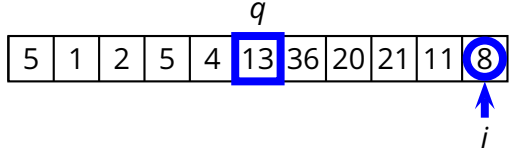
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



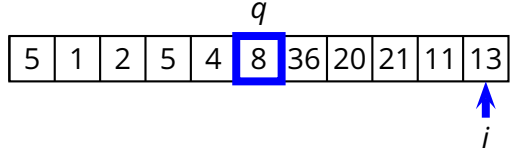
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



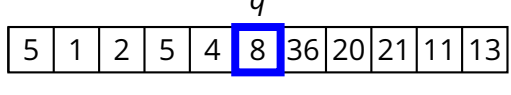
- Start with $q = 1$
  - i.e., *assume all elements are greater than the pivot*
- Scan the array left-to-right, starting at position 2
- If an element $A[i]$ is less than or equal to pivot, then swap it with the current q position and shift q to the right
- *Loop invariant*
  - $begin \leq k < q \Rightarrow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



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



- 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)
1  q = begin
2  v = A[end]
3  for i = begin to end
4       if A[i] ≤ v
5            swap A[i] and A[q]
6            q = q + 1
7  return q − 1
```

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

Complexity of PARTITION

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

Complexity of PARTITION

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

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

Complexity of QUICKSORT

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

Complexity of QUICKSORT

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

- Worst case

Complexity of QUICKSORT

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

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

Complexity of QUICKSORT

```
QUICKSORT(A, begin, end)
1  if begin < end
2        q = PARTITION(A, begin, end)
3        QUICKSORT(A, begin, q − 1)
4        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)
1  if begin < end
2        q = PARTITION(A, begin, end)
3        QUICKSORT(A, begin, q − 1)
4        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)
1  if begin < end
2        q = PARTITION(A, begin, end)
3        QUICKSORT(A, begin, q − 1)
4        QUICKSORT(A, q + 1, end)
```

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

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

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

Complexity of QUICKSORT (2)

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

Complexity of QUICKSORT (2)

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

- Best case

Complexity of QUICKSORT (2)

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

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

Complexity of QUICKSORT (2)

```
QUICKSORT(A, begin, end)
1  if begin < end
2        q = PARTITION(A, begin, end)
3        QUICKSORT(A, begin, q − 1)
4        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)
1  if begin < end
2        q = PARTITION(A, begin, end)
3        QUICKSORT(A, begin, q − 1)
4        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

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

Complexity of QUICKSORT (2)

```
QUICKSORT(A, begin, end)
1  if begin < end
2      q = PARTITION(A, begin, end)
3      QUICKSORT(A, begin, q − 1)
4      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

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
BUBBLE-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
BUBBLE-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
BUBBLE-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
BUBBLE-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***

Binary Heap

Binary Heap

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

Binary Heap

- 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-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-size* is the number of keys in H
- Two kinds of binary 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.*heap-size* is the number of keys in H
- Two kinds of binary heaps
  - max-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.*heap-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.*heap-size* is the number of keys in H
- Two kinds of binary heaps
  - max-heaps
  - min-heaps
- Useful applications

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-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-size* is the number of keys in H
- Two kinds of binary heaps
  - max-heaps
  - min-heaps
- Useful applications
  - sorting
  - priority queue

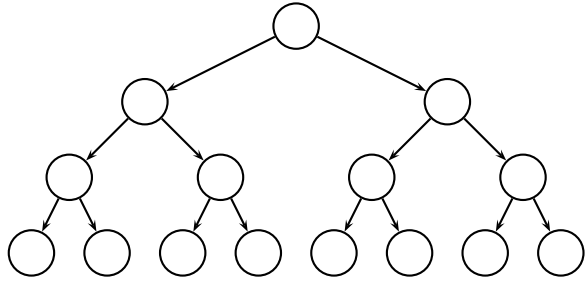
Binary Heap: Structure

Binary Heap: Structure

- Conceptually a full binary tree

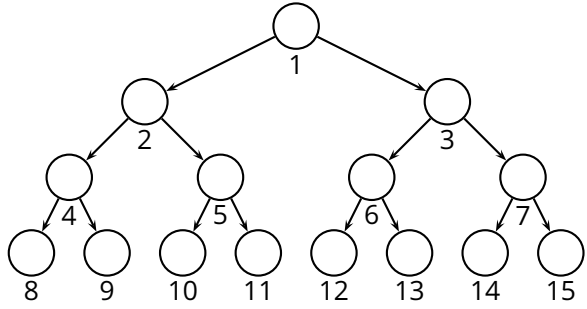
Binary Heap: Structure

- Conceptually a full binary tree



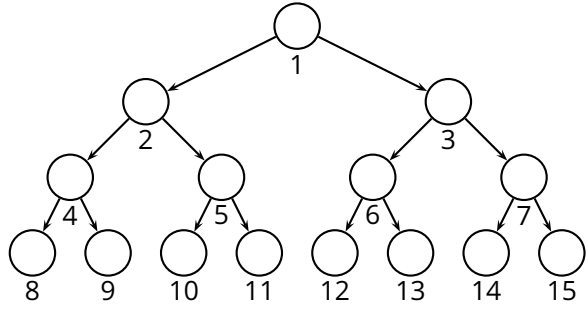
Binary Heap: Structure

- Conceptually a full binary tree



Binary Heap: Structure

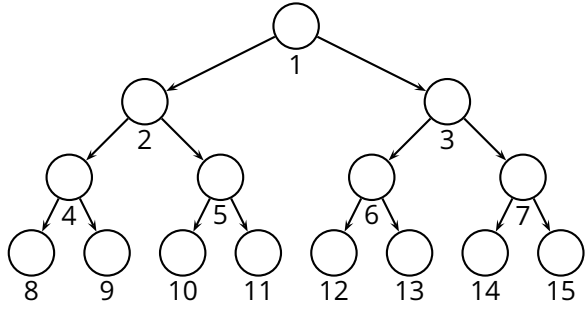
- Conceptually a full binary tree



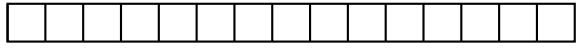
- Implemented as an array

Binary Heap: Structure

- Conceptually a full binary tree

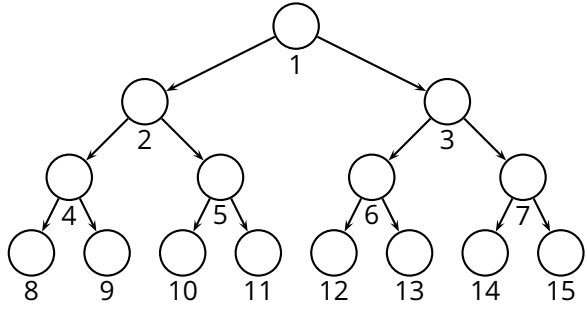


- Implemented as an array

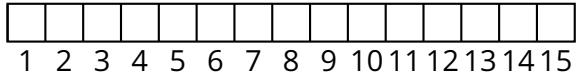


Binary Heap: Structure

- Conceptually a full binary tree

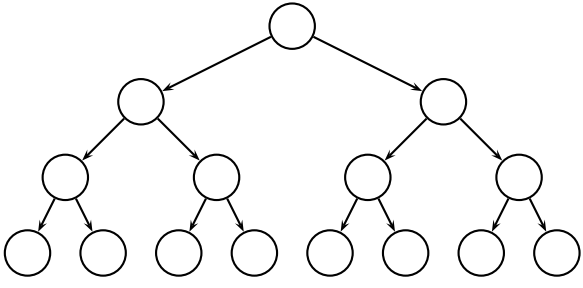


- Implemented as an array

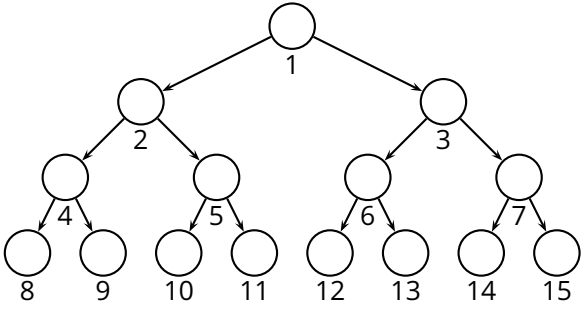


Binary Heap: Properties

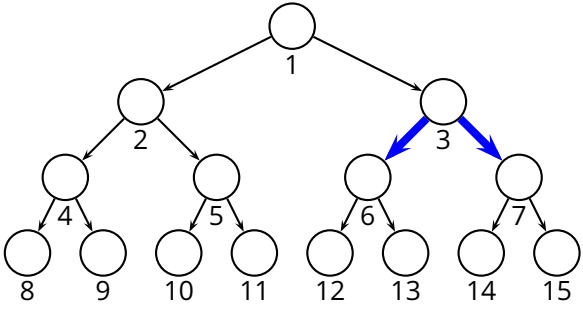
Binary Heap: Properties



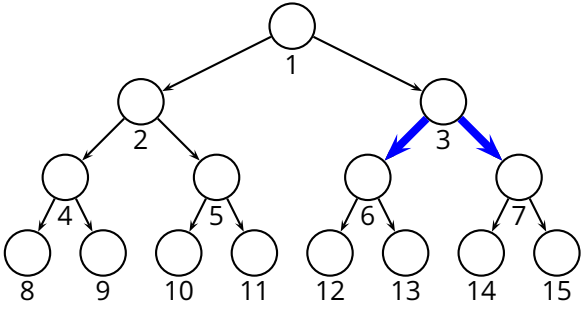
Binary Heap: Properties



Binary Heap: Properties

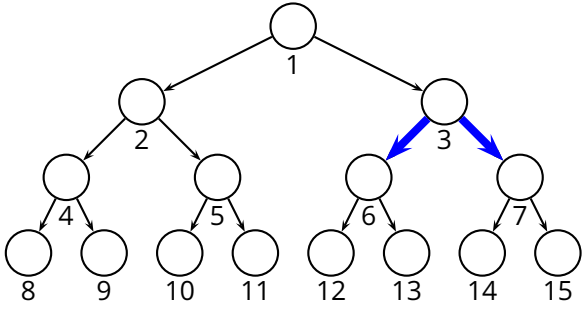


Binary Heap: Properties

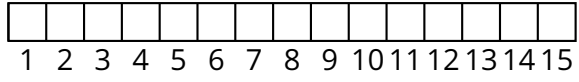


PARENT(i)
    return $\lfloor i/2 \rfloor$
LEFT(i)
    return $2i$
RIGHT(i)
    return $2i + 1$

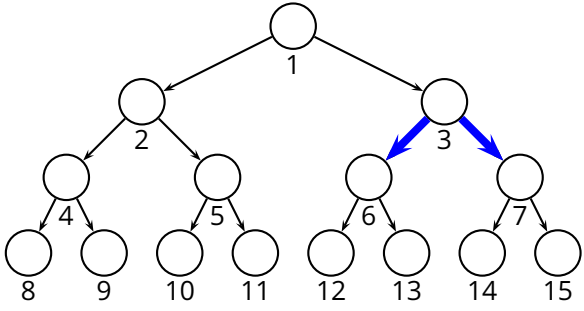
Binary Heap: Properties

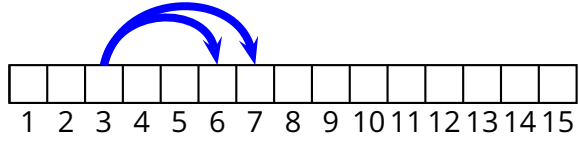


PARENT(i)
    return $\lfloor i/2 \rfloor$
LEFT(i)
    return $2i$
RIGHT(i)
    return $2i + 1$



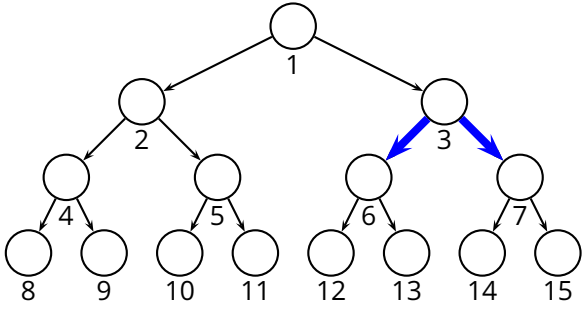
Binary Heap: Properties

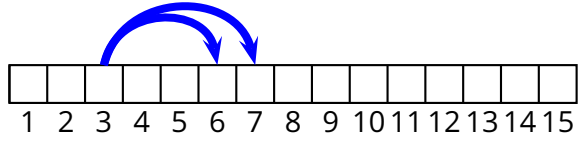




PARENT(i)
    return $\lfloor i/2 \rfloor$
LEFT(i)
    return $2i$
RIGHT(i)
    return $2i + 1$

Binary Heap: Properties





PARENT(i)
    return $\lfloor i/2 \rfloor$
LEFT(i)
    return $2i$
RIGHT(i)
    return $2i + 1$

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

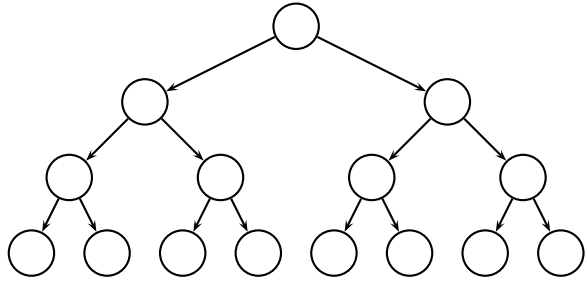
Example

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

Example

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

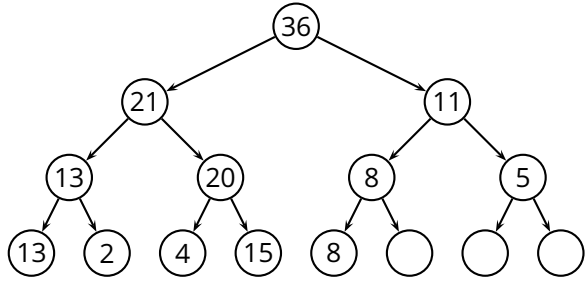
  E.g.,



Example

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

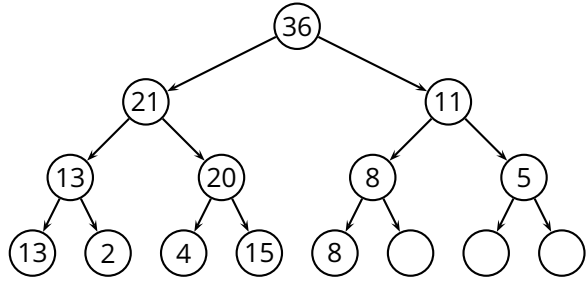
E.g.,



Example

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

  E.g.,

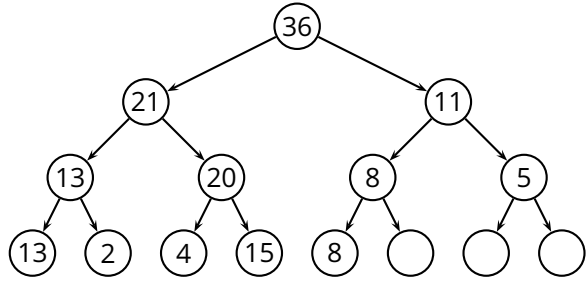


- Where is the max element?

Example

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

  E.g.,

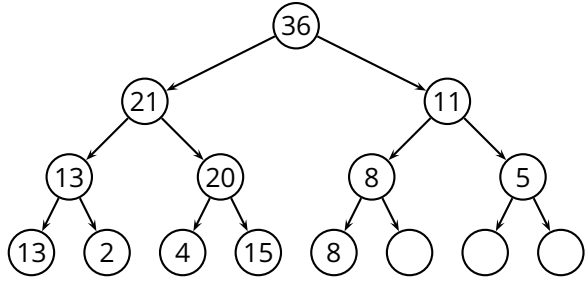


- Where is the max element?
- How can we implement **HEAP-EXTRACT-MAX**?

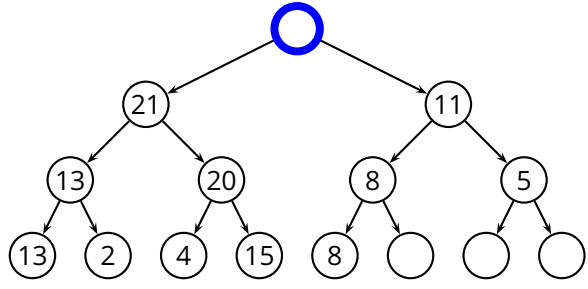
Heap-Extract-Max

- **HEAP-EXTRACT-MAX** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*

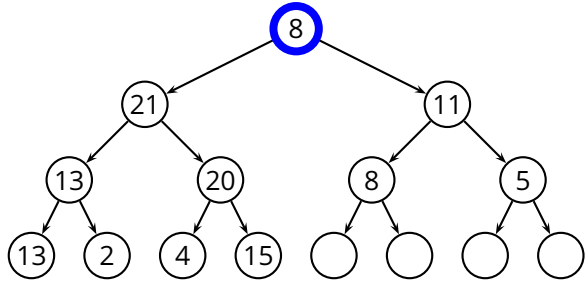
- **HEAP-EXTRACT-MAX** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



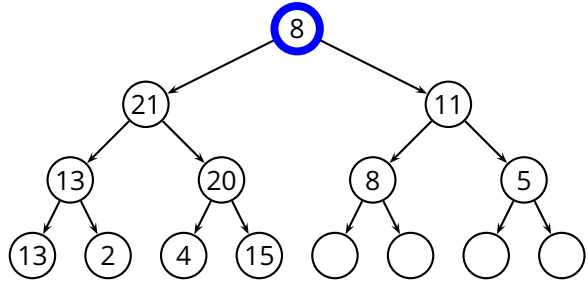
- **HEAP-EXTRACT-MAX** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



- **HEAP-EXTRACT-MAX** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



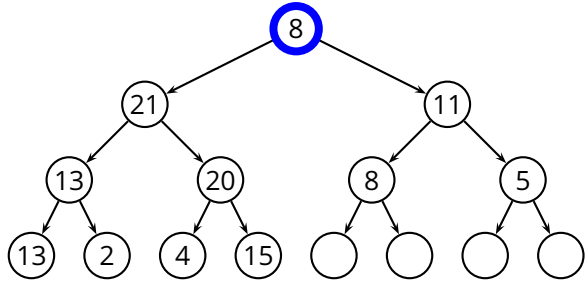
- **HEAP-EXTRACT-MAX** procedure
  - extract the max key
  - rearrange the heap to maintain the *max-heap property*



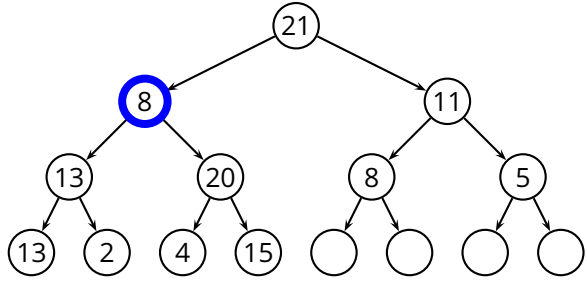
- Now we have two subtrees where the *max-heap property* holds

- **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*

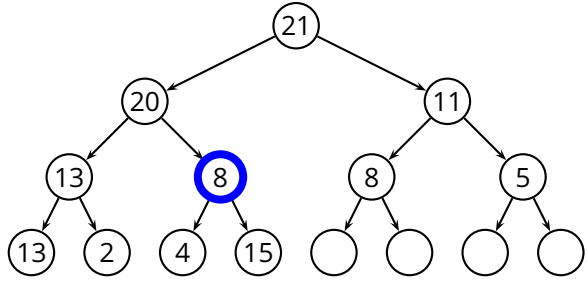
- **MAX-HEAPIFY**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node i
  - *goal:* rearrange the heap to maintain the *max-heap property*



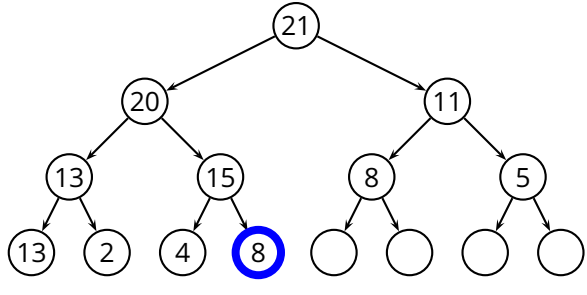
- **MAX-HEAPIFY**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node i
  - *goal:* rearrange the heap to maintain the *max-heap property*



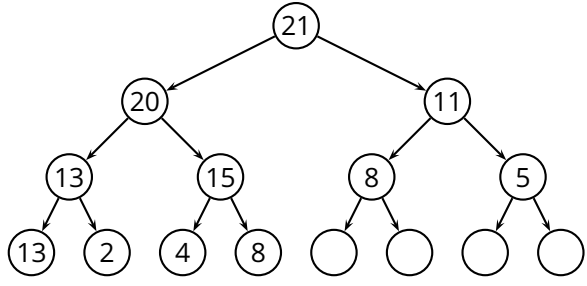
- **MAX-HEAPIFY**(A, i) procedure
  - *assume:* the *max-heap property* holds in the subtrees of node i
  - *goal:* rearrange the heap to maintain the *max-heap property*



- **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)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        MAX-HEAPIFY(A, largest)
```

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        MAX-HEAPIFY(A, largest)
```

- Complexity of **MAX-HEAPIFY**?

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        MAX-HEAPIFY(A, largest)
```

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

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        MAX-HEAPIFY(A, largest)
```

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

$$\boxed{T(n) = \Theta(\log n)}$$

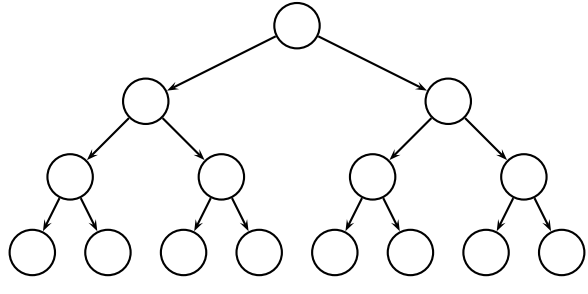
Building a Heap

Building a Heap

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

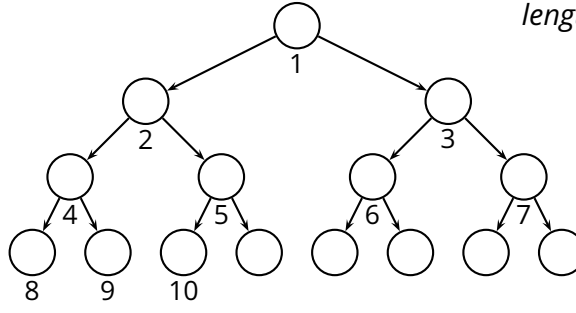
Building a Heap

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



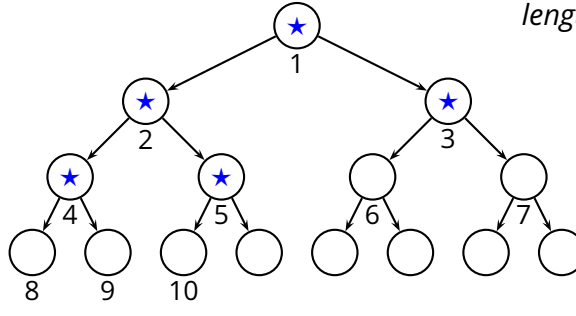
Building a Heap

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



Building a Heap

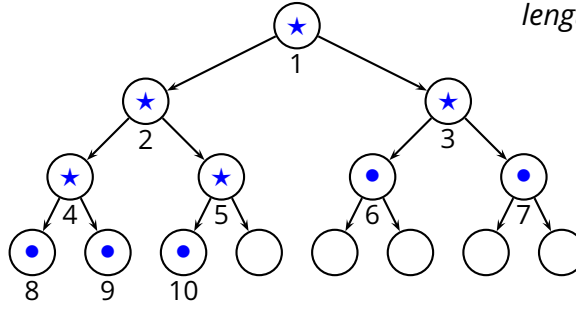
```
BUILD-MAX-HEAP(A)
1  A.heap-size = length(A)
2  for i = ⌊length(A)/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



$length(A) = 10$

Building a Heap

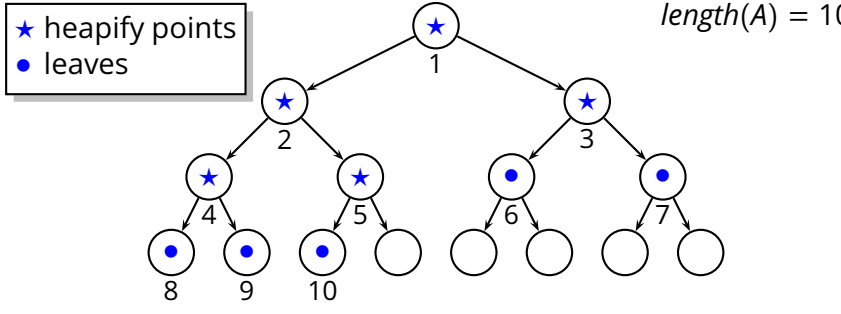
```
BUILD-MAX-HEAP(A)
1  A.heap-size = length(A)
2  for i = ⌊length(A)/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



$length(A) = 10$

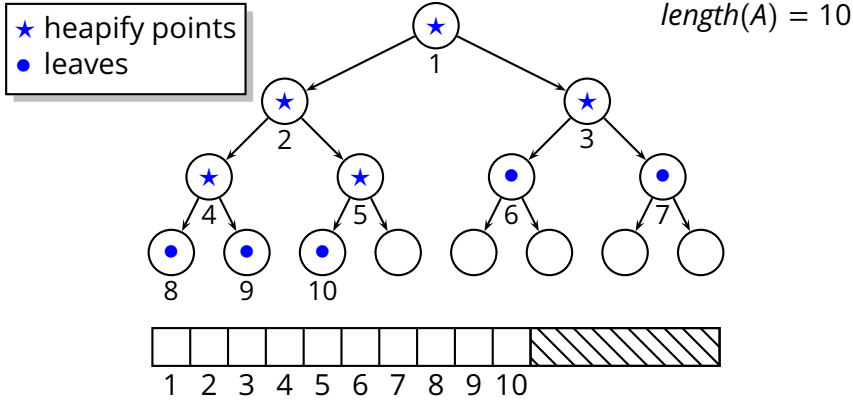
Building a Heap

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



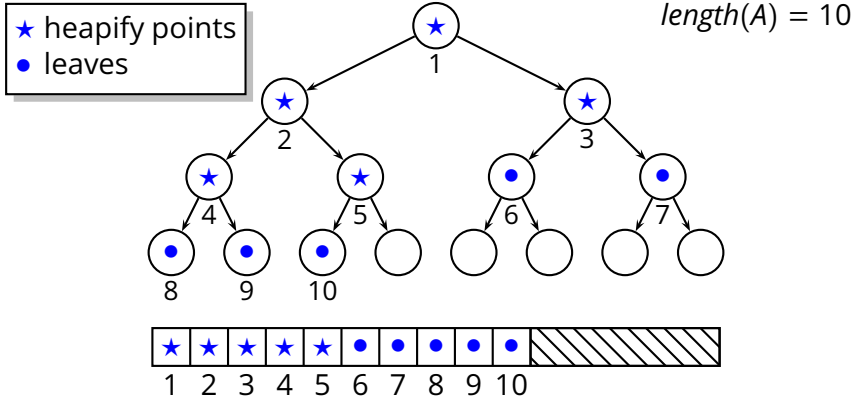
Building a Heap

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



Heap Sort

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

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

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

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

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

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

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

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

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

$$\boxed{T(n) = \Theta(n \log n)}$$

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

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

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

$$\boxed{T(n) = \Theta(n \log n)}$$

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

Summary of Sorting Algorithms

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
BUBBLE-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
BUBBLE-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
BUBBLE-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

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
BUBBLE-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				

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
BUBBLE-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
BUBBLE-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
BUBBLE-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