SFU CMPT-307 2008-2 Lecture: Week 3

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on May 20, 2008, 5.30pm-8.20pm

Important basic procedures for max-heaps

1. **Max-Heapify**, runs in time $O(\log n)$ time, a key procedure that maintains the *max-heap property* (if we change a value in the root of a heap, this procedure will correct the heap, so that it's again a heap)
2. **Build-Max-Heap**, runs in time $O(n)$, produces a max-heap from unsorted data
3. **Heap-Sort**, runs in time $O(n \log n)$, sorts array in place (uses above)

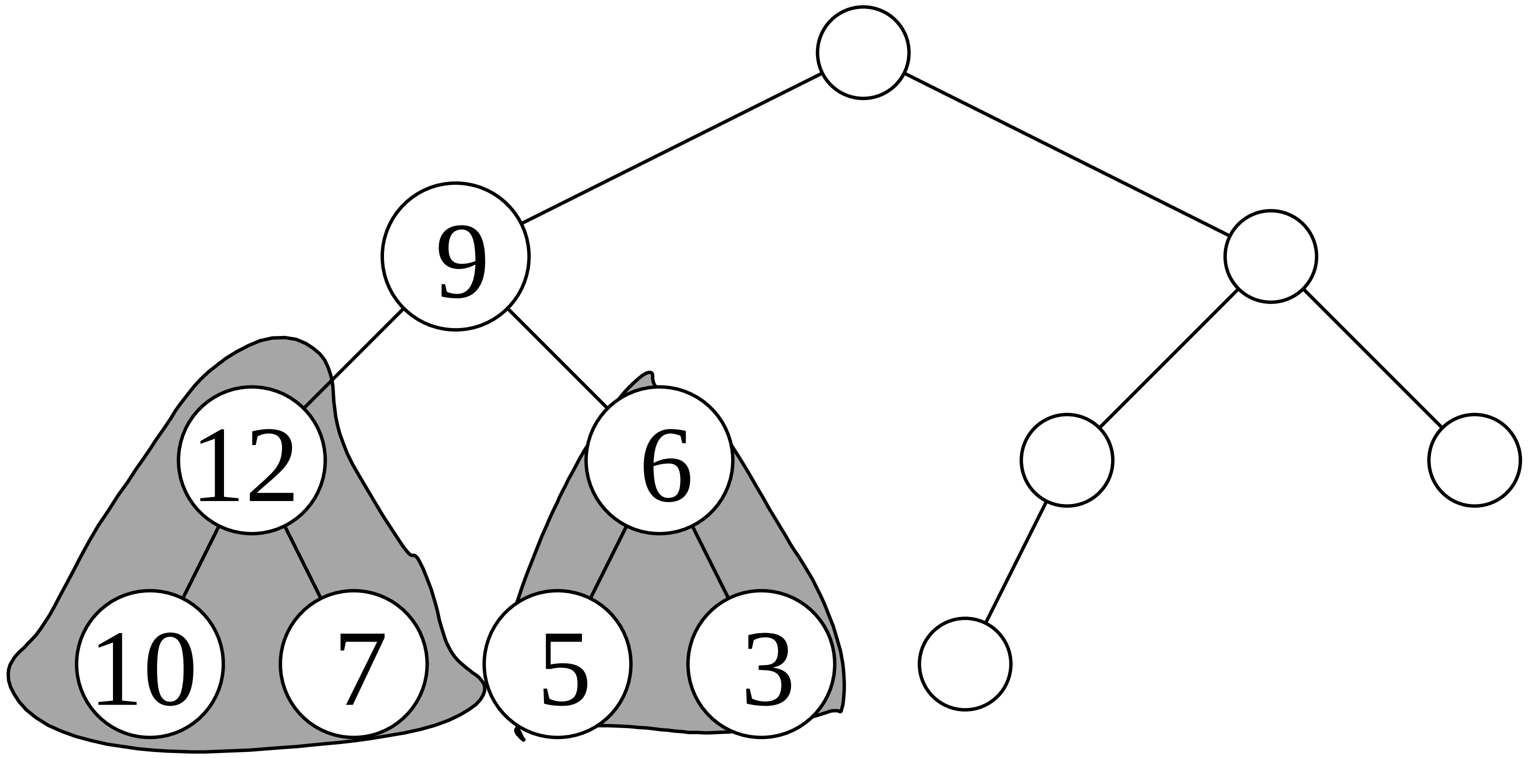
Also **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key** (run in time $O(\log n)$), **Heap-Maximum**, (run in time $O(1)$), used when the heap is used to implement a *priority queue*

Max-Heapify

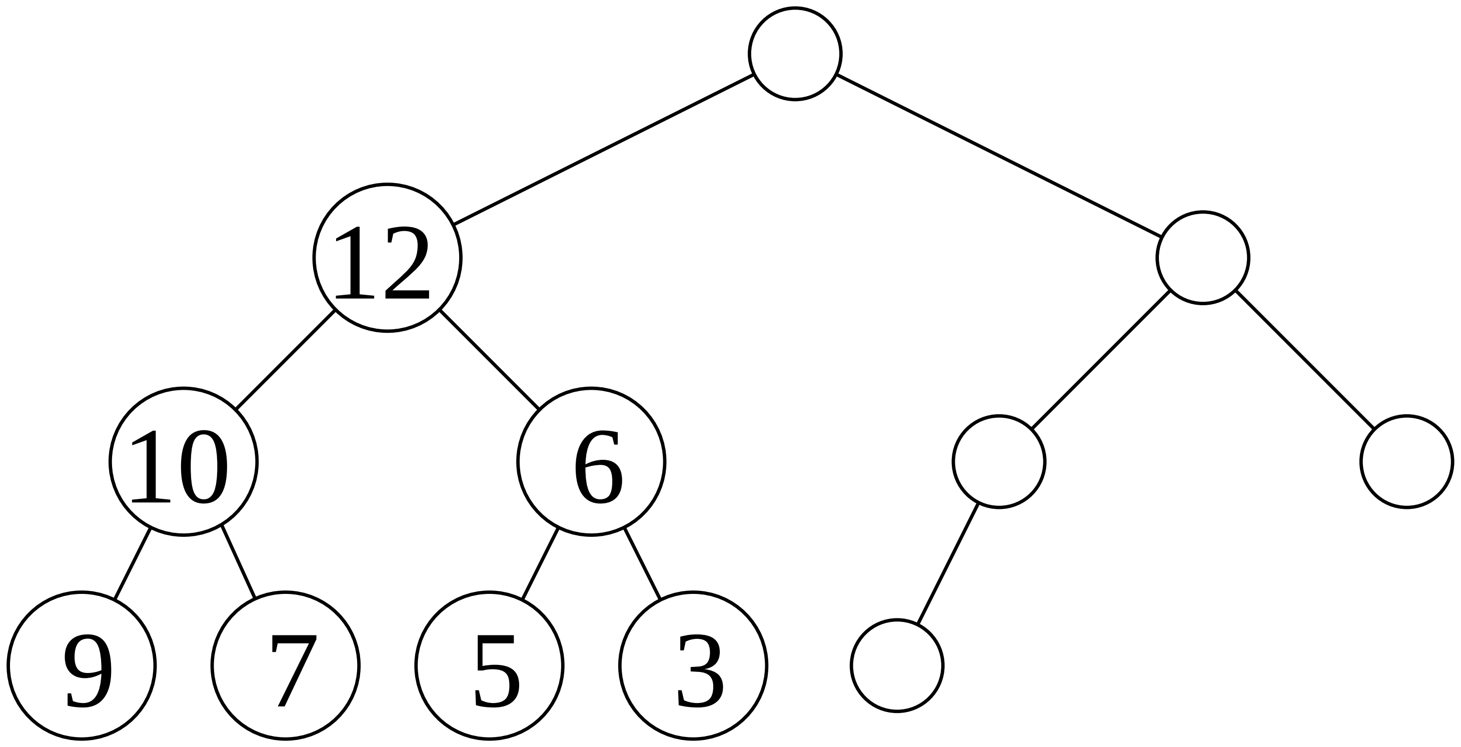
Maintains the max-heap property

Inputs are an array A and an index i

Assumption: sub-trees rooted in $\textsc{Left}(i)$ and $\textsc{Right}(i)$ are proper max-heaps, but $A[i]$ may be smaller than its children



Task of **Max-Heapify** is to let $A[i]$ float down in the max-heap below it so that heap rooted in i becomes proper max-heap



Max-Heapify(A, i)

```
1:  $\ell \leftarrow$ LEFT$(i)$
2:  $r \leftarrow$ RIGHT$(i)$

3:  **if** $\ell \leq$ heap-size$(A)$ and $A[\ell] > A[i]$ **then**
4:      largest $\leftarrow \ell$
5:  **else**
6:      largest $\leftarrow i$
7:  **end if**
8:  **if** $r \leq$ heap-size$(A)$ and $A[r] > A[$largest$]$ **then**
9:      largest $\leftarrow r$
10: **end if**

11: **if** largest $\neq i$ **then**
12:     exchange $A[i] \leftrightarrow A[$largest$]$
13:     **Max-Heapify**$(A,$ largest$)$
14: **end if**
```

Idea:

Lines 1–2 are just for convenience

Lines 3–10 find the largest of elements $A[i]$, $A[\ell]$, and $A[r]$

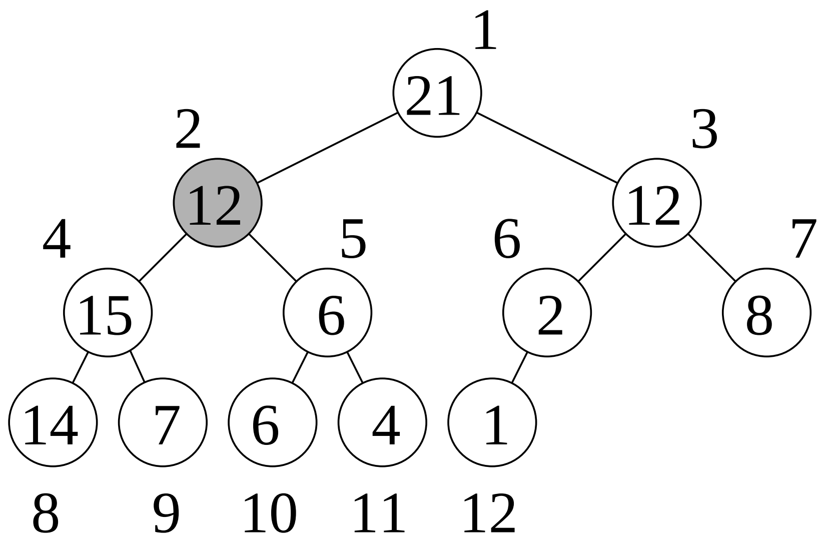
Lines 11–14

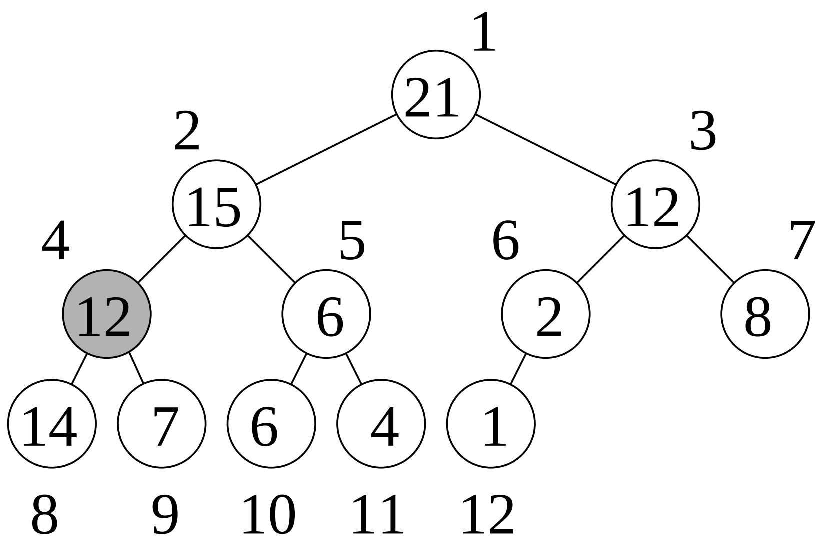
1. first check if there's anything to be done at all,
2. and if yes,
   (a) move the "misplaced element" one level down,
   (b) and make a recursive call one level deeper on this element

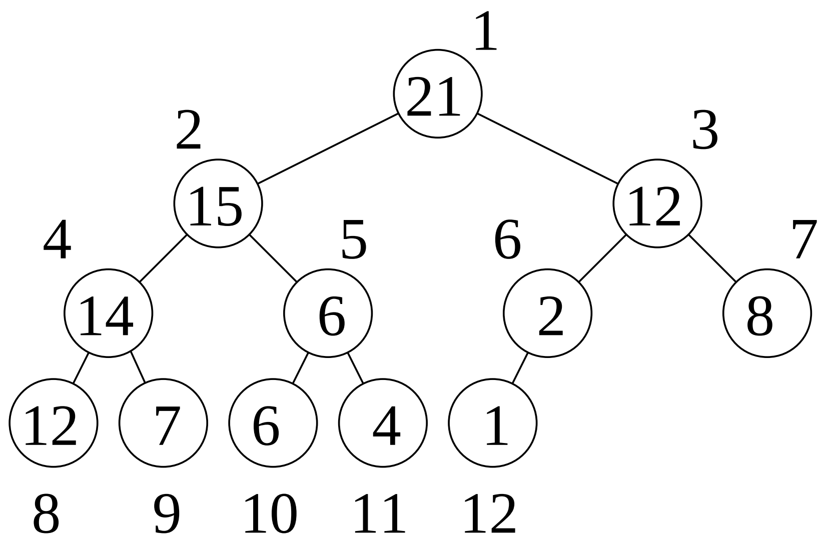
We know that after the exchange we have the largest of $A[i]$, $A[\ell]$, and $A[r]$ in position i, so among these three, everything is OK.

However, further down may still be problems.

Also note that we check $\ell \leq$ heap-size(A) and $r \leq$ heap-size(A), so that we don't go checking outside of the heap – the recursion will end if these tests fail


1
21
2
12
3
12
4
15
5
6
6
2
7
8
14
7
6
4
1
8
9
10
11
12


1
21
2
15
3
12
4
12
5
6
6
2
7
8
14
7
6
4
1
8
9
10
11
12


1
21
2
15
3
12
4
14
5
6
6
2
7
8
12
7
6
4
1
8
9
10
11
12

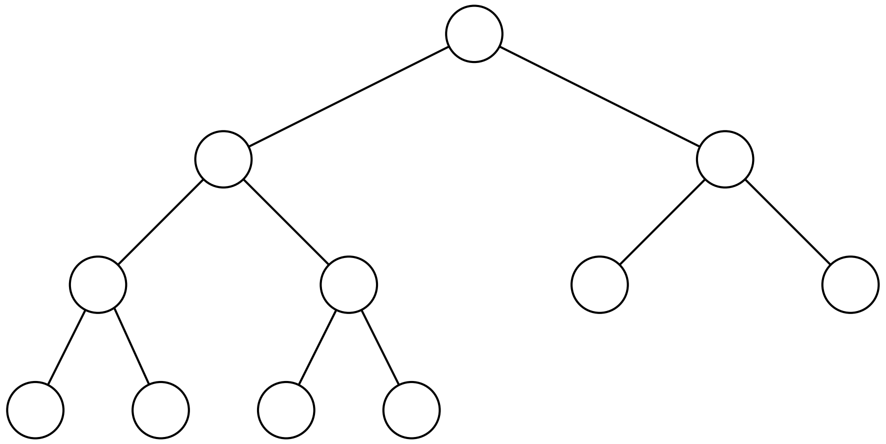
Running time of **Max-Heapify** on subtree of size n rooted at i is

- $\Theta(1)$ for finding largest and possibly swapping
- plus time to run **Max-Heapify** on sub-tree rooted in one of the children of i

Size of i's sub-tree in consideration is about
$(n-1)/2$ if complete trees

We allow for "nearly" complete trees, but here the size of any sub-tree is at most $\lceil 2n/3 \rceil$

This worst-case occurs if last level is exactly half full



$n = 11$: $2 \cdot 11/3 = 22/3 = 7.33\ldots$

This gives

$$T(n) \leq T(2n/3) + \Theta(1)$$

Case 2 of Master Theorem gives $T(n) = \Theta(\log n)$

Alternative proof: running time of **Max-Heapify** on node of height h (counted from bottom!) is $O(h)$ and $h = O(\log n)$ by Assignment Problem 2.5.

Building a heap

Easy using **Max-Heapify**

Suppose we have given an unordered array $A[1], \ldots, A[n]$ with $n = \text{length}(A)$

One can show that the elements

$$A[\lfloor n/2 \rfloor + 1],\ A[\lfloor n/2 \rfloor + 2],\ \ldots,\ A[n]$$

are the *leaves* of a heap..

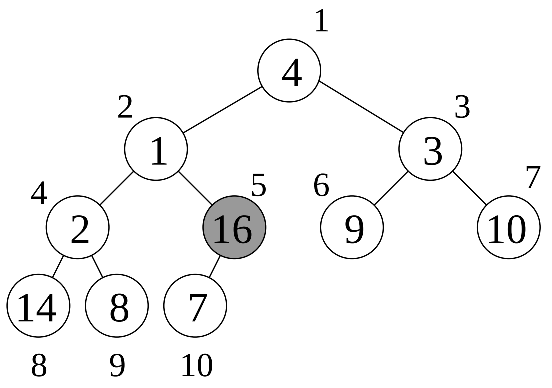
Thus, it's OK to initially consider them as 1-element heaps, and run **Max-Heapify** "on top" of them, once for each non-leaf element (1-element heaps are always proper heaps!).

Build-Max-Heap(A)

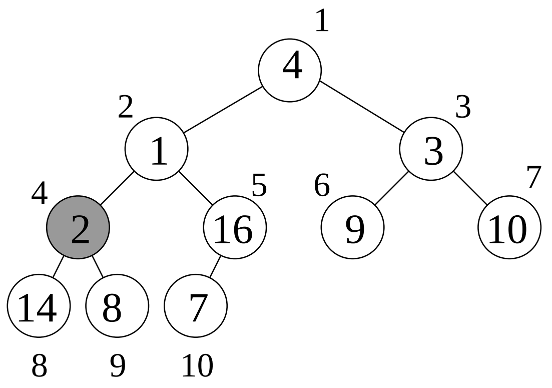
```
1: heap-size$(A) \leftarrow$ length$(A)$
2: for $i \leftarrow \lfloor \text{length}(A)/2 \rfloor$ downto $1$ do
3:     Max-Heapify$(A, i)$
4: end for
```

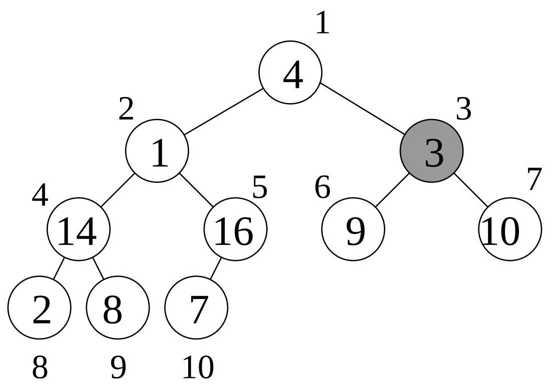
4	1	3	2	16	9	10	14	8	7

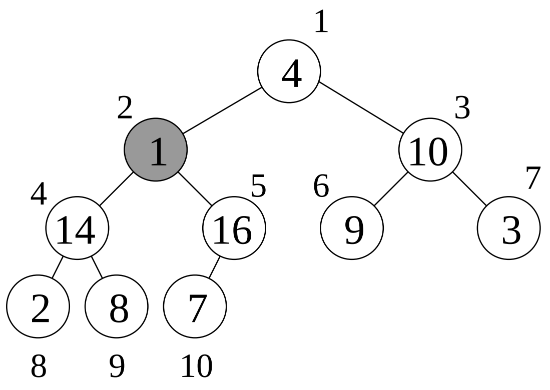
Input array A

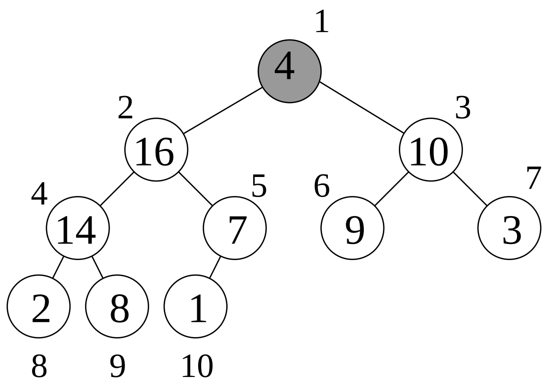


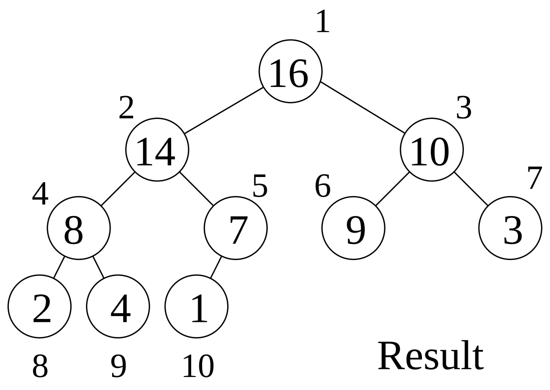
Binary tree representing A











Result

Build-Max-Heap Correctness

Loop invariant:

> At the start of each iteration, each node $i+1, i+2, \ldots, n$ is a root of a proper max-heap

Initialization: Initially, $i = \lfloor n/2 \rfloor$.
Nodes $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ are leaves.
Leaves are **always** roots of trivial max-heaps.

Maintenance: Observe, children nodes of i are numbered higher than i.
By invariant, both are roots of max-heaps.
Thus, we can call Max-Heapify(A, i). Now i is a root of max-heap.
Max-Heapify does not destroy the max-heap property of $i+1, i+2, \ldots, n$, thus they're still roots of max-heaps.
Decrementing i reestablishes invariant for next iteration.

Termination: Now $i = 0$, thus $1, 2, \ldots, n$ are roots of max-heaps (in particular, 1 is).

Build-Max-Heap Running time

Simple bound: calls to Max-Heapify cost $O(\log n)$, there are $O(n)$ of them, thus $O(n \log n)$.

Correct, but **not tight** (asymptotically).

Truth is $\Theta(n)$.

First observation: time for Max-Heapify depends on height of node (clearly).

Second observation: n-element heap has height $\lfloor \log n \rfloor$ (Problem 2.5).

Third observation: it has at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (Problem 3.1).

Assignment Problem 3.1. (deadline: May 27, 5:30pm)
Prove by mathematical induction on h that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

Hint: For the induction step, your induction hypothesis is

- For any n, n-element heap has at most $\lceil n/2^{h} \rceil$ nodes of height $h-1$.

and you want to prove that

- For any n, n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

Hence, if you are proving the claim for a heap H with n elements, then you can apply the induction hypothesis on any heap (for instance on a heap which contains only a part of H).

We know: Time required by Max-Heapify on node of height h is $O(h)$.

Thus, running time of Build-Max-Heap is upper-bounded by

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

By formula

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

and substituting $x = 1/2$ we obtain

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

and thus

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

Hence, Build-Max-Heap runs in linear time.

Heapsort

Now it's very easy to write down the algorithm for Heapsort.

Idea as follows:

1. Given unsorted array A, build heap on A, using Build-Max-Heap
2. Extract largest element (is in $A[1]$), and move it to the end of array (swap $A[1]$ and $A[n]$)
3. Decrease the size of the heap by 1
4. The root might not satisfy the heap property (that's where the element formerly in $A[n]$ now is), hence, using Max-Heapify, correct the heap
5. Extract the 2nd-largest element (again, in $A[1]$), and so on…

Heapsort(A)

```
1: Build-Max-Heap$(A)$
2: **for** $i \leftarrow$ length$(A)$ **downto** $2$ **do**
3:     exchange $A[1] \leftrightarrow A[i]$
4:     heap-size$(A) \leftarrow$ heap-size$(A) - 1$
5:     Max-Heapify$(A, 1)$
6: **end for**
```

Running time: $O(n \log n)$ (Build-Max takes $O(n)$, and then $O(n)$ rounds with $O(\log n)$ each).

Correctness

Loop invariant:

At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1 \ldots i]$ is a max-heap containing the i smallest elements of $A[1 \ldots n]$, and the subarray $A[i + 1 \ldots n]$ contains the $n - i$ largest elements in sorted order.

Initialization: After running **Build-Max-Heap**, the elements $A[1 \dots n]$ form a max-heap. Since, innitially $i = n$, the first subarray is $A[1 \dots n]$ (a max-heap), while the second is empty, and so satisfies all conditions trivially.

Maintenance: Since $A[1]$ contains the maximal element of $A[1 \dots i]$, but is smaller than any of the elements in $A[i + 1 \dots n]$, after swapping $A[1]$ and $A[i]$:

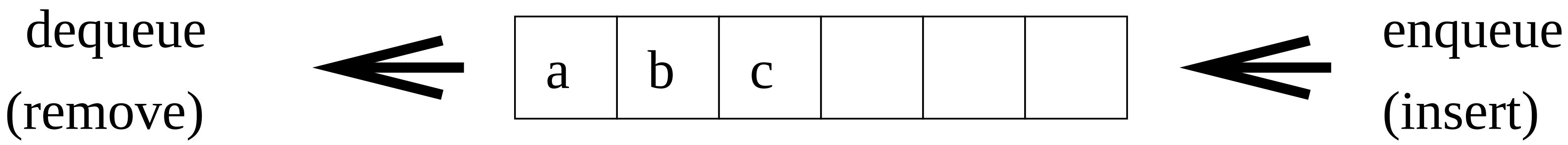
- $A[i \dots n]$ is sorted (since $A[i + 1 \dots n]$ was sorted)
- elements in $A[1 \dots i - 1]$ are smaller than elements in $A[i \dots n]$
- $A[1 \dots i - 1]$ is a max-heap except for the root

Lines 4-5 correct the max-heap property in the root, hence, in the end of the iteration $A[1 \dots i - 1]$ is a proper max-heap.

Termination: in the end, we have that the loop invariant is satisfied for $i = 1$: elements $A[2 \dots n]$ are sorted and greater than $A[1]$… done

(Simple) Queues

What is a queue?



We **insert** new elements at the **tail**, and **remove** elements from the **head**.

The standard queue is a **First-In-First-Out** (FIFO) buffer.

Many applications, for instance CD burning:

- **reader process** reads files from a hard disk and inserts data into a queue
- **writer process** reads from the queue and writes onto CD

Priority queues

They are different: there's no "real" FIFO rule anymore.

A **priority queue** maintains set S of elements, each with a **key** (priority).

Two kinds: **max-priority queues** and **min-priority queues**, usually implemented by **max-heaps** and **min-heaps**.

Max-priority queue

Operations:

- **Insert**(S, x) inserts element x into set S
- **Maximum**(S) returns element of S with largest key
- **Extract-Max**(S) removes and returns element of S with largest key
- **Increase-Key**(S, x, k) increases x's key to new value k, assuming k is at least as large as x's old key

Min-priority queue is used via operations: **Insert**, **Minimum**, **Extract-Min**, and **Decrease-Key**.

What are they good for?

Example for **Max-priority queue**:

job scheduling on a shared computer

- jobs with priorities, can be stored in a max-priority queue
- each time a new job is to be scheduled, it's got to be one of highest priority (**Extract-Max** operation)
- new jobs can be inserted using **Insert** operation
- in order to avoid "starvation", priorities can be increased (**Increase-Key** operation)

Example for **Min-priority queue**:

event-driven simulator

- items in the queue are events to be simulated, and the key is the time of the event when it should occur (happen)
- events are simulated in time order, hence we always extract the event with the smallest time from the queue and simulate it
- new events are procuded, so they have to be inserted in the queue

Heaps are very convenient here:

- using max-heaps, we know that the largest element is in $A[1]$: we have $O(1)$ access to largest element
- removing/inserting elements and increasing keys means that we (basically) can call **Max-Heapify** or a similar procedure (fixing the heap from bottom up) at the right place (relatively efficient operation $\mathcal{O}(\log n)$)

Min-priority queues analogous.

Implementation

We're considering **max-priority queues**, implemented using **max-heaps**

Let's start with a simple operation.

```
Heap-Maximum($A$)
1: return $A[1]$
```

implements **Maximum** operation in $O(1)$ time.

Extract max

Now, how to we **remove** the largest element from the queue/heap so that we will still have a proper max-heap?

Heap-Extract-Max(A)

```
1: **if** heap-size$(A) < 1$ **then**
2:     **error** “heap underflow”
3: **end if**
4: max $\leftarrow A[1]$
5: $A[1] \leftarrow A[\text{heap-size}(A)]$
6: heap-size$(A) \leftarrow$ heap-size$(A) - 1$
7: **Max-Heapify**$(A, 1)$
8: **return** max
```

implements **Extract-Max**

Running time is $O(\log n)$

Increase key

Suppose that the element which key is to be increased is identified by index i

We first update the key $A[i]$

Clearly, this can destroy the max-heap property, thus we need to find a new place for this element

The idea is to move the updated element as far up toward the root as necessary

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

```
1: if key$< A[i]$ then
2:     error “new key is smaller than current key”
3: end if
4: $A[i] \leftarrow$ key
5: while $i > 1$ and $A[\text{Parent}(i)] < A[i]$ do
6:     exchange $A[i] \leftrightarrow A[\text{Parent}(i)]$
7:     $i \leftarrow \text{Parent}(i)$
8: end while
```

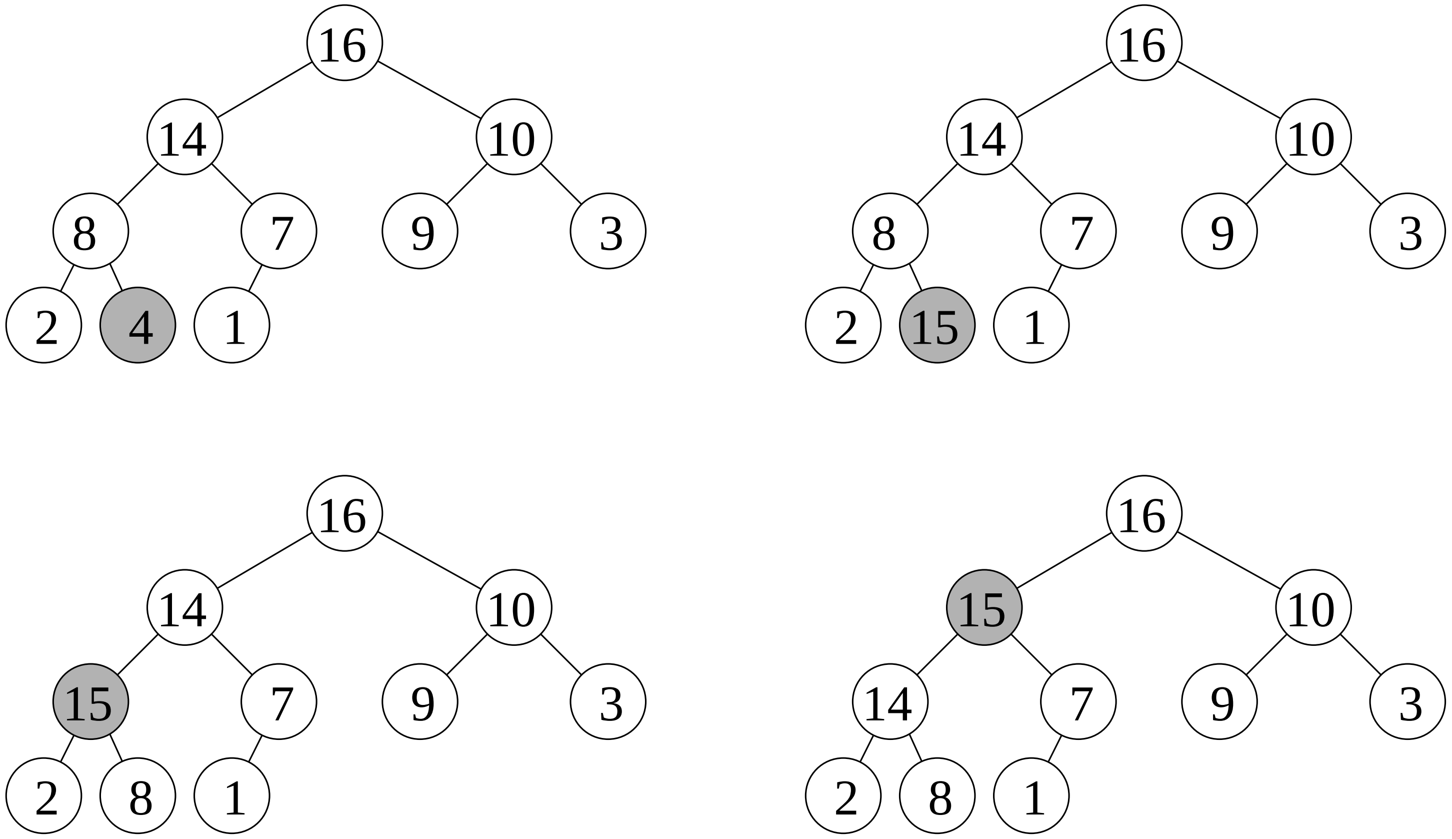
Running time is $O(\log n)$ (height of tree)

Example:

Heap-Increase-Key$(A, 9, 15)$

The node with index 9:

update the key from 4 to 15



Assignment Problem 3.2. (deadline: May 27, 5:30pm)
Argue the correctness of **Heap-Increase-Key** using the following loop invariant

- At the start of each iteration of the **while** loop of lines 5–8, the array $A[1 \ldots \text{heap-size}(A)]$ satisfies the max-heap property with possible one exception: $A[i]$ may be larger than $A[\text{Parent}(i)]$.

Insert

Inserting is now easy:

1. add a new element to the end of heap
2. set its key to desired value

Heap-Insert(A, key)

```
1: heap-size$(A)$ $\leftarrow$ heap-size$(A) + 1$
2: $A$[heap-size$(A)$] $\leftarrow -\infty$
3: **Heap-Increase-Key**$(A, \text{heap-size}(A), \text{key})$
```

Running time is $O(\log n)$

Conclusion: all considered operations can be implemented to run in time $O(\log n)$, **Maximum**$()$ even in $O(1)$

Note: there are other, more efficient implementations around!

Exercise 3.1. Consider an implementation of priority queue using a sorted array instead of a max-heap. What are the running time of the four operations needed for priority queue?

A lower bound for comparison-based sorting

We've seen a few sorting algorithms:

Selection-Sort: in place, upper bound $O(n^2)$

Merge-Sort: upper bound $O(n \log n)$

Heap-Sort: in place, upper bound $O(n \log n)$

Can we get any better algorithm?
Or is $\Theta(n \log n)$ an inherent barrier?

a comparison sort – information about the elements is collected only via comparing the elements

We prove that any *comparison sort* must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements.

input sequence $\langle a_1, a_2, \dots, a_n \rangle$
for a_i and a_j we can perform any of the tests
$a_i < a_j?$, $a_i \leq a_j?$, $a_i > a_j?$, $a_i \geq a_j?$
to determine their relative order

we are not interested in actual values of the elements

for simplicity let's us assume that $a_i \neq a_j$ for all $i \neq j$, and hence we can assume that all comparisons have the form $a_i \leq a_j?$

Decision tree

- a **full binary tree** (every node has either zero or two children)
- represents comparisons between elements performed by particular algorithm run on all possible inputs
- only **comparisons** are relevant, everything else is ignored

- **internal nodes** are labelled $i : j$ for $1 \le i, j \le n$, meaning elements a_i and a_j are compared
- **edges** are labelled "$\le$" or $>$, depending on the outcome of the comparisons
- **leaves** are labelled with a permutation $\boxed{\pi(1), \pi(2), \ldots, \pi(n)}$ – representing the output of the algorithm, i.e., at this point $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$
- a **branch** from the root to a leaf describes a sequence of comparisons (nodes and edges) resulting in a permutation (a leaf)
- a **reachable node** – a node into which we can get on an input; especially we are interested in reachable leaves

Example:

$\boxed{\text{SELECTION-SORT}}$

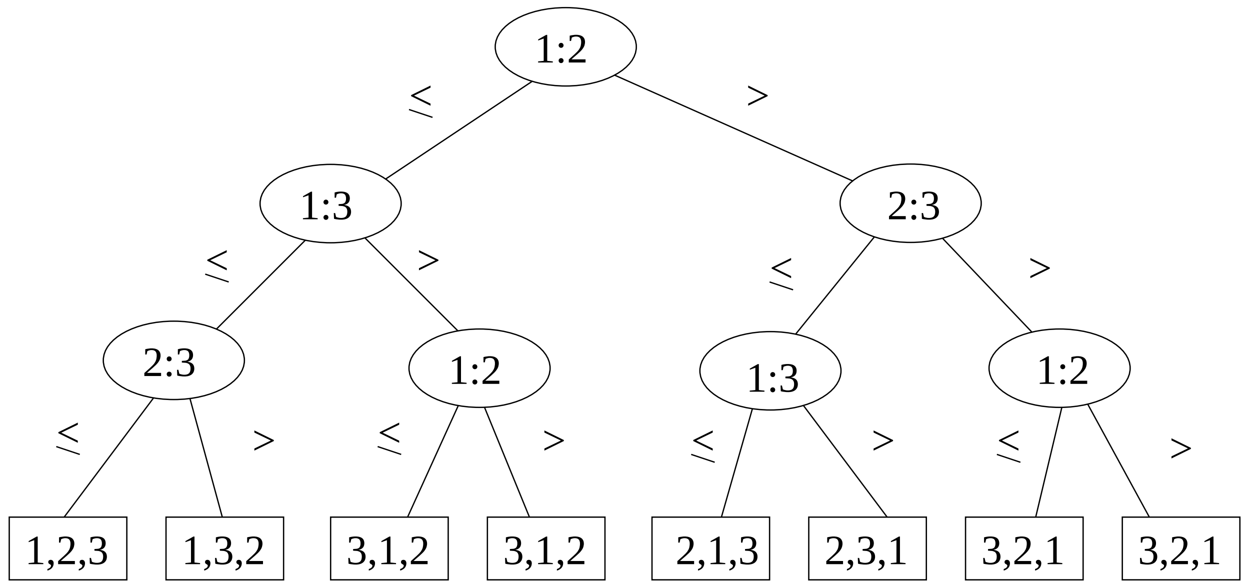
```
1: for $i \leftarrow 1$ to $n-1$ do
2:     /* 3–10: find min in $A[i], \ldots, A[n]$ */
3:     smallest_element $\leftarrow A[i]$
4:     smallest_position $\leftarrow i$
5:     for $j \leftarrow i+1$ to $n$ do
6:         if $A[j] <$ smallest_element then
7:             smallest_element $\leftarrow A[j]$
8:             smallest_position $\leftarrow j$
9:         end if
10:    end for
11:    swap $A[i]$ and $A[$smallest_position$]$
12: end for
```

Note: distinguish between the input sequence $a_1, a_2, \ldots, a_n$ and the array $A[1], A[2], \ldots, A[n]$ used to store the sequence.

in the beginning we have
$A[1] = a_1$, $A[2] = a_2$, …, $A[n] = a_n$

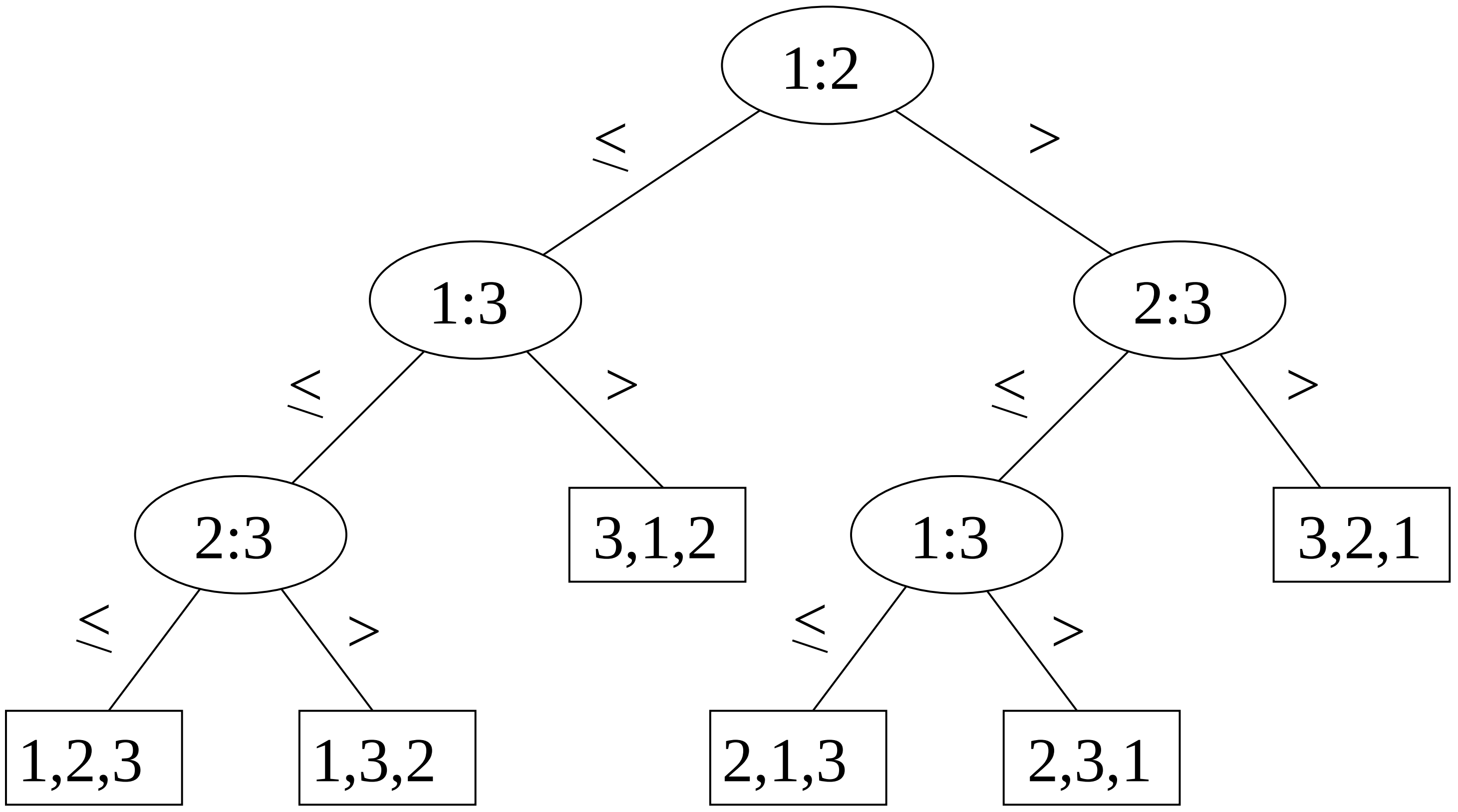
but in the end we want
$A[1] = a_{\pi(1)}$, $A[2] = a_{\pi(2)}$, …, $A[n] = a_{\pi(n)}$

Decision tree for **Selection-Sort** on 3-element inputs



```
SELECTION-SORT
for $i \leftarrow 1$ to $n-1$ do
  /* 3–10: find min in $A[i], \ldots, A[n]$ */
  smallest_element $\leftarrow A[i]$
  smallest_position $\leftarrow i$
  for $j \leftarrow i+1$ to $n$ do
    if $A[j] <$ smallest_element then
      smallest_element $\leftarrow A[j]$
      smallest_position $\leftarrow j$
    end if
  end for
  swap $A[i]$ and $A$[smallest_position]
end for
```

More efficient decision tree



Assignment Problem 3.3. (deadline: May 27, 5:30pm)
Draw an “efficient” decision tree for inputs with 4 elements.

Hint: The height of the tree should be 5.

Any correct sorting algorithm must be able to produce each permutation of its input

Thus, a necessary condition is that each of the $n!$ permutations must appear in a reachable leaf of the decision tree

Lower bound for worst case

Length of longest path from root of decision tree to any leaf represents worst case number of comparisons

Lower bound on heights of all decision trees where each permutation appears as leaf is thus lower bound on running time of **any** comparison based sort algorithm

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

Proof. Sufficient to determine the height of a decision tree in which each permutation appears as a leaf

Consider decision tree of height h with ℓ leaves corresponding to a comparison sort on n elements

Each of the n permutations of the input appears as a leaf, therefore $n! \leq \ell$

Binary tree of height h has at most 2^h leaves, thus $n! \leq \ell \leq 2^h$

Take logs: $h \geq \log(n!) = \Omega(n \log n)$

Exercise 3.2. Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Quicksort

Quicksort is based on Divide&Conquer paradigm

its worst-case running time is $\Theta(n^2)$

so why should we consider it?

efficient in average:
the expected running time is $\Theta(n \log n)$ with a **small** constant (hidden in Theta-notation)!

in place

in practice, we don't care that it performs "very bad" for some few inputs, if it performs "very good" on the most of them
$\Longrightarrow$
the best practical choice for sorting

Input: array $A[1\ldots n]$

how to sort a subarray $A[p\ldots r]$:

Divide: Compute (in some way) some index q, and partition $A[p\ldots r]$ into two (**possibly empty**) subarrays $A[p\ldots q-1]$ and $A[q+1\ldots r]$ s.t.

- each element of $A[p\ldots q-1]$ is less than or equal to $A[q]$, and
- each element of $A[q+1\ldots r]$ is greater than $A[q]$
- element $A[q]$ is called a **pivot**

Conquer: Sort the two subarrays $A[p\ldots q-1]$ and $A[q+1\ldots r]$ recursively

Combine: Subarrays are sorted in-place, nothing is needed here; the entire $A[p\ldots r]$ is sorted

the performance of **Quicksort** depends *extremely* on how the **pivots** are selected!

```
**Quicksort**$(A, p, r)$
1: if $p < r$ then
2:     $q \leftarrow$ Partition$(A, p, r)$
3:     Quicksort$(A, p, q - 1)$
4:     Quicksort$(A, q + 1, r)$
5: end if
```

To sort the entire array A, call
Quicksort$(A, 1, \text{length}(A))$

Note: this is a very “naive” implementation because it recurses down to the very end.

Efficient implementations do special things at the bottom of recursion (calls **optimal** procedures to sort sub-arrays of lengths up to, say, 7)

So, a more “practical” solution would look something like this:

```
**Quicksort**$(A, p, r)$
1: if $r - p = 2$ then
2:     Sort2$(A, p)$
3: else if $r - p = 3$ then
4:     Sort3$(A, p)$
5: else if $r - p = 4$ then
6:     Sort4$(A, p)$
7: else if $r - p = 5$ then
8:     Sort5$(A, p)$
9: else if $r - p = 6$ then
10:     Sort6$(A, p)$
11: else if $r - p = 7$ then
12:     Sort7$(A, p)$
13: else if $p < r$ then
14:     $q \leftarrow$ Partition$(A, p, r)$
15:     Quicksort$(A, p, q - 1)$
16:     Quicksort$(A, q + 1, r)$
17: end if
```

Partition

The most important part of **Quicksort**.

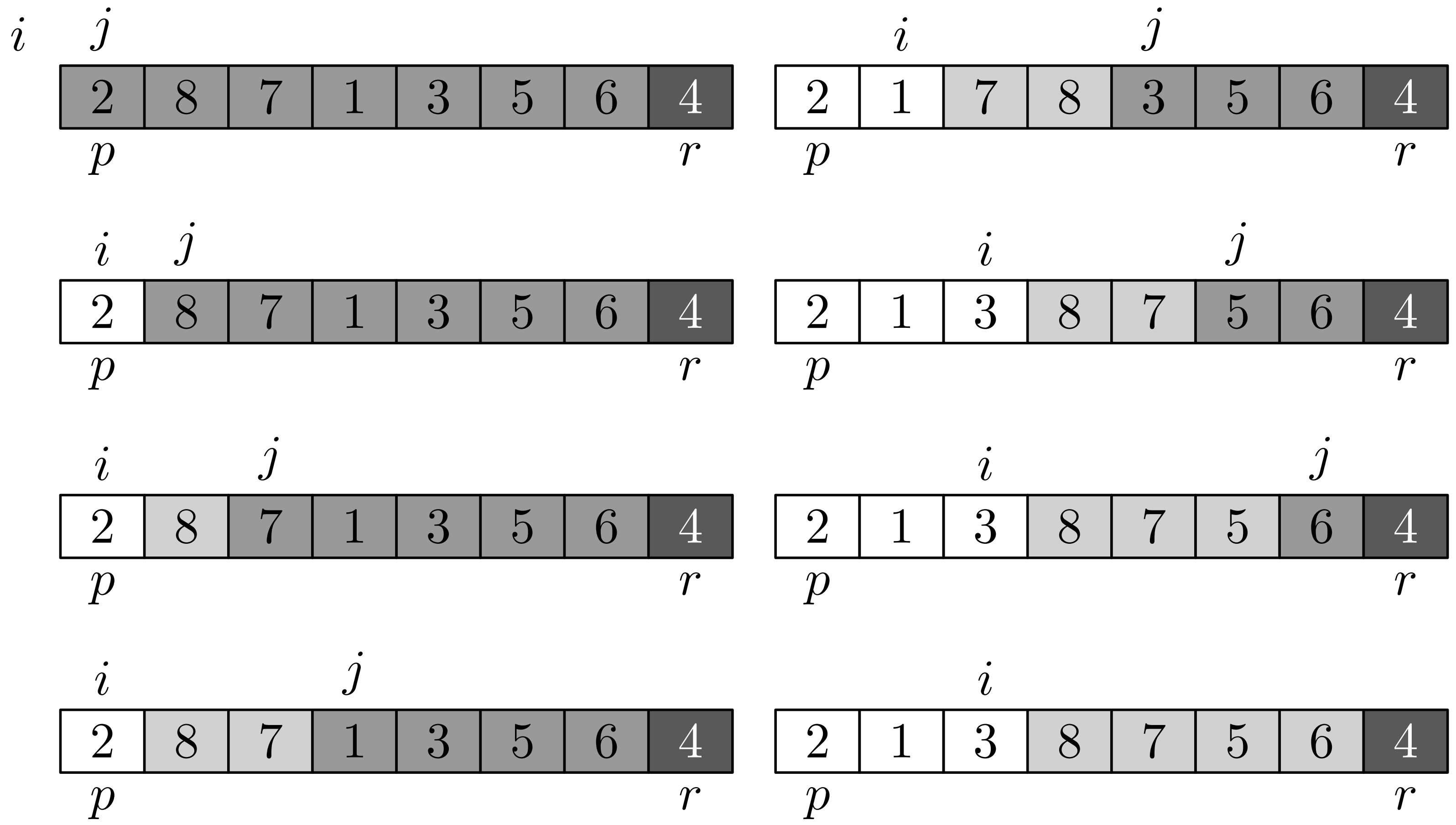
Partition(A, p, r)

```
 1: $x \leftarrow A[r]$        /* choose a pivot $x$ */
 2: $i \leftarrow p - 1$
 3: for $j \leftarrow p$ to $r - 1$ do
 4:     if $A[j] \leq x$ then
 5:         $i \leftarrow i + 1$
 6:         exchange $A[i] \leftrightarrow A[j]$
 7:     end if
 8: end for
 9: exchange $A[i+1] \leftrightarrow A[r]$
10: return $i + 1$
```

```
**Partition**$(A, p, r)$
 1: $x \leftarrow A[r]$        /* choose a pivot $x$ */
 2: $i \leftarrow p - 1$
 3: **for** $j \leftarrow p$ **to** $r - 1$ **do**
 4:     **if** $A[j] \leq x$ **then**
 5:         $i \leftarrow i + 1$
 6:         exchange $A[i] \leftrightarrow A[j]$
 7:     **end if**
 8: **end for**
 9: exchange $A[i + 1] \leftrightarrow A[r]$
10: return $i + 1$
```

the loop invariant:

- at the beginning of each iteration of the loop 3–8
  1. each element of $A[p \ldots i]$ is smaller than or equal to pivot x
  2. each element of $A[i + 1 \ldots j - 1]$ is greater than x
  3. elements $A[j \ldots r - 1]$ are so-far arbitrary
  4. element $A[r] = x$


i j
2 8 7 1 3 5 6 4
p r
i j
2 8 7 1 3 5 6 4
p r
i j
2 8 7 1 3 5 6 4
p r
i j
2 8 7 1 3 5 6 4
p r
i j
2 1 7 8 3 5 6 4
p r
i j
2 1 3 8 7 5 6 4
p r
i j
2 1 3 8 7 5 6 4
p r
i
2 1 3 8 7 5 6 4
p r

Correctness:

enough to prove first 2 conditions (condition 4 satisfied by line 1)

Initialization: before 1st iteration, $i = p - 1$ and $j = p$; first two conditions are trivial

Maintenance: 2 cases:

$\boxed{A[j] > x}$ – only j is increased in the loop; enough to check condition 2 for element $A[j - 1]$

$\boxed{A[j] \leq x}$ – i is incremented, then $A[i]$ and $A[j]$ are swapped, then j is incremented; enough to check condition 1 for $A[i]$ (ok, because of swapping) and condition 2 for $A[j - 1]$ (ok, by the loop invariant)

Termination: $j = r$, hence there are no elements in part 3, the others are divided as required

finally, to get correct order of these 3 parts, we swap the first element of the second part with $A[r]$

running time $\Theta(n)$, where $n = r - p + 1$

This particular implementation of **Partition** is **not** good because it always uses the rightmost element as pivot.

It's easy to come up with examples that force this implementation to be very bad (meaning overall running time $\Omega(n^2)$).

What we want is an **even** partition of $A[p \ldots r]$ into two sub-arrays of (about) the same size.

That's the second very important feature of efficient implementations of **Quicksort**.

There are several not-too-bad ways:

- look at, say, 5 **fixed** array elements and pick the median
- pick a **randomly chosen** element
- look at, say 5 **randomly chosen** elements and pick the median
- many more

In practice, the random variant usually works best (unless you happen to know something about your inputs)