Premaster Course Algorithms 1

Chapter 2: Heapsort and Quicksort

Christian Scheideler

SS 2019

Heapsort

Motivation: Consider the following sorting algorithm

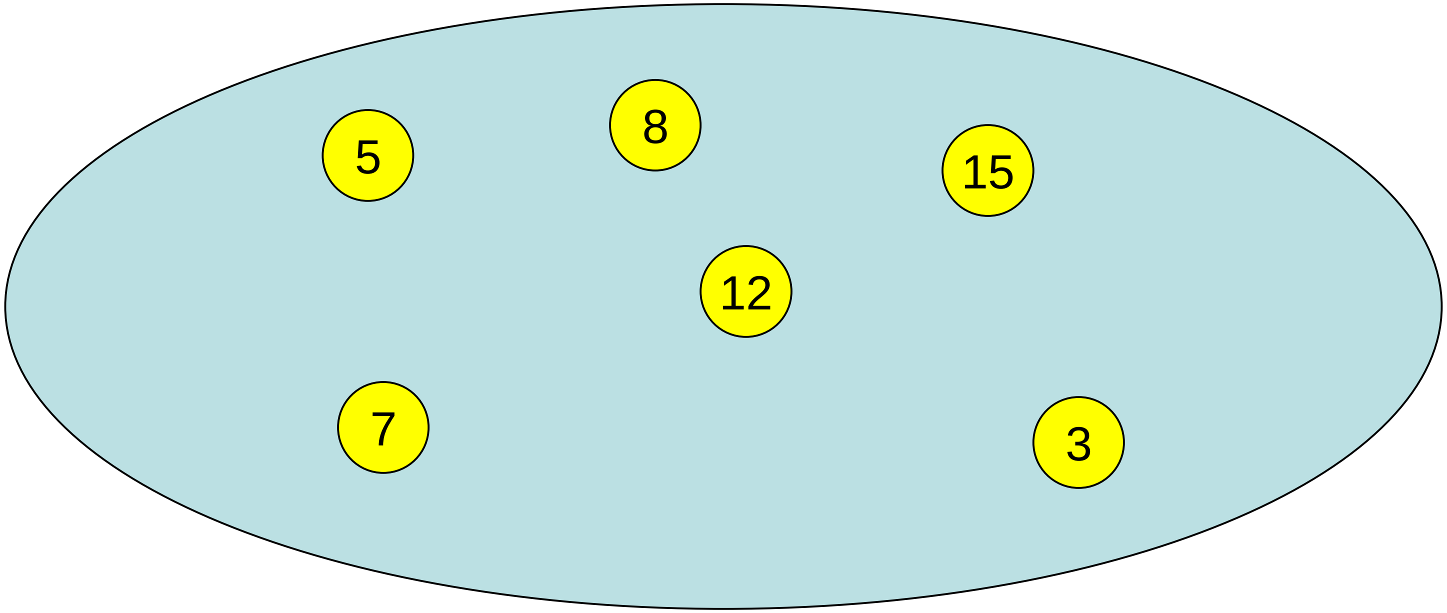
Input: Array *A*
Output: Numbers in *A* in ascending order

```
Max-Sort(A):
  for i←length(A) downto 2 do
    m←Max-Search(A[1..i]) // returns index of maximum
    A[m]↔A[i]
```

Is there a data structure that allows us to find the maximum quicker than with Max-Search implemented in the same way as Min-Search in Chapter 1?

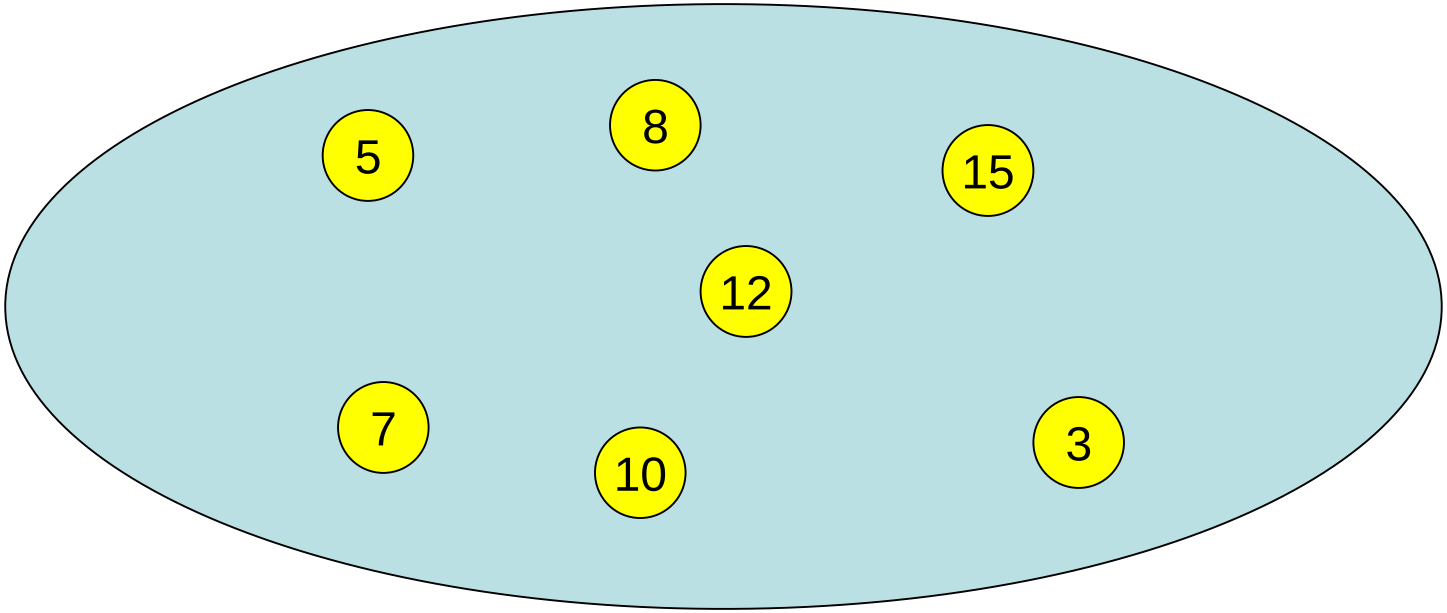
→ priority queues

Priority Queue



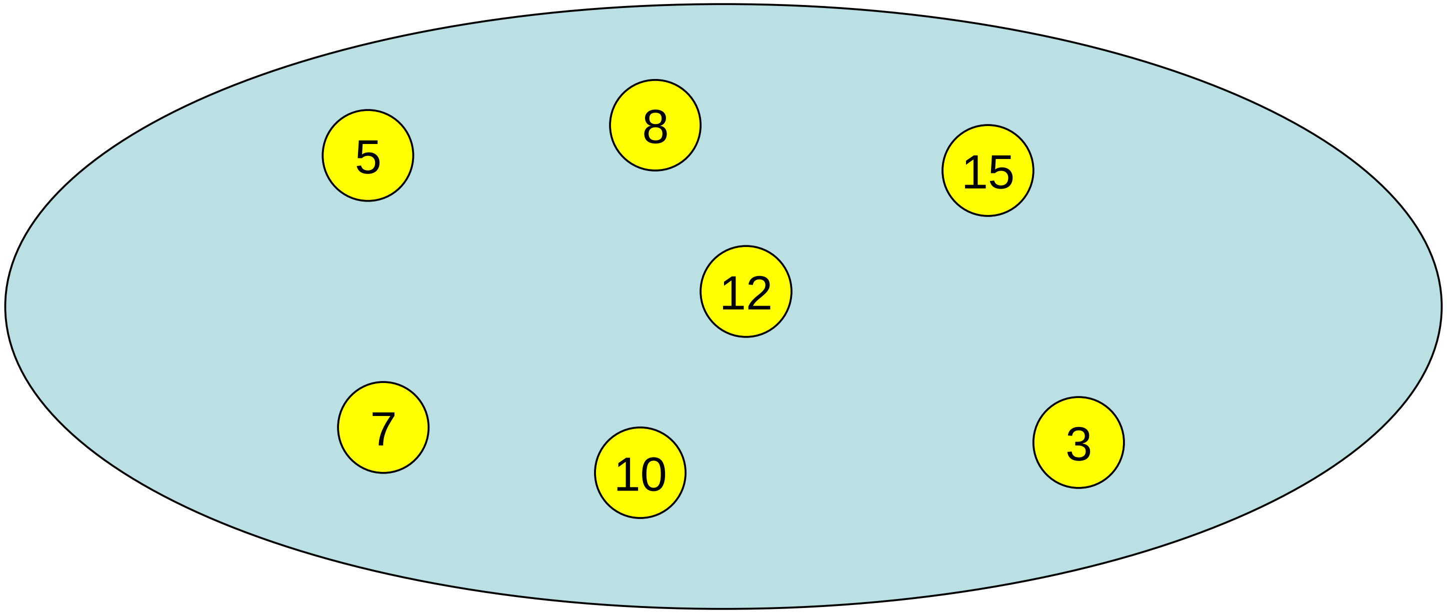
Priority Queue

insert(10)



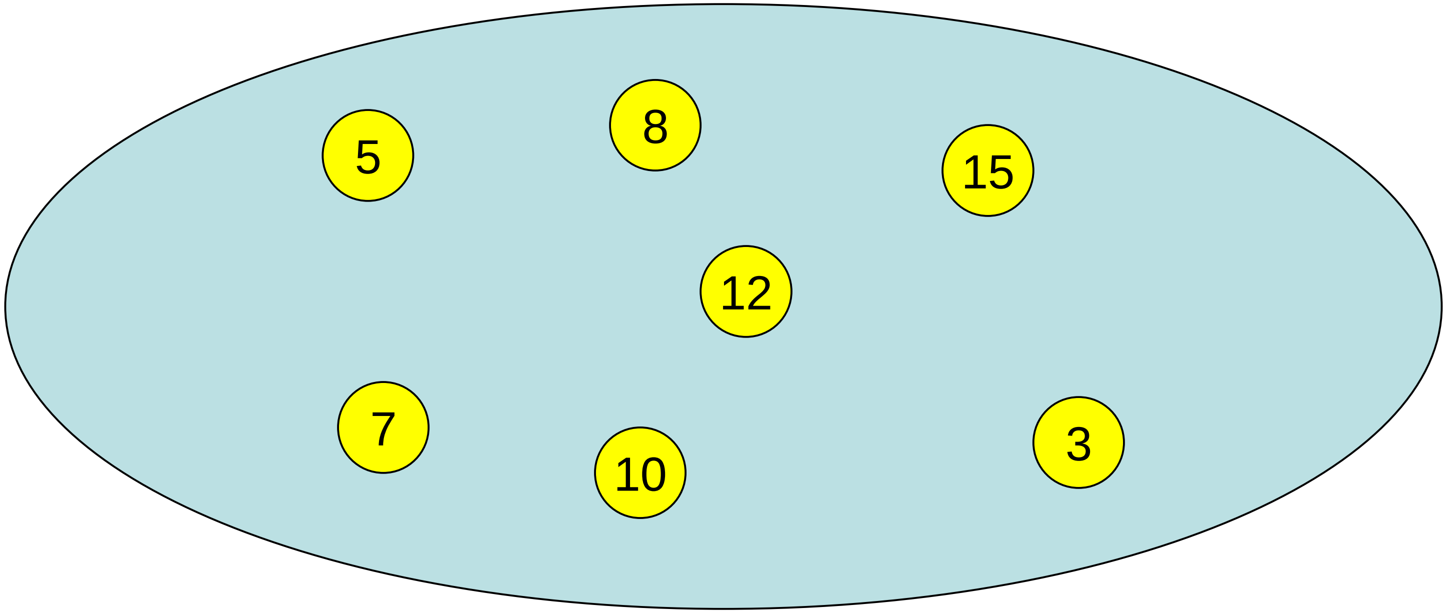
Priority Queue

min() outputs 3 (minimal element)



Priority Queue

deleteMin()



Priority Queue

M: set of elements in priority queue

Every element e identified by key(e).

Operations:

- M.build($\{e_1,\ldots,e_n\}$): M:=$\{e_1,\ldots,e_n\}$
- M.insert(e: Element): M:=M∪{e}
- M.min: outputs e∈M with minimal key(e)
- M.deleteMin: like M.min, but additionally M:=M\{e}, for that e with minimal key(e)

Priority Queue

- Priority Queue based on unsorted list:
  - build({e_1,…,e_n}): time O(n)
  - insert(e): O(1)
  - min, deleteMin: O(n)
- Priority Queue based on sorted array:
  - build({e_1,…,e_n}): time O(n log n) (needed for sorting)
  - insert(e): O(n)   (rearrange elements in array)
  - min, deleteMin: O(1)

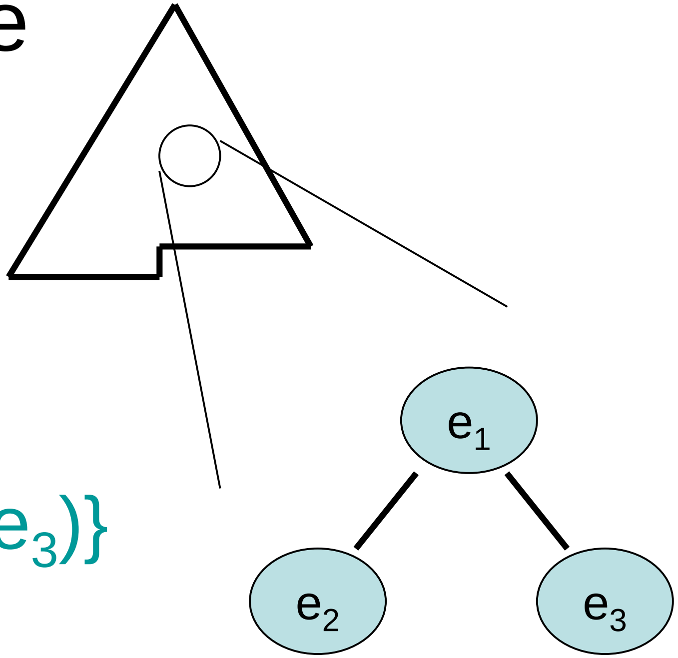
Better structure needed than list or array!

Binary Heap

Idee: use binary tree instead of list

Preserve two invariants:

- Form invariant:complete binary tree up to lowest level

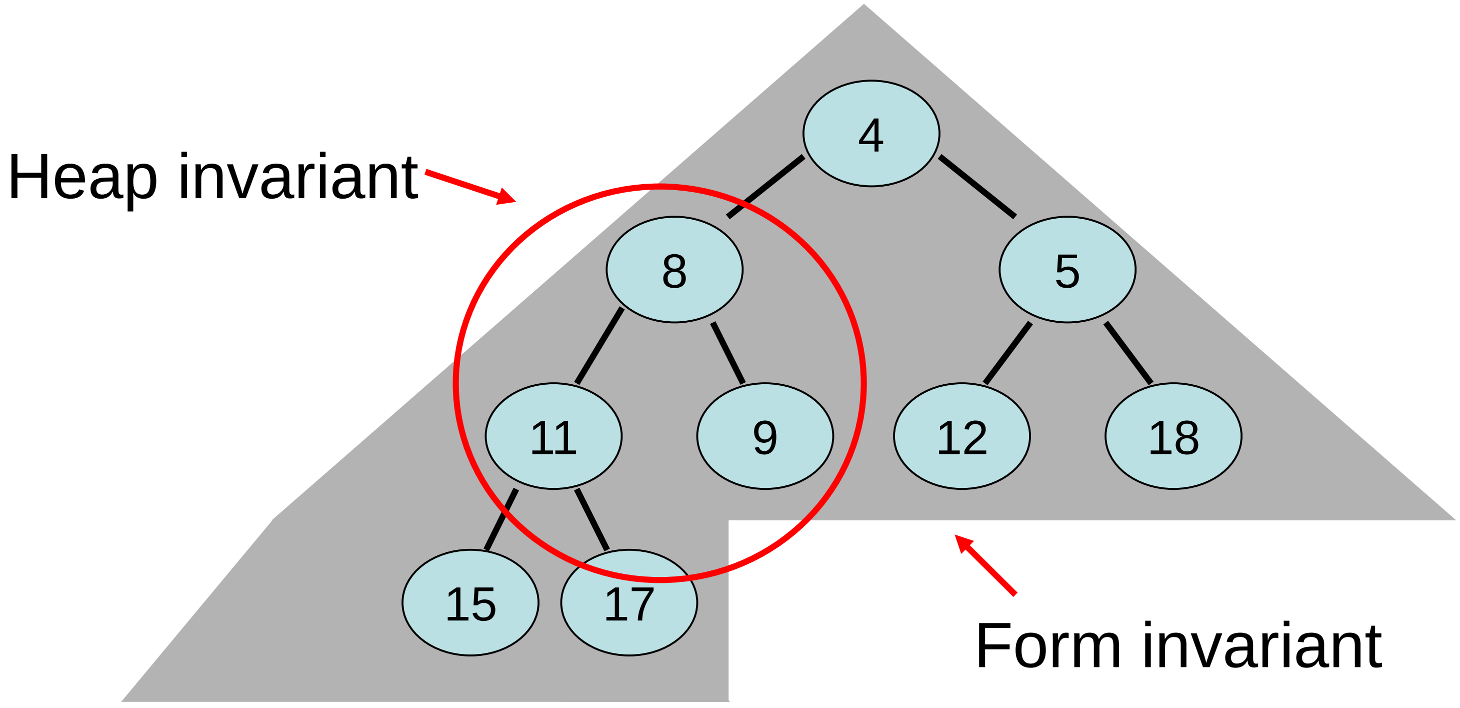


- Heap invariant:

$key(e_1) \leq \min\{key(e_2),key(e_3)\}$

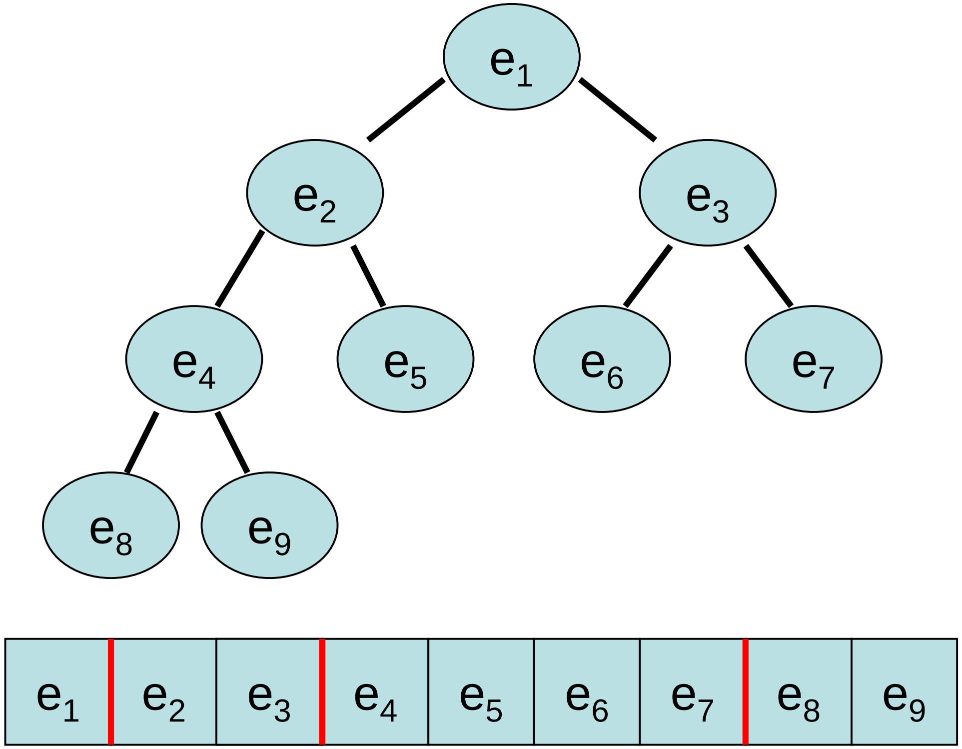
Binary Heap

Example:



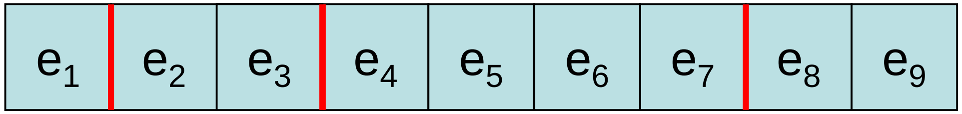
Binary Heap

Representation of binary tree via array:



Binary Heap

Representation of binary tree via array:



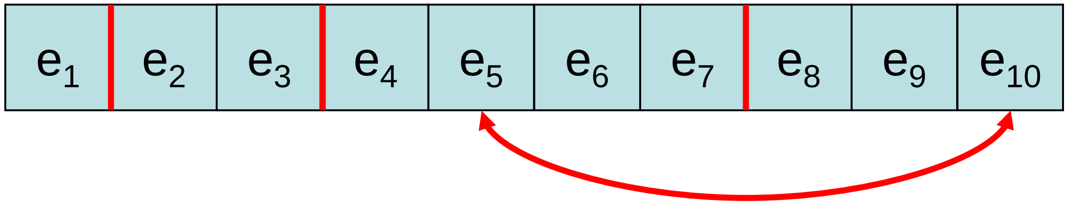
e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9

- H: Array [1..N] of Element (N $\geq$ #elements n)
- Children of e in H[i]: in H[2i], H[2i+1]
- Form invariant: H[1],…,H[n] occupied
- Heap invariant: for all $i \in \{2,\dots,n\}$,

$$key(H[i]) \geq key(H[\lfloor i/2 \rfloor])$$

Binary Heap

Representation of binary tree via array:



insert(e):

- Form invariant: n:=n+1; H[n]:=e
- Heap invariant: as long as e is in H[k] with $k>1$ and $key(e)<key(H[\lfloor k/2 \rfloor])$, switch e with parent

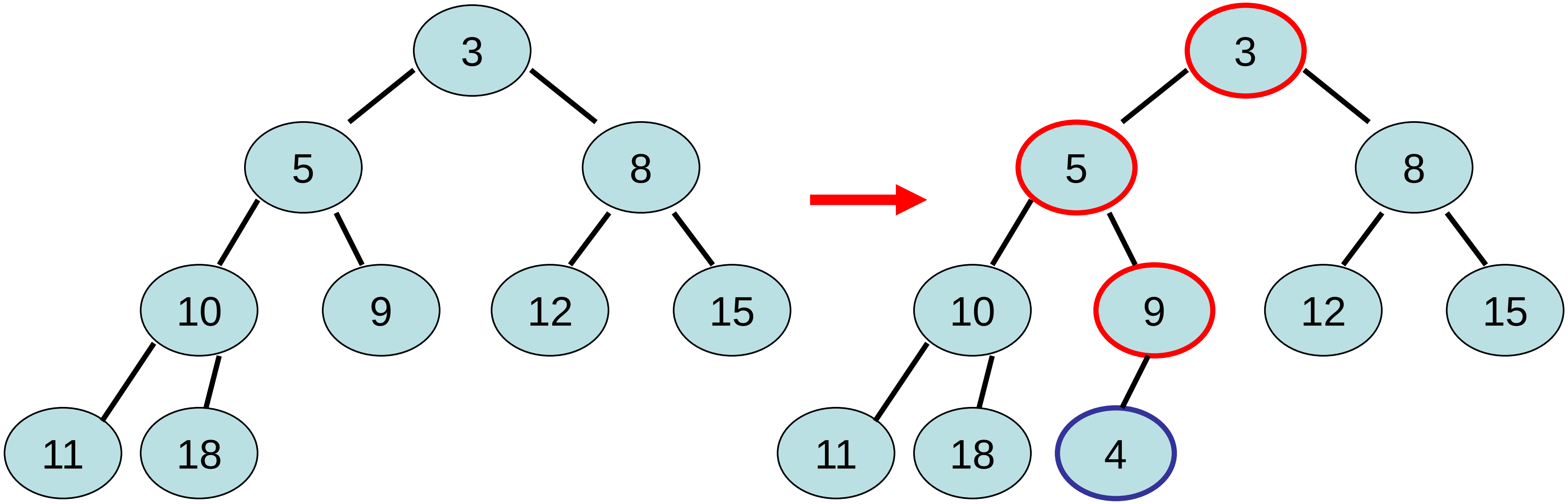
Insert Operation

```
insert(e: Element):
   n:=n+1; H[n]:=e
   heapifyUp(n)

heapifyUp(i: Integer):
   while i>1 and key(H[i])<key(H[⌊i/2⌋]) do
      H[i] ↔ H[⌊i/2⌋]
      i:=⌊i/2⌋
```

Runtime: $O(\log n)$

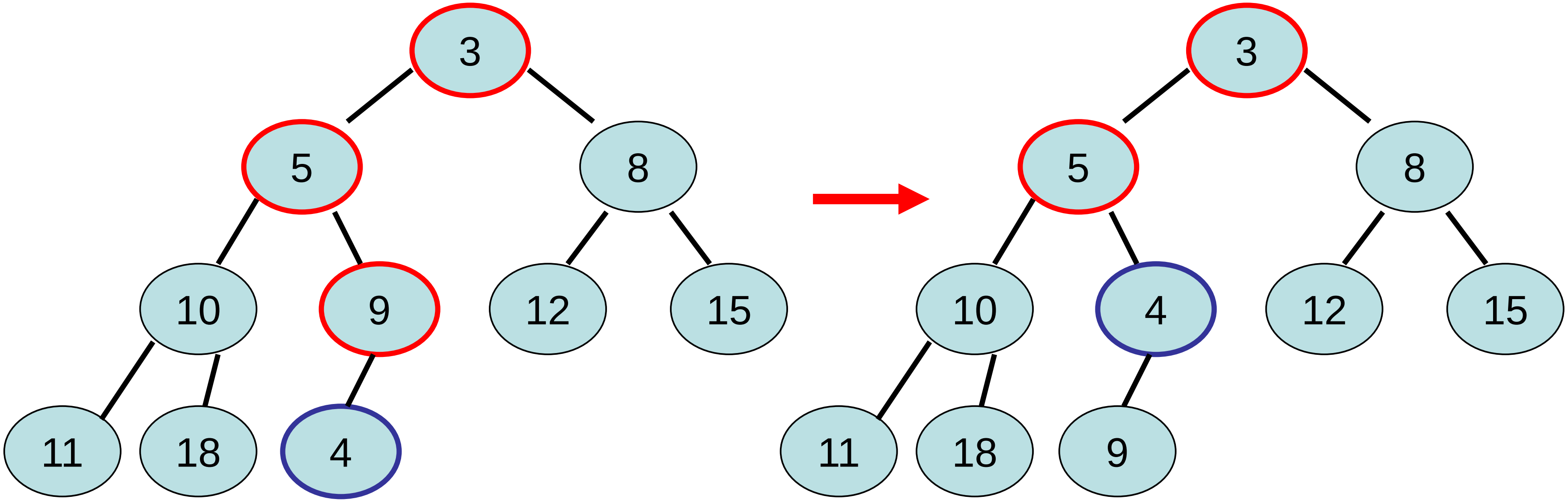
Insert Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

⬭ : nodes that may violate invariant

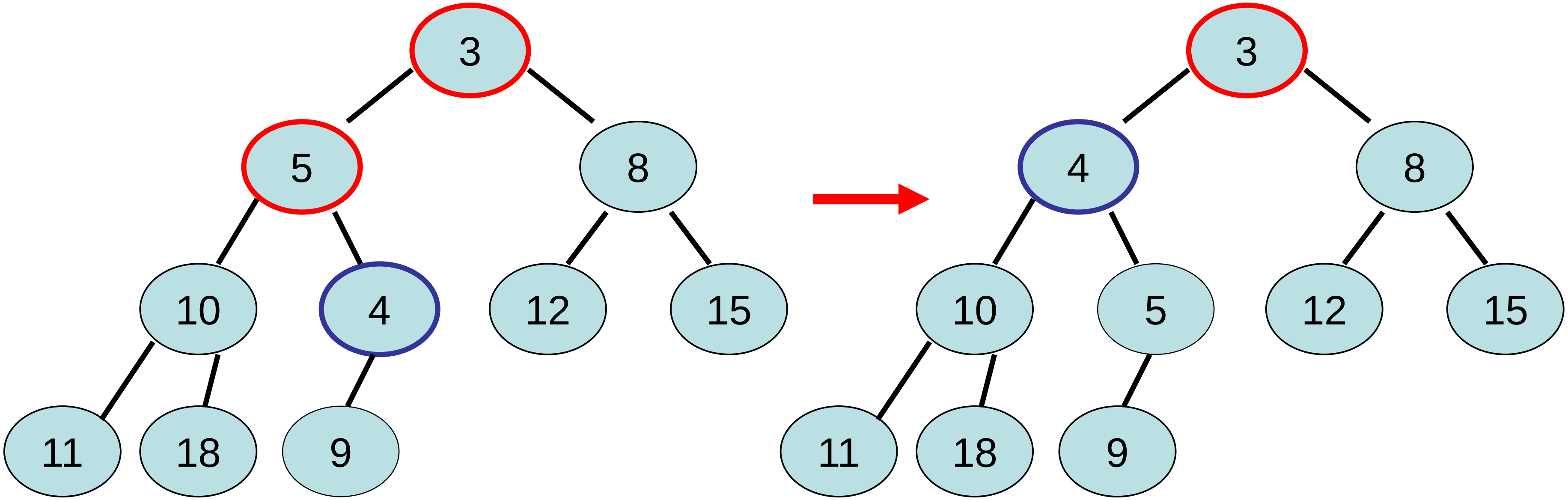
Insert Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

⬭ : nodes that may violate invariant

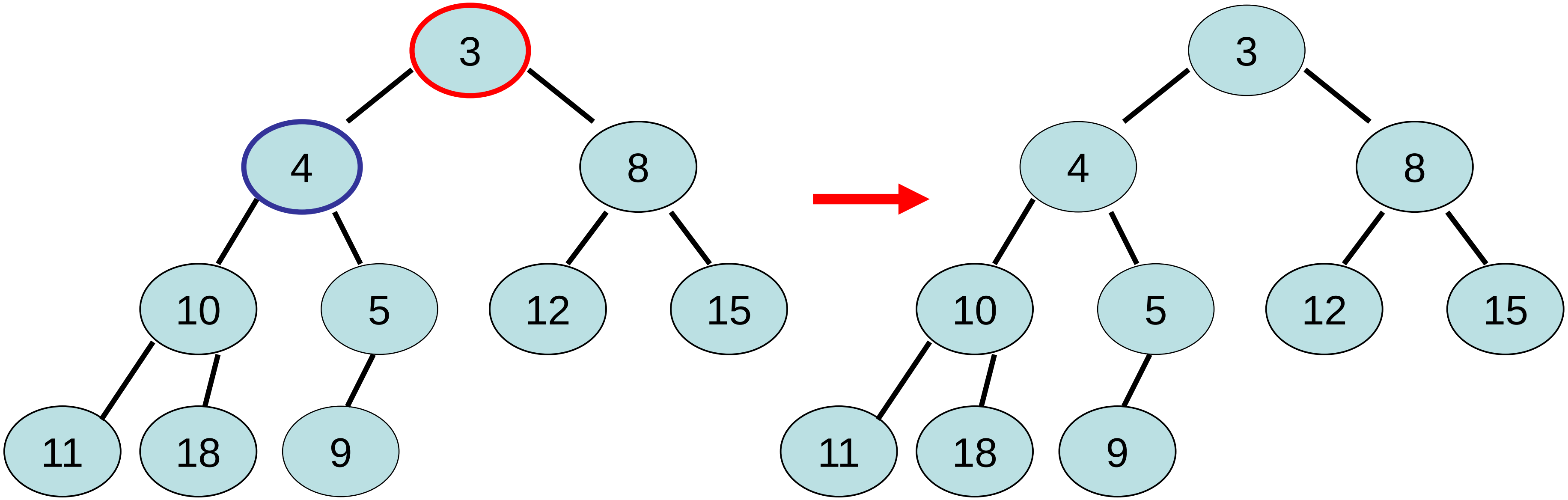
Insert Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

○ : nodes that may violate invariant

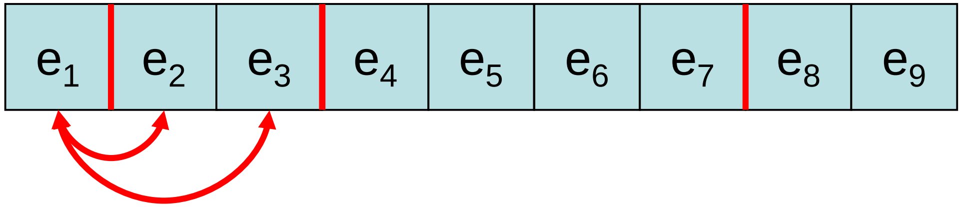
Insert Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

⭕ : nodes that may violate invariant

Binary Heap



deleteMin:

- Form invariant: H[1]:=H[n]; n:=n-1
- Heap invariant: start with e in H[1]. Switch e with the child with minimum key until $H[k] \leq \min\{H[2k],H[2k+1]\}$ for the current position k of e or e is in a leaf

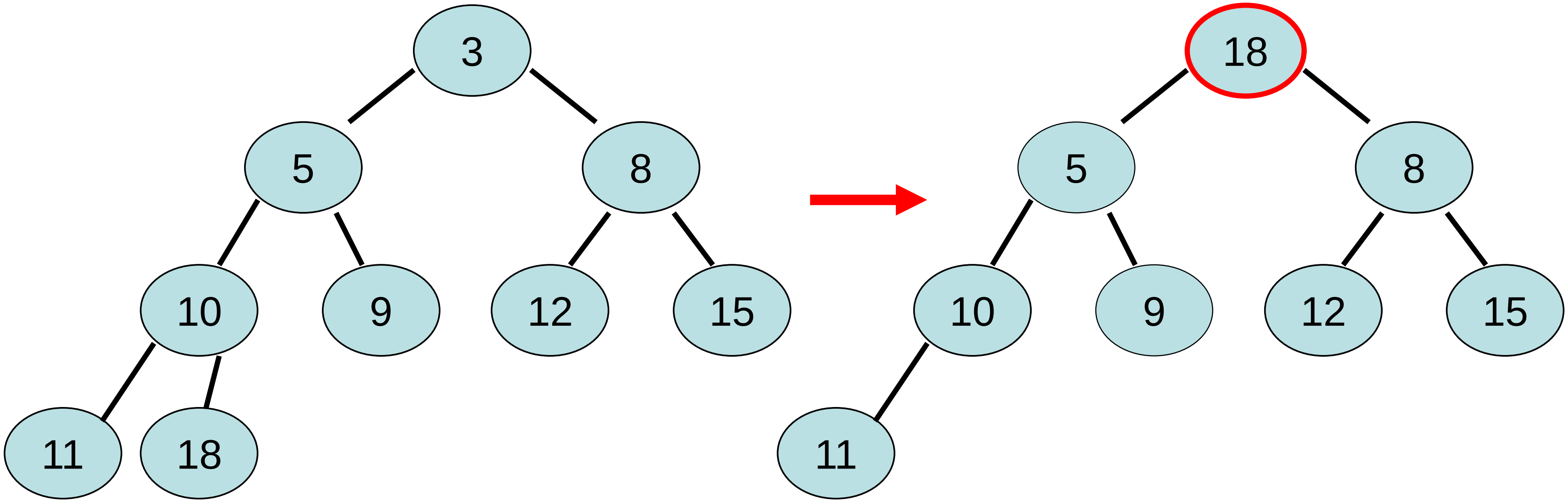
Binary Heap

Runtime: O(log n)

```
deleteMin():
    e:=H[1]; H[1]:=H[n]; n:=n-1
    heapifyDown(1)
    return e

heapifyDown(i: Integer):
    while 2i≤n do                     // i is not a leaf position
        if 2i+1>n then m:=2i  // m: pos. of the minimum child
        else
            if key(H[2i])<key(H[2i+1]) then m:=2i
                                       else m:=2i+1
        if key(H[i])≤key(H[m]) then return  // heap inv. holds
        H[i] ↔ H[m]; i:=m
```

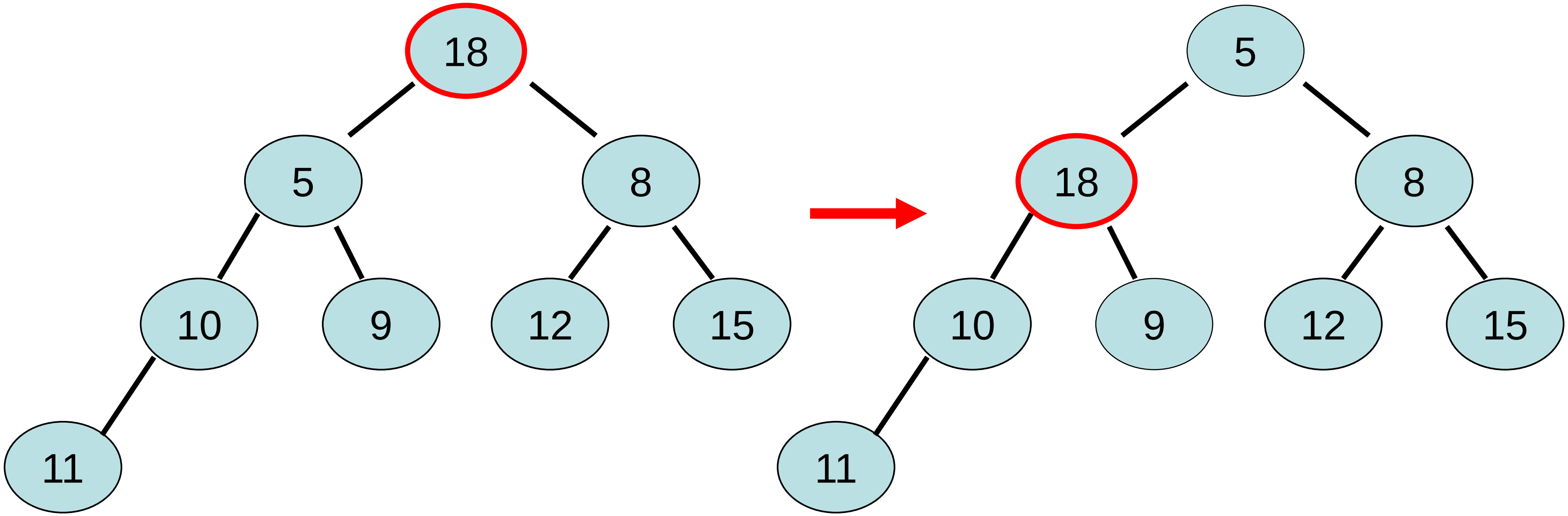
deleteMin Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

○ : nodes that may violate invariant

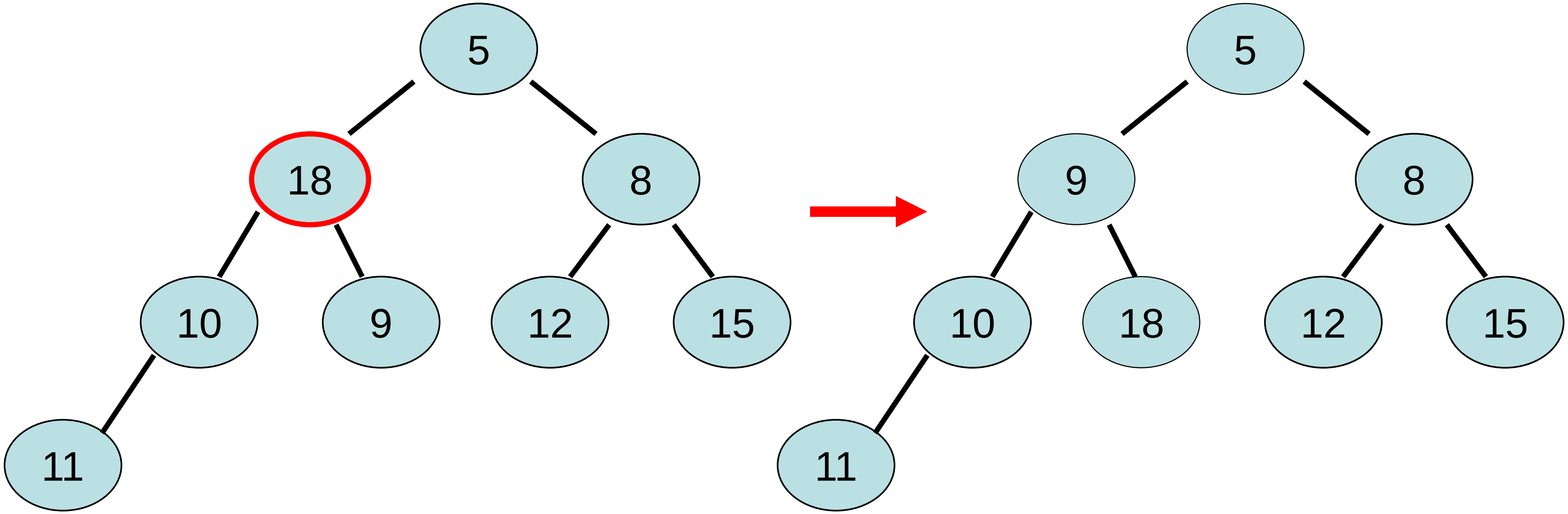
deleteMin Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

○ : nodes that may violate invariant

deleteMin Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

○ : nodes that may violate invariant

Binary Heap

build($\{e_1,\dots,e_n\}$):

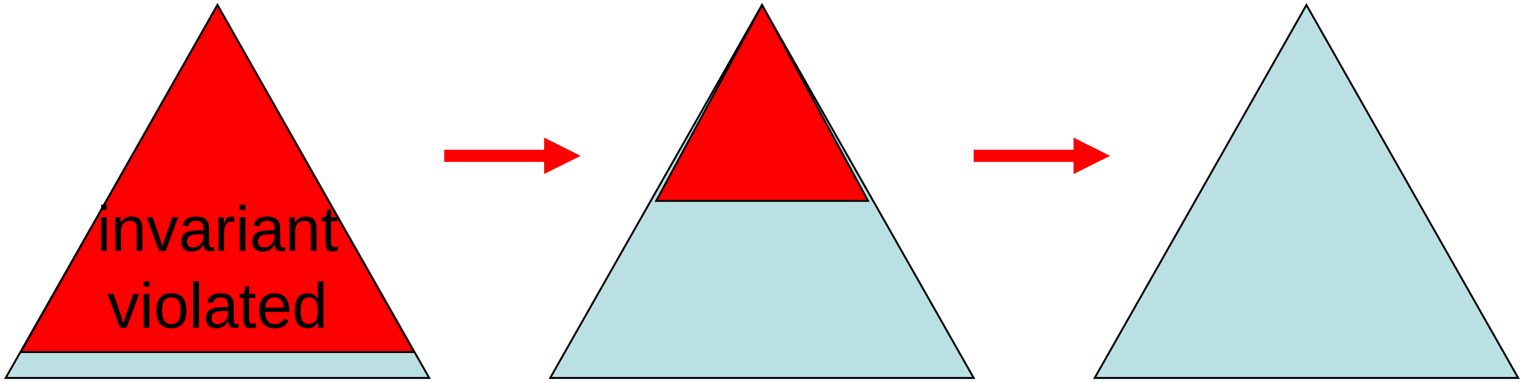
- Naive implementation: via n insert(e) operations. Runtime $O(n \log n)$
- Better implementation:

```
build({e_1,...,e_n}):
    for i:=⌊n/2⌋ downto 1 do
        heapifyDown(i)
```

- Runtime (with $k=\lceil \log n \rceil$):
  $O(\sum_{0\le l<k} \underbrace{2^l}_{\text{number of nodes in level l}} \underbrace{(k-l)}_{\text{runtime of heapifyDown for level l}}) = O(2^k \sum_{j\ge 1} j/2^j) = O(n)$

Binary Heap

Call HeapifyDown(i) for $i=\lfloor n/2 \rfloor$ down to 1:



Invariant: $\forall j>i$: H[j] min w.r.t. subtree of H[j]

Binary Heap

Runtime:

- build({e_1,…,e_n}): O(n)
- insert(e): O(log n)
- min: O(1)
- deleteMin: O(log n)

Heapsort

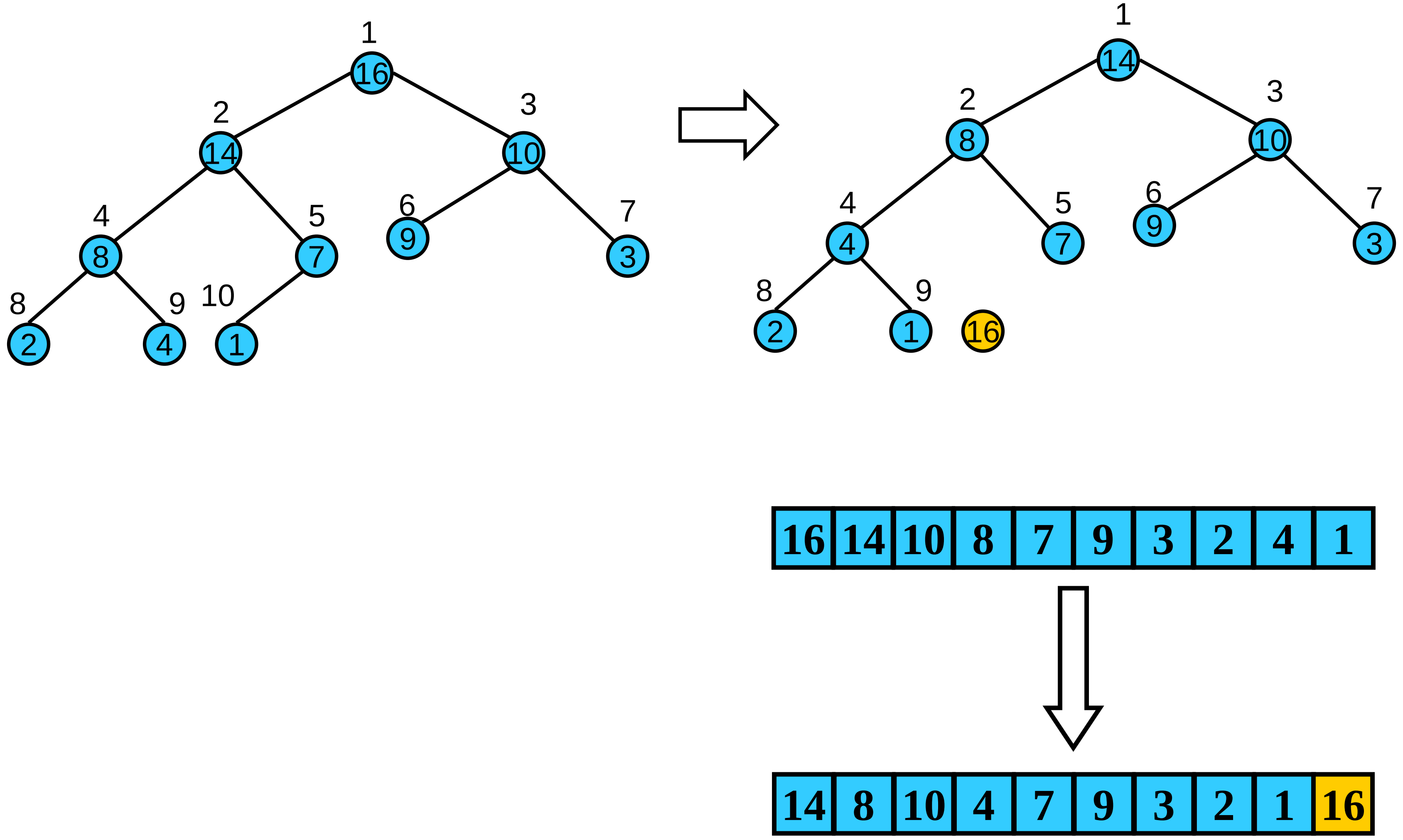
Input: Array *A*

Output: numbers in *A* in ascending order

```
Heapsort(A):
  Build-Max-Heap(A)  // like build, but for max-heap
  for i←length(A) downto 2 do
    A[i]:=DeleteMax(A)   // A[i] ← maximum in A[1..i]
```

Correctness: follows from correctness of Build-Max-Heap(A) and DeleteMax(A)

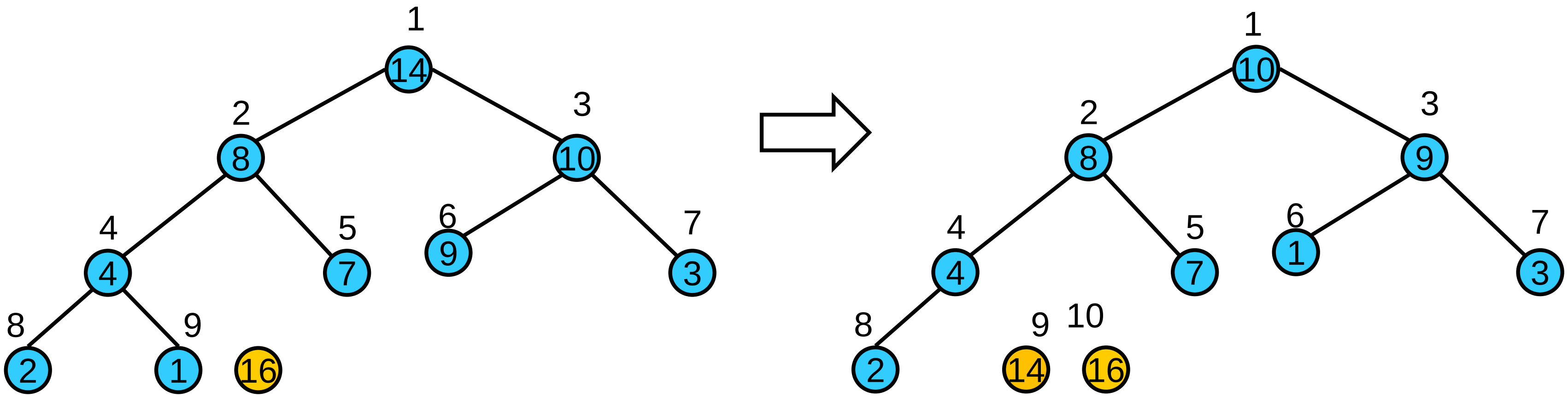
Illustration of Heapsort

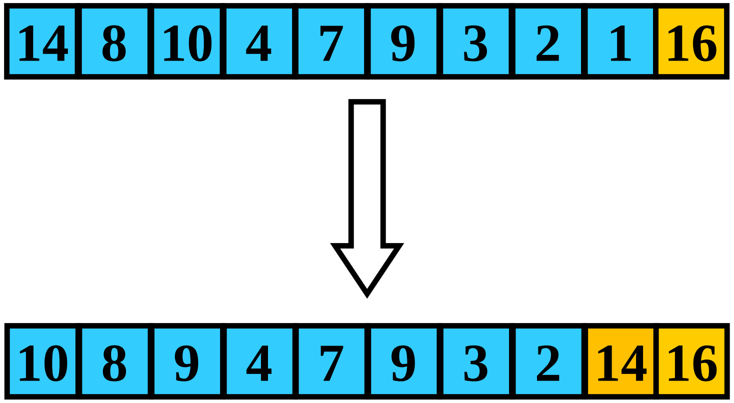


16	14	10	8	7	9	3	2	4	1

14	8	10	4	7	9	3	2	1	16

Illustration of Heapsort

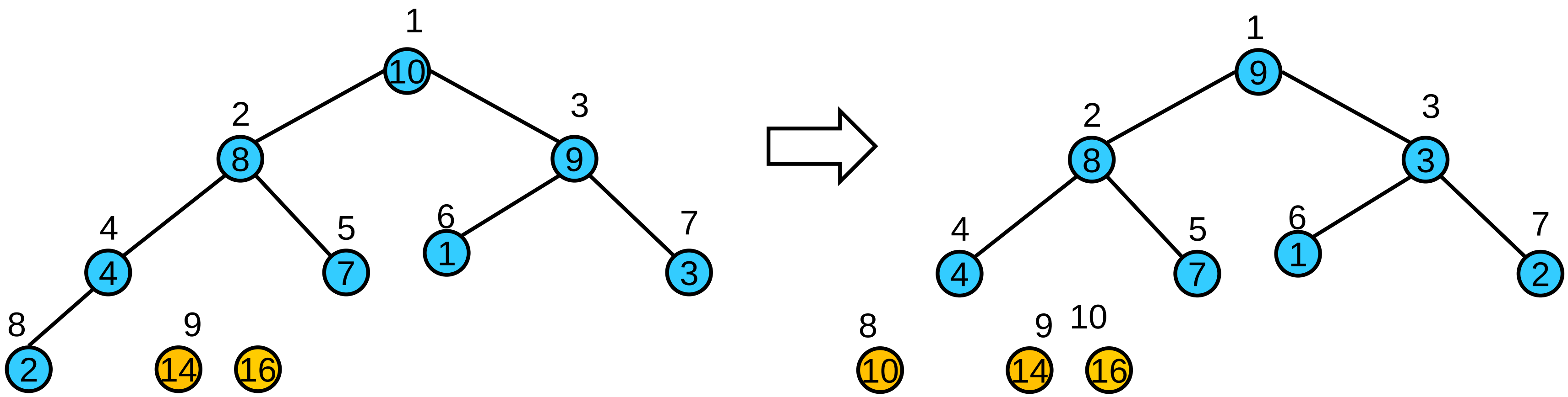




14	8	10	4	7	9	3	2	1	16

10	8	9	4	7	9	3	2	14	16

Illustration of Heapsort



10	8	9	4	7	9	3	2	14	16

9	8	3	4	7	1	2	10	14	16

Runtime of Heapsort

Theorem 2.1: Heapsort has a runtime of $O(n \log n)$.

Proof:

```
Heapsort(A):
  Build-Max-Heap(A)
    for $i \leftarrow$ length(A) downto 2 do
      A[i] $\leftarrow$ DeleteMax(A)
```

runtime:

$O(n)$

$\sum_{i=2}^{n} (O(1)+T(l))$

$O(\log n)$

$O(n \log n)$

Quicksort

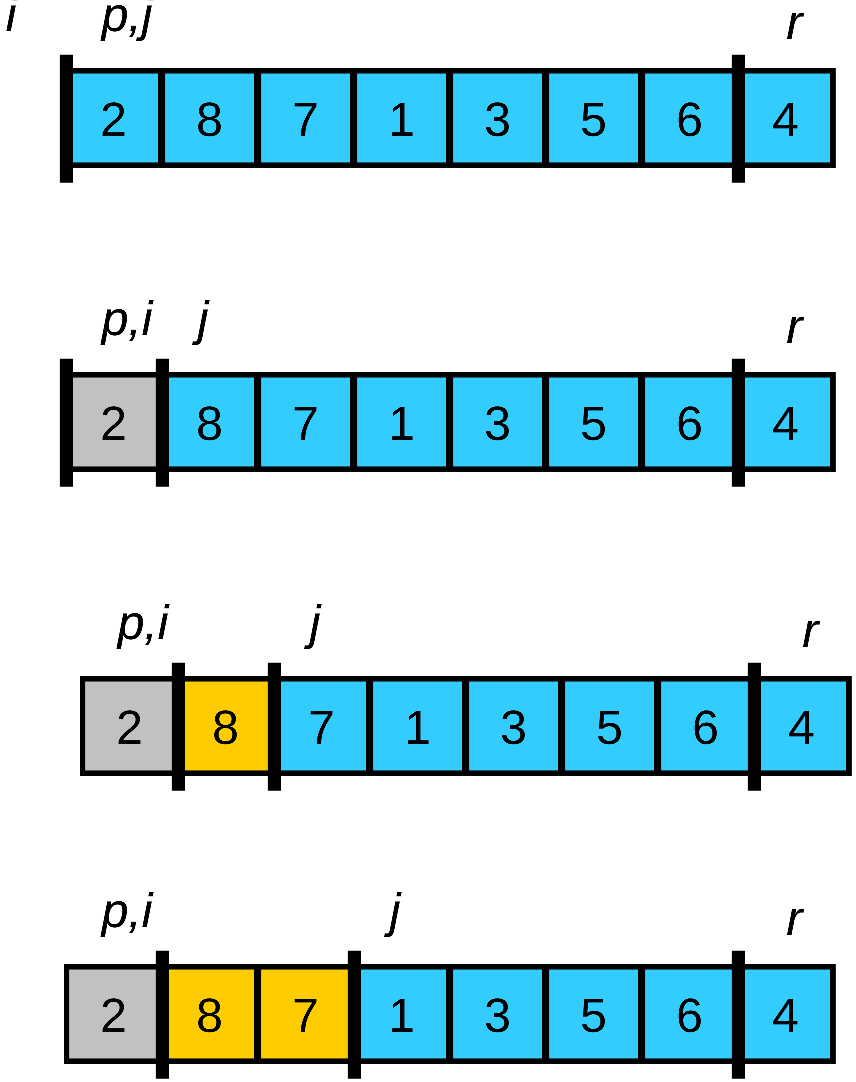
```
Algorithm Quicksort(A,p,r):
If p<r then
    q:=Partition(A,p,r)
    Quicksort(A,p,q-1)
    Quicksort(A,q+1,r)
```

Quicksort on array A called with:
Quicksort(A,1,length(A))

Quicksort

```
Algorithm Partition(A,p,r):
x:=A[r]                 // x: pivot element,
i:=p-1                  // x used for comparisons
for j:=p to r-1 do
    if A[j]≤x then
        i:=i+1
        A[i]↔A[j]
A[i+1]↔A[r]
return i+1
```

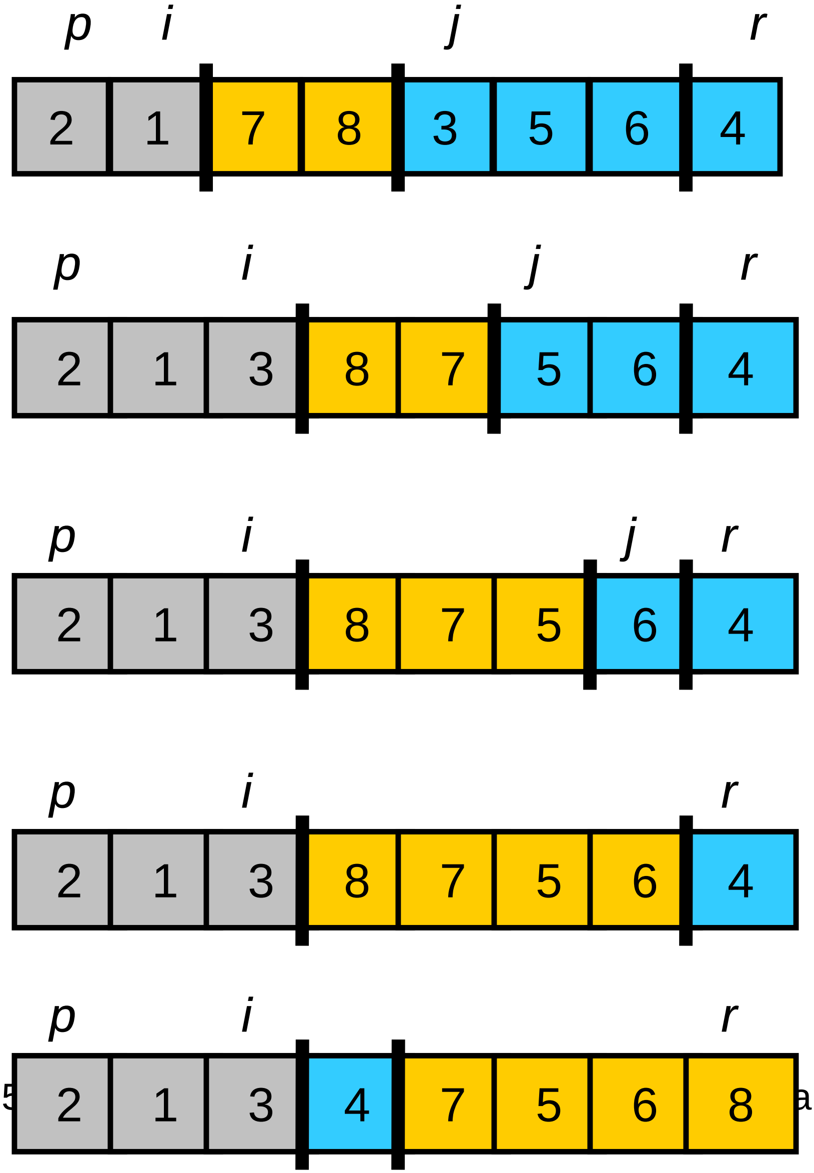
Quicksort



Algorithm Partition(A,p,r):

```
x:=A[r]
i:=p-1
for j:=p to r-1 do
    if A[j]≤x then
        i:=i+1
        A[i]↔A[j]
A[i+1]↔A[r]
return i+1
```

Quicksort



Algorithm Partition(A,p,r):

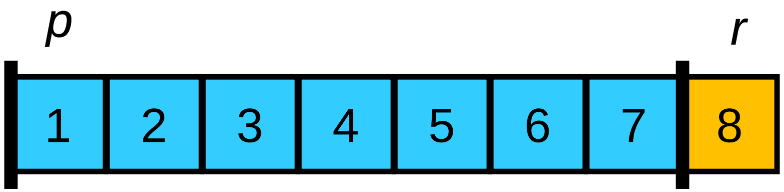
```
x:=A[r]
i:=p-1
for j:=p to r-1 do
    if A[j]≤x then
        i:=i+1
        A[i]↔A[j]
A[i+1]↔A[r]
return i+1
```

Runtime of Quicksort

Theorem 2.2: Quicksort has a worst-case runtime of $\Theta(n^2)$.

Proof:

- Suppose that the elements in A are already stored in ascending (resp. descending) order. Then the index returned by Partition(A,p,r) will always be r (resp. p).



- This will cause the recursions to be highly unbalanced ($T(n) = T(1)+T(n-1) + c\cdot n$).

Randomized Quicksort

Theorem 2.3: Suppose that the pivot x in Parition(A,p,r) is chosen uniformly at random from A[p..r]. Then the expected runtime of Quicksort is $O(n \log n)$.

Proof:

- W.l.o.g. we assume that $A=(a_1,\ldots,a_n)$ is a permutation of $\{1,\ldots,n\}$.
- We define the binary random variable $X_{i,j}$ to be 1 if and only if i and j are compared.
- Altogether, the number of comparisons is $\Sigma_{i<j} X_{i,j}$. This dominates the runtime of Quicksort, so it remains to compute $E[\Sigma_{i<j} X_{i,j}] = \Sigma_{i<j} E[X_{i,j}]$.
- It holds that $E[X_{i,j}]=Pr[X_{i,j}=1]=2/(j-i+1)$.
  (Proof: whiteboard, or see book).
  Inserting that into the sum gives the theorem.

Next Chapter

Topic: elementary data structures