Sorting and Heaps

Lecture #2 of Algorithms & Data Structures (Module 7)

Rom Langerak

E-mail: r.langerak@utwente.nl

February 7, 2019

Plan for today

Several sorting algorithms:

- Insertion sort
- Quicksort
- Mergesort

Heaps:

- What is a heap?
- How to build a heap?
- Priority queues, heapsort

Why look at sorting?

We look at sorting because

- sorting is done frequently and has many applications
- ideas for sorting give insight on how to improve algorithms
- ingenious and optimal algorithms have been found

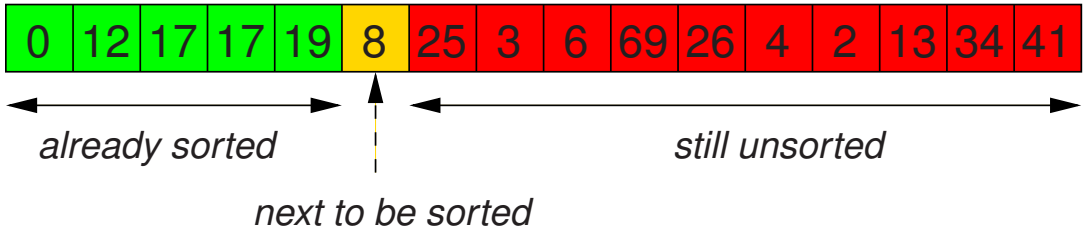
We assume the following

- the set to be sorted is organized as an array (and not a list)
- data objects are sorted in nondecreasing order of their keys
- the measure of work is the number of comparisons of keys

You know selection and bubble sort; here we study other algorithms

besides the algorithm principles we'll focus on their analysis!

Insertion sort – Strategy



- Traverse the (unsorted) array from left-to-right
- Pick the first element that is not considered so far
- Insert this in sorted (left) part by doing element-wise comparisons
- This algorithm works for other linear structures such as *lists*

Insertion sort – Algorithm

```
def insertionSort(E):
    for j in range(1,len(E)):
        v=E[j]
        i=j-1  #now insert v into sorted E[0,j)
        while i>=0 and E[i]>v:
            E[i+1]=E[i]
            i=i-1
        E[i+1]=v
```

insertion sort is in-place, i.e., does not require extra storage

Analysis of insertion sort – 1

In worst case, each next element to be moved to its new position, ends at the front of the array. This will be the case if the elements appear in reverse order.

- this requires a comparison with all preceding elements
- in worst case, to find the slot for the i-index element, i comparisons are needed

This yields:

$$W(n) = \sum_{i=0}^{n-1}(i) = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$

Analysis of insertion sort – 2

Average case analysis assumptions:

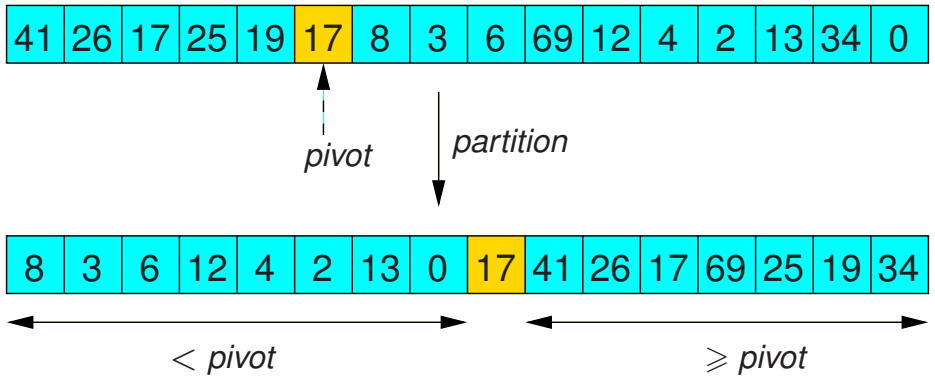
- all permutations of the elements are equally likely to occur
- all elements to be sorted are distinct

$A(n) = \sum_{i=0}^{n-1} X_i$

where $X_i =$ expected # comparisons to find slot for element i

Further calculations (see book) lead to $A(n) \approx \frac{n \cdot (n+3)}{4} - \ln n \quad \in \Theta(n^2)$
Insertion sort is quadratic in worst and average case!

Quicksort – Strategy



- Choose an element from the array to be sorted – called *pivot*
- Partition array in two parts: (i) smaller than and (ii) at least the pivot

  several partitioning strategies do exist
- Sort parts recursively and squeeze pivot inbetween sorted parts

This is a prime example of the divide-and-conquer paradigm! [Hoare 1962]

Quicksort – Algorithm

```
def quickSort(E,left,right):
    if right>left:
        i=partition(E,left,right)    #i is split point
        quickSort(E,left,i-1)        #sort left part
        quicksort(E, i+1,right)      #sort right part
```

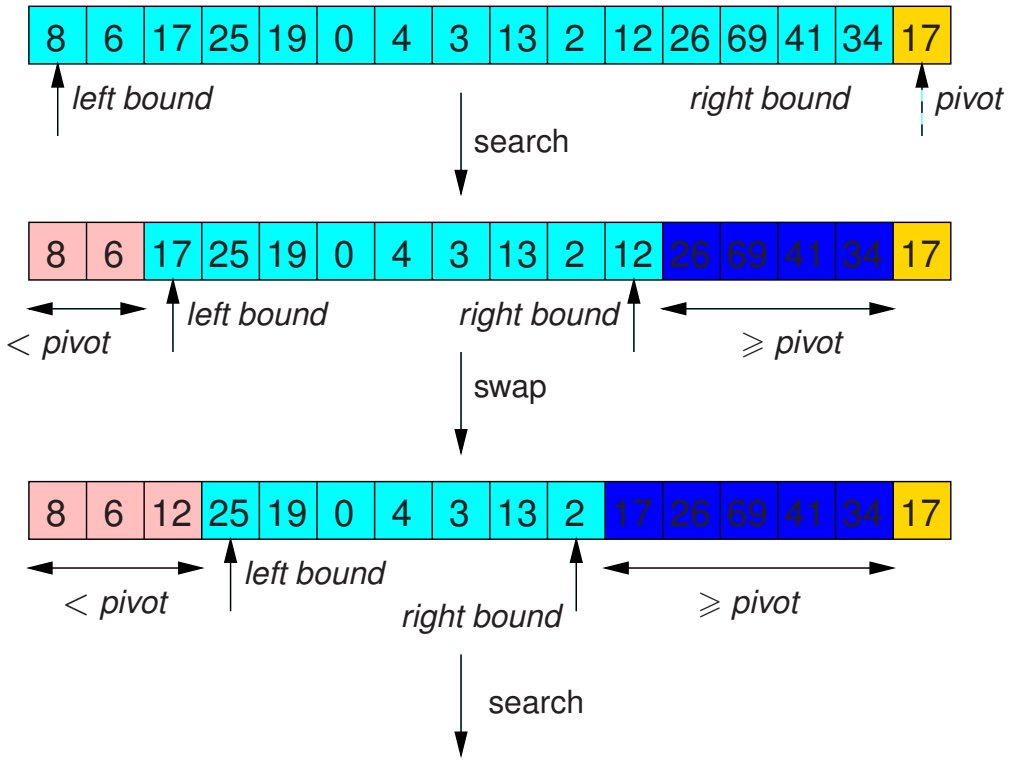
Partitioning for quicksort – 1

Once a pivot is selected, partitioning can be done in $O(n)$, e.g.:

- maintaining 3 regions: $<$ pivot, $\geqslant$ pivot and "unexamined"
- move left bound to the right as long as element $<$ pivot
- move right bound to the left as long as element $\geqslant$ pivot
- swap the two elements encountered on the left and right
- continue until left and right search meet

This partitioning algorithm differs from the one in the book!

Partitioning for quicksort – 2



Partitioning for quicksort – Algorithm

```
def partition{E,left,right):
    i,j=left,right
    pivot=E[right]              #pick some pivot
    while i<j:
        while E[i]<pivot and i<j:
            i=i+1                   #move left bound
        while E[j-1]>=pivot and i<j:
            j=j-1                   #move right bound
        if i<j:
            E[i],E[j-1]=E[j-1],E[i]
            i=i+1
            j=j-1
    if pivot<E[i]:
        E[i],E[right]=E[right],E[i]
    return i
```

Partitioning - correctness invariants

E[left,i)<pivot, E[j, right+1)≥pivot
while ($i < j$)

{ **while** ($E[i] < pivot \;\&\&\; i < j$) $i = i+1$;
i=j or E[i]≥pivot, E[left,i)<pivot

while ($E[j\text{-}1] \geqslant pivot \;\&\&\; i < j$) $j = j-1$;
i=j or E[j-1]<pivot, E[j, right+1)≥pivot

if ($i < j$) E[i]≥pivot, E[j-1]<pivot, i≠j-1
{ *swap*($E[i]$,$E[j\text{-}1]$) $i = i+1$; $j = j-1$ };
E[left,i)<pivot, E[j, right+1)≥pivot

}
E[left,i)<pivot, E[i, right+1)≥pivot

Quicksort – Space usage

At first sight, it looks like an in-place sorting algorithm. It is not

Recursive calls require storage of all the *left* and *right* parameters

In worst-case, *partition* splits only a single element at a time

In worst-case, $\Theta(n)$ stack storage is needed for n elements

Optimization possible (check book!) to obtain $\Theta(\log n)$ storage

Quicksort – Worst-case analysis

In worst case, pivot is the smallest (or largest) element in array

- the splitting into the smaller and larger part is as unbalanced as possible
- one part is empty, whereas the other part contains the remaining elements
- this appears e.g., when the array is already ascending (or descending)!
- this yields $n-1$ levels in the recursion tree

This yields: $W(n) = \sum_{i=0}^{n-1}(i) = \frac{n\cdot(n-1)}{2} \in \Theta(n^2)$

This is as bad as insertion sort, bubble sort, selection sort, and so on

So what is quick about quicksort?

Quicksort – Best-case analysis

Divide-and-conquer works best if division is as equal as possible

- split the array of n elements into two sub-arrays of size $n/2$
- this yields $\log n$ levels in the recursion tree

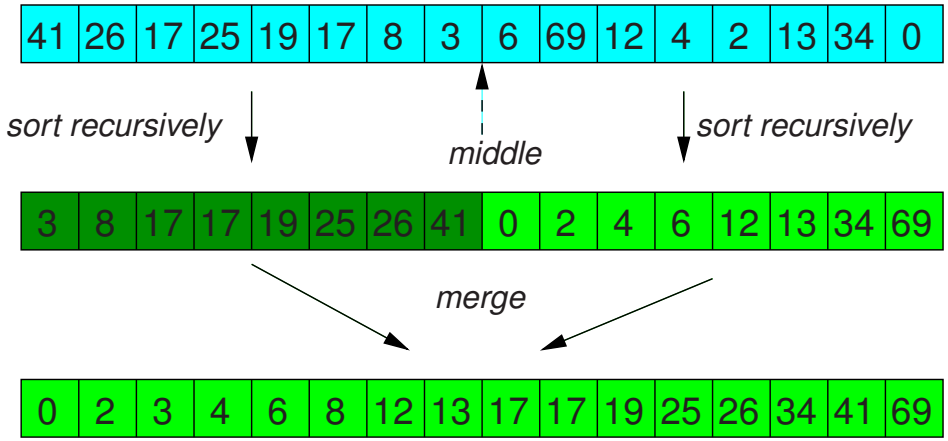
Partitioning is linear in the size; i.e., on each level this is $O(n)$

This yields $B(n) = 2{\cdot}B(n/2) + c{\cdot}n$ for $n > 1$, and $B(1) = 1$

Applying the Master theorem yields: $B(n) \in \Theta(n{\cdot}\log n)$

It appears that this also holds for the average case: $A(n) \in \Theta(n \log n)$

Mergesort – Strategy



- Do an optimal split: divide the array to be sorted in two halves
- Sort parts recursively
- Merge the sorted sub-arrays into a single sorted array

Yet another prime example of the divide-and-conquer paradigm!

Mergesort – Algorithm

```
def mergeSort(E,left,right):
    if right>left:
        mid=(right+left)//2
        mergeSort(E,left,mid)
        mergeSort(E,mid+1,right)
        merge(E,left,mid,right)
```

Merging can be done in linear time; How?

Mergesort – Analysis

For the worst-case behaviour we obtain:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \quad \text{with} \quad W(1) = 1$$

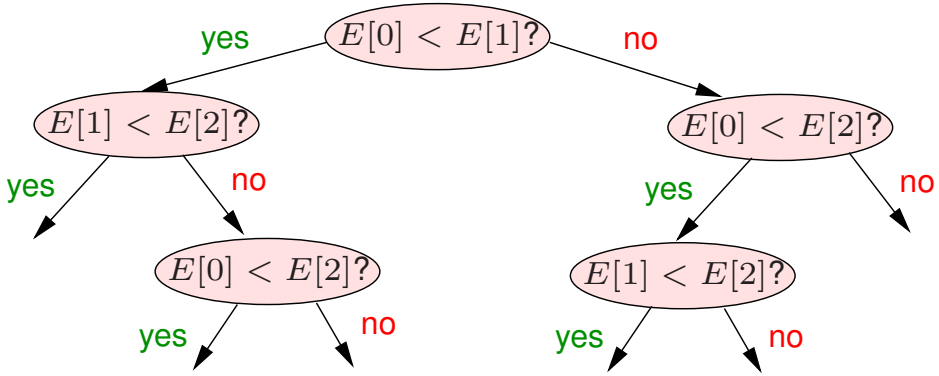
By the Master theorem, this yields $W(n) \in \Theta(n \cdot \log n)$

Average-case behaviour is in $\Theta(n \cdot \log n)$

Space usage: $\Theta(n)$ for the copy of the array at merging

More details can be found in the book

Can sorting be more efficient?



View comparison-based sorting algorithm as a decision tree:

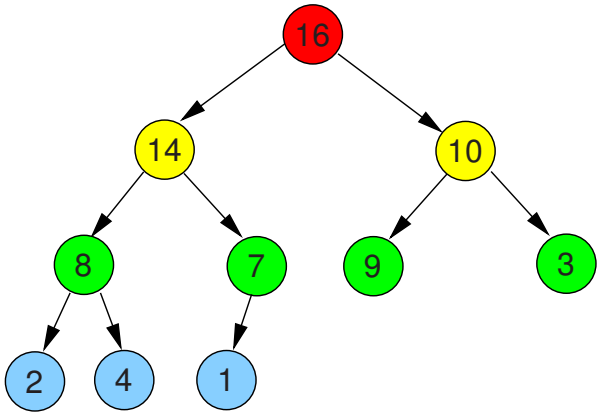
- decision tree describes the sequence of comparisons carried out
- sorting a different permutation of input data yields another path in tree
- # comparisons in worst case = length of longest path = level k of tree
- as only binary comparisons are used it is a binary tree with $n!$ leaves
- $n! \leqslant 2^k$ and thus $k \geqslant \lceil \log(n!) \rceil$ comparisons needed in worst case

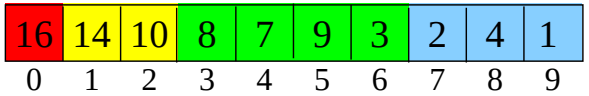
as $\lceil \log(n!) \rceil \approx n \cdot \log n - 1.4 \cdot n$, *we cannot do any better than* $n \cdot \log n$

Complexity of sorting algorithms

Algorithm	*Worst case*	*Average*	*Space usage*
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	in place
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot \log n)$	$\Theta(\log n)$ extra
Mergesort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$\Theta(n)$ extra

Trees in arrays





The elements in an array E can be seen as the nodes of a binary tree.

- $E[0]$ is the root of the tree
- $E[2i+1]$ is the left- and $E[2i+2]$ is the right-child of $E[i]$

Heaps

A binary tree with a representation in an array has special properties.

- all leaves are on at most two adjacent levels
- all levels – except possibly the lowest – are completely filled
- all leaves on the lowest level occur “to the left”

Such a special binary tree is a heap if for the keys in the nodes:

- each node key exceeds that of all its children, or *this is a maxheap*
- the keys of all its children exceed the key of the node. *this is a minheap*

Previous slide shows a maxheap.

Inserting an element in a maxheap

To add an element x to a maxheap H with n elements

- put x into the first (left-most) open position
- if x is greater than its parent, swap them, and repeat at a higher level

Doing such an insertion requires $O(\log n)$ comparisons

- the level k of a heap of n elements is bounded: $n \leqslant 2^{k+1} - 1$
  $\Rightarrow \quad k = \lceil \log(n+1) \rceil - 1$

To re-order an array into a maxheap do not use repeated insertions, but heapify (or fixheap) instead [Floyd 1964]

Heapify – Strategy

Consider $E[i]$ and assume its left and right subtrees are heaps

- but $E[i]$ may be smaller than its children

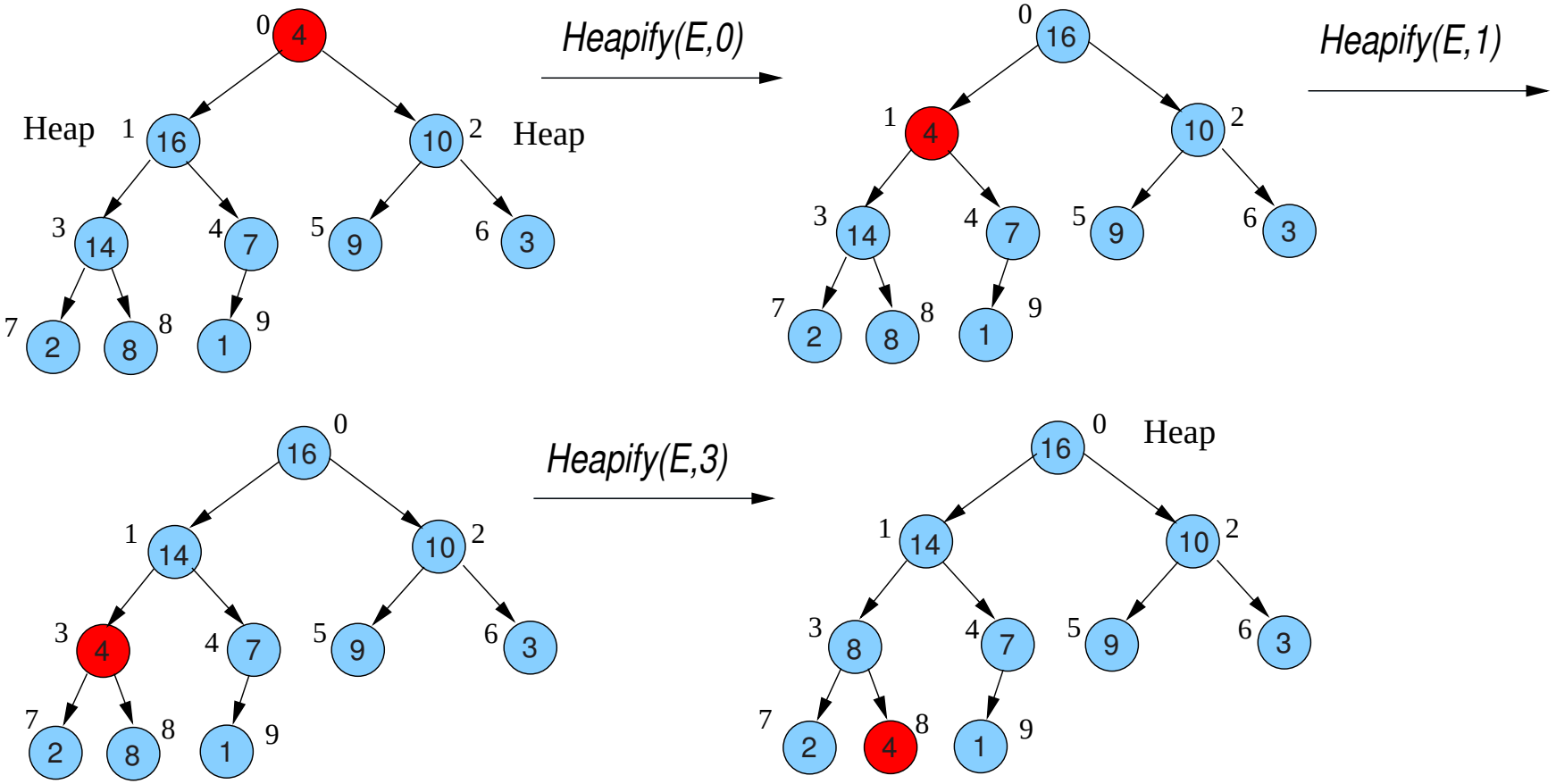
Construct a heap from the left- and right subtrees and $E[i]$, let the value of $E[i]$ "float down" in the structure such that the structure rooted at $E[i]$ is a heap

- determine the maximum of the values $E[i]$ and of its children
- if $E[i]$ is the largest, then the subtree rooted at it is a heap. Done.
- otherwise, swap $E[i]$ with the largest and heapify the corresponding subtree

Heapify – Algorithm

```
def heapify(E,i):
    left,right=2*i+1,2*i+2
    if left<E.heapsize and E[left]>E[i]:
        max=left
    else:
        max=i
    if right<E.heapsize and E[right]>E[max]:
        max=right   # max is index of  max{E[i],E[left],E[right]}
    if max!=i:       #so not a heap
        E[i],E[max]=E[max],E[i]
        heapify(E,max)              # heapify subtree with root max
```

Heapify – Example



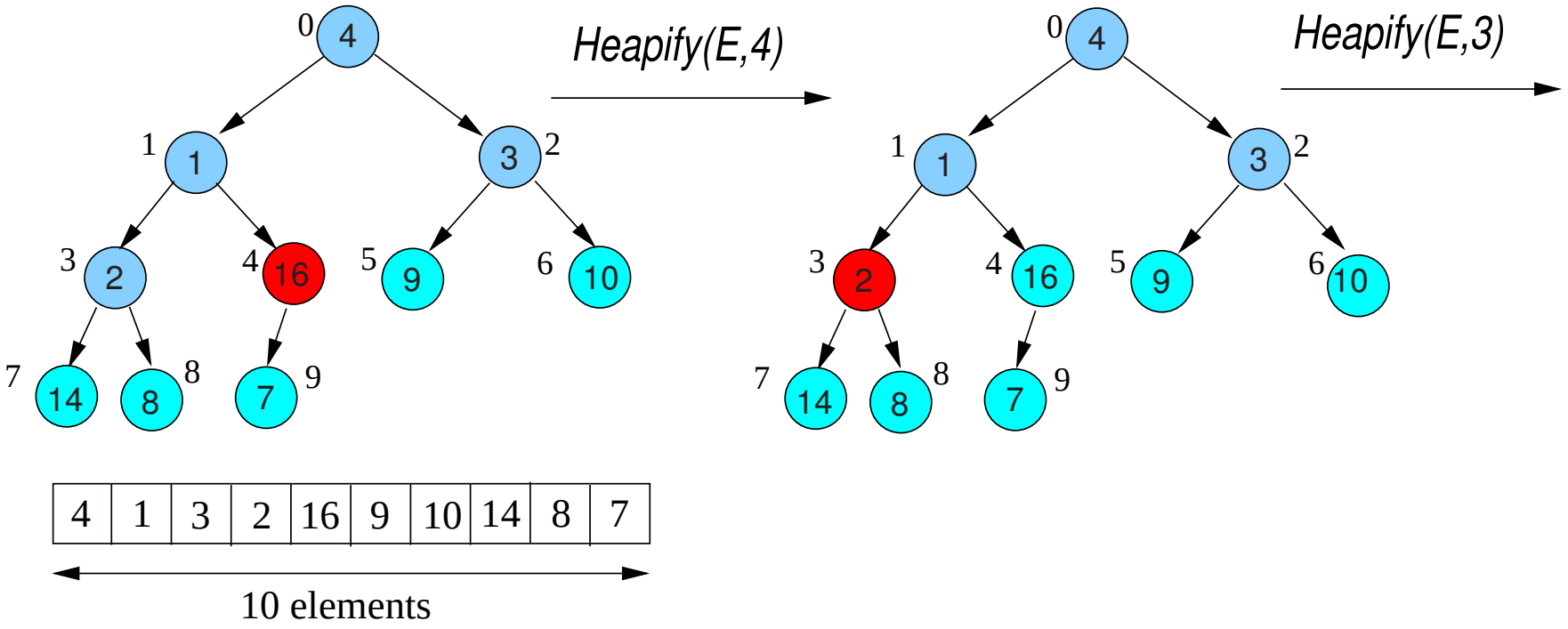
Constructing a maxheap – Algorithm

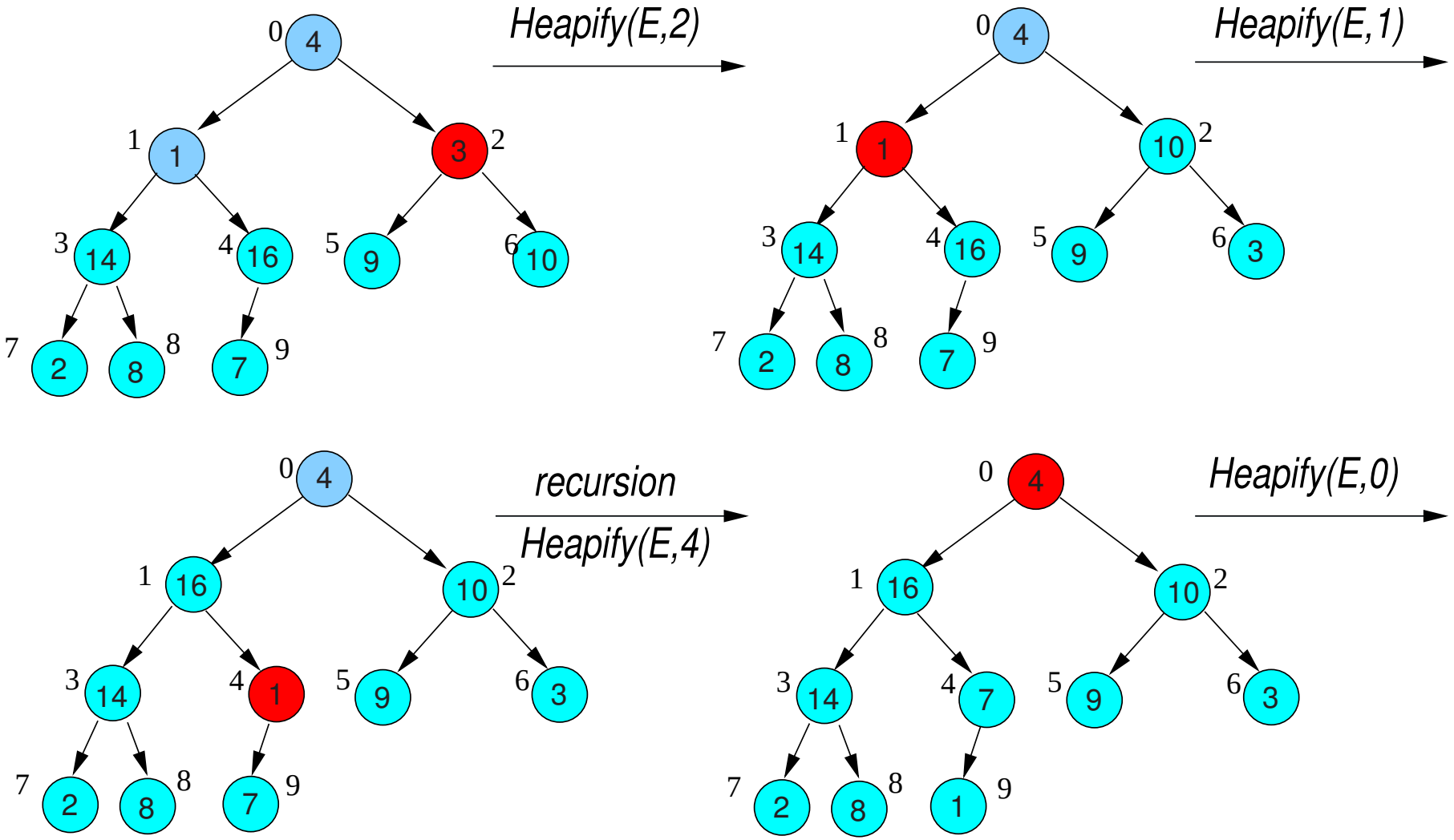
```
def buildHeap(E):
    E.heapsize=len(E)
    lastparent=(len(E)-2)//2
    for i in range(lastparent,-1,-1):
        heapify(E,i)
```

This algorithm can also be written in a recursive way (easier to analyse!):

```
def constructHeap(E,i):
    if 2*i+1<len(E):
        constructHeap(E,2*i+1)
    if 2*1+2<len(E):
        constructHeap(E,2*i+2)
    heapify(E,i)
```

Constructing a heap – Example




Heapify(E,2)
Heapify(E,1)
recursion
Heapify(E,4)
Heapify(E,0)

Constructing a maxheap – Complexity intuition

constructHeap splits the problem of creating a heap of n elements into two subproblems of creating heaps each with half the number of elements, and applies *Heapify* to merge the two into a single maxheap in $\log n$ steps.

$$W(n) = 2 \cdot W(n/2) + \log n$$

Apply the Master theorem!

Constructing a maxheap – Complexity intuition

constructHeap splits the problem of creating a heap of n elements into two subproblems of creating heaps each with half the number of elements, and applies *Heapify* to merge the two into a single maxheap in $\log n$ steps.

$$W(n) = 2 \cdot W(n/2) + \log n$$

Apply the Master theorem!

$b = c = 2$, so $E = 1$; and $f(n) = \log n$, so $f(n) \in O(n^{E-0.5})$. The theorem says:

$$W(n) \in \Theta(n)$$

Priority queues

Consider objects that are equipped with a key (or, priority)

- assume each key is associated to at most one data object

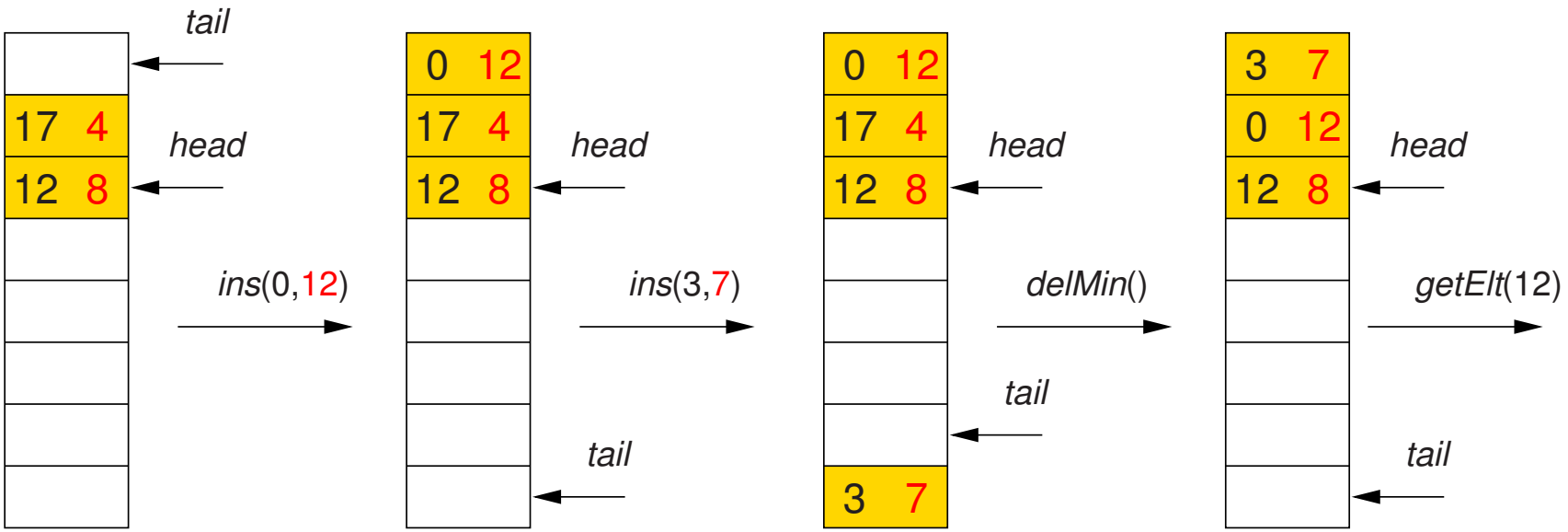
Objects are ordered according to their priority

A priority queue pq stores a collection of such objects and supports:

- *isEmpty*( ) returns true if pq is empty, and false otherwise
- *insert*(e,k) insert element e with key k into the queue pq
- *getMin*( ) returns the element with smallest key; requires non-empty pq
- *delMin*( ) deletes the element with smallest key; requires non-empty pq
- *getElt*(k) returns the object with key k in pq; requires k to be in pq
- *decrKey*(e, k) sets key of e to k; requires e in pq and $k <$ *getKey*(pq, e)

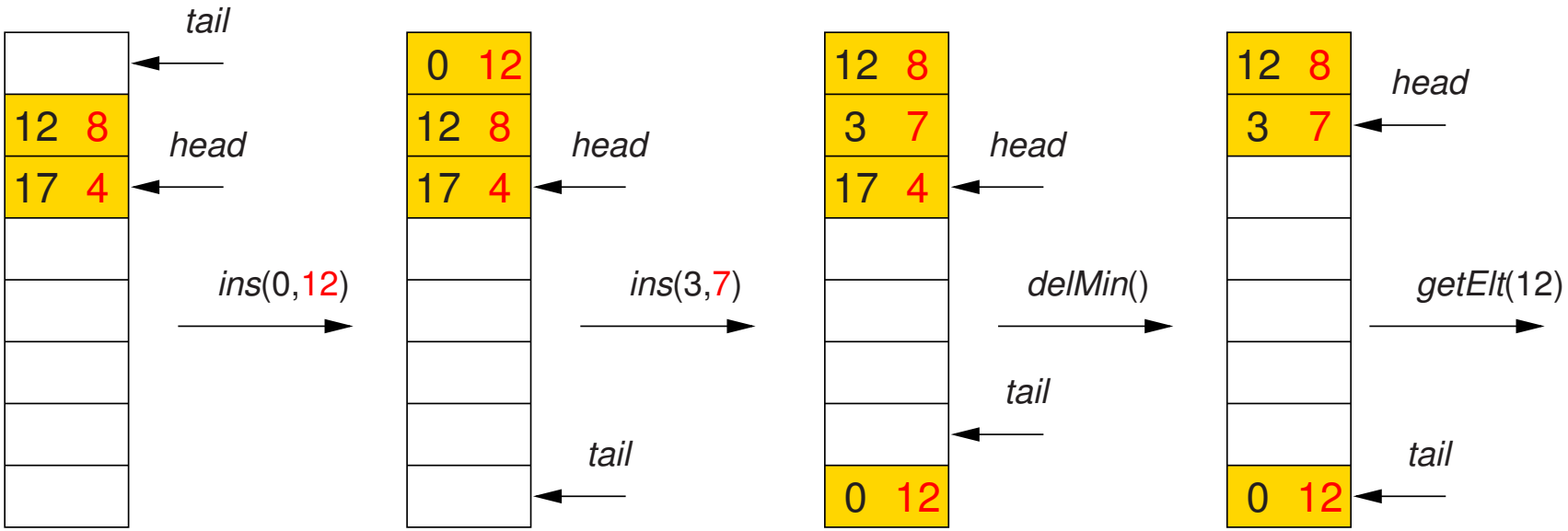
can be very efficiently implemented using heaps!

Priority queue – Unsorted bounded array implementation



black = element; red = key

Priority queue – Sorted bounded array implementation



Comparing priority queue implementations

Implementation / *Operation*	Unsorted array	Sorted array	Heap
isEmpty( )	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert(e, k)	$\Theta(1)$	$\Theta(n)^*$	$\Theta(\log n)$
getMin( )	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delMin( )	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
getElt(k)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
decrKey(e, k)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$

* this includes shifting all elements “to the right” of k

Heapsort – Algorithm

Idea: you know where the maximum element of a (max)heap is.

```
def heapSort(E):
    buildHeap(E)
    for i in range(len(E)-1,0,-1):
        E[0],E[i]=E[i],E[0]
        E.heapsize=heapsize-1
        heapify(E,0)
```

This algorithm sorts array E in nondecreasing order.

Heapsort – Complexity analysis

Worst case complexity of *Heapify* is at most $\lfloor 2 \cdot \log n \rfloor$ for n nodes

The worst case complexity of *buildHeap* is $\Theta(n)$

This yields for heapsort:

$$W(n) = \sum_{i=1}^{n-1} 2 \cdot \lfloor \log i \rfloor \leqslant 2 \cdot \int_1^n (\log e) \ln x \, dx = 2 \cdot n \cdot \log n + c_1 \cdot n + c_2$$

No extra space is needed (tail recursion in *Heapify* can be removed):

heapsort sorts in $\Theta(n \cdot \log n)$ and sorts in-place

Complexity of sorting algorithms

Algorithm	*Worst case*	*Average*	*Space usage*
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	in place
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot \log n)$	$\Theta(\log n)$ extra
Mergesort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$\Theta(n)$ extra
Heapsort	$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$	in place