



Theory of algorithms (2nd lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- Heap data structure
 - Maintaining the heap property
 - Building a heap
 - Heapsort
 - Priority queues
- Sorting in linear time
 - Counting sort
 - Radix sort
 - Bucket sort
- Exercises



Heap data structure

PARENT(i)

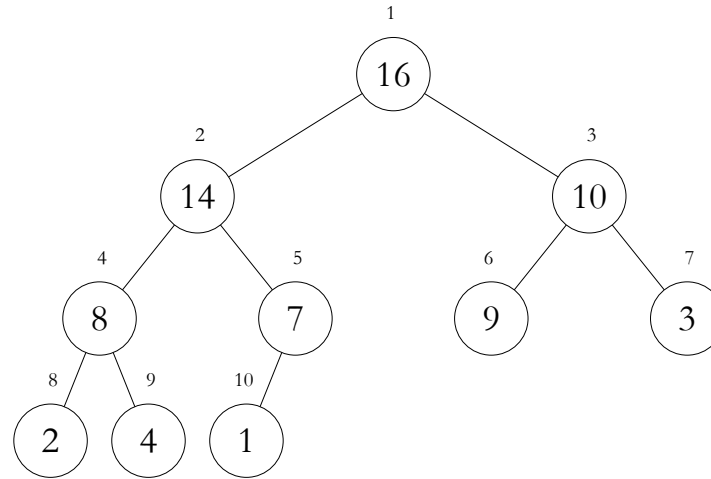
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$



Max-heap property:

$A[\text{PARENT}(i)] \geq A[i], i > 1$

Min-heap property:

$A[\text{PARENT}(i)] \leq A[i], i > 1$

A

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

**A max-heap as a binary tree
and as an array**

Exercises

- Is the max-heap property satisfied with array A ? If not, how many pairs of elements do violate it?
 - $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$
- Give a min-heap data structure that contains the elements of array A !
 - $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$



Maintaining the heap property

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else
6       $\text{largest} = i$ 
7  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
8       $\text{largest} = r$ 
9  if  $\text{largest} \neq i$ 
10     exchange  $A[i]$  with  $A[\text{largest}]$ 
11     MAX-HEAPIFY( $A, \text{largest}$ )
```

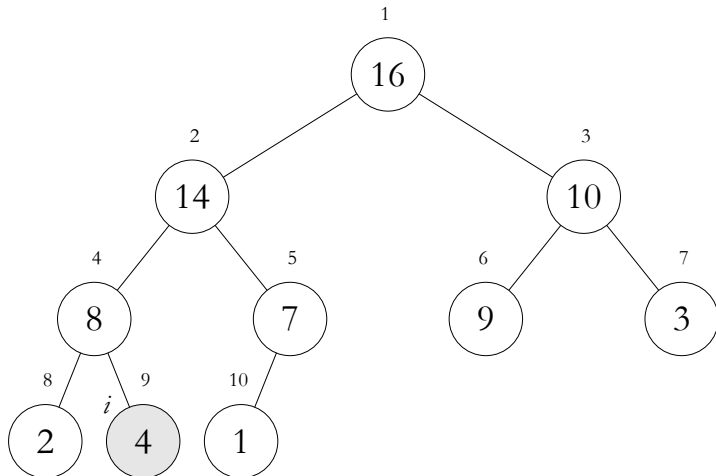
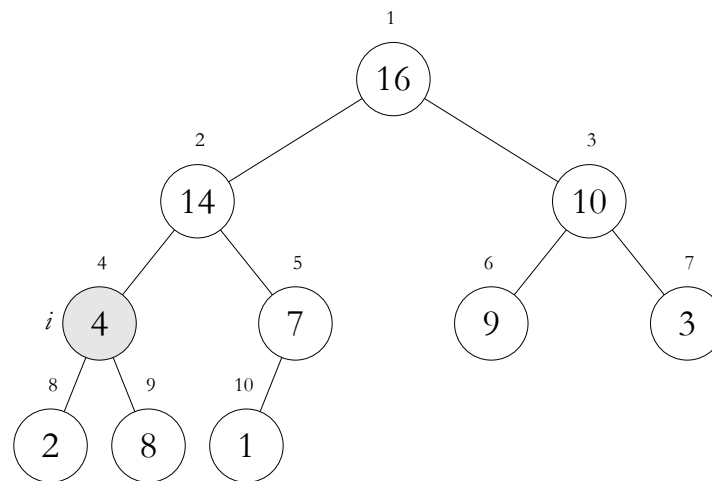
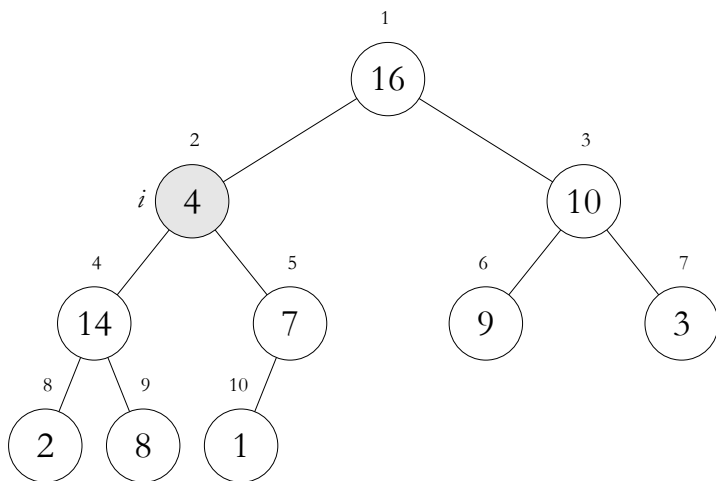
Criteria: The binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property.

MAX-HEAPIFY lets the value $A[i]$ „float down” in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Efficiency: The running time is $O(h)$, where h is the height of the tree ($h = O(\lg n)$, $n = A.\text{heap.size}$).



Maintaining the heap property



The operation of MAX-HEAPIFY(A , 2)

Exercises

- Illustrate the operation of MAX-HEAPIFY($A, 3$) call with array A ? Give the contents of the array as a binary tree after each exchange.
 - $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$



Building a heap

BUILD-MAX-HEAP(A)

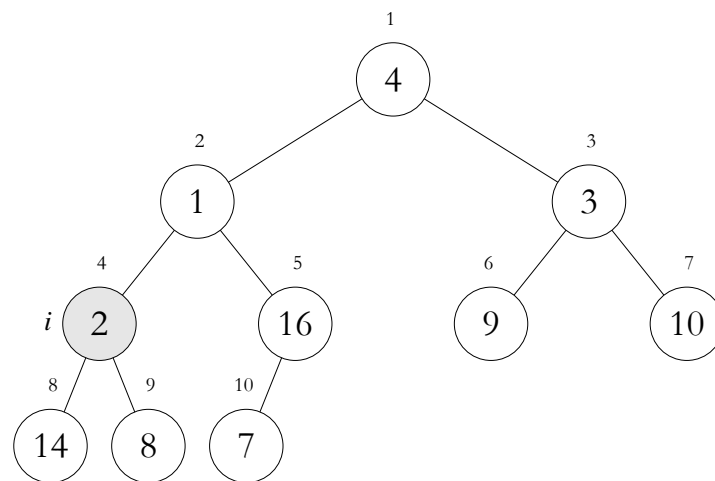
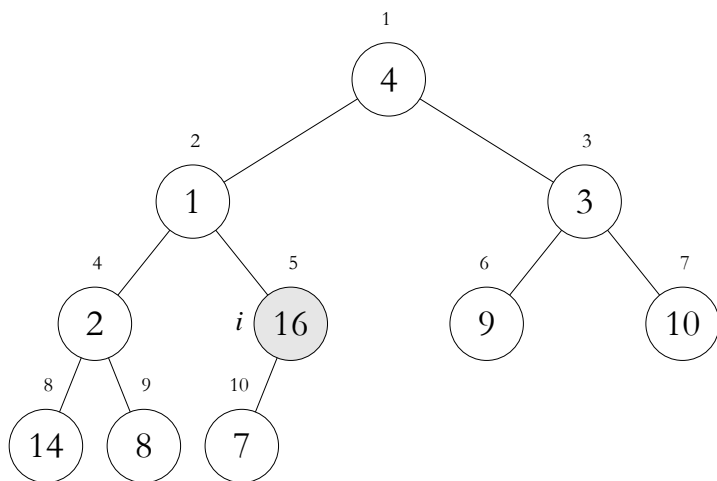
```
1   $A.heap-size = A.length$   
2  for  $i = \lfloor A.length / 2 \rfloor$  downto 1  
3      MAX-HEAPIFY( $A, i$ )
```

Efficiency: The „simple estimated” $O(n \lg n)$ upper bound is not asymptotically tight, the $O(n)$ running time can be proved, thus we can build a max-heap from an unsorted array in linear time.

Building a heap

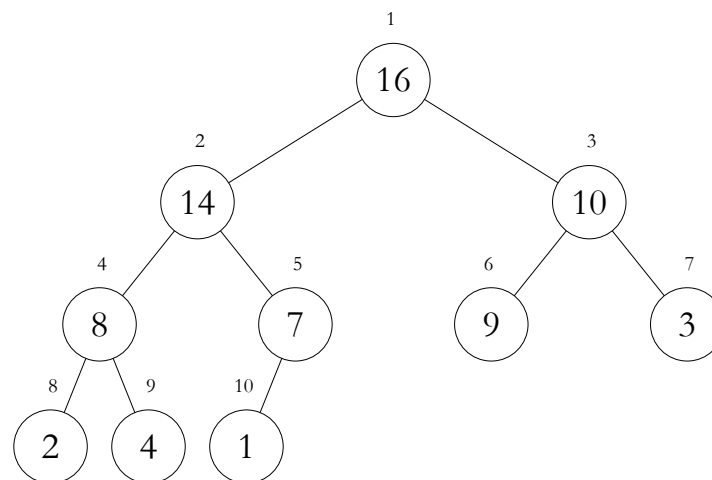
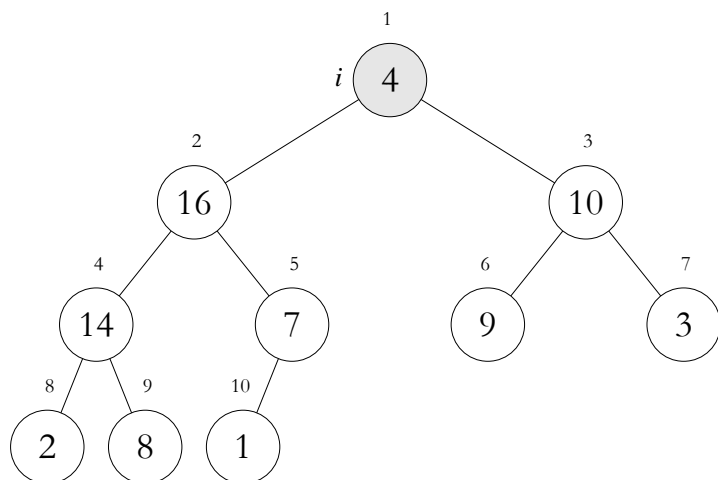
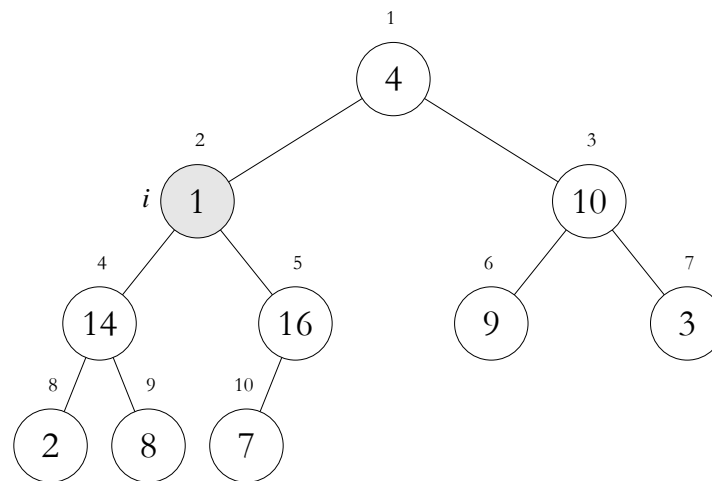
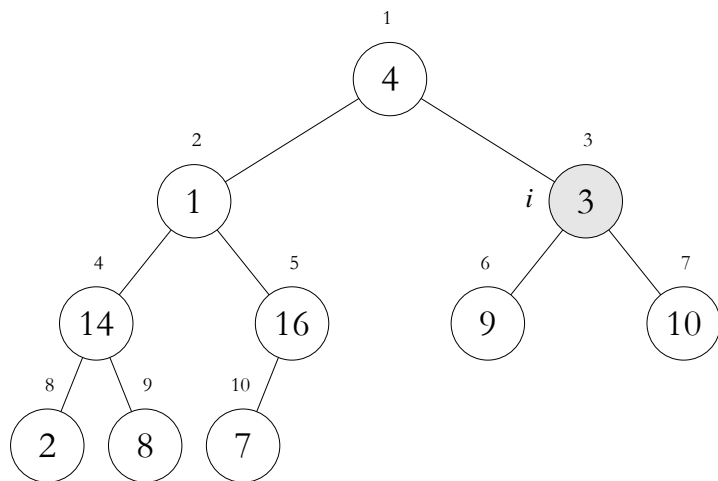
A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



The operation of BUILD-MAX-HEAP

Building a heap



The operation of BUILD-MAX-HEAP

Exercises

- Illustrate the operation of BUILD-MAX-HEAP with array A . Give the contents of the array as a binary tree after each step of the iteration.
 - $A = \langle 8, 2, 1, 5, 6, 9, 4, 3, 7 \rangle$



Heapsort

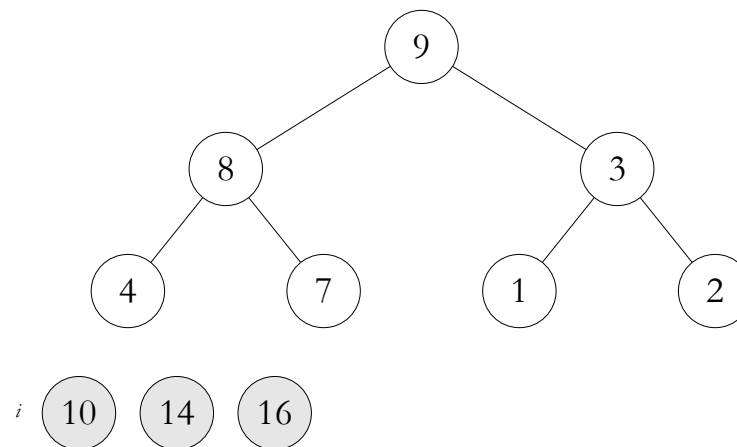
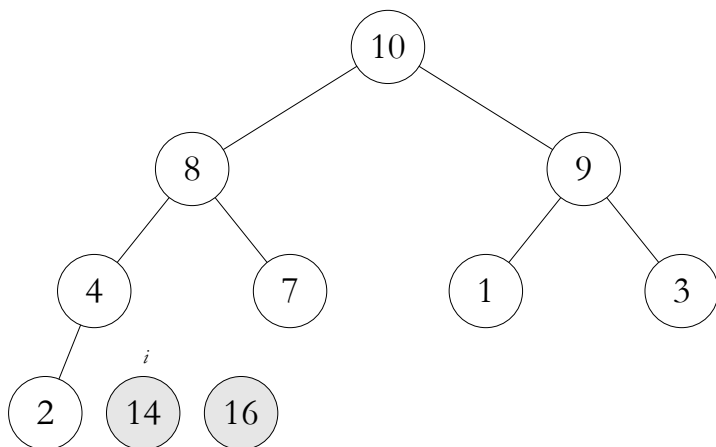
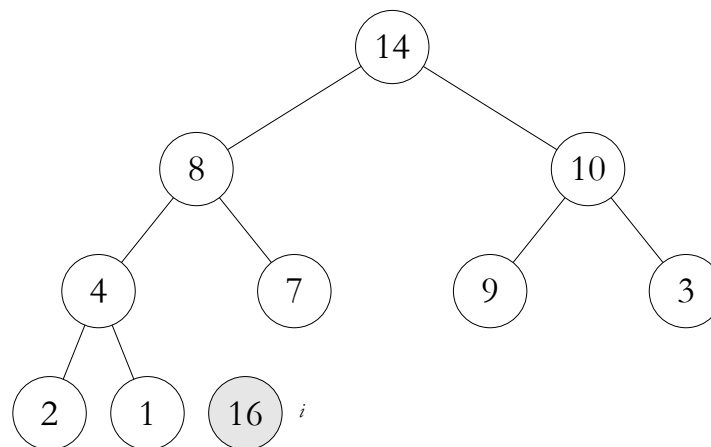
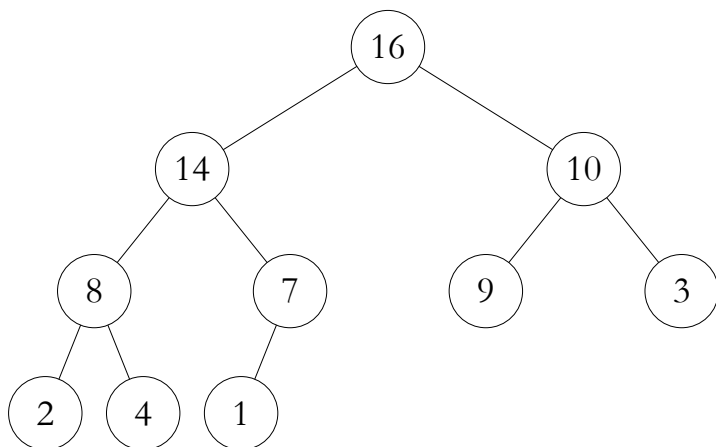
HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```

Efficiency: Since the call to BUILD-MAX-HEAP takes time $O(n)$ and MAX-HEAPIFY takes $O(\lg n)$ the running time is $O(n \lg n)$.



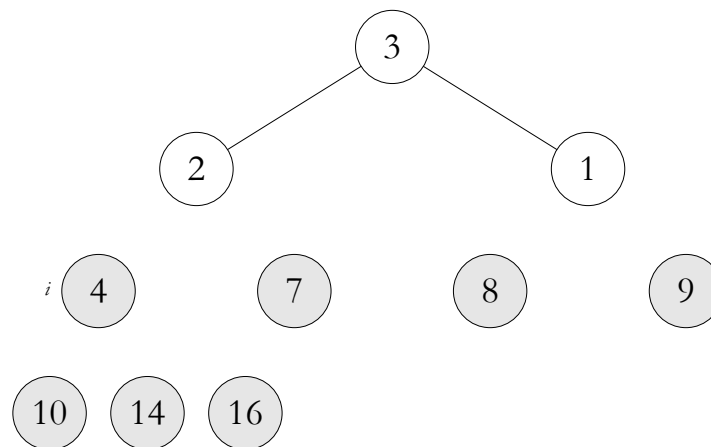
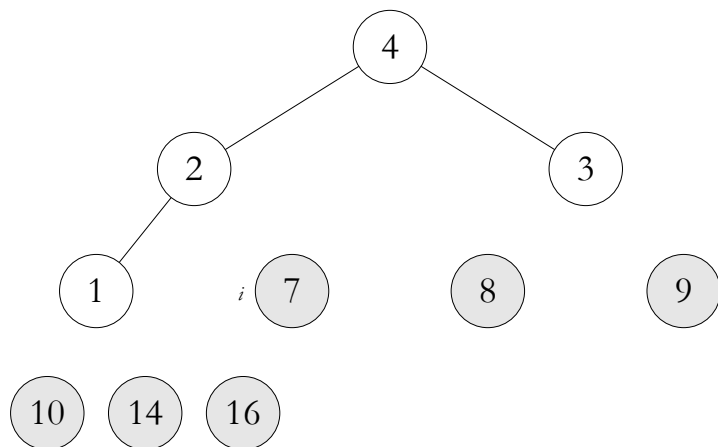
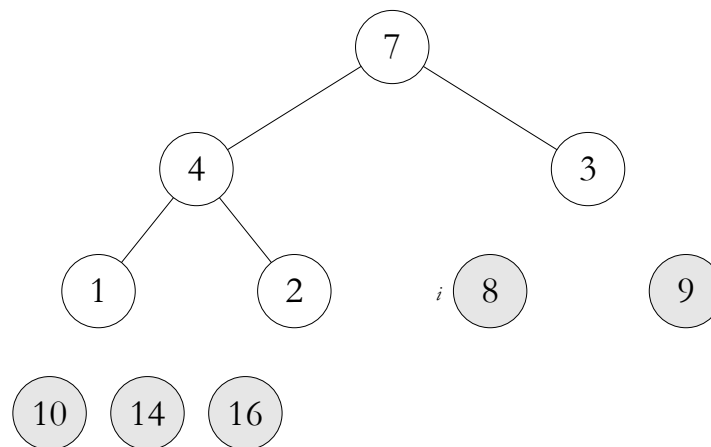
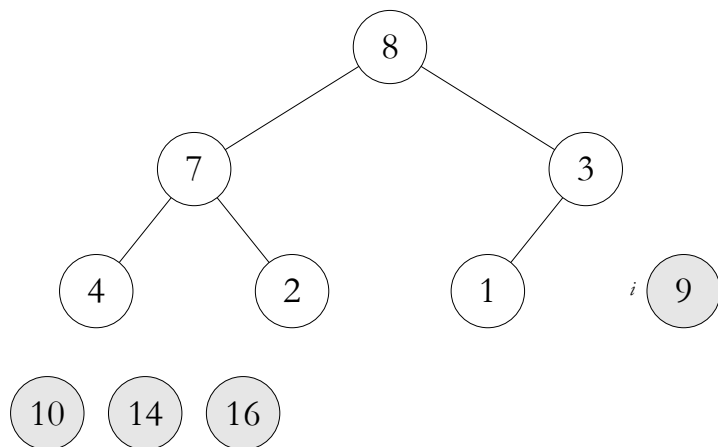
Heapsort



The operation of HEAPSORT

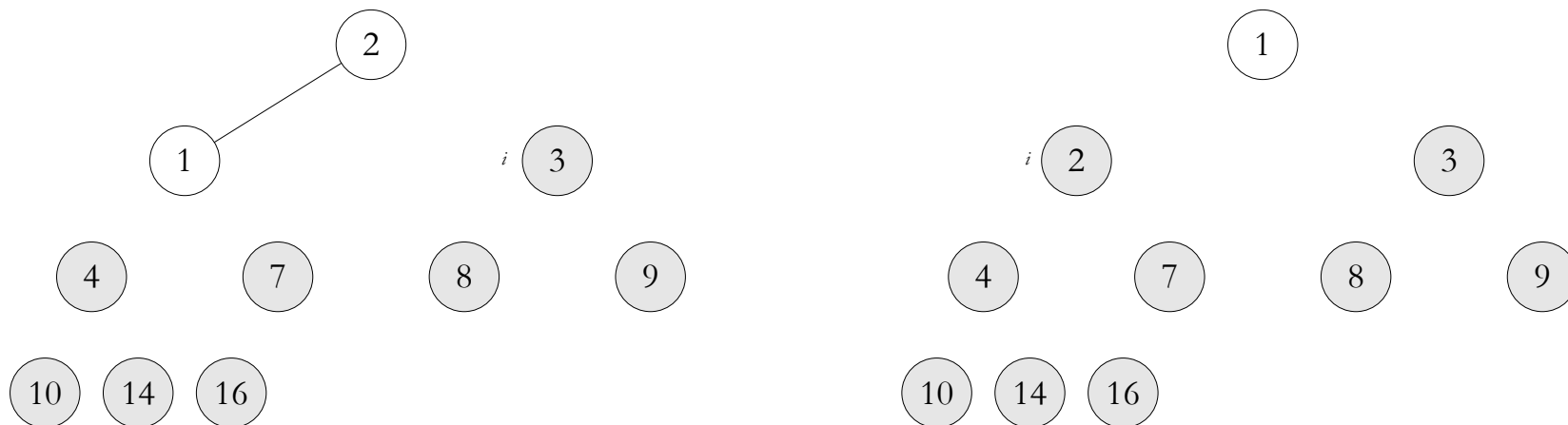


Heapsort



The operation of HEAPSORT

Heapsort



A

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

The operation of HEAPSORT

Exercises

- Illustrate the operation of HEAPSORT with array A . Give the contents of the array as a binary tree after the exchanges of BUILD-MAX-HEAP and after each step of the iteration of HEAPSORT.
 - $A = \langle 8, 2, 1, 5, 6, 9, 4, 3, 7 \rangle$



Priority queues

- A **priority queue** is a data structure for maintaining a set S of elements, each with an associated value called a **key**.
- A **max-priority queue** supports the following **operations**:
 - $\text{INSERT}(S, x)$ inserts the element x into the set S .
 - $\text{MAXIMUM}(S)$ returns the element of S with the largest key.
 - $\text{EXTRACT-MAX}(S)$ removes and returns the element of S with the largest key.
 - $\text{INCREASE-KEY}(S, x, k)$ increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

$\text{HEAP-MAXIMUM}(A)$

```
1  return A[1]
```

Efficiency: The running time is $\Theta(1)$.



Priority queues

HEAP-EXTRACT-MAX(A)

```
1  if  $A.heap-size < 1$ 
2      error „heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.heap-size]$ 
5   $A.heap-size = A.heap-size - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  return  $max$ 
```

Efficiency: The running time is $O(\lg n)$, due to $O(\lg n)$ time of MAX-HEAPIFY.



Priority queues

HEAP-INCREASE-KEY(A, i, key)

```
1  if  $key < A[i]$ 
2      error „new key is smaller than current key”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[PARENT(i)] < A[i]$ 
5      exchange  $A[i]$  with  $A[PARENT(i)]$ 
6       $i = PARENT(i)$ 
```

MAX-HEAP-INSERT(A, key)

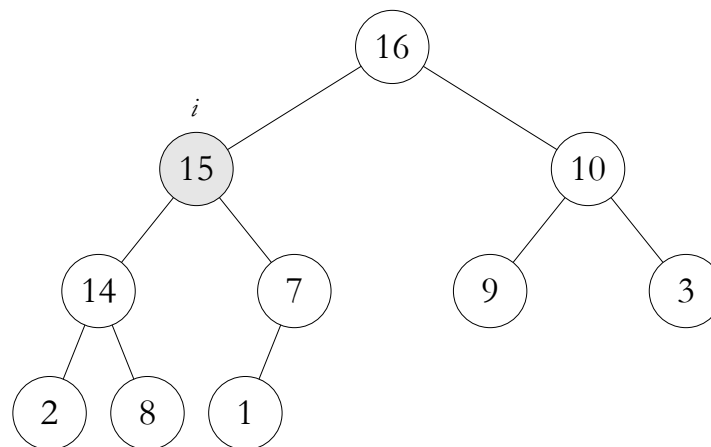
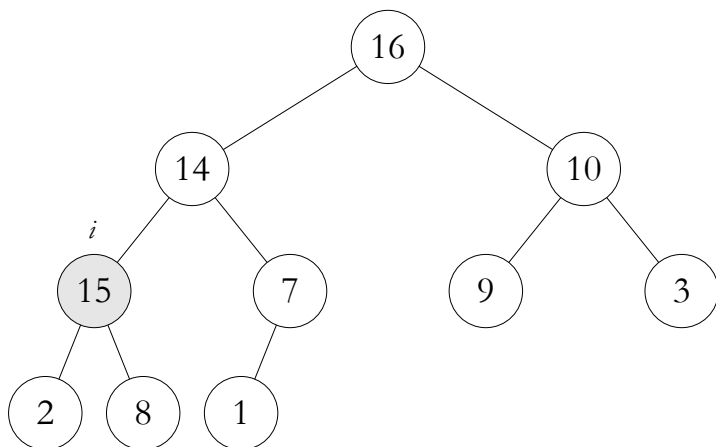
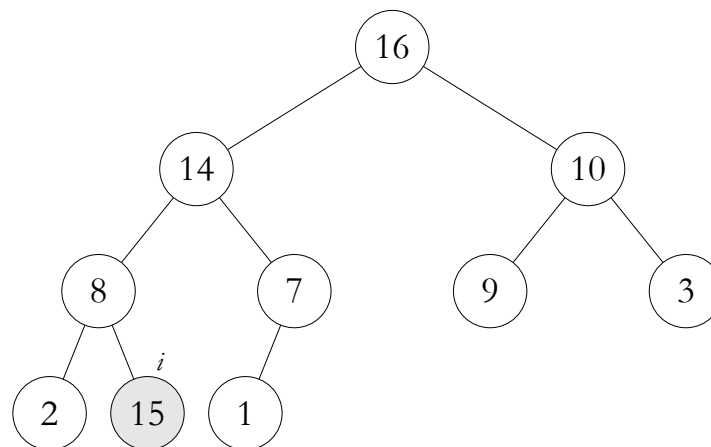
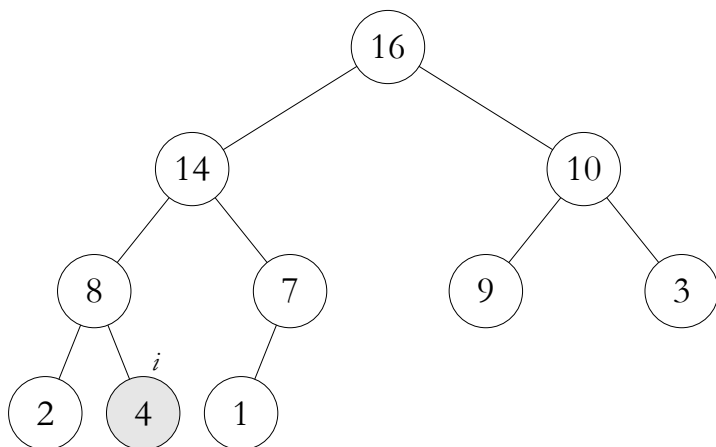
```
1   $A.heap-size = A.heap-size + 1$ 
2   $A[A.heap-size] = -\infty$ 
3  HEAP-INCREASE-KEY( $A, A.heap-size, key$ )
```

Efficiency: The running time of both procedures on an n -element heap is $O(\lg n)$.

Corollary: A heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.



Priority queues



The operation of HEAP-INCREASE-KEY(A, 9, 15)

Priority queues

BUILD-MAX-HEAP'(A)

```
1  A.heap-size = 1
2  for i = 2 to A.length
3      MAX-HEAP-INSERT(A, A[i])
```

Efficiency: The running time to build an n -element heap is $O(n \lg n)$.



Exercises

- Illustrate the operation of `MAX-HEAP-INSERT(A, 20)` call with array A . Give the contents of the array as a binary tree after each exchange.
 - $A = \langle 16, 15, 10, 8, 14, 9, 3, 2, 4, 1, 7 \rangle$
- Illustrate the operation of `BUILD-MAX-HEAP'` with array A . Give the contents of the actual max-heap as a binary tree after each step of the iteration.
- Do the procedures `BUILD-MAX-HEAP` and `BUILD-MAX-HEAP'` give the same results?

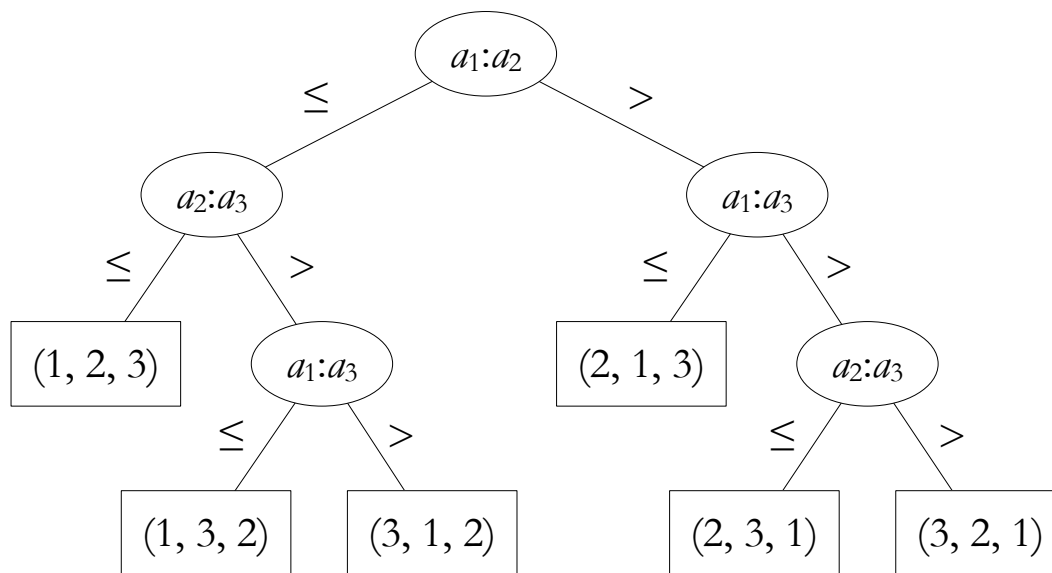


Comparison sorts

We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

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

Corollary: Heapsort and merge sort are asymptotically optimal comparison sorts.



**The decision tree for insertion sort
operating on three elements**

Sorting in linear time

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ 
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i-1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ 
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

Criteria: Each of the n input elements is an integer in the range 0 to k , for some integer k .

Efficiency: When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Stability: The counting sort is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array.



Sorting in linear time

	1	2	3	4	5	6	7	8
<i>A</i>	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
<i>C</i>	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
<i>B</i>							3	

	0	1	2	3	4	5
<i>C</i>	2	2	4	6	7	8

	1	2	3	4	5	6	7	8
<i>B</i>		0				3	3	

	0	1	2	3	4	5
<i>C</i>	1	2	4	5	7	8

	0	1	2	3	4	5
<i>C</i>	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
<i>B</i>		0					3	

	0	1	2	3	4	5
<i>C</i>	1	2	4	6	7	8

	1	2	3	4	5	6	7	8
<i>B</i>	0	0	2	2	3	3	3	5

The operation of COUNTING-SORT

Exercises

- Illustrate the operation of COUNTING-SORT on array A . What elements are in array C after three elements are put into array B ?
 - $A = \langle 6, 5, 2, 6, 1, 3, 6, 2, 7, 5 \rangle$
- What will happen if we rewrite the line 10 of the COUNTING-SORT with the line below?
10 **for** $j = 1$ **to** $A.length$



Sorting in linear time

RADIX-SORT(A, d)

```

1  for  $i = 1$  to  $d$ 
2      use a stable sort to sort array  $A$  on digit  $i$ 

```

Criteria: Each element in the n -element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

329	720	720	329
457	355	329	355
657	436	436	436
839	⇒ 457	⇒ 839	⇒ 457
436	657	355	657
720	329	457	720
355	839	657	839
	↑	↑	↑

The operation of RADIX-SORT



Exercises

- Illustrate the operation of RADIX-SORT on the following list of English words. Give the contents of the array after each step of the iteration.
 - cow, dog, sea, row, box, bar, ear, dig, big, tea, now, fox



Sorting in linear time

BUCKET-SORT(A)

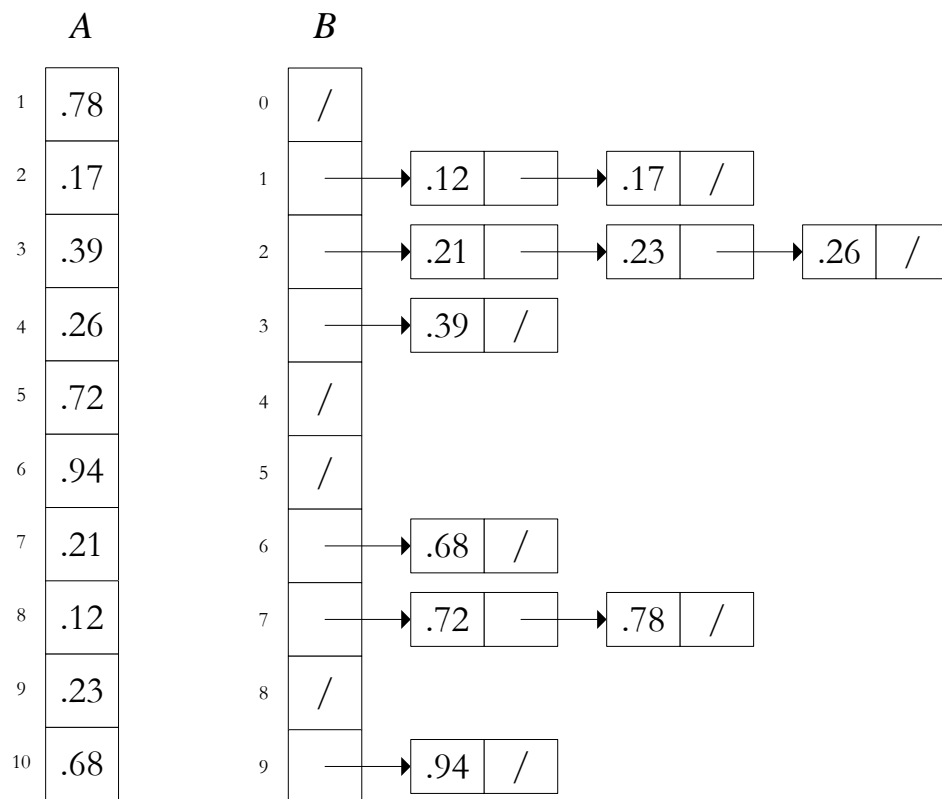
```
1   $n = A.length$ 
2  let  $B[0..n-1]$  be a new array
3  for  $i = 0$  to  $n-1$ 
4      make  $B[i]$  an empty list
5  for  $i = 1$  to  $n$ 
6      insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$ 
7  for  $i = 0$  to  $n-1$ 
8      sort list  $B[i]$  with insertion sort
9  concatenate the lists  $B[0], B[1], \dots, B[n-1]$  together in order
```

Criteria: The input n elements are uniformly distributed numbers in the interval $[0, 1)$.

Efficiency: The **average-case** running time is $\Theta(n)$ due to **expected** running time of line 8 is $O(1)$.



Sorting in linear time



The operation of BUCKET-SORT for $n=10$

Exercises

- Illustrate the operation of BUCKET-SORT on the array A . Give the result of the procedure. Which element is the last element of the „biggest” bucket?
 - $A = \langle 0.65, 0.52, 0.23, 0.68, 0.12, 0.38, 0.61, 0.29, 0.72, 0.53 \rangle$



Summary

Algorithm	Worst-case	Average-case/expected
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(n+k)$	$\Theta(n+k)$
Radix sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Efficiency of the sorting algorithms

