

Data Structures and Algorithms

Week 4

1. About sorting algorithms
2. Heapsort
 - Complete binary trees
 - Heap data structure
3. Quicksort
 - a popular algorithm
 - very fast on average

Previous Week

- Divide and conquer
- Merge sort
- Tiling
- Recurrences
 - repeated substitutions
 - substitution
 - master method
- Example recurrences

DSA09

M. Böhlen

2

Why Sorting

- “**When in doubt, sort**” – one of the principles of algorithm design.
- Sorting is used as a subroutine in many algorithms:
 - Searching in databases: we can do binary search on sorted data
 - Element uniqueness, duplicate elimination
 - A large number of computer graphics and computational geometry problems.

DSA09

M. Böhlen

3

Why Sorting/2

- Sorting algorithms represent different algorithm design techniques.
- The lower bound for sorting $\Omega(n \log n)$ is used to prove lower bounds of other problems.

DSA09

M. Böhlen

4

Sorting Algorithms so far

- Insertion sort, selection sort, bubble sort
 - Worst-case running time $\Theta(n^2)$
 - In-place
- Merge sort
 - Worst-case running time $\Theta(n \log n)$
 - Requires additional memory $\Theta(n)$

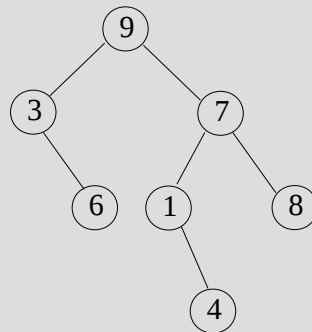
Selection Sort

```
SelectionSort(A[1..n]):  
  for i := 1 to n-1  
  A: Find the smallest element among A[i..n]  
  B: Exchange it with A[i]
```

- A takes $\Theta(n)$ and B takes $\Theta(1)$: $\Theta(n^2)$ in total
- Idea for improvement: smart data structure to
 - do A and B in $\Theta(1)$
 - spend $O(\log n)$ time per iteration to maintain the data structure
 - get a total running time of $O(n \log n)$

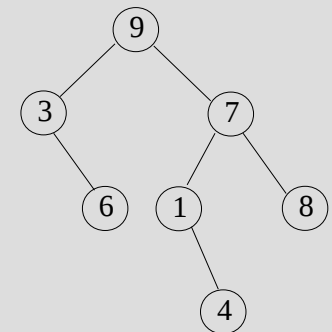
Binary Trees

- Each node may have a left and right **child**.
 - The left child of 7 is 1
 - The right child of 7 is 8
 - 3 has no left child
 - 6 has no children
- Each node has at most one **parent**.
 - 1 is the parent of 4
- The **root** has no parent.
 - 9 is the root
- A **leaf** has no children.
 - 6, 4 and 8 are leaves



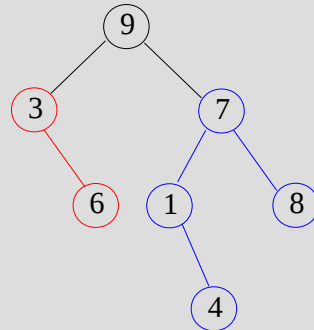
Binary Trees/2

- The **depth** (or **level**) of a node x is the length of the path from the root to x.
 - The depth of 1 is 2
 - The depth of 9 is 0
- The **height** of a node x is the length of the longest path from x to a leaf.
 - The height of 7 is 2
- The height of a tree is the height of its root.
 - The height of the tree is 3



Binary Trees/3

- The right subtree of a node x is the tree rooted at the right child of x .
 - The right subtree of 9 is the tree shown in blue.
- The left subtree of a node x is the tree rooted at the left child of x .
 - The left subtree of 9 is the tree shown in red.



Complete Binary Trees

- A **complete binary tree** is a binary tree where
 - all leaves have the same depth.
 - all internal (non-leaf) nodes have two children.
- A **nearly complete binary tree** is a binary tree where
 - the depth of two leaves differs by at most 1.
 - all leaves with the maximal depth are as far left as possible.

Heaps

- A binary tree is a **binary heap** iff
 - it is a nearly complete binary tree
 - each node is greater than or equal to all its children
- The properties of a binary heap allow
 - an efficient storage as an array (because it is a nearly complete binary tree)
 - a fast sorting (because of the organization of the values)

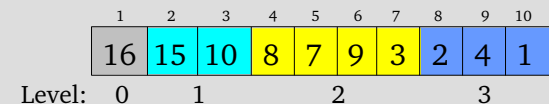
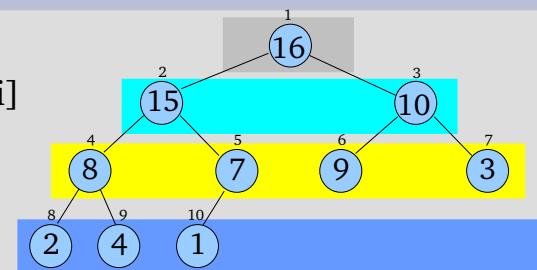
Heaps/2

Heap property
 $A[\text{Parent}(i)] > A[i]$

Parent(i)
 return $\lfloor i/2 \rfloor$

Left(i)
 return $2i$

Right(i)
 return $2i+1$



Heapify: Running Time

- The running time of Heapify on a subtree of size n rooted at i includes the time to
 - determine relationship between elements: $\Theta(1)$
 - run Heapify on a subtree rooted at one of the children of i
 - $2n/3$ is the worst-case size of this subtree (half filled bottom level)
 - $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log n)$
- Alternatively
 - Running time on a node of height h : $O(h) = O(\log n)$

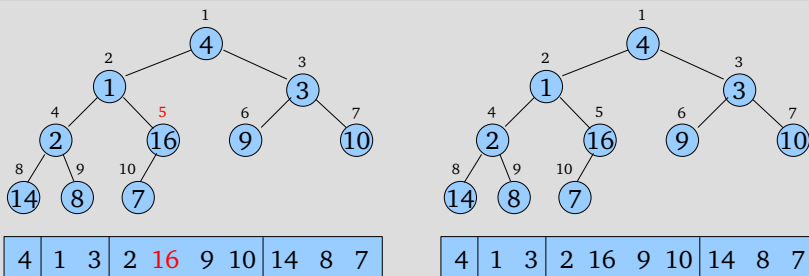
Building a Heap

- Convert an array $A[1\dots n]$ into a heap.
- Notice that the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)\dots n]$ are 1-element heaps to begin with.

```

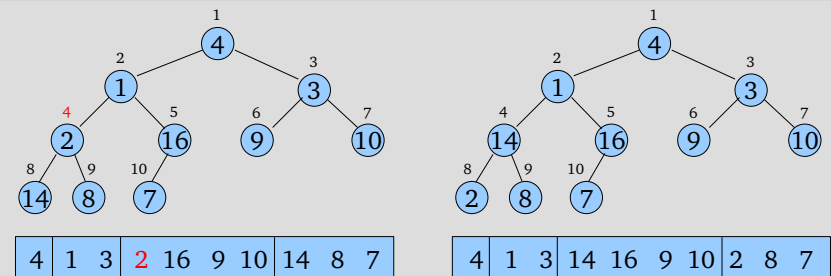
BuildHeap(A)
  for i := ⌊n/2⌋ to 1 do
    Heapify(A, i, n)
    
```

Building a Heap/2



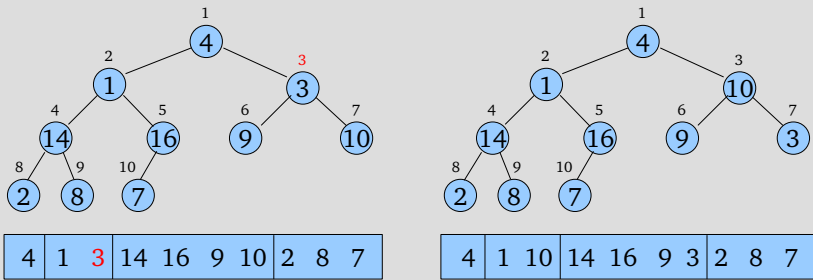
- Heapify($A, 7, 10$)
- Heapify($A, 6, 10$)
- Heapify($A, 5, 10$)

Building a Heap/3



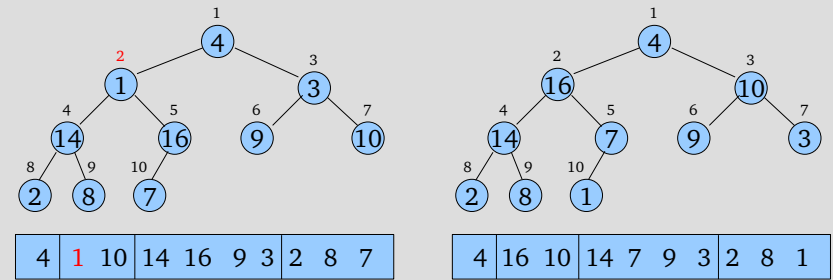
- Heapify($A, 4, 10$)

Building a Heap/4



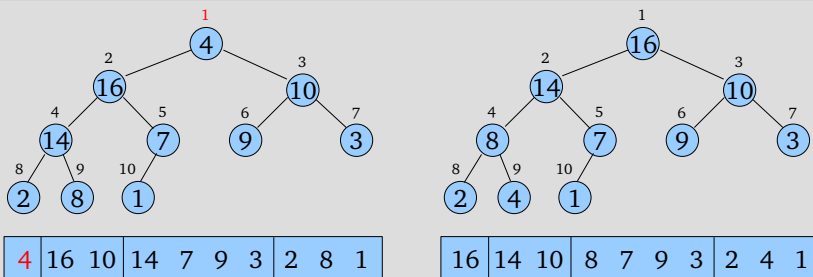
- Heapify(A, 3, 10)

Building a Heap/5



- Heapify(A, 2, 10)

Building a Heap/6



- Heapify(A, 1, 10)

Building a Heap: Analysis

- Correctness: induction on i , all trees rooted at $m > i$ are heaps.
- Running time: n calls to Heapify = $n O(\log n) = O(n \log n)$
- Non-tight bound but good enough for an overall $O(n \log n)$ bound for Heapsort.
- Intuition for a tight bound:
 - most of the time Heapify works on less than n element heaps

Quick Sort

- Characteristics
 - Like insertion sort, but unlike merge sort, sorts in-place, i.e., does not require an additional array.
 - Very practical, average sort performance $O(n \log n)$ (with small constant factors), but worst case $O(n^2)$.

Quick Sort – the Principle

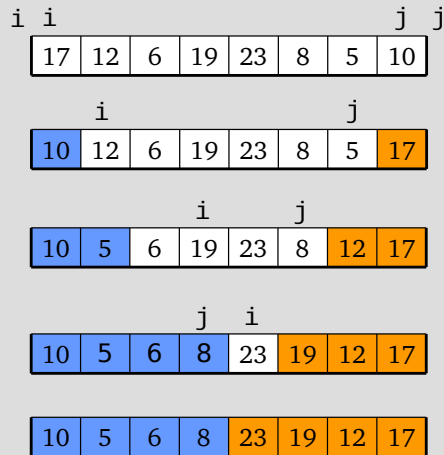
- To understand quick sort, let's look at a high-level description of the algorithm.
- A divide-and-conquer algorithm
 - **Divide:** partition array into 2 subarrays such that elements in the lower part \leq elements in the higher part.
 - **Conquer:** recursively sort the 2 subarrays
 - **Combine:** trivial since sorting is done in place

Partitioning

Partition(A, l, r)

```

01 x := A[r]
02 i := l-1
03 j := r+1
04 while TRUE
05   repeat j := j-1
06     until A[j] ≤ x
07   repeat i := i+1
08     until A[i] ≥ x
09   if i < j
10     then switch A[i] ↔ A[j]
11     else return i
    
```



Quick Sort Algorithm

- Initial call **Quicksort(A, 1, n)**

Quicksort(A, l, r)

```

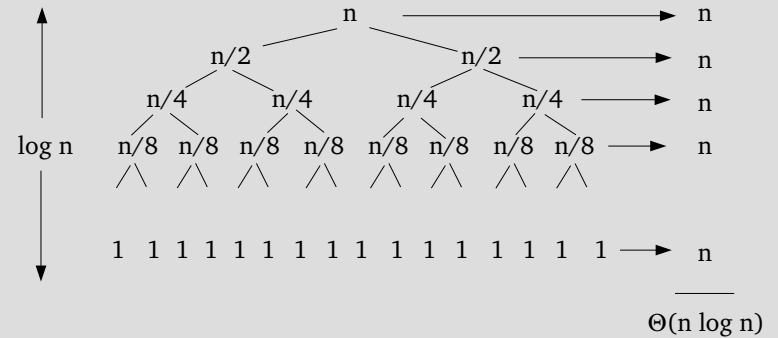
01 if l < r
02   m := Partition(A, l, r)
03   Quicksort(A, l, m-1)
04   Quicksort(A, m, r)
    
```

Analysis of Quicksort

- Assume that all input elements are distinct.
- The running time depends on the distribution of splits.

Best Case

- If we are lucky, Partition splits the array evenly $T(n) = 2T(n/2) + \Theta(n)$

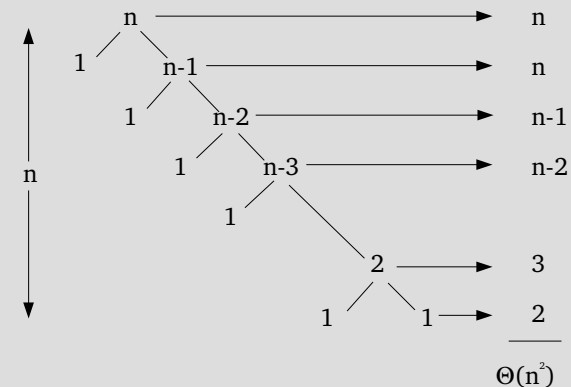


Worst Case

- What is the worst case?
- One side of the partition has one element.
- $T(n) = T(n-1) + T(1) + \Theta(n)$

$$\begin{aligned}
 &= T(n-1) + 0 + \Theta(n) \\
 &= \sum_{k=1}^n \Theta(k) \\
 &= \Theta\left(\sum_{k=1}^n k\right) \\
 &= \Theta(n^2)
 \end{aligned}$$

Worst Case/2



Randomized Quicksort

1 4 7 5

- Assume all elements are distinct.
- Partition around a random element.
- Consequently, all splits (1:n-1, 2:n-2, ..., n-1:1) are equally likely with probability 1/n.
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

Randomized Quicksort/2

RandomizedPartition(A, l, r)

```
01 i := Random(l, r)
02 exchange A[r] and A[i]
03 return Partition(A, l, r)
```

RandomizedQuicksort(A, l, r)

```
01 if l < r then
02   m := RandomizedPartition(A, l, r)
03   RandomizedQuicksort(A, l, m)
04   RandomizedQuicksort(A, m+1, r)
```

Summary

- Nearly complete binary trees
- Heap data structure
- Heapsort
 - based on heaps
 - worst case is $n \log n$
- Quicksort:
 - partition based sort algorithm
 - popular algorithm
 - very fast on average
 - worst case performance is quadratic

Summary/2

- Comparison of sorting methods.
- Absolute values are not important; relate values to each other.
- Relate values to the complexity ($n \log n$, n^2).
- Running time in seconds, $n=2048$.

	ordered	random	inverse
Insertion	0.22	50.74	103.8
Selection	58.18	58.34	73.46
Bubble	80.18	128.84	178.66
Heap	2.32	2.22	2.12
Quick	0.72	1.22	0.76

Next Week

- Dynamic data structures
 - Pointers
 - Lists, trees
- Abstract data types (ADTs)
 - Definition of ADTs
 - Common ADTs