SCC0602 - Algoritmos e Estruturas de Dados I

Advanced Sort

Professor: André C. P. L. F. de Carvalho, ICMC-USP
PAE: Rafael Martins D'Addio
Monitor: Joao Pedro Rodrigues Mattos

Today

- Sorting algorithms
  - Bubblesort
    - Simple and similar to Mergesort
  - Quicksort
    - Popular algorithm, very fast on average
  - Selection sort
    - Simple algorithm, inefficient for large structures
  - Heapsort
    - Heap data structure

Importance of Sorting

- One of the principles of algorithm design
  - "When in doubt, sort"
- Sorting is used as a subroutine in many algorithms:
  - Searching in databases, to allow binary search to be applied to sorted data
  - Element uniqueness, by duplicate elimination
  - Several computer graphics and computational geometry problems
    - Find the closest pair

Importance of Sorting

- A large number of sorting algorithms have been developed
  - Representing different algorithm design techniques
- Lower bound for sorting, $\Omega(n \log n)$, is often used to prove lower bounds of other problems

Definitions

- Input:
  - A sequence of n items $a_1, a_2, \ldots, a_n$
- Output:
  - A permutation (reordering) $a_1', a_2', \ldots, a_n'$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- The items to be sorted are usually part of a collection of data, named record
- Usually, a file store the records $R_1 \ldots R_n$

Definitions

- Each record R_i has:
  - A key K_i
  - Possibly other (satellite) data
- Input: n *records*, $R_1 \ldots R_n$, from a *file*
- Output: n *records*, $R_1' \ldots R_n'$, from a *file ordered by the value of k_i*

Key	Other data

Record

Definitions

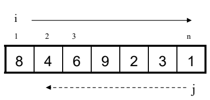
- *Sorting*: defines permutation $\Pi = (p_1, \dots, p_n)$ of n records with the keys in non-decreasing order
  - $Kp_1 \leq \dots \leq Kp_n$
- *Permutation*: a one-to-one function from $\{1, \dots, n\}$ onto itself
  - There are $n!$ distinct permutations of n items
- *Rank*: Given a collection of n keys, the *rank* of a key is the number of keys before it
  - $Rank(K_j) = |\{K_i \mid K_i < K_j\}|$
  - If the keys are distinct, the rank of a key gives its position in the sorted sequence

Definitions

- Internal Sort
  - Data to be sorted are all stored in the main memory
- External Sort
  - Some of the data to be sorted might be stored in an external, slower, device
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size

Bubblesort

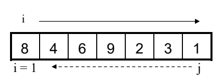
- Repeatedly pass through the array to be sorted
- Swap adjacent elements that are not in the correct order



- Easier to implement, but usually slower than insertion sort

Bubblesort

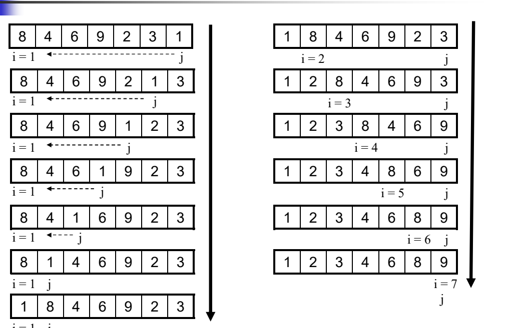
```
Bubblesort(A)
  for i ← 1 to length[A]
    do for j ← length[A] downto i + 1
         do if A[j] < A[j-1]
              then exchange A[j] ↔ A[j-1]
```



- What is the complexity of bubblesort?
  - Exercise

Bubblesort

```
Bubblesort(A)
  for i ← 1 to length[A]
    do for j ← length[A] downto i + 1
         do if A[j] < A[j-1]
              then exchange A[j] ↔ A[j-1]
```



Sorting Algorithms so far

- Insertion sort, selection sort, bubble sort
  - Worst-case running time $\Theta(n^2)$
  - Sort in place
    - Use a constant number of items outside the array
- Merge sort
  - Worst-case running time $\Theta(n \log n)$, but requires additional memory $\Theta(n)$
  - Does not sort in place

Quicksort

- Main characteristics
  - Like insertion sort, sorts in-place
    - Unlike merge sort
- Worst case $O(n^2)$
  - But, on average, its complexity is $O(n \log n)$
    - With small constant factors
- In practice, the best choice for sorting
- Works well in virtual memory environments

 13

Quicksort

- A divide-and-conquer algorithm
  - **Divide**: partition array into 2 subarrays with elements in the lower part <= elements in the higher part
    - For such, uses a pivot
  - **Conquer**: recursively sort the 2 subarrays
  - **Combine**: trivial since sorting occurs in place

 14

Quicksort Algorithm

```
Quicksort(A,p,r)
   if p < r
   then q ← Partition(A,p,r)
            Quicksort(A,p,q)
            Quicksort(A,q+1,r)
```

Initial call: *Quicksort(A, 1, length[A])*

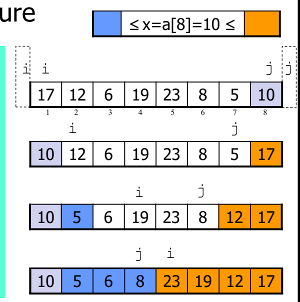
 15

Partitioning

- Linear time procedure

```
Suppose array A[p..r]

Partition(A,p,r)
   x ←A[r]  /* pivot */
   i ←p-1
   j ←r+1
   while TRUE
      repeat j ←j-1
         until A[j] ≤ x
      repeat i ←i+1
         until A[i] ≥ x
      if i<j
      then exchange A[i]↔A[j]
      else return j
```



 16

Analysis of Quicksort

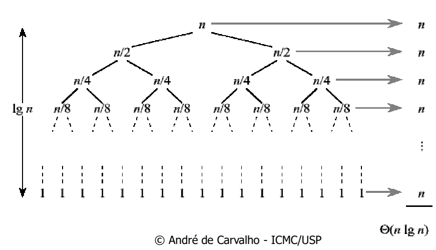
- Assume that all input items are distinct
  - Exchange items with the same value
- Running time depends on the distribution of array splits
  - Whether they are balanced
  - Which element (pivot) is used for partitioning

 17

Best Case

- Partition splits the array evenly

$$T(n) = 2T(n/2) + \Theta(n)$$



 18

Worst Case

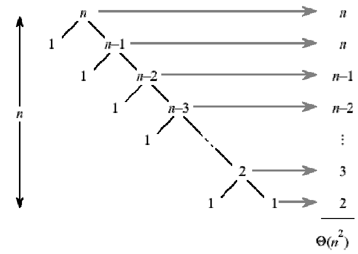
- One side of the partition has only one item

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{if } n > 1 \end{cases}$$

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

 19

Worst Case



 20

Worst Case

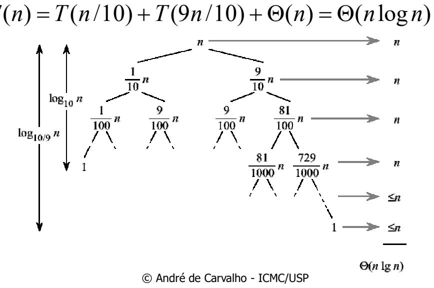
- Worst case appear when
  - The input is sorted (Ex.: 1, 2, 3)
  - The input is reverse sorted (Ex.: 3, 2, 1)
- Same recurrence for the worst case of insertion sort
- However, sorted input produces the best case for insertion sort: $\Theta(n)$

 21

Analysis of Quicksort

- Suppose the split is 1/10 : 9/10

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$



 22

Average Case Scenario

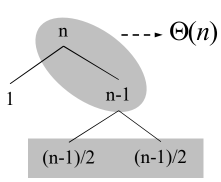
- Suppose, we alternate best and worst cases to get an average behavior

$L(n) = 2U(n/2) + \Theta(n)$ *Best*

$U(n) = 2L(n-1) + \Theta(n)$ *Worst*

Substituting:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(nlgn) \end{aligned}$$



 23

Average Case Scenario

- How to be sure that we are usually lucky?
  - Partition around the "middle" (n/2th) element?
    - No difference
  - Partition around a random element (works well in practice)
- Randomized algorithm
  - Running time is independent of the input ordering
  - No specific input triggers worst-case behaviour
  - The worst-case is only determined by the output of the random-number generator

 24

Randomized Quicksort

- Assume all elements are distinct
- Partition around a random element
  - All splits (1:n-1, 2:n-2, ..., n-1:1) become 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

```
Randomized-Partition(A,p,r)
  i ←Random(p,r)
  exchange A[r]↔A[i]
  return Partition(A,p,r)
```

```
Randomized-Quicksort(A,p,r)
  if p<r then
     q ←Randomized-Partition(A,p,r)
     Randomized-Quicksort(A,p,q)
     Randomized-Quicksort(A,q+1,r)
```

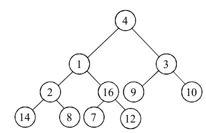

Selection Sort

```
Selection-Sort(A[1..n]):
   For i → n downto 2
A:    Find the largest element in A[1..i]
B:    Exchange it with A[i]
```

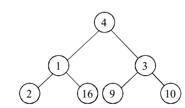
- A takes $\Theta(n)$ and B takes $\Theta(1)$: $\Theta(n^2)$ in total
- Possibility of improvement:
  - Use a smart *data structure* to do both A and B in $\Theta(1)$
  - Spend only $O(lg\ n)$ time in each iteration reorganizing the structure
  - Result: total running time of $O(n\ log\ n)$

Binary trees

- Binary tree: tree in which each node is either a leaf or has degree ≤ 2
  - Full binary: a binary tree in which each node is either a leaf or has degree exactly 2
  - Complete binary tree: a full binary tree in which all leaves are on the same level



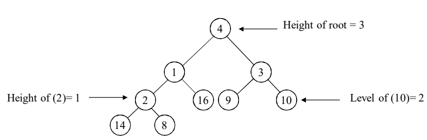
Full binary tree



Complete binary tree

Binary trees

- Height of a node: number of edges on the longest simple path from the node to a leaf
- Level of a node: length of a path from the root to the node
- Height of a tree: height of its root node

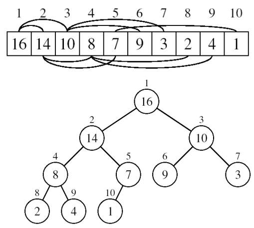


Heap Sort

- Uses an array *A* as a binary heap data structure
  - Array *A* can be seen as a nearly complete binary tree
    - Each node in the tree is an item in *A*
  - The value in the root is larger than or equal to all its children
    - The left and right subtrees are again binary heaps
- Does sort in place
  - Array *A* has two external attributes
    - length[*A*]: number of items in A
    - heap-size[*A*]: number of items in the heap stored in A
    - No item after A[heap-size[A]] is an item of the heap

Heap Sort

- In a heap stored as an array *A*
  - Root of tree is A[1]
  - Left child of A[i] = A[2i]
  - Right child of A[i] = A[2i + 1]
  - Parent of A[i] = A[⌊i/2⌋]
  - Heapsize[A] ≤ length[A]
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves



 31

Heap Sort

- Implicit tree links:
  - Children of node *i* are nodes 2*i* and 2*i*+1
  - Parent of node *i* is node ⌊*i/2*⌋
- Why is this useful?
  - In the binary representation
    - Multiplication (division) by two is left (right) shift
    - To add 1, just add to the lowest bit

 32

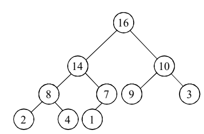
Heap types

- Max-heaps
  - Largest element at root, have the *max-heap property:*
    - for all nodes i, excluding the root: A[PARENT(i)] ≥ A[i]
- Min-heaps
  - Smallest element at root, have the *min-heap property:*
    - for all nodes i, excluding the root: A[PARENT(i)] ≤ A[i]

 33

Operations on heaps

- Adding nodes:
  - New nodes are always inserted at the bottom level (left to right)
- Deleting nodes:
  - Nodes are removed from the bottom level (right to left)



 34

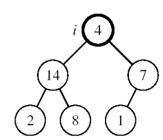
Operations on heaps

- Maintain/Restore the max-heap property
  - Max-Heapify
- Create a max-heap from an unordered array
  - Build-Max-Heap
- Sort an array in place
  - Heapsort

 35

Maintaining heap properties

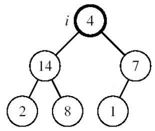
- Max-Heapify
- Binary trees rooted at Left(*i*) and Right(*i*) are heaps
  - However, *A*[*i*] may be smaller than its children, violating the max-heap property
- To eliminate the violation:
  - Exchange *A*[*i*] with larger child
  - Move down the tree until node is not smaller than its children



 36

Maintaining heap properties

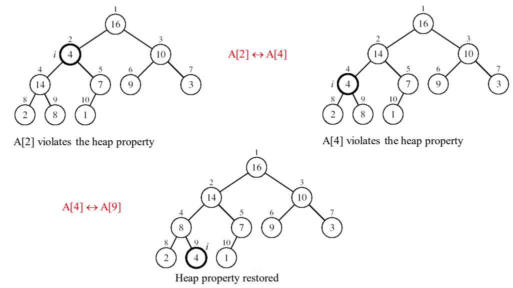
- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - *A[i]* may be smaller than its children



Max-Heapify(A, i)
n ← heap-size(A)
l ← Left (i)
r ← Right(i)
***if** l ≤ n and A[l] > A[i]*
***then** largest ←l*
***else** largest ←i*
***if** r ≤ n and A[r] > A[largest]*
***then** largest ←r*
***if** largest ≠ i*
***then** exchange A[i] ↔ A[largest]*
Max-Heapify (A, largest)

Example

Max-Heapify (A, 2)



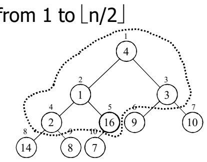
Max-Heapify running time

- Intuitively:
  - Max-Heapify runs a path from the root to a leaf
    - Longest path: *h*
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons: *2h*
    - Height of the heap (*h*) is ⌊*lgn*⌋
  - Running time: *O(h) = O (lgn)*
- Running time of Max-Heapify: *O(lgn)*

Building a Heap

- Convert an array A[1...n] into a heap
  - Consider *n* = length[A]
  - Elements in the subarray A[(⌊n/2⌋ + 1)...*n*], which are leaves, are already 1-element heaps
  - Apply *Max-Heapify* to elements from 1 to ⌊n/2⌋

Build-Max-Heap (A)
n ← length(A)
***for** i ← ⌊n/2⌋ **downto** 1*
***do** Max-Heapify (A,i)*



A:

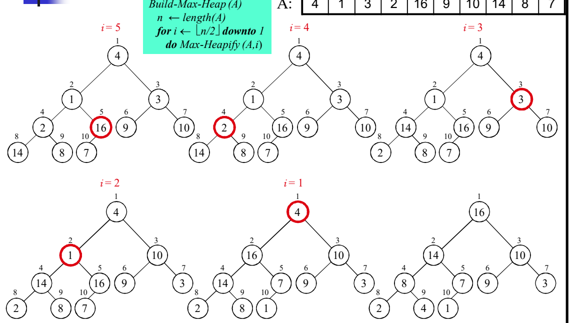
4	1	3	2	16	9	10	14	8	7

Building a Heap

Build-Max-Heap (A)
n ← length(A)
***for** i ← ⌊n/2⌋ **downto** 1*
***do** Max-Heapify (A,i)*

A:

4	1	3	2	16	9	10	14	8	7



Build-Max-Heap running time

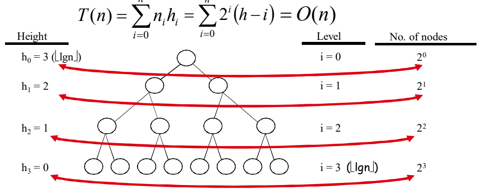
Build-Max-Heap (A)
n ← length(A)
*for i ← ⌊n/2⌋ **downto** 1*
do Max-Heapify (A,i) *O(lgn)* } *O(n)*

- Running time: *O(nlgn)*
  - As sometimes heaps are built for other reasons, it would be nice to have a tight bound
  - It is possible to derive a tighter bound
    - Time for Max-Heapify to run at a node varies with the height of the node
    - Heights of most nodes are small

Build-Max-Heap running time

- Max-Heapify takes *O(h)* ⇒ Cost of Max-Heapfy on a node *i* ~ the height of node *i* in the tree

$$T(n)=\sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$



$h_i = h - i$ (height of the heap rooted at level i)
$n_i = 2^i$ (number of nodes at level i)

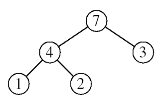
Build-Max-Heap running time

$T(n)=\sum_{i=0}^{h} n_i h_i$	Cost of Max-Heapfy at level i ∗ number of nodes at that level
$=\sum_{i=0}^{h} 2^i (h-i)$	Replace the values of n_i and h_i computed before
$=\sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$	Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$
$=2^h \sum_{k=0}^{h} \frac{k}{2^k}$	Change variables: k = h - i
$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$	The sum above is smaller than the sum of all elements to ∞ and h = lgn
$=O(n)$	The sum above is smaller than 2

Running time of Build-Max-Heap : *T(n) = O(n)*

Heapsort

- Goal: sort an array using heap representations
- Procedure:
  - Build a max-heap from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call Max-Heapfy on the new root
  - Repeat process until only one node remains

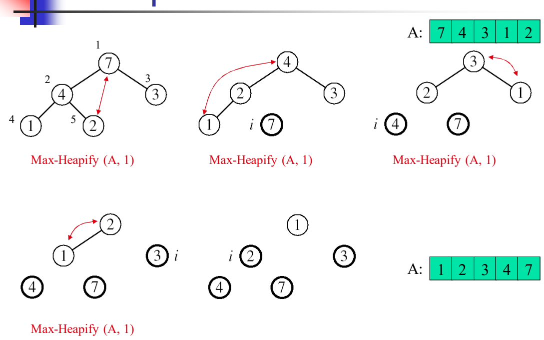


Heapsort running time

Heapsort (A)		
Build-Max-Heap (A)	O(n)	*O(n)*
for *i ← length[A]* ***downto*** *2*	n-1 times	n-1 times
do *exchange A[1] ↔ A[i]*	O(1)	
heap-size[A] ← heap-size[A]-1	O(1)	
Max-Heapfy (A,1)	O(lgn)	*O(lgn)*

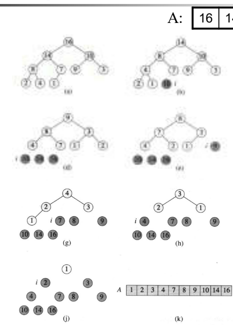
- We discard the previous root when applying Max-Heap (to the remaining heap)
- Running time is *O(n lg n)* + Build-Heap(*A*) time, which is *O(n)*

Example 1



Example 2

A: 16 14 10 9 8 7 4 3 2 1

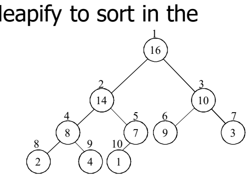


Summary

- Heapsort uses a heap data structure to improve selection sort and make the running time asymptotically optimal
- Running time is *O(n log n)*
  - Like merge sort, but unlike selection, insertion, or bubble sorts
- Sorts in place
  - Like insertion, selection or bubble sorts, but unlike merge sort

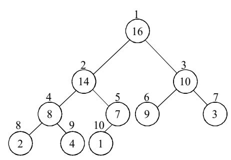
Summary

- Why Max-Heapify instead of Min-Heapify
  - It is not easy to recover the elements in increasing order if we use Min-Heapify
    - See heap below
  - We could use Min-Heapify to sort in the decreasing order



Exercise

- Assuming the data in a max-heap are distinct, what are the possible locations of the second-largest element?



Exercise

1. Given a max heap B of height h
   a) What is the maximum number of nodes in B?
   b) What is the maximum number of leaves?
   c) What is the maximum number of internal nodes?

Exercise

- Demonstrate, step by step, the operation of Build-Heap on the array

  A=[5, 3, 17, 10, 84, 19, 6, 22, 9]

Exercise

- Let A be a heap of size n. Give the most efficient algorithm for the following tasks:
  (a) Find the sum of all elements
  (b) Find the sum of the largest lgn elements

Next Week

- Hashing

 55

Acknowledgement

- A large part of this material were adapted from
  - Simonas Šaltenis, Algorithms and Data Structures, Aalborg University, Denmark
  - Mary Wootters, Design and Analysis of Algorithms, Stanford University, USA
  - George Bebis, Analysis of Algorithms CS 477/677, University of Nevada, Reno
  - David A. Plaisted, Information Comp 550-001, University of North Carolina at Chapel Hill

 56

Questions

 57