

**CSCE 310J: Data Structures & Algorithms**

**Transform & Conquer!**

Dr. Steve Goddard  
goddard@cse.unl.edu

<http://www.cse.unl.edu/~goddard/Courses/CSCE310J>

Design and Analysis of Algorithms - Chapter 6 1

**CSCE 310J: Data Structures & Algorithms**

Q Giving credit where credit is due:

- Most of the lecture notes are based on the slides from the Textbook's companion website
  - <http://www.aw.com/cssupport/>
- Some examples and slides are based on lecture notes created by Dr. Ben Choi, Louisiana Technical University and Dr. Chuck Cusack, UNL
- I have modified many of their slides and added new slides.

Design and Analysis of Algorithms - Chapter 6 2

**Transform and Conquer**

Solve problem by transforming into:

- Q a more convenient instance of the same problem (*instance simplification*)
  - presorting
  - Gaussian elimination
- Q a different representation of the same instance (*representation change*)
  - balanced search trees
  - heaps and heapsort
  - polynomial evaluation by Horner's rule
  - Fast Fourier Transform
- Q a different problem altogether (*problem reduction*)
  - reductions to graph problems
  - linear programming

Design and Analysis of Algorithms - Chapter 6 3

**Instance simplification - Presorting**

Solve instance of problem by transforming into another simpler/easier instance of the same problem

**Presorting:**  
Many problems involving lists are easier when list is sorted.

- Q searching
- Q computing the median (selection problem)
- Q computing the mode
- Q finding repeated elements

Design and Analysis of Algorithms - Chapter 6 4

**Selection Problem**

Find the  $k^{\text{th}}$  smallest element in  $A[1], \dots, A[n]$ . Special cases:

- *minimum*:  $k = 1$
- *maximum*:  $k = n$
- *median*:  $k = \lceil n/2 \rceil$

- Q **Presorting-based algorithm**
  - sort list
  - return  $A[k]$
- Q **Partition-based algorithm (Variable decrease & conquer):**
  - pivot/split at  $A[s]$  using partitioning algorithm from quicksort
  - if  $s=k$  return  $A[s]$
  - else if  $s < k$  repeat with sublist  $A[s+1], \dots, A[n]$ .
  - else if  $s > k$  repeat with sublist  $A[1], \dots, A[s-1]$ .

Design and Analysis of Algorithms - Chapter 6 5

**Notes on Selection Problem**

- Q **Presorting-based algorithm:**  $\Omega(n \lg n) + \Theta(1) = \Omega(n \lg n)$
- Q **Partition-based algorithm (Variable decrease & conquer):**
  - worst case:  $T(n) = T(n-1) + (n+1) \rightarrow \Theta(n^2)$
  - best case:  $\Theta(n)$
  - average case:  $T(n) = T(n/2) + (n+1) \rightarrow \Theta(n)$
  - **Bonus:** also identifies the  $k$  smallest elements (not just the  $k^{\text{th}}$ )
- Q **Special cases max, min:** better, simpler linear algorithm (brute force)
- Q **Conclusion:** Presorting does *not* help in this case.

Design and Analysis of Algorithms - Chapter 6 6

### Finding repeated elements

- Presorting-based algorithm:
  - use mergesort (optimal):  $\Theta(n \lg n)$
  - scan array to find repeated adjacent elements:  $\Theta(n)$ $\} \Theta(n \lg n)$
- Brute force algorithm:  $\Theta(n^2)$
- Conclusion: Presorting yields significant improvement
- Similar improvement for mode
- What about searching?

Design and Analysis of Algorithms - Chapter 6 7

### Taxonomy of Searching Algorithms

- Elementary searching algorithms
  - sequential search
  - binary search
  - binary tree search
- Balanced tree searching
  - AVL trees
  - red-black trees
  - multi-way balanced trees (2-3 trees, 2-3-4 trees, B trees)
- Hashing
  - separate chaining
  - open addressing

Design and Analysis of Algorithms - Chapter 6 8

### Left- and Right-Rotations

The BST property still holds after a rotation.

Right-Rotate(T,y) and Left-Rotate(T,x) are indicated by arrows.

Design and Analysis of Algorithms - Chapter 6 9

### Binary Tree Rotations

Left Rotation on (15, 25)

Design and Analysis of Algorithms - Chapter 6 10

### Balanced trees: AVL trees

- For every node, difference in height between left and right subtree is at most 1
- AVL property is maintained through rotations, each time the tree becomes unbalanced
- $\lg n \leq h \leq 1.4404 \lg(n+2) - 1.3277$   
 average:  $1.01 \lg n + 0.1$  for large  $n$
- Disadvantage: needs extra storage for maintaining node balance
- A similar idea: red-black trees (height of subtrees is allowed to differ by up to a factor of 2)

Design and Analysis of Algorithms - Chapter 6 11

### Balance factor

Algorithm maintains balance factor for each node. For example:

Design and Analysis of Algorithms - Chapter 6 12

### General case: single R-rotation

Design and Analysis of Algorithms - Chapter 6

13

### Double LR-rotation

Design and Analysis of Algorithms - Chapter 6

14

### AVL tree rotations

Q Small examples:

- 1, 2, 3
- 3, 2, 1
- 1, 3, 2
- 3, 1, 2

Q Larger example: 4, 5, 7, 2, 1, 3, 6

Q See figures 6.4, 6.5 for general cases of rotations;

Design and Analysis of Algorithms - Chapter 6

15

### Heapsort

**Definition:**  
A *heap* is a binary tree with the following conditions:

Q it is essentially complete:

Q The key at each node is  $\geq$  keys at its children

Design and Analysis of Algorithms - Chapter 6

16

### Heaps (or not)?

(a) A 2-tree

(b) A complete binary tree

(c) Heap 1

(d) Heap 2

Design and Analysis of Algorithms - Chapter 6

17

### Definition implies:

Q Given  $n$ , there exists a unique binary tree with  $n$  nodes that is essentially complete, with  $h = \lfloor \lg n \rfloor$

Q The root has the largest key

Q The subtree rooted at any node of a heap is also a heap

Design and Analysis of Algorithms - Chapter 6

18

### Heapsort Strategy

Q If the elements to be sorted are arranged in a heap, we can build a sorted sequence in reverse order by

- repeatedly removing the element from the root,
- rearranging the remaining elements to reestablish the partial order tree property,
- and so on.

Q How does it work?

Design and Analysis of Algorithms - Chapter 6 19

### Heapsort Algorithm:

- Build heap
- Remove root –exchange with last (rightmost) leaf
- Fix up heap (excluding last leaf)

Repeat 2, 3 until heap contains just one node.

Design and Analysis of Algorithms - Chapter 6 20

### Heap construction

Q Insert elements in the order given breadth-first in a binary tree

Q Starting with the last (rightmost) parental node, fix the heap rooted at it, if it does not satisfy the heap condition:

- exchange it with its largest child
- fix the subtree rooted at it (now in the child's position)

Example: 2 3 6 7 5 9

Design and Analysis of Algorithms - Chapter 6 21

### Heap construction Strategy (divide and conquer)

Q base case is a tree consisting of one node

Design and Analysis of Algorithms - Chapter 6 22

### Construct Heap Outline

Q Input: A heap structure H that does not necessarily have the partial order tree property

Q Output: H with the same nodes rearranged to satisfy the partial order tree property

```
void constructHeap(H) // Outline
if (H is not a leaf)
    constructHeap (left subtree of H);
    constructHeap (right subtree of H);
    Element K = root(H);
    fixHeap(H, K);
return;
```

Q  $T(n) = T(n-r-1) + T(r) + 2 \lg(n)$  for  $n > 1$  where  $r$  is the number of nodes in the right subheap

Q  $T(n) \in \Theta(n)$ ; heap is constructed in linear time.

Design and Analysis of Algorithms - Chapter 6 23

### Root deletion

The root of a heap can be deleted and the heap fixed up as follows:

- exchange the root with the last leaf
- compare the new root (formerly the leaf) with each of its children and, if one of them is larger than the root, exchange it with the larger of the two.
- continue the comparison/exchange with the children of the new root until it reaches a level of the tree where it is larger than both its children

Design and Analysis of Algorithms - Chapter 6 24

### Heapsort Outlines

```

    heapSort(E, n) // Outline
    construct H from E, the set of n elements to be sorted
    for (i = n; i >= 1; i--)
        curMax = getMax(H)
        deleteMax(H);
        E[i] = curMax;
    deleteMax(H) // Outline
    • copy the rightmost element of the lowest level of H into K
    • delete the rightmost element on the lowest level of H
    • fixHeap(H, K); // reinsert K into a heap H with a vacant root
      assumed
  
```

Design and Analysis of Algorithms - Chapter 6 25

### Heapsort in action

Actually, figures b-e show deleteMax() in action

Design and Analysis of Algorithms - Chapter 6 25

### Fixheap Outline

```

    fixHeap(H, K) // Outline
    if (H is a leaf)
        insert K in root(H);
    else
        set largerSubHeap to leftSubtree(H) or rightSubtree(H),
        whichever has larger key at its root. This involves one key
        comparison.
        if (K.key >= root(largerSubHeap).key)
            insert K in root(H);
        else
            insert root(largerSubHeap) in root(H);
            fixHeap(largerSubHeap, K);
    return;
  
```

FixHeap requires  $2h$  comparisons of keys in the worst case on a heap with height  $h$ .  $T(n) \approx 2 \lg(n)$

Design and Analysis of Algorithms - Chapter 6 27

### Representation

Use an array to store breadth-first traversal of heap tree:

Example:

- Left child of node  $j$  is at  $2j$
- Right child of node  $j$  is at  $2j+1$
- Parent of node  $j$  is at  $j/2$
- Parental nodes are represented in the first  $n/2$  locations

Design and Analysis of Algorithms - Chapter 6 28

### Bottom-up heap construction algorithm

```

    Algorithm HeapBottomUp(H[1..n])
    //Constructs a heap from the elements of a given array
    // by the bottom-up algorithm
    //Input: An array H[1..n] of orderable items
    //Output: A heap H[1..n]
    for i ← [n/2] downto 1 do
        k ← i; v ← H[k]
        heap ← false
        while not heap and 2 * k ≤ n do
            j ← 2 * k
            if j < n //there are two children
                if H[j] < H[j+1] j ← j+1
            if v ≥ H[j]
                heap ← true
            else H[k] ← H[j]; k ← j
        H[k] ← v
  
```

Design and Analysis of Algorithms - Chapter 6 29

### Analysis of Heapsort

See algorithm HeapBottomUp in section 6.4

- Fix heap with “problem” at height  $j$ :  $2j$  comparisons
- For subtree rooted at level  $i$  it does  $2(h-i)$  comparisons
- Total for heap construction phase:
 
$$\sum_{i=0}^{h-1} 2^{(h-i)} 2^i = 2(n - \lg(n+1)) = \Theta(n)$$
 # nodes at level  $i$

Design and Analysis of Algorithms - Chapter 6 30

### Analysis of Heapsort (continued)

**Recall algorithm:**

$\Theta(n)$  1. Build heap

2. Remove root –exchange with last (rightmost) leaf

$\Theta(\log n)$  3. Fix up heap (excluding last leaf)

Repeat 2, 3 until heap contains just one node.

$n - 1$  times

**Total:**  $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$

• **Note:** this is the worst case. Average case also  $\Theta(n \log n)$ .

Design and Analysis of Algorithms - Chapter 6 31

### Priority queues

Q A *priority queue* is the ADT of an ordered set with the operations:

- find element with highest priority
- delete element with highest priority
- insert element with assigned priority

Q Heaps are very good for implementing priority queues

Design and Analysis of Algorithms - Chapter 6 32

### Insertion of a new element

Q Insert element at last position in heap.

Q Compare with its parent and if it violates heap condition exchange them

Q Continue comparing the new element with nodes up the tree until the heap condition is satisfied

**Example:**

**Efficiency:**

Design and Analysis of Algorithms - Chapter 6 33

### Bottom-up vs. Top-down heap construction

Q **Top down:** Heaps can be constructed by successively inserting elements into an (initially) empty heap

Q **Bottom-up:** Put everything in and then fix it

Q Which one is better?

Design and Analysis of Algorithms - Chapter 6 34