Transform and Conquer

Solve problem by transforming into:

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

Selection Problem

Find the k\textsuperscript{th} smallest element in A[1]...A[n]. Special cases:
- minimum: k = 1
- maximum: k = n
- median: k = \lfloor n/2 \rfloor

Presorting-based algorithm
- sort list
- return A[k]

Partition-based algorithm (Variable decrease & conquer):
- pivot/split at A[i] using partitioning algorithm from quicksort
- if i=k return A[i]
- else if i<k repeat with sublist A[i+1]...A[n]
- else if i>k repeat with sublist A[1]...A[i-1]

Notes on Selection Problem

- Presorting-based algorithm: \Theta(n \log n) + \Theta(1) = \Theta(n \log n)
- 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\textsuperscript{th})
- Special cases max, min: better, simpler linear algorithm (brute force)
- Conclusion: Presorting does not help in this case.
Design and Analysis of Algorithms

Chapter 6

Finding repeated elements

- Presorting-based algorithm:
  - use mergesort (optimal): \( \Theta(n \log n) \)
  - scan array to find repeated adjacent elements: \( \Theta(n) \)
- Brute force algorithm: \( \Theta(n^2) \)

Conclusion: Presorting yields significant improvement

Similar improvement for mode

What about searching?

Left- and Right-Rotations

- The BST property still holds after a rotation.

Binary Tree Rotations

- Left Rotation on (15, 25)

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
- \( \lg n \leq h \leq 1.44404 \lg (n + 2) - 1.3277 \) 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)

Balance factor

- Algorithm maintains balance factor for each node. For example:
**General case: single R-rotation**

- Small examples:
  - 1, 2, 3
  - 3, 2, 1
  - 1, 3, 2
  - 3, 1, 2

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

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

**Double LR-rotation**

**AVL tree rotations**

- Small examples:
  - 1, 2, 3
  - 3, 2, 1
  - 1, 3, 2
  - 3, 1, 2

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

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

**Heapsort**

**Definition:**

A heap is a binary tree with the following conditions:

- It is essentially complete:

- The key at each node is ≥ keys at its children

**Heaps (or not)?**

**Definition implies:**

- Given n, there exists a unique binary tree with n nodes that is essentially complete, with h = ⌈lg n⌉

- The root has the largest key

- The subtree rooted at any node of a heap is also a heap
Heapsort Strategy

- 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.

- How does it work?

Heapsort Algorithm:

1. Build heap
2. Remove root (exchange with last (rightmost) leaf)
3. Fix up heap (excluding last leaf)

Repeat 2, 3 until heap contains just one node.

Heap construction

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

- Starting with the last (rightmost) parental node, fix the heap rooted at it, if it does not satisfy the heap condition:
  1. exchange it with its largest child
  2. fix the subtree rooted at it (now in the child’s position)

Example: 2 3 6 7 5 9

Heap construction strategy (divide-and-conquer)

- Base case is a tree consisting of one node

Construct Heap Outline

- Input: A heap structure H that does not necessarily have the partial order tree property
- 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;
- 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
- T(n) = O(n); heap is constructed in linear time.

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
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

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 is root. This involves one key comparison.
    if (key [root(largerSubHeap).key])
      insert K in root(H);
    - else
      insert root(largerSubHeap) in root(H);
      fixHeap(largerSubHeap, K);
      return;
- fixHeap requires 2h comparisons in the worst case on a heap with height k. T(n) = 2 lg n

Bottom-up heap construction algorithm

Algorithm HeapBottomUp(H[1..n])
//Constructs a heap from the elements of a given array
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
For i = ⌊n/2⌋ downto 1 do
  heap = false
  while not heap and 2 * k ≤ n do
    if j < n //there are two children
      if H[j] < H[i + k]
        j ← j + 1
      else
        heap ← true
        H[i] ← H[2j]; k ← j
        H[2j] ← 0

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:
  \[
  \sum_{j=1}^{\lfloor n/2 \rfloor} 2^{(h-i)} \leq 2 (n - \log (n + 1)) = \Theta(n)
  \]
  # nodes at level i
Recall algorithm:
1. Build heap
2. Remove root - exchange with last (rightmost) leaf
3. Fix up heap (excluding last leaf)

Repeat 2, 3 until heap contains just one node.

\[
T = n - 1 \text{ times}
\]

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

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

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

Heaps are very good for implementing priority queues

Insertion of a new element
- Insert element at last position in heap.
- Compare with its parent and if it violates heap condition exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example:

Efficiency:

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

Bottom-up: Put everything in and then fix it

Which one is better?