CSCE 310J: Data Structures & Algorithms

Transform & Conquer!

Dr. Steve Goddard
goddard@cse.unl.edu

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

CSCE 310J: Data Structures & Algorithms

- **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/cssuport/**
  - **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.**

Transform and Conquer

Solve problem by transforming into:

- **a more convenient instance of the same problem (*instance implification*)**
  - **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**

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.

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

Selection Problem

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

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

- **Presorting-based algorithm**
  - **sort list**
  - **return A[k]**
- **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],…A[n].**
  - ***else if*** **$s>k$ repeat with sublist A[1],…A[s-1].**

Notes on Selection Problem

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

Finding repeated elements

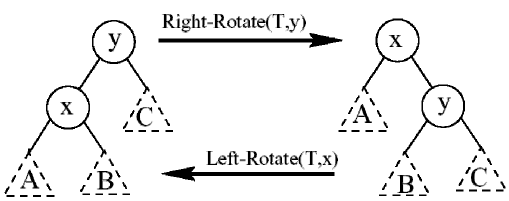
- **Presorting-based algorithm:**
  - use mergesort (optimal): Θ(*n*lg*n*)
  - scan array to find repeated <u>adjacent</u> elements: Θ(*n*)
  - } Θ(*n*lg*n*)
- **Brute force algorithm: Θ(n^2)**
- **<u>Conclusion:</u> Presorting yields <u>significant</u> improvement**
- **Similar improvement for mode**
- **What about searching?**

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

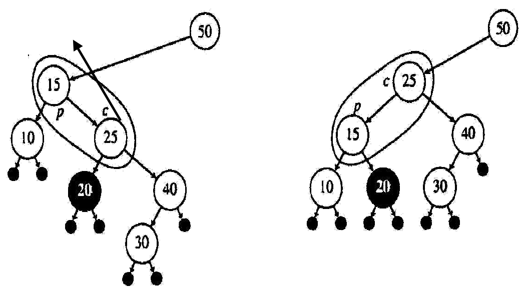
Left- and Right-Rotations

- **The BST property still holds after a rotation.**



Binary Tree Rotations

- **Left Rotation on (15, 25)**

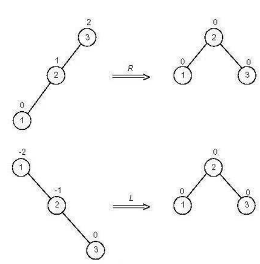


Balanced trees: AVL trees

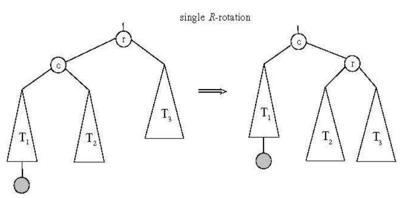
- **For every node, difference in height between left and right subtree is at most 1**
- **AVL property is maintained through *rotations*, each time the tree becomes unbalanced**
- **lg *n* ≤ *h* ≤ 1.4404 lg (*n* + 2) - 1.3277**
  <u>average:</u> 1.01 lg *n* + 0.1 for large *n*
- **Disadvantage: needs 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

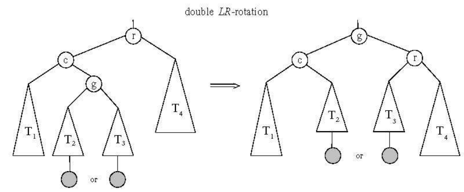
- **Algorithm maintains *<u>balance factor</u>* for each node. For example:**



General case: single R-rotation



Double LR-rotation



AVL tree rotations

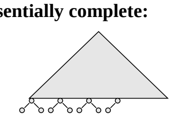
- **Small examples:**
  - **1, 2, 3**
  - **3, 2, 1**
  - **1, 3, 2**
  - **3, 1, 2**
- **Larger example: 4, 5, 7, 2, 1, 3, 6**
- **See figures 6.4, 6.5 for general cases of rotations;**

Heapsort

<u>Definition:</u>

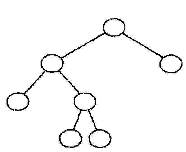
A *heap* is a binary tree with the following conditions:

- **it is essentially complete:**

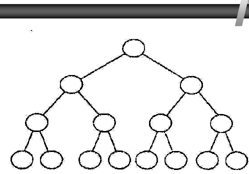


- **The key at each node is ≥ keys at its children**

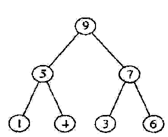
Heaps (or not)?



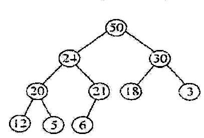
(a) A 2-tree



(b) A complete binary tree



(c) Heap 1



(d) Heap 2

Definition implies:

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

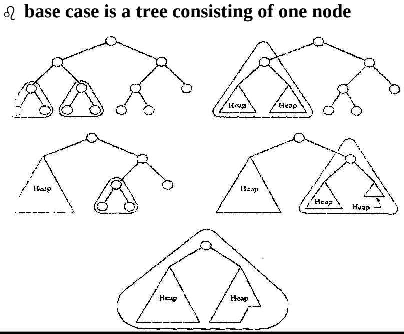
Heap construction

- 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
- $T(n) \in \theta(n)$; 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

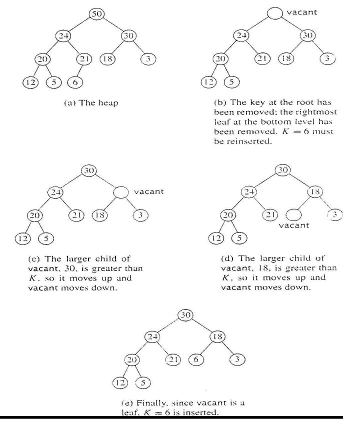
- **heapSort(E, n) // Outline**

```
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;
```

- **deteleMax(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

Heapsort in action

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



Fixheap Outline

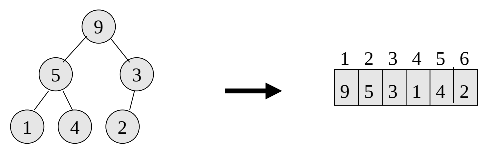
- **fixHeap(H, K) // Outline**

```
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 (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) ≈ 2 lg(n)**

Representation

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



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

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
```


Analysis of Heapsort

See algorithm HeapBottomUp in section 6.4

- **Fix heap with "problem" at height *j*: 2*j* 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*

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 $\Theta(n \log n)$.

Priority queues

- **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:

Bottom-up vs. Top-down heap construction

- **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?**