CSC 8301- Design and Analysis of Algorithms

Lecture 7

Transform and Conquer I
Algorithm Design Technique

Transform and Conquer

This group of techniques solves a problem by a *transformation* to

- a simpler/more convenient instance of the *same* problem (*instance simplification*)
- a different representation of the *same* instance (*representation change*)
- a *different* problem for which an algorithm is already available (*problem reduction*)

Transform and Conquer

- *Instance simplification*
  - Presorting
  - Gaussian Elimination
- *Representation change*
  - Binary Search Trees
  - Heaps
  - Horner's rule for polynomial evaluation
- *Problem reduction*
  - Example: compute lcm(a,b) by computing gcd(a,b)

Instance simplification - Presorting

Presorting

- sorting ahead of time, to make repetitive solutions faster

Many problems involving lists are easier when list is sorted, e.g.,

- searching
- computing the median (selection problem)
- checking if all elements are distinct (element uniqueness)

Also:

- Topological sorting helps solving some problems for dags
- Presorting is used in many geometric algorithms

How fast can we sort ?

Efficiency of algorithms involving presorting depends on efficiency of the sorting algorithm used

Theorem (see Sec. 11.2): $\lceil \log_2 n! \rceil \approx n \log_2 n$ comparisons are necessary in the worst case to sort a list of size n by any comparison-based algorithm

Note: About $n\log_2 n$ comparisons are also sufficient to sort array of size n (by mergesort)

Searching with presorting

Problem: Search for a given K in A[0..n-1]

Presorting-based algorithm:

Stage 1 Sort the array by, say, mergesort

Stage 2 Apply binary search

Efficiency: $\Theta(n\log n) + O(\log n) = \Theta(n\log n)$

Good or bad?

Why do we have our dictionaries, telephone directories, etc. sorted?

Instance Simplification – Element Uniqueness

- Presorting-based algorithm
  Stage 1: sort by efficient sorting algorithm (e.g. mergesort)
  Stage 2: scan array to check pairs of adjacent elements

  Efficiency: $\Theta(n\log n) + O(n) = \Theta(n\log n)$

- Brute force algorithm
  – Compare all pairs of elements
  – Efficiency: $O(n^2)$

Instance simplification – Gaussian Elimination

You are familiar with systems of two linear equations:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

Unless $a_{11}/a_{21} = a_{12}/a_{22}$, the system has a unique solution

Instance simplification – Gaussian Elimination

You are familiar with systems of two linear equations:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

Unless $a_{11}/a_{21} = a_{12}/a_{22}$, the system has a unique solution:

- Multiply the first equation by $-a_{21}/a_{11}$
  $$-a_{21}x_1 - (a_{21}a_{12}/a_{11})\,x_2 = -a_{21}b_1/a_{11}$$
- Add the above equation to the 2nd one in the system
  $$(a_{22} - a_{21}a_{12}/a_{11})x_2 = b_2 - a_{21}b_1/a_{11}$$
- Extract x_2 from this equation, substitute in the 1st

Instance simplification – Gaussian Elimination

Given: A system of n linear equations in n unknowns with an arbitrary coefficient matrix.

Transform to: An equivalent system of n linear equations in n unknowns with an upper triangular coefficient matrix.

Solve the latter by substitutions starting with the last equation and moving up to the first one.

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{array} \longrightarrow \begin{array}{rl} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n & = b_1 \\ a_{22}x_2 + \dots + a_{2n}x_n & = b_2 \\ \\ a_{nn}x_n & = b_n \end{array}$$

Gaussian Elimination (cont.)

The transformation is accomplished by a sequence of elementary operations on the system's coefficient matrix (which don't change the system's solution):

for $i \leftarrow 1$ to n-1 do
replace each of the subsequent rows (i.e., rows i+1, …, n) by a difference between that row and an appropriate multiple of the i-th row to make the new coefficient in the i-th column of that row 0

Example of Gaussian Elimination

Solve

$$\begin{aligned} 2x_1 - 4x_2 + x_3 &= 6 \\ 3x_1 - x_2 + x_3 &= 11 \\ x_1 + x_2 - x_3 &= -3 \end{aligned}$$

Example of Gaussian Elimination

Solve
$$2x_1 - 4x_2 + x_3 = 6$$
$$3x_1 - x_2 + x_3 = 11$$
$$x_1 + x_2 - x_3 = -3$$

Gaussian elimination

```
2  -4   1   6
3  -1   1  11   row2 – (3/2)*row1
1   1  -1  -3   row3 – (1/2)*row1
```

```
2  -4     1    6
0   5  -1/2    2      Repeat
0   3  -3/2   -6   row3–(3/5)*row2
```

```
2  -4     1     6
0   5  -1/2     2
0   0  -6/5  -36/5
```

Backward substitution

$x_3 = (-36/5) / (-6/5) = 6$
$x_2 = (2+(1/2)*6) / 5 = 1$
$x_1 = (6 – 6 + 4*1)/2 = 2$

Pseudocode & Efficiency of Gaussian Elimination

Stage 1: Reduction to the upper-triangular matrix

for $i \leftarrow 1$ to n-1 do
    for $j \leftarrow i$+1 to n do
        $temp \leftarrow A[j, i] / A[i, i]$ ($A[i,i]$ must be non-zero!)
        for $k \leftarrow i$ to n+1 do
        $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$

Pseudocode & Efficiency of Gaussian Elimination

Stage 2: Backward substitution

for $j \leftarrow n$ downto 1 do
    $t \leftarrow 0$
    for $k \leftarrow j+1$ to n do
        $t \leftarrow t + A[j, k] * x[k]$
    $x[j] \leftarrow (A[j, n+1] - t) / A[j, j]$

Efficiency: $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

Transform and Conquer

Representation Change

Searching Problem

Problem: Given a (multi)set *S* of keys and a search key *K*,
find an occurrence of *K* in *S*, if any

- There is no single algorithm that fits all situations best
- Searching must be considered in the context of:
  - file size (internal or external)
  - dynamics of data (static vs. dynamic)

- Dictionary operations (dynamic data):
  - find (search)
  - insert
  - delete

Taxonomy of Searching Algorithms

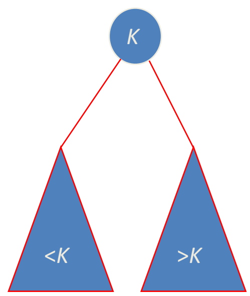
- List searching
  - sequential search
  - binary search

- Tree searching
  - binary search tree
  - binary balanced trees: AVL trees, red-black trees
  - multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees

- Hashing
  - open hashing (separate chaining)
  - closed hashing (open addressing)

Binary Search Tree

Arrange keys in a binary tree with the *binary search tree property*:



Examples: 5, 3, 1, 10, 12, 7, 9   1, 2, 3, 4, 5, 6, 7

Bonus: inorder traversal produces sorted list

Dictionary Operations on Binary Search Trees

- Searching – straightforward
- Insertion – search for key, insert at leaf where search terminated
- Deletion – 3 cases:
  - deleting key at a leaf
  - deleting key at node with single child
  - deleting key at node with two children

- Efficiency depends of the tree's height: $\lfloor \log_2 n \rfloor \leq h \leq n$-1, with height average (random files) be about $3\log_2 n$

- Thus all three operations have
  - worst case efficiency: $\Theta(n)$
  - average case efficiency: $\Theta(\log n)$

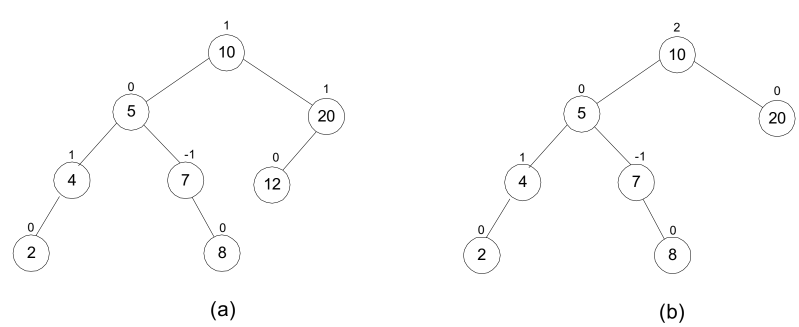
Balanced Search Trees

Attractiveness of *binary search tree* is marred by the bad (linear) worst-case efficiency. Two ideas to overcome it are:

- To rebalance binary search tree when a new insertion makes the tree "too unbalanced"
  - *AVL trees*
  - *red-black trees*
- To allow more than one key per node of a search tree
  - *2-3 trees*
  - *2-3-4 trees*
  - *B-trees*

Balanced trees: AVL trees

Definition An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)



Which of these is an AVL tree?

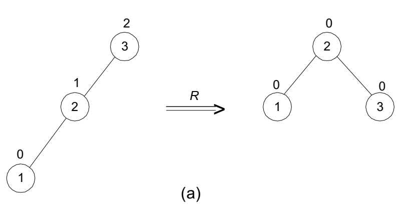
AVL Trees – Insert Operation

- Example 1: insert the keys 3, 2 and 1 in an AVL tree in this order

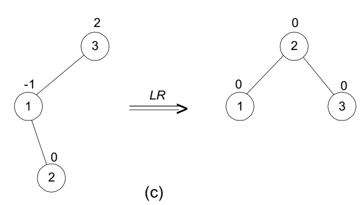
- Example 2: insert the keys 3, 1 and 2 in an AVL tree in this order

Rotations

- If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of 4 *rotations*. *The rotation is always performed for a subtree rooted at an "unbalanced" node closest to the new leaf*.

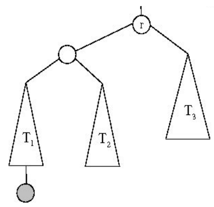


Single *R*-rotation

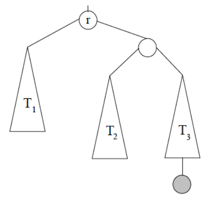


Double *LR*-rotation

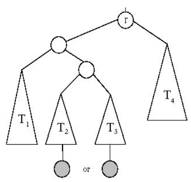
Unbalanced Cases (after insertion)



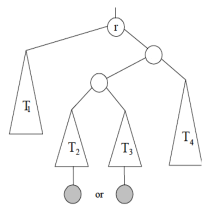
Left-Left: Left subtree of Left child
R-rotation(*r*)



Right-Right: Right subtree of Right child
L-rotation(*r*)

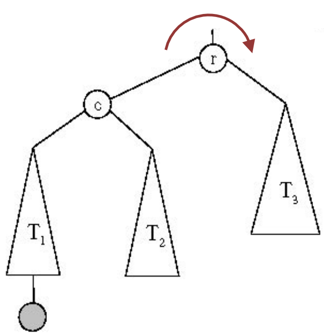


Right-Left: Right subtree of Left child
LR-rotation(*r*)

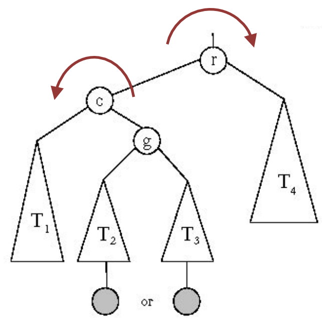


Left-Right: Left subtree of Right child
RL-rotation(*r*)

General case: Single R-rotation



General case: Double LR-rotation

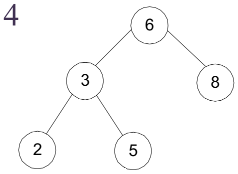


AVL tree construction - an example

Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7

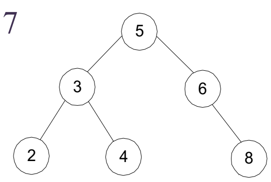
AVL tree construction - an example (cont.)

Insert 4



AVL tree construction - an example (cont.)

Insert 7



Analysis of AVL trees

- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
  average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)

- Search and insertion are O(log n)

- Deletion is more complicated but is also O(log n)

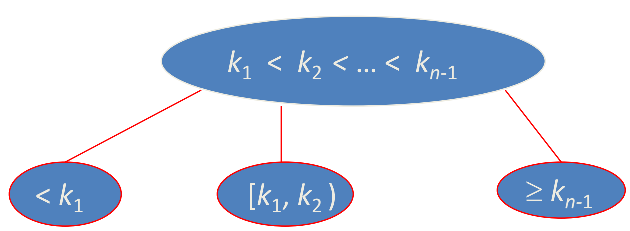
- Disadvantages:
  - frequent rotations
  - complexity

- A similar idea: *red-black trees* (height of subtrees is allowed to differ by up to a factor of 2)

Multiway Search Trees

Definition A *multiway search tree* is a search tree that allows more than one key in the same node of the tree

Definition A node of a search tree is called an *n-node* if it contains n-1 ordered keys (which divide the entire key range into n intervals pointed to by the node's n links to its children):

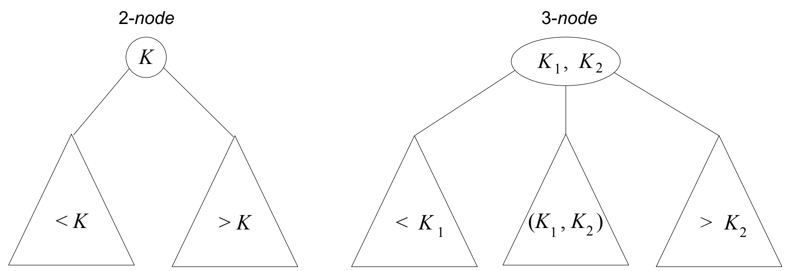


Note: Every node in a classical binary search tree is a 2-node

2-3 Tree

Definition A *2-3 tree* is a search tree that

- may have 2-nodes and 3-nodes
- height-balanced (all leaves are on the same level)



A 2-3 tree is constructed by successive insertions of keys given, with a new key always inserted into a leaf of the tree. If the leaf is a 3-node, it's split into two with the middle key promoted to the parent.

2-3 tree construction – an example

Construct a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7

Analysis of 2-3 trees

- $\log_3 (n + 1) - 1 \leq h \leq \log_2 (n + 1) - 1$
- Search, insertion, and deletion are in $\Theta(\log n)$
- The idea of 2-3 tree can be generalized by allowing more keys per node
  - 2-3-4 trees
  - B-trees

Homework

Exercises 6.1: 1, 2, 3, 7, 9, 11a
Exercises 6.2: 1, 4
Exercises 6.3: 1, 2, 3, 4, 7

Read Sections 6.1, 6.2, 6.3 and 7.4

Next: More representation change methods:
Heaps, Heapsort and Horner's Rule