VARIATIONS ON SEARCH TREES

by

David Toman

University of Waterloo

List of Slides

1
2 Application: Sorting
3 Application: List as a Tree
4 Application: List as a Tree (cont.)
5 Application: List as a Tree (cont.)
6 Variants of Search Trees
7 Searching in Multiple Dimensions
8 Quad and $\mathbf{k}$-$\mathbf{d}$ Trees
9 Splay Trees
10 Splaying a Tree
11 Splay Operations: zig
12 Splay Operations: zig-zag
13 Splay Operations: zig-zig
14 Amortized Analysis
15 The Case Analysis
16 Putting it together
17 Summary

Application: Sorting

IDEA: use search tree to **sort** n elements

- insert elements one-by-one into the tree: $O(n \log n)$
- traverse the tree *in-order*: $O(n)$

$\Rightarrow$ combined: $O(n \log n)$ which is *optimal*

$\Rightarrow$ especially important when we *already have* the tree, e.g., in a DBMS

Application: List as a Tree

IDEA: store a *list ADT* in a tree in order to improve the **`get`**, **`insert`**, and **`delete`** n-th element efficiently

$\Rightarrow$ a linked-list implementation: $O(n)$

$\Rightarrow$ we hope for $O(\log n)$ using a balanced search tree

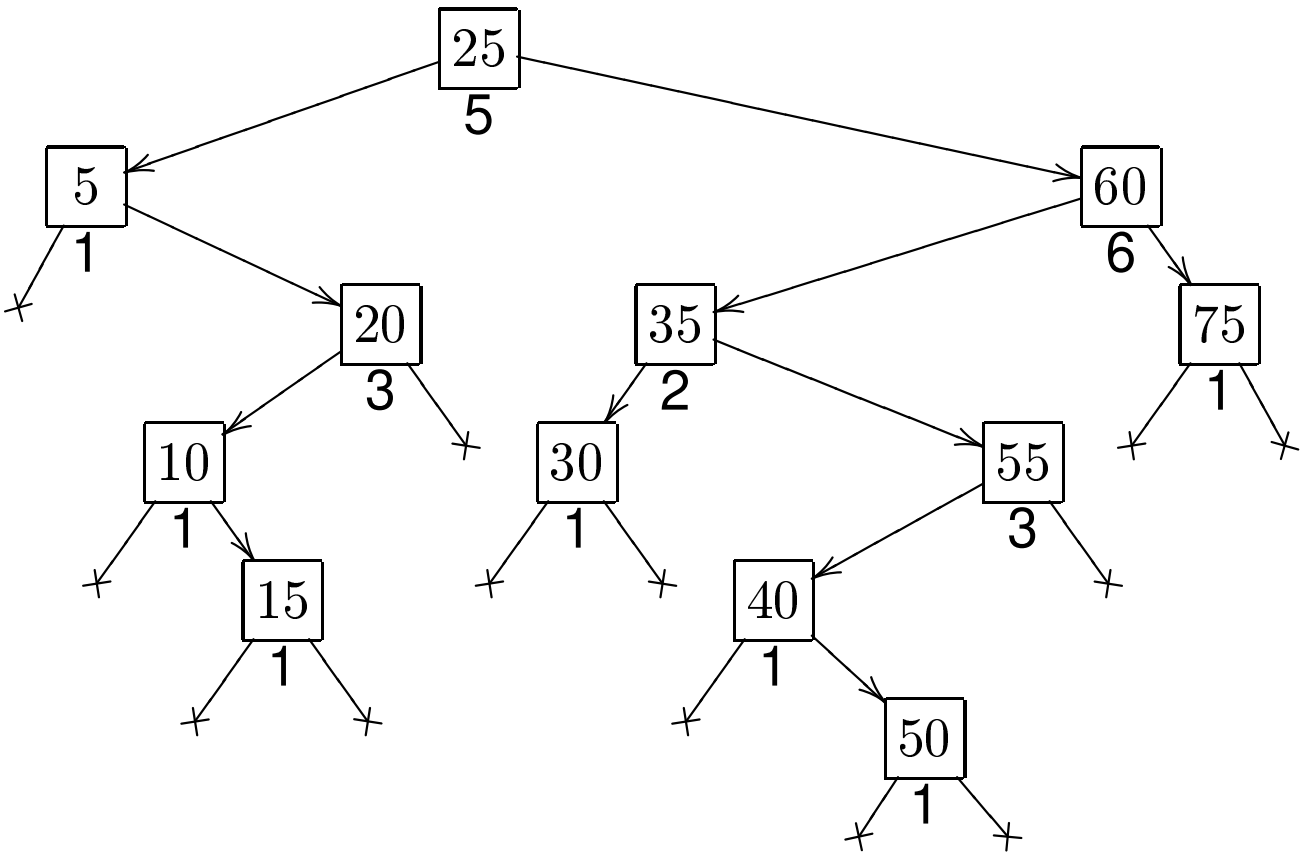
but how to find i-th element in a tree?

AND how to *insert/delete* the i-th element?

SOLUTION: every node in the tree (in addition to the actual value) keeps track of the number of nodes residing in its left subtree (+1)—a *rank* of the node.

$\Rightarrow$ we use *rank* to find the k-th element

Application: List as a Tree (cont.)




```
function find-kth(n:node, k:integer): node
  if (k=n.rank) then find-kth:=n;
  if (k<n.rank) then
       find-kth:=find-kth(n.left,k);
  if (k>n.rank) then
       find-kth:=find-kth(n.right,k-n.rank);
```

Application: List as a Tree (cont.)

- insertions/deletions: $O(\log n)$

  $\Rightarrow$ but must remember to adjust the *rank*s

  $\Rightarrow$ in addition to re-balancing the tree!

- length of a list: $O(\log n)$

  $\Rightarrow$ traversal "to the right" adding *rank*s

- concatenation of lists: $O(\log n)$

  $\Rightarrow$ remember to make sure the result is balanced!

Variants of Search Trees

- Variants of balancing condition(s):
  - $\Rightarrow$ $HB(k)$-trees
  - $\Rightarrow$ 2-3 trees
  - $\Rightarrow$ red-black trees
- Multi-dimensional searching
  - $\Rightarrow$ quad trees
  - $\Rightarrow$ k-d trees
- External storage
  - $\Rightarrow$ B-trees
- On-fly restructuring
  - $\Rightarrow$ Splay-trees

Searching in Multiple Dimensions

PROBLEM: we often need to search using **several** properties of records

$$(\texttt{MAJOR=MATH}) \wedge (\texttt{3} \leq \texttt{TERM} \leq \texttt{6})$$

$\Rightarrow$ inverted file(s)

represented by a tree for efficiency

secondary key searching

EXAMPLE: retrieve all cities that lie within

$$(21.49^\circ \leq \texttt{LAT} \leq 37.41^\circ) \wedge (70.34^\circ \leq \texttt{LON} \leq 75.72^\circ)$$

$\Rightarrow$ there are *many* cities that satisfy the first (or the second) condition *independently*

Quad and k-d Trees

IDEA: make a tree structure that simultaneously
splits isearch ranges in d dimensions

$\Rightarrow$ keys have d components (coordinates)

- Quad tree

  $\Rightarrow$ every node partitions the space to 2^k "quads"
  $\Rightarrow$ access recursively using coordinates
  of the accessed element
  $\Rightarrow$ every node has 2^k children

- k-d tree

  $\Rightarrow$ every node partitions the space to 2 parts
  $\Rightarrow$ round-robin over the dimensions
  $\Rightarrow$ still a binary tree

Splay Trees

IDEA: get rid of explicit balancing (and the necessary balancing information). Instead:

$\Rightarrow$ restructure the tree on every access using fixed rules

- worst case $O(n)$ operations

  $\Rightarrow$ and this *may happen*

- guaranteed $m \cdot O(\log n)$ for m operations

  $\Rightarrow$ **amortized** running time

Splaying a Tree

Operations performed in two steps:

1. the actual operation

   $\Rightarrow$ find, insert, or delete
   — same as in ordinary search tree

2. splaying

   $\Rightarrow$ promote the "accessed" node to the root of the tree
   — using *splaying* rotations

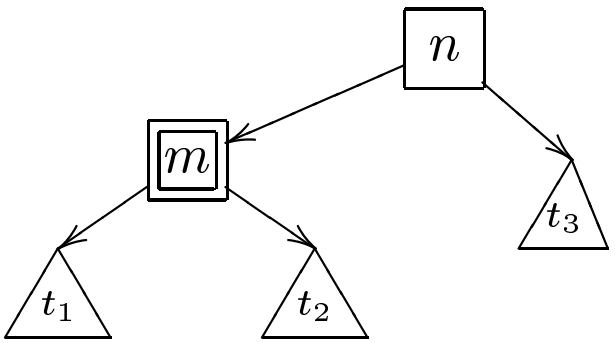
Observation: the "promotion to root" (roughly)

$\rightarrow$ halves the depth of "deep nodes" on the access path
$\rightarrow$ adds two to the depth of "shallow nodes" on the path

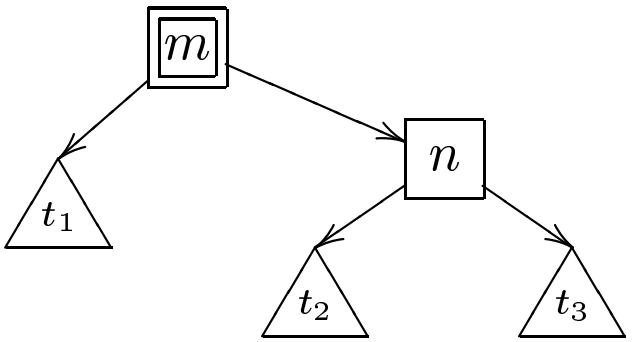
Splay Operations: zig

Situation: parent of m is the root

- before:



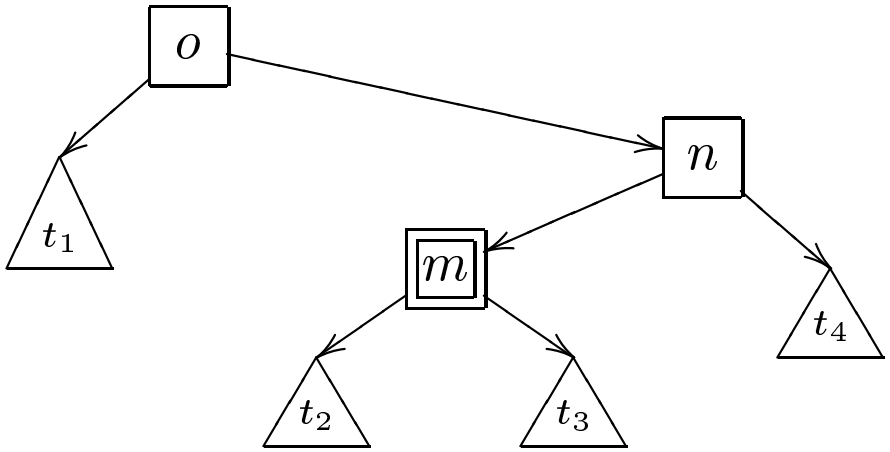
- after:



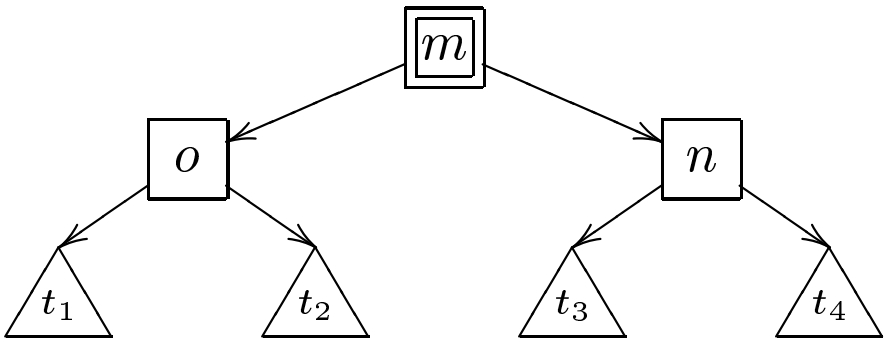
Splay Operations: zig-zag

Situation: m is between its parent n and grandparent o:

- before:



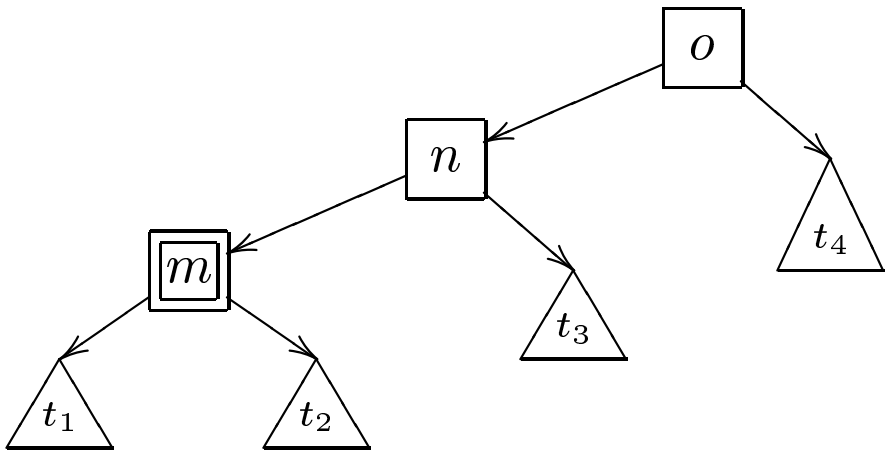
- after:



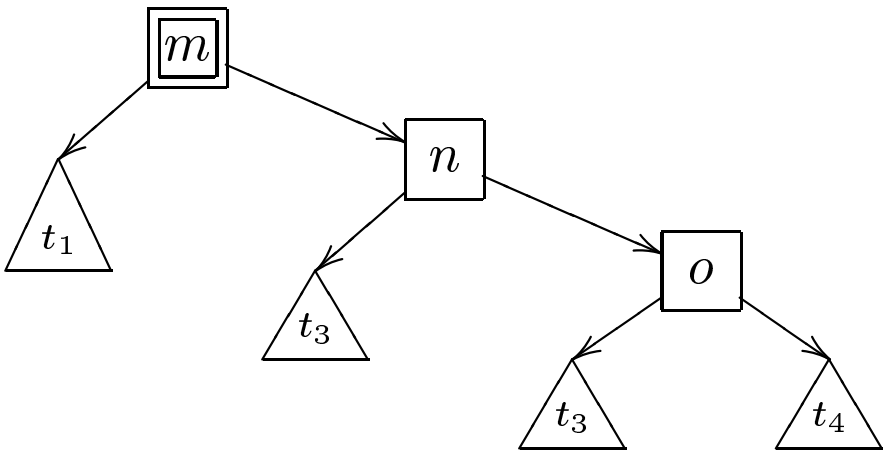
Splay Operations: zig-zig

Situation: m is smaller (bigger) than both its parent n and grandparent o:

- before:



- after:



Amortized Analysis

IDEA 1:

(1) nodes on top of the tree are cheap to access
$\Rightarrow$ we save time (no matter how bad the tree is)

(2) nodes deep on the tree are expensive to access, but at the same time we reduce future access cost to most nodes along the access path to that node
$\Rightarrow$ savings for the future

IDEA 2: we capture the "savings" using a **potential** function for every node:

1. $S(n)$ — the number of descendents of n
2. $R(n) := \log S(n)$
3. total potential function $\sum_{n \in T} R(n)$

The Case Analysis

Case1: zig

$$\begin{aligned} T_{\text{zig}} &= 1 + R_f(m) + R_f(n) - R_i(m) - R_i(n) \\ &\leq 1 + R_f(m) - R_i(m) \qquad \text{as } R_i(n) \geq R_f(n) \\ &\leq 1 + 3(R_f(m) - R_i(m)) \qquad \text{as } R_f(m) \geq R_i(m) \end{aligned}$$

Case 2: zig-zag

$$\begin{aligned} T_{\text{zigzag}} &= 2 + R_f(m) + R_f(n) + R_f(o) - R_i(m) - R_i(n) - R_i(o) \\ &= 2 + R_f(n) + R_f(o) - R_i(m) - R_i(n) \\ &\leq 2 + R_f(n) + R_f(o) - 2R_i(m) \qquad \text{as } R_i(n) \geq R_i(m) \\ &\leq 2R_f(m) - 2R_i(m) \quad \text{as } R_f(n) + R_f(o) \leq 2R_f(m) - 2 \end{aligned}$$

The last step: $a + b \leq c \Rightarrow \log a + \log b \leq 2\log c - 2$

Case 3: zig-zig

$$\begin{aligned} T_{\text{zigzig}} &= 2 + R_f(m) + R_f(n) + R_f(o) - R_i(m) - R_i(n) - R_i(o) \\ &= 2 + R_f(n) + R_f(o) - R_i(m) - R_i(n) \\ &\leq 2 + R_f(m) + R_f(o) - 2R_i(m) \qquad \text{as } R_i(m) \leq R_i(n) \\ &\leq 2 + R_f(m) + R_f(o) - 2R_i(m) \qquad \text{as } R_f(m) \geq R_f(n) \\ &\leq 3R_f(m) - 3R_i(m) \quad \text{as } R_i(m) + R_f(o) \leq 2R_f(m) - 2 \end{aligned}$$

Putting it together

The complete *splaying* at node m is thus the sum of all the individual splay steps:

- for depth of m even:

$$\begin{aligned} T = & \; 3R_0(m) - 3R_2(m) \\ & + 3R_2(m) - 3R_4(m) \\ & \qquad \ddots \\ & \qquad + 3R_{d-2}(m) - 3R_d(m) \\ = & \; 3(R_0(m) - R_d(m)) \end{aligned}$$

- for depth of m odd:

$$\begin{aligned} T &= 1 + 3R_0(m) - 3R_1(m) + [3R_1(m) - 3R_d(m)] \\ &= 1 + 3(R_0(m) - R_d(m)) \end{aligned}$$

As $R_0(m)$ is $\log N$ we have $T \in O(\log N)$ amortized.

Summary

- trees are useful to implement algorithms and ADTs

  $\Rightarrow$ sorting in $O(n \log n)$
  $\Rightarrow$ ordered list with access k-th element in $O(\log n)$

- there are many variants of trees

  $\Rightarrow$ and various questions we want to ask

- *amortization* is a **very** powerful technique for algorithm design and analysis

  $\Rightarrow$ worse than worst-case analysis
  $\Rightarrow$ *better* than average-case analysis
  NO bad input sequences!