SEARCHING AND SEARCH TREES

by

David Toman

University of Waterloo

List of Slides

1
2 Searching for Things
3 Sequential Search
4 Binary Search
5 Searching in Trees
6 Binary Search Tree
7 Finding Entries
8 Insertion
9 Deletion (Example)
10 Deletion (cont.)
11 What's the Complexity?
12 How to make Trees Balanced?
13 Balanced (AVL) Trees
14 How to Regain Balance?
15 How to Regain Balance? (II)
16 Summary

Searching for Things

Basic Problem: how to find things we stored

$\Rightarrow$ with a given identifier (key)

A **dictionary** ADT

- **`make_empty`**

  $\Rightarrow$ creates a new (empty) dictionary)

- **`find`**(k)

  $\Rightarrow$ finds a record identified by k

- **`insert`**(k)

  $\Rightarrow$ inserts (new) record identified by k

- **`delete`**(k)

  $\Rightarrow$ deletes record identified by k

Variations: records have (other) values
multiple records with the same identification

Sequential Search

IDEA: store elements in a *list*

$\Rightarrow$ e.g., in the order of their arrival

- insertion: $O(1)$
- deletion: $O(n)$ (even on average)

Improvements:

- if we know the probabilities of accessing elements

  $\Rightarrow$ order the list accordingly
- reorganization heuristics (e.g., Move To Front)

  $\Rightarrow$ cashing/cache-eviction policies

Binary Search

IDEA: use ordering on keys to speed-up search

$\Rightarrow$ assumes keys *are ordered*

Searching in an ordered array—**binary search**:

```
function BinS(A[i..j]:array, k:key): boolean
  if (j<i) then BinS := false;
  else
    m := (i+j)/2;          { floor }
    if (k<A[m]) then
       BinS := BinS(A[i..m-1],k)
    else
    if (k>A[m]) then
       BinS := BinS(A[m+1..j],k)
    else
    BinS := true;
```

running time: $f(n) = f(n/2) + O(1) \rightarrow f \in O(\log n)$

Searching in Trees

IDEA: use a *tree* structure to implement an *ordered array*

- a *binary tree*: hope for $O(\log n)$ dictionary operations

$$\texttt{tree} = [\,] + \tau \times \texttt{tree} \times \texttt{tree}$$

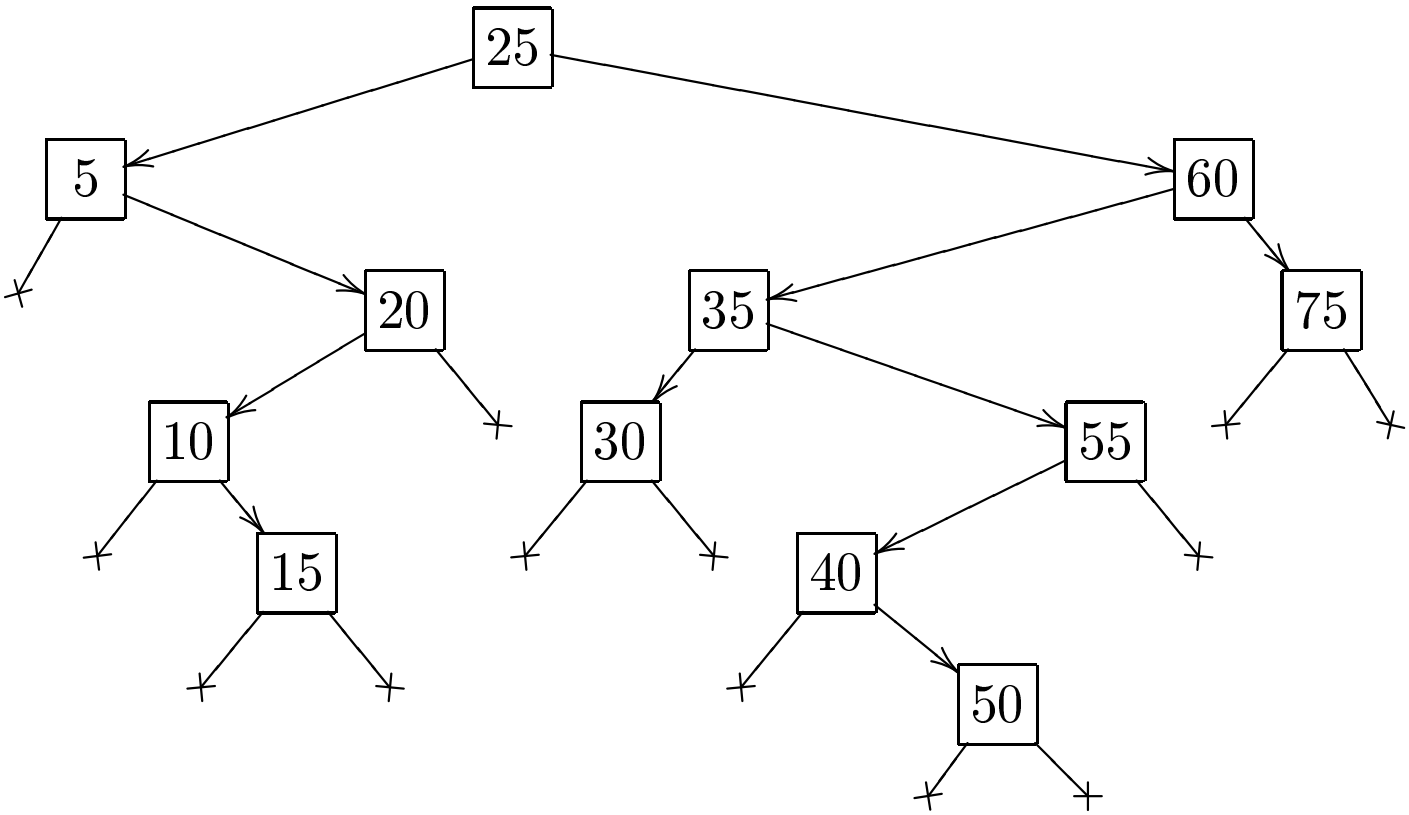
  where τ is the type of keys (we assume integers)

- with an additional **search tree property**:

  if $n = (k, t_1, t_2)$ is a node in a search tree then
  (1) all nodes in t_1 contain keys smaller than k
  (2) all nodes in t_2 contain keys bigger than k

NOTE: all "data" is stored in the internal nodes;
leaves serve only as place-holders

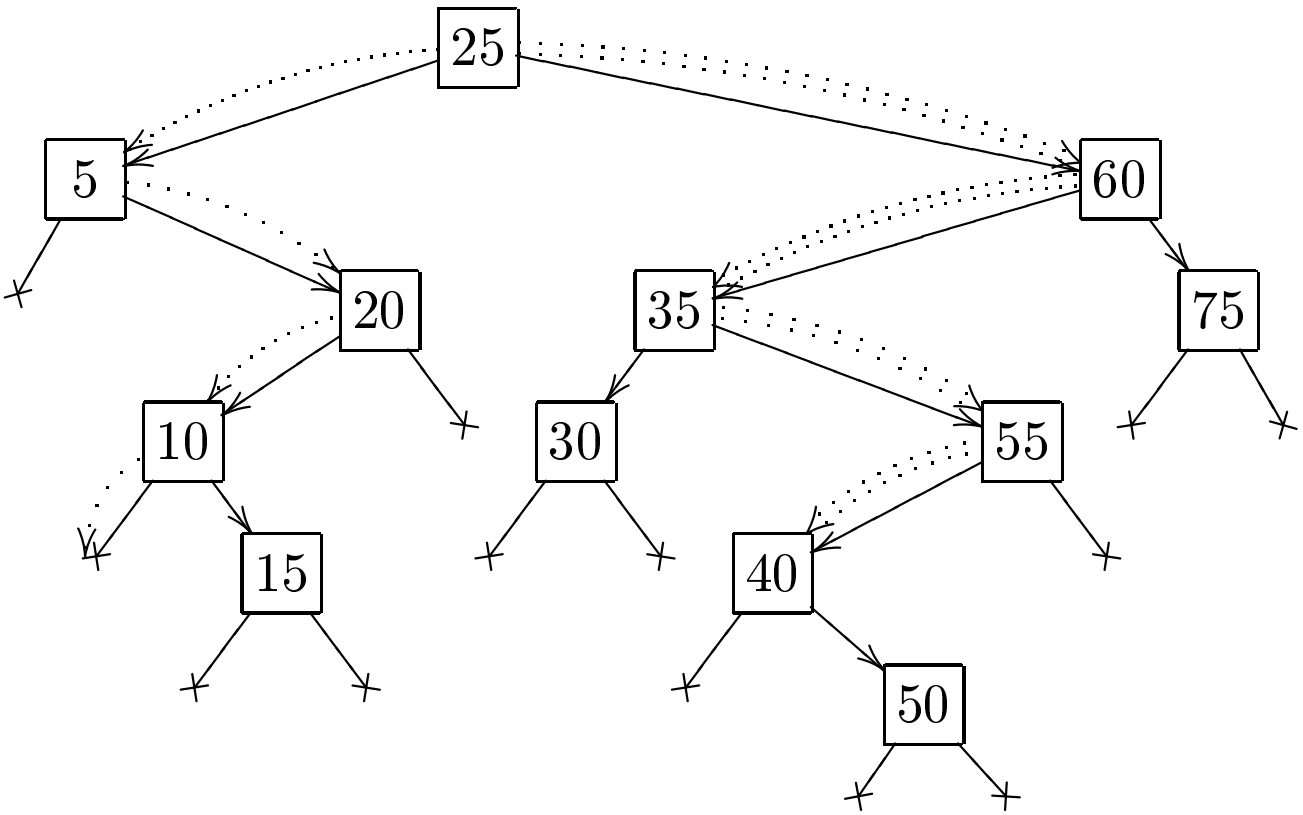
Binary Search Tree



5 10 15 20 25 30 35 40 50 55 60 75

Finding Entries

find(8) (result=false); **find**(40) (result=true)

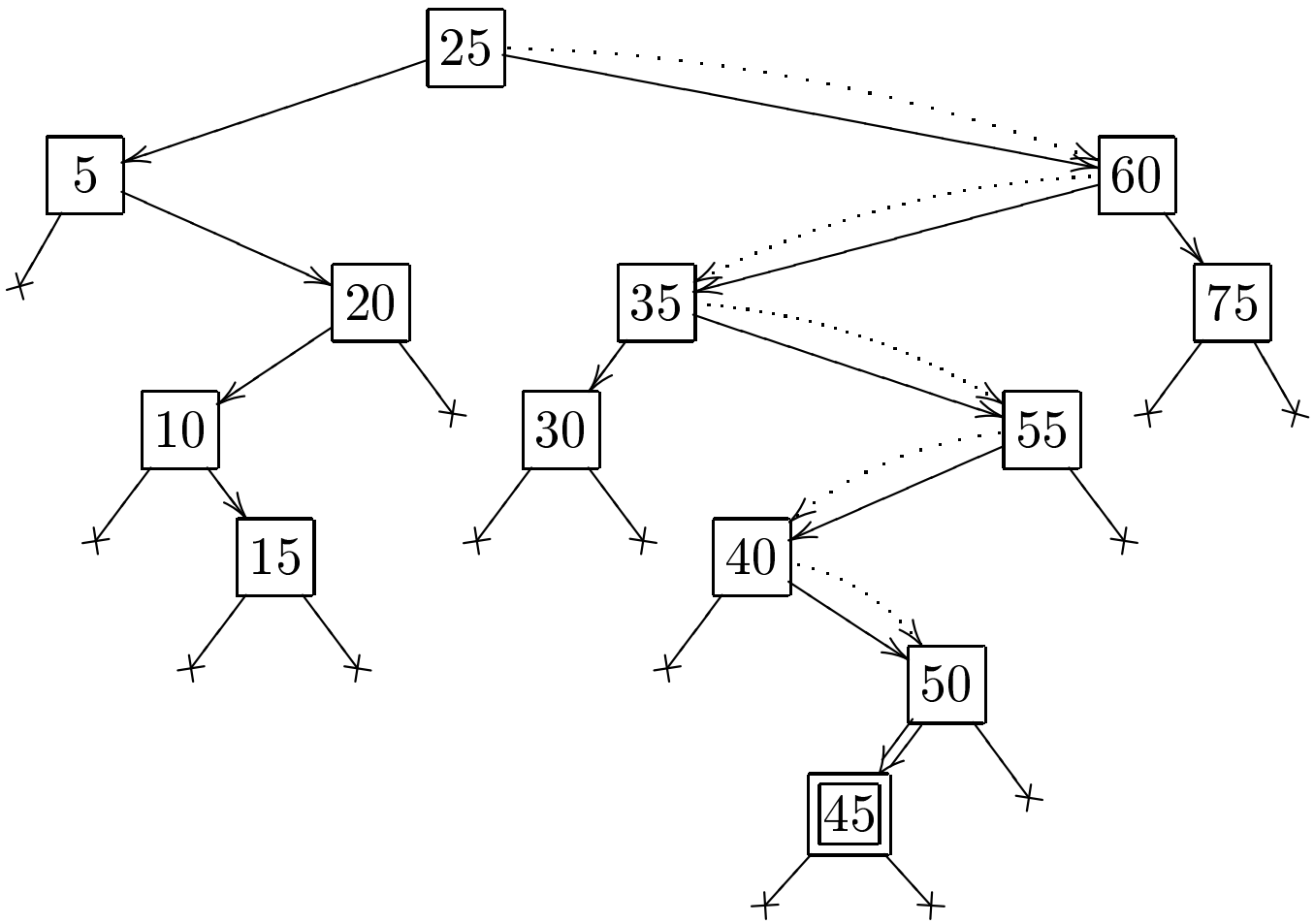


Insertion

Similar to **`find`**

$\Rightarrow$ when not found, add to the last visited node

`insert`(45):

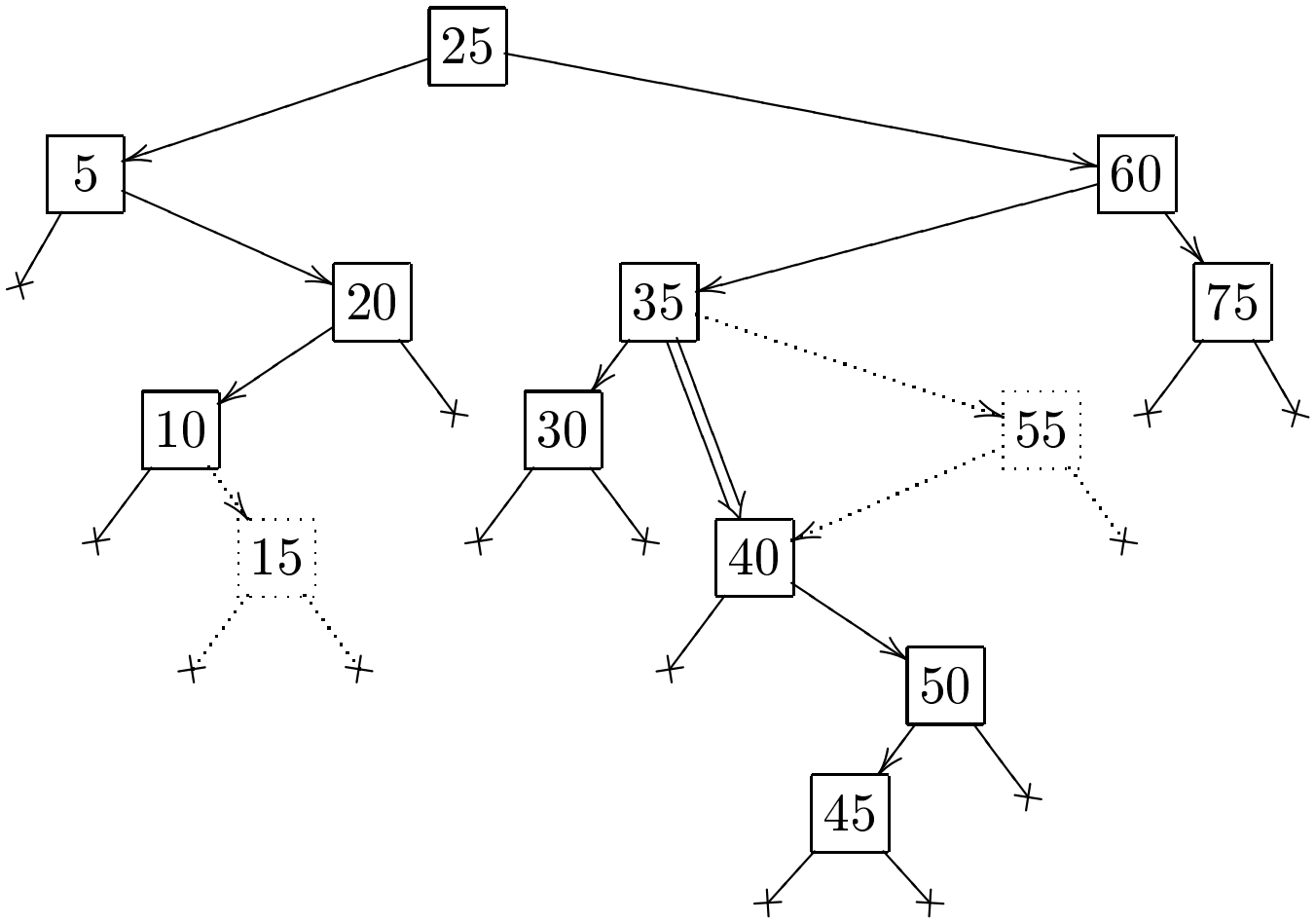


Deletion (Example)

Run **`find`**; when found try to remove the node

Easy cases: the deleted key has at most one child:

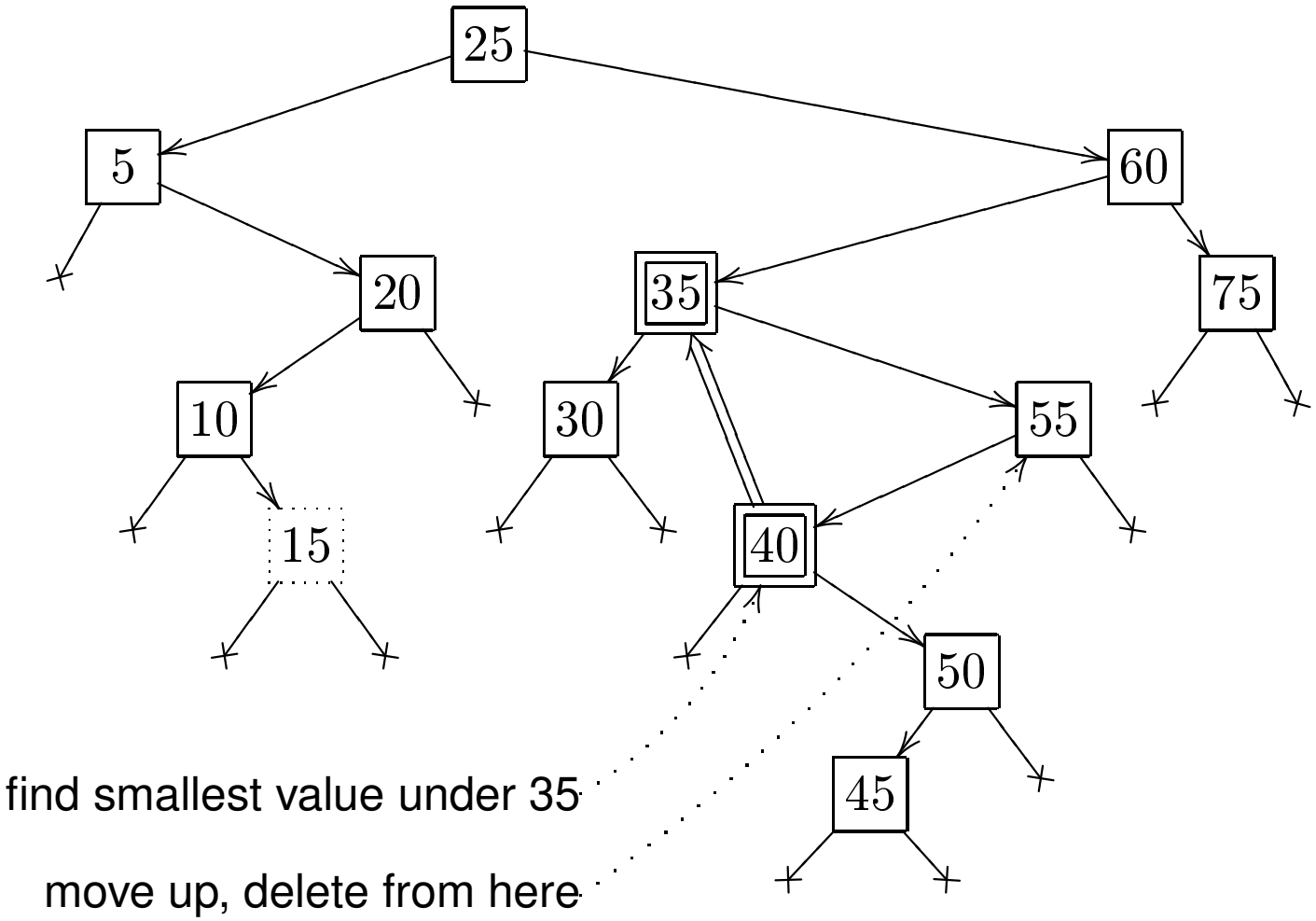
`delete`(15); **`delete`**(55)



Deletion (cont.)

The hard case: the deleted key has both children:

delete(35);



What's the Complexity?

- all operations run in $O(k)$ for k the **height** of the tree
  $\Rightarrow$ defined recursively on the structure of the tree

$$h(t) = \begin{cases} 0 & t = [\,] \\ 1 + \max(h(t_1), h(t_2)) & t = \texttt{node}(v, t_1, t_2) \end{cases}$$

  $\Rightarrow$ extremal cases for trees with n nodes?

  worst case: the tree is a *list*: $O(n)$

  best case: the tree is **balanced**: $O(\log n)$

- MAKE and KEEP search trees *almost* balanced

How to make Trees Balanced?

- keep almost balanced: height within $O(\log n)$

  $\Rightarrow$ AVL trees

- restructure on fly

  $\Rightarrow$ Splay Trees

- keep perfect balance

  $\Rightarrow$ B-trees

Balanced (AVL) Trees

Adel'son-Vel'skii and Landis, 1962

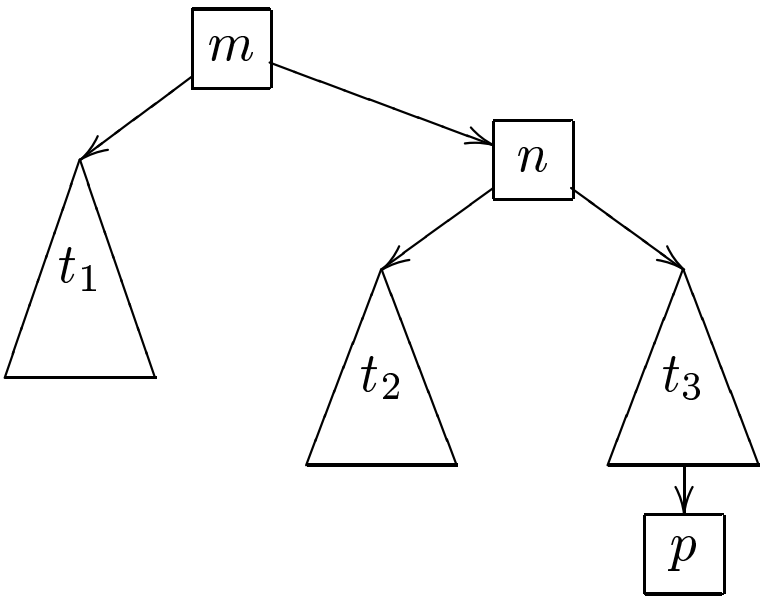
IDEA: allow only trees that obey

> for every node: the difference of heights
> of the left and right subtrees differs by at most 1.

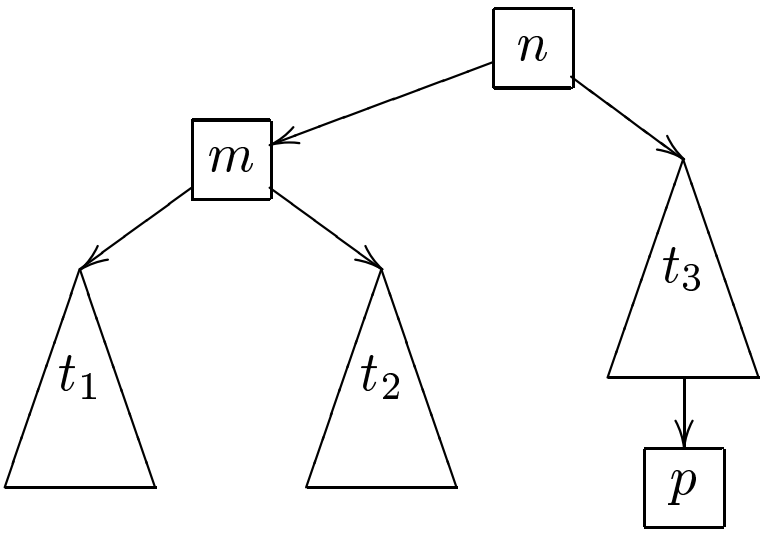
Theorem. The height of an AVL tree storing n keys is $O(\log n)$.

How to Regain Balance?

- node m fails the AVL test after adding node p:

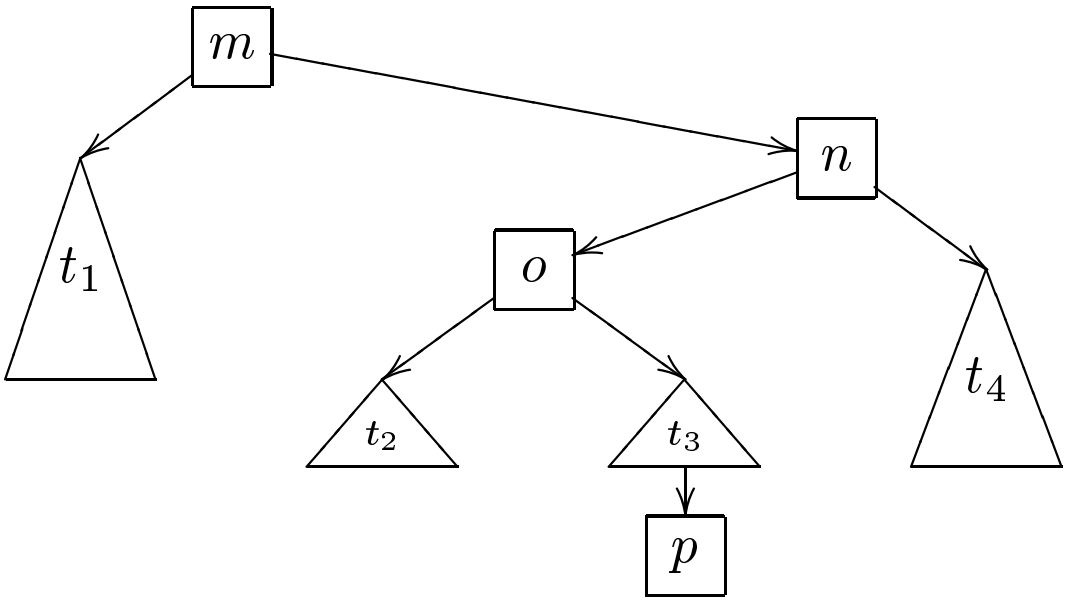


- single rotation regains balance:

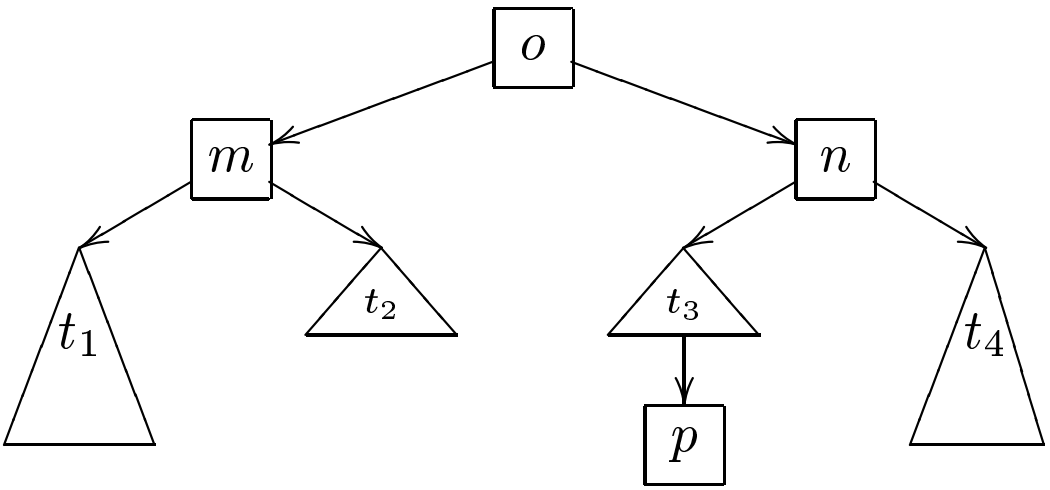


How to Regain Balance? (II)

- node m fails the AVL test after adding node p:



- double rotation regains balance:



Summary

- Searching in a *dictionary* ADT

  $\Rightarrow$ many many many applications

  tables in databases

  symbol tables in compilers

  ...

- Key idea: binary search on *linearly ordered* keys

  $\Rightarrow$ this gives us $O(\log n)$ performance

- Balanced trees

  $\Rightarrow$ dynamic version od sorted array