Lecture 17

Trees

TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
8 november 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

17.1

Content

Innehåll

17.2

1 Symbol tables

Symbol tables

- Abstraction of key-value pairs
  - *Insert* a value with a specified key
  - Given a key, *search* the corresponding value

17.3

1.1 Abstract data types

ADT Set

- Domain: sets of keys
- Typical operations:
  - size() the number of keys in the set
  - isEmpty() check whether the set is empty or not
  - contains(k) returns **true** if k is in the set, otherwise **false**
  - put(k) inserts k in the set
  - remove(k) removes k from the set

17.4

ADT Map

- Domain: sets of items/pairs (*key*, *value*) The sets are partial functions that map keys to values
- Typical operations:
  - size() the number pf pairs in the set
  - isEmpty() checks whther a set is empty
  - get(k) retrieve the information acciated to k or **null** if the key does not exist
  - put(k,v) adds (k,v) to the set and returns **null** if k is new; otherwise it replaces the value of v and returns the old value
  - remove(k) removes element (k,v) from the set and returns v; otherwise it returns **null** if the element does not exist

17.5

ADT Map

- Example:
  - course database: (code, name)
  - memory allocation (address, value)
  - matrix: ((row, column), value)
  - Lunch menu: (day, right)
- Static Mapping: no updates allowed
- Dynamic Mapping: uppdates *are* allowed

17.6

ADT Dictionary

- Domain: sets of pairs (*key*, *value*) The sets are relations between keys and values!
- Typical operations:
  - size() number of pairs in the set
  - isEmpty() checks whether a set is empty
  - find(k) returns any element associated to key k or **null** if no element matches with k
  - findAll(k) returns all elements with key k
  - insert(k,v) adds (k,v) to the set and returns the new element
  - remove(k,v) removes and returns pair (k,v); returns **null** if the element does not exist
  - entries() returns the collection of all elements

17.7

ADT Dictionary

- Example:
  - Swedish-english dictionary ..., (jakt, yacht), (jakt, hunting), ...
  - Telephone directory (several numbers allowed)
  - Relation between LiU ID and completed courses
  - Lunch menu (with more choices): (day, right)
- Static Dictionary: no updates allowed
- Dynamic Dictionary: updates *are* allowed

17.8

1.2 Implementation

Implementation: Map, Dictionary

- Table/array: sequence of memory areas of the same size
  - Unordered: no particular order between $T[i]$ and $T[i+1]$
  - Ordered: ... but here $T[i] < T[i+1]$
- Linked lists
  - Unordered
  - Ordered
- (Binary) search tree
- Hashing
- Skip-listing

17.9

Table representation of Dictionary

Unordered table:
find using *linear search*

- search failed: n comparisons $\Rightarrow O(n)$ time
- successful search, in the worst case: n comparisons $\Rightarrow O(n)$ time
- successful search, average case with uniform distribution of the requests: $\frac{1}{n}(1+2+\ldots+n) = \frac{n+1}{2}$ comparisons $\Rightarrow O(n)$ time

17.10

Table representation of Dictionary

Ordered table (the keys are linearly ordered):
find by *binary search*

- lookup: $O(\log n)$ time
- . . . updates are expensive!!

17.11

2 Trees

2.1 Basic concepts

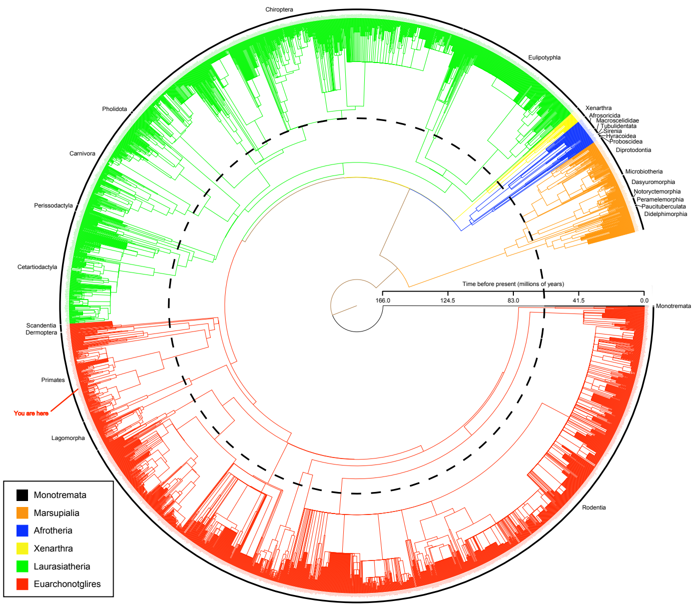
Why trees?

Tree structures arise naturally in many situations

- File system
- Hierarchical classification system
- Decision trees
- Hierarchical organization of
  - Organizations: department, area, group
  - Document: book, chapter, section
  - XML-document
- For representing order or priorities

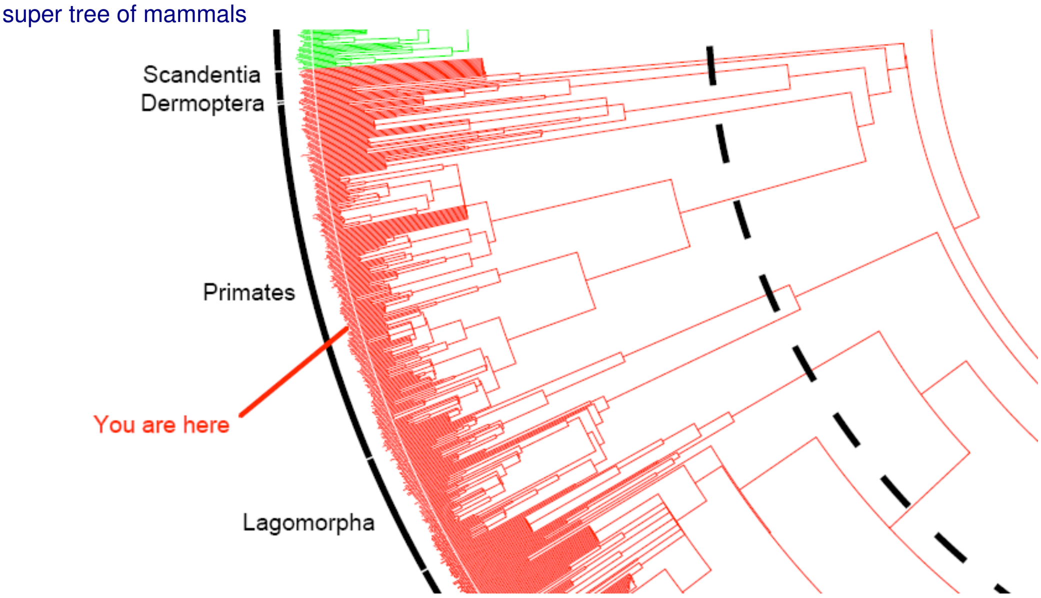
17.12

A super tree of mammals



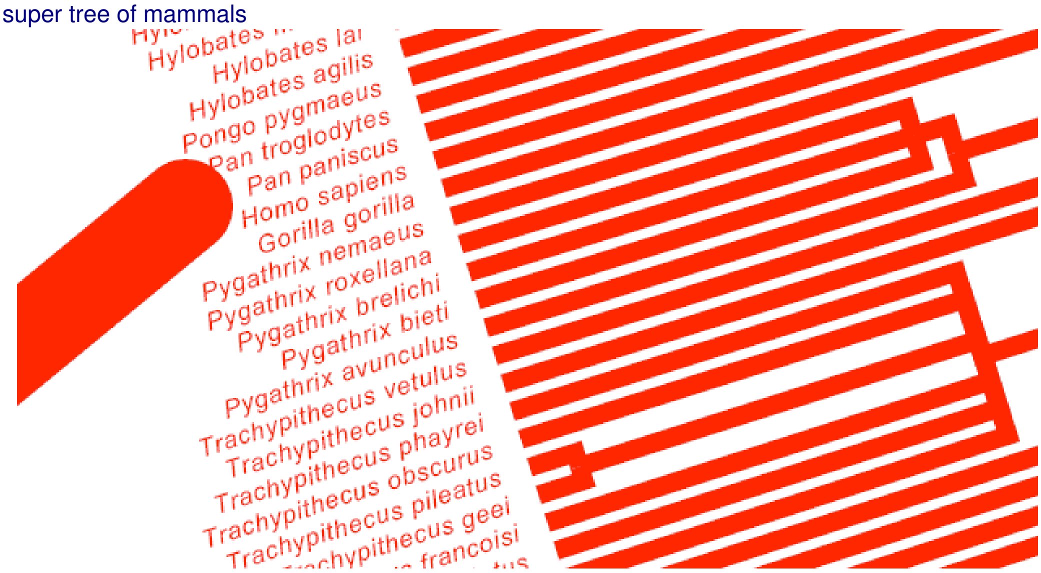
17.13

A super tree of mammals



17.14

A super tree of mammals



17.15

Terminology

- A *(rooted) tree* $T = (V, E)$ comprises a set V of *nodes* and *edges* E, where an edge is a pair $(u, v) \in V \times V$.
- Nodes (sometimes called *corners*) $v \in V$ store data in a *parent-child* relationship.
- A parent-child relationship between u and v is shown as directed edge $(u, v) \in E$, when the direction is from u to v.
- Each node has at most one parent node; but can have many *siblings*.
- At most, there is one node with no parent – *root node*.

17.16

More terminology

- A node *degree* is the number of node's children .
- A node with 0 child is a *leaf* or a *outer/external* node. Other nodes are *inner/internal*.
- A path is a sequence of nodes $(v_1, v_2, \ldots, v_k)$, where $k > 0$ such that v_i, v_{i+1} is an edge for $i = 1, \ldots, k-1$.
- The length of a path $(v_1, v_2, \ldots, v_k)$ is $k-1$. Note that the length of the path (v_1) is 0.
- A node n is a *parent* to another node v iff there exists a path from n to v in T.
- A node n is a *descendant* to a node v iff there is a path from v to n in T.

17.17

Again, more terminology

- *Depth* $d(v)$ of a node v is the length of the path from root to v.
- *Height* $h(v)$ of a node v is the length of the longest path from v to any descendant of v.
- *Height* $h(T)$ of a tree T is the height of the root node.

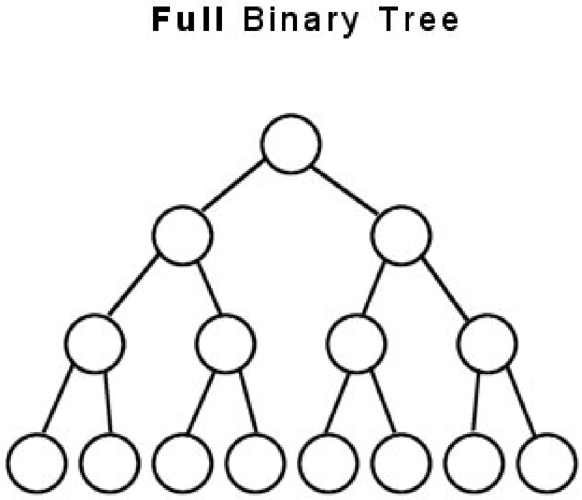
17.18

Different tree types

- *Ordered tree*: linear order between each node's children
- *Binary tree*: ordered tree with degree≤ 2 for each node. A node can have a left child and a right child
- *Empty binary tree*: binary tree with no nodes
- *Full binary tree*: not empty, the degree of each node (number of children) is either 0 or 2. Consequence: the number of leaves = 1 + the number of internal nodes
- *Perfect binary tree*: full binary tree, all leaves have the same depth. Consequence: the number of leaves = 2^h for a perfect binary tree of height h
- *Complete binary tree*: approximation to perfect tree for $2^h \leq n < 2^{h+1} - 1$. In the distance $h-1$, every level, except possibly the last, is completely filled and all nodes are as far left as possible.

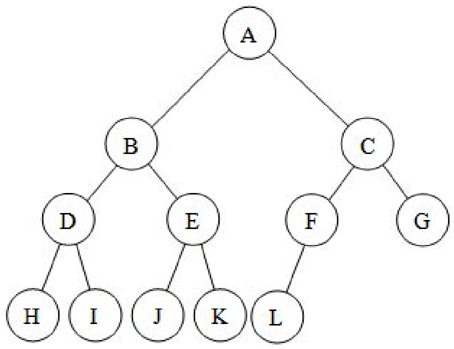
17.19

Full binary tree



17.20

Complete binary tree



17.21

2.2 ADT trees

Operations on a node v within a tree T

- *parent*(v) returns the parent of v, **error** if v is the root node
- *children*(v) returns te set of children of v
- *firstChild*(v) returns the first child of v or **null** if v is a leaf
- *rightSibling*(v) returns the right hand sibling of v or **null** if ther isn't
- *leftSibling*(v) returns the left hand sibling of v or **null** there isn't
- *isLeaf*(v) returns **true** iff v is a leaf
- *isInternal*(v) returns **true** iff v is not a leaf

- *isRoot*(v) returns **true** iff v is the root node
- *depth*(v) returns the depth of v in T
- *height*(v) returns the height of v in T

17.22

Operations on a whole tree T

- *size*() returns the number of nodes in T
- *root*() returns the root node of T
- *height*() returns the height of T

Additionally, for *a binary tree*

- *left*(v) returns the left hand child of v or **error**
- *right*(v) returns the right hand child of v or **error**
- *hasLeft*(v) checks whether v has a left hand child
- *hasRight*(v) checks whether v has a right hand child

17.23

2.3 Representation of binary trees

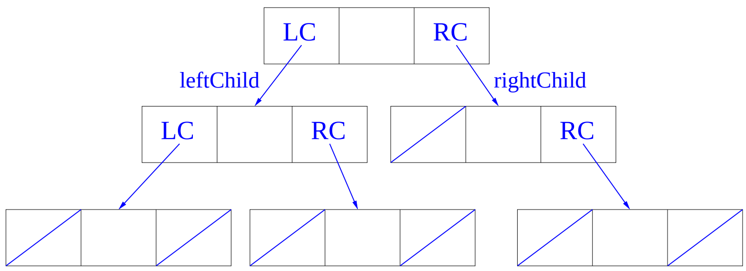
A linked representation

class treeNode<T>   *nodeInfo*: T   *N*: integer   *children*: array[1..*N*] of treeNode<T>

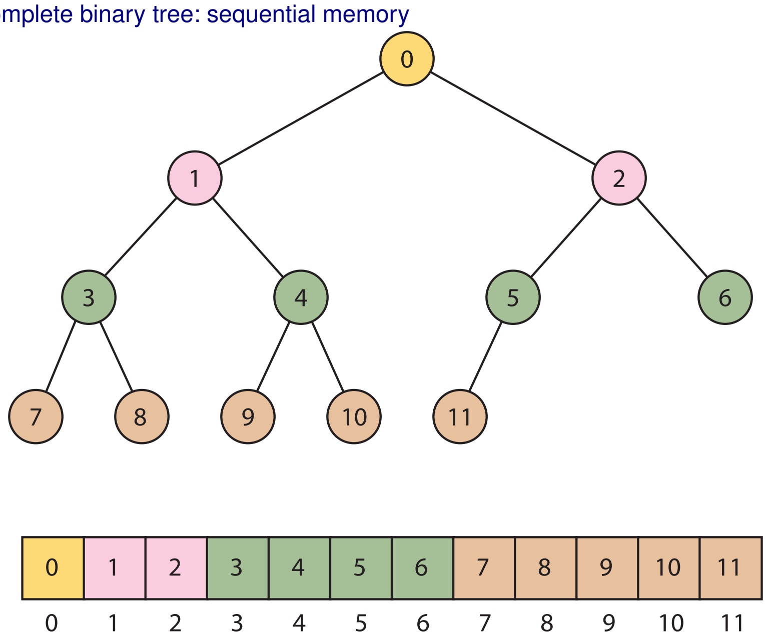
Or for binary trees

class treeNode<T>   *nodeInfo*: T   *leftChild*: treeNode<T>   *rightChild*: treeNode<T>

17.24



Complete binary tree: sequential memory



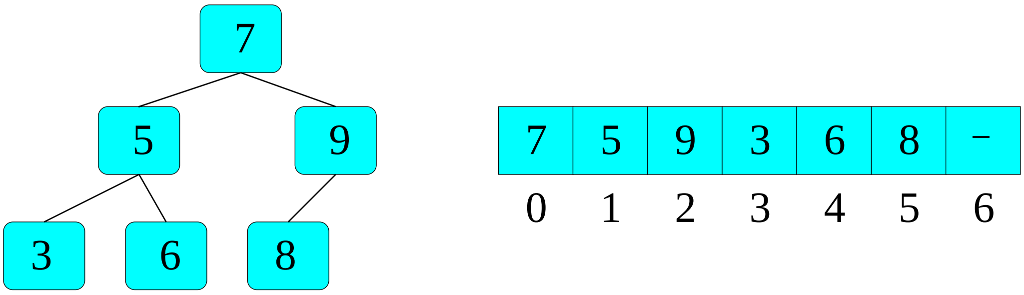
17.25

Sequential memory

Use a table<key,info>[0..n-1]

- leftChild$(i) = 2i+1$ (returns **null** if $2i+1 \geq n$)
- rightChild$(i) = 2i+2$ (returns **null** if $2i+2 \geq n$)
- isLeaf$(i) = (i < n)$ and $(2i+1 > n)$
- leftSibling$(i) = i-1$ (returns **null** if $i = 0$ or odd(i))
- rightSibling$(i) = i+1$ (returns **null** if $i = n-1$ or even(i))
- parent$(i) = \lfloor (i-1)/2 \rfloor$ (returns **null** if $i = 0$)
- isRoot$(i) = (i = 0)$

17.26



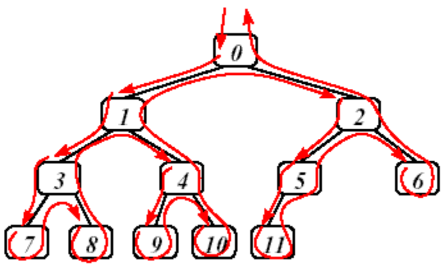
2.4 Traversing a tree

Traversing a tree

Consider a tree T as if it is a building: nodes are rooms, edges are doors, root node is the entry How to explore an unknown (bike-free) labyrinth and step out again? *make sure that there is always a wall to the right*

Generic routines for traversing trees:

procedure VISIT(node v)
  for all u ∈CHILDREN(v) **do**
    VISIT(u)



Call visit$(root(T))$, each node in T will be visited exactly once.

17.27

Traversing trees

procedure PREORDERVISIT(node v)
  DOSOMETHING(v) ▷ before each child node
  for all u ∈CHILDREN(v) **do**
    PREORDERVISIT(u)

procedure POSTORDERVISIT(node v)
  for all u ∈CHILDREN(v) **do**
    POSTORDERVISIT(u)
  DOSOMETHING(v) ▷ after all children

17.28

Traversing trees (only binary trees)

procedure INORDERVISIT(node v)
  INORDERVISIT(LEFTCHILD(v))
  DOSOMETHING(v) ▷ after all left hand descendants
  INORDERVISIT(RIGHTCHILD(v))

17.29

Traversing trees

procedure LEVELORDERVISIT(node v)
  $Q \leftarrow$ MAKEEMPTYQUEUE()
  ENQUEUE(v, Q)
  while not ISEMPTY(Q) **do**
    $v \leftarrow$ DEQUEUE(Q)
    DOSOMETHING(v)
    for all $u \in$ CHILDREN(v) **do**
      ENQUEUE(u, Q)

Also known as width first. 17.30

2.5 Binary search trees

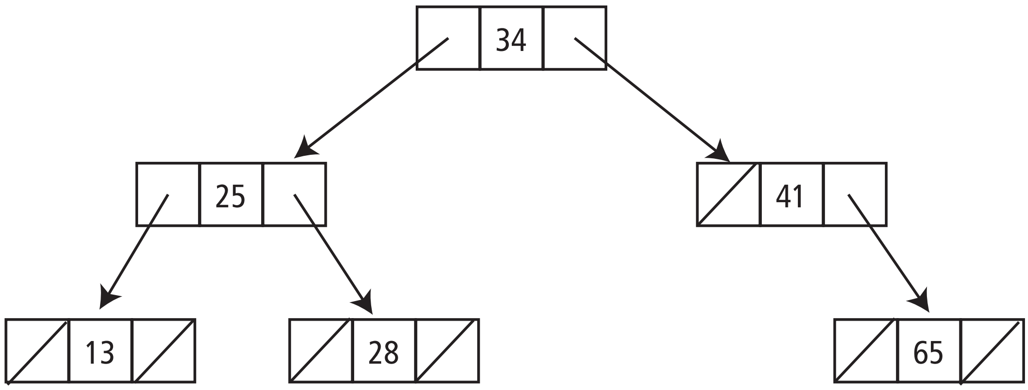
Binary search trees

Ett *A binary search tree* (BST) is a binary tree such that:

- the information associated with a node is linearly ordered e.g. (key,value).

The key in each node is:

- greater than (or equal to) the keys of all left descendants, and
- less than (or equal to) the key of all right descendants.



17.31

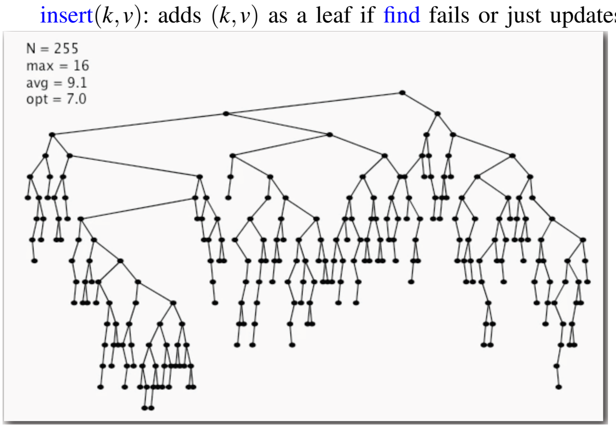
ADT Map through binary search trees

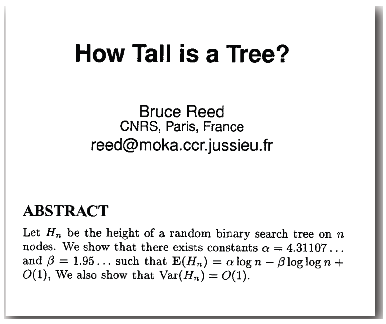
procedure FIND(k, v)
  if KEY(v) $= k$ **then return** k
  else if $k <$ KEY(v) **then**
    FIND(k,LEFTCHILD(v)) ▷ Processing missing if no leftChild
  else
    FIND(k,RIGHTCHILD(v)) ▷ Processing missing if no rightChild

Worst case: HEIGHT(T) + 1 comparisons. 17.32

ADT Map through binary search trees

insert(k, v): adds (k, v) as a leaf if find fails or just updates the corresponding node if find succeeds



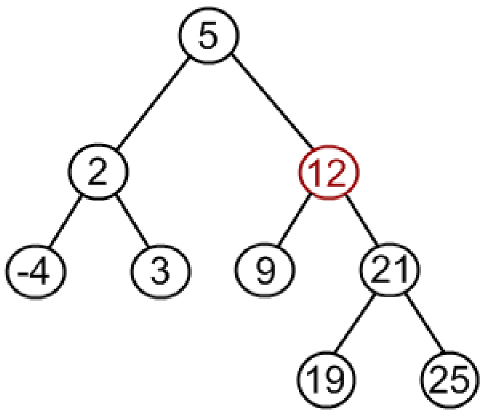

How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT

Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathrm{E}(H_n) = \alpha \log n - \beta \log\log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

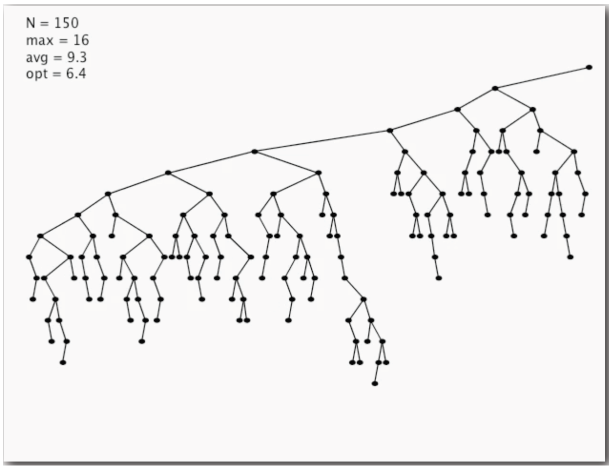
Worst case: HEIGHT(T) + 1 comparisons. (Exponential when the keys are inserted in random order.) 17.33



ADT Map through binary search trees

remove(k): find, then...

- if v is a leaf, remove v
- if v has a child u, replace v with u
- if v has 2 children, replace v with its successor in the order
- (alternatively, with its predecessor in the order)



Surprising result: The trees no longer random ⇒ time $\sqrt{\text{HEIGHT}}$ per operation!

Worst case: $\text{HEIGHT}(T)+1$ comparisons. 17.34

ADT Map through binary search trees

Remove node 12 from the tree. 17.35

ADT Map through binary search trees

E.g. 19 is the successor value for 12 in the sub-tree. 17.36

ADT Map through binary search trees

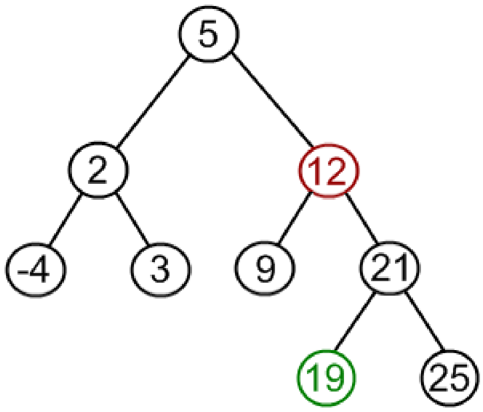
Replace node 12 by node 19. 17.37

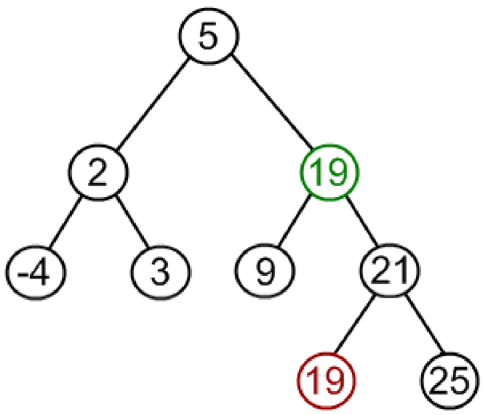
ADT Map through binary search trees

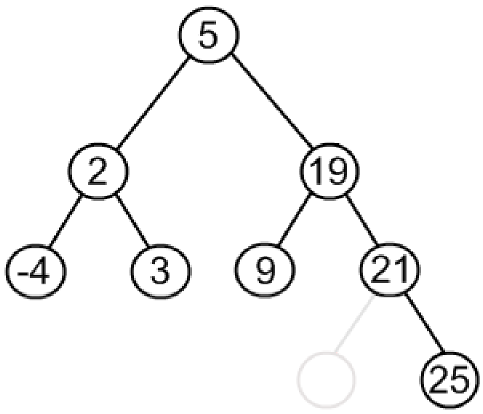
Delete the duplication of node 19. 17.38

Binary search trees are not unique

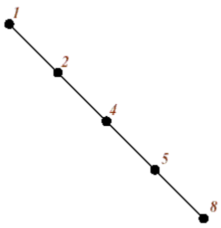
The same data can generate different binary search trees


5
2
12
-4
3
9
21
19
25

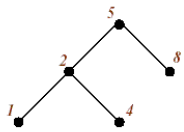

5
2
19
-4
3
9
21
19
25


5
2
19
-4
3
9
21
25

insert: 1,2,4,5,8



insert: 5,2,1,4,8



17.39

Successful lookup

BST if the worst case

- BST degenerated into linear sequence
- the expected number of comparisons is $(n+1)/2$

Balanced BST

- the depth of the leaves do not differ by more than 1
- $O(\log_2 n)$ comparisons

17.40

Let's keep them balanced!

Some common balanced trees:

- AVL-tree
- (2,3)-tree, (a,b)-tree,
- ... Red-Black trees, B-tree
- Splay-trees

17.41

2.6 AVL-trees

AVL-trees

- Self-balancing BST/height balanced tree BST
- AVL = Adelson-Velskii and Landis, 1962
- Idea: Keep track of balance information in each node
- AVL-propertyFor each internal node v in T the heights of the two subtrees of v differ by at most one
  ... in another word... For each internal node v in T, $b(v) \in \{-1, 0, 1\}$ where

$$b(v) = \text{height}(\text{leftChild}(v)) - \text{height}(\text{rightChild}(v))$$

Otherwise, a rebalancing is needed to restore this property. 17.42

Maximal height of AVL-tree

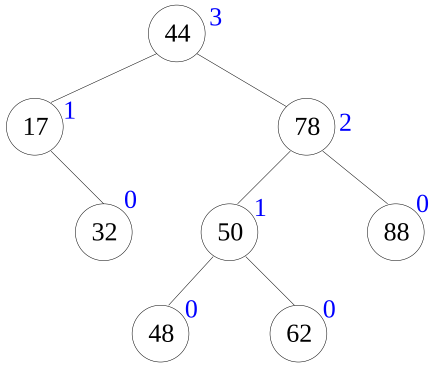
Proposition 1. The height of a AVL-tree having n elements is $O(\log n)$.

What will the result ...

Proposition 2. We can do find, insert and remove in a AVL-tree in time $O(\log n)$ while preserving the AVL-property.

17.43

Example: a AVL-tree



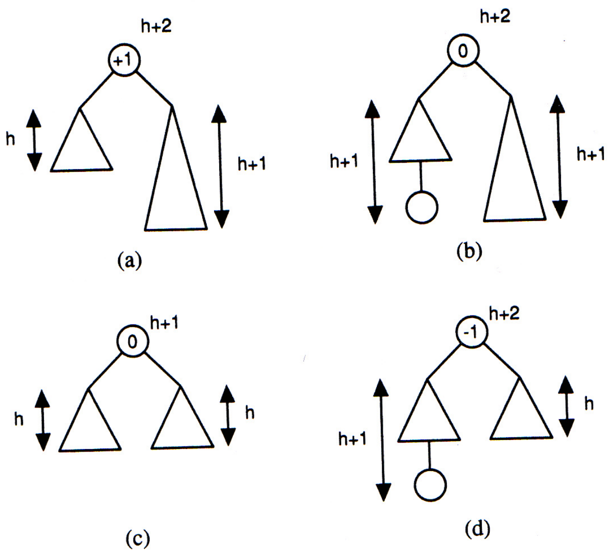
17.44

Inserting in a AVL-tree

- The new node leads to change the tree height, which must be balanced.
  - One can keep track of the height of the trees in different ways:
    - Storing height explicitly in each node
    - Storing the balance factor for nodes
- The change is usually described as a right or left rotation of a subtree.
- It is enough with one rotation to get the tree back into balance.

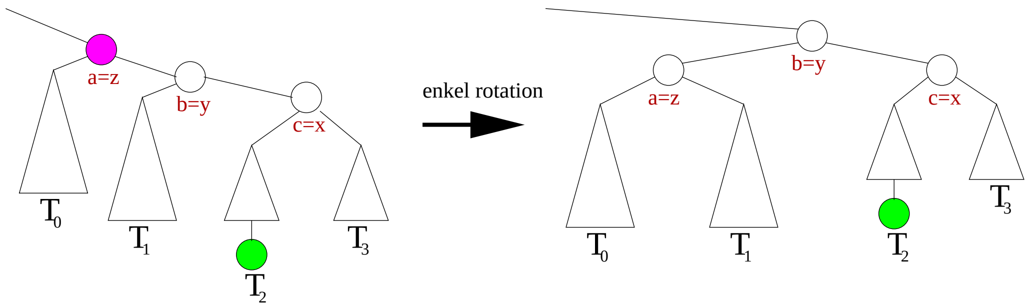
17.45

Inserting in AVL-trees (simple cases)



17.46

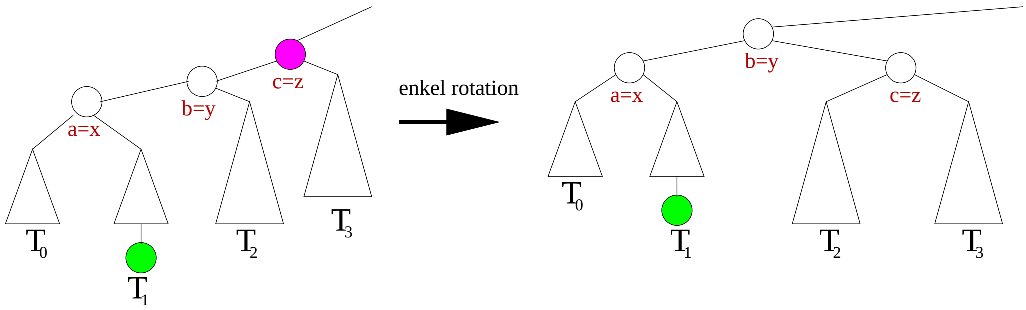
Four different rotations



If $b = y$ it is called a simple rotation."Rotate up y over z"

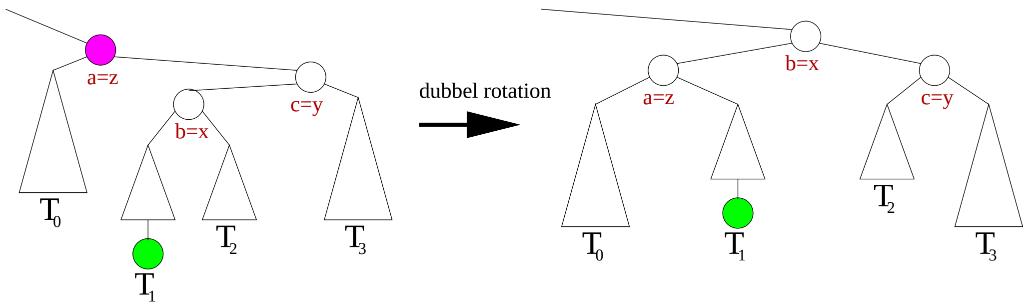
17.47

Four different rotations



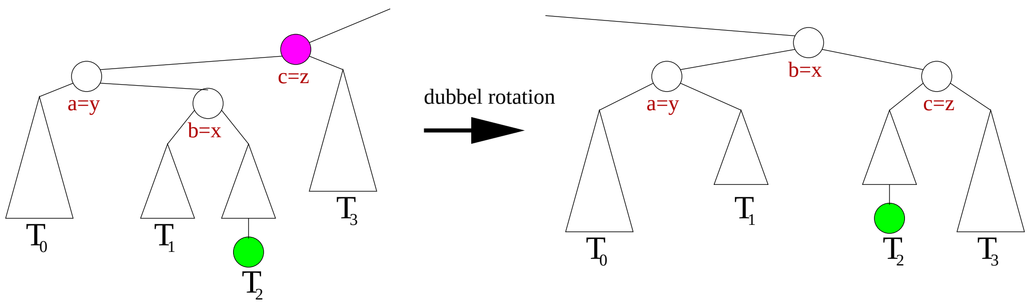
If $b = y$ it is called a simple rotation."Rotate up y over z" 17.48

Four different rotations



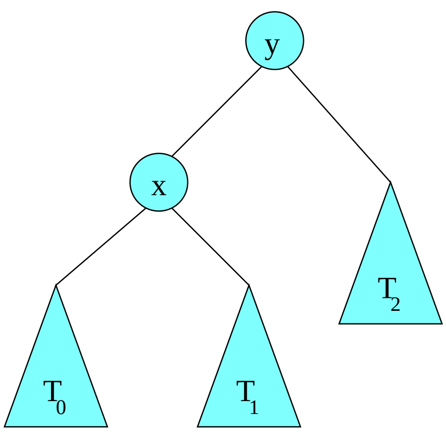
If $b = x$ it is called a double rotation."Rotate up x over y and then over z" 17.49

Four different rotations



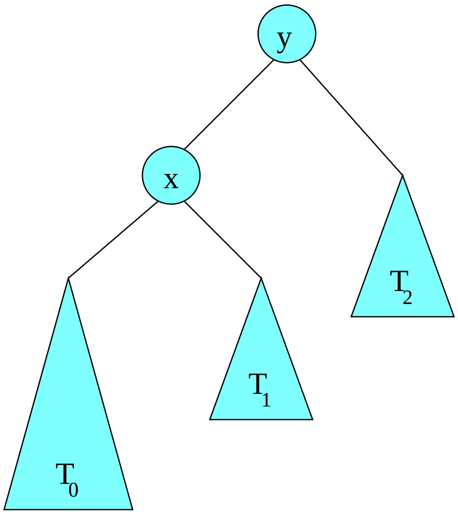
If $b = x$ it is called a double rotation."Rotate up x over y and then over z" 17.50

Another way to describe balancing



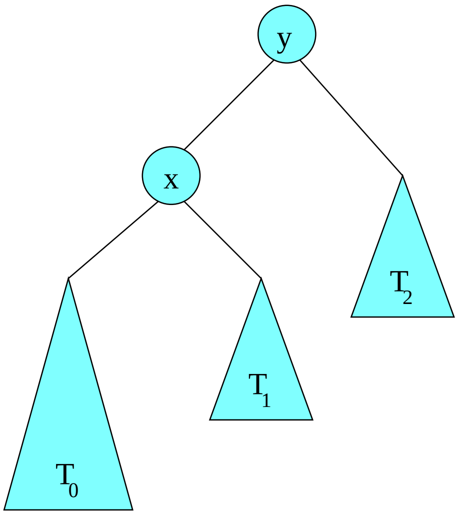
Assume that we have the balance . . . 17.51

Another way to describe balancing



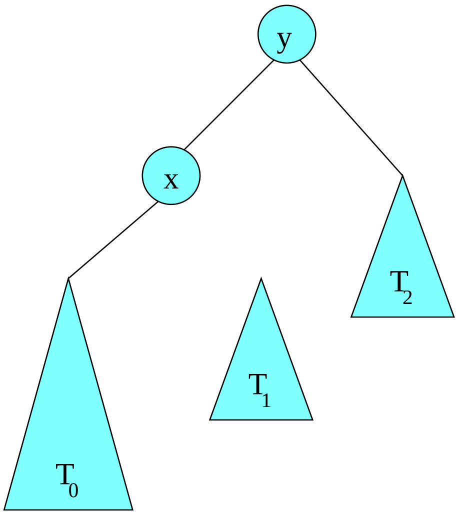
. . . then lost as something mess it up. 17.52

Another way to describe balancing



Do a simple rotation 17.53

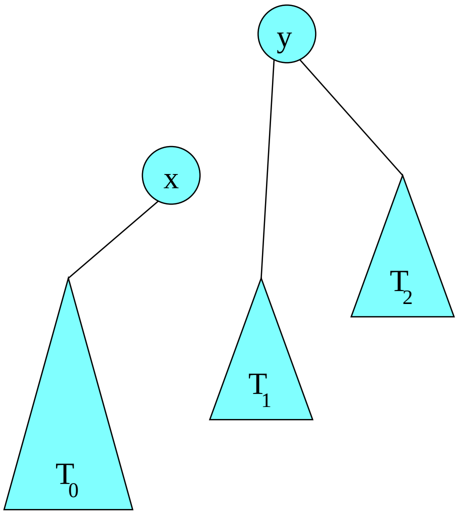
Another way to describe balancing



Do a simple rotation

17.54

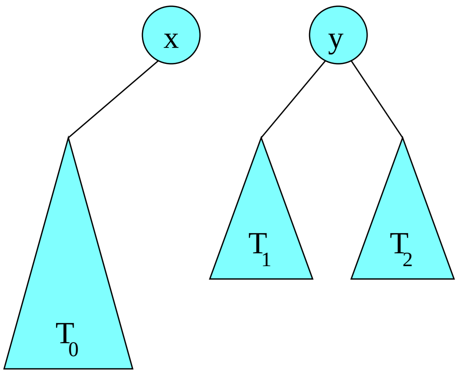
Another way to describe balancing



Do a simple rotation

17.55

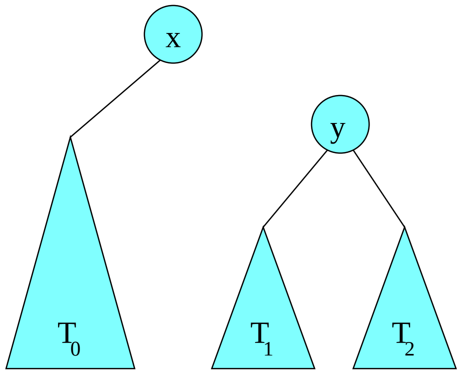
Another way to describe balancing



Do a simple rotation

17.56

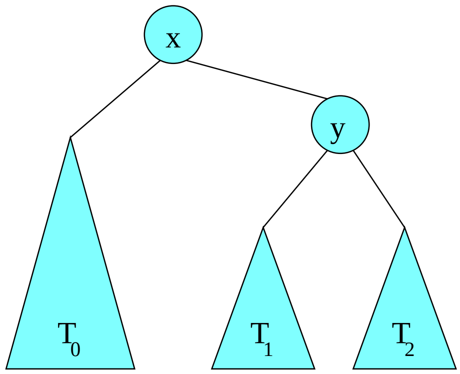
Another way to describe balancing



Do a simple rotation

17.57

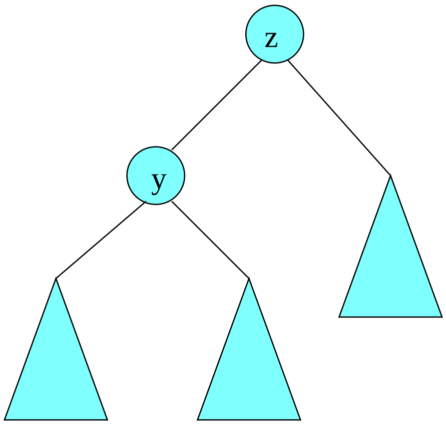
Another way to describe balancing



Done!

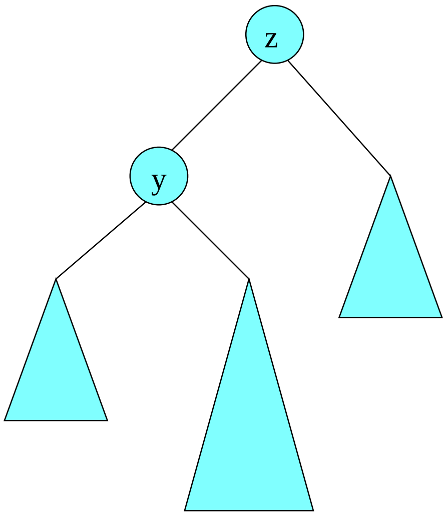
17.58

Another way to describe balancing



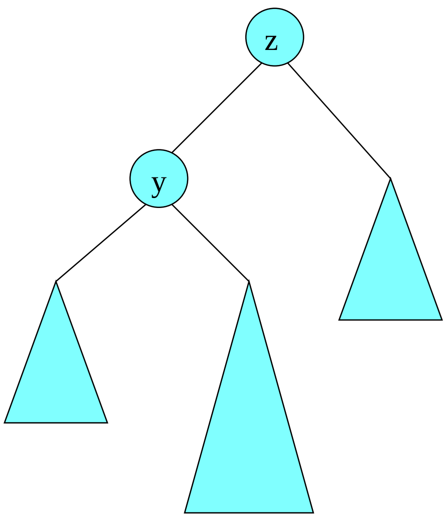
Another example. . . 17.59

Another way to describe balancing



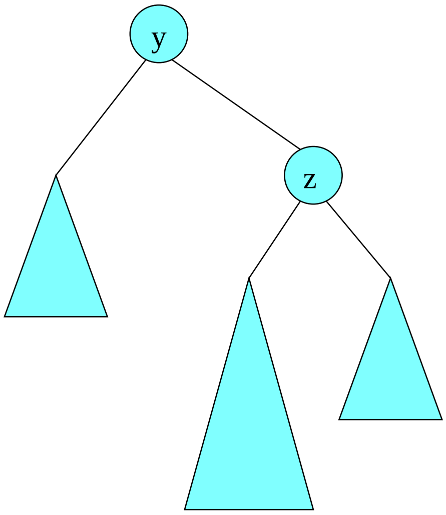
. . . This time, we drop something in another place. 17.60

Another way to describe balancing



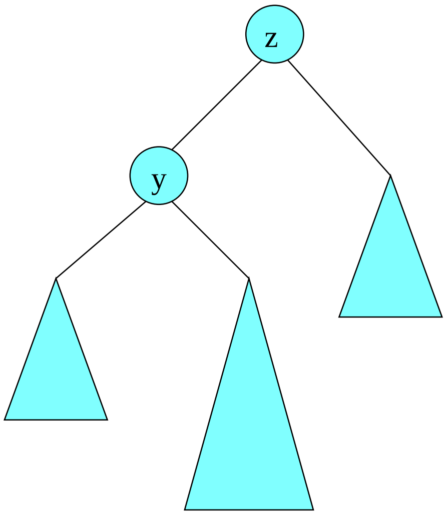
Try a simple rotation again . . . 17.61

Another way to describe balancing



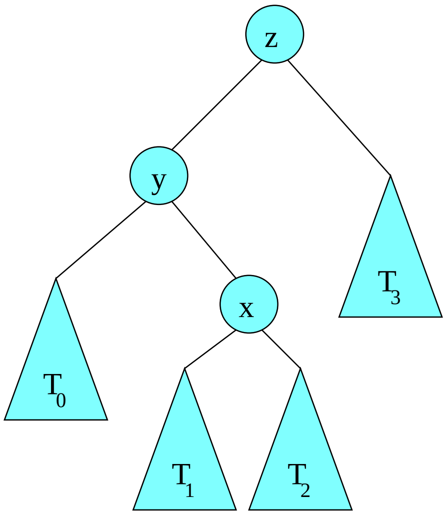
. . . hmm, we have not got the balance 17.62

Another way to describe balancing



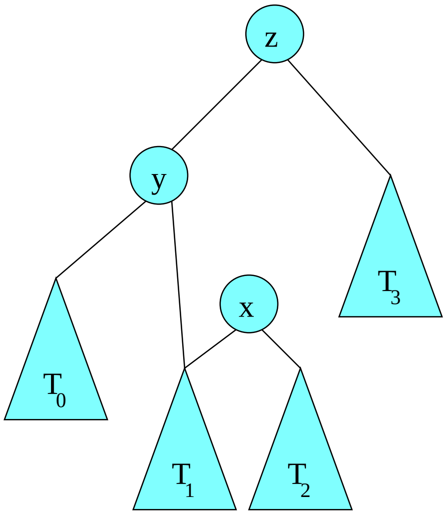
Start from scratch . . . and look at the structure in *y* 17.63

Another way to describe balancing



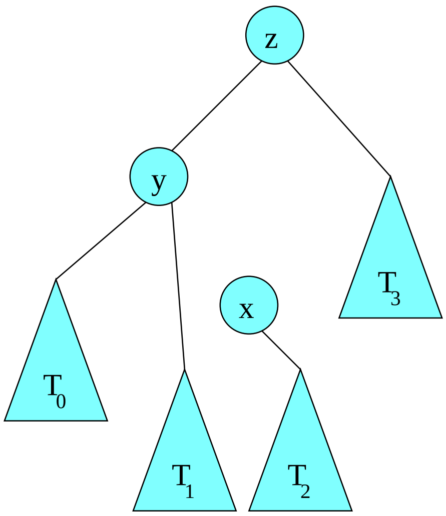
We'll have to make a double rotation 17.64

Another way to describe balancing



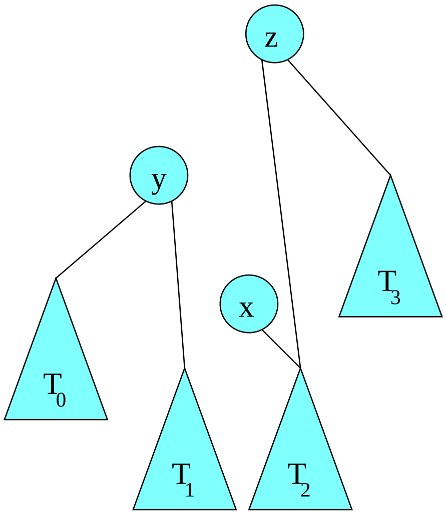
We'll have to make a double rotation 17.65

Another way to describe balancing



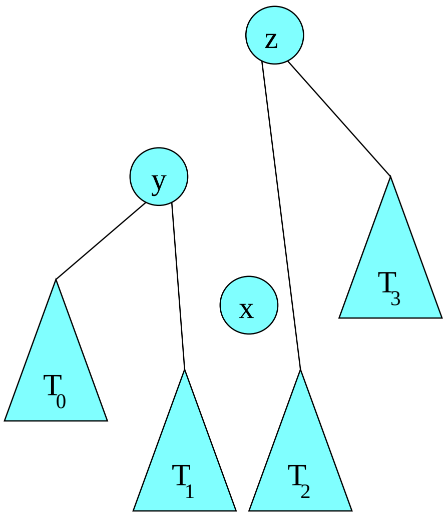
We’ll have to make a double rotation 17.66

Another way to describe balancing



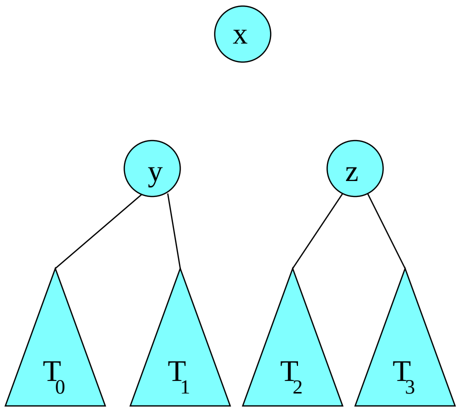
We’ll have to make a double rotation 17.67

Another way to describe balancing



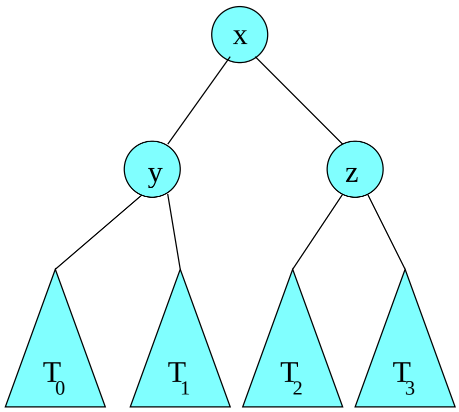
We’ll have to make a double rotation 17.68

Another way to describe balancing



We'll have to make a double rotationn 17.69

Another way to describe balancing



Done! 17.70

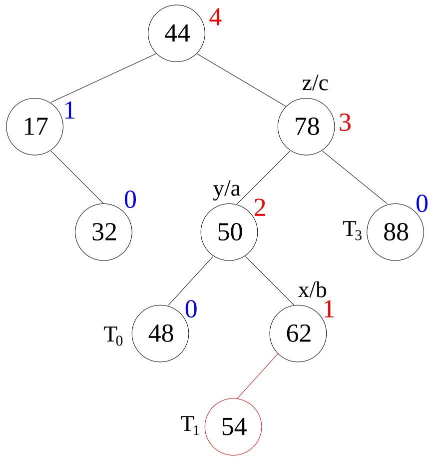
Insertion algorithms

- Starting from the new node and look up until finding a node x such that its "grandparent" node z is not balanced. Mark x's parent with y.
- Make a reconstruction of the tree like this:
  - Rename x, y, z to a, b, c based on their disorder-order.
  - Let T_0, T_1, T_2, T_3 be an enumeration (not ordered) of subtrees to x, y and z. (None of the subtrees has x, y and z as root).
  - z exchanged to b, its children are now a and c.
  - T_0 and T_1 are children to a; T_2 and T_3 are children to c.

17.71

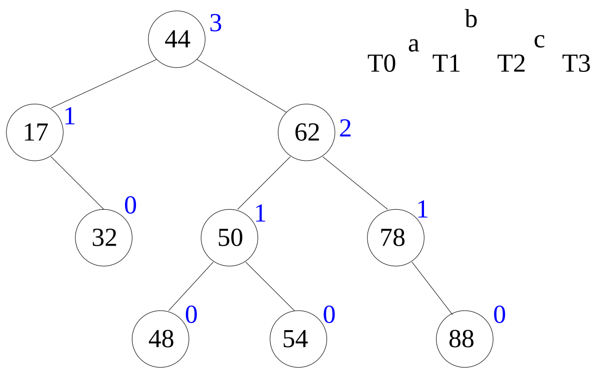
Example: inserting in a AVL-tree

17.72



Example: inserting in a AVL-tree

17.73



Removing nodes from a AVL-tree

- find and remove are the same as in binary search trees
- Update the balance information on the way back to the root
- If not balanced: restructure . . . but. . .
  - When we restore the balance in one place, we can cause an imbalance in another
  - We must repeat balancing (or keep controlling the balance) until we reach the root
  - A maximum of $O(\log n)$ rebalancing

17.74

2.7 $(2,3)$-trees

New approach: Drop some of the requirements

- AVL-tree: *binary* tree, accepts certain (minor) imbalance. . .
- Remember: Full binary tree: non-empty, the degree of each node is either 0 or 2. Perfect binary tree: full, all leaves have the same depth
- Can we build and maintain a perfect tree (if we ignore "binary")? Then we would always know the search time in the worst case exactly!

17.75

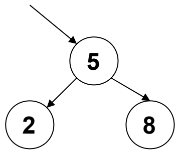
$(2,3)$-tree

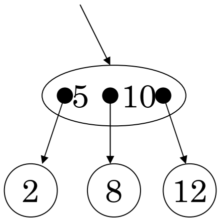
Previously:

- Ett "pivot element"
- If we look more to the right
- If we look less to the left

Now:

- Allow multiple (namely 1–2) ”pivot elements”
- The number of children of an internal node is the number of pivot elements + 1 (i.e. 2–3)





17.76

More general (a,b)-trees

- Each node is either a leaf or has c children, then $a \leq c \leq b$ Each node has between $a-1$ to $b-1$ pivot elements
- $2 \leq a \leq (b+1)/2$ (but the root needs to have at least 2 children (or none) even when $a > 2$ items)

- find works in the same way as previously defined
- insert must check that the node does not become overloaded (in such a case, the node must be *divided*)
- remove can lead to *merge* nodes or *transfer* values between nodes

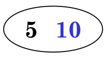
Proposition 3. The height of a (a,b)-tree containing n elements is $\Omega(\log n/\log b)$ and $O(\log n/\log a)$. Höjden av ett (a,b)-träd som lagrar n dataelement är $\Omega(\log n/\log b)$ och $O(\log n/\log a)$.

$\Rightarrow$ Flattening trees, but needs more processing at node level.

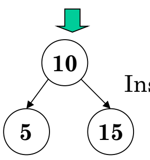
17.77

Insertion in a (a,b)-tree with $a=2$ and $b=3$

Insert(10)



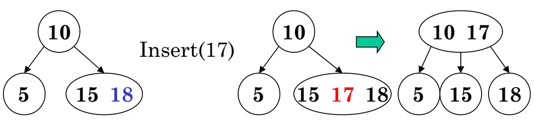
Insert(15)



Insert(18)

- As long as there is space in the child we find, add an element to the child ...

- If full, split up the new node and move the selected pivot element upward... ...This can happen repeatedly

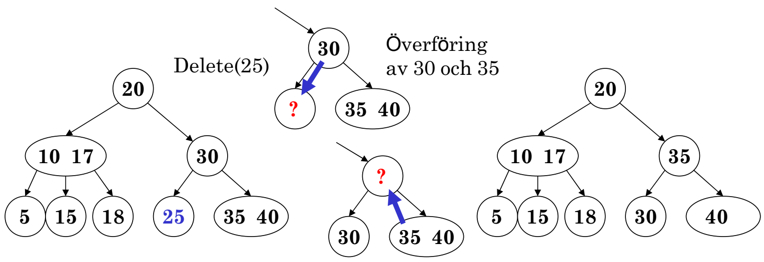


17.78

Removing an element from a $(2,3)$-tree

3 cases:

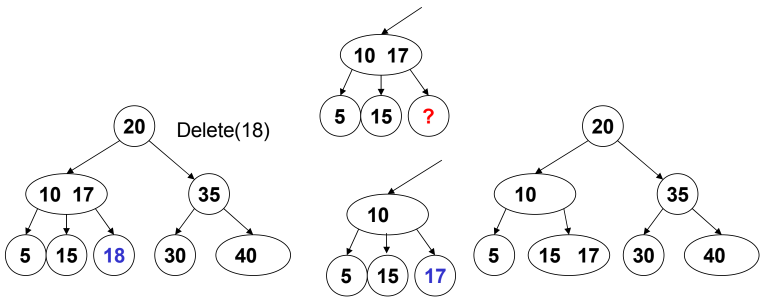
- No conditions are broken by the removal
- A leaf is removed (becomes empty) Associate another key to the leaf by re-arrangement, …ok if we have siblings with 2+ elements



17.79

Removing an element from a $(2,3)$-tree

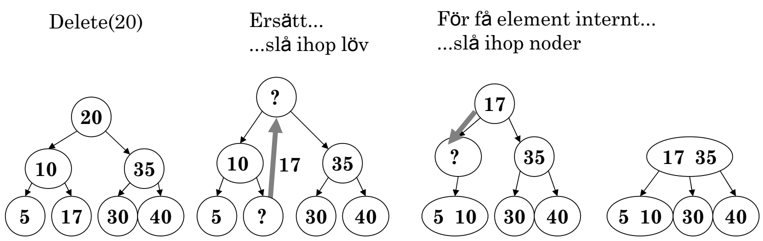
- If A leaf is removed (becomes empty)
- Another key will be associated to the leaf (from the parent level), or
- Merge it with a neighbor



17.80

Removing an element from a $(2,3)$-tree

- An internal node becomes empty En intern nod blir tom The root: replace with predecessors or successors in the order Then repair inconsistencies with appropriate merges and transfers…



17.81

2.8 B-trees

B-trees

- Used to maintain an index of external data (such as contents on a disk memory)
- is a (a,b)-tree where $a = \lceil b/2 \rceil$, i.e. $b = 2a - 1$
- We can choose b so that a full node just takes up a block on the disk
- By choosing $a = \lceil b/2 \rceil$, we always fill an entire block on the disk when two blocks are merged together!

- B-trees (and variants) used in many file systems and databases
  - Windows: HPFS
  - Mac: HFS, HFS+
  - Linux: ReiserFS, XFS, Ext3FS, JFS
  - Databases: ORACLE, DB2, INGRES, PostgreSQL

17.82