Föreläsning 15

Trees

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
08-11 November 2024

IDA, Linköpings universitet

15.1

Content

Contents

15.2

1 Symbol tables

Symbol tables

- Abstraction of key-value pairs
  - *insert* a value with a specified key
  - Given a key, *search* for a corresponding value

15.3

1.1 Abstract datatypes

1.2 Implementation

Implementation: Set, multiset, Map, Dictionary

- Table/array: sequence of adjacent memory locations
  - Unordered: no order required between $T[i]$ and $T[i+1]$
  - Ordered: ...order required between the keys $T[i] < T[i+1]$
- Linked lists
  - unordered
  - ordered
- (Binary) search trees
- Hashing
- Skip-lists

15.4

Table representation of a Dictionary

unordered table:
find with *linear search*

- unsuccessful look-up: n comparisons $\Rightarrow O(n)$ time complexity
- successful look-up, worst case: n comparisons $\Rightarrow O(n)$ time complexity
- successful look-up, average case with uniform partition of the query positions: $\frac{1}{n}(1+2+\ldots+n) = \frac{n+1}{2}$ comparisons $\Rightarrow O(n)$ time complexity

15.5

Table representation of a Dictionary

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

- look-up: $O(\log n)$ time complexity
- ... updates are however expensive!!

15.6

2 Trees

2.1 Basic concepts

Why trees?

Tree-like structures appear naturally in many situations

- File systems
- Decision trees
- Hierarchical organizations of
  - Document: book, chapter, section
  - XML-document
- To capture an ordering or a priority

15.7

Terminology

- A *(rooted) tree* $T = (V, E)$ consists in a set V of *nodes* and *edges* E, where each edge is a pair $(u, v) \in V \times V$.
- Nodes $v \in V$ store data in a *parent-child* relationship.
- A parent-child relationship between the parent node u and the child node v is expressed with a directed edge $(u, v) \in E$, from u to v.
- Each node has at most a parent; it can have many *siblings*.
- There are at most one node without a parent – the *root node*.

15.8

More terminology

- The *degree* of a node is the number of its children
- A node without children is a *leaf* or an *external* node. All other nodes are *internal* nodes.
- A path is a sequence of nodes $(v_1, v_2, \ldots, v_k)$, where $k > 0$ and (v_i, v_{i+1}) is an edge for each for $i = 1, \ldots, k-1$.
- The length of a path $(v_1, v_2, \ldots, v_k)$ is $k-1$. Observe the length of the path (v_1) with a single node is 0.
- A node n is an *ancestor* to a node v iff there is 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.

15.9

More terminology

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

15.10

Some tree types

- *Ordered tree*: linear ordering (as in left, right, or first, second etc) between the children of each node. Do not confuse with Sorted trees.
- *Binary tree*: ordered tree where each node has a degree ≤ 2. A node can have a left child and a right child.
- *Empty binary tree* (**null**): a binary tree without nodes.
- *Full binary tree*: non-empty binary tree where each node has a degree of 0 or 2. Consequence (by induction on number of nodes): #leaves = 1 + #internal nodes.
- *Perfect binary tree*: full binary tree where all leaves have the same depth. Consequence (induction on height) : #nodes = $2^{h+1}-1$ where h is the height of the tree.
- *Complete binary tree*: An approximation of perfect trees where rows are filled row after row from left to right. Consequence: a complete binary tree with height h and n nodes satisfies $2^h \leq n \leq 2^{h+1}-1$.

15.11

2.2 ADT tree

Operations on a node v of a tree T

- *parent*(v) returns the parent of v, **error** if v is a root node
- *children*(v) returns set of children of v
- *firstChild*(v) returns first child of v or **null** if v is a leaf
- *rightSibling*(v) returns right sibling to v or **null** if no right sibling
- *leftSibling*(v) returns left sibling of v or **null** if no left sibling
- *isLeaf*(v) returns **true** iff v is a leaf
- *isInternal*(v) returns **true** iff v is not a leaf node
- *isRoot*(v) returns **true** iff v is a root node
- *depth*(v) returns depth of v in T
- *height*(v) returns height of v in T

15.12

Operations on a tree T

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

In addition, for a *binary tree*

- *left*(v) returns left child of v or **error**
- *right*(v) returns right child of v or **error**
- *hasLeft*(v) checks if v is a left child
- *hasRight*(v) checks if v is a right child

15.13

2.3 Representation of binary trees

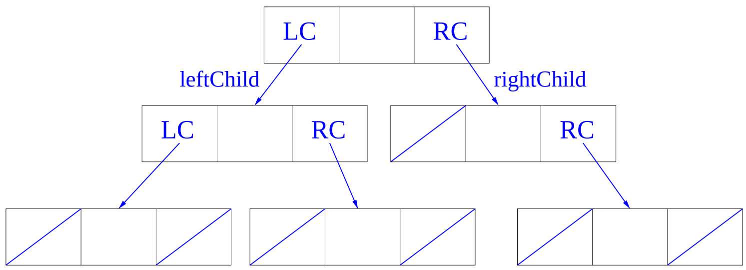
A linked representation

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

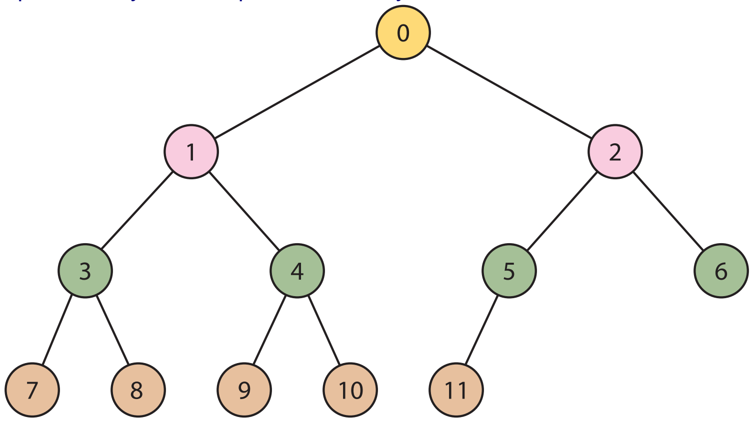
Or, for a binary tree

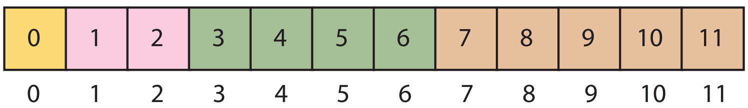
class treeNode<T> *nodeInfo*: T *leftChild*: treeNode<T> *rightChild*: treeNode<T>

15.14



Complete binary tree: sequential memory



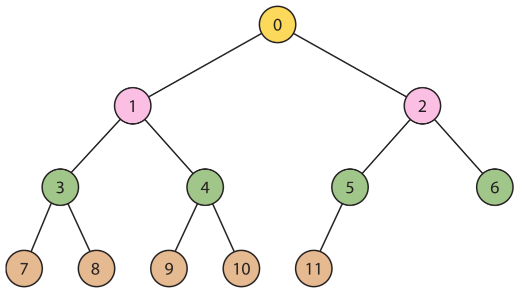


15.15

Sequential memory

Use a table 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$(i) = (i = 0)$

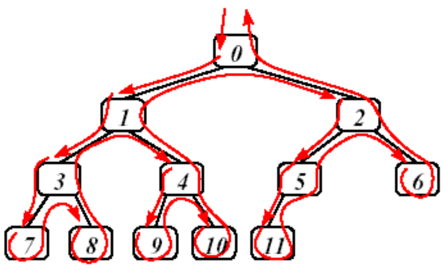


15.16

2.4 Tree traversals

Traversal of a tree Generic routine for traversing a tree

procedure VISIT(node v)
  for all $u \in$CHILDREN(v) **do**
    VISIT(u)



Call visit$(root(T))$ and each node in T will be visited exactly once! 15.17

Traversing a tree

```
procedure PREORDERVISIT(node v)
    DOSOMETHING(v)                                ▷ before children
    for all u ∈CHILDREN(v) do
        PREORDERVISIT(u)

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

15.18

Traversing a tree (here, for binary trees)

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

15.19

Traversing a tree

```
procedure LEVELORDERVISIT(node v)
    Q ←MAKEEMPTYQUEUE()
    ENQUEUE(v,Q)
    while not ISEMPTY(Q) do
        v ←DEQUEUE(Q)
        DOSOMETHING(v)
        for all u ∈CHILDREN(v) do
            ENQUEUE(u,Q)
```

A breadth first traversal. 15.20

2.5 Binary search trees

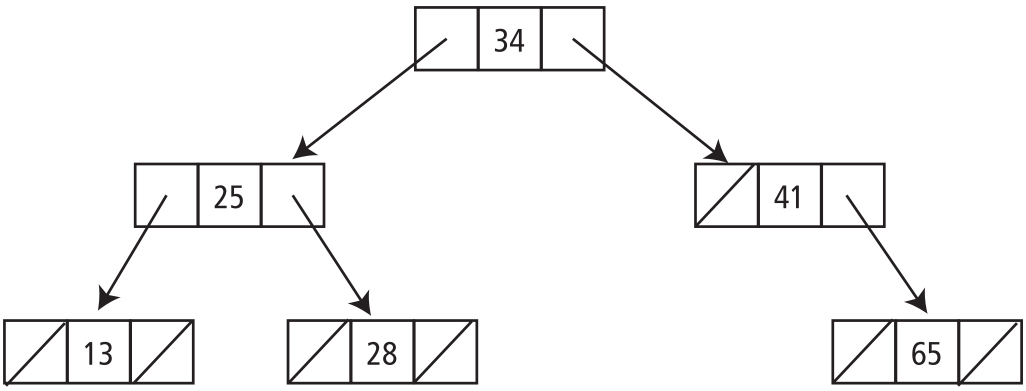
Binary search trees

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

- information associated with a node is (key,value). The keys are ordered as foolows.

The key in each node is:

- larger than or equal to each key appearing in all left descendants, and
- less than the key appearing in all right decendants.



15.21

ADT Map with a binary search tree

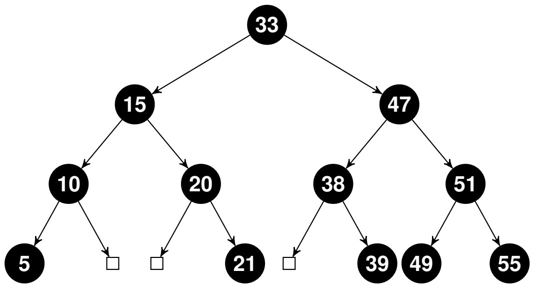
```
procedure FIND(k, v)
    if v = null then return null
    else if KEY(v) = k then return v
    else if k <KEY(v) then
        FIND(k,LEFTCHILD(v))                      ▷ unsuccessful if no leftChild
    else
        FIND(k,RIGHTCHILD(v))                     ▷ unsuccessful if no rightChild
```

Worst case: $\text{HEIGHT}(T)+1$ comparisons. 15.22

ADT Map with a binary search tree

insert(k, v): insert (k, v) as a new leaf if unsuccessful find, otherwise update the node

procedure FIND(k, v)
  if $v =$ `null` **then return** `null`
  else if KEY$(v) = k$ **then return** v
  else if $k <$KEY(v) **then**
    FIND$(k$,LEFTCHILD$(v))$
  else
    FIND$(k$,RIGHTCHILD$(v))$

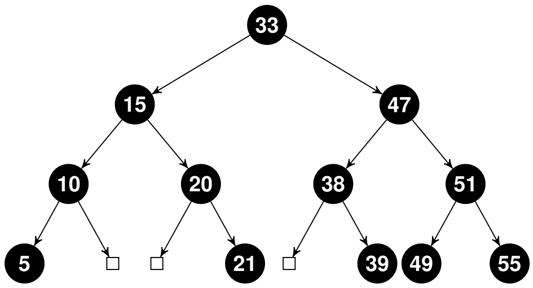


Worst case: HEIGHT$(T) + 1$ comparisons 15.23

ADT Map with a binary search tree

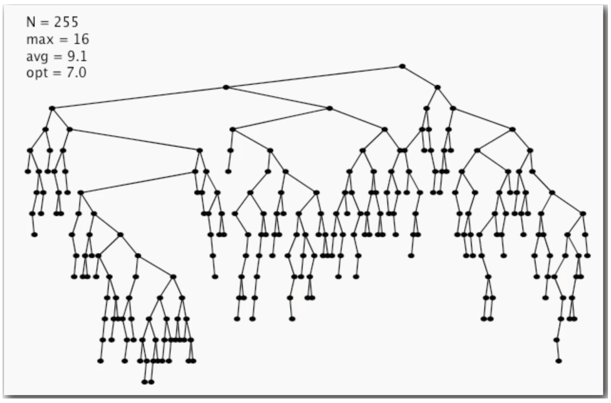
remove(k): find, then...

- if v is a leaf (e.g., 5, 49), remove v
- if v has a child u, replace v with u (e.g., 10, 20)
- if v has two children (e.g., 15, 33), replace v with its successor in inorder and remove the successor
- (alternatively with its predecessor in inorder and remove the predecessor)

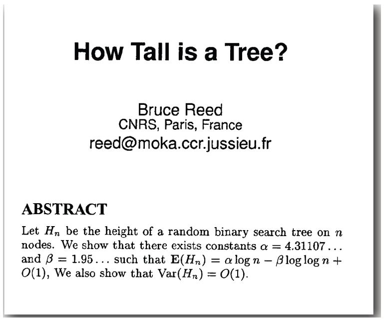


Worst case: HEIGHT$(T) + 1$ comparisons. 15.24

ADT Map with binary search tree



Heights of randomly chosen binary trees



How Tall is a Tree?

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

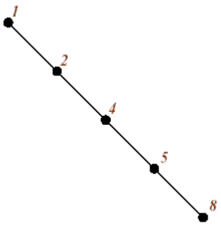
ABSTRACT

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 $\mathbf{E}(H_n) = \alpha \log n - \beta \log\log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

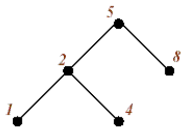
Worst case: $\text{HEIGHT}(T)+1$ comparisons. 15.25

Binary search trees are not unique

Same data can result in different binary search trees



insert: 1,2,4,5,8



insert: 5,2,1,4,8 15.26

Successful look-up

BST in worst case

- BST degenerates to a linear sequence
- expected number of comparisons is $(n+1)/2$

Balanced BST

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

15.27

Therefore — Strive to maintain them balanced!

Some common balanced trees:

- AVL-trees
- (2,3)-trees, (a,b)-trees,
- Red-black trees,
- B-trees,
- Splay-trees

15.28

2.6 AVL-trees

AVL-tree

- Self balancing BST
- AVL = Adelson-Velskii and Landis, 1962
- Idea: Maintain balance information at each node
- AVL-property
  - The difference in height between the children of each node is at most 1
  - alternatively, let $b(v) = \text{height}(\text{leftChild}(v)) - \text{height}(\text{rightChild}(v))$ for node v in T. An AVL-tree T satisfies $b(v) \in \{-1,0,1\}$ for each v in T.

15.29

Maximal height of an AVL-tree

Proposition 1. Height of an AVL-tree with n nodes is $O(\log n)$.

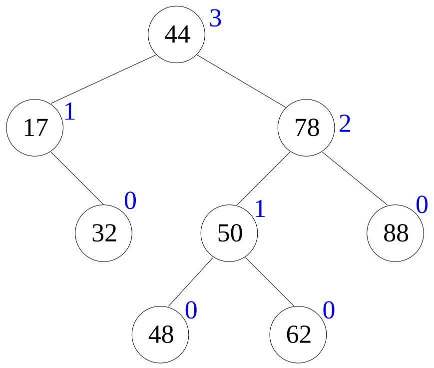
As a result,

Proposition 2. find, insert and remove can be written, for AVL-trees, to have time complexity in $O(\log n)$ while preserving the AVL-property.

15.30

Exampel: an AVL-tree

15.31

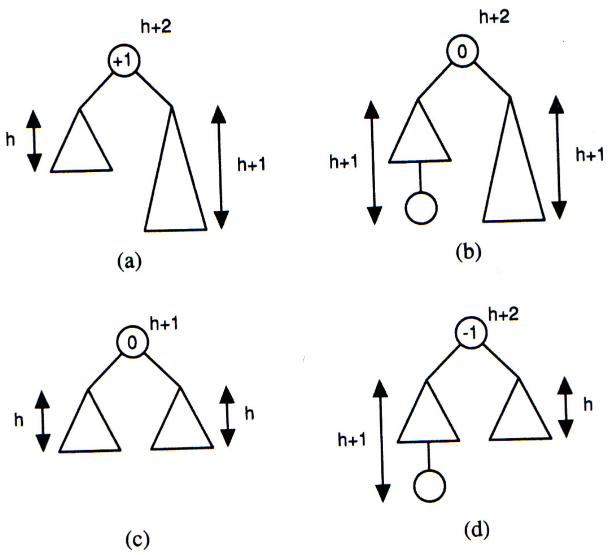


Insert in an AVL-tree

- The new node might change the heights in a way that the tree needs to be balanced.
  - You can track heights of the subtrees by
    - storing the hights explicitly in each node
    - storing the difference in each node
- Balancing is usually described with right or left rotations of subtrees.
- It is enough to use rotations to balance the tree.

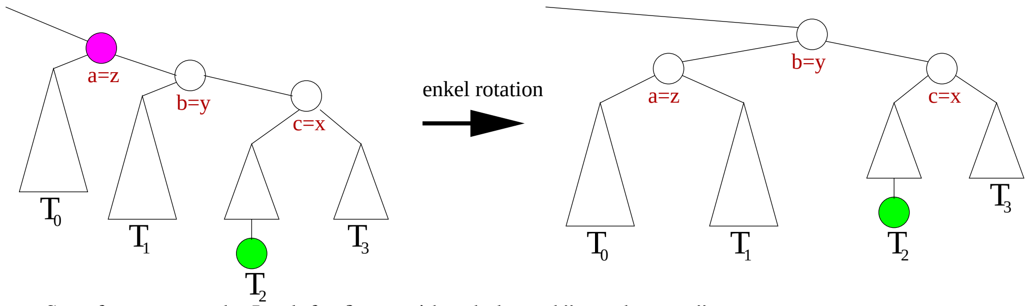
15.32

Insert in an AVL-tree (simple case)



15.33

Four different rotations

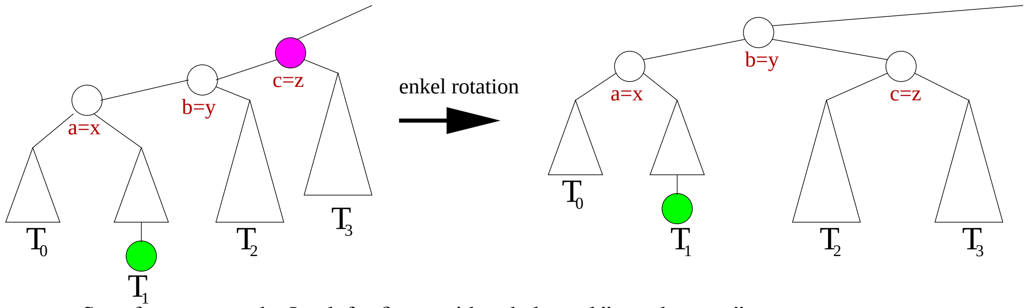


- Start from new node. Look for first x with unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

Simple rotation if $b = y$: "Rotate y up over z"

15.34

Fyra olika rotationer

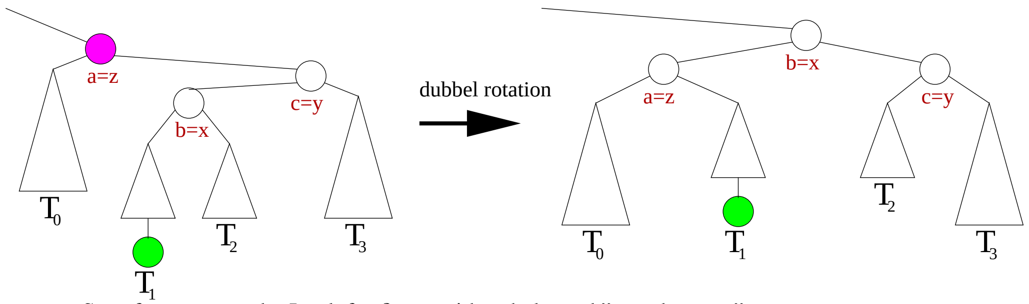


- Start from new node. Look for first x with unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

Simple rotation if $b = y$: "Rotate y up over z"

15.35

Fyra olika rotationer

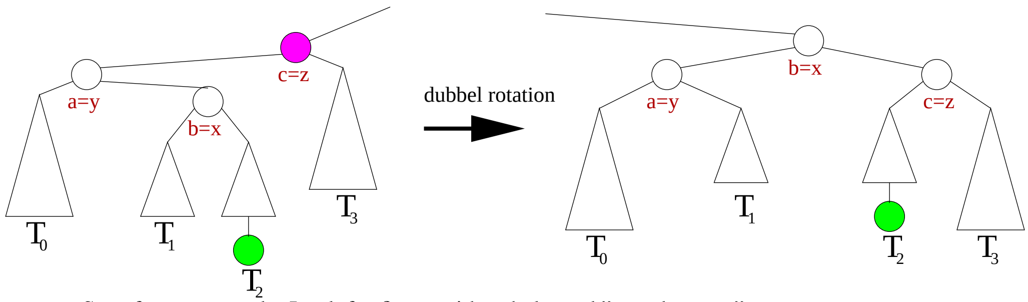


- Start from new node. Look for first x with unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

Double rotation if $b = x$: "Rotate x up over y", "then over z"

15.36

Fyra olika rotationer



- Start from new node. Look for first x with unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

Double rotation if $b = x$: "Rotate x up over y", "then over z"

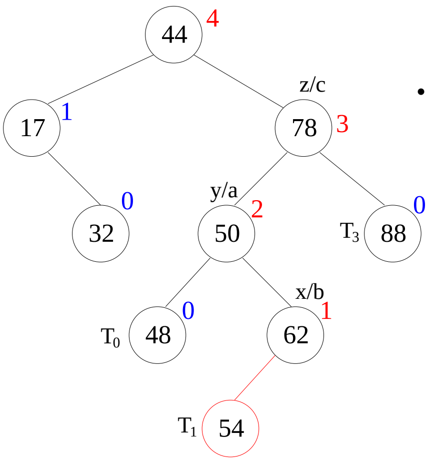
15.37

Insertion algorithm

- Start from the new node. Look for the first x with an unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on the occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of the subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

15.38

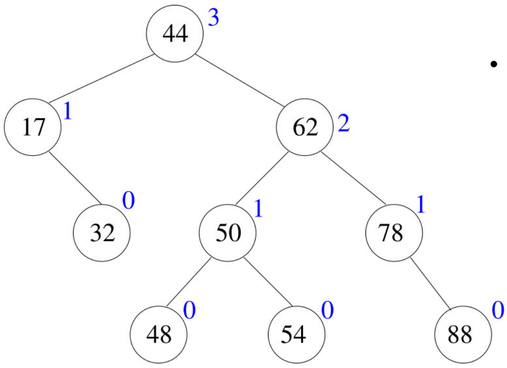
Exempel: insertion in an AVL-tree



- Start from the new node. Look for the first x with an unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on the occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of the subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

15.39

Exempel: insertion in an AVL-tree



- Start from the new node. Look for the first x with an unbalanced "grand-parent" z. Denote with y the parent of x.
  - Rename x, y, z to a, b, c based on the occurence in an inorder traversal
  - Let T_0, T_1, T_2, T_3 be an enumeration, in an inorder traversal, of the subtrees of x, y och z. (none of x, y or z is root to these subtrees.)
  - Replace z by b. The children of b are now a and c.
  - T_0 and T_1 are children to a. T_2 and T_3 are children to c.

15.40

Deletion in an AVL-tree

- find and remove are similar to a simple binary search tree
- Update the balance information on the way up to the root
- If unbalanced, restructure using rotations:
  - when restoring balance in a part, we can create unbalance in another place
  - Repeat balancing untill the root
  - At most $O(\log n)$ rebalancings

15.41

2.7 $(2,3)$-tree

Another approach: drop some requirements

- AVL-tree: *binary* trees, accept some controlled unbalance…
- Recall
  - Full binary trees: non-empty trees with node degrees of 0 or 2
  - Perfect binary trees: full where all leaves have the same depth
- Maintain a perfect tree and drop the binary requirement? obtained tree would be perfectly balanced.

15.42

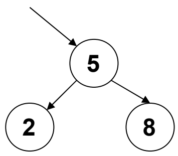
$(2,3)$-tree

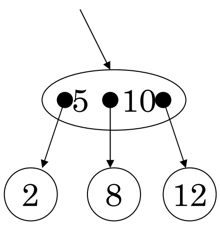
in a binary search tree:

- a "pivot" element
- If larger, look to the right
- If smaller, look to the left

In a $(2,3)$-tree:

- Allow several (here 1–2) pivot elements
- Number of children of an internal node is 1 plus the number of pivot elements (here 2–3)





15.43

More generally (a,b)-tree

- a,b satisfy $2 \leq a \leq (b+1)/2$
- Each internal node, except for the root, has a to b children
- The root is either a leaf or it has 2 to b children

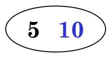
- find as in a BST with the additional pivots
- insert has to handle overfull nodes, in which case nodes have to be divided
- remove has to handle underfull nodes, in which case values need to be transferred between the nodes, or nodes need to be merged

Proposition 3. Height+1 of an (a,b)-tree with n nodes is between $\log_b(n+1)$ and $\log_a(n+1)$.

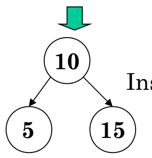
$\Rightarrow$ more flat trees, but more work in the nodes 15.44

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

Insert(10)



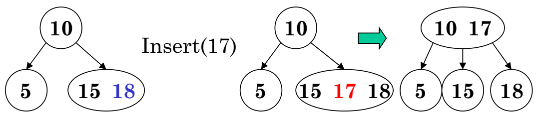
Insert(15)



Insert(18)

- If there is place in a child, add the element...

- If full, divide the node and promote the pivot element up. This may need to be repeated.

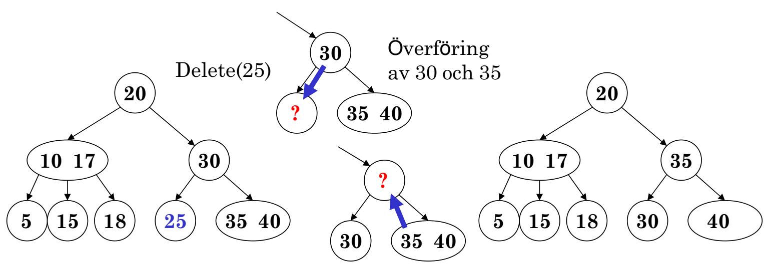


15.45

Deletion in a $(2,3)$-tree

We consider three cases:

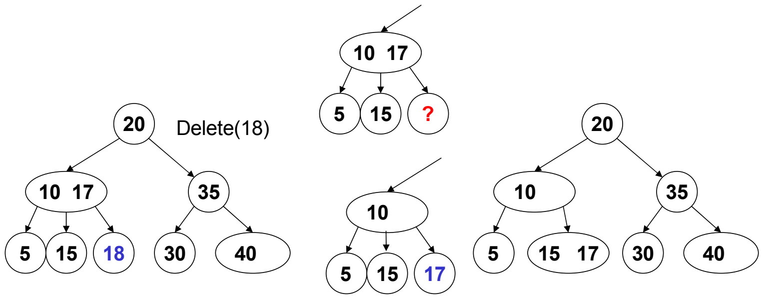
- A key is deleted without violating the requirements
- The last key in a leaf node is deleted and becomes empty
  - transfer some key from another node: ok if a sibling has 2+ elements
  - otherwise, merge
- A key in an internal node is deleted



15.46

Deletion in a $(2,3)$-tree

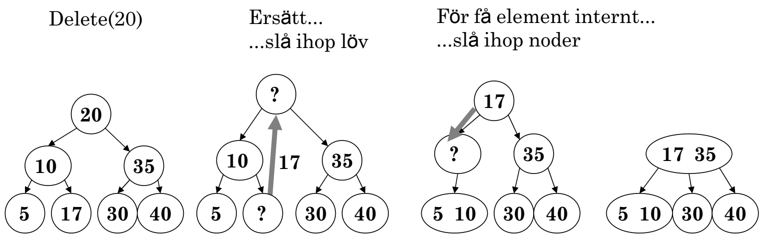
- A key is deleted without violating the requirements
- The last key in a leaf node is deleted and becomes empty
  - transfer some key from another node: ok if a sibling has 2+ elements
  - otherwise, merge
- A key in an internal node is deleted



15.47

Deletion in a $(2,3)$-tree

- A key in an internal node is deleted
  - replace predecessor or successor in order and repair inconsistencies with replacements and merging



15.48

2.8 B-tree

B-tree

- Used for indexing external data: (e.g. content on a hard drive)
- A B-tree is an (a,b)-tree where $a = \lceil b/2 \rceil$
- We can choose b so that it exactly occupies a hard drive memory block
- With $a = \lceil b/2 \rceil$ we ensure internal nodes are half full and merging results in a block

- B-tree (and variants of such as B+-trees) are used in many filesystems and databases
  - Windows: HPFS
  - Mac: HFS, HFS+
  - Linux: ReiserFS, XFS, Ext3FS, JFS
  - Databaser: ORACLE, DB2, INGRES, PostgreSQL

15.49