Algorithms and Data Structures II. Lecture Notes: Trees

Tibor Ásványi
Department of Computer Science
Eötvös Loránd University, Budapest
asvanyi@inf.elte.hu

August 27, 2021

Contents

References

[1] Ásványi, T, Algorithms and Data Structures I. Lecture Notes
http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs1/AlgDs1LectureNotes.pdf

[2] Burch, Carl, B+ trees
(See http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs2/B+trees.pdf)

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
Introduction to Algorithms (Third Edititon), *The MIT Press*, 2009.

[4] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[5] Narashima Karumanchi,
Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[6] Neapolitan, Richard E., Foundations of algorithms (Fifth edition),
Jones & Bartlett Learning, 2015. ISBN 978-1-284-04919-0 (pbk.)

[7] Shaffer, Clifford A.,
A Practical Introduction to Data Structures and Algorithm Analysis,
Edition 3.1 (C++ Version), 2011
(See http://aszt.inf.elte.hu/~asvanyi/ds/C++3e20110103.pdf)

[8] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++
(Fourth Edition),
Pearson, 2014.

[9] Wirth, N., Algorithms and Data Structures,
Prentice-Hall Inc., 1976, 1985, 2004.
(See http://aszt.inf.elte.hu/~asvanyi/ds/AD.pdf)

1 Introduction

We need *dictionaries*, i.e. large sets of data with efficient insertion, search and deletion operations (where removing the maximal or minimal element of the dictionary and removing an element with a given key are the cases of deletion). We already have two solutions where the average performance (AT) of these operation is good but the worst case performance (MT) is too slow (n is the size of the set / data structure):

- Binary search trees (BSTs): $AT(n) \in \Theta(\log n), MT(n) \in \Theta(n)$.
- Hash tables: $AT(n) \in \Theta(1)$ (if the load factor is not too high), $MT(n) \in \Theta(n)$. (Removing the maximal and minimal elements is not supported.)

We are going to discuss some kinds of *balanced search trees* where we can guarantee $MT(n) \in \Theta(\log n)$ for each dictionary operation.

- *AVL trees*: balanced BSTs for storing data in the central memory.[1]
- *B+ trees*: perfectly balanced multiway search trees optimized for storing data on hard disks.[2]

In the next section we consider *AVL trees*. In the last one we describe *B+ trees*.

2 AVL Trees

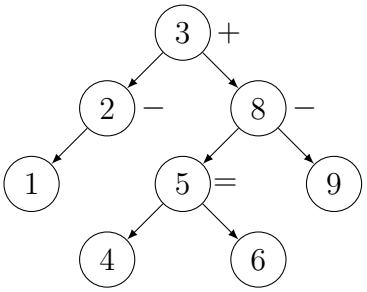
Being BSTs, AVL trees have linked representation typically. In these notes, we suppose that they have linked representation. Everywhere letter n denotes the size (number of nodes) of the tree, and h denotes the height of the tree.

Definition 2.1 *The* balance *of node* $(*p)$ *of a binary tree is* $p \to b = h(p \to right) - h(p \to left)$. *This means that the* balance *of a node is the height of its right subtree* minus *the height of its left subtree.*[3]

[1] *Red-black trees* are also specially balanced BSTs and can provide another efficient solution.

[2] *B trees* are an older solution.

[3] The *balance* notion above is sometimes called *height-balance*, in order to distinguish it from other kinds of *balances*. For example, the *size-balance* of a node is the size of its right subtree *minus* the size of its left subtree. A node is *size-balanced*, **iff** its *size-balance* is in $\{-1, 0, 1\}$. A tree is *size-balanced*, **iff** all of its nodes are *size-balanced*. In these notes, *balance* means *height-balance* by default.



The binary tree on the left in

- simple textual form:
  (((1) 2−) 3+ (((4) 5= (6)) 8− (9)))
- elegant parenthesized form:
  { [(1) 2−] 3+ [(⟨4⟩ 5= ⟨6⟩) 8− (9)] }

Figure 1: The same binary tree in graphical and textual representations. The balance of each leaf is zero, so we omit the balances of the leaves. We show the balance of each internal node next to it (or next to the key identifying it). In the example trees, we use the following notation for the balances: $0 \sim =$; $1 \sim +$; $2 \sim ++$; $-1 \sim -$; $-2 \sim --$

Example 2.2 *See the BST on* Figure 1. *Deleting node* ① *from it, we receive BST* { [2] 3++ [(⟨4⟩ 5= ⟨6⟩) 8− (9)] }. *And inserting node* ⑦ *into the BST on* Figure 1, *we receive the following BST.*
{ [(1) 2−] 3++ [(⟨4⟩ 5+ ⟨ 6+ {7} ⟩) 8−− (9)] }.

Exercise 2.3 ***Draw each tree which is given in textual representation in these lecture notes. Drawing them provides easier and deeper understanding, and better long-term knowledge. One can see a pattern above, at*** **Figure 1.**

Definition 2.4 *Node* $(*p)$ *is* perfectly balanced, **iff** *its balance,* $p \to b = 0$.[4]

Definition 2.5 *Node* $(*p)$ *is* balanced, **iff** *its balance, i.e.* $p \to b \in \{-1, 0, 1\}$. *Otherwise it is* imbalanced.

Consequence 2.6 *The leaves of a binary tree are* perfectly balanced, *because both of the right and left subtrees of a leaf are empty.*

Definition 2.7 *A binary tree is* balanced, **iff** *each (internal) node of it is balanced. It is* imbalanced, **iff** *at least one of its nodes is imbalanced.*

Definition 2.8 *A binary tree is* perfectly balanced, **iff** *each (internal) node of it is perfectly balanced.*

Consequence 2.9 *A binary tree is* perfectly balanced, **iff** *it is* complete.

[4] **iff** means **if and only if**.

Definition 2.10 *An* AVL tree *is a* balanced BST.[5]

Remember that $MT(h) \in \Theta(h)$ for the basic operations of BSTs (insertion, search and deletion). Fortunately, the BST and the balanced tree properties can be maintained efficiently throughout insertions and deletions on balanced BSTs, and the next theorem guarantees that the height of an AVL tree never goes far form the ideal case. Thus we can guarantee the high efficiency of these operations on AVL trees.

Theorem 2.11 *Given a nonempty balanced binary tree,*

$$\lfloor \log n \rfloor \leq h \leq 1.45 \log n, \quad \text{i.e.} \quad h \in \Theta(\log n)$$

We provide an outline of the proof of this theorem. The lower bound given there is the lower bound of the height of all the binary trees, thus it is also true for balanced binary trees. The upper bound given here follows from the properties of *Fibonacci trees*.

Definition 2.12 *A binary tree is* Fibonacci tree, **iff** *each* internal *node of it is balanced, but not perfectly balanced.*

These are called *Fibonacci trees*, because a Fibonacci tree with height $h > 0$ consists of a root *plus* a left subtree with height $h-1$ *and* a right subtree with height $h-2$ or vice versa where both subtrees are Fibonacci trees. And this recursive description is similar to the recursive definition of *Fibonacci numbers*.

It is easy to prove that among the balanced binary trees with a given height, the Fibonacci trees have the smallest size. Let f_h be the size of a nonempty Fibonacci tree with height h. Clearly
$f_0 = 1$, $f_1 = 2$, $f_h = 1 + f_{h-1} + f_{h-2} \quad (h \geq 2)$.

[5]It is easy to prove that the *size-balanced binary trees* are special cases of the *nearly complete binary trees*. And we know that the height h of a nearly complete binary tree is minimal among the binary trees of a given size n, i.e. $h = \lfloor \log n \rfloor$, plus $MT(h) \in \Theta(h)$ for the basic operations of BSTs (insertion, search and deletion), consequently these operations are most efficient on nearly complete and especially size-balanced BSTs. Thus one may ask, why we do not use *size-balanced* or *nearly complete* BSTs instead of AVL trees. Well, the *size-balanced* and the *nearly complete* properties turn out too strong requirements in most cases, because we cannot maintain these properties efficiently throughout insertions and deletions on BSTs. Therefore these properties are often lost throughout insertions and deletions, and the height of a resulting BST may go far from the ideal case. Thus we use AVL trees, because the AVL tree property can be maintained extremely efficiently throughout insertions and deletions, and the height of an AVL tree is quite close to the minimal height $h = \lfloor \log n \rfloor$ even in the worst case.

These formulas are similar to the definition of the Fibonacci sequence: $F_0 = 0$, $F_1 = 1$, $F_h = F_{h-1} + F_{h-2} \quad (h \geq 2)$.

It can be proved with mathematical induction that $f_h = F_{h+3} - 1$ where h is a natural number. Clearly $f_h \leq n$ where n is the size of a balanced binary tree with height h. With some mathematical skill, we can get the upper bound given in the previous theorem. (We omit the details.)

From the previous results, it can be proved that $MT(n) \in \Theta(\log n)$ for the basic operations of AVL trees (insertion, search and deletion). In order to see it, still we have to see that $MT(h) \in \Theta(h)$ even for the AVL tree version of these operations where each operation keeps the AVL tree property.

In order to achieve this aim, first we describe our representation of the nodes of AVL trees. Then we develop the algorithms of the basic operations of the AVL trees. It will be clear that these operations go down and possibly up only once in the tree, and the number of elementary operations can be limited with the same constant at each level, so $MT(h) \in \Theta(h)$ is true. See the representation first.

Node
+ $key : \mathcal{T}$ // $\mathcal{T}$ is some known type + $b : -1..1$ // the balance of the node + $left, right$: Node*
+ Node() { $left := right := \oslash$; $b := 0$ } // create a tree of a single node + Node(x:$\mathcal{T}$) { $left := right := \oslash$; $b := 0$; $key := x$ }

We can see that the balance of a node is stored explicitly in the node. When a tree is modified, clearly the balances of some nodes change. Thus we have to adjust the data members b of these nodes. This process will be called *rebalancing* in these notes. When we illustrate an operation on a BST, we will show the balance values stored in the node objects. (Although a stored balance value and the corresponding proper one must be equal before the operation, and also after it, they may be different during the transformation.)

Let us consider the efficiency and some details of the basic operations of AVL trees. Remember that the height of an AVL tree is $\Theta(\log n)$. Thus the functions search(t, k), min(t) and max(t) of BSTs [1] (which do not modify the tree and just go down in the tree once) can be applied to AVL trees with $MT(n) \in \Theta(\log n)$ efficiency.

Procedures insert(t, k), del(t, k), remMin$(t, minp)$ and remMax$(t, maxp)$ of BSTs [1] applied to AVL trees also run on AVL trees with $MT(n) \in \Theta(\log n)$ efficiency, and the result will be a BST, but possibly imbalanced. (See Example 2.2. The input of both operations is the same AVL tree, but

the outputs are imbalanced BSTs.) Clearly, after many modifications the resulting tree may become too high for efficient run of the operations of BSTs. Therefore we need some modifications on these recursive procedures: When we return from a recursive call, we check whether the actual node of the tree became imbalanced, and if so, we perform the appropriate *rotations* in order to make it balanced. The extra elementary operations needed can be limited by the same constant at each level, so efficiency $MT(n) \in \Theta(\log n)$ remains true.

The *rules of rotations* of AVL trees can be found on Figures 2-7. One can see that *each rotation* keeps the original inorder traversal of the tree. Consequently, if the input of a rotation is a BST, its output is also a BST.

And after an addition or deletion, we consider the smallest imbalanced subtree containing the position of the insertion or deletion, and apply the appropriate rotation to it. Thus its subtrees are balanced, and the result of the rotation is a balanced subtree. Sometimes some rotations must be done also at higher levels of the tree afterwards.

2.1 AVL trees: insertion

First we consider procedure insert(t, k) of BSTs [1].

For example, given AVL tree { [2] 4+ [(6) 8= (10)] }.

- Inserting key 1, we receive AVL tree
  { [(1) 2−] 4= [(6) 8= (10)] }.
- Alternatively, inserting key 3, we receive AVL tree
  { [2+ (3)] 4= [(6) 8= (10)] }.
- Alternatively, inserting key 9, we receive imbalanced BST
  { [2] 4++ [(6) 8+ ({9} 10−)] }.
- Alternatively, inserting key 7, we receive imbalanced BST
  { [2] 4++ [(6+ {7}) 8− (10)] }.

We can see that in the last 2 cases the tree became imbalanced. If we want to receive an AVL tree, we have to make the appropriate rotations on the tree. Let us see some detailed examples.

Example 2.13 *Insert key* 3 *into AVL tree* { [2] 4+ [(6) 8= (10)] }.

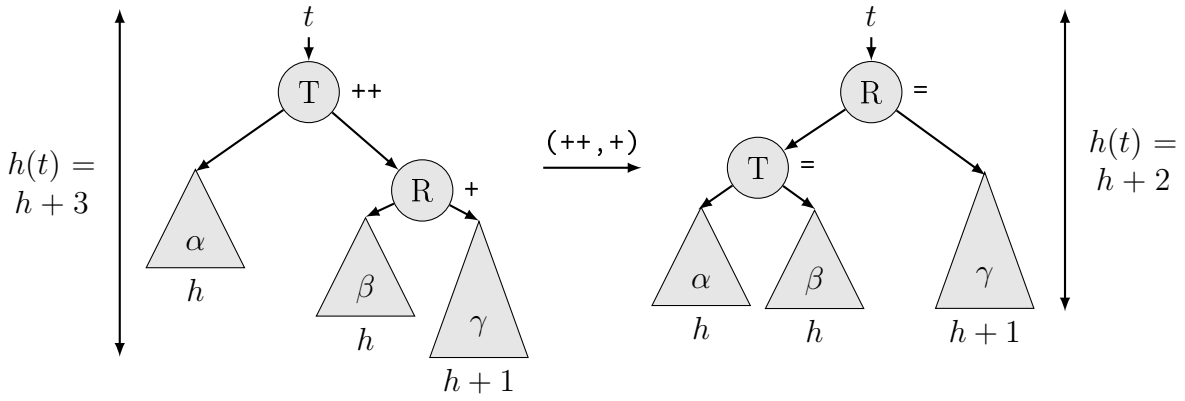


Figure 2: rotation $(++, +)$.

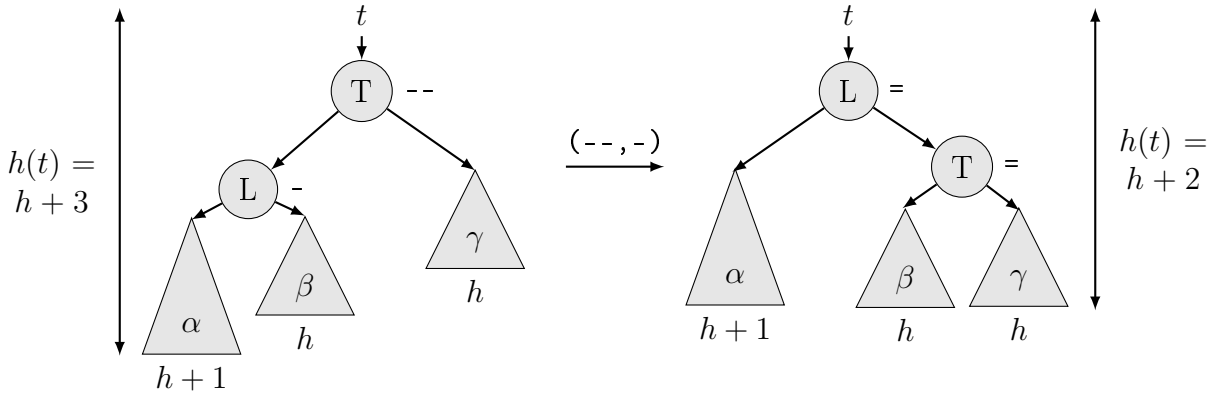


Figure 3: rotation $(--, -)$.

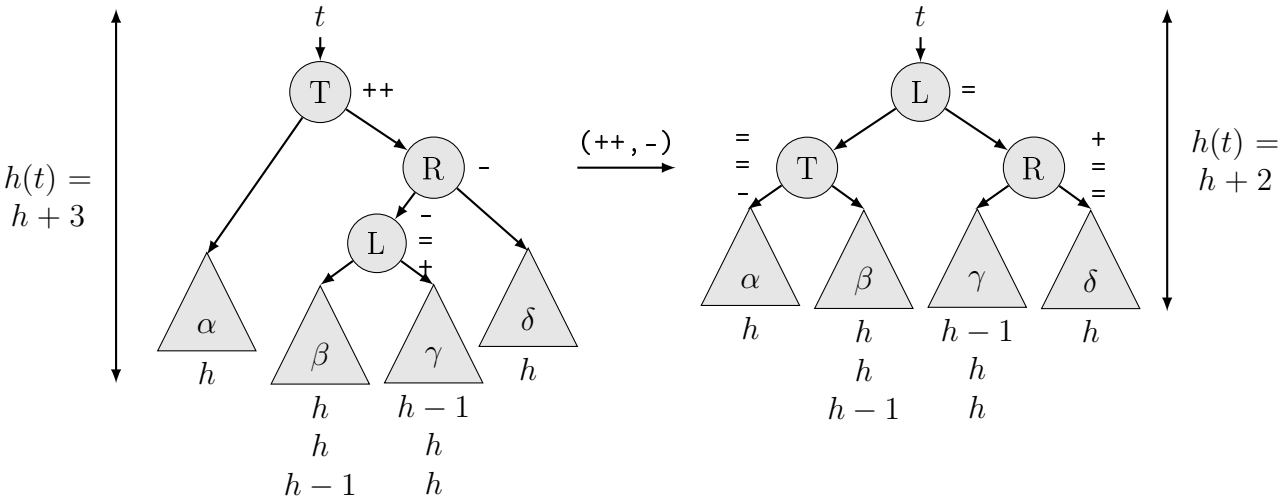


Figure 4: rotation $(++, -)$.

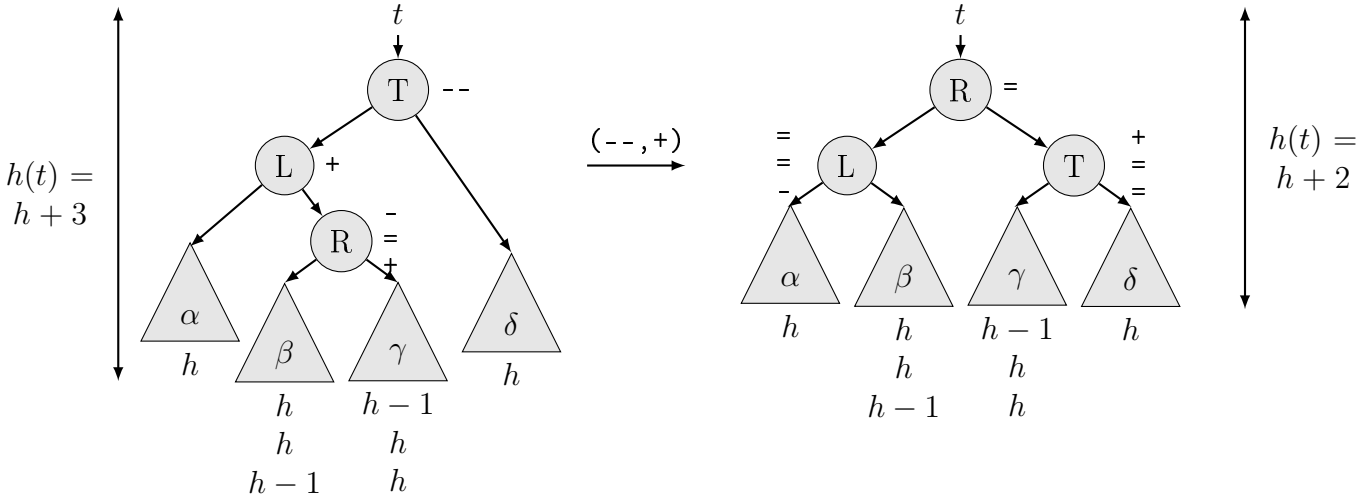


Figure 5: rotation $(--, +)$.

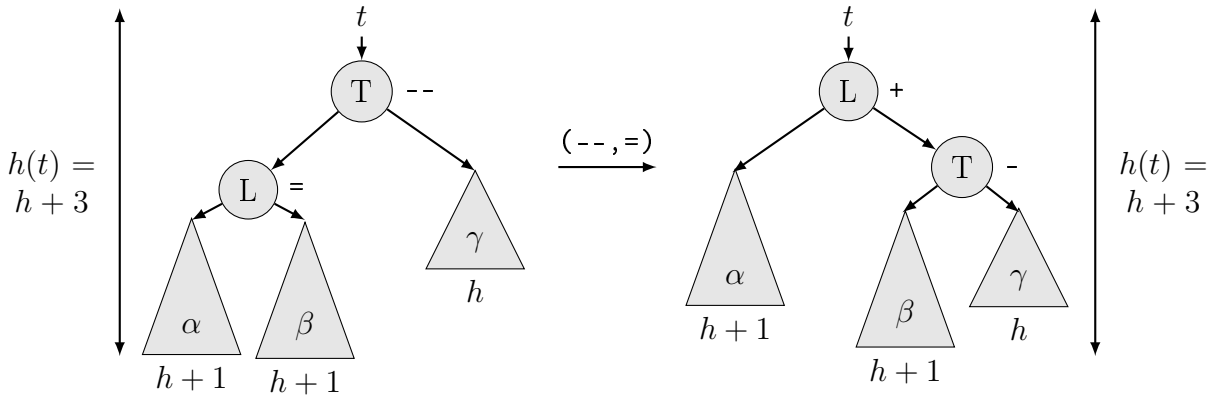


Figure 6: rotation $(--, =)$.

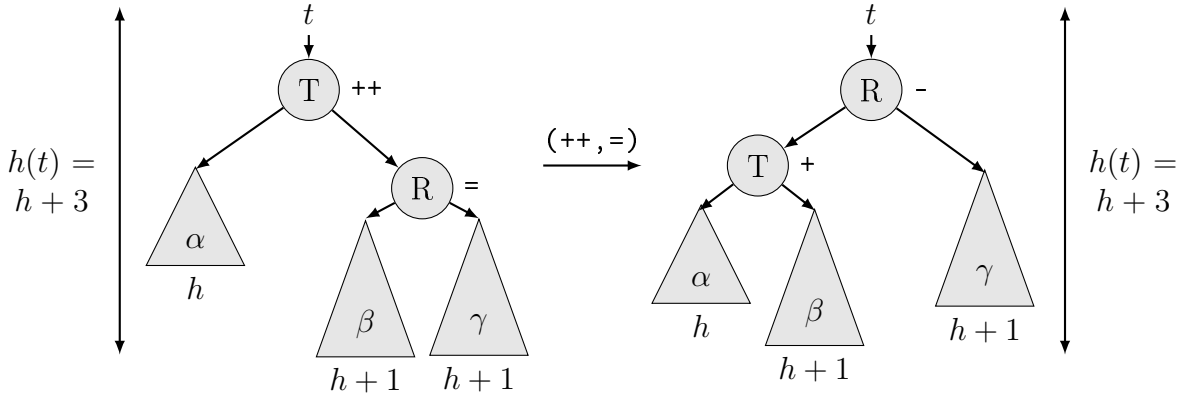


Figure 7: rotation $(++, =)$.

Solution: The key of the root of { [2] 4+ [(6) 8= (10)] } is 4.
3<4, so we go to the left subtree: [2].
3>2, so a new leaf ③ is inserted into the right, empty subtree of node ②.
Now we are at the new leaf ③. The actual subtree was empty. After inserting key 3, The actual subtree is (3). It has become higher.
Considering the balance information explicitly stored, this is the whole tree: { [2= (3)] 4+ [(6) 8= (10)] }. It is not rebalanced yet.
(Some of) the ancestors of the new leaf must be rebalanced.
We go up in the tree. First we arrive at node ②.
Its right subtree became higher, so its balance is increased by one.
Now, this is the whole tree: { [2+ (3)] 4+ [(6) 8= (10)] }.
The actual subtree is [2+ (3)], and it has become higher.
We step up in the tree. We arrive at the root node: ④.
Its left subtree has become higher, so its balance is reduced by one.
The actual subtree is the whole tree: { [2+ (3)] 4= [(6) 8= (10)] }.
We have finished rebalancing, and we can see that the result of insertion is a balanced BST, i.e. an AVL tree. Thus we have also finished insertion. We do not need any rotation here.

Exercise 2.14 *Insert key* 1 *into AVL tree* { [2] 4+ [(6) 8= (10)] }, *and show the details.*

Example 2.15 *Insert key* 9 *into AVL tree* { [2] 4+ [(6) 8= (10)] }.

Solution: We insert key 9. The key of the root is 4.
9>4, so we go to the right subtree: [(6) 8= (10)].
9>8, so we go to the right subtree: (10).
9<10, so we go to its left subtree, which is empty.
Thus we put the new node here: { [2] 4+ [(6) 8= (⟨9⟩ 10=)] }.
Then we go up and rebalance the ancestors at each level of the tree, but we stop when the first imbalanced node is found.
The subtree corresponding to this imbalanced node is the whole tree now: { [2] 4++ [(6) 8+ (⟨9⟩ 10−)] }.
We can use rotation (++, +) now (see also Figure 2):

$$[\ \alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{R}{+}\ \gamma)\ ] \to [\ (\alpha\ \mathrm{T}{=}\ \beta)\ \mathrm{R}{=}\ \gamma\ ]$$

Notice that in these rotation schemes the Greek letters are the subtrees and the English uppercase letters followed by the balance signs (like ++, +, = etc.) are the keys of the nodes.

We use the rotation scheme above on the imbalanced BST above where α=[2]=(2) ; T=4 ; β=(6) ; R=8 ; γ = (⟨9⟩ 10−) = [(9) 10−].
Finally we receive AVL tree { [(2) 4= (6)] 8= [(9) 10−] }.

Example 2.16 *Let us see an example of using rotation* $(++,-)$. *Insert key* 7 *into AVL tree* { [2] 4+ [(6) 8= (10)] }.

Solution: In order to keep the BST property, we find the appropriate empty subtree and insert node ⑦ there.
We receive tree { [2] 4+ [(6= ⟨7⟩) 8= (10)] }. Then we go up and rebalance the ancestors of the new leaf ⑦ at each level of the tree, but we stop when the first imbalanced node is found. We receive the following imbalanced BST:
{ [2] 4++ [(6+ ⟨7⟩) 8− (10)] }.
Rotation$(++,-)$ is needed (see also Figure 4):

$$\{\ \alpha\ \mathrm{T}{+}{+}\ [(\beta\ \mathrm{L}{-}{=}{+}\ \gamma)\ \mathrm{R}{-}\ \delta]\ \} \to \{\ [\alpha\ \mathrm{T}{=}{=}{-}\ \beta]\ \mathrm{L}{=}\ [\gamma\ \mathrm{R}{+}{=}{=}\ \delta]\ \}$$

In this $(++,-)$ rotation scheme, T++ is the root of the imbalanced BST. Its right child is R−. The left child of R is L.

L is the right-left grandchild of the imbalanced node T++. After the rotation, L becomes the new root of this (sub)tree. L's parent and grandparent become its two children. and the four subtrees: α, β, γ *and* δ *remain in their original order. The inorder traversal of the whole (sub)tree remains the same.*

L may have balance −, =, or +. After rotation, the new balance of T and R depends on the old balance of L. If the old balance of L was −, the new balance of T will be =, and that of R will be +, etc. (See Figure 4 for more details.) We use this later rotation scheme on the previous imbalanced BST { [2] 4++ [(6+ ⟨7⟩) 8− (10)] } where α=[2]=(2) ; T=4 ; β=⦸ ; L=6 ; γ=⟨7⟩=(7) ; R=8 ; δ=(10). The old balance of L=6 was +, so the new balance of T=4 will be −, and the new balance of R=8 will be =. Applying this rotation rule, we receive AVL tree
{ [(2) 4−] 6= [(7) 8= (10)] }.

Let us notice, if the balance of node L was b before applying this $(++,-)$ rotation rule, then (after applying this rotation rule) the new balance of T will be b_t, and that of R will be b_r where

$$b_t = -\lfloor (b+1)/2 \rfloor \quad \text{and} \quad b_r = \lfloor (1-b)/2 \rfloor .$$

We get similarly the rotation rules of the cases when the balance of the root of the smallest imbalanced subtree is −− (see also Figures 3 and 5.):

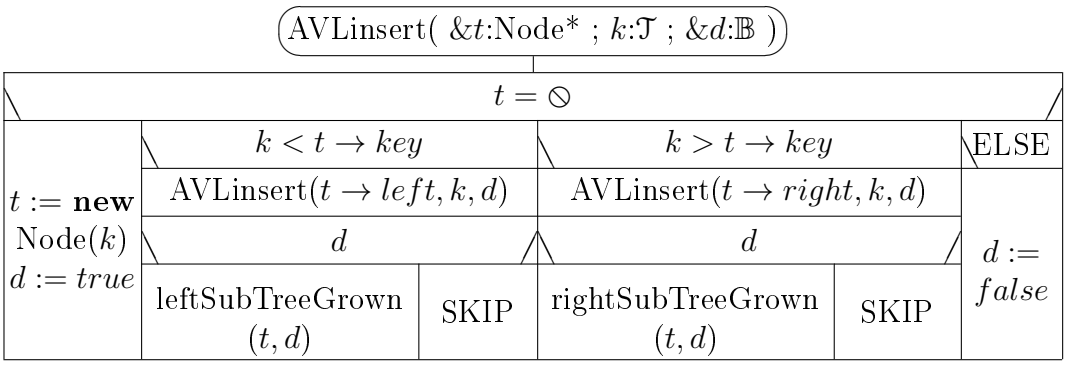
$$[\ (\alpha\ \mathrm{L}{-}\ \beta)\ \mathrm{T}{-}{-}\ \gamma\ ] \to [\ \alpha\ \mathrm{L}{=}\ (\beta\ \mathrm{T}{=}\ \gamma)\ ]$$
$$\{\ [\alpha\ \mathrm{L}{+}\ (\beta\ \mathrm{R}{-}{=}{+}\ \gamma)]\ \mathrm{T}{-}{-}\ \delta\ \} \to [\ (\alpha\ \mathrm{L}{=}{=}{-}\ \beta)\ \mathrm{R}{=}\ (\gamma\ \mathrm{T}{+}{=}{=}\ \delta)\ ]$$
$$b_l = -\lfloor (b+1)/2 \rfloor \quad \text{and} \quad b_t = \lfloor (1-b)/2 \rfloor$$

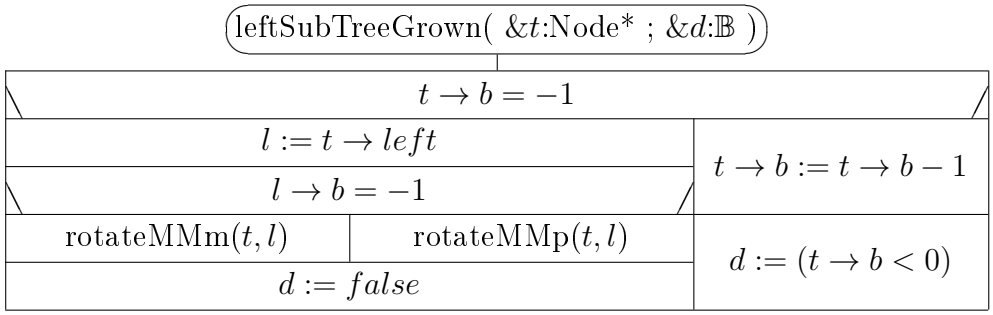
where b was the balance of node R before applying the $(--,+)$ rotation rule; but after applying this rule the new balance of L will be b_l, and that of T will be b_t.

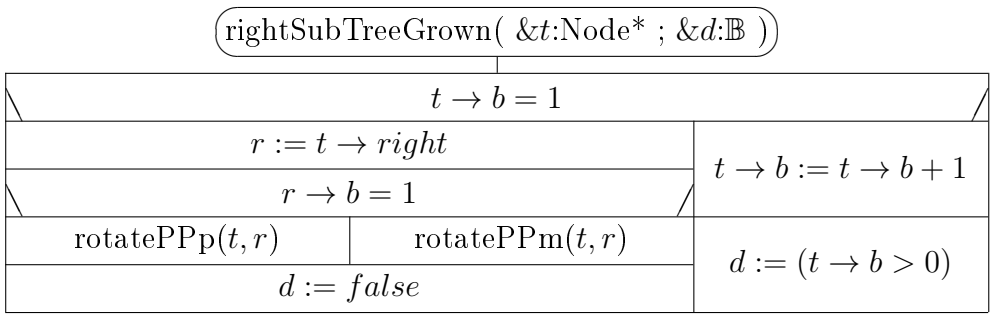
In the $(--,+)$ rotation scheme, T$--$ is the root of the imbalanced BST. Its left child is L$+$. The right child of L is R:

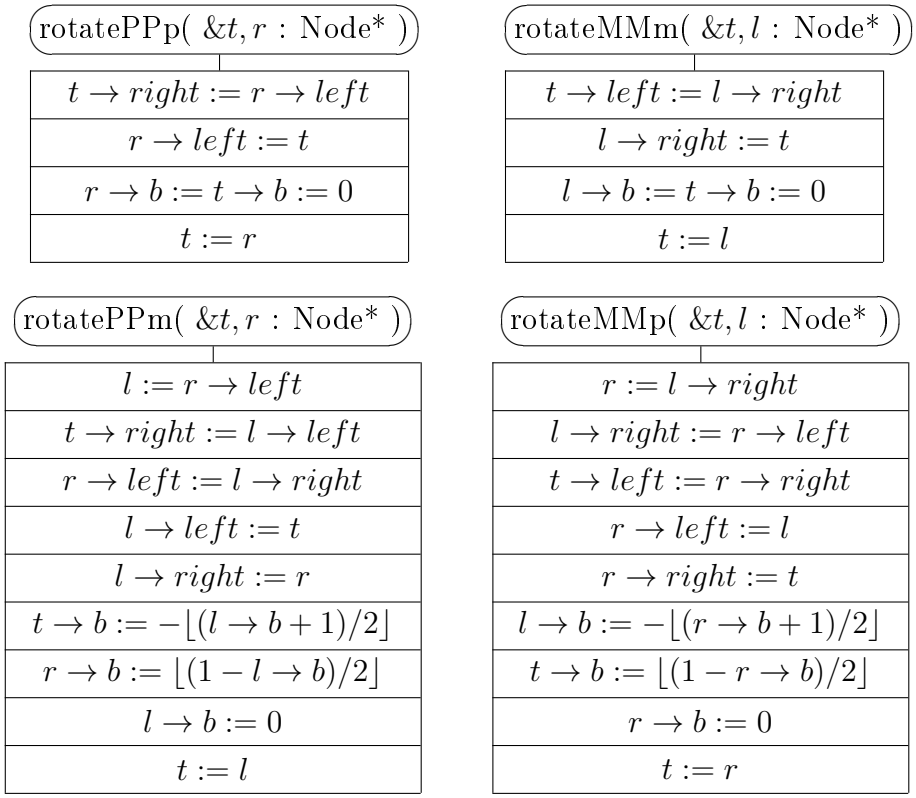
R is the left-right grandchild of the imbalanced node T$--$. After the rotation, R becomes the new root of this (sub)tree. R's parent and grandparent become its two children. and the four subtrees: α, β, γ and δ remain in their original order. The inorder traversal of the whole (sub)tree remains the same.

Let us see the detailed codes of insertion. Compared to the insert procedure of BSTs, we use an extra parameter here: d is a reference parameter of Boolean type. After the call, it is true, if the height of the actual subtree increased by one, and it is false, if its height did not change.









Notice that the rotation rules $(--,=)$ and $(++,=)$ on Figures 6 and 7 are never applied in insertion. (They will be applied in deletion.)

And one insertion applies maximum one of the rotation rules on Figures 2-5 and maximum once, because a successful insertion creates a new leaf, and then we go up level by level in the tree rebalancing the ancestors of the new leaf until

- we find an imbalanced node. In this case, the height of the actual subtree has been increased by one. Then we apply the appropriate rule, i.e. one of those on Figures 2-5. And each of them decreases the height of the actual subtree by one. (See Figures 2-5 for the details.) Thus the original height of this subtree has been restored, and we just leave insertion. Consequently, **in insertion, we never modify the balances of the nodes outside of the smallest imbalanced subtree.**

- rebalancing a node we find that it is perfectly balanced. This means that one of its direct subtrees has grown to the height of the other. Therefore the height of the subtree corresponding to the actual node has not been changed by the insertion. Thus we have finished rebalancing.

We did not find imbalanced node, and the insertion has been finished without rotation.

- we arrive at the root of the whole tree but do not find imbalanced node. This case is similar to the previous one: no rotation is needed, because the BST remained balanced.

Example 2.17 *Let us see an example of the first case above, especially when the imbalanced node is not the root of the whole tree. Insert key* 7 *into AVL tree* { [(1) 2−] 3+ [(⟨4⟩ 5= ⟨6⟩) 8− (9)] }.

Solution: After inserting key 7, but still before rebalancing we have the following tree: { [(1) 2−] 3+ [(⟨4⟩ 5= ⟨ 6= {7} ⟩) 8− (9)] }.
We rebalance nodes ⑥, ⑤, and ⑧. We stop rebalancing here, because node ⑧−− has become imbalanced (thus node ③+ is not rebalanced):
{ [(1) 2−] 3+ [(⟨4⟩ 5+ ⟨ 6+ {7} ⟩) 8−− (9)] }
Next we apply rotation $(--, +)$ at node ⑧−−.
The actual rule of rotation is
[(α L+ ⟨β R+ γ⟩) T−− δ] → [(α L− β) R= (γ T= δ)]
with α=⟨4⟩, L=5, β=⦸, R=6, γ={7}=⟨7⟩, T=8, δ=(9)=⟨9⟩
(see also Figure 5).
The result: { [(1) 2−] 3+ [(⟨4⟩ 5−) 6= (⟨7⟩ 8= ⟨9⟩)] }.

Exercise 2.18 *Let us modify the* Node *type of* AVL trees *so that we store the height of the corresponding subtree in each node, but we do not store the balances.*

The four simpler rules of rotations on figures 2, 3, 6, 7 can be reduced to two rules + recalculating the heights of the modified subtrees. The two simple rotations:
Right-to-left rotation: $[\alpha\ T\ (\beta\ R\ \gamma)] \to [(\alpha\ T\ \beta)\ R\ \gamma]$
Left-to-right rotation: $[(\alpha\ L\ \beta)\ T\ \gamma] \to [\alpha\ L\ (\beta\ T\ \gamma)]$
Which one should be used in the different cases?

The more complex rules of rotations which correspond to figures 4 and 5 can be considered double rotations *now, because we can get them as two simple rotations applied at the appropriate points of the the tree + recalculating the heights of the modified subtrees.*

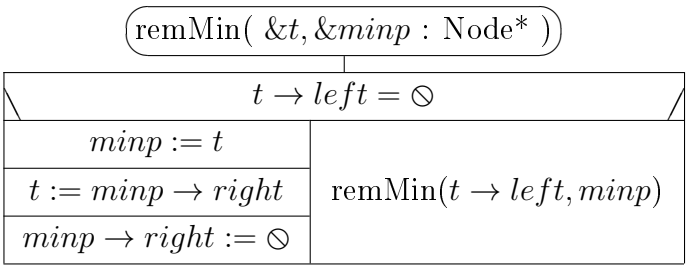
How to use the simple rotations above in the different cases?

Exercise 2.19 *At structogram (and/or C/C++) level, implement insertion using the new rules of rotations developed in Exercise 2.18.*

Compare your implementation of insertion to the corresponding structograms given above. What can we say about the complexity of the code? What can we tell about efficiency issues?

2.2 AVL trees: removing the minimal (maximal) node

On nonempty BSTs, procedure remMin removes the minimal node (i.e. the node with minimal key):



Applying this to AVL tree { [2] 4+ [(6) 8= (10)] }, we receive the imbalanced BST { 4++ [(6) 8= (10)] } while *minp* refers to the minimal node of the original tree where $minp \to key = 2 \wedge minp \to left = minp \to right = \oslash$. Notice that we show the theoretical balances above. At representation level, the code above does not do anything with the balances (b attributes). After removing the minimal node, we have to go up from the place of it, and level by level rebalance the ancestors of it. If we find a node with "++" balance, the appropriate rotation must be applied. (Notice that balance "−−" is not possible when we remove the minimal node.)

Rotation $(++, =)$ is needed here (see Figure 7 for the details):

$$\{\ \alpha\ \mathrm{T}{+}{+}[\beta\ \mathrm{R}{=}\ \gamma]\ \} \to \{\ [\alpha\ \mathrm{T}{+}\ \beta]\ \mathrm{R}{-}\ \gamma\ \}.$$

Notice that this rotation does not modify the height of the actual subtree. Therefore, neither more rebalancing nor more rotations are needed after this rotation, even if this rotation is applied to a proper subtree of the whole BST. The final result is { [4+ (6)] 8− (10) } here.

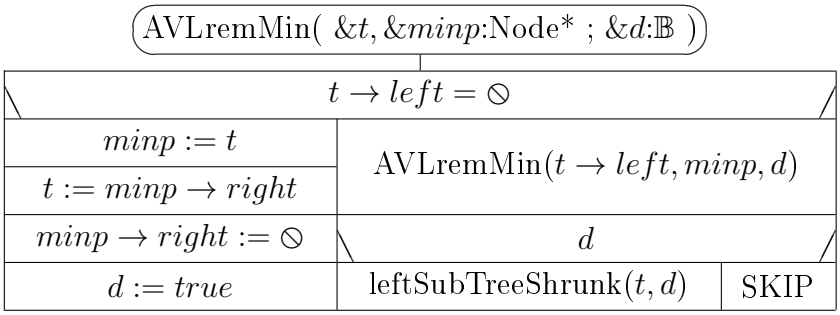
Notwithstanding, when rotation $(++, +)$ or $(++, -)$ is needed, then the rotated subtree becomes lower, and after the rotation, rebalancing goes on. Then it is possible that two or more rotations are needed at different levels. However, each rotation needs constant time, and the number of levels is not more than $1.45 \log n$. Thus the maximal runtime is still $\Theta(\log n)$.

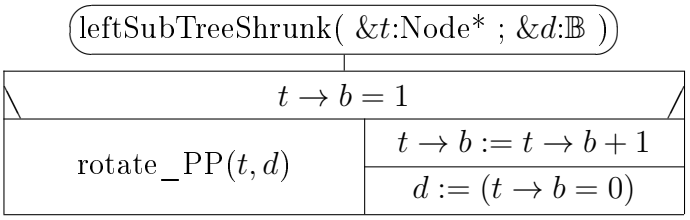
Example 2.20 *Remove the minimal node from AVL tree*
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] } .
We need two rotations here.

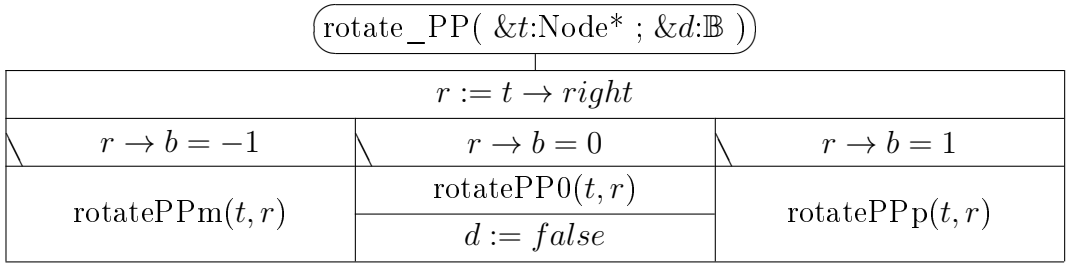
Solution: We remove the leftmost node, i.e. node ①. The appropriate subtree becomes empty. Its height is decreased by one. Then the balance of node ② becomes "++". The subtree rooted by node ② needs rotation

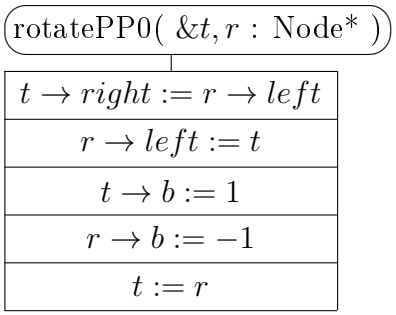
(++,+): [2++ (3+ ⟨4⟩)] → [(2) 3= (4)]. The height of this subtree is decreased by one. (Rotation (++,+) always decreases the height of the rotated subtree by one.) Consequently the balance of node ⑤ is increased by one. The whole tree now:
{ [(2) 3= (4)] 5++ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] }.
The right child of node ⑤++ is ⑩−. Thus we need rotation (++, −). The left child of ⑩− is ⑦+. After rotation (++, −) (see Figure 4 for the details), always the right-left grandchild of the root of the rotated subtree becomes its root. This is ⑦+ now. Its original parent and grandparent becomes its two children, and the four subtrees are arranged in order:
{ [(⟨2⟩ 3= ⟨4⟩) 5− (6)] 7= [(⟨8⟩ 9−) 10= (11+ ⟨12⟩)] }.
The balances of the three nodes are determined by the old balance of the original grandson (⑦+ here), according to Figure 4. The four subtrees are not changed. (Just the brackets were rewritten here for better reading.)

Based on the rules discussed above, we can complete the code of procedure remMin($t, minp$) above. For AVL trees, we need an extra Boolean parameter d which is *true*, **iff** the height of the appropriate subtree has been decreased by one. (It cannot decrease by two or more.) When we return from a recursive call, we check d. If it is *true*, we have to go on with rebalancing and possibly rotations. If d is *false*, neither subsequent rebalancing nor rotations are needed.





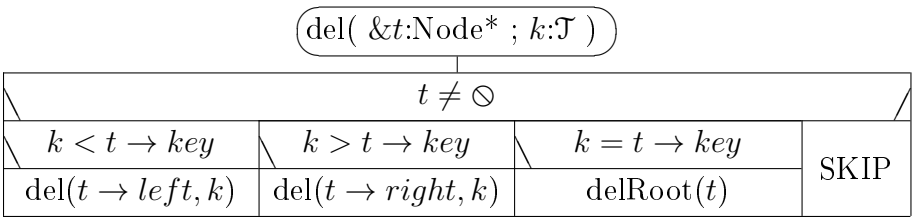




Exercise 2.21 *Based on procedure* AVLremMin$(t, minp, d)$ *above, write the structograms of procedure* AVLremMax$(t, maxp, d)$. *You will need the code of rotation rules* $(--, -)$, $(--, +)$ *and* $(--, =)$ *defined on Figures 3, 5 and 6. You should write new code for rotation rule* $(--, =)$ *defined on Figure 6.*

2.3 AVL trees: deletion

Procedures del(t, k) and delRoot(t) for BSTs:



delRoot(&t:Node*)

$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p := t$	$p := t$	remMin$(t \to right, p)$
$t := p \to right$	$t := p \to left$	$p \to left := t \to left$
		$p \to right := t \to right$
delete p	**delete** p	**delete** t ; $t := p$

These are completed below as it was done in case of procedure AVLremMin$(t, minp, d)$ in section 2.2. We add Boolean parameter d which is *true*, **iff** the height of the appropriate subtree has been decreased by one. (It cannot decrease by two or more.) When we return from a recursive call, we check d. If it is *true*, we have to go on with rebalancing and possibly rotations. If d is $false$, neither subsequent rebalancing nor rotations are needed.

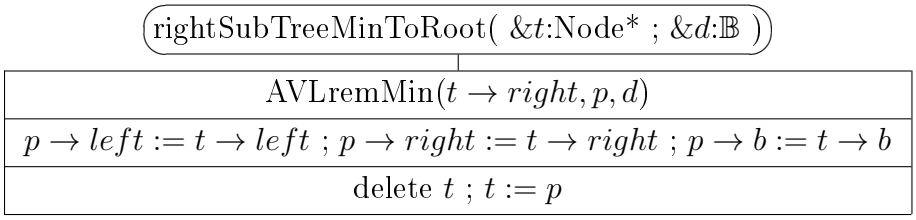
AVLdel(&t:Node* ; k:$\mathcal{T}$; &d:$\mathbb{B}$)

<table>
<tr><td colspan="5">$t \neq \oslash$</td></tr>
<tr><td colspan="2">$k < t \to key$</td><td colspan="2">$k > t \to key$</td><td>$k = t \to key$</td></tr>
<tr><td colspan="2">AVLdel$(t \to left, k, d)$</td><td colspan="2">AVLdel$(t \to right, k, d)$</td><td rowspan="3">AVLdelRoot (t, d)</td></tr>
<tr><td colspan="2">d</td><td colspan="2">d</td></tr>
<tr><td>leftSubTreeShrunk (t, d)</td><td>SKIP</td><td>rightSubTreeShrunk (t, d)</td><td>SKIP</td></tr>
</table>

$d :=$ $false$

AVLdelRoot(&t:Node* ; &d:$\mathbb{B}$)

<table>
<tr><td>$t \to left = \oslash$</td><td>$t \to right = \oslash$</td><td colspan="2">$t \to left \neq \oslash \wedge t \to right \neq \oslash$</td></tr>
<tr><td>$p := t$</td><td>$p := t$</td><td colspan="2" rowspan="2">rightSubTreeMinToRoot(t, d)</td></tr>
<tr><td>$t := p \to right$</td><td>$t := p \to left$</td></tr>
<tr><td>**delete** p</td><td>**delete** p</td><td colspan="2">d</td></tr>
<tr><td>$d := true$</td><td>$d := true$</td><td>rightSubTreeShrunk(t, d)</td><td>SKIP</td></tr>
</table>



When rotation $(++,=)$ or $(--,=)$ is performed, it does not modify the height of the actual subtree. Therefore, neither more rebalancing nor more rotations are needed after one of these rotations, even if the rotation is applied to a proper subtree of the whole BST.

Notwithstanding, when rotation $(++,+)$, $(--,-)$, $(++,-)$ or $(--,+)$ is needed (see Figures 2-5), then the rotated subtree becomes lower, and after the rotation, rebalancing goes on. Then it is possible that two or more rotations are needed at different levels. However, each rotation needs constant time, and the number of levels is not more than $1.45 \log n$. Thus the maximal runtime of deletion is also $\Theta(\log n)$.

Exercise 2.22 *Based on procedure* leftSubTreeShrunk(t, d) *write procedure* rightSubTreeShrunk(t, d) *together with its auxiliary procedures. (See rotation rule* $(--,=)$ *given on Figure 6.)*

Example 2.23 *Delete key 2 from AVL tree*
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] } .

Solution: We delete node ②. AVLremMin is called on its right subtree, and node ③ is removed from it, so it will be AVL tree (4). No rotation is needed during AVLremMin, but the height of this right subtree is decreased by one, so AVLremMin returns with $d = true$. Node ② is substituted by node ③ which inherits the balance of node ②. Then its balance is decreased by one because $d = true$. Thus the actual subtree becomes [(1) 3= (4)]. The height of this subtree has been decreased by one. Consequently the balance of node ⑤ is increased by one. The whole tree now:
{ [(1) 3= (4)] 5++ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] }.
The right child of node ⑤++ is ⑩−. Thus we need rotation $(++,-)$. The left child of ⑩− is ⑦+. After rotation $(++,-)$ (see Figure 4 for the details), always the right-left grandchild of the root of the rotated subtree becomes its root. This is ⑦+ now. Its original parent and grandparent becomes its two children, and the four subtrees are arranged in order:
{ [(⟨1⟩ 3= ⟨4⟩) 5− (6)] 7= [(⟨8⟩ 9−) 10= (11+ ⟨12⟩)] }.

The balances of the three nodes are determined by the old balance of the original grandchild (⑦+ here), according to Figure 4. The four subtrees are not changed. (Just the brackets were rewritten here for better reading.)

Example 2.24 *Delete key 5 from AVL tree*
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)] } .

Solution: In order to delete node ⑤, first AVLremMin is called on its right subtree: [(⟨6⟩ 7+ ⟨ {8} 9− ⟩) 10− (11+ ⟨12⟩)]. AVLremMin is called recursively on its left subtree: (⟨6⟩ 7+ ⟨ {8} 9− ⟩). We remove its lefmost node ⑥. The balance of its parent ⑦+ is increased by one. The corresponding subtree is (7++ ⟨ {8} 9− ⟩). Rotation (++, −) is performed on it. The result is (⟨7⟩ 8= ⟨9⟩). Its height is decreased by one, so the recursive call to AVLremMin returns with $d = true$, and the balance of ⑩− is increased by one. The corresponding subtree is
[(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)].
Its height is decreased by one, so the first call to AVLremMin returns with $d = true$. After the run of AVLremMin the whole tree is
{ [(1) 2+ (3+ ⟨4⟩)] 5+ [(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)] }.
Node ⑤ is substituted by node ⑥:
{ [(1) 2+ (3+ ⟨4⟩)] 6+ [(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)].
$d = true$ still shows that height of its right subtree was decreased by one, so the balance of ⑥+ is also decreased by one. The final AVL tree is
{ [(1) 2+ (3+ ⟨4⟩)] 6= [(⟨7⟩ 8= ⟨9⟩) 10= (11+ ⟨12⟩)].

Exercise 2.25 *Write an alternative code of procedure* AVLdel(t, k, d). *It shall be similar to the previous one, but if key k is in a node with two children, it shall substitute this node with the maximal node of its left subtree.*

3 General trees

Compared to binary trees, a node of a *rooted tree* may have unlimited (although finite) number of children. A rooted tree is *ordered* if the order of the children of nodes is important. An ordered rooted tree is also called *general tree*. (See Figure 8.) We will not define empty subtrees of *general trees*.

Using *general trees* we can model hierarchical structures like

- directory hierarchies in computers
- (block) structure of programs

- different kinds of expression in mathematics and computer science
- family trees etc.

General trees can be represented with linked binary trees in a natural way. Type Node of the nodes of general trees is given below.

- Pointer $firstChild$ refers to the first child of a node.
  $p \to firstChild = \oslash$, **iff** node $(*p)$ is a leaf.
- Pointer *nextSibling* refers to the next sibling in a list of children.
  $p \to nextSibling = \oslash$, **iff** $(*p)$ is the root node of the tree, or it is the last sibling in a list of children.

Node
+ $firstChild, nextSibling$: Node*   + key : $\mathcal{T}$
+ Node() { $firstChild := nextSibling := \oslash$ }   + Node(x:$\mathcal{T}$) { $firstChild := nextSibling := \oslash$; $key := x$ }

In the nodes, there might be *parent* pointers referring to the parent of the children. We will not use parent pointers here.

Exercise 3.1 *Try to invent alternative (linked) representations of general trees. Compare your representations to the linked binary representation above. Consider memory needs and flexibility.*

In the textual or parenthesized representation of general trees we start with the root of the actual (sub)tree. Thus a nonempty tree fits the general scheme

$$(R \;\; t_1 \ldots t_n)$$

where R is the content of the root node and $t_1 \ldots t_n$ are the direct subtrees of node R.

Example 3.2 *In general tree* { 1 [2 (5)] (3) [4 (6) (7)] }, 1 *is in the root. Its children are the nodes with keys* 2, 3 *and* 4. *The corresponding subtrees are* [2 (5)], (3) *and* [4 (6) (7)] *in order. The leaves of the tree are the nodes with keys* 5, 3, 6 *and* 7. *(See* Figure 8.*)*

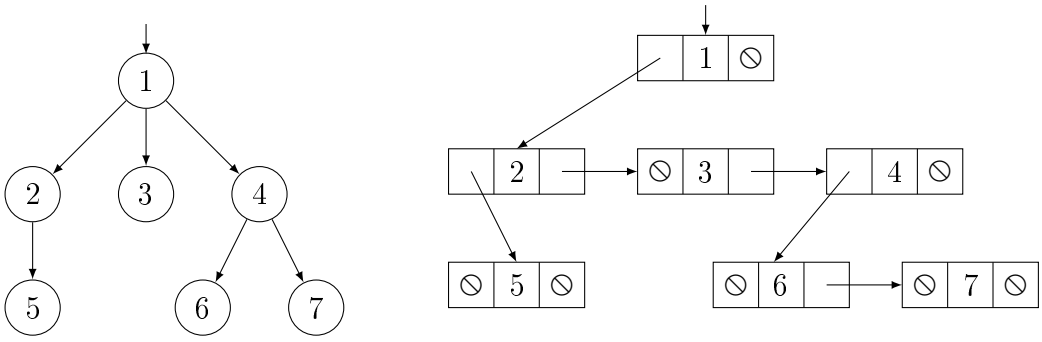


Figure 8: The abstract structure of a general tree on the left and its linked binary representation on the right. Its textual or parenthesized representation is { 1 [2 (5)] (3) [4 (6) (7)] }. (The different kinds of brackets may vary.)

Exercise 3.3 *Write a procedure printing a* general tree *given in* linked binary representation. *The result should be in textual representation.*

Write another procedure which can build a general tree *in* linked binary representation. *Its input is a textfile where the tree is given in parenthesized form.*

Write these printing and reading procedures for the linked representations you invented in Exercise 3.1.

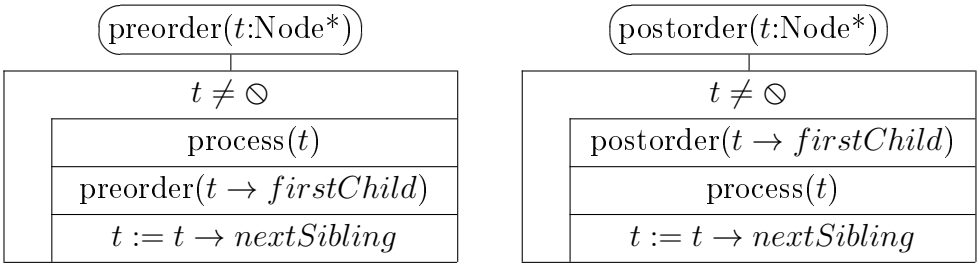
(The printing procedures are usually more straightforward than those which build up the linked representation from the textual form.)

In order to make the *preorder* traversal of a general tree, we have to traverse its binary representation also with *preorder* traversal where $firstChild$ corresponds to $left$ and *nextSibling* corresponds to *right*.

But the the *postorder* traversal of a general tree requires the *inorder* traversal of its binary representation.[6]

The *preorder* and *postorder* traversals of a general tree are given below, provided that it is given in linked binary representation.

[6] A search in the file system of a computer requires *preorder* traversal, while evaluating an expression requires *postorder* traversal of the general (i.e. theoretical) expression tree.



Exercise 3.4 *Write the traversals of general trees above for the representations you invented while solving* Exercise 3.1.

Given a general tree *in some representation, write a procedure copying it into another given representation.*

4 B+ trees and their basic operations

See file **http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs2/B+trees.pdf**.