New Knowledge on AVL-Trees

Markus E. Nebel

Johann Wolfgang Goethe - Universität
Fachbereich Informatik
D-60054 Frankfurt/Main
Germany

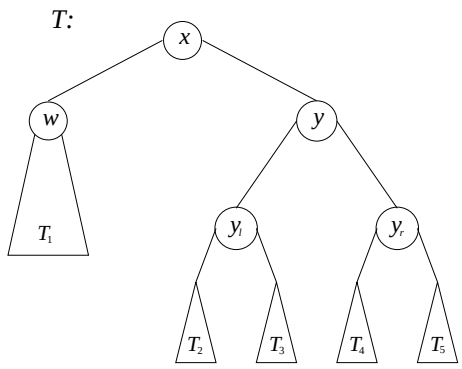
Abstract

In this note we prove the following close relation between AVL- and binary search trees: Building an AVL-tree T for n distinct keys implies a sequence of rebalancing rotations that are applied immediately. If we first construct the binary search tree for those n keys and then perform a final rebalancing run in which the same sequence of rotations is executed then the resulting tree is the same as T.

Key words: Algorithms, data structures

1 Introduction

It is well known how to construct a binary search tree (BST) from n distinct keys to handle the data efficiently. Since BST's tend to be balanced (their average height is of order $4.31107 \log n$ [Dev86]), operations like the insertion, deletion or look up of keys can be performed at low-cost [BYG91].
It is possible to improve the behaviour of BST's by an algorithm due to G.M. **A**del'son-**V**el'skii and E. M. **L**andis, the so called AVL-trees, which generates binary search trees with a maximal height of $1.4404 \log n$ by means of rotation operations (see [AdVL62]).
In this note we investigate the relation between the BST's and the AVL-trees. We prove that even if AVL-trees perform rebalancing rotations between the insertion of different keys we will achieve the same tree when all keys are inserted without rebalancing which is delayed to a final rebalancing run. This property alone is of interest by itself but it also might have consequences to applications as we will see in section 4.



T_1	T_2	T_3	T_4	T_5
h	$h-1$ DL	$h-1$ DL	$h-1$ L	$h-1$ L
h	$h-1$ DL	$h-1$ DL	$h-1$ -	$h-2$ -
h	$h-1$ DL	$h-1$ DL	$h-2$ -	$h-1$ -
h	$h-1$ -	$h-2$ -	$h-1$ L	$h-1$ L
h	$h-2$ -	$h-1$ -	$h-1$ L	$h-1$ L

Fig. 1. Rotation types in dependence on subtree heights and insertion positions.

2 AVL-Trees

In this section we recall the fundamental definitions of AVL-trees. We preassume the reader to be familiar with the notion of binary search trees and the corresponding parameters and algorithms. If not, refer to [Knu73] to get the needed information.

The *balance-degree* of a node is the difference of the heights of its left and its right subtree. A BST is called *height balanced* if its nodes have a balance-degree of modulus less or equal to 1. The AVL-algorithm forces BST's to be height balanced by performing a rebalancing rotation whenever the balance degree of a node gets out of range due to an insertion or deletion. Note, that in case of an insertion one rotation suffices to rebalance the tree. Further, it is known that the probability of a rotation in an insertion step is between 0.37 and 0.73 and also that the fraction of balanced nodes is between 0.56 and 0.78 [BYGZ90].

We need two different kinds of rotations, single and double, each of them exists for a left and a right subtree out of balance. Because of this symmetry we will only discuss the single-left rotation (L) and double-left rotation (DL) in detail. Figure 1 shows all insertion situations, where a left-rotation or double-left-rotation might be needed. Both will turn the node marked with x of tree T into the left son of the new root-node. This is the node marked with y in the case of a left-rotation and the node marked with y_l in the case of a double-left-rotation. We say that the particular rotation is performed *around* x. The table of Figure 1 shows the different conditions for the heights of the subtrees T_1 to T_5 together with the corresponding rotation types implied by an insertion. For instance, line 3 of the table signifies, that a DL must be performed, if an insertion in subtree T_2 or T_3 causes its height to grow from $h-1$ to h. In all cases the balance of node x gets out of range. Figure 2 shows the structure of tree T after a single-left (resp. double-left) rotation around

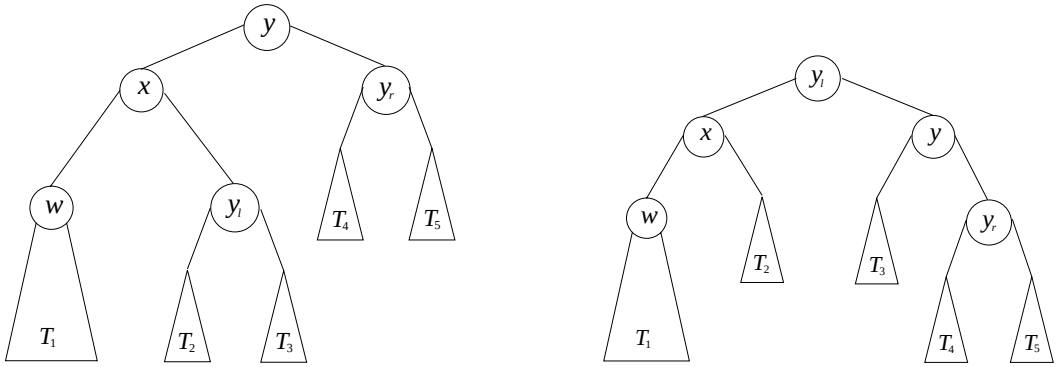


Fig. 2. The tree of Figure 1 after a single-left (left tree) and after a double-left rotation (right tree).

x was performed. Note, that the property of being a search tree is respected by the rotation operations, i.e. the inorder of the tree is equal to the sorted sequence of the keys stored.

3 The Results

In this section we give a formal description and a proof of the property of AVL-trees stated in the introduction. We will consider an arbitrary set of keys K together with an order $(K, <)$ which is used to compare the keys. We regard those AVL-trees (BST's), which are generated by starting with an empty tree and inserting a permutation of distinct keys of K by the standard AVL- (BST-) insertion algorithm.
Let T be a BST and $k \in K$. We write $k \in T$ if the key k is stored in T. For a set $\mathcal{T}$ of BST's $\mathcal{K}(\mathcal{T}) := \{k \in K | (\exists T \in \mathcal{T})(k \in T)\}$. The intersection of two trees T_1 and T_2 is denoted by $T_1 \cap T_2$. It is defined as

$$T_1 \cap T_2 := \begin{cases} \emptyset & : T_1 = \emptyset \vee T_2 = \emptyset \vee \mathfrak{r}(T_1) \neq \mathfrak{r}(T_2) \\ \mathfrak{r}(T_1) \circ (T_1^l \cap T_2^l, T_1^r \cap T_2^r) & : \text{otherwise} \end{cases} .$$

Here, $\emptyset$ denotes the empty tree, $\mathfrak{r}(T)$ represents the root of tree T. T^l (resp. T^r) denotes the left (resp. right) subtree of T and $x \circ (T, T')$ stands for the tree with root x, the left subtree T and the right subtree T'. In words, $T_1 \cap T_2$ is the substructure that both trees have in common (including node labels), when the two trees are laid one on top of the other (one root upon the other). Analogously, we denote by $T_1 \backslash T_2$ the forest that we get when deleting the nodes [1] of T_1 that it has in common with $T_1 \cap T_2$. The notation $T_2 \subseteq T_1$ is used whenever T_2 is a substructure of T_1, i.e. there is a subtree T_1' of T_1

[1] All edges that are incident with at least one of these nodes are deleted as well.

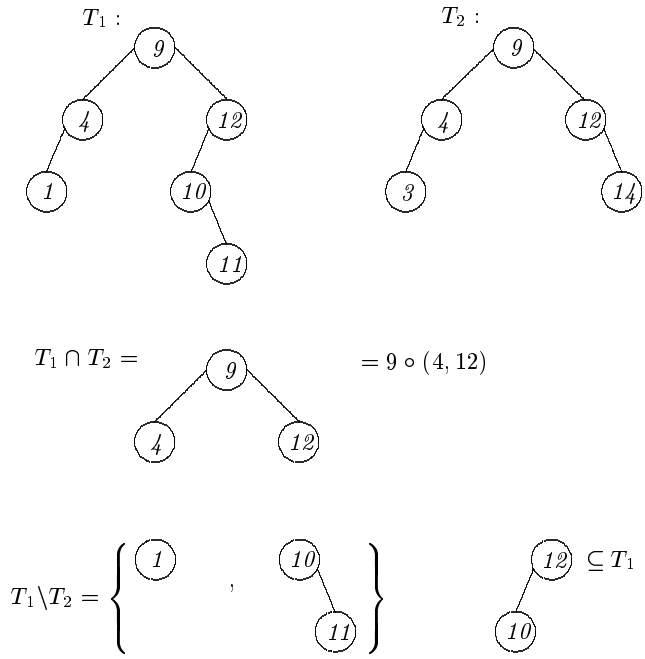


Fig. 3. An example for the notation used within the paper.

with $T_1' \cap T_2 = T_2$. The *size* of a substructure is the number of its nodes. Let $S := \{T, T_1, T_2, \ldots, T_m\}$ be a set of BST's. Then $T \Cup \{T_1, \ldots, T_m\} := \{T' \in \mathsf{BST} | T' \cap T = T \wedge T' \backslash T = \{T_1, \ldots, T_m\}\}$. This is the set of all BST's that we can get by attaching the trees T_i, $1 \leq i \leq m$, to the leaves of tree T.
In Figure 3 these notations are shown in an example.

It is well known, that the extended tree for a BST T with n keys has exactly $n+1$ leaves [2] $l_1, l_2, \ldots, l_{n+1}$ each of them could be the location of the next insertion. Each leaf is associated with an interval that determines the keys that might be inserted at its position. If the leaf is the right (resp. left) son of the node storing key x, the interval is $]x : m[$ (resp. $]m : x[$) where m denotes the $\min\{z | x < z \wedge z \in \mathcal{K}(T)\}$ (resp. $\max\{z | z < x \wedge z \in \mathcal{K}(T)\}$); it is ∞ ($-\infty$), if that minimum (resp. maximum) does not exist. Let $\mathcal{I}_i$ denote the interval associated with the i-th leaf of tree T. Then $(\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n+1})$ is called *characteristic* of T. If we insert the keys of permutation p' (all keys of p' are distinct from those that are already in T) into T, the new keys are stored in place of some of the leaves l_i (or as successor to them). Thus we generate a tree T' with subtrees T_i (possibly empty) that are rooted in the positions of the leaves l_i of T, $1 \leq i \leq n+1$. We call such trees *frontier-trees of T with respect to p'*. Note, that $T' \backslash T = \{T_1, \ldots, T_{n+1}\}$ holds. Further the characteristic of a tree does only depend on the nodes stored in the tree and not on its structure. Thus, an AVL-rotation does not change the characteristic of a tree.
The class of AVL-trees is a subclass of the BST's so every definition that is

[2] These leaves represent the NIL-pointers of the nodes. In our figures we picture them as $\Box$.

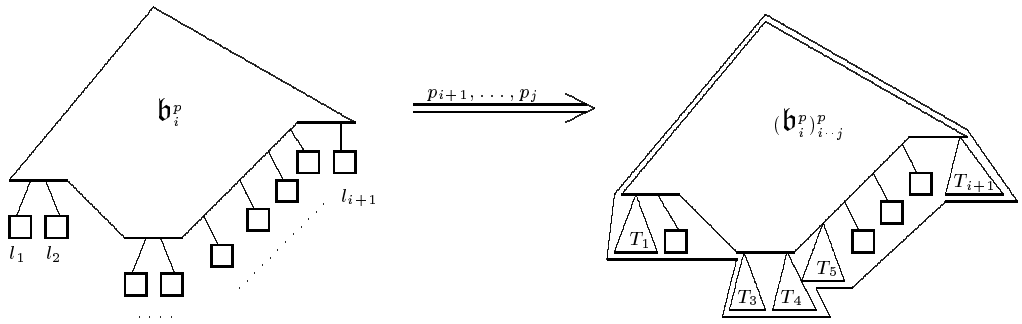


Fig. 4. The creation of frontier-trees and their location.

made for a binary search tree is also valid for an AVL-tree.

Definition 1 *Let $\mathfrak{b}$ be a* BST *and x be a node of $\mathfrak{b}$. Then, we denote the left son of x by $x.l$ and the right son of x by $x.r$. If x is a leaf then both, $x.l$ and $x.r$, denote the empty tree.* ■

We will identify a node by the key that is stored in it. Since all keys are distinct, this representation is unique.

Definition 2 *Let K be a set of keys, $|K| = n$, and p be an arbitrary permutation of the elements of K. By $\mathfrak{b}_i^p$ we denote the* BST*, which results from the insertion of the first $i \leq n$ keys of p into the empty tree. In an analogous way we define the* AVL*-tree $\mathfrak{a}_i^p$. The notation $\mathfrak{b}_{i..j}^p$ is used to represent the* BST *that results from inserting the keys $p_{i+1}, \ldots, p_j$ into tree $\mathfrak{b}$.* ■

If the context is clear, we skip the superscript p. Note that $(\mathfrak{b}_i^p)_{i..j}^p = \mathfrak{b}_j^p$ holds.

Definition 3 *Let $\mathfrak{b}$ be a* BST *and let x be a node of $\mathfrak{b}$. By $L(\mathfrak{b}, x)$ we denote the tree which results from a left-rotation around the node x. The trees $R(\mathfrak{b}, x)$, $DL(\mathfrak{b}, x)$ and $DR(\mathfrak{b}, x)$ denote the corresponding trees for a right-, a double-left- and a double-right-rotation around the node x, respectively. For $r \in \{L, R, DL, DR\}$ and key x we denote the rotation r around x by the tuple (r, x).* ■

Definition 4 *Let $\mathfrak{b}$ be a* BST *and let x be a node of $\mathfrak{b}$. The* domain *$d_\mathfrak{b}(r, x)$ of a rotation r around x, $r \in \{L, R, DL, DR\}$, is the substructure of $\mathfrak{b}$ of maximal size before the rotation which is changed by the rotation.* ■

Figures 1 and 2 clarify, that $d_\mathfrak{b}(L, x) = x \circ (\emptyset, x.r)$, $d_\mathfrak{b}(R, x) = x \circ (x.l, \emptyset)$, $d_\mathfrak{b}(DL, x) = x \circ (\emptyset, x.r \circ ((x.r).l, \emptyset))$ and $d_\mathfrak{b}(DR, x) = x \circ (x.l \circ (\emptyset, (x.l).r), \emptyset)$. If the context is obvious, we will skip the index in the notation. Figure 4 illustrates how $\mathfrak{b}_i^p$ grows to $(\mathfrak{b}_i^p)_{i..j}^p$ by inserting the keys $p_{i+1}, \ldots, p_j$. The new keys create new subtrees T_k (the frontier trees) which are rooted at the positions of the leaves l_k, $1 \leq k \leq i + 1$.

We will now introduce the most important property of BST's which makes it

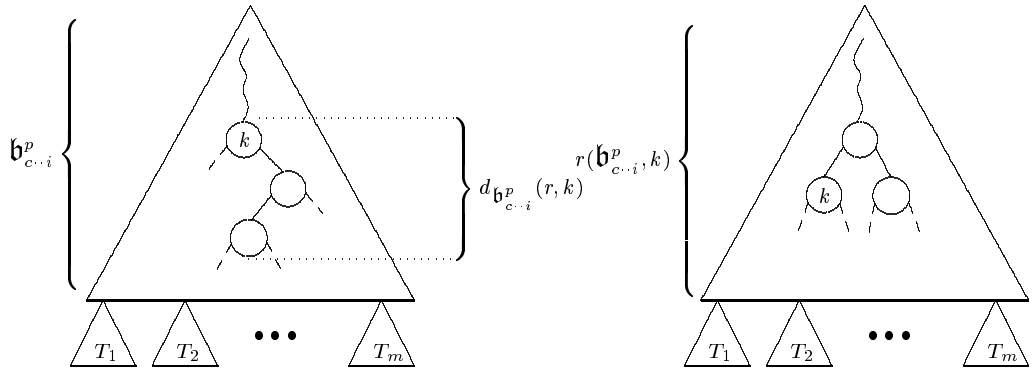


Fig. 5. Tree $\mathfrak{b}^p_{c\cdot\cdot i}$ before the application of rotation r around k (left) and afterwards (right). The domain of r is completely disjoint from the frontier-trees $T_1, \ldots, T_m$.

quite easy to prove the main theorem.

Lemma 5 *Let K be a set of keys, $|K| = n$, $p = p_1, p_2, \ldots, p_n$ be a permutation of the elements of K and $\mathfrak{b}$ be an arbitrary* BST *(possibly empty) that does not store any key of K. Let $(c, i, j) \in \mathbb{N}_0^3$ such that $0 \leq c \leq i \leq j \leq n$. If (r, k), $k \in K$, is a rotation such that $d_{\mathfrak{b}^p_{c\cdot\cdot i}}(r,k) \subseteq \mathfrak{b}^p_{c\cdot\cdot i}$, then*

$$(r(\mathfrak{b}^p_{c\cdot\cdot i}, k))^p_{i\cdot\cdot j} = r(\mathfrak{b}^p_{c\cdot\cdot j}, k).$$

Proof: We have $\mathfrak{b}^p_{c\cdot\cdot i} \cap \mathfrak{b}^p_{c\cdot\cdot j} = \mathfrak{b}^p_{c\cdot\cdot i}$ and $\mathfrak{b}^p_{c\cdot\cdot j} \backslash \mathfrak{b}^p_{c\cdot\cdot i} = \{T_1, T_2, \ldots, T_m\}$ the set of frontier-trees of $\mathfrak{b}^p_{c\cdot\cdot i}$ with respect to the insertion of the keys $p_{i+1}, \ldots, p_j$. The assumption $d_{\mathfrak{b}^p_{c\cdot\cdot i}}(r,k) \subseteq \mathfrak{b}^p_{c\cdot\cdot i}$ implies that $r(\mathfrak{b}^p_{c\cdot\cdot i}, k) \cap r(\mathfrak{b}^p_{c\cdot\cdot j}, k) = r(\mathfrak{b}^p_{c\cdot\cdot i}, k)$ and thus $r(\mathfrak{b}^p_{c\cdot\cdot j}, k) \backslash r(\mathfrak{b}^p_{c\cdot\cdot i}, k) = \{T_1, T_2, \ldots, T_m\}$, i.e.

- the set of frontier-trees $\{T_1, \ldots, T_m\}$, considered as subtrees of $\mathfrak{b}^p_{c\cdot\cdot j}$, remains unchanged,
- restricted to the tree $\mathfrak{b}^p_{c\cdot\cdot i}$, the rotation (r, k) behaves on $\mathfrak{b}^p_{c\cdot\cdot j}$ in the same way as it behaves on $\mathfrak{b}^p_{c\cdot\cdot i}$.

We refer to Figure 5 to get a visualization of those facts. Since the characteristic of any BST is left unchanged by any rotation, we know that $r(\mathfrak{b}^p_{c\cdot\cdot i}, k)$ has the same frontier-trees $\{T_1, \ldots, T_m\}$ with respect to the insertion of the keys $p_{i+1}, \ldots, p_j$ as $\mathfrak{b}^p_{c\cdot\cdot i}$. If we now compare $(r(\mathfrak{b}^p_{c\cdot\cdot i}), k)^p_{i\cdot\cdot j}$ and $r(\mathfrak{b}^p_{c\cdot\cdot j}, k)$ we find that the father of T_ν is the same in both trees and that, if T_ν is a left (resp. right) son of that father in $(r(\mathfrak{b}^p_{c\cdot\cdot i}), k)^p_{i\cdot\cdot j}$, then it is also a left (resp. right) son of its father in $r(\mathfrak{b}^p_{c\cdot\cdot j}, k)$, $1 \leq \nu \leq m$. Otherwise the keys stored in T_ν would not belong to the interval $\mathcal{I}_\nu$ which is a contradiction. This shows the proposition of the lemma since $(\mathfrak{b}^p_{c\cdot\cdot j} \cap \mathfrak{b}^p_{c\cdot\cdot i}) \uplus (\mathfrak{b}^p_{c\cdot\cdot j} \backslash \mathfrak{b}^p_{c\cdot\cdot i}) = \{\mathfrak{b}^p_{c\cdot\cdot j}\}$ and thus we have argued that all parts of $r(\mathfrak{b}^p_{c\cdot\cdot j}, k)$ must coincide with those of $(r(\mathfrak{b}^p_{c\cdot\cdot i}), k)_{i\cdot\cdot j}$. ■

Note, that the condition $d_{\mathfrak{b}^p_{c\cdot\cdot i}} \subseteq \mathfrak{b}^p_{c\cdot\cdot i}$ is always fulfilled if the rotation (r, k) is induced by the AVL-algorithm.

Now, we are able to prove the following theorem:

Theorem 6 *Let $K = \{k_1, ..., k_n\}$ and let $p = p_1, p_2, \ldots, p_n$ be a permutation of the elements of K. Furthermore, let (r_i, k_i), $1 \leq i \leq m$, $r_i \in \{L, R, DL, DR\}$, $k_i \in K$, be the sequence of $m \in \mathbb{N}$, $m < n$, rotations that occur during the insertion of p into an empty* AVL*-tree, together with the corresponding nodes (keys) around which the particular rotations are performed. Then*

$$\mathfrak{a}_n^p = r_m(r_{m-1}(...r_2(r_1(\mathfrak{b}_n^p, k_1), k_2), ..., k_{m-1})k_m).$$

Proof: Let $\{j_1, j_2, \ldots, j_m\}$ be the set of indices such that the insertion of p_{j_i} implies the rotation (r_i, k_i), $1 \leq i \leq m$. Then we have

$$\mathfrak{a}_n^p = (r_m((\ldots(r_2((r_1(\mathfrak{b}_{j_1}^p, k_1))_{j_1 \cdot\cdot j_2}^p, k_2)\ldots)_{j_{m-1} \cdot\cdot j_m}^p), k_m))_{j_m \cdot\cdot n}^p, \qquad (1)$$

since the right-hand side of (1) is nothing else but a formal description of the insertion-process for the AVL-tree $\mathfrak{a}_n^p$. Now the application of Lemma 5 yields $(r_1(\mathfrak{b}_{j_1}^p, k_1))_{j_1 \cdot\cdot j_2}^p = (r_1(\mathfrak{b}_{j_2}^p, k_1))$ and thus $r_2((r_1(\mathfrak{b}_{j_1}^p, k_1))_{j_1 \cdot\cdot j_2}^p, k_2) = r_2(r_1(\mathfrak{b}_{j_2}^p, k_1), k_2)$. Applying this identity to (1) yields

$$\mathfrak{a}_n^p = (r_m((\ldots r_3((r_2(r_1(\mathfrak{b}_{j_2}^p, k_1), k_2))_{j_2 \cdot\cdot j_3}^p, k_3)\ldots)_{j_{m-1} \cdot\cdot j_m}^p), k_m))_{j_m \cdot\cdot n}^p.$$

Now we may apply Lemma 5 to the BST $r_2(r_1(\mathfrak{b}_{j_2}^p, k_1), k_2))_{j_2 \cdot\cdot j_3}^p$ in order to get $r_2((r_1(\mathfrak{b}_{j_2}^p, k_1))_{j_2 \cdot\cdot j_3}^p, k_2)$ and by a second application $r_2(r_1(\mathfrak{b}_{j_3}^p, k_1), k_2)$. Thus $r_3((r_2(r_1(\mathfrak{b}_{j_2}^p, k_1), k_2))_{j_2 \cdot\cdot j_3}^p, k_3) = r_3(r_2(r_1(\mathfrak{b}_{j_3}^p, k_1), k_2), k_3)$ holds. Again, this identity can by applied to equation (1). The iteration of the same arguments finally proves the theorem. ■

4 Consequences to Applications

Under certain assumptions, the result of section 3 can be used in an applied way. One example will be presented in the rest of this note.

If, in practice, a large set of data has to be handled efficiently it is a well known heuristic to proceed in the following way: During an insertion phase the data is used to build up a BST in the normal way. Afterwards, to improve the quality of the resulting tree a rebalancing run is performed. This could be implemented by a postorder-traversal of the tree where nodes are rebalanced whenever necesarry using AVL-rotations. The resulting tree T_h is height balanced but since the order and the type of the rotations performed is not related to the order and the type used by the AVL-algorithm, to the knowledge of the author it is not known if T_h is the same tree as if the AVL-algorithm would

have been applied. However, this is an important question since the worst-case height of a height balanced tree and the average height of an AVL-tree differ by roughly 40 percent [BYG91]. The reason for that heuristic is the belief that there is a speed-up of the insertion-phase, because no rotation is performed, and also that the total number of rotations is reduced because of *automatic rebalancation* during the insertion phase, i.e. subtrees out of balance get rebalanced by subsequent insertions.
Under certain assumptions, it is possible to base the idea of that heuristic on theoretical knowledge using the result of section 3. To do that we introduce a queue $\mathcal{Q}$ to store tuples (k, r) where k denotes a pointer to the node storing key k and r represents a rotation type. Since nodes do not change their location in the memory, those pointers are valid for the lifetime of the tree. The keys are inserted into the tree using the normal AVL-algorithm with the restriction that any rotation, that is needed, is not performed explicitly, but is stored in $\mathcal{Q}$. We assume to know how to decide the rotation-types and -places efficiently without generating the AVL-tree. After the tree has been built we use the information stored in $\mathcal{Q}$ to generate the AVL-tree according to Theorem 6. In that way we get exactly the same tree as if we had used the normal AVL-algorithm. Note that it is not possible to decide the rotation-types and -places by just using the normal marking of the nodes which keeps track of the nodes balance-degree and update it as if the rotations have been performed.
We can optimize the rebalancing-run by observing that some combinations of rotations leave a tree unchanged. For example, consider the application of a left-rotation around x on a tree T. If we apply afterwards a right-rotation around $x.r$, we get tree T again. There are other combinations of rotations with the same property and it is an interesting task to find any characterization of the set of all such combinations. If we store the information additionally needed to detect such combinations for adjacent entries stored in $\mathcal{Q}$ (in case of the previous example we would need the knowledge of the right son of x) it is possible to reduce the number of rotations needed. In a multiprocessor or distributed environment it is possible to perform this *preprocessing* in parallel to the generation of the search tree. It remains an open problem of how many rotations can be saved that way.

References

[AdVL62] Adel'son-Vel'skii, G.M., Landis, E.M.: *An Algorithm for the Organization of Information.* Dokladi Akademia Nauk SSSR, 146 (2), 263-266, 1962.

[BYG91] Baeza-Yates, R., Gonnet, G.H.: *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, Mass., 1991.

[BYGZ90] Baeza-Yates, R., Gonnet, G.H., Ziviani, N.: *Expected Behaviour Analysis of AVL-Trees.*
Proceedings Scandinavian Workshop in Algorithmic Theory, SWAT '90, LNCS 447, Springer-Verlag, Bergen, Norway, 2:143-159, July 1990.

[Dev86] Devroye, L.: *A Note on the Height of Binary Search Trees.* Journal of the ACM, Vol. **33**, 489-498.

[Knu73] D. E. Knuth: The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, 1973