Annales Mathematicae et Informaticae
44 (2015) pp. 5–13
http://ami.ektf.hu

Representation transformations of ordered lists*

Tibor Ásványi

Eötvös Loránd University, Faculty of Informatics
Budapest, Hungary
asvanyi@inf.elte.hu

Submitted September 15, 2014 — Accepted March 5, 2015

Abstract

Search and update operations of dictionaries have been well studied, due to their practical significance. There are many different representations of them, and some applications prefer this, the others that representation. A main point is the size of the dictionary: for a small one a sorted array can be the best representation, while for a bigger one an AVL tree or a red-black tree might be the optimal choice (depending on the necessary operations and their frequencies), and for an extra large one we may prefer a B+-tree, for example.

Consequently it can be desirable to transform such a collection of data from one representation into another, efficiently. There is a common feature of the data structures mentioned: they can be considered strictly ordered lists. Thus in this paper we start a new topic of interest: *How to transform a strictly ordered list form one representation into another, efficiently?* What about the time and space complexities of such transformations?

Keywords: strictly increasing list, representation-transformation, data structure (DS), linear, array, binary tree (BT), balanced, search tree

MSC: 68P05, 68P10, 68P20, 68Q25

1. Introduction

In this paper we consider strictly increasing lists. They can be represented in several different ways. For example, with a *linear data structure (LDS)* (e.g. array,

*Supported by Eötvös Loránd University, Faculty of Informatics.

linked list, sequential file), with a *binary search tree (BST)* (e.g. unbalanced BST, AVL tree, red-black tree), with a *B-tree*, *B+-tree*, etc. [1, 2, 3].

Their common features are that they can be traversed increasingly in $\Theta(n)$ time: the linear traversal of a LDS has linear operational complexity; similarly, the inorder traversal of a tree needs $\Theta(n)$ time. And the search-and-update operations can run in $O(n)$ time. [1, 2]

In this paper we use three asymptotic computational complexity measures (each time we consider the worst case by default): $O(g(n))$ (upper bound), $\Omega(g(n))$ (lower bound), and $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ [2].

Sorted arrays support only the *search* with $O(\log(n))$ operational complexity, but for the *balanced search tree*[1] representations each of the *search, insert, delete* operations have this complexity. On linked lists, and sequential files we cannot perform any search-and-update operation in $O(\log(n))$ time. Thus we concentrate on the *sorted array*, and *balanced search tree* representations of such lists. The *search, insert, delete* operations have been well studied. Sometimes we have to *transform these lists* from one representation into another. Consequently we pay attention to these representation-transformations. We ask, how to transform a strictly ordered list L from one representation into another, *efficiently*?

Undeniably, the operational complexity of such a transformation is $\Omega(n)$: each item must be processed. In some cases, it is also $O(n)$: Undoubtedly, a linear representation of L can be produced in $O(n)$ time, because the input representation of L can be traversed also in $O(n)$ time. Thus, regardless of the input representation of L, a linear representation of L can be generated with $\Theta(n)$ atomic operations. Besides, a balanced search tree representation of L can be generated in $O(n\log(n))$ time, because a single insert needs $O(\log(n))$ atomic operations. Nevertheless this method does not use the information that the input is *sorted.*

Consequently this is our question: Given an input representation of L, when and how can we produce a balanced search tree representation of it, with an operational complexity $\Theta(n)$, or at least better than $\Theta(n\log(n))$? We give a partial answer to this question. We invent three algorithms. With operational complexity $\Theta(n)$, we transform (1) a *strictly increasing array* into an *AVL* tree; (2) a *strictly increasing array* into a *red-black* tree; (3) an *AVL* tree into a *red-black* tree.

2. Main results

In order to expound these algorithms (a) we define *size-balanced* BSTs, and an algorithm transforming a *strictly increasing array* into such a size-balanced BST; (b) we prove that a size-balanced BST is almost complete, and so (c) it is an *AVL* tree; (d) we colour the almost complete BSTs as *red-black* trees; (e) we find a special property of *AVL* trees, and invent an algorithm colouring them as *red-black* trees.

(a-c) are needed for transforming a *strictly increasing array* into an *AVL* tree (Section 2.1). (a,b,d) result in the transformation of a *strictly increasing array* into

[1] AVL tree, red-black tree, SBB-tree, rank-balanced tree, B-tree, B+-tree, etc.

a *red-black* tree (Section 2.2). The theorems and algorithm of (e) in Subsection 2.3 form the high point of this section.

2.1. Strictly increasing array to AVL tree

First we enumerate the necessary notions. By trees we mean rooted ordered trees [2]. Remember that NIL is the empty tree. The leaves of a nonempty tree have no child. The non-leaves are the internal nodes.

If $t \neq NIL$ is a binary tree (BT), $\text{left}(t)$ is its left and $\text{right}(t)$ is its right subtree.

If t is a BT, $s(t)$ is its size, i.e. $s(t) = 0$, if $t = NIL$; $s(t) = 1 + s(\text{left}(t)) + s(\text{right}(t))$, otherwise. $h(t)$ is its height, i.e. $h(t) = -1$, if $t = NIL$; $h(t) = 1 + \max(h(\text{left}(t)), h(\text{right}(t)))$, otherwise.

If r is the root node of a BT $t \neq NIL$, $\text{left}(r) = \text{left}(t)$, $\text{right}(r) = \text{right}(t)$, and $\text{root}(t) = r$. Provided that t is a BT, $n \in t$, *iff* $t \neq NIL \wedge (n = \text{root}(t) \vee n \in \text{left}(t) \vee n \in \text{right}(t))$.

$d_t(n)$ is the depth of node n in BT t. If $t \neq NIL$, $d_t(\text{root}(t)) = 0$. If n is a node of a BT t and $\text{left}(n) \neq NIL$, $d_t(\text{root}(\text{left}(n))) = d_t(n) + 1$. If $\text{right}(n) \neq NIL$, $d_t(\text{root}(\text{right}(n))) = d_t(n) + 1$. Node n is *strictly binary* $(SB(n))$, *iff* $\text{left}(n) \neq NIL \wedge \text{right}(n) \neq NIL$.

Clearly, $h(t) = \max\{d_t(n) \mid n \in t\}$, if $t \neq NIL$. A BT t is complete, *iff* $(\forall n \in t)(d_t(n) < h(t) \rightarrow SB(n))$.

Notice that for any leaf n of a complete BT t, $d(n) = h(t)$; and $s(t) = 2^{h(t)+1} - 1$. A BT t is *almost complete* $(AC(t))$, *iff* $(\forall n \in t)(d_t(n) < h(t) - 1 \rightarrow SB(n))$.

Notice that a BT is AC, *iff* compared to the appropriate complete BT, nodes may be missing only from its lowest level: Figure 1 shows such a tree. Clearly, for a leaf n of an AC BT t, $d_t(n) \in \{h(t), h(t) - 1\}$. The nodes of t at depth $h(t) - 1$ may have one or two children, or may be leaves. $s(t) \in [2^{h(t)}, 2^{h(t)+1} - 1]$.

```
                 ------------7----------
                /                       \
         -----4-----                 ---10----
        /           \               /         \
       2             5             9           12
      / \           / \           / \          / \
     1   3       NIL   NIL       8   NIL     11   NIL
```

Figure 1: almost complete BST: the places of the missing nodes are shown by NILs

A node n of a BT is height-balanced, *iff* $|h(\text{right}(n)) - h(\text{left}(n))| \leq 1$. A BT t is height-balanced, *iff* $(\forall n \in t)$, n is height-balanced. An AVL tree is a height-balanced BST.

A node n of a BT is size-balanced, *iff* $|s(\text{right}(n)) - s(\text{left}(n))| \leq 1$. A BT t is size-balanced, *iff* $(\forall n \in t)$, n is size-balanced.

We transform a strictly increasing array into an equivalent size-balanced BST in linear time:

We take the middle item of a nonempty array. This will label the root node of the tree. Next we transform the left and right sub-arrays into the appropriate subtrees, recursively. An empty array is transformed into an empty tree.

The resulting size-balanced BST is also an AVL tree, as it follows from the next two theorems.

Theorem 2.1. *A size-balanced binary tree is also almost complete (AC).*

Proof. We use mathematical induction with respect to the height $h(t)$ of the size-balanced tree t. If $h(t) = -1$, then t is empty, and $AC(t)$. Let us suppose that we have this property for trees with $h(t) \leq h$. Let $h(t) = h + 1$. Then $t \neq NIL \wedge h(\text{left}(t)) \leq h \wedge h(\text{right}(t)) \leq h$. It follows by induction that left(t) and right(t) are almost complete. Also $|\,\text{size}(\text{left}(t)) - \text{size}(\text{right}(t))| \leq 1$. (Furthermore, remember that a complete binary tree of height h has the size $2^{h+1} - 1$, and an almost complete binary tree with size in $[2^{h+1}, 2^{h+2} - 1]$ has height $h+1$.) Now we enumerate the possible cases about the subtrees of t, and prove that $AC(t)$ in each case. If the two (almost complete) subtrees have the same size, their heights are also equal, and $AC(t)$. If the smaller subtree is complete, then the bigger one has an extra leaf at its extra level, and $AC(t)$. If the smaller subtree is not complete, then the bigger one has the same height, and $AC(t)$. □

Theorem 2.2. *An almost complete binary tree is also height-balanced.*

Proof. We can suppose $t \neq NIL$. First, if $AC(t)$, the leaves of t have depth $h(t)$ or $h(t) - 1$. Thus $h(\text{left}(t)), h(\text{right}(t)) \in \{h(t) - 1, h(t) - 2\}$. Consequently, $|h(\text{right}(t)) - h(\text{left}(t))| \leq 1$. As a result, root$(t)$ is balanced. Next, let us suppose that $lr(t) \in \{\text{left}(t), \text{right}(t)\}$. Now, if $AC(t)$, then $(\forall n \in lr(t))(d_t(n) < h(t) - 1 \rightarrow SB(n))$. Therefore $(\forall n \in lr(t))(d_{lr(t)}(n) < h(t) - 2 \rightarrow SB(n))$. We also have $h(lr(t)) \leq h(t) - 1$. For these reasons $(\forall n \in lr(t))(d_{lr(t)}(n) < h(lr(t)) - 1 \rightarrow SB(n))$. As a result, $AC(lr(t))$. Thus each (direct or indirect) subtree of t is AC, and if a subtree is nonempty, its root node is balanced. Finally, each node of t is balanced. □

Corollary 2.3. *A size-balanced BST is also an AVL tree.*

Proof. A size-balanced BST is almost complete, thus height-balanced. □

Consequently, the algorithm we defined above transforms a strictly increasing array into an equivalent AVL tree. It takes $\Theta(n)$ time, because each item of the array is processed once. Besides the $\Theta(n)$ size of its output, it needs $\Theta(\log(n))$ working memory: this is the height of the recursion. Provided that we need the heights of the nonempty subtrees in their root nodes (as it is usual with AVL trees), we can return the height of a subtree when we return from the appropriate recursive call, and compute the height of a subtree with a given root node from the heights of the two subtrees of that node.

2.2. Strictly increasing array to red-black tree

We have an algorithm transforming a strictly increasing array into an almost complete BST. We also have the height of the tree. Here we need an additional flag showing whether the tree is complete or not. Clearly, a nonempty BT is complete, *iff* its too subtrees are also complete, and their heights are the same. Thus the computation of this flag is also easily merged into the algorithm above.

Next, if we prove that an almost complete BST (with its height and flag) can be coloured in linear time, as a red-black tree, then we have also the algorithm transforming a strictly increasing array into such a tree.

Definition 2.4. A red-black tree is a BST with red and black nodes: The root node is black. We regard NILs as pointers to black, external leaves. For each node, all simple paths from the node to descendant NIL-leaves contain the same number of black nodes. If a node is red, then both its children are black. [2] (See Figure 2.)

```
              ---------BLACK---------
             /                       \
      ----red----                 --BLACK--
     /           \                /        \
  BLACK         BLACK          red         NIL
  /   \         /   \         /   \
NIL   NIL     NIL  NIL      NIL   NIL
```

Figure 2: Red-black tree

Based on this definition, the algorithm of *colouring* is simple: consider the complete levels of an almost complete BST, and paint the nodes black. If the tree is not complete, the nodes at the lowest, partially filled level remain, and we paint them red. (See Figure 3.) Unquestionably this procedure needs $\Theta(n)$ time and $\Theta(\log(n))$ working memory. The algorithm computing its input (the almost complete tree, its height, and flag) has the same measures. Consequently, the whole transformation has these time and space requirements.

```
             ----------BLACK---------
            /                        \
      ---BLACK---                 --BLACK--
     /           \                /        \
   red           red           red         NIL
  /   \         /   \         /   \
NIL   NIL     NIL  NIL      NIL   NIL
```

Figure 3: Almost complete tree painted as red-black tree

2.3. AVL tree to red-black tree

We colour an AVL tree t as a red-black tree. We use a postorder and a preorder traversal. As a result, our procedure needs $\Theta(n)$ time and $\Theta(\log(n))$ (proportional to the height of t) working memory [1].

Definition 2.5. Minimal height of a binary tree t: $m(t) = -1$, if $t = NIL$; $1 + \min(m(\text{left}(t)), m(\text{right}(t)))$, otherwise.

Theorem 2.6. *If t is a height-balanced tree then $m(t) \le h(t) \le 2m(t) + 1$.*

Proof. It comes with mathematical induction with respect to $m(t)$. If $m(t) = -1 \Rightarrow t = NIL \Rightarrow h(t) = -1 \Rightarrow m(t) \le h(t) \le 2m(t) + 1$. Let us suppose that we have this property for trees with $m(t) = k$. Let $m(t) = k + 1$. We can suppose that $m(\text{left}(t)) = k$. By induction: $k \le h(\text{left}(t)) \le 2k+1$. The tree t is height-balanced. Therefore $h(\text{right}(t)) \le 2k + 2$. Thus $k + 1 \le 1 + \max(h(\text{left}(t)), h(\text{right}(t))) = h(t) \le 2k + 3 = 2(k + 1) + 1$. As a result: $m(t) \le h(t) \le 2m(t) + 1$. (Notice that $m(t) \le h(t)$ for any binary tree.) □

In the *colouring* algorithm, first we calculate $m(t)$. Based on the definition, this can be done with a postorder traversal of t. In a typical AVL tree, for each non-NIL subtree s of t, the $h(s)$ attributes are already present. (If not, the computation of the $h(s)$ values can be easily merged into the postorder traversal.)

Next, with a preorder traversal of t, we *colour* t. (See Figure 4). We paint $m(t) + 1$ nodes black on each simple path from the root to a NIL-leaf. (The NIL-leaves are also considered black, but we do not paint them.)

```
PreCondition of the first call:
I0: t is AVL tree and b = m(t)+1 and
    h(s) is calculated for each subtree s of t

procedure colour( t : BinTree; b : integer )
    /* I1: b>=0 and b-1 =< m(t) and h(t) =< 2*b */
    if( t \= NIL ) {
    /* Note: paint b nodes black on each branch of t */
        if( h(t) < 2*b ) {
            colour(t) := black
            b := b-1 }
        else /* I2: 0 =< b =< m(t) and h(t) = 2*b */
            colour(t) := red
        colour(left(t),b)
        colour(right(t),b) }
end of procedure colour
```

Figure 4: Colouring an AVL tree `t` as a red-black tree

Now we are going to prove the correctness of the *colouring* algorithm in Figure 4.

Terminology: In the rest of this section we use $I0$ (i.e. the PreCondition), invariants $I1$, $I2$, and other logical statements. Let us suppose that $Ij, Ik \in \{I0, I1, I2\}$; P, Q are arbitrary statements. When we say that *Ij with P induces Ik with Q*, we mean: If *Ij and P* are true when the program is at the place of Ij, then *Ik and Q* will hold when the run of the program next time arrives at the place of Ik.

Lemma 2.7. *$I0$ induces $I1$ with $h(t) < 2b$.*

Proof. Based on the definition of $m(t)$, $m(t) \geq -1$. Consequently $b = m(t) + 1$ implies $b \geq 0 \wedge b - 1 \leq m(t)$. Theorem 2.6 implies $h(t) \leq 2m(t) + 1$. Considering $b = m(t)+1$ we receive $h(t) \leq 2m(t)+1 = 2(b-1)+1 = 2b-1$. Thus $h(t) < 2b$. □

Lemma 2.8. *$I1$ with $t \neq NIL \wedge h(t) < 2b$ induces $I1$ in both recursive calls.*

Proof. Let $s(t)$ be the left or right subtree parameterizing the appropriate recursive call. Thus we need to prove $I1^{b \leftarrow b-1, t \leftarrow s(t)}$ i.e. that the following three conditions hold:

(1) $b - 1 \geq 0$: We know that $h(t) \geq 0$ (since $t \neq NIL$) and $h(t) < 2b$. Consequently, $b > 0$, and therefore $b - 1 \geq 0$.

(2) $b - 2 \leq m(s(t))$: From $I1$, $b - 1 \leq m(t)$. From the definition of $m(t)$, $m(t) \leq 1 + m(s(t))$. As a result, $b - 1 \leq 1 + m(s(t))$, i.e. $b - 2 \leq m(s(t))$.

(3) $h(s(t)) \leq 2(b - 1)$: $h(t) < 2b$, i.e. $h(t) \leq 2b - 1$; $h(s(t)) \leq h(t) - 1$; thus $h(s(t)) \leq 2b - 2$. □

Lemma 2.9. *$I1$ with $t \neq NIL \wedge h(t) \geq 2b$ induces $I2$.*

Proof. $h(t) \leq 2b$ and $h(t) \geq 2b$ implies $h(t) = 2b$. $0 \leq b$ remains true. Considering Theorem 2.6 we have $2b \leq 2m(t) + 1$; thus $b \leq m(t) + 1/2$ i.e. $b \leq m(t)$. □

Lemma 2.10. *$I2$ induces $I1$ with $h(t) < 2b$ in both recursive calls.*

Proof. Let $s(t)$ be the left or right subtree parameterizing the appropriate recursive call. Thus we have to prove $(I1 \wedge h(t) < 2b)^{t \leftarrow s(t)}$ i.e. $b \geq 0 \wedge b - 1 \leq m(s(t)) \wedge h(s(t)) < 2b$. $b \geq 0$ remains true. From $b \leq m(t)$ and $m(t) \leq 1 + m(s(t))$ we have $b - 1 \leq m(s(t))$. $h(t) = 2b$ implies $h(s(t)) < 2b$. □

Theorem 2.11. *Provided that the precondition $I0$ holds, procedure colour paints the nodes of tree t so that t becomes a red-black tree.*

Proof. Lemmas 2.7, 2.8, 2.9, and 2.10 imply that $I1$ and $I2$ are invariants of the program. $I1$ means that when we arrive at an external leaf, i.e. $t = NIL$, $0 \leq b \leq m(t) + 1 = -1 + 1$, as a result $b = 0$. In the program b is decreased (by 1), exactly when a node is painted black. Because b is decreased to zero on each branch of any subtree while we go to a NIL-leaf, we have the same number of black nodes on these paths. Lemma 2.7 implies that the root node of the tree is painted black. Lemma 2.10 makes sure that both children of a red node will also be black. These have the effect of receiving a red-black tree. □

Let a *crb-tree* be a *BST* which can be coloured as a *red-black tree*. Then an AVL tree is also a crb-tree. This also follows from both of the following results.

(1) Bayer proved that the class of *SBB-trees* properly contains the AVL trees [4], and we know from 4.7.2 in [3] that the SBB-trees and the red-black trees are structurally equivalent.

(2) Rank-balanced trees are a relaxation of AVL trees, and form a proper subclass of crb-trees [5].

Our achievements and these results are unrelated. In this subsection our contributions are the notion of the *minimal height of an AVL tree*, theorems 2.6 and 2.11, and our *efficient colouring algorithm* proved.

3. Conclusions

This was our question: How to transform a *strictly increasing list L* from one representation into another, *efficiently*?

Summarizing this paper, we already know that given an input representation of L, we can produce another representation of it in $\Theta(n)$ time, if this other representation is a linear data structure, an *AVL* or *red-black* tree. In some cases we have direct transformations, in other cases we need a temporary array.

In three cases we invented the necessary algorithms, theorems, lemmas, and proofs. The first two, (*sorted array $\rightarrow$ balanced BST*) programs create new trees; but the second half of the second algorithm, and the third (*AVL* tree $\rightarrow$ *red-black* tree) procedure do not make structural changes on the actual tree, just paint its nodes *black*, and *red*. Each of the three programs needs $\Theta(\log(n))$ working memory.

Our algorithms and theorems imply three relations among four classes of BSTs: *size-balanced BSTs* $\subset$ *almost complete BSTs* $\subset$ *AVL trees* $\subset$ *crb-trees*. Actually, each of the first three classes is a proper subclass of the next one. For example[2] BST (((1)2(3))4(5)) is almost complete, but not size-balanced; AVL tree ((((1)2)3(4))5(6(7))) is not almost complete; red-black tree $((1b)2b((3b)4r(5b(6r))))$ is not height-balanced.

Open questions: If L is transformed into another type of balanced search trees (*not* into an *AVL* or *red-black* tree); for example, into a B-tree, we know that the operational complexity of the transformation is $\Omega(n)$, and $O(n\log(n))$. Here we still need more sharp results. Maybe, from a strictly increasing list, each kind of balanced search trees can be generated in $\Theta(n)$ time? Are there some cases, when the time complexity is more than $\Theta(n)$, but less than $\Theta(n\log(n))$?

If the input representation of L is a search tree (or a linked list or a sequential file), and the output is an *AVL* or *red-black* tree, we can make the transformation in $\Theta(n)$ time, but – with the exception of the *AVL tree* $\rightarrow$ *red-black tree* program

[2]Using the notation *(left-subtree root right-subtree)* where the empty subtrees are omitted.

– we actually need a temporary array, thus $\Theta(n)$ working space. We ask: In which cases can we reduce the memory needed?

References

[1] WEISS, MARK ALLEN, Data Structures and Algorithm Analysis, *Addison-Wesley*, 1995, 1997, 2007, 2012, 2013.

[2] CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L., STEIN, C., Introduction to Algorithms, *The MIT Press*, 2009. (Ebook: `http://bit.ly/IntToAlgPDFFree`)

[3] WIRTH, N., Algorithms and Data Structures, *Prentice-Hall Inc.*, 1976, 1985, 2004. (Ebook: `http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf`)

[4] BAYER R., Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms, *Acta Informatica 1*, 290–306 (1972), *Springer-Verlag*, 1972.

[5] HAEUPLER B., SEN S., TARJAN R.E., Rank-Balanced Trees, *Algorithms and Data Structures: 11th International Symposium, WADS 2009, Banff, Canada, August 2009*, pp 351–362, *Springer-Verlag, LNCS 5664*, 2009.