BIT 27 (1987), 330–339

ON TOP-DOWN SPLAYING *

ERKKI MÄKINEN

University of Tampere, Department of Computer Science, P.O. Box 607, SF-33101 Tampere, Finland

Abstract.

The splay tree is a self-adjusting binary search tree which has a good amortized performance. This paper studies some properties of top-down splay trees. Different ways to charge for the primitive operations of top-down splaying are discussed. We also give some empirical results concerning the behaviour of different top-down restructuring algorithms.

CR Categories: E.1, F.2.2.

General Terms: Algorithms.

Additional Keywords and Phrases: binary search trees, rotation, self-adjusting data structures, amortized complexity.

1. Introduction.

Let U be a totally ordered universe. The *dictionary problem* is that of maintaining one or more subsets of U under the following operations: *access*, *insert*, *delete*, *join*, and *split*. Several data structures with the worst-case time bound of $O(\log n)$ per dictionary operation are given in the literature. Reducing the worst-case time per dictionary operation has been the ultimate goal in most of these data structures. A different view was introduced in [3], where a self-adjusting binary search tree, called *splay tree*, was presented. This data structure uses *splaying* as a restructuring heuristic, which is applied during each dictionary operation. Splaying moves the accessed item to the root of the tree and roughly halves the depth of every node along the access path. It is then hoped that frequently accessed items are often near to the root of the tree. As a consequence, splay trees are efficient in an *amortized* sense, i.e. when the average time per operation over a worst-case sequence of operations is concerned (see [5] for further details concerning amortization).

The purpose of this paper is to study some properties of splay trees. In chapter 2 we introduce splaying methods from [3]. In chapter 3 we compare

Received July 1986. Revised June 1987.

*) This work was supported by the Academy of Finland.

the basic splaying methods, *bottom-up* and *top-down splaying*. In chapter 4 we elaborate the amortized analysis of top-down splaying. Especially, we study how to charge for the primitive operations of the top-down splaying algorithm when determining the amortized time. In chapter 5 we give some empirical results concerning the behaviour of different top-down restructuring methods. We shall see that there are simpler methods which often perform better than top-down splaying. However, there are access sequences on which these simpler methods are poor, i.e. they are not efficient in an amortized sense.

2. Top-down splaying algorithm.

We can start the splaying process either from the root (top-down splaying) or from the node to be accessed (bottom-up splaying). Both methods handle two nodes at a time and make rotations if the two nodes are both left (resp. right) children. Complete descriptions for the algorithms can be found in [3].

The main disadvantage of bottom-up splaying is that it requires the ability to get from a node to its parent. Different ways to implement this are discussed in [3].

On the other hand, the top-down algorithm performs the restructuring operations while going down from the root. During the access and concurrent splaying the tree is divided into three trees: a *left tree*, a *right tree*, and a *middle tree*. The passed nodes are joined to the bottom right of the left tree or to the bottom left of the right tree, depending on whether they contain items smaller or greater than the search item. Joining is executed by a restructuring primitive called *link*, which will be discussed in greater detail later on.

When the node x containing the accessed item is reached, we complete top-down splaying by making x to be the new root, the root of the left (resp. right) tree to be the left (resp. right) child of the root and by joining the left (resp. right) child of x to be the right (resp. left) bottom of the left (resp. right) tree.

Sleator and Tarjan [3] have sketched the implementation of top-down splaying by using a version of Dijkstra's guarded command language. We shall now do the same for those parts of the algorithm which are essential in the sequel by using Pascal.

The nodes are supposed to have the following type (itemtype is not specified):

```
type node = record item: itemtype;
                   left, righ: ↑node
            end;
```

We need five variables of type ↑node to point to the current node (t), the root of the left tree ($l$tree), the root of the right tree ($r$tree), the last node in

symmetric order in the left tree (*r*mostin*l*tree), and the first node in symmetric order in the right tree (*l*mostin*r*tree).

Left rotation makes the right child of the root to be the new root, and the root to be the left child of the new root. This can be done by a Pascal procedure having four assignment statements and one local variable as follows:

```
procedure rotate_left;
  var lchild: ↑node;
  begin
    lchild: = t;
    t: = t↑.right;
    lchild↑.right: = t↑.left;
    t↑.left: = lchild
  end;
```

Right rotation can be defind analogously.

The link operation needs one assignment statement less than rotation:

```
procedure link_left;
  begin
    rmostinltree↑.right: = t;
    rmostinltree: = t;
    t: = t↑.right
  end;
```

During the first execution of link_left it is assumed that the left tree is initialized to contain a dummy node to which *r*mostin*l*tree points. This node is neglected when the resulting tree is composed.

Simple top-down splaying is defined to be a method which differs from the above described top-down splaying so that two nodes are handled in a splaying step only when they both are left or right children. Otherwise, we use one of the link primitives to join the first node to the left or the right tree [3].

The overall purpose of splaying is to reduce the time spent in a sequence of dictionary operations by restructuring the tree so that frequently accessed items are near the root. This is done by (1) moving the accessed item to the root and (2) making rotations. A top-down restructuring method using (1) is introduced already in [4]. We define *Stephenson's variant* to be the one where no rotations are made but instead the nodes are always directly joined to the left and right trees by using the link primitives. Unfortunately, for Stephenson's variant there are arbitrarily long access sequences such that the time per access is $O(n)$ [1]. Notice that Stephenson's variant is a top-down version of the move-to-root heuristic of [1].

3. Top-down VS. Bottom-up splaying.

Let T be a splay tree and let x be a node in T. We denote the resulting tree of top-down (resp. bottom-up) splaying in T at x by $td(x, T)$ (resp. $bu(x, T)$). There are trees for which $td(x, T)$ and $bu(x, T)$ do not coincide. The purpose of this chapter is to characterize such trees.

We need some notations and definitions. The *access path* of a node x in a tree with root t is a sequence of nodes $t = x_0, x_1, \ldots, x_n = x$, where each x_i, $i = 0, \ldots, n-1$, is the parent of x_{i+1}. Then the *depth* of x is n. The *direction path* of a node x having an access path $x_0, x_1, \ldots, x_n$ is the sequence $k_1 k_2 \ldots k_n$, where for each $i = 1, \ldots, n$

$$k_i = \begin{cases} l, \text{ if } x_i \text{ is the left child of } x_{i-1} \\ r, \text{ if } x_i \text{ is the right child of } x_{i-1}. \end{cases}$$

The *zig-zig step* of the bottom-up algorithm is the one where we make a rotation. Similarly, the *zig-zag step* handles a left child and a right child without performing a rotation. The *zig step* is used as a terminal step when the access path is of odd length.

The following theorem gives a sufficient condition for the equality of $td(x, T)$ and $bu(x, T)$.

THEOREM 1. *Let T be a tree and let x be a node of even depth in T. Then the trees $td(x, T)$ and $bu(x, T)$ coincide.*

PROOF. The proof is an induction on the length of the access path of x. If the access path has length 2, then it is straight-forward to verify that $td(x, T)$ and $bu(x, T)$ are equal.

Suppose now that the theorem holds for all paths of even length at most $2i$, $i > 0$, and let a node x be of depth $2i+2$. The direction path of x ends with *rl*, *ll*, *lr*, or *rr*. We consider the case *rr* in detail; the other cases are similar. We have the situation described in figure 1.

Let T' denote the entire tree after the first zig-zig step of the bottom-up algorithm. In T' the subtree of figure 1 has the form described in figure 2. On the other hand, the top-down algorithm ends with a left rotation and a left link, which join the subtree of figure 3 to the left tree. This follows from the assumption that the access path is of even length. The subtree C will be joined to the last node in symmetric order in the left tree, i.e. to y. Similarly, the subtree D will be joined to the right tree. Hence, we have $td(x, T) = td(x, T')$. By the induction hypothesis, we have $td(x, T') = bu(x, T')$ and the theorem follows. ■

Similarly, we can characterize the case where $td(x, T)$ differs from $bu(x, T)$.

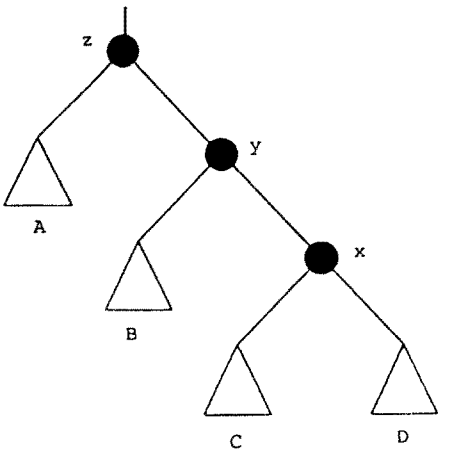


Fig. 1. (Note that it does not matter whether z is the left or the right child of its parent.)

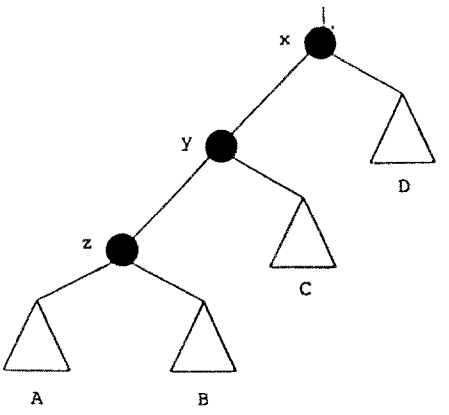


Fig. 2.

If w is a string of characters, we use the notation w^+ to stand for any string obtained by concatenating w, e.g. w, ww, $www\ldots$

THEOREM 2. *Let T be a tree and let x be a node in T of odd depth i, $i \geq 3$. Then $td(x, T)$ differs from $bu(x, T)$ excepting the case where the direction path of x has the form $l(rl)^+$ or the form $r(lr)^+$.*

PROOF. Suppose that the direction path of x does not have the form $l(rl)^+$ or the form $r(lr)^+$. Let $y_0, y_1, \ldots, y_{2n}, y_{2n+1} = x$ be the access path of x, and let y_{i+1} and y_i be the first nodes from the end of the path having the property that both y_{i+1} and y_i are left (resp. right) children. We suppose that y_{i+1} and y_i are left children; the other case is similar. We have the situation of figure 4.

Two cases appear depending on whether or not y_{i+1} is of even depth. Suppose first that it is. This means that when applying the bottom-up algorithm, zig-zag steps transform x to be the right child of y_{i+1}. An additional zig-zag step then transforms the tree to the form described in figure 5. All the three possibilities to continue the bottom-up splaying give a tree where y_{i-1} is the parent of y_i.

Since y_{i+1} is of even depth in T, we do a right rotation and a right link in y_{i-1} when applying the top-down algorithm. The left tree and the right tree have the forms described in figure 6. Hence, in the tree obtained by the bottom-up algorithm, y_{i-1} is the parent of y_i, but in the tree obtained by the top-down algorithm, y_i is the parent of y_{i-1}. This proves that the trees $td(x, T)$ and $bu(x, T)$ cannot coincide.

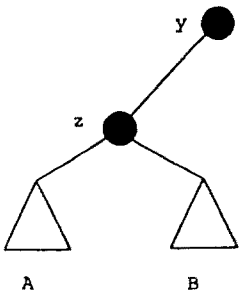


Fig. 3.

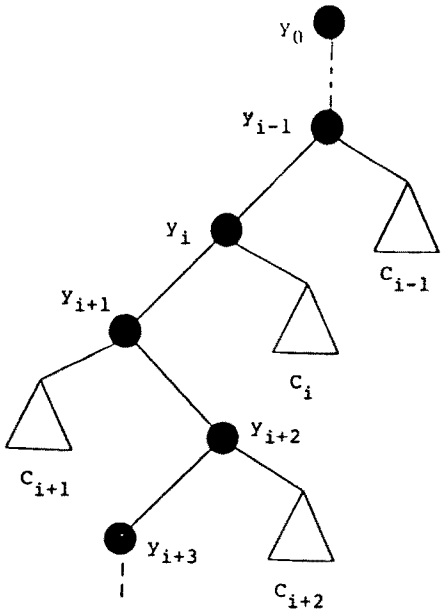


Fig. 4. Tree T with y_{i+1} and y_i left children.

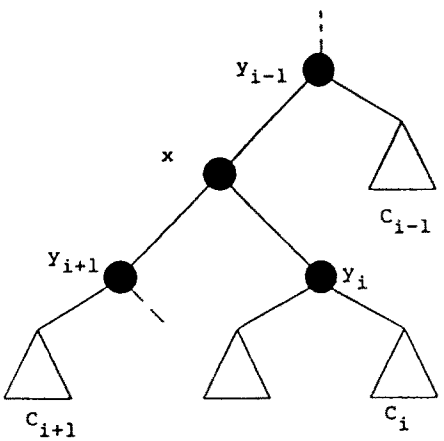
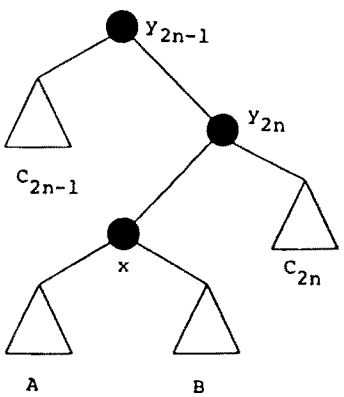


Fig. 5.



Now consider the case where y_{i+1} is of odd depth. The bottom-up algorithm transforms x to be the left child of y_i by using zig-zag steps. By applying a zig-zag step and completing the splaying, we obtain the tree where y_i is the parent of y_{i-1}. When applying the top-down algorithm, we do a right link in y_i and a left link y_{i+1}. Hence, y_{i-1} is the parent or sibling of y_i in the right tree and also in the resulting tree. As above we can conclude that the resulting trees cannot coincide.

The proof that the algorithms produce equal trees when the direction path is of the form described in the theorem is similar to those given above and is omitted. ■

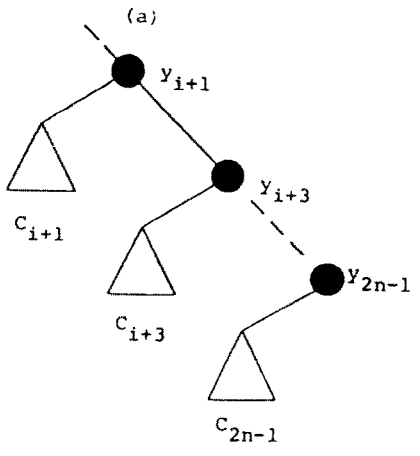


Fig. 6. (a) left tree.

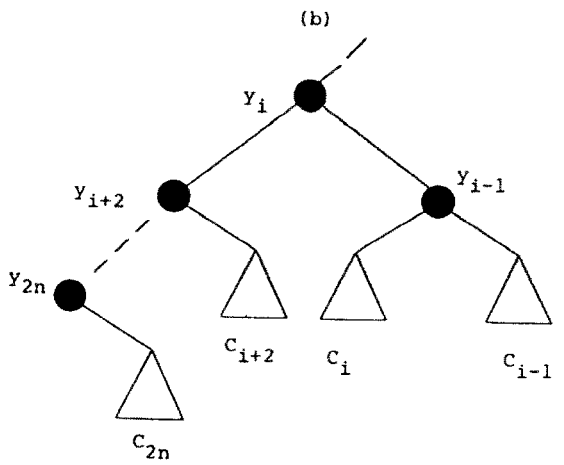


Fig. 6. (b) right tree.

4. How to charge for the primitives of top-down splaying.

Sleator and Tarjan [3] studied the amortized complexity of splaying by using a potential function. Suppose each item in a splay tree has a positive weight. The *size* $s(x)$ of a node x is defined to be the sum of the weights of all items in the subtree rooted at x. Then the *rank* $r(x)$ of x is defined to be $\log s(x)$ (where log stands for binary logarithm), and the potential of the tree is defined to be the sum of the ranks of all its nodes.

It is proved in [3] that the amortized time to bottom-up splay a tree with root t at a node x is at most $3(r(t)-r(x))+1$. This proof is based on the assumption that one unit is a suitable charge for a rotation. The same upper bound is given, although without a proof, for the amortized time of top-down splaying. We now give a simple proof for this fact. The proof uses the assumption that one unit is charged for both rotation and link primitives. This assumption is called the *equal charge assumption.*

THEOREM 3. *The amortized time under the equal charge assumption to top-down splay a tree with root t at a node x is at most* $3(r(t)-r(x))+1$.

PROOF. If the node x is of even depth then, by theorem 1, the resulting tree is as in the bottom-up case. Hence, we can apply the approximation of [3].

Now suppose that x is of odd depth. If the depth of x is 1, then the amortized time is at most $1+r(t)-r(x)$ [3]. Suppose (the induction hypothesis) that if x is of depth at most $2i-1$, $i>0$, then the amortized time is at most $3(r(t)-r(x))+1$.

Consider a tree T in which x has depth $2i+1$. Let y be the node in the access path of x having depth 2. Furthermore, let T^y be the subtree of T

having root y. The tree $td(x, T)$ can be created by first top-down splaying in T^y at x, then replacing T^y by $td(x, T^y)$ and top-down splaying in the new tree at x.

By the induction hypothesis, the amortized time of the first operation is at most $3(r(y)-r(x))+1$. After this operation x is of depth 2 and the rank of x is the same as the rank of y before the operation. Depending on the form of the new direction path of x, an upper bound $3(r(t)-r(x))$ or $2(r(t)-r(x))$ can be proved for the amortized time of the latter operation [3]. In both cases the amortized time of the entire splaying operation is at most $3(r(t)-r(x))+1$. This completes the proof. ■

Contrary to bottom-up splaying, top-down splaying has two different ways (rotation and link) to move on along the access path. This raises the question whether or not it is reasonable to charge equally for these operations when determining the amortized time of the algorithm. In chapter 2 we implemented the primitives by using four assignment statements and one local variable for rotation and three assignment statements for link. This suggests that 3/4 is a reasonable amount to charge for link if one unit is charged for rotation. This assumption is called the *reduced charge assumption.* Following the proof of [3, lemma1] it is easy to prove the following.

THEOREM 4. *The amortized time under the reduced charge assumption to top-down splay a tree with root t at a node x is at most* $2.79(r(t)-r(x))+1$.

5. Empirical results.

The purpose of this chapter is to gain further insight into the behaviour of different kinds of top-down restructuring algorithms by experimenting their function on some test data. The form of experiments is as follows. Successive access operations are done to initially empty trees. If the accessed item is not found, then it is inserted to the trees. Integers from given intervals are used as item types.

Since self-adjusting data structures are introduced especially for skew access patterns and for patterns which may change, uniformness is not the most important criterion when choosing the test data. Our first experiment uses a naive random number generator [2, p. 145]. Table 1 shows the total charges for rotation and link operations when integers are generated from the interval [0...1000]. The restructuring primitives are charged in proportion to the number of Pascal-statements needed in their implementation: four units is charged for rotations and three units for links. The total charges shown are obtained by using 1 as the seed number of the random number generator; they represent well the general behaviour of the algorithms.

Table 1. *Total charges in a tree with 1000 nodes.*

numbers of accesses	top-down splaying	simple top-down splaying	Stephenson's variant
5000	186795	192647	167607
25000	927800	959769	826695
100000	3706134	3835902	3295509

Some comments concerning table 1 are in order. First, Stephenson's variant seems to be more efficient than top-down splaying. Second, the total charges of top-down and simple top-down splaying are almost equal, top-down splaying being slightly better.

Our second test data imitate locality as follows. We fill in a tree with 1000 nodes by making 5000 access operations to initially empty trees as above. During this phase no charges are counted for the restructuring primitives. After that, every nth access is made from the interval [0...1000] and all

Table 2. *Total charges for a skew access pattern in a tree with 1000 nodes, $n = 3$.*

numbers of accesses	top-down splaying	simple top-down splaying	Stephenson's variant
10000	356577	368726	314826
25000	888616	918832	784941
500000	1775195	1834403	1567590

other accesses are made from the interval [800-900] or from the interval [200...300] so that the first 1000 accesses are made from the first interval, then 1000 from the latter, and so on. Tables 2 and 3 show the total charge counts for the values $n = 3$ and $n = 11$, respectively. These tables show the same ratio between the charges of top-down splaying and Stephenson's variant as table 1 (charge of Stephenson's variant/charge of top-down splaying ≈ 0.89).

Table 3. *Total charges for a skew access pattern in a tree with 1000 nodes, $n = 11$.*

numbers of accesses	top-down splaying	simple top-down splaying	Stephenson's variant
10000	370715	383033	328965
25000	925447	955824	820785
50000	1848475	1910347	1639929

The difference between the charges of top-down and simple top-down methods shown in tables 1-3 can be explained by considering the numbers of rotations done. Table 4 shows the numbers of the restructuring primitives in the case of table 1.

Table 4. *Numbers of the restructuring primitives in the case of table I.*

numbers of accesses	top-down splaying		simple top-down splaying	
	rotations	links	rotations	links
5000	15279	41893	19643	38025
25000	73541	211212	95166	193035
100000	292284	845666	378882	773458

(The cases of tables 2 and 3 are similar.) Simple top-down algorithms perform rotations more often than the basis form of top-down splaying and also the total number of the restructuring primitives is greater in simple top-down splaying.

Our last test data illustrate an access sequence on which Stephenson's variant performs poorly. Following [1, Thm. 3.5] we make the following access sequence to initially empty trees: 1, 2,..., 1000, 1, 2,...,500, 1, 2,... 500. Table 5 shows the total charges. Note that in this extreme case simple top-down splaying is better than the basic form top-down splaying.

Table 5. *Total charges for a special access sequence.*

numbers of accesses	top-down splaying	simple top-down splaying	Stephenson's variant
1500	12147	10529	1125747
2000	20666	17975	1501494

We have compared three different top-down restructuring methods by counting the numbers of Pascal statements executed in their restructuring primitives. We have seen that Stephenson's variant "normally" performs better than the more complicated splaying algorithms. ("Normally" includes many access distributions not studied in this chapter.) More importantly, we have found that simple top-down splaying is not an improvement over the basic form of the algorithm. This answers a question by Sleator and Tarjan [3, p. 670].

REFERENCES

1. B. Allen and J. Munro, *Self-organizing binary search trees*, J. ACM 25 (1978), 526–535.
2. N. Cooper and M. Clancy, *Oh! Pascal!* (2nd edition). Norton, New York and London, 1985.
3. D. D. Sleator and R. E. Tarjan, *Self-adjusting binary search trees*, J. ACM 32 (1985), 652–686.
4. C. J. Stephenson, *A method for constructing binary search trees by making insertions at the root*, Int. J. Comput. Inf. Sci. 9 (1980), 15–29.
5. R. E. Tarjan, *Amortized computational complexity*, SIAM J. Appl. Discrete Meth. 6 (1985), 306–318.