Self-Adjusting Binary Trees

Daniel Dominic Sleator
Robert Endre Tarjan

Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

We use the idea of *self-adjusting* trees to create new, simple data structures for priority queues (which we call heaps) and search trees. Unlike other efficient implementations of these data structures, self-adjusting trees have no balance condition. Instead, whenever the tree is accessed, certain adjustments take place. (In the case of heaps, the adjustment is a sequence of exchanges of children, in the case of search trees the adjustment is a sequence of rotations.) Self-adjusting trees are efficient in an amortized sense: any particular operation may be slow but any sequence of operations must be fast.

Self-adjusting trees have two advantages over the corresponding balanced trees in both applications. First, they are simpler to implement because there are fewer cases in the algorithms. Second, they are more storage-efficient because no balance information needs to be stored. Furthermore, a self-adjusting search tree has the remarkable property that its running time (for any sufficiently long sequence of search operations) is within a constant factor of the running time for the same set of searches on any fixed binary tree. It follows that a self-adjusting tree is (up to a constant factor) as fast as the optimal fixed tree for a particular probability distribution of search requests, even though the distribution is unknown.

1. Introduction

In this paper we present new ways of using binary trees to store heaps (otherwise known as "priority queues") and search trees (also called "dictionaries", "lists", or "sorted sets"). The ideas and techniques of analysis that we use for these two problems promise to be applicable to other data structure problems.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0-89791-099-0/83/004/0235 $00.75

Standard tree structures for representing heaps (e.g. leftist trees [9,12]) and search trees (e.g. AVL trees [1], 2-3 trees [2], trees of bounded balance [13]) obtain their efficiency by obeying an explicit balance condition that indirectly bounds the length of the relevant paths in the tree. With such a condition any single access or update operation takes O(log n) time in the worst case, where n is the number of items in the tree.

We describe ways of doing away with any explicit balance condition while retaining the ability to do access and update operations efficiently. Rather than maintaining balance, we adjust the tree during each operation using simple adjustment heuristics. These adjustments are the same as those used in balanced trees (exchanging children in the case of heaps and performing single rotations in the case of search trees). The difference is that they are applied in a uniform fashion without regard to balance. The result is that the trees behave (in an amortized sense) as though they are balanced. This approach has the following advantages (in both applications):

(i) We can save space of at least one bit per node in the tree structure, since no balance information needs to be maintained.

(ii) Balanced tree algorithms are plagued by a multiplicity of cases. Our algorithms are simpler to state and to program.

(iii) In most balanced search tree schemes the tree remains static when only search operations are done. Since self-adjusting search trees adapt to the input sequence dynamically, they can perform better (by an arbitrary factor) than a fixed tree when the access pattern is non-uniform.

Self-adjusting trees have two disadvantages, the significance of which depends on the application. One is that more adjustments are made than in the corresponding balanced structures. (Maintaining a self-adjusting search tree requires more rotations than a balanced tree, and maintaining a self-adjusting heap takes more swapping of children than a leftist heap.) The cost of a rotation in a search tree, which we assume to be O(1), depends upon the application. If rotations are unusually expensive, self-adjusting search trees may be inefficient.

The other possible disadvantage is that by a carefully chosen sequence of operations it is possible to construct a very unbalanced binary tree. Thus the worst-case bound per operation is $O(n)$, not $O(\log n)$. However with our adjustment heuristics the running time per operation is $O(\log n)$ when amortized over any sequence of operations. That is, a sequence of m operations ($m \geq n$) will take $O(m\log n)$ time in the worst-case, even though a few operations in the sequence may take $\Omega(n)$ time. Since almost all uses of heaps and search trees involve a sequence of operations rather than just a single operation, an amortized bound is generally as useful as a bound on each operation. The only situation in which this might not be true is a real-time application in which it is important to have a worst-case bound on the running time of each individual operation.

There is little previous work on self-adjusting binary search trees. Allen and Munro [3] (getting their start from Rivest's work [14] on self-organizing linear lists used for sequential search) proposed two adjustment heuristics based on single rotation: *single exchange*, in which an accessed item is rotated one step toward the tree root, and *move to root*, in which an accessed item is moved all the way to the tree root by rotating at every edge along the access path. Allen and Munro proved that move to root is efficient on the average, but simple exchange is not. Bitner [7] studied the average-case behavior of several other heuristics.

Our results are much stronger than those of Bitner and Allen and Munro. Their heuristics are efficient for an average sequence of operations, but there are pathological sequences for which the running time is $\Omega(n)$ per operation.

A self-adjusting search tree has the further remarkable property that its running time for any sufficiently long sequence of search operations is within a constant factor of the running time for the same set of searches on any fixed binary tree. It follows that a self-adjusting tree is as efficient (to within a constant factor) as the optimal fixed tree for a particular probability distribution of search requests. Such an optimal tree can only be constructed under the optimistic assumption that the access probabilities are available in advance.

Another application of self-adjusting search trees is in the data structure for dynamic trees of Sleator and Tarjan [15,16,17]. We can substitute self-adjusting trees for biased trees [4,5,6] in that structure without affecting the running time. The resulting data structure is significantly simpler since weights no longer have to be maintained.

In Section 2 we describe *self-adjusting heaps*, prove a bound on their running time, and present programs to implement them. In Section 3 we describe *self-adjusting search trees*, and prove that they have the claimed properties. In Section 4 we give programs for two versions of self-adjusting search trees, and in Section 5 we discuss additional results and future work.

2. Self-Adjusting Heaps

A *heap* is a data structure consisting of a set of items selected from a totally ordered universe, on which the following operations are possible.

findmin(h): Return the minimum item in heap h.

deletemin(h): Delete the minimum item from heap h and return it.

insert(i, h): Insert item i into heap h, not previously containing i.

meld(h_1, h_2): Return the heap formed by combining disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.

There are several ways to implement heaps in a self-adjusting fashion. The one we discuss in detail is related to the leftist trees of Crane [9] and Knuth [12]. These heaps are so simple that we call them simply *self-adjusting heaps*. A self-adjusting heap is a binary tree with one item per internal node. (All external nodes are null.) Each node x has three fields associated with it, denoted *item*(x), *left*(x), and *right*(x). The *left* and *right* fields are pointers to the left and right children, and the *item* field contains the item of that node. The items are stored in *heap order*: If x and y are nodes and x is the parent of y, then $item(x) \leq item(y)$. To identify and access the heap we use a pointer to the tree root.

At the end of this section we give programs for the heap operations; here we give an informal description of how the operations are implemented. Since heap order implies that the root is the minimum element, we can perform *findmin* in constant time by returning the item at the root. The other two operations are implemented using meld. To do *deletemin* we meld the left and right subtrees of the root and return the (old) root. To do *insert* we make a one-item heap out of the item to be inserted and meld it with the existing heap.

To do *meld* we first delete all the edges (but not the nodes) on the right paths (paths from the roots to **null** nodes through right children) of the two trees. This creates a forest of trees whose roots have no right child. The trees are then connected together in heap order by a new right path through all of the roots. In other words we merge the right paths of the two trees. (See Figure 1.) The time for a meld is proportional to the length of the new right path.

To make this algorithm efficient we must keep right paths short. Leftist trees accomplish this by maintaining the following property: from any node, the right path is a shortest path to an external node. Maintaining this property requires storing at every node the minimum distance to an external node, and, after a meld backing up along the merged path recomputing distances and swapping left and right children as necessary to maintain the leftist property. The length of the right path in a leftist tree of n nodes is at most $\lfloor \lg n \rfloor$, so each of the heap operations has an $O(\log n)$ worst-case time bound.

In our self-adjusting version of this data structure we meld by merging the right paths of the two trees and then

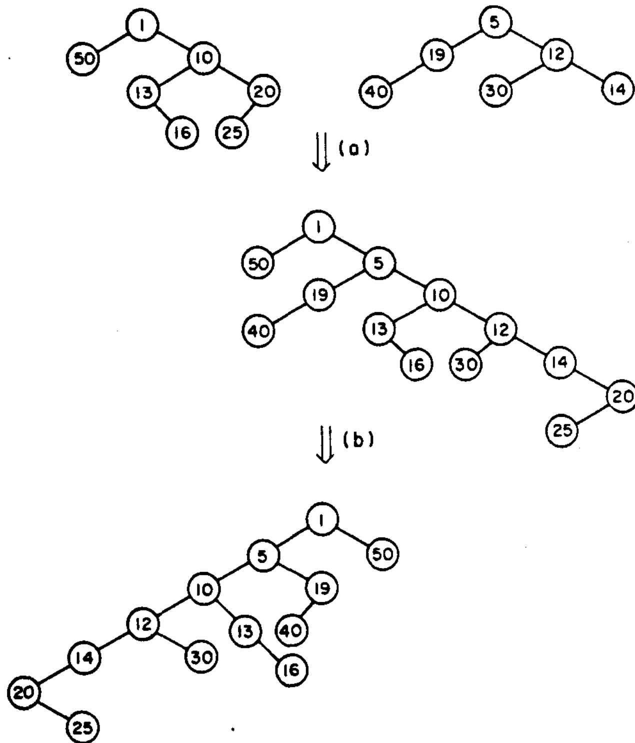

Figure 1. A meld of two self-adjusting heaps.
(a) Merge of right paths.
(b) Swapping of children along path formed by merge.

swapping the left and right children of **every** node on the merged path. (See Figure 1.) This makes the potentially long right path formed by the merge into a left path. The theorem and corollary below bound the time needed by a sequence of melds, and by an arbitrary sequence of self-adjusting heap operations.

Theorem 1: In a sequence of melds starting with singleton heaps, the number of edges deleted during the melds is at most $3\Sigma\lfloor\lg(n_i)\rfloor$, where n_i denotes the number of nodes in the tree resulting from the ith meld.

Proof: This proof is based on the ideas used by Sleator [15], and Sleator and Tarjan [16] to bound the number of "splice" operations in a network flow algorithm. We define the *weight* of each node in a heap to be the number of nodes in the subtree rooted there. We use these weights to divide the edges into two classes: *heavy* and *light*. The edge connecting a node x to its parent $p(x)$ is heavy if the weight of $p(x)$ is less than twice that of x and light if the weight of $p(x)$ is at least twice that of x. Two facts follow immediately:

Fact 1: Of the edges from a node to its children, at most one can be heavy.

Fact 2: The number of light edges on the path from a node x to the root of a tree of weight w is at most $\lfloor\lg(w)\rfloor$.

To get the bound we focus our attention to the number of right heavy edges. (These are the heavy edges that connect a node to its right child.) This quantity (which we denote by *RH*) starts at zero. As we meld, *RH* fluctuates, but it never falls below zero. Let a and b be the two trees to be melded by the ith meld. Let n_a and n_b be their weights, and let $n_i=n_a+n_b$ be the weight of the tree resulting from the meld.

We wish to bound the total length of all the meld paths (the right paths traversed in the trees to be melded). To do this we consider the effect of the ith meld on *RH* . By Fact 2 the number of light edges on the meld path of heap a is at most $\lfloor\lg(n_a)\rfloor$. Similarly the number of such edges in heap b is at most $\lfloor\lg(n_b)\rfloor$. Thus the total number of light edges on the two paths is at most $2\lfloor\lg(n_i)\rfloor-1$. (See Figure 2.)

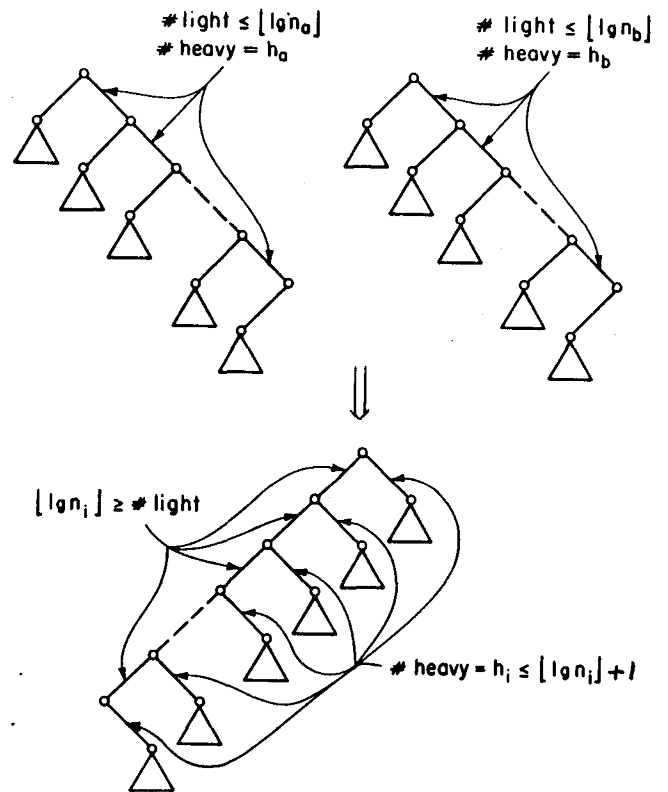

Figure 2. The movement of heavy edges in meld.

Let h_a be the number of heavy edges on the meld path of heap a, and let h_b be the number on the meld path of heap b. Let h_i be the number of right heavy edges incident to the leftmost path of the tree produced by the ith meld. Fact 1 tells us that each edge counted by h_i (except possibly a bottom one) corresponds to a light edge in the leftmost path of the heap produced by the ith meld. By Fact 2 the number of such light edges is at most $\lfloor\lg(n_i)\rfloor$, so $h_i \leq \lfloor\lg(n_i)\rfloor+1$. The only right heavy edges removed in the meld process are those counted by h_a and h_b. The only ones added by the meld are those counted by h_i. Thus *RH* decreases by at least h_a+h_b, then increases by at most $h_i \leq \lfloor\lg(n_i)\rfloor+1$.

Since RH is nonnegative the total increase bounds the total decrease. Therefore the number of heavy edges on all the meld paths is at most $\Sigma(\lfloor \lg(n_i) \rfloor + 1)$. Furthermore the number of light edges on all the meld paths is at most $\Sigma(2\lfloor \lg(n_i) \rfloor - 1)$. Combining these estimates gives the result. □

Note. The version of meld described above (and used in the proof) is not quite the same as that presented below. In the actual implementation the two right paths are traversed from the top down. When one of the paths ends the other is simply attached to it, and the process terminates. Only those nodes that are traversed have their children exchanged. (This differs from the description above in which all nodes on the right paths are always traversed.) The same theorem holds for the actual implementation, and the same proof works with slight modification. RH still increases by at most $\lfloor \lg(n_i) \rfloor + 1$, and it decreases by at least the number of heavy edges on the meld paths. The original analysis holds for the light edges. □

Corollary 1: A sequence of m findmin, deletemin, insert, and meld operations takes $O(\Sigma \lg(n_i))$ time, where n_i is the weight of the largest tree involved in the ith operation.

Proof: The time for findmin is $O(1)$, and insert is just a special case of meld. Thus to get the result we only have to modify the above proof to consider deletemin. Deletemin simply removes the root, then does a meld. The only relevant effect of deleting the root is that it may decrease RH by one. This only improves our bound on the length of the meld paths, so we have the result. □

What follows is an implementation of the four operations on self-adjusting heaps. The data structure is as we described it in the text; each node has three fields: *item*(x), *left*(x) and *right*(x). The programs are written in a variant of Dijkstra's guarded command language [10]; we have used the symbol "|" to denote Dijkstra's "box", and the symbol "←→" to denote the "swap" operator. The variables of type **heap** are actually pointers to nodes. Parallel assignments all take place simultaneously, so the result is well-defined even if the same variables appear on the left and right sides.

```
item function findmin(heap h);
    return item(h)
end findmin;

heap function insert(item i, heap h);
    create a new node to which n points;
    left(n), right(n), item(n) := null, null, i;
    return meld(n, h)
end insert;

heap function deletemin(modifies heap h);
    heap i;
    i := h;
    h := meld(left(h), right(h));
    return i
end deletemin;
```

We have included two versions of meld; a recursive one, *rmeld*, and an iterative one, *imeld*. The function *rxmeld* is supplied to avoid doing extra tests for **null** in the recursive version.

```
heap function rmeld(heap h1, h2);
    if h2=null → return h1
    | h2≠null → return rxmeld(h1, h2) fi
end meld;

heap function rxmeld(heaph1, h2);
    if h1=null → return h2 fi;
    if item(h1) > item(h2) → h1←→h2 fi;
    left(h1), right(h1) := rxmeld(right(h1), h2), left(h1);
    return h1
end rmeld;
```

In the iterative version of meld the invariant at the beginning of the loop is that there are three heaps rooted at x, h_1, and h_2 that contain all of the nodes. Node y is in the heap rooted at x, and its left child is eventually going to be the meld of heaps h_1 and h_2.

```
heap function imeld(heap h1, h2);
    heap x, y;
    if h1=null → return h2 | h2=null → return h1 fi;
    if item(h1) > item(h2) → h1←→h2 fi;
    x, y, h1, right(h1) := h1, h1, right(h1), left(h1);
    do h1≠null →
        if item(h1) > item(h2) → h1←→h2 fi;
        y, left(y), h1, right(h1) := h1, h1, right(h1), left(h1)
    od;
    left(y) := h2;
    return x
end imeld;
```

Note. The swapping of h_1 and h_2 in the loop can be avoided by writing different pieces of code for the cases *item*(h_1) > *item*(h_2) and *item*(h_1) ≤ *item*(h_2). The four-way parallel assignment can be written with four separate assignments as:

```
left(y) := h1;
y := h1;
h1 := right(y);
right(y) := left(y);
```

With this implementation each iteration of the loop takes four assignments and two comparisons. □

We have tested the iterative and recursive versions of self-adjusting heaps (exactly as shown above) as well as the

iterative and recursive versions of leftist heaps. (The iterative version we used is on page 619 of [12], and the recursive version is in [17].) Preliminary results indicate that recursive self-adjusting heaps and both forms of leftist heaps are about equally fast. However the iterative version of self-adjusting heaps is significantly faster than the others.

The leftist heap algorithms must make two passes over the merged path: one pass down to connect the pieces together, and one pass up to swap children and update the distance fields. The recursive version does this by saving the path in the recursion stack, and the iterative version does this by reversing pointers on the way down, and then fixing them on the way up. The iterative version avoids the overhead of recursion at the cost of more pointer assignments. The iterative version of self-adjusting heaps is fast because it has no recursive calls, does no extra pointer manipulation, and makes only one pass over the merged path. These advantages make up for that fact that the average meld path is longer in a self-adjusting heap than in a leftist heap. According to Brown [8], binomial heaps are faster than leftist heaps. It would be interesting to find out how self-adjusting heaps compare to binomial heaps.

In some applications it is useful to have another form of delete:

delete(x): Delete node x from the heap containing it, and return the resulting heap.

It is impossible to implement this type of delete with the data structure described above. To implement *delete*(x) it is necessary to find the parent of x so that its pointer to x can be changed. This means that we need a pointer from each node to its parent. If there is such a pointer then *delete*(x) can be done as follows: first do *deletemin*(x) (which removes node x from the tree rooted at x), then connect the resulting tree to the parent of x. All of the other operations can be modified in a straightforward fashion to update parent pointers.

There is a way to allow deletion in self-adjusting heaps while still using only two pointers per node. In node x we keep a *down* pointer and an *across* pointer. If x is the root, then *across*(x) is **null**. If x is an only child or a right child then *across*(x) points to the parent of x. If x is a left child and x has a sibling, then *across*(x) points to that sibling. If x has no children then *down*(x) is **null**, otherwise *down*(x) points to the leftmost child of x. This representation might be called a "triangular heap" since a node and its two children are connected by a cyclic "triangle" of pointers. Knuth [11] calls this the "binary tree representation of a tree, with right threads". Notice that if a node is an only child there is no distinction between it being a left child or a right child. This doesn't matter since the tree is heap ordered, and the algorithm can assume that an only child is a left child. By following at most two pointers from a node we can access its parent or its left or right child, which is all we need to implement all of the heap operations.

3. Self-Adjusting Search Trees

The data structure we call a "search tree" might more appropriately be called a "symmetrically ordered binary tree", because most of its applications have nothing to do with searching. In the most general sense, a symmetrically ordered binary tree is a data structure that is used to represent a list of items. The fundamental property of the list of items that is captured by the symmetrically ordered binary tree is the order of the items in the list. The kind of operations that a symmetrically ordered binary tree can efficiently support are those that involve manipulation of nearby items in the list. In general a symmetrically ordered binary tree can represent the items as internal or as external nodes in the tree, but in the trees we discuss the items will be in the internal nodes. In a symmetrically ordered binary tree the items are arranged in symmetric order: if x is a node containing item i, then every item in the left subtree of x comes before i in the list, and every item in the right subtree of x comes after i in the list.

The basic operation that is generally used to modify a symmetrically ordered binary tree is the *single rotation*, because a rotation maintains the symmetric order of the nodes. (Case 1 of Figure 4 shows a rotation.) Rotations are used in AVL trees [1], trees of bounded balance [13], biased binary trees [6], and many others. Our self-adjusting symmetrically ordered binary trees are no exception. (The reader may be relieved hear that, having dispelled any misunderstanding about what a search tree is, we shall proceed to call our data structure a self-adjusting search tree.)

For the purposes of the discussion that follows we have assumed that the object to be represented is a list of numbers ordered by value. In a node x, there are three fields: *item*(x) (the number stored in node x), and *left*(x) and *right*(x) (pointers to the left and right subtrees of x). Every external node is **null** and we access and identify a tree with a pointer to the tree root. We shall discuss the following operations:

access(i,s): If item i is in tree s return a pointer to its location, otherwise return **null**.

insert(i,s): Insert item i into tree s, and return the resulting tree.

delete(i,s): Delete item i from tree s if it is there, and return the resulting tree.

join2(s_1,s_2): Return a tree representing the items in s_1 followed by those of s_2, destroying s_1 and s_2. (This assumes all items of s_1 are less than those of s_2.)

join3(s_1,i,s_2): Return a tree representing the items in s_1 followed by item i, followed by the items of s_2. This destroys s_1 and s_2. (This assumes that items of s_1 are less than i, and i is less than the items of s_2.)

split(i,s): Assuming item i is in tree s, return a tree s_1 containing all those items of s less than i and a tree s_2 containing all those items greater than i. This operation destroys s.

The following operation is unique to self-adjusting search trees, and is the one from which we build all of the others.

splay(i,s): Return a tree representing the same list represented by s. If i is in the tree, then it becomes the root. If i is not in the tree, then either the immediate successor of i or the immediate predecessor of i becomes the root. This operation destroys s.

To do *access*(i,s) we *splay*(i,s); then i is in the tree if and only if it is at the root. To do *insert*(i,s) we *splay*(i,s), then break the resulting tree into two trees, one with items less than i, one with items greater than i. (This is just breaking either the left or the right link from the root.) Then we make these two trees the children of a new root with item i. To do *join*2(s_1, s_2) we *splay*(*infinity*,s_1), which makes the rightmost node of s_1 into the root; then we attach s2 as the right child of this root. To do *join*3(s_1, i, s_2) we make a node containing item i, and make its left child s_1 and its right child s_2. To do *delete*(i,s) we *splay*(i,s), delete the root, and *join*2 the left and right subtrees. To do *split*(i,s) we *splay*(i,s) and return the left and right subtrees of the root. (See Figure 3.)

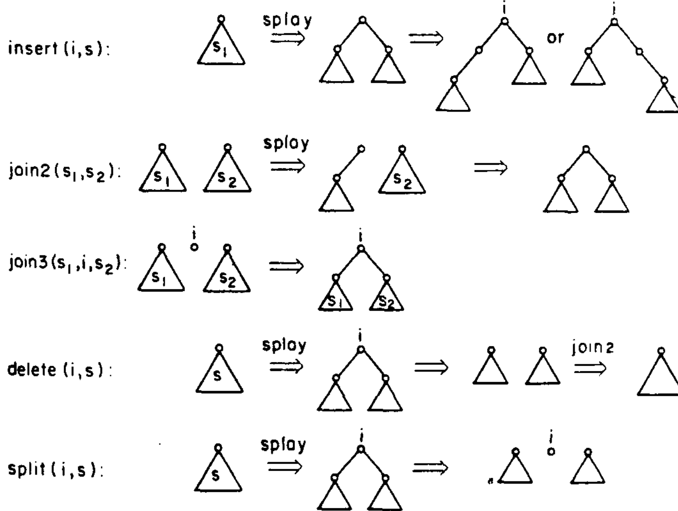

Figure 3. How the operations are implemented using splay.

To do *splay*(i,s) we first use the item fields to find the vertex that is going to be moved to the root. We start with y equal to the root of s and repeat the following search step until y=**null** or *item*$(y)=i$: If $i<$*item*(y), replace y by its left child; if $i>$*item*(y), replace y by its right child. Let x be the last non-null vertex reached by this process; this is the vertex to be moved to the root. To finish the splay we begin at node x and traverse the path to the root, performing a single rotation at each node. The rotations are done in pairs, in an order that depends on the structure of the tree. The following *splay step* is repeated until x is the tree root (see Figure 4): If x has a parent but no grandparent, rotate at $p(x)$ (the parent of x). If x has a grandparent and x and $p(x)$ are both left or both right children, rotate at $p(p(x))$ then at $p(x)$. If x has a grandparent and x is a left and $p(x)$ a right child, or vice-versa, rotate at $p(x)$ and again at the new $p(x)$. The overall effect of the splay is to move x to the root while rearranging the rest of the original path from x to the root so that any node in that path is about half as far from the root as it used to be. Figure 5 shows a series of splays on a tree that starts out being a long left path.

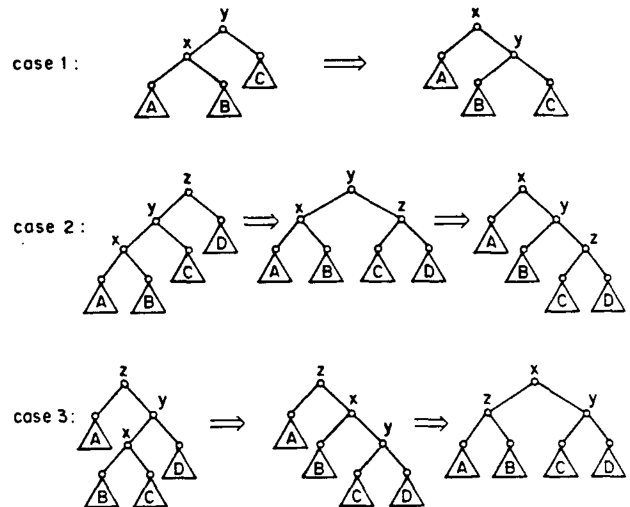

Figure 4. A splay step starting at node x.

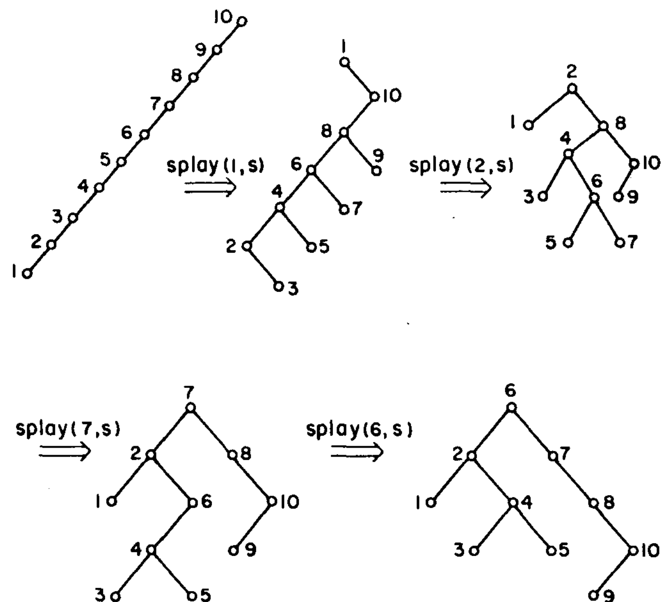

Figure 5. Four splay operations.

Splaying is reminiscent of the path compaction heuristics (path halving in particular) used in efficient algorithms for disjoint set union [17,18]. Although the techniques that Tarjan and van Leeuwen [18] used to analyze path halving can be modified to apply to splaying, there is a simpler analysis, which follows.

To analyze the running time of a sequence of tree operations we use a *credit invariant*. (This is what we called a *token invariant* or *chip invariant* in our previous work with Bent [4,5,6].) We assign to each item i an *individual weight* $iw(i)$. These weights are real numbers greater than or equal to one, whose values we shall choose later. We define the *total weight* $tw(x)$ of a node x to be the sum of the individual weights of all descendants of x, including x itself. Finally, we define the *rank* of a node x to be $r(x) = \lfloor \lg(tw(x)) \rfloor$. We maintain the following *credit invariant*: Any internal node x holds $r(x)$ credits.

Each credit represents the power to do a fixed amount of work (e.g. rotations, comparisons, or edge traversals). Each time we do any work we must pay for it with a credit. If we modify the structure we may have to put in credits to maintain the credit invariant, or we may be able to remove credits (because less are required after the modification than before) and use them to do work. If we have a structure that initially has C credits in it and we do a sequence of n operations where the ith one requires $c(i)$ net credits (number spent on work + number put in the tree − number taken out), and the final structure has C' credits in it, then the running time of the sequence is at most $C - C' + \Sigma c(i)$. The quantity $c(i)$ is called the *credit time* of the ith operation. The following lemma tells us the credit time of the splay operation.

Lemma 1. Splaying a tree with root v at a node x while maintaining the credit invariant takes $3(r(v) - r(x)) + 1$ credits.

Proof. We shall need the following *rank rule*: If s and t are siblings with equal rank, and their parent is p, then $r(p) \geq 1 + r(s)$. This follows from the fact that $tw(s) \geq 2^{r(s)}$ and $tw(t) \geq 2^{r(t)}$, so $tw(p) \geq tw(s) + tw(t) \geq 2^{r(s)+1}$. Thus $r(p)$ is at least $r(s) + 1$.

Consider a splay step involving the nodes x, $y = p(x)$, and $z = p(p(x))$, where $p(\ )$ denotes the parent function before the step. Let $r(\ )$ and $r'(\ )$, $tw(\ )$ and $tw'(\ )$ denote the rank and total weight functions before and after the step, respectively. To this step we allocate $3(r'(x) - r(x))$ credits and one additional credit if this is the last step. There are three cases (see Figure 4):

Case 1: Node z is undefined. This is the last step of the splay and the extra credit allocated to the step pays for the work. We have $r'(x) = r(y)$. Thus the number of credits that must be added to the tree to maintain the invariant is $r'(y) - r(x) \leq r'(x) - r(x)$, which is one third of the remaining credits on hand.

Case 2: Node z is defined and x and y are both left or both right children. We have $r'(x) = r(z)$. The number of credits that must be added to the tree to maintain the invariant is $r'(y) + r'(z) - r(y) - r(x) \leq 2(r'(x) - r(x))$, which is two thirds of the credits on hand. If $r'(x) > r(x)$, there is at least one extra credit on hand to pay for the step. Otherwise, $r'(x) = r(x) = r(y) = r(z)$. In this case $r'(z) < r(x)$ by the rank rule. (The rank rule is applied to the tree occurring after one rotation, with root y of rank $r'(x) = r(z)$, left subtree rooted at x with rank $r(x)$, and right subtree rooted at z of rank $r'(z)$.) Also $r'(y) \leq r(y)$. Thus by putting the credits from y onto y and putting all but one of the credits from x onto z we maintain the invariant and get one credit with which to pay for the operation.

Case 3: Node z is defined and x is a left and y is a right child or vice-versa. As in Case 2, $r'(x) = r(z)$. In addition we have that $tw'(y) \leq tw(y)$, so $r'(y) \leq r(y)$. To maintain the invariant on x and y we need only move credits from z and y. To satisfy the invariant on z we use the credits on x and need an additional $r'(z) - r(x) \leq r'(x) - r(x)$, which is one third of the credits on hand. If $r'(x) > r(x)$ then there is at least one extra credit on hand to pay for the step. Otherwise $r'(x) = r(x) = r(y) = r(z)$, and by the rank rule either $r'(y) < r'(x)$ or $r'(z) < r'(x)$ or both. We can use the credit from the node that decreased in rank to pay for the operation.

Summing over all steps of a splay, we find that the total number of credits used is at most $3(r'(x) - r(x)) + 1 = 3(r(v) - r(x)) + 1$, where $r'(\ )$ and $r(\ )$ denote the rank function before and after the entire splay. □

In order to complete the analysis we must consider the effect of insertion, deletion, join2, join3, and split on the ranks of nodes. For the moment let us define the individual weight of every item to be 1. Then every node has a rank in the range $[0, \lfloor \lg n \rfloor]$, and the lemma gives a bound of $3 \lfloor \lg n \rfloor + 1$ credits for splaying. To insert a new item i we first do a splay, then put the new item at the root. The number of credits needed at the root is $\lfloor \lg n \rfloor$. Joining two trees also requires at most $\lfloor \lg n \rfloor$ new credits at the root. (In both cases n is the size of the new tree.) A three way join requires at most $\lfloor \lg n \rfloor$ credits at the root. To delete, we do a splay, remove the root, then do a two way join. This needs no extra credits beyond those used by the two splays because the credits on the deleted root can be placed on the root of the final tree. Split needs no credits beyond those used in the splay.

Suppose we start with a set of singleton trees, do a series of operations and end up with a forest of trees. The number of credits is zero initially, and at the end it is at least zero. Combining this with the above paragraph gives us the following theorem:

Theorem 2: The total time required for a sequence of m self-adjusting search tree operations, starting with singleton trees, is $O(m \log n)$, where n is the number of items.

Our analysis of splay allows us to get an analogous but more general result when the individual weights of the nodes are not all the same. Suppose the initial configuration consists of a set of separate nodes, and the number of credits on node i with individual weight $iw(i)$ is $\lfloor \lg(iw(i)) \rfloor$. After a sequence of operations we reach a final configuration with the same set of nodes grouped into arbitrary trees. In this final forest of trees, the number of credits on node i with total weight $tw(i)$ is $\lfloor \lg(tw(i)) \rfloor$. Since $tw(i) \geq iw(i)$, the number of credits in the final configuration is at least as many as in the initial configuration. This means that the total running time of the sequence of operations is bounded by the number of credits allotted to the operations. Recall that this allotment of credits to an operation is called the *credit time* of the operation. The following theorem bounds the credit time of each of the basic operations as a function of the weights of the nodes involved.

Theorem 3:

The credit time of *splay*(x,s) is $O(\lg \frac{tw(s)}{tw(x)})$.

The credit time of *split*(x,s) is $O(\lg \frac{tw(s)}{tw(x)})$.

The credit time of *join*3(s_1,i,s_2) is $O(\lg \frac{tw(s_1)+tw(s_2)}{iw(i)})$.

The credit time of *join*2(s_1,s_2) is $O(\lg \frac{tw(s_1)+tw(s_2)}{tw(x)})$, where x is the rightmost node of tree s_1.

The credit time of *insert*(x,s) is $O\left(\lg \frac{tw'(s)}{\min(tw(x^-),\, tw(x),\, tw(x^+))}\right)$, where x^- is the node immediately before x in the final tree, x^+ is the one immediately after, and $tw'(s)$ is the total weight of s after the operation.

The credit time of *delete*(x,s) is $O\left(\lg \frac{tw(s)}{\min(tw(x^-),\, tw(x))}\right)$, where x^- is the node immediately before x in the initial tree.

Proof: All of these results follow by considering the credit times of the appropriate splay operations and combining these with the changes in the credits needed on various nodes. □

The remarkable thing about this result is that the algorithm achieves these bounds without actually having any information about the weights. This means that whatever the running time is, it must simultaneously satisfy the bounds given in Theorem 3 for all weight distributions.

The credit times for split, three-way join, insert, and delete given in Theorem 3 are the same as those that Bent, Sleator, and Tarjan give for biased trees [6]. The credit times for two-way joins on self-adjusting trees and biased trees are not comparable, because in self-adjusting trees the items are stored in the internal nodes and in biased trees they are stored in the external nodes. Self-adjusting trees are simpler than biased trees because no weight information needs to be stored or updated. The only situation in which biased trees may have an advantage is if worst-case per-operation running time is important. In locally biased trees, access operations have a good worst-case time bound; in globally biased trees, all the operations have a good worst-case time bound [6].

Another interesting consequence of Theorem 3 is that we can relate the behavior of a self-adjusting tree to that of any static tree.

Theorem 4: Let t be the number of comparisons that occur in a sequence of searches from the root in a static binary search tree with n nodes. The time to do the same sequence of splay operations in a self-adjusting search tree is $O(t+n^2)$.

Proof: Let the root of the static tree be r. Let the *depth* of a node x in the static tree (denoted $d(x)$) be the distance from x to the root, r. ($d(r)=0$.) We assign individual weights to the nodes as follows: For the root r, $iw(r)=3^d$ (where d is the largest depth in the tree). For any other vertex x, $iw(x)=3^{-d(x)}iw(r)$. With this definition the deepest node has weight 1. It is easy to show by induction that $3iw(x) \geq tw(x)$ for all nodes x. (Here $tw(x)$ denotes the total weight of x in the static tree.) In particular we have $3iw(r) \geq tw(r)$, from which it follows that $iw(x) \geq 3^{-d(x)-1}tw(r)$. Rearranging and taking logarithms gives us

$$(\lg 3)(d(x)+1) \geq \lg \frac{tw(r)}{iw(x)}.$$

The left hand side of this inequality is lg3 times the number of comparisons needed to search for x in the static tree. The right hand side is the credit time to splay at x in a self adjusting tree with the individual weights as specified above.

It remains for us to show that the number of credits initially in the self-adjusting tree is $O(n^2)$. It is clear that the total weight of any node in the self-adjusting tree is at most $tw(r)$. But $tw(r) \leq 3iw(r)=3^{d+1} \leq 3^n$, because d, the largest depth in the tree, is at most $n-1$. This means that the number of credits on each node is at most $(\lg 3)n$, so the total number of credits in the tree initially is at most $(\lg 3)n^2$. □

A corollary of this result is that the running time of a self-adjusting tree is within a constant factor of the running time of the optimal static tree for any particular distribution. The surprising thing about this is that the self-adjusting tree behaves this way without knowing anything about the distribution in advance. (Note however that the self-adjusting tree takes some time to "learn" the distribution. This learning time is embodied in the $O(n^2)$ overhead.)

There is another interesting result concerning the running time of sequences of splays. It follows from the alternate analysis of splay (involving ideas from path compression), and we do not prove it here.

Theorem 5 ($\log(\Delta t)$ theorem): The credit time of a splay of node x is $O(\log(1+\Delta t))$, where Δt is the number of splays done between the current splay of x and the previous splay of x.

4. Implementation of Self-Adjusting Trees

There are several slightly different formulations of the splay operation that all have the desirable properties discussed in Section 3. The method of Section 3 is easy to describe, but results in a fairly complicated program. In this section we present programs for two splay algorithms: a top-down method, and a bottom-up method.

In the first formulation the information stored in each node is the same as that described in Section 3: each node has *left*, *right* and *item* fields. The difference is that here the splay operation rearranges the tree on the way down, instead of searching down and rearranging the tree on the way up. (We do not present programs for the other operations, because they are easy to write given splay.)

```
global tree dummy;
dummy := create tree node;

tree function splay(item i, tree s);
    string state;
    tree l, r, lp, rp;
    if s=null → return null fi;
    left(dummy) := null; right(dummy) := null;
    state := "N";     l := r := dummy;
    do s≠null and i > item (s) →
        if state≠"L" → right(l) := s; lp := l; state := "L"
         | state="L" → right(l) := left(s); right(lp) := s;
                       left(s) := l; state := "N"
        fi;
        l := s; s := right(s)
     | s≠null and i < item(s) →
        if state≠"R" → left(r) := s; rp := r; state := "R"
         | state="R" → left(r) := right(s); left(rp) := s;
                       right(s) := r; state := "N"
        fi;
        r := s; s := left(s);
    od;
    if s≠null → right(l) := left(s); left(r) := right(s)
     | s=null →
        if r=dummy → s := l; right(lp) := left(l)
         | r≠dummy → s := r; left(rp) := right(r) fi
    fi;
    left(s) := right(dummy); right(s) := left(dummy);
    return s
end splay;
```

Variables of type **tree** are pointers to nodes in the tree. Initially s (assumed to be non-**null**) points to the root of the tree, and the "left" tree and "right" trees are empty. (The *right* and *left* fields of the dummy node point to the roots of the left and right tree respectively. The dummy node obviates special purpose code for the case when the left or the right tree is **null**.) Splay walks down the tree building up the left tree from all those subtrees to the left of i and building the right tree from all those subtrees to the right of i. (The variables l and r and lp and rp point to the places in the left and right trees where the building takes place.) When it reaches either **null** or the item it was looking for, it stops and puts the left tree, the node it was looking for, and the right tree together and returns the result. The three-value variable *state* is used to alter the way in which a tree is added to the left tree or the right tree. This variable is what enables this version of splay to achieve the time bounds of Section 3.

In many potential uses of self-adjusting trees the process of searching for an item is unnecessary. In these cases we already have a pointer to the node we are interested in, and we want to find out something about the relationship between it and the rest of the nodes. (An example of this is in the data structure for dynamic trees, in which we want to find out the minimum cost node of all those to the left of a particular node.) In these applications the item field is useless, but in its place we need a *parent* pointer. With a parent pointer we can find our way to the root of a tree given only a pointer to some node in it. If desired we can avoid using extra parent pointers by using the "triangular" representation method at the end of Section 2.

The following is a bottom-up implementation of splay that uses parent pointers. The input to splay is a pointer to a node. Splay makes that node into the root of the tree. The parent of the root is **null**.

```
tree function rotateleft(tree a);
    tree b;
    b := right(a);
    right(a), parent(left(b)) := left(b), a;
    left(b), parent(a) := a, b;
    return b
end rotateleft;

tree function rotateright(tree a);  [analogous to rotateleft]

procedure splay(tree x);
    string state;
    tree l, r, y, z;
    state="N";
    l, r, z := left(x), right(x), x;
    y := parent(x);
    do y≠null →
        if right(y)=z →
            right(y), parent(l) := l, y;
            z, l, y := y, y, parent(y);
            if state≠"L" → state := "L"
             | state="L" → l := rotateleft(l);
                           state := "N" fi
         | left(y)=z →
            left(y), parent(r) := r, y;
            z, r, y := y, y, parent(y);
            if state≠"R" → state := "R"
             | state="R" → r := rotateright(r);
                           state := "N" fi
        fi
    od;
    left(x), parent(l) := l, x;
    right(x), parent(r) := r, x;
    parent(x) := null
end splay;
```

The function *rotateleft* does a left rotation of a node (a) and its right child (b). It returns the new parent (b). In the case that the left child of b is **null**, an assignment to the parent of **null** takes place. Allowing this simplifies the code, and only costs one extra node.

This implementation of splay is analogous to the first one, except that the traversal is bottom-up instead of top-down. The variables l and r point to the current left and right trees. The variable *state* is used as before to guide the rotations. The advantage of this bottom-up method over the one described in Section 3 is that there are fewer pointer updates.

Split, join and insert can be implemented essentially as described in Section 3. The operation *delete*(x) can be implemented with only one call to splay. First the children of x are joined together, then the resulting tree is placed where x was in the oritinal tree. (We needn't traverse the path from x to the root, nor splay along that path.)

5. Additional Results

Our results on self-adjusting binary trees raise a number of general and specific questions about self-adjusting data structures, and we are continuing our work in this area. Below we mention one major application and several variants of self-adjusting search trees and heaps.

Self-adjusting search trees can be used in place of biased trees in the dynamic tree structure of Sleator and Tarjan [15,16,17]. The result is a considerable simplification of that structure. Details may be found in [17], and will appear in the full version of this paper.

There is another version of splay that might be called "move half way to the root". In this version the splayed node does not move all the way to the root, but its distance to the root is halved. The only difference between it and the bottom-up version is that in case 2 (see Figure 4) only the first rotation is done. The theorems of Section 3 apply to this version of splay. The reason it is not as useful as the move to root versions is that the join and split operations have to be implemented separately, rather than following automatically from splay.

It is possible to implement self-adjusting search trees with all of the items stored in the external nodes. Splay then moves a specified external node to within two steps of the root. The algorithm to do this is a simple modification of the algorithm to do bottom-up splay given in Section 4. One advantage of this is that the actual time to join two trees is constant, although the credit time is O(log n).

Another alternative implementation of splay is one in which the ranks are actually kept in the nodes. Rotations are only done in the splay when the rank of a node is equal to the rank of its grandparent (using the splay of Section 3). With this implementation the number of credits in the tree decreases by at least one with each rotation. This means that if we do a series of searches the number of rotations is bounded by the number of credits initially in the tree. Although this version does far fewer rotations, it lacks much of the beauty of the other forms of self-adjusting search trees. Weights must be specified in advance and kept in the nodes, and a different version of splay must be used for the split and join operations.

There are also other versions of self-adjusting heaps that deserve mention. We can build a heap directly out of a self-adjusting search tree by making each node of the search tree correspond to one of the keys in the heap. (The keys can be stored in any an arbitrary order.) In each node of the self-adjusting search tree we store two fields, a *key* field and a *minkey* field. The *key* field is just the key represented by that node. The *minkey* field contains the minimum key in the subtree rooted there. To do a deletemin we first find the node with the minimum key by walking down the tree from the root always taking the child that has the minimum *minkey* field, until we get to the node whose *key* is the *minkey* of the root. We then use our self-adjusting search tree routine to delete this node from the tree. (The routines can easily be modified to maintain the *minkey* field.) The advantage of this form of heap is that the keys can also be maintained in total order that is independent of the heap order of the keys. (Splitting and joining of these heaps based on the symmetric order of the nodes is possible.)

We can use a similar method to represent a heap by a self-adjusting search tree with all of the keys stored in the external nodes. In this version each external node has a *key* field, and each internal node has a *minkey* field. The advantage of this scheme is that we can meld two heaps (by joining the self-adjusting trees) in constant actual time, as mentioned earlier in this section.

We conjecture that under a suitable measure of complexity, self-adjusting trees perform within a constant factor of any binary search tree scheme, on any sequence of operations. We have formulated a rigorous version of this conjecture and are attempting to prove it. Details will appear in the full paper.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information," *Soviet Math. Dokl.*, **3** (1962), 1259-1262.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] B. Allen and I. Munro, "Self-organizing search trees," *Journal ACM* **25** (1978), 526-535

[4] S. W. Bent, *Dynamic Weighted Data Structures*, TM STAN-CS-82-916 Computer Science Dept., Stanford University, Stanford, CA 94305, 1982.

[5] S. W. Bent, D. D. Sleator and R. E. Tarjan, "Biased 2-3 trees," *Proc. Twenty-First Annual IEEE Symp. on Foundations of Computer Science* (1980), 248-254

[6] S. W. Bent, D. D. Sleator and R. E. Tarjan, "Biased search trees," to appear.

[7] J. R. Bitner, "Heuristics that dynamically organize data structures," *SIAM Journal on Computing* **8** (1979), 82-110.

[8] M. R. Brown, *The Analysis Of a Practical and Nearly Optimal Priority Queue*, TM STAN-CS-77-600 Computer Science Dept., Stanford University, Stanford, CA 94305, 1977.

[9] C. A. Crane, *Linear Lists and Priority Queues as Balanced Binary Trees*, TM STAN-CS-72-259 Computer Science Dept., Stanford University, Stanford, CA 94305, 1972.

[10] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.

[11] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[13] J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," *SIAM journal on Computing*, **2** (1973) 33-43.

[14] R. L. Rivest, "On Self-organizing sequential search heuristics," *Comm. ACM* **19** (1976), 63-67.

[15] D. D. Sleator, *An* O(*nm*log *n*) *Algorithm for Maximum Network Flow*, TM STAN-CS-80-831 Computer Science Dept., Stanford University, Stanford, CA 94305, 1980.

[16] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal Computer and System Sciences*, to appear; see also *Thirteenth Annual ACM Symposium on Theory of Computing* (1981), 114-122.

[17] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983, to appear.

[18] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *Journal ACM*, submitted.