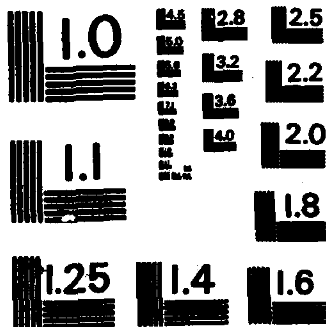
AD-A122 046 DYNAMIC WEIGHTED DATA STRUCTURES(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE S W BENT JUN 82 STAN-CS-82-916 N00014-76-K-0330 1/1

UNCLASSIFIED F/G 12/1 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A122046

June 1982

Report No. STAN-CS-82-916

12

Dynamic Weighted Data Structures

by

Samuel W. Bent

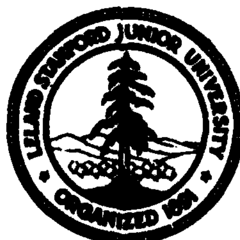
Contract N00014-76-K-0330

Department of Computer Science

Stanford University
Stanford, CA 94305

DTIC
ELECTE
DEC 3 1982
S D
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited



DTIC FILE COPY

044-402

82 12 03 017

Dynamic Weighted Data Structures

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

by
Samuel Watkins Bent
May 1982

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Dedication

To Lucinda, Scott, and Laurie

Acknowledgments

The path to the Ph.D. degree is long and often difficult. Now that I am at the end of it, I would like to express my deepest appreciation to all the wonderful people who helped me along it.

For all their help throughout my life, I thank my father, Gardner Bent, and my mother, Mickey Bent.

For financial support, I thank the National Merit Scholar Program, and the National Science Foundation, whose graduate fellowship I held for three years.

For inspiration, knowledge, and advice, I thank my teachers Ted Sage, Juris Hartmanis, John Hopcroft, Don Knuth, Andy Yao, and especially Bob Tarjan, who deserves a medal for his patience.

For providing an excellent environment for research, a powerful computer facility, and the TEX typesetting system, I thank the Stanford Computer Science Department and Don Knuth (again). I also thank the MACSYMA project at M.I.T., whose amazing system produced with very little help from me.

For professional support and encouragement, I thank my "computer" friends Lyle Ramshaw, Mark Brown, Terry Roberts, Jim Boyce, Harry Mairson, Jay Gischer, Paul Asente, Dan Sleator, Alex Strong, Phyllis Winkler, and most of all John Gilbert, who more than once kept me from losing hope.

For friendship and belief in me, I thank more people than I can count, including Lucinda Mercer, Scott Sakaguchi, Laurie Miller, the Military Way Chorus, the Savoyards, the Stanford Symphony, Margie Weiss, Hank Rosak, Danny Montoro, Kris Yenney, Peter Jaffe, Jeannine Wagar, and Bonnie Hampton, who taught me much more than how to play the cello.

Finally, for hours of artistic satisfaction, I thank Johannes Brahms, Ludwig van Beethoven, Igor Stravinsky, Gian-Carlo Menotti, and Johann Sebastian Bach.

Dynamic Weighted Data Structures

Samuel W. Bent

This thesis discusses implementations of an abstract data structure called a *dynamic dictionary*. Such a data structure stores a collection of items, each of which is equipped with a *key* and a *weight*. Among the operations we might wish to perform on such a collection are:

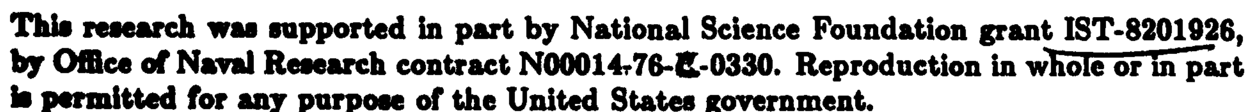
(a) accessing an item, given its key
(b) inserting a new item
(c) deleting an item
(d) joining two collections into one
(e) splitting a collection into two
(f) changing the weight of an item

Operations (b)–(f) provide the dynamic nature of the data structure.

In addition we want the implementation to respect the weights, so that accessing a heavy item is quicker than accessing a light one. In an optimal binary tree, the path length to an item of weight w in a collection of total weight W is proportional to $\log(W/w)$. By relaxing the optimality constraint and considering different kinds of trees, it is possible to retain this logarithmic access time (with a larger constant factor), and simultaneously achieve similar logarithmic times for the dynamic operations.

Two new data structures are proposed, *biased 2-3 trees* and *biased weight-balanced trees*. They achieve the logarithmic time bounds provided the cost is amortized over a sequence of operations. These data structure have applications to the network flow problem and to the design of "self-organizing" data structures.

This research was supported in part by National Science Foundation grant IST-8201926, by Office of Naval Research contract N00014-76-C-0330. Reproduction in whole or in part is permitted for any purpose of the United States government.



Chapter 1

Introduction and Motivation

One morning I awoke to find Sherlock Holmes intently rearranging note cards on the breakfast table. As his friend and physician, I was pleased to see that he had emerged from the lethargic stupor into which he had fallen in the last few weeks; it meant that once again he had found a project worthy of his great intellect. With the greatest curiosity as to the nature of his labor I greeted him.

"What are you doing, Holmes?"

"Organising my master criminal index," he said without removing his eyes from the table. "I find it a matter of some difficulty to arrange these cards in a manner suited to my needs."

I glanced at the cards and noticed each was labelled with the name of one of London's fiends, followed by dates, descriptions, *modi operandi*, and all the other facts Holmes had gleaned in his tenure as the world's foremost consulting investigator.

"Why don't you just arrange them alphabetically?" I asked.

"Excellent, my dear Watkins! You have reduced the problem to the utmost simplicity and applied the soundest logic to produce a solution. However, it won't do."

My smile fell suddenly. "Why not?"

"You see, Watkins, you would treat Moriarty the same as a common pickpocket. Somehow I want to place the cards of archvillains such as Colonel Moran and Irene Adler in a more prominent position than those of the minions and laborers of the criminal world. Yet I must not ignore the alphabet, for clues often appear in the form of monogrammed handkerchiefs or initials in correspondence. Furthermore, as a man's evil star ascends, so should his card achieve greater prominence, which should then fade as Lestrade and his

colleagues (with whatever humble assistance we may provide) curb the activities of the offender."

"Why, Holmes, that seems impossible!" I cried.

"Never say 'impossible', Watkins. Surely man is clever enough to overcome the difficulties nature provides him."

Just then, Mrs. Hudson ushered into the room the man who introduced us to the curious case of the Giant Rat of Sumatra, a tale I hope to be able to add some day to the public record. However, our conversation over that breakfast table lingered in my mind for many years, and I often tried to invent a simple system that would satisfy Holmes' requirements.

To state the problem more precisely, Holmes wishes to manage collections of *items*. Each item contains information about a single criminal, most of which is unimportant to the organization of the index. For that task, all that is important is the name of the felon, which we may call the *key* of the item, and the importance Holmes attaches to him, which we may call the *weight* of the item. Holmes needs to gain access to items, to split a collection into smaller pieces, to unite collections into larger ones, to change the weights of items, and to add or delete items. Each of these operations must take into account the weights; they should proceed very quickly on the more important items at the expense of proceeding more slowly on the less important ones.

In short, Holmes wants an implementation of an abstract data structure called a *dynamic dictionary*, in which the cost of each operation is a function of the weights involved, both of the operand and of the entire dictionary.

1.1. Dynamic dictionaries.

A *dynamic dictionary* is an abstract data structure that stores a collection of *items*. Each item may have a number of attributes, most of which depend on the application. For the purpose of maintaining the data structure, the important attributes of an item are its *key* and its *weight*.

The keys are drawn from a totally ordered set K, called the *key space*. Typically the keys are integers or alphabetic strings, with the usual ordering relation. For simplicity, we will assume that a dictionary can store only one item with a particular key. (This assumption can be dispensed with either by a convention about equal keys or by enlarging the key to disambiguate equal keys; both techniques are standard, and neither affects any of the implementations to be discussed here.) In some applications, the keys and the ordering

of the items are implicit in the data structure, in which case it makes no sense to talk about the key space $\mathcal{K}$ or equal keys.

The weights are real numbers strictly greater than zero. (In some applications it may be necessary to restrict the weights to be bounded away from zero by choosing some number $\epsilon > 0$ as a lower bound.) They are assigned by the user of the data structure, who presumably has chosen them to indicate the relative importance of the items. The manner in which an implementation is expected to respect the weights will be discussed soon.

In the context of a particular dictionary D, suppose we are given a key K. The *item of* K, denoted $I(K)$, is defined to be the item in D whose key is K, if there is such an item; it is undefined if there is not. If the item of K is defined, it is unique, since D may contain at most one item with key K.

The *weight of* K, denoted $W(K)$, is the weight of the item of K; it is undefined if the item of K is also undefined.

Our key K partitions the items in D into three classes, namely those items whose keys are less than K, those items whose keys are greater than K, and the item of K itself. The first two of these classes are called the *left-items of* K and the *right-items of* K, respectively.

If the key of each item in D is less than the keys of all items in another dictionary D', we say that D *precedes* D'. This relation between dictionaries is a necessary condition for the JOIN operation to be well-defined (see below).

With this terminology, we may define the following operations on a dynamic dictionary D (or on a pair of dictionaries D_1 and D_2):

1. ACCESS. Given a key K, return the item of K, or an indication that no such item exists.

2. JOIN. If D_1 precedes D_2, construct a new dictionary D containing all the items of D_1 and D_2, and discard the old dictionaries. (JOIN is undefined if D_1 does not precede D_2.)

3. SPLIT. Given a key K, split D into three parts: a new dictionary D_1 containing the left-items of K, the item of K, and a new dictionary D_2 containing the right-items of K.

4. DELETE. Given a key K, discard the item of K from D.

5. PROMOTE. Given a key K and a real number δ, add δ to the weight of K.

6. DEMOTE. Given a key K and a real number δ, subtract δ from the weight of K, provided that the resulting weight is still positive (or greater than the lower bound ϵ).

7. INSERT. Given an item, add it to D, provided that its key is different from all keys in D. (If D already holds K, do nothing or signal an error.)

Using these definitions, we get a data structure that is used in a *top-down* manner. A typical command consists of a key, a dictionary, and an operation. Before the operation can be carried out, we must first search the dictionary for the approporiate item, starting from a root associated with the name of the dictionary. The search will involve comparing the given key with keys in the dictionary and making decisions based on the total ordering of $\mathcal{K}$.

An alternative way to use a dictionary is the *bottom-up* manner. Instead of a key, we are given an explicit pointer to an item, so we need not do any searching. Rather, we apply the operation directly to the item, in the context of whatever dictionary happens to contain it. With this style of query, the ACCESS operation is replaced by a new operation:

8. FIND. Given a pointer to an item, return the name of the dictionary containing that item.

Since no searching is done, there is no need to have keys at all. The ordering among items can be implicit in the way the items came to be in the same dictionary. Whenever we JOIN two dictionaries D_1 and D_2 (in that order), we simply *define* that the items in D_1 precede the items in D_2.

The way a dictionary is used depends on the needs of the application. The more familiar manner of use, and the one assumed by the algorithms presented here, is the top-down manner. However, some important applications assume bottom-up use. Fortunately, it is fairly simple to adapt the top-down algorithms to bottom-up ones, and the analyses presented here carry through with little change.

1.2. Performance goals and entropy.

The definition of an abstract data structure says nothing about how the operations should be implemented, nor about how fast they should be. However, the user of a dynamic dictionary expects the implementation to favor the heavier items, in the sense that an ACCESS of a heavy item should take less time than an ACCESS of a light one, a SPLIT at a heavy item should be faster than a SPLIT at a light one, etc.

The weights have the following meaning to the implementor of a dynamic dictionary. The implementor assumes that an item is queried with probability equal to the proportion of the total weight represented by that item. Thus if a dictionary D contains (at a particular

time) items $I_1, I_2, \ldots, I_k$ with weights $w_i = W(I_i)$ for $i = 1, \ldots, k$, it has total weight

$$W = \sum_{1 \le i \le k} w_i,$$

and the probability of a command involving item I_i is (assumed to be)

$$\frac{w_i}{W}.$$

The user is expected to assign weights to items with this in mind.

As we use trees to implement dictionaries, we make the following standard definition.

Definition 1.1. Let T be a tree and let $\mathbf{w} = (w_1, \ldots, w_k)$ be the list of the weights of the items stored in T. The *total weighted path length* of T, denoted $L(T)$, is given by

$$L(T) = \sum_{1 \le i \le k} l_i w_i,$$

where l_i is the length of the path from the root of T to the ith item. The *average weighted path length* of T is simply

$$\frac{L(T)}{W} = \sum_{1 \le i \le k} \frac{w_i}{W} l_i.$$

We will measure the efficiency of an implementation of an operation by the weighted average of the times needed for that operation, as it is applied to each item in the dictionary. More precisely, consider a fixed implementation of the operations and a fixed dictionary D with items as above. If OP is one of the operations ACCESS, FIND, SPLIT, or DELETE, and if T is the running time function, define the *cost* $C(\text{OP})$ of the operation OP, as applied to D, by the formula

$$C(\text{OP}) = \sum_{1 \le i \le k} \frac{w_i}{W} T(\text{OP}(I_i)). \tag{1.1}$$

Each of these operations needs to select an item from the entire dictionary based on comparisons among the keys. Shannon's theorem on perfect encoding [12, p. 50] says that the weighted average, taken over all items, of the number of binary comparisons necessary just to do this selection is at least $H(w_1, \ldots, w_k)$, where H is the discrete entropy function,

defined by the formula

$$\begin{aligned} H(w_1, \ldots, w_k) &= -\sum_{1 \le i \le k} \frac{w_i}{W} \lg \frac{w_i}{W} \\ &= \sum_{1 \le i \le k} \frac{w_i}{W} \lg \frac{W}{w_i}. \end{aligned} \tag{1.2}$$

(Here $\lg x$ means $\log_2 x$.) Comparing (1.2) with (1.1), we see that the best we can hope to do is to implement the operations to run in time $\lg(W/w_i)$. Of course, the operations involve much more than selecting an item, so we will be content with an implementation running in time proportional to this lower bound. In other words, our goal is to implement each operation to run in time $O(\log(W/w_i))$.

The other operations have similar "best" running times, differing only in the way the extra parameters enter into the picture. The JOIN operation takes as arguments two dictionaries D_1 and D_2 with total weights W_1 and W_2. By symmetry, we may assume that $W_1 \ge W_2$ and define the cost by the formula

$$C(\text{JOIN}) = \frac{W_2}{W_1 + W_2} T(\text{JOIN}(D_1, D_2)).$$

By analogy with the above discussion, our goal for the JOIN operation is a running time of

$$O(\log \frac{W_1 + W_2}{W_2}).$$

Similarly, the PROMOTE operation should run in time $O(\log((W+\delta)/w_i))$, and the DEMOTE operation should run in time $O(\log(W/(w_i - \delta)))$.

Determining a goal for the INSERT operation poses a thorny problem. At first glance, it seems tempting to expect a running time of

$$O(\log \frac{W + w}{w})$$

to INSERT a new item of weight w into a dictionary of total weight W. But since the key of the new item might lie between the keys of two very light items, and since the dictionary must respect the key ordering, it may be necessary for the INSERT operation to find the two light items, and this could take a long time. More precisely, suppose w_1 and w_2 are

the weights of the items in D that immediately precede and follow the new item, according to their keys. Then if $w_0 = \min(w, w_1, w_2)$, the INSERT operation should run in time

$$O(\log \frac{W+w}{w_0}),$$

allowing us enough time not only to insert the new item but also to examine its two new neighbors.

In the worst case we might find $w_0 = w_{\min}$, the weight of the lightest item in the dictionary. Our revised goal is then much worse than our original one if w is large. But in two important special cases this pessimistic analysis is irrelevant. First, if the new item itself has weight equal to $w_{\min}$, the two goals are equal. And second, in the fortunate event that the new item precedes or follows all the items in the dictionary, we can view the new item as a dictionary unto itself, and view the INSERT as a special case of a JOIN. The analysis of the JOIN operation then gives us a running time goal of $O(\log((W+w)/w))$.

The remarks in the preceding paragraphs apply to any operation involving an item or key that does not appear in the dictionary. On the one hand, it is tempting to ask that the time necessary to perform an operation depend only on the operands (the item and the dictionary), and not on any fine structure of the dictionary such as the weights of particular items in it. But on the other hand, it seems reasonable to expect the operation to examine the "gap" where the item would be if it were in the dictionary, and thus to depend on the weights of the endpoints of that gap. For now we choose reason over temptation.

Definition 1.2. Any implementation of an operation which runs within the bounds mentioned above is said to achieve *logarithmic performance* (or to have *logarithmic behavior*, or to run in *logarithmic time*).

This thesis describes in detail two different implementations of dynamic dictionaries, called *biased 2-3 trees* and *biased weight-balanced trees*. These two data structures perform all the operations of Section 1.1 in logarithmic time (with one technical exception for biased 2-3 trees), provided that we *amortize* the running time over a sequence of operations. The nature of the amortization is particularly pleasant: starting with a forest of trivial dictionaries, in which each item of the universe is a dictionary unto itself, the total time needed for any sequence is at most the sum of the logarithmic bounds for the operations in the sequence, even though the time needed for a single operation may be more than its corresponding bound. In other words, any particular operation may use more time than the logarithmic bound, but only if previous operations used correspondingly less.

Furthermore, the amortization applies only to operations that alter the dictionaries, such as JOIN or SPLIT. The ACCESS and FIND operations always run in logarithmic time, and if they do not use all the time they are entitled to, the extra time does not need to be made available to later operations.

1.3. Applications.

The initial motivation for this research was an application to the maximum network flow problem. By using an earlier version of biased 2-3 trees [6] as the basis of a sophisticated array of data structures, Sleator [36] presented an implementation of Dinits' algorithm for finding a maximum flow in a network with n vertices and m edges that runs in time $O(mn \log n)$, improving on the best previously known bound by a factor of $\log n$. The technique he used, which relies on biased 2-3 trees to represent efficiently certain partially explored paths in the network, applies to several other problems, such as the transshipment problem and finding nearest common ancestors [37]. These applications use dictionaries in the bottom-up manner, and never need to INSERT in the middle of a tree.

Another application is a "self-organizing" data structure. The problem is to handle queries on items whose access probabilities are not known *a priori*, but to give an item more importance if it appears to be accessed more frequently. Dynamic dictionaries are well-suited to the task. Each ACCESS is coupled with a PROMOTE that increases the weight of an item by 1. Thus the weight of an item is its reference count; in the long run the ratio of an item's weight to the total weight will converge to its (unknown) access probability. The INSERT problem (see Section 1.2) is irrelevant here since we insert new items with weight 1, the minimum weight in the tree. This method gives logarithmic performance and maintains a tree with optimal path length (up to a constant factor) without using heuristics. Of course the complication of saving reference counts and building sophisticated data structures must be weighed against the simplicity of heuristic methods [7, 35], which use simple trees or lists and do not store counts.

Some large scale data-base problems might benefit from these algorithms, especially if the access pattern changes drastically with time. For example, an airline reservation system could have seasonal patterns of flights: people fly south in the winter and north in the summer. Of course, the weights must be highly skewed in order to achieve a savings over less complicated data structures.

1.4. Related work.

The notion of a dynamic weighted data structure is intended to simultaneously generalise two well-studied classes of data structures. The first class is that of *dynamic data structures*, in which most of the dynamic operations defined in Section 1.1 are supported, but in which all items are assumed to be of equal importance. The second class is that of *weighted data structures*, in which items are of unequal importance, but in which the set of items to be stored and the assignment of weights to these items is fixed beforehand. This section briefly reviews the work that has been done on these two classes as well as some earlier work on dynamic weighted data structures.

1.4.1. Dynamic Data Structures.

A great deal of work has been done on the the problem of implementing a data structure which supports some or all of the six operations

ACCESS
FIND
JOIN
SPLIT
DELETE
INSERT

as defined in Section 1.1, in the case where all the items have equal weights. Of course, the PROMOTE and DEMOTE operations are not applicable in this case.

Although linear data structures such as arrays, linked lists, and hash tables can be used to support these operations, they do so by favoring some of them over others. For example, we can JOIN two linked lists in time $O(1)$, but an ACCESS may take time of order n; we can INSERT into a hash table in expected time $O(1)$, but a SPLIT takes time of order n. We take the point of view that this phenomenon is undesirable — we wish to minimize the time needed for the worst operation.

The most appropriate implementations, therefore, are those in which all the operations cost about the same. The arguments in Section 1.2 specialize in the case that all the weights are the same (where we can take this common value to be 1), and indicate that we may expect to perform any of the legal operations in time $O(\log n)$ on a dictionary storing n items. A large class of data structures with this property exists, namely the class of balanced trees.

Balanced trees lead to logarithmic performance by keeping the bulk of the tree "balanced" among the various subtrees. No subtree is allowed to possess too much or too

little of the bulk. Various measures of bulk have been used, various balance conditions have been proposed, and various algorithms have been designed to implement the dynamic operations while maintaining the balance conditions. The most important measures are height, number of nodes, number of children, and path length, leading to AVL trees, BB trees, 2-3 trees (and their generalization B-trees), and RB-trees, respectively.

AVL trees, invented by Adel'son-Vel'skiĭ and Landis [1], are binary trees in which the bulk of a subtree is measured by its height. The heights of the two subtrees under a common parent must not differ by more than 1; the resulting trees are often called height-balanced for this reason. By storing the difference of the children's heights at each parent, and applying the simple operations of single and double rotation, it is possible to implement the ACCESS, INSERT, and DELETE operations in time $O(\log n)$. A nice description of these algorithms appears in Knuth [22]. In his dissertation, Crane showed how to use AVL trees to implement all six dynamic operations [10].

BB (bounded balance) trees are binary trees in which the bulk of a subree is measured by the number of nodes in that subtree. If a node has l nodes in its left subtree and r nodes in its right subtree, its balance is defined to be

$$\frac{l+1}{l+r+2}.$$

This balance is required to lie between α and $1-\alpha$, for some suitably chosen α. Nievergelt and Reingold show how to implement INSERT and DELETE in these trees by defining single and double rotation operations, and by describing when they are applicable [31, 34]. (In [34], the trees are called "weight-balanced". The "weight" in the name is merely the number of nodes in the tree; it is not the same as the weight of an item as defined in Section 1.1.) The biased weight-balanced trees of Chapter 3 specialize to these trees if we set the weight of each item to 2, use a more liberal value of α, and notice that there will never be any sub-item nodes.

B-trees [5] are multiway trees in which the bulk of a node is measured by the number of children it has. For m-ary B-trees, each internal node is required to have between $\lceil m/2 \rceil$ and m children, and all leaves are required to be the same distance from the root. The simplest case (and most important theoretically) is a 3-ary B-tree, also known as a 2-3 tree, in which each internal node has 2 or 3 children. It is fairly straightforward to implement all six dynamic operations using 2-3 trees, as described in Chapter 4 of [2].

Guibas and Sedgewick invented RB-trees (short for Red-Black trees, also known as dichromatic trees); these are binary trees in which each edge is colored either red or black

and all leaves are the same distance from the root, only counting black edges. Furthermore, certain local configurations of red edges are disallowed. By choosing these configurations appropriately, RB-trees are seen to generalize both AVL and B-trees, as well as other types of trees [14].

As a partial step toward a weighted structure, some proposals have been made that enable an unweighted structure to handle local reference efficiently [8, 26], or that make the likelihood of consecutive expensive rebalancing steps small [18, 19, 28].

1.4.2. Weighted Data Structures.

Some work has been done when the items are weighted, but when no dynamic operations are used. Knuth [25] shows how to construct the binary search tree with the optimal weighted path length in time $O(n^2)$; Hu and Tucker [17], and more recently Garsia and Wachs [13], also construct optimal trees in time $O(n \log n)$ under more restrictive hypotheses.

Huffman trees [20, 24, 33] are optimal trees in which the items have weights but no keys; that is, their relative order in the tree is immaterial. They can be constructed in time $O(n \log n)$ under very general hypotheses.

Various schemes for nearly-optimal weighted trees have appeared. Fredman [11] shows how to construct a nearly-optimal tree in time $O(n)$, and Bayer [4] gives a good bound on how close to optimal it gets. Many heuristics and empirical results have also been shown for weighted trees [3, 9, 32, 40].

1.4.3. Dynamic Weighted Data Structures.

Mehlhorn [29,30] proposes an implementation for dynamic dictionaries called *D-trees* (for Dynamic trees) in which it is possible to achieve logarithmic behavior for the ACCESS, PROMOTE, and DEMOTE operations. However, there is a potentially non-linear storage penalty inherent in his solution. He shows how to avoid this penalty (with *compact D-trees*), but the necessary manipulations are quite complicated. Our implementation is much simpler and uses only linear space, although we only achieve amortized logarithmic behavior (see below). But the amortization applies only to the operations that change the dictionary, and not to an ACCESS (or a FIND), which can always be done in logarithmic time in the worst case. Furthermore, the SPLIT operation, which is essential in the network flow application, is fast in our implementation. Mehlhorn does not discuss the SPLIT operation, but his implementation does not appear to admit a fast algorithm for it.

Unterauer [39] defines $B_{1/2}$ trees. He claims they support INSERT, DELETE, and weight changes, but his operations involve searching the tree to find the successor (in key order)

of the relevant node, and doing rebalancing along that search path. Such an approach cannot give logarithmic behavior, because any operation might lead to the bottom of the tree, regardless of the weights of the operands.

Knuth [23] shows how to maintain a binary tree that has minimum weighted path length under the operations of increasing or decreasing a weight by 1, in essentially optimum running time. This structure, like Huffman trees, has weights but no keys.

1.5. Summary of results.

In this thesis we propose two new implementations for dynamic dictionaries, called *biased 2-3 trees* and *biased weight-balanced trees*. They achieve logarithmic performance (with one minor exception for biased 2-3 trees), but only when we *amortize* the cost of maintaining the data structures over a sequence of operations. In other words, a particular operation may take more than logarithmic time, but the extra time is less than the time saved in previous operations.

Biased weight-balanced trees are more complicated than biased 2-3 trees, as they store real numbers and have three kinds of nodes (as opposed to storing integers in two kinds of nodes). However, the added complication circumvents the one technical imperfection in biased 2-3 trees.

The way amortization is presented here is more explicit than previous appearances of the idea. We introduce a physical analog, the poker chip, which seems to help in understanding where all the time is eventually spent. We prove an algorithm runs in the appropriate time by adding up the number of chips available to the operation (either from the initial allocation or from extra chips left over from previous operations), and showing that the total exceeds the number of chips the operation needs to spend.

It is interesting to compare the time bounds for this data structure with those for the weighted path compression implementation of the disjoint set data structure [38]. In the latter case, one can do a sequence of n UNION and FIND operations in time $O(n\alpha(n,n))$ but a particular operation may take a long time (of order $\log n$). However, a long operation will be followed *in the future* by enough short ones to achieve the amortized time bound. In the present case, whenever we do a long operation we can prove that enough short operations have been done *in the past* to achieve the amortized time bound. This may be useful for real-time applications.

Chapter 2

Biased 2-3 Trees

2.1. Definitions and notation.

Biased 2-3 trees are a generalization of 2-3 trees as defined by Aho, Hopcroft, and Ullman [2]. Whereas all the leaves in a 2-3 tree are the same distance away from the root, leaves in a biased 2-3 tree appear at various distances. This allows heavier items to be closer to the root than lighter ones.

A biased 2-3 tree contains two kinds of nodes, called *item* and *non-item* nodes. An item node corresponds to an external node (leaf) of a 2-3 tree; it has no children and contains one item. A non-item node corresponds to an internal node of a 2-3 tree; it may have either two or three children, but contains no items. The item nodes are arranged so that traversing the tree in symmetric order visits the items in order of increasing key value.

We will be somewhat sloppy about distinguishing between a node and the tree rooted at that node; the context should resolve any ambiguity.

If the dictionary is used in a "top-down" manner (as defined in Section 1.1), then non-item nodes should also contain additional "access keys" to guide searches for items. One possible scheme, as described in Chapter 4 of [2], is to store at each non-item node n two extra keys, called L and M, equal to the largest items in the left and middle subtrees, respectively, of n. These values must be updated at each node along the path from the root to n whenever the tree is altered at node n, but it is usually a simple matter to determine the new values. The time spent doing this is dominated by the time spent altering the tree, since any alteration requires traversing this path, at least in the schemes described

here. We will not mention the quantities L and M further, as no new ideas are needed to maintain them; they have the same meaning as for 2-3 trees.

Unbiased 2-3 trees are balanced by requiring that all leaves are the same distance from the root. Biased 2-3 trees have a more complicated balance condition. Before giving this condition, we define a measure of a node, called its *rank*, roughly corresponding to the height of a node in a 2-3 tree.

Definition 2.1. The *rank* of a node n, denoted $r(n)$, is defined as follows:

If n is an item node storing item I, then $r(n) = \lfloor \lg W(I) \rfloor$.

If n is a non-item node, then $r(n) = 1 + \max(r(m))$, where m ranges over the children of n.

Next we define a useful distinction among the children of non-item nodes.

Definition 2.2. A *major node* is one whose rank is maximum among all its siblings. A *minor node* is a node that is not major.

In other words, a major node is one that is responsible for its parent's rank being as large as it is. With this terminology, $r(n) = 1 + r(n')$, where n' is a major child of n.

In an unbiased 2-3 tree, all items have the same weight, say 1, so the rank of each leaf is 0. Inductively, we see that the ranks of the children of an internal node are all the same, and that the rank of a node equals its height. In a biased 2-3 tree, the ranks of leaves may vary, but we would still like the ranks of the children of an internal node to be equal (that is, we would like all its children to be major nodes). This turns out to be too restrictive; we cannot achieve this goal if one of the children is a very heavy item node. But we can come close, as the following definitons indicate.

Definition 2.3. Two trees S and T are *c-compatible*, for a given integer c, if $r(S) \leq c$, $r(T) \leq c$, and whenever one of $r(S)$ and $r(T)$ is strictly less than c, the other tree is simply an item node with rank c.

Definition 2.4. A non-item node with rank $c + 1$ is *balanced* if each adjacent pair of its children is c-compatible. A tree is *balanced* if all its internal nodes are.

In other words a node with rank $c + 1$ is balanced if each of its children with rank less than c is adjacent only to item nodes with rank c. In still other words, a tree is balanced if each of its minor nodes is adjacent only to major item nodes.

A biased 2-3 tree, then, is a balanced tree made up of binary and ternary non-item nodes and item nodes. Besides the information already mentioned (the item itself for an item node and access keys for a non-item node), each node must also contain enough

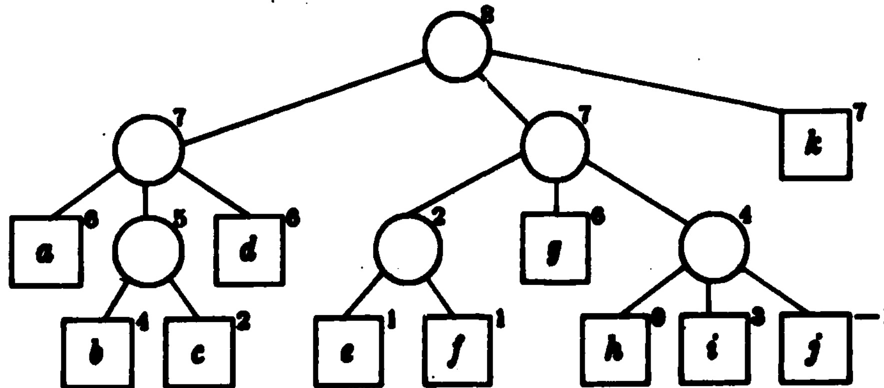


Figure 2.1.
A biased 2-3 tree.

information to determine its rank, as the algorithms for maintaining the trees depend strongly on the ranks. For simplicity we will assume each node actually contains its rank explicitly; in some applications it may be more efficient or convenient to use a less direct system, such as storing the difference in rank between a node and its parent.

This completes the definition of a biased 2-3 tree. Figure 2.1 shows a typical example. Item nodes are shown as squares, non-items nodes as circles. The letter inside an item node is the key of the item stored there; the number above a node is its rank. Note that ranks may be negative and that they increase along the path toward the root by at least one. Access keys are not shown.

Next we define a few useful notions.

Definition 2.5. The *weight* of a node n, denoted $W(n)$, is the total weight of all the items in the subtree rooted at n.

The weight of a node is the measure in which the user is presumably interested. We have discretised it through the rank function, but we will have to ensure that our algorithms, although dealing with ranks, repect the weights in some sense. See Section 2.2 for more details.

If S is a non-trivial tree (that is, if its root is a non-item node), define S_l to be its leftmost subtree, S_r to be its rightmost subtree, and S_m to be its "middle" subtree (provided S is ternary).

While maintaining a biased 2-3 tree, we may detach a node from its parent and replace it with another. Usually the new node is the root of a subtree very nearly equal to the tree

rooted at the old node, perhaps with a node added or deleted. When we reattach the new node, we have to check that the parent is still balanced. The following definition captures the notion that the new node can replace the old.

Definition 2.6. Let R and S be trees with ranks r and s, and let c be an integer. Then *R broadens S below c* if

i) $s \leq r \leq c$,

and ii) if $s = r = c$ and S is an item node, then R is also an item node.

The broadening relation is transitive, that is if R broadens S which in turn broadens T, then R broadens T, below c; it is monotone in c, that is if R broadens S below c, then it also does so below any $a \geq c$; and it extends the parent relation, that is if R is the parent of S, then R broadens S below r. It tells us when we can replace one node with another, as the next lemma indicates.

Lemma 2.7. If S is a node in a biased 2-3 tree whose parent has rank $c+1$, and if R broadens S below c, then the tree obtained by replacing S with R is a (balanced) biased 2-3 tree.

Proof. Suppose T is a sibling adjacent to S, so S and T are c-compatible. We must prove that R and T are also c-compatible.

If $s < c$, then T must be an item node with rank c, which is therefore c-compatible with any node R with rank $r \leq c$. If $t < c$, then S must be an item node with rank $s = c$; therefore $r = c$ by (i) and R is an item node by (ii), so R is c-compatible with T. The only other possibility is that $s = t = c$, in which case $r = c$ by (i), and so R is c-compatible with T.

2.2. Weight, rank, and the ACCESS operation.

Although we store the rank of a non-item node, the amount of weight represented by the subtree rooted at a node is usually more important to the user. The rank of a node ought to be related to its weight in a significant way. The following lemma shows that a node with large rank represents a substantial amount of weight.

Lemma 2.8. For any node n in a biased 2-3 tree,

$$W(n) \geq 2^{r(n)-1}.$$

Proof. If n is an item node, a stronger result is true since $\lg W(n) \geq \lfloor \lg W(n) \rfloor = r(n)$, so $W(n) \geq 2^{r(n)}$.

If n is a non-item node, it has among its children either

a) one major item node n' at rank $r-1$, so $W(n) \geq W(n') \geq 2^{r(n)-1}$ (by the strong result for item nodes), or
b) at least two major nodes n_1 and n_2 at rank $r-1$, so $W(n) \geq W(n_1) + W(n_2) \geq 2 \cdot 2^{r(n)-2} = 2^{r(n)-1}$.

We ACCESS the item with key K by comparing K with the access keys of nodes (beginning at the root) and recursively searching the appropriate subtree. This is the standard tree search algorithm.

Lemma 2.9. The ACCESS time for an item is proportional to the length of the path from the item to the root.

Proof. At each step along the path from the root to the item node containing the desired item, we do a constant amount of work comparing the search key with a small (fixed) number of access keys and deciding which subtree to search. Thus the total amount of work is proportional to the length of this path.

The next lemma shows that this path is not very long.

Lemma 2.10. The length of the path from an item to the root is at most $\lg(W/w) + 2$, where w is the weight of the item and W is the total weight of the tree.

Proof. Let l be the length of the path. Since the rank of the item is $\lfloor \lg w \rfloor$, and since ranks increase by at least 1 at each step along the path toward the root, the root must have rank at least $\lfloor \lg w \rfloor + l$. By Lemma 2.8, we have

$$
\begin{aligned}
W &\geq 2^{\lfloor \lg w \rfloor + l - 1} \\
&\geq w 2^{l-2},
\end{aligned}
$$

so

$$
l \leq \lg \frac{W}{w} + 2.
$$

The cost of most operations on biased 2-3 trees needs to be amortized, but not so for the ACCESS operation (nor for the FIND operation, for which the results of this section also hold). The ACCESS operations can always be done in logarithmic time, without amortization.

Figure 2.2.
Case 1, two large nodes.

Figure 2.3.
Case 2, S is a large item node.

2.3. The JOIN operation.

The fundamental algorithm for maintaining biased 2-3 trees is called *partial* JOIN (or PJOIN for short). It takes as input two trees S and T and an integer c, and tries to JOIN the trees into a single tree with rank c (or less). If this is not possible, it returns two c-compatible trees. For simplicity, the algorithm is given assuming that S has rank at least as large as T; the other case is handled symmetrically.

Algorithm. PJOIN S and T at rank c.

Input: Two trees S and T, with ranks s and t respectively. Integer c.

Preconditions: S precedes T, and $t \leq s \leq c$.

Output: Either (i) one tree R, or (ii) two c-compatible trees R_1 and R_2.

Postconditions: Either (i) $r \leq c$, R broadens both S and T below c, and R stores the items of S and T in key order, or (ii) R_1 and R_2 are c-compatible, R_1 broadens S below c, R_2 broadens T below c, R_1 precedes R_2, and R_1 and R_2 store the items of S and T in key order.

By symmetry, assume $s \geq t$. Call a node *large* is it has rank c, otherwise call it *small* if it has rank less than c. A large node will become a major node if it is attached to a new parent with rank $c + 1$. There are five cases, depending on the configuration of S and T; the algorithm simply uses the first case that applies.

Case 1. [S and T are both large nodes.] If $s = t = c$, then return in case (ii) with $R_1 = S$ and $R_2 = T$. See Figure 2.2.

Case 2. [S is a large item node.] (At this point, $t < c$.) If S is simply an item node and $s = c$, return in case (ii) with $R_1 = S$ and $R_2 = T$. See Figure 2.3.

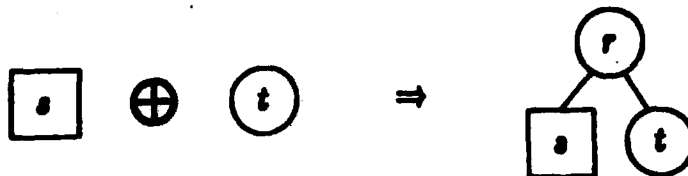


Figure 2.4.
Case 3, S is a small item node.

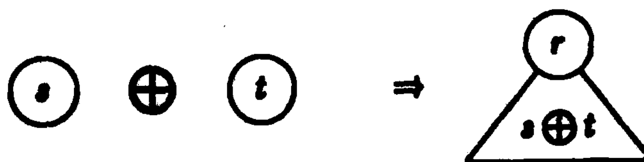


Figure 2.5.
Case 4a, two unequal small nodes.

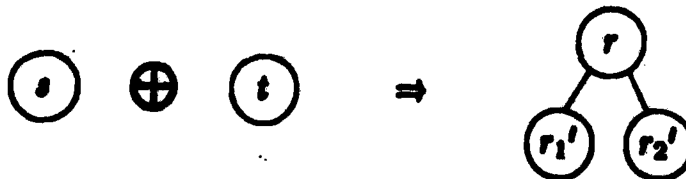


Figure 2.6.
Case 4b, two equal small nodes.

Case 3. **[S is a small item node.] (At this point, if S is an item node, then $s < c$.) If S is an item node, create a new binary non-item node R with rank $s + 1$, set $R_l \leftarrow S$ and $R_r \leftarrow T$, and return in case (i). See Figure 2.4.**

Case 4. **[S and T are both small nodes.] (At this point, S is not an item node.) If $s < c$, recursively PJOIN S and T at rank s, then distinguish two subcases:**

a) If the recursive PJOIN produced one tree R', return in case (i) with $R = R'$. See Figure 2.5.

b) Otherwise it produced two s-compatible trees R'_1 and R'_2, so create a new binary non-item node R with rank $s + 1$, set $R_l \leftarrow R'_1$ and $R_r \leftarrow R'_2$, and return in case (i). See Figure 2.6.

Case 5. **[S is a large non-item node, T is a small node.] (At this point, $t < s = c$, and**

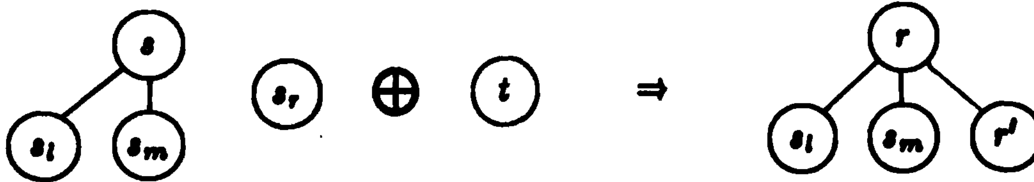


Figure 2.7.
Case 5a, S is a large non-item node.

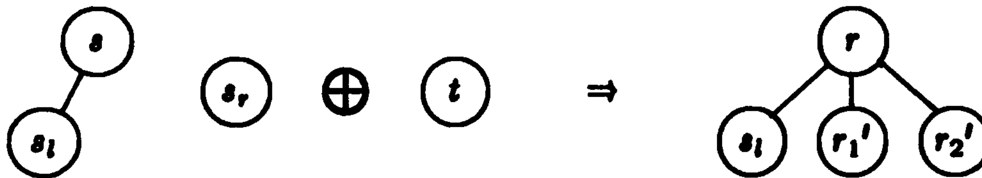


Figure 2.8.
Case 5b, S is a binary non-item node.

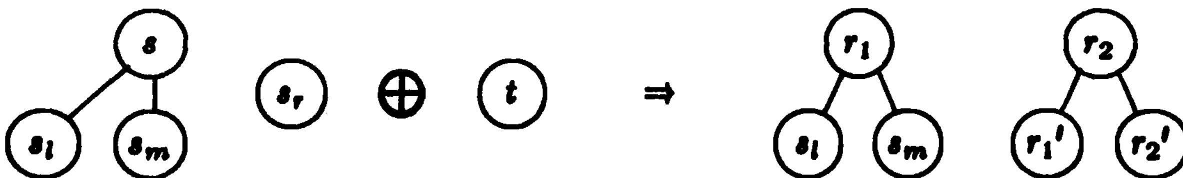


Figure 2.9.
Case 5c, S is a ternary non-item node.

S is not an item node.) Otherwise, recursively PJOIN S_r and T at rank $s-1$, then distinguish three subcases:

a) [Replace S_r.] If the recursive PJOIN returned one tree R', then set $S_r \leftarrow R'$ and return in case (i) with $R = S$. See Figure 2.7.

b) [Change binary to ternary.] Otherwise the recursive PJOIN returned two $(s-1)$-compatible trees R'_1 and R'_2. If S was binary, then set $S_m \leftarrow R'_1$, $S_r \leftarrow R'_2$, and return in case (i) with $R = S$. See Figure 2.8.

c) [Split ternary node.] If S was ternary, then set $S_r \leftarrow S_m$, and $S_m \leftarrow \emptyset$. Create a new binary non-item node R with rank s, set $R_l \leftarrow R'_1$, $R_r \leftarrow R'_2$, and return in case (ii) with $R_1 = S$ and $R_2 = R$. See Figure 2.9.

Proposition 2.11. **The PJOIN algorithm is correct.**

Proof. It suffices to show that the PJOIN algorithm produces balanced trees satisfying the postconditions, assuming the input satisfies the preconditions. It is easy to see that the items end up in the correct order, so it remains to prove that the broadening and compatibility conditions are met, and that the resulting trees are balanced. There is nothing to prove in Cases 1 and 2, because any tree broadens itself. In Case 3, S and T are s-compatible so R is a balanced tree, and since $t \leq s < r = s + 1 \leq c$, R broadens both S and T below c.

Having disposed of the base cases, we may assume that any recursive calls to PJOIN work correctly. In Case 4a, R' broadens S and T below s, hence R broadens S and T below $c > s$. Case 4b is like Case 3, noting that R broadens R_1' and R_1' broadens S_r so R broadens S below c (similarly for T). In Case 5a, S_r was $(s-1)$-compatible with its neighbor, so R' must be too, since it broadens S_r below $c-1$; thus R is a balanced tree that broadens S and its child R', and hence it also broadens T. In Case 5b, S_r was $(s-1)$-compatible with S_l, so R_1' must be too; thus R is a balanced tree that broadens S and its children, hence it also broadens T. Finally in Case 5c, R_1 and R_2 are balanced trees at rank s, since S_l and R_1' were respectively $(s-1)$-compatible with S_m and R_2'; they are c-compatible since they both have rank $s = c$, and they broaden S and T below c because S is not an item node and $t < c$.

The algorithm to JOIN S and T is now easily written as "PJOIN S and T at rank $s+1$." Since both nodes are small, the PJOIN will start in Case 3 or 4, and will return one tree R with all the desired properties.

2.4. Charging arguments.

Our analysis of the JOIN and SPLIT operations will have to take into account the property that time used by one operation might have to be charged against an earlier operation. For bookkeeping purposes, imagine allocating *poker chips* to an operation in a quantity proportional to the time we expect the operation to take. Our algorithms can then spend one poker chip to do a fixed amount of processing, usually corresponding to one level of recursion. If they ever need more chips than they were allocated, they must find the extra chips somewhere in the data structure. Conversely, if they finish the operation before running out of chips, they may leave the surplus in the data structure for future operations to use. The chips are not part of the data structure, but merely a useful fiction to help us prove the time bounds.

The following definition describes the way in which chips affect the rank of a node.

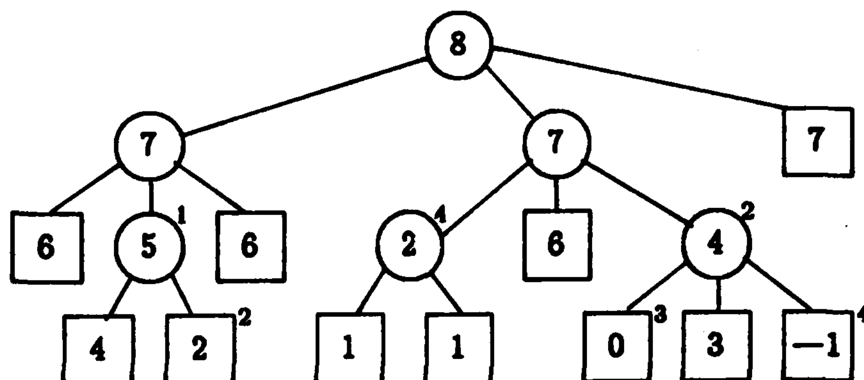


Figure 2.10.
A completely cast biased 2-3 tree.

Definition 2.12. A biased 2-3 tree with rank j is *cast to rank* k if there are $k - j$ chips piled on its root (assuming $j \leq k$).

Intuitively, the extra chips make the tree appear to have rank k, in that we can allocate chips to operations involving this tree as though it had rank k, knowing that any additional processing we may have to do because the tree really has smaller rank can be paid for using the chips in the cast.

The main difference between biased 2-3 trees and unbiased 2-3 trees is that the children of a node in a biased 2-3 tree might have different ranks, whereas the children of any node in a unbiased 2-3 tree all have the same height. We can overlook this difference if we place casts on the children to make them *appear* to have the same rank. We say a tree is *completely cast* if it satisfies the following invariant.

Chip Invariant. Each child of a node with rank k is cast to rank $k - 1$.

Figure 2.10 shows the tree of Figure 2.1, completely cast. Since the keys are the same, we have put the ranks inside the nodes; the number above a node is now the number of extra chips placed on that node as part of a cast.

Our problem now is: Given a completely cast tree (or trees), an operation to perform, and a time bound to meet, can we use a number of poker chips proportional to the time bound to pay for the operation, leaving the resulting tree (or trees) completely cast? A positive answer to this question will justify the phrase "chip invariant".

2.5. Running time of the JOIN operation.

Following the notation of the JOIN algorithm, let s and t be the ranks of the trees S and T, and let c be the rank of the cast in which the PJOIN is to occur. The JOIN algorithm runs in time proportional to $s - t + 2$ (amortized); that is to say, given $s - t + 2$ chips the algorithm can pay for the computation it does according to the accounting scheme described below. In general this scheme pays one chip for each level of recursion, except for a few special cases in which a recursive call to PJOIN is done "for free". These free calls involve only transfer of control and are not available to external users; their expense is charged to the calling procedure.

The JOIN algorithm spends one chip against the possibility that the top-level call to PJOIN turns out to be "free"; the other $s - t + 1$ chips are given to the PJOIN algorithm. The next theorem proves that the PJOIN algorithm has enough chips to spend, and thus that the JOIN algorithm runs in time $s - t + 2$ as desired.

Theorem 2.13. The PJOIN algorithm runs in time $c - t$ (amortized).

Proof. The analysis of the PJOIN operation divides into cases according to the cases taken by the algorithm. In each case we have available the $c - t$ chips allotted to the operation (which we say come from the cashier), plus perhaps some additional chips found in the tree due to the chip invariant. We spend chips in each case for three reasons: to pay for recursive calls to PJOIN, to cast resulting trees to the rank required to maintain the chip invariant, and to pay for the work actually done in the case itself (which we call overhead).

Cases 1, 2, and 4a are *not* charged overhead, and a special argument is needed to justify this. The only work done in these cases is transfer of control; they do not modify the data structure (except by recursive call). The overhead in these cases is charged *to the caller*. This may seem wrong, and indeed it would be wrong either if the caller were an external user (rather than one of the procedures responsible for maintaining the data structure) or if there were an unbounded chain of recursive calls involving only these cases. However, external users call JOIN rather than PJOIN, and the maximum length of any chain involving only these cases is 2 (Case 4 may call on Cases 1 or 2, but no other calls among these cases are possible). It seems to be necessary to resort to this piece of creative accounting in order to prove the theorem in the strong form given here.

The following tables summarize the number of chips in each case. The theorem follows by observing that the tables are correct and complete, and that the number of available chips always exceeds the number spent.

Case 1:

Given		Needed	
$c - t$	from cashier	0	
0		0	,

since $t = c$ in this case.

Case 2:

Given		Needed	
$c - t$	from cashier	$c - t$	to cast R_2

Case 3:

Given		Needed	
$c - t$	from cashier	$s - t$	to cast T
		$c - (s + 1)$	to cast R
		1	overhead
$c - t$		$c - t$	

Case 4a:

Given		Needed	
$c - t$	from cashier	$s - t$	recursive call
		$c - s$	to cast R
$c - t$		$c - t$	

Case 4b:

Given		Needed	
$c - t$	from cashier	$s - t$	recursive call
		$c - (s + 1)$	to cast R
		1	overhead
$c - t$		$c - t$	

Case 5:

Given		Needed	
$c - t$	from cashier	$(s - 1) - \min(s_r, t)$	recursive call
$(s - 1) - s_r$	on S_r	$c - s$	to cast R, R_1, and R_2
		1	overhead
$c + (s - 1) - s_r - t$		$c - \min(s_r, t)$	

We need no chips to cast R, R_1 and R_2, since $s = c$ in Case 5. There are enough chips because $(s-1) - s_r \geq 0$ (since S_r is a child of S), and because $(s-1) - t \geq 0$ (since S is large and T is small). So regardless of whether s_r or t is smaller, the "given" total is as large as the "needed" total.

We have proved that the JOIN algorithm JOINs two trees together in time proportional to the difference in their ranks. Unfortunately, this is not quite good enough to achieve logarithmic performance. It is easy to show that a tree with rank t may have total weight about 3^t. To JOIN it to a tree with rank s and total weight about 2^s takes time proportional to $s-t$, whereas we had hoped it to take time proportional to $\lg(2^s/3^t) = s-t-(\lg 3-1)t$. The extra term is very annoying, but it seems to be price we pay for discretizing the weights into integer ranks. Unbiased 2-3 trees have this same problem; they can be JOINed in time proportional to their height difference, but not in time proportional to the logarithm of their size ratio.

One mitigating fact is that the running time of JOIN satisfies a "telescoping" property. If we have a sequence $T_1,\ldots,T_n$ of trees with successively increasing ranks, JOINing them all into one large tree T takes time $O(r(T)-r(T_1))$, since the successive rank differences t_i-t_{i-1} and $t_{i+1}-t_i$ have cancelling terms. This property is often good enough for many applications; in particular, it is used implicitly in the proof that the SPLIT algorithm works in logarithmic time.

2.6. The SPLIT Operation.

The SPLIT operation takes a tree S with rank s and a key K, and returns an item node I storing the item of K, as well as two trees L and R storing the left- and right-items of K. The trees may be null if the corresponding set of items is empty. For simplicity we assume that S contains the item of K at rank k; a simple change to Case 1 of the algorithm allows the SPLIT operation to work when K is not in S, but its running time becomes more difficult to analyse due to the uncertainty about the length of the path to the "missing" key. See Section 1.2, where this problem also arises with the INSERT operation.

The SPLIT operation works as follows:

Algorithm. SPLIT S at K.

Input: A tree S with rank s. A key K.

Preconditions: S contains the item of K in an item node with rank k.

Output: A tree L (or null), an item node I, and a tree R (or null).

Postconditions: L stores the left-items of K, I stores the item of K, and R stores the right-items of K. Also $r(L) \le s$ and $r(R) \le s$.

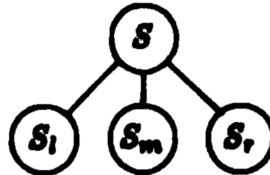


Figure 2.11.
A biased 2-3 tree about to be split.

Figure 2.12.
Case 1, S is an item node.

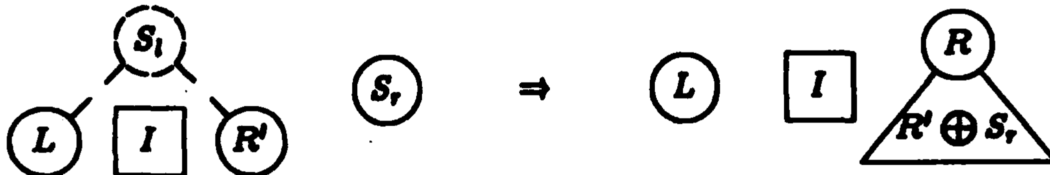


Figure 2.13.
Case 2a, split the left subtree of a binary node.

There are four cases; the algorithm uses the one that applies. Figure 2.11 shows a tree ready to be split.

Case 1. **[S is an item node.] If S is an item node, set $L \leftarrow R \leftarrow \emptyset$, $I \leftarrow S$, and return. See Figure 2.12.**

Case 2. **[K is in left subtree.] If S_l contains K, then recursively SPLIT S_l at K, obtaining L, I, and R'. Now distinguish two subcases:**

a) If S was binary, then PJOIN R' and S_r at rank s to obtain R, and return. See Figure 2.13.

b) If S was ternary, then set $S_l \leftarrow S_m$ and $S_m \leftarrow \emptyset$, PJOIN R' and S at rank s

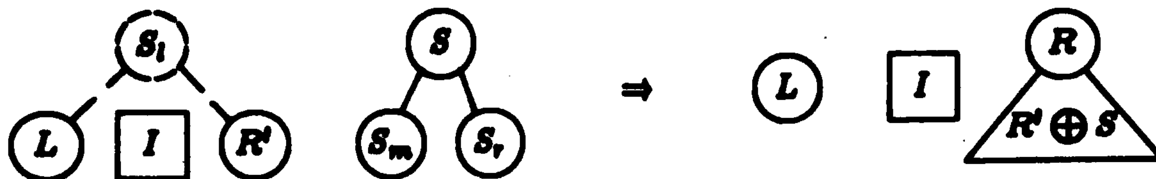


Figure 2.14.
Case 2b, split the left subtree of a ternary node.

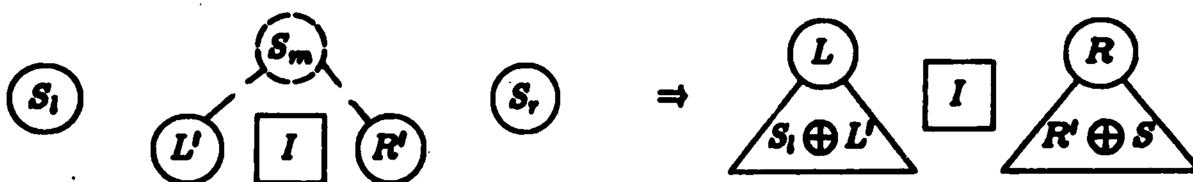


Figure 2.15.
Case 3, split the center subtree of a ternary node.

to obtain R, and return. See Figure 2.14.

Case 3. [K is in the center subtree.] If S_m contains K, then recursively SPLIT S_m at K to obtain L', I, and R'. Now PJOIN S_l and L' at rank s to obtain L, PJOIN R' and S_r at rank s to obtain R, and return. See Figure 2.15.

Case 4. [K is in the right subtree.] Case 4 is completely symmetric with Case 2. See Figure 2.16 and Figure 2.17.

Proposition 2.14. The SPLIT algorithm is correct.

Proof. Case 1 is obviously correct, since S stores the item of K by assumption. The other cases return balanced trees storing the correct sets of items and with the proper ranks, since the PJOIN algorithm is correct. The only subtle point is proving that the calls to PJOIN return *one* tree as we have assumed, in other words that the PJOIN returns in Case (i) (see Section 2.3). In Cases 2a and 3 we PJOIN (at rank s) two trees with ranks $s-1$ or less, so the PJOIN is called in Case 3 or 4 and must return one tree. In Case 2b we PJOIN (at rank s) a binary tree at rank s and a tree at rank $s-1$ or less; the PJOIN is called in Case 5b and returns one tree.

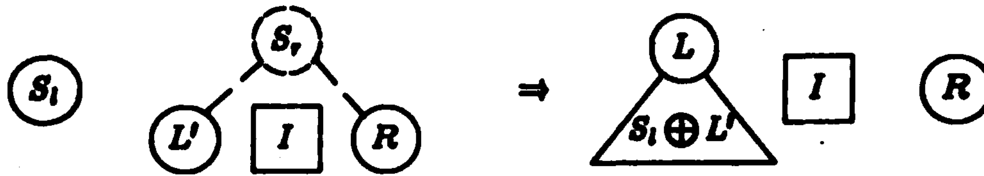


Figure 2.16.
Case 4a, split the right subtree of a binary node.

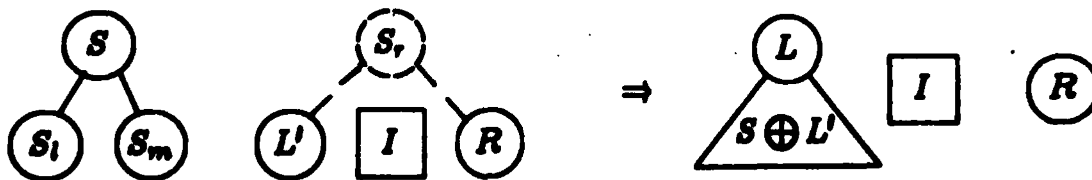


Figure 2.17.
Case 4b, splitting the right subtree of a ternary node.

2.6.1. Running time.

As usual, the running time of the SPLIT operation is analyzed using poker chips. We assume the input tree satisfies the chip invariant, and that we are supposed to cast the non-null output trees to rank c, where $c \geq s$. The recursive calls in Cases 2, 3, and 4 should all cast their results to rank $s-1$. The casts were not mentioned in the SPLIT algorithm to emphasize that they are needed only for the running time analysis.

Theorem 2.15. The SPLIT algorithm uses at most $3(c-k)+1$ chips.

Proof. As for PJOIN the proof is mostly by construction of tables showing the number of available chips and the number of chips needed. However, the arguments are slightly more complicated; after adding up the columns we will subtract a certain quantity of chips. As long as we subtract a larger number from the "Given" column than from the "Needed" column, we do not affect the result. In each case, this will be true because the subtrees involved all have rank at most $s-1$.

Case 1:

Given		Needed	
$3(c-k)+1$	from cashier	1	overhead

Case 2a:

Given		Needed	
$3(c-k)+1$	from cashier	1	overhead
$(s-1)-s_l$	on S_l	$3(s-1-k)+1$	recursive SPLIT
$(s-1)-r'$	on R'	$s-\min(r',s_r)$	PJOIN
$(s-1)-s_r$	on S_r	$c-s$	to cast R
		$c-(s-1)$	to cast L
$-(\;s-1-s_l'$		$0\;)$	
$-(\;s-r'+s-s_r-1$		$s-\min(r',s_r)\;)$	
$3c-3k)$		$2c+s-3k$	

Case 2b:

Given		Needed	
$3(c-k)+1$	from cashier	1	overhead
$(s-1)-s_l$	on S_l	$3(s-1-k)+1$	recursive SPLIT
$(s-1)-r'$	on R'	$s-r'$	PJOIN
		$c-s$	to cast R
		$c-(s-1)$	to cast L
$-(\;s-1-s_l$		$0\;)$	
$3c+s-3k-r'$		$2c+s-3k-r'$	

Case 3:

Given		Needed	
$3(c-k)+1$	from cashier	1	overhead
$(s-1)-s_m$	on S_m	$3(s-1-k)+1$	recursive SPLIT
$(s-1)-s_l$	on S_l	$s-\min(s_l,l')$	PJOIN
$(s-1)-s_r$	on S_r	$s-\min(r',s_r)$	PJOIN
$(s-1)-l'$	on L'	$c-s$	to cast L
$(s-1)-r'$	on R'	$c-s$	to cast R
$-(\;s-1-s_m$		$0\;)$	
$-(\;s-s_l+s-l'-1$		$s-\min(s_l,l')\;)$	
$-(\;s-r'+s-s_r-1$		$s-\min(r',s_r)\;)$	
$3c-3k-1$		$2c+s-3k-1$	

Since $s \leq c$, all these tables have the required property that the given column totals to more than the needed column.

2.7. Other operations.

The remaining operations on dynamic dictionaries can all be implemented for biased 2-3 trees in terms of the JOIN and SPLIT operations. They all work in logarithmic time (except for INSERT, as discussed in Section 1.2). The critical observation is that the SPLIT algorithm leaves its output cast to the rank of its input. Thus we can DELETE an item from a tree by doing a SPLIT at that item and using the chips on the two resulting trees to JOIN them together again at essentially no extra cost. Similarly, the PROMOTE and DEMOTE operations can be done by doing a SPLIT at the appropriate node, changing its weight, and reattaching the node into the two trees, using the chips from the casts to pay for the JOINS. This technique also works for INSERT, but since the original SPLIT may have to go very deep into the tree, the time for INSERT is at worst proportional to the difference in ranks between the root and the lightest node in the tree (including the new node). However, this method of doing an INSERT satisfies the goals mentioned in Section 1.2.

Algorithm. Delete K from S.

Step 1. [Detach the item.] SPLIT S at K to form L, I, and R.

Step 2. [Reassemble the tree.] JOIN L and R to form S'.

Taking the casts into account, Step 1 needs $3(s-k)+1$ chips and Step 2 needs $s-s+2$, for a total of $3(s-k)+3$. By Lemma 2.10, this is at most $3\lg(W/w)+9$.

Algorithm. DEMOTE K in S by δ.

Step 1. [Detach the item.] SPLIT S at K to form L, I, and R.

Step 2. [Change the weight.] Decrease $W(I)$ by δ (checking that $W(I) > 0$), and update $r(I)$.

Step 3. [Reattach right tree.] JOIN I and R to form R'.

Step 4. [Reassemble the tree.] JOIN L and R' to form S'.

Taking casts into account, the steps have respective costs of at most $3(s-k)+1$, 1, $s-t+2$, and $s+1-s+2$, where $t = \lfloor\lg(w-\delta)\rfloor$ is the new rank of I. Since

$$\begin{aligned} s-t = (s-k)+(k-t) &< (\lg(W/w)+2)+\lfloor\lg w\rfloor - \lfloor\lg(w-\delta)\rfloor \\ &\le \lg W - \lg w + \lg w - \lg(w-\delta) + 1 \\ &= \lg\frac{W}{w-\delta}+1, \end{aligned}$$

and since $\lg(W/w) \le \lg(W/(w-\delta))$, the total cost is at most $4\lg(W/(w-\delta))+14$, by Lemma 2.10.

Algorithm. PROMOTE K in S by δ.

Step 1. [Detach the item.] SPLIT S at K to form L, I, and R.

Step 2. [Change the weight.] Increase $W(I)$ by δ and update $r(I)$.

Step 3. [Reattach right tree.] JOIN I and R to form R'.

Step 4. [Reassemble the tree.] JOIN L and R' to form S'.

Taking casts into account, the steps have respective costs of at most $3(s-k)+1$, 1, $\max(t-s+2, s-t+2)$, and $\max(t+1-s+2, s+1-s+2)$, where $t = \lfloor \lg(w+\delta) \rfloor$ is the new rank of I. If $t \leq s$, then since $t \geq k$ we have $s - t \leq s - k$, and the whole algorithm needs at most $4\lg((W+\delta)/w) + 15$ chips, by Lemma 2.10. If $t \geq s$, then since $s \geq k > \lg w - 1$ we have $t - s \leq \lg(W+\delta) - (\lg w - 1)$, and the whole algorithm needs at most $5\lg((W+\delta)/w) + 15$ chips. In either case, the algorithm needs at most $5\lg((W+\delta)/w) + 15$ chips.

Algorithm. INSERT K into S.

Step 1. [Disassemble the tree.] SPLIT S at K to form L and R.

Step 2. [Reassemble right tree.] JOIN $I(K)$ and R to form R'.

Step 3. [Reassemble the tree.] JOIN L and R' to form S'.

The number of chips needed depends on k, the rank at which the search for K terminates (namely the rank of one of the items neighboring the gap where K belongs), and on $t = \lfloor \lg W(K) \rfloor$, the rank of the new item. The steps have respective costs at most $3(s-k)+1$, $\max(s-t+2, t-s+2)$, and $\max(s+1-s+2, t+1-s+2)$, which means the algorithm has the behavior described in Section 1.2.

Chapter 3

Biased Weight-Balanced Trees

3.1. Introduction and definitions.

Just as we can obtain an efficient dynamic weighted data structure from a 2-3 tree by relaxing the balance constraints near a heavy item node, we can obtain a second implementation of a dynamic weighted dictionary by relaxing the balance constraints in a weight-balanced tree. The resulting class of trees will have logarithmic performance, even for the JOIN operation. Thus it is possible to eliminate the discretization problem present in biased 2-3 trees, but at the price of having to store and manipulate real numbers (weights) in the internal nodes rather than integers (ranks).

Weight-balanced trees were proposed by Nievergelt and Reingold [31, 34], who originally (and more appropriately) called them trees of bounded balance. Generalizing their definition, which applied only to the case where the weight of a node was 1 + (the number of nodes in its subtree), define a weight-balanced tree with balance factor α to be a binary tree in which each node n has a weight $w(n)$ which satisfies the following constraints:

1. [*Positivity.*] The weight $w(n) > 0$ for all nodes n.
2. [*Additivity.*] If n is the parent of nodes n_1 and n_2, then $w(n) = w(n_1) + w(n_2)$.
3. [*Balance.*] If n is the parent of node n', then $w(n') \geq \alpha w(n)$.

The additivity condition implies that the weight of any node is simply the sum of the weights of the leaves in the subtree rooted at n, and in particular the weight of the root is the total weight W of all the leaves in the tree. Another way of stating the balance

condition is that the ratio $\rho(n_1) = w(n_1)/w(n_2)$ of the weights of node n_1 and its sibling n_2 satisfies $\alpha/(1-\alpha) \leq \rho(n_1) \leq (1-\alpha)/\alpha$.

We shall extend the definition of a weight-balanced tree to allow heavy item nodes. Unlike the case of biased 2-3 trees, items will be allowed to appear in internal nodes. This blurs the distinction between data-carrying nodes and bookkeeping nodes and makes the algorithms somewhat more complicated; this is required simply because a binary tree is not as flexible as a 2-3 tree.

A biased weight-balanced tree is a binary tree with two kinds of nodes, called *item* nodes and *non-item* nodes. Each item node stores one item, and they are arranged in the tree so that listing the item nodes in symmetric order gives a list of nodes sorted by ascending key value. Item nodes may have zero, one, or two children. A non-item node is simply a binary internal node; it has exactly two children.

A leaf must be an item node, but not all item nodes are leaves. Nodes whose parents are item nodes play a special role in the algorithms that manipulate biased weight-balanced trees. Such nodes are intuitively undesirable because they make the tree more complicated, yet they are necessary to maintain proper balance, so we give them a name.

Definition 3.1. A *subitem node* is a node whose parent is an item node. A *normal node* is a node whose parent is a non-item node.

Definition 3.2. The following fuctions are defined on nodes in a biased weight-balanced tree:

- The *mass* $m(n)$. If n is an item node storing item I, then $m(n) = W(I)$, the weight of the item. If n is a non-item node it has no mass, that is $m(n) = 0$.
- The *weight* $w(n)$. If n is a leaf, then $w(n) = m(n)$. The weight of internal nodes is derived from the additivity condition, stated below in Definition 3.3.
- The *balance* $\beta(n)$. Let p be the parent of n. If n is a normal node then $\beta(n) = w(n)/w(p)$. Otherwise if n is a subitem node then $\beta(n) = w(n)/m(p)$.
- The *ratio* $\rho(n)$. If n' is the sibling of n, then $\rho(n) = w(n)/w(n')$.
- The *degree* $d(n)$. If n is an item node, then $d(n)$ is the number of children it has. (We could also define $d(n) = 2$ for non-item nodes.)

When no confusion can occur, we will drop the parentheses from the balance and ratio functions, writing βn and ρn for $\beta(n)$ and $\rho(n)$.

In the algorithms and proofs of this chapter, we will usually use the name of a node n to stand for its weight $w(n)$ in contexts where a weight is expected (we will always indicate

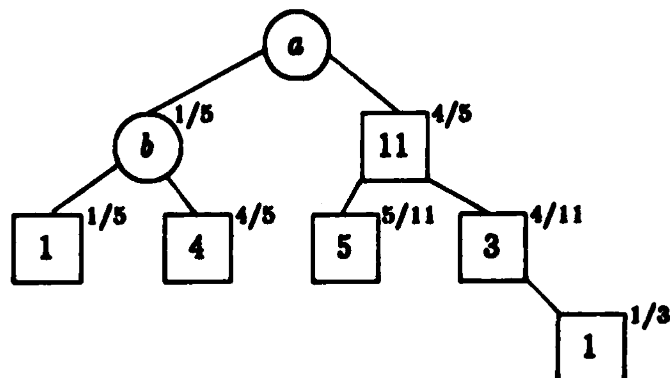


Figure 3.1.
A biased weight-balanced tree.

the mass of a node explicitly). If n is an item node, then "n" will mean $w(n)$, which is larger than $m(n)$ if n has children. In contexts where a tree is expected, "n" will mean the subtree rooted at n.

Figure 3.1 shows a biased weight-balanced tree. Squares represent item nodes and circles represent non-item nodes. The number inside an item node is its mass; the number over a node is its balance. Non-item node a has weight 25. Non-item node b has weight 5 and ratio $5/20 = 1/4$. Its sibling is an item node with weight 20 and mass 11.

Definition 3.3. A biased weight-balanced tree is in the class BWB$[\alpha, \alpha']$ if it satisfies the following three conditions:

1. [*Positivity.*] The weight $w(n) > 0$ for all nodes n.
2. [*Additivity.*] If n is the parent of n_1 and n_2, then $w(n) = m(n) + w(n_1) + w(n_2)$.
3. [*Balance.*] If n is a normal node then $\beta(n) \geq \alpha$. Otherwise, if n is a subitem node then $\beta(n) \leq \alpha'$.

Here α and α' are real numbers; we will determine feasible and desirable values for α and α' in Section 3.9. The positivity condition is included for similarity with the definition of weight-balanced trees; it can be derived from the definitions of weight and mass and from the additivity condition. A node satisfying the balance condition is called *balanced*. Thus the tree of Figure 3.1 is in the class BWB$[1/5, 5/11]$.

By the additivity condition, the weight of any node n is simply the sum of the weights of all the items stored in the subtree rooted at n. In particular, the weight of the root is the total weight W of all the items stored in the tree. We allow item nodes to have children,

but only if these children are not too heavy compared with the item; this is the import of the second part of the balance condition.

Several consequences of the balance and additivity conditions deserve to be mentioned. The next lemma gives us four ways to prove a normal node n_1 is balanced.

Lemma 3.4. If n is a non-item node with children n_1 and n_2, then the following are equivalent:

(a) $\beta(n_1) \geq \alpha$,
(b) $\beta(n_2) \leq 1 - \alpha$,
(c) $\rho(n_1) \geq \alpha/(1 - \alpha)$,
(d) $\rho(n_2) \leq (1 - \alpha)/\alpha$.

Proof.

$(a) \Rightarrow (b)$: If $n_1/n \geq \alpha$, then $n_2 = n - n_1 \leq n - \alpha n$, so $n_2/n \leq 1 - \alpha$.

$(b) \Rightarrow (c)$: If $n_2/n \leq 1 - \alpha$, then $n_1/n_2 = (n - n_2)/n_2 \geq 1/(1 - \alpha) - 1 = \alpha/(1 - \alpha)$.

$(c) \Rightarrow (d)$: This is immediate because $\rho(n_2) = 1/\rho(n_1)$.

$(d) \Rightarrow (a)$: If $n_2/n_1 \leq (1-\alpha)/\alpha$, then $n_1/n = ((n_1+n_2)/n_1)^{-1} \geq (1+(1-\alpha)/\alpha)^{-1} = \alpha$.

The following lemma relates the weight of an item node to its mass and to the weights of its children.

Lemma 3.5. If n is an item node with children (subitem nodes) n_1 and n_2, which may be null, then

(a) $$n_1 \leq \frac{\alpha'}{1 + \alpha'} n,$$

(b) $$n_1 + n_2 \leq \frac{2\alpha'}{1 + 2\alpha'} n,$$

(c) $$m(n) \geq \frac{1}{1 + d(n)\alpha'} n,$$

(d) $$m(n) \geq \frac{1}{1 + \alpha'}(n - n_1).$$

Proof. Since each child of n has weight at most $\alpha' m(n)$, Case (a) is true because

$$n_1 \leq \alpha' m(n) = \alpha'(n - n_1 - n_2) \leq \alpha'(n - n_1).$$

Case (b) is true because

$$n_1 + n_2 \leq 2\alpha' m(n) = 2\alpha'(n - (n_1 + n_2)).$$

Case (c) is true because

$$m(n) \geq n - d(n)\alpha' m(n).$$

Similarly, Case 4 is true because

$$m(n) = n - n_1 - n_2 \geq n - n_1 - \alpha' m(n).$$

3.2. More about entropy.

Given a list of weights $\mathbf{w} = (w_1, \ldots, w_k)$ with total weight $W = \sum_{1 \leq i \leq k} w_i$, recall that we defined the entropy of the list by the formula

$$H(w_1, \ldots, w_k) = \sum_{1 \leq i \leq k} \frac{w_i}{W} \lg \frac{W}{w_i}. \tag{3.1}$$

The following lemma describes what happens to the entropy when we concatenate two lists of weights.

Lemma 3.6. Suppose two lists of weights $\mathbf{w}_1 = (w_1, \ldots, w_m)$ and $\mathbf{w}_2 = (w_{m+1}, \ldots, w_n)$ are given. Let $\mathbf{w} = (w_1, \ldots, w_m, w_{m+1}, \ldots, w_n)$ be the concatenation of the two lists, and let $W_1 = \sum_{1 \leq i \leq m} w_i$, $W_2 = \sum_{m+1 \leq j \leq n} w_j$, and $W = \sum_{1 \leq i \leq n} w_i$ be the total weights of the various lists. Then

$$W\,H(\mathbf{w}) = W_1 H(\mathbf{w}_1) + W_2 H(\mathbf{w}_2) + W\,H(W_1, W_2).$$

Proof. The proof is a straightforward calculation:

$$\begin{aligned}
W_1 H(\mathbf{w}_1) + W_2 H(\mathbf{w}_2) &= -\left(W_1 \sum_{1 \leq i \leq m} \frac{w_i}{W_1} \lg \frac{w_i}{W_1} + W_2 \sum_{m+1 \leq j \leq n} \frac{w_j}{W_2} \lg \frac{w_j}{W_2}\right) \\
&= -W\left(\sum_{1 \leq i \leq m} \frac{w_i}{W} \lg \frac{w_i}{W}\frac{W}{W_1} + \sum_{m+1 \leq j \leq n} \frac{w_j}{W} \lg \frac{w_j}{W}\frac{W}{W_2}\right) \\
&= -W\left(\sum_{1 \leq i \leq n} \frac{w_i}{W} \lg \frac{w_i}{W} + \frac{W_1}{W} \lg \frac{W}{W_1} + \frac{W_2}{W} \lg \frac{W}{W_2}\right) \\
&= W\,H(\mathbf{w}) - W\,H(W_1, W_2).
\end{aligned}$$

As a notational convenience, define

$$H_\alpha = H(\alpha, 1-\alpha)$$

for real numbers α such that $0 \le \alpha \le 1$. As a function of α in this range, H_α is symmetric around $\alpha = 1/2$ and convex, since

$$\frac{d^2}{d\alpha^2} H_\alpha = \frac{-1}{\alpha(1-\alpha)\ln 2} < 0,$$

so

$$0 = H_0 = H_1 < H_\alpha < H_{1/2} = 1, \qquad \text{if } 0 < \alpha < 1.$$

Next we bound the entropy of balanced siblings in a biased weight-balanced tree.

Lemma 3.7. Let n_1 and n_2 be siblings in a biased weight-balanced tree. If n_1 and n_2 are both balanced, then $H(n_1, n_2) \ge H_\alpha$.

Proof. By definition,

$$\begin{aligned} H(n_1, n_2) &= \frac{n_1}{n} \lg \frac{n}{n_1} + \frac{n_2}{n} \lg \frac{n}{n_2} \\ &= -\big(x \lg x + (1-x) \lg(1-x)\big), \end{aligned}$$

where $x = (n_1/n)$. The lemma follows because $\alpha \le x \le 1-\alpha$, and because H is convex and symmetric around $x = 1/2$.

3.3. Path length.

Our goal in defining biased weight-balanced trees is to have the length of the path in a tree of total weight W from the root to an item node of mass w proportional to $\log(W/w)$. The following proposition shows we achieve this goal.

Proposition 3.8. Let T be a tree in the class BWB$[\alpha, \alpha']$, with total weight W. The length of the path from the root to an item node with mass w is at most $\lceil \log_\kappa (W/w) \rceil$, where

$$\kappa = \min(\frac{1}{1-\alpha}, \frac{1+\alpha'}{\alpha'}) > 1.$$

Proof. At each step along the path from n to the root, the weight goes up either by a factor of $1/(1-\alpha)$ at normal nodes, by Lemma 3.4(b), or by a factor of $(1+\alpha')/\alpha'$ at subitem nodes, by Lemma 3.5(a). Since we start with weight $\geq w$ and end with weight W, the path can have length at most $\lceil \log_\kappa (W/w) \rceil$. •

However we can say even more about the weighted average, taken over all items, of the path lengths.

Theorem 3.9. Let T be a tree in the class BWB$[\alpha, \alpha']$. Let $\mathbf{w} = (w_1, \ldots, w_k)$ be the list of item weights in T, and let $W = \sum w_i$ be the total weight. Then the total path length $L = L(T)$ satisfies

$$L \leq \frac{1}{H_\alpha} W H(\mathbf{w}) + 2\alpha' W. \tag{3.2}$$

Proof. The proof is by induction on the structure of the tree. The base case is that of a single item node. In this case the path length $L = 0$, whereas the right side of (3.2) is always positive.

Now given a non-trivial tree, let T_1 and T_2 be its left and right subtrees, with weight lists $\mathbf{w}_1$ and $\mathbf{w}_2$, total weights W_1 and W_2, and total path lengths L_1 and L_2. There are two cases, depending on whether the root is an item node or a non-item node.

If the root is a non-item node, then $W = W_1 + W_2$. Applying the theorem inductively, we find that

$$L_1 \leq \frac{1}{H_\alpha} W_1 H(\mathbf{w}_1) + 2\alpha' W_1 \qquad \text{and} \qquad L_2 \leq \frac{1}{H_\alpha} W_2 H(\mathbf{w}_2) + 2\alpha' W_2.$$

The path to any item in T can be broken down into a path within the appropriate subtree, preceded by one step from the root of T to the root of the subtree; thus the total path length of T is simply the sum of the total path lengths of the two subtrees, plus the weighted sum of 1 for each item. That is

$$\begin{aligned} L &= L_1 + L_2 + \textstyle\sum_{1 \leq i \leq k} 1 \cdot w_i \\ &= L_2 + L_2 + W \\ &\leq \frac{1}{H_\alpha}(W_1 H(\mathbf{w}_1) + W_2 H(\mathbf{w}_2)) + (1 + 2\alpha')W. \end{aligned}$$

Applying the entropy lemma (Lemma 3.6), we find that

$$L \le \frac{1}{H_\alpha} W H(\mathbf{w}) + \left(1 + 2\alpha' - \frac{1}{H_\alpha} H(W_1, W_2)\right) W.$$

The tree is balanced, so $H(W_1, W_2) \ge H_\alpha$ by Lemma 3.7, and we conclude that

$$L \le \frac{1}{H_\alpha} W H(\mathbf{w}) + 2\alpha' W.$$

If the root r is an item node, then the total weight $W = W_1 + m(r) + W_2$, although either subtree might be empty. As in the case of non-item nodes, the total path length is the sum of the path lengths in the two subtrees, plus 1 for each (weighted) item except the item at the root, that is

$$L = L_1 + L_2 + W_1 + W_2.$$

If we let $\mathbf{w}'$ be the list of weights of all items in T except the root item and apply the theorem inductively, we find that

$$\begin{aligned}
L &\le \frac{1}{H_\alpha}(W_1 H(\mathbf{w}_1) + W_2 H(\mathbf{w}_2)) + (1 + 2\alpha')(W_1 + W_2) \\
&= \frac{1}{H_\alpha}((W_1 + W_2) H(\mathbf{w}') - (W_1 + W_2) H(W_1, W_2)) + (1 + 2\alpha')(W_1 + W_2) \\
&= \frac{1}{H_\alpha}\Big(W H(\mathbf{w}) - W H(W_1 + W_2, m(r)) - m(r) H(m(r)) \\
&\qquad - (W_1 + W_2) H(W_1, W_2)\Big) + (1 + 2\alpha')(W_1 + W_2) \\
&\le \frac{1}{H_\alpha} W H(\mathbf{w}) + (1 + 2\alpha') \frac{2\alpha'}{1 + 2\alpha'} W \\
&= \frac{1}{H_\alpha} W H(\mathbf{w}) + 2\alpha' W,
\end{aligned}$$

by Lemma 3.6 (twice), the positivity of H, and Lemma 3.5(b).

To compare this theorem to Proposition 3.8, suppose all the paths were as long as the bound given there. Then the total path length would be $\sum w_i \log_\kappa(W/w_i) = W H(\mathbf{w}) / \lg \kappa$. Thus the average path is much shorter than the worst case, since

$$\lg \kappa < -\lg(1 - \alpha) < H_\alpha,$$

for $\alpha < 1/2$.

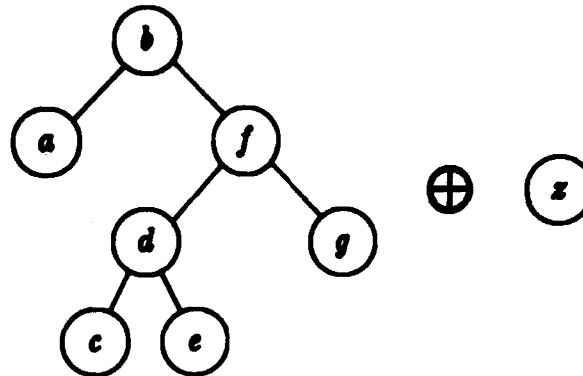


Figure 3.2.
Two biased weight-balanced trees about to be joined.

3.4. The JOIN operation.

As in the case of biased 2-3 trees, all the operations can be defined in terms of the JOIN and SPLIT operations.

The algorithm to JOIN two biased weight-balanced trees is similar in spirit to the algorithm to insert a new node in a weight-balanced tree. The basic idea is to slide the smaller tree down one side of the larger until we find the "level" where the smaller tree belongs, according to its weight. The addition of more weight within the larger tree may require us to rotate nodes along the path of insertion, or to add new non-item nodes.

The new feature of this algorithm is its accomodation of heavy item nodes. We have allowed light nodes to exist near a heavy item node as its children, unlike weight-balanced trees in which nearby nodes have roughly equal weights. But if we add enough weight to one of these children, it should move out from under the heavy item to participate in the tree on its own merit.

Here follow the details of the algorithm to JOIN two biased weight-balanced trees. We assume that the lighter tree is to be JOINed to the right of the heavier tree, as the opposite case is completely symmetric. In Figure 3.2 we have labeled the topmost nodes of the heavier tree a, b, c, d, e, f, g, and the root of the lighter tree z. Recall that we use the name of a node to refer to its weight as well; for example we express the condition that the left tree is heavier by saying $b \geq z$, instead of $w(b) \geq w(z)$. Only the ratios of the weights matter, so by rescaling we will assume that $b = 1$, and therefore that $z \leq 1$.

Algorithm. JOIN two biased weight-balanced trees.

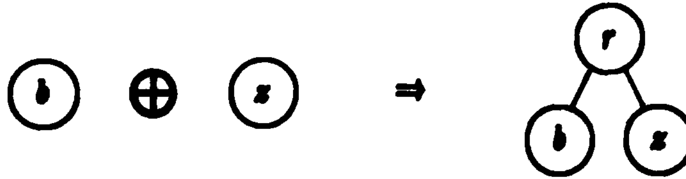


Figure 3.3.
Case 1, x is heavy.

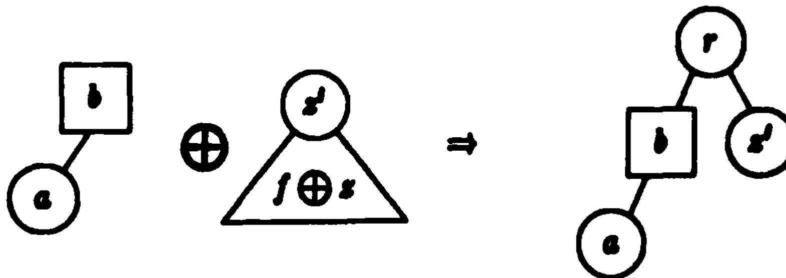


Figure 3.4.
Case 2a, b is an item node and x' is heavy.

Input: Two trees S and T, with weights b and x respectively.

Preconditions: S precedes T, $b \geq x$.

Output: One tree R.

Postconditions: R stores the items of S and T in key order.

There are seven cases, according to the weights of the nodes and the configuration of the heavier tree. The algorithm simply uses the first case that applies. We assume that α and α' satisfy certain inequalities that will be discussed later.

Case 1. [x is heavy.] If $x \geq \alpha/(1-\alpha)$, create a new node r as the root of the new tree, and attach b and x as its left and right children. See Figure 3.3.

Case 2. [b is an item node.] (At this point, $x < \alpha/(1-\alpha)$.) If b is an item node, first detach f from b and recursively JOIN f and x to form x'. (Of course, this step is omitted if node f is null.) Then distinguish two cases:

a) If $x' \geq \alpha(1-f)/(1-\alpha)$, then attach b and x' as children of a new root r. See Figure 3.4.

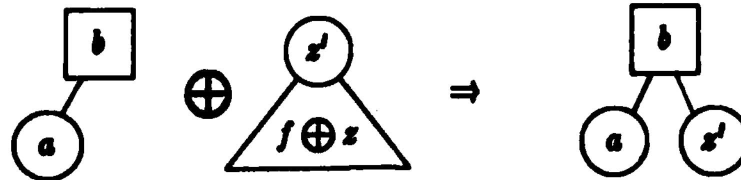


Figure 3.5.
Case 2b, b is an item node and z' is light.

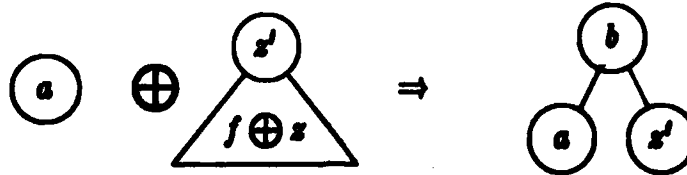


Figure 3.6.
Case 3, a is heavy.

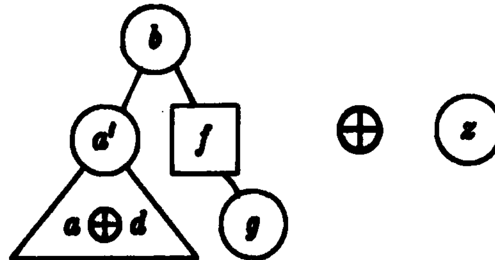


Figure 3.7.
Case 4a, a' balances item node f.

b) Otherwise attach z' as the right child of b and let $r = b$. See Figure 3.5.

Case 3. [a is heavy.] (At this point, $z < \alpha/(1-\alpha)$, and a is a normal node.) If $a \geq \alpha/(1-\alpha)$, recursively JOIN f and z to form z', and attach z' as the new right child of b. See Figure 3.6.

Case 4. [f is an item node.] (At this point, $z < \alpha/(1-\alpha)$, a is normal, and $a < \alpha/(1-\alpha)$.) If f is an item node, first detach d from f and recursively JOIN a and d to form a'. (If d is null then $a' = a$.) Then distinguish two cases:

a) If $a' \geq \alpha/(1-\alpha)$, then attach a' as the left child of the root b and go to Case 3, which will apply directly. See Figure 3.7.

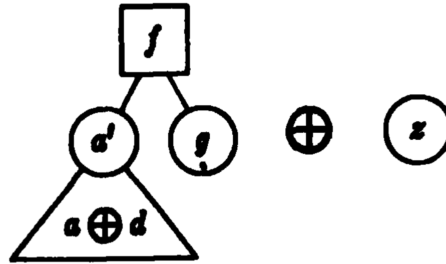


Figure 3.8.
Case 4b, a' does not balance item node f.

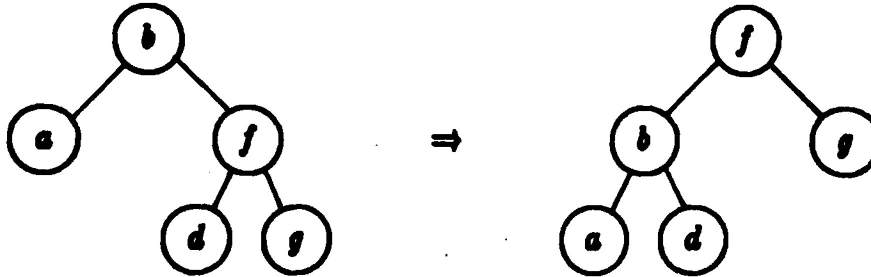


Figure 3.9.
Case 5, g is heavy (single rotation).

b) Otherwise if $a' < \alpha/(1-\alpha)$, attach a' as the left child of f (which now becomes the root), discard the old root b, and go to Case 2, which will apply directly. See Figure 3.8.

Case 5. [g is heavy.] (At this point, $z < \alpha/(1-\alpha)$, a is normal, $a < \alpha/(1-\alpha)$, and g is normal.) If $g \geq \alpha$, do a "single rotation", that is attach a and d as the children of b, and attach b and g as the children of the new root f. Then go to Case 3, which will apply directly. See Figure 3.9.

Case 6. [d is an item node.] (At this point, $z < \alpha/(1-\alpha)$, a is normal, $a < \alpha/(1-\alpha)$, g is normal, and $g < \alpha$.) If d is an item node, first detach c and e from d and recursively JOIN a and c to form a', and recursively JOIN e and g to form g'. Then attach d and g' as the children of f, and attach a' and f as the children of b. See

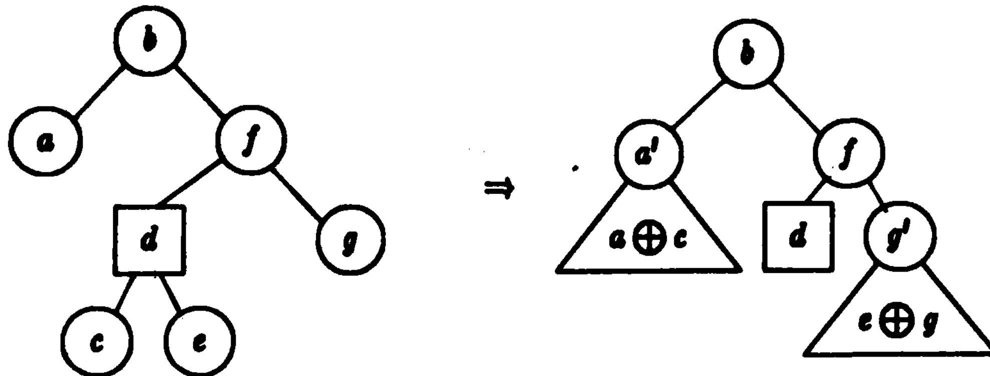


Figure 3.10.
Case 6, d is an item node.

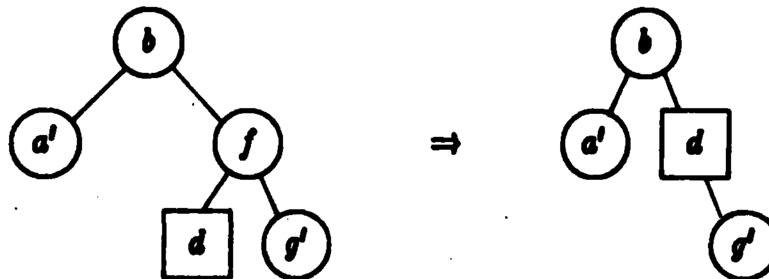


Figure 3.11.
Case 6c, d is a heavy item node.

Figure 3.10. Now distinguish several cases:

a) If $a' \geq \alpha/(1-\alpha)$ then go to Case 3, which will apply directly.

b) If $a' < \alpha/(1-\alpha)$ and $g' \geq \alpha$, then go to Case 5, which will apply directly.

c) Otherwise if $a' < \alpha/(1-\alpha)$ and $g' < \alpha$, then reattach g' as the right child of d, discard f, attach d as the right child of the root b, and go to Case 4, which will apply directly. See Figure 3.11.

Case 7. [All other cases.] (At this point, $x < \alpha/(1-\alpha)$, a is normal, $a < \alpha/(1-\alpha)$, g is normal, $g < \alpha$, and d is a non-item node.) If none of the preceding cases apply, then do a "double rotation", that is, attach a and c as children of b, attach e and g as children of f, and attach b and f as children of the new root d. Then go to

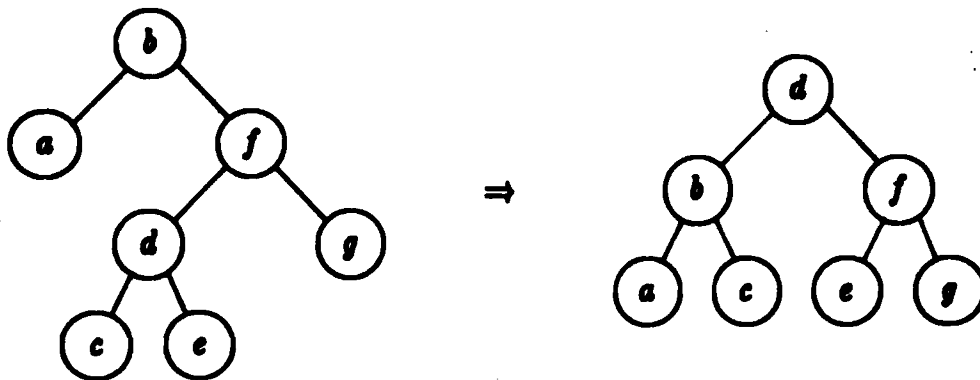


Figure 3.12.
Case 7, double rotation.

Case 3, which will apply directly. See Figure 3.12.

3.5. Correctness of the JOIN algorithm.

In proving the correctness of the JOIN algorithm, we assume several relations between α and α'. Afterwards we will collect these relations and determine feasible and desirable values for these parameters.

In proving that the transformed trees are balanced, two simple arguments occur frequently. If we add mass to the subtree rooted at some normal node n, or if we remove mass from the subtree rooted at its sibling n', the ratio $\rho(n)$ increases, so node n remains balanced. In this event we will say that node n *balances incrementally*. Or it may happen that after a transformation the subtree of n has less mass than the subtree of some different node n_1, while the subtree of n' has more mass than the subtree of n_1's sibling. In this event, $\rho(n_1) \geq \rho(n)$, so n_1 must be balanced; we will say that node n_1 *balances incrementally over* n, to emphasize which nodes are being compared. It is possible that n will be a descendant of n_1 after the transformation.

Theorem 3.10. The JOIN algorithm is correct.

Proof. The new tree R is easily seen to store the items of S and T in the correct order. It suffices to prove that R is balanced. Lemma 3.4, which gave us four ways to prove a node balanced, is the main tool. We use it tacitly throughout the proof.

Case 1. (See Figure 3.3.) Node b is balanced because

$$\beta b = \frac{b}{r} = \frac{b}{b+z} = \frac{1}{1+z} \geq \frac{1}{2} \geq \alpha,$$

which is true assuming

$$\alpha \leq \frac{1}{2}. \tag{3.3}$$

Node z is balanced because $\rho z = z \geq \alpha/(1-\alpha)$.

Case 2. The left tree is still balanced after removing node f. Suppose $z' \geq \alpha(1-f)/(1-\alpha)$, so Case 2a applies (Figure 3.4). Then z' is balanced because $\rho z' = z'/(1-f) \geq \alpha/(1-\alpha)$. Also, by Lemma 3.5(a), b is balanced because

$$\beta b = \frac{1-f}{1+z} \geq \frac{1-\alpha'/(1+\alpha')}{1+\alpha/(1-\alpha)} = \frac{1-\alpha}{1+\alpha'} \geq \alpha,$$

which is true assuming

$$\frac{1}{1+\alpha'} \geq \frac{\alpha}{1-\alpha}. \tag{3.4}$$

Otherwise in Case 2b (Figure 3.5), z' balances under b because by Lemma 3.5(c) we have

$$\beta z' = \frac{z'}{m(b)} < \frac{\alpha(1-f)/(1-\alpha)}{(1-f)/(1+\alpha')} = \frac{\alpha(1+\alpha')}{1-\alpha} \leq \alpha',$$

which is true assuming

$$\frac{\alpha'}{1+\alpha'} \geq \frac{\alpha}{1-\alpha}. \tag{3.5}$$

Case 3. (See Figure 3.6.) Node z' balances incrementally over f. Node a balances because

$$\beta a = \frac{a}{1+z} > \frac{\alpha/(1-\alpha)}{1+\alpha/(1-\alpha)} = \alpha.$$

Case 4. Before the forming of a', we knew by Lemma 3.5(a) that

$$d \leq \frac{\alpha'}{1+\alpha'} f = \frac{\alpha'}{1+\alpha'}(1-a).$$

In Case 4a (Figure 3.7), a' balances incrementally and f balances because

$$\beta f = f = 1 - (a + d) \geq 1 - a - \frac{\alpha'}{1+\alpha'}(1-a)$$
$$= \frac{1-a}{1+\alpha'} \geq \frac{(1-2\alpha)/(1-\alpha)}{1+\alpha'} \geq \alpha,$$

which is true assuming

$$\frac{1-2\alpha}{\alpha(1-\alpha)} \geq 1 + \alpha'. \tag{3.6}$$

Also, Case 3 will apply by construction since $a' \geq \alpha/(1-\alpha)$.

In Case 4b (Figure 3.8), a' balances under f because by Lemma 3.5(d),

$$\beta a' = \frac{a'}{m(f)} \leq \frac{a'}{(1-a')/(1+\alpha')} < \frac{\alpha(1+\alpha')}{1-2\alpha} \leq \alpha',$$

which is true assuming

$$\frac{\alpha'}{1+\alpha'} \geq \frac{\alpha}{1-2\alpha}. \tag{3.7}$$

Also, the new root f is an item node, so Case 2 will apply.

Case 5. (See Figure 3.9.) Node g is balanced by construction, since $\beta g = g \geq \alpha$. Node b is balanced incrementally over a. Node a is also balanced incrementally. Node d is balanced because in the original tree, before the transformation, $d \geq \alpha f = \alpha(1-a)$, so after the rotation,

$$\rho d = \frac{d}{a} \geq \frac{\alpha(1-a)}{a} > 1 - 2\alpha \geq \frac{\alpha}{1-\alpha},$$

which is true assuming

$$1 - 2\alpha \geq \frac{\alpha}{1-\alpha}. \tag{3.8}$$

Also, Case 3 will apply because, in the original tree, $g \leq (1-\alpha)f \leq (1-\alpha)^2$, so that after the rotation

$$b \geq 1 - (1-\alpha)^2 \geq \frac{\alpha}{1-\alpha},$$

which is true assuming

$$2\alpha - \alpha^2 \geq \frac{\alpha}{1-\alpha}. \tag{3.9}$$

Case 6. First we must check that the original transformation preserves balance (see Figure 3.10). Both a' and g' balance incrementally. Before the transformation we have

$$d = 1 - a - g > 1 - \frac{\alpha}{1-\alpha} - \alpha = \frac{1 - 3\alpha + \alpha^2}{1-\alpha},$$

and thus by Lemma 3.5(c),

$$m(d) \geq \frac{1 - 3\alpha + \alpha^2}{(1-\alpha)(1+2\alpha')}. \tag{3.10}$$

After the transformation d is balanced because

$$\begin{aligned} \rho d = \frac{m(d)}{e+g} &= \frac{1}{\beta e + g/m(d)} \\ &> \frac{1}{\alpha' + \alpha(1-\alpha)(1+2\alpha')/(1-3\alpha+\alpha^2)} \\ &= \frac{1 - 3\alpha + \alpha^2}{\alpha(1-\alpha) + \alpha'(1-\alpha-\alpha^2)} \\ &\geq \frac{\alpha}{1-\alpha}, \end{aligned}$$

which is true assuming

$$\frac{1 - 3\alpha + \alpha^2}{\alpha(1-\alpha) + \alpha'(1-\alpha-\alpha^2)} \geq \frac{\alpha}{1-\alpha}. \tag{3.11}$$

Furthermore, since c was balanced before the transformation, by Lemma 3.5(a) we have

$$c \leq \frac{\alpha'}{1+\alpha'} d \leq \frac{\alpha'}{1+\alpha'}(1-\alpha)(1-a),$$

so f is balanced after the transformation because

$$\begin{aligned} \beta f = f = 1 - (a + c) &\geq 1 - (a + \frac{(1-\alpha)\alpha'}{1+\alpha'}(1-a)) \\ &= (1-a)(\frac{1+\alpha\alpha'}{1+\alpha'}) \\ &> (\frac{1-2\alpha}{1-\alpha})(\frac{1+\alpha\alpha'}{1+\alpha'}) \\ &\geq \alpha, \end{aligned}$$

which is true assuming

$$\frac{1+\alpha\alpha'}{1+\alpha'} \geq \frac{\alpha(1-\alpha)}{1-2\alpha}. \qquad (3.12)$$

All we have to check in Cases 6a and 6b is that Cases 3 and 5, respectively, apply. But this is true by construction. Finally in Case 6c (Figure 3.11), the right child of the root is an item node, so Case 4 will apply; we need only show the new tree is balanced. But after this transformation a' is unchanged, so nodes a' and d still balance. Node g' balances because

$$m(d) = 1 - a' - g' > 1 - \frac{\alpha}{1-\alpha} - \alpha = \frac{1-3\alpha+\alpha^2}{1-\alpha},$$

so

$$\beta g' = \frac{g'}{m(d)} < \frac{\alpha(1-\alpha)}{1-3\alpha+\alpha^2} \leq \alpha',$$

which is true assuming

$$\alpha' \geq \frac{\alpha(1-\alpha)}{1-3\alpha+\alpha^2}. \qquad (3.13)$$

Case 7. (See Figure 3.12.) Nodes a, b, and g balance incrementally over a, a, and g respectively. Before the double rotation, we had

$$d = 1-(a+g) > 1 - (\frac{\alpha}{1-\alpha} + \alpha) = \frac{1-3\alpha+\alpha^2}{1-\alpha}.$$

Node c balances afterwards because

$$\rho c = \frac{c}{a} > \frac{\alpha d}{\alpha/(1-\alpha)} > 1-3\alpha+\alpha^2 \geq \frac{\alpha}{1-\alpha},$$

which is true assuming

$$1-3\alpha+\alpha^2 \geq \frac{\alpha}{1-\alpha}. \qquad (3.14)$$

Node e balances because

$$\rho e = \frac{e}{g} > \frac{\alpha d}{\alpha} \geq \frac{1-3\alpha+\alpha^2}{1-\alpha} > \frac{\alpha}{1-\alpha},$$

which is true assuming

$$1-3\sigma+\alpha^2 \geq \alpha. \qquad (3.15)$$

Node f balances because, before the rotation,

$$c \leq (1-\alpha)^2 f = (1-\alpha)^2(1-a);$$

so afterwards

$$\begin{aligned}\beta f = f = 1-(a+c) &\geq 1-(a+(1-\alpha)^2(1-a)) \\ &= (1-a)(2\alpha-\alpha^2) \\ &> \frac{1-2\alpha}{1-\alpha}\alpha(2-\alpha) \\ &\geq \alpha,\end{aligned}$$

which is true assuming

$$(1-2\alpha)(2-\alpha) \geq 1-\alpha. \tag{3.16}$$

Finally, Case 3 will now apply because

$$\begin{aligned}b = a+c \geq \alpha+\alpha d &> \alpha(1+\frac{1-3\alpha+\alpha^2}{1-\alpha}) \\ &= \frac{\alpha}{1-\alpha}(2-4\alpha+\alpha^2) \\ &\geq \frac{\alpha}{1-\alpha},\end{aligned}$$

which is true assuming

$$2-4\alpha+\alpha^2 \geq 1. \tag{3.17}$$

3.6. Running time of the JOIN algorithm.

As in the case of biased 2-3 trees, the running time of the JOIN algorithm for biased weight-balanced trees will be analyzed by using poker chips to account for the cost of elementary operations. We will also leave chips in the tree to be used by later operations, allowing us to amortize the cost over a sequence of operations.

Let

$$\tau(x) = \begin{cases} 1, & \text{if } 1 \leq x \leq \dfrac{1-\alpha}{\alpha}; \\ 3\lceil \log_p \dfrac{\alpha x}{1-\alpha} \rceil + 1, & \text{if } \dfrac{1-\alpha}{\alpha} < x. \end{cases} \tag{3.18}$$

This is the running-time function, defined for $x \geq 1$; in other words, $\tau(x)$ is the number of chips the cashier allots to a JOIN of two trees S and T whose weight ratio $W(S)/W(T)$ is x, assuming $W(S) \geq W(T)$. Clearly $\tau(x) = O(\log(x))$; in fact

$$\tau(x) = \frac{3}{\lg p} \lg x + O(1), \tag{3.19}$$

so we have solved the discretization problem that was the one technical flaw in biased 2-3 trees (see Section 2.5). Biased weight-balanced trees achieve "true" logarithmic performance.

The constant p is some number greater than 1. We obtain the best bound on the running time by choosing p as large as possible, however the choice of p is constrained by the choices of α and α'. These choices will be discussed in Section 3.9.

We use the notation

$$\tau(x, y) = \max(\tau(x/y), \tau(y/x))$$

to denote the running time of a JOIN of two trees of weights x and y, when we do not know *a priori* which tree is heavier; we also let $\tau(x, 0) = \tau(0, x) = 0$, since JOIN costs nothing when one operand is empty.

Since we must amortize the running time, we will need another function

$$\chi(x) = \begin{cases} 1, & \text{if } 1 \geq x \geq \dfrac{\alpha}{(1-\alpha)p}; \\ 3\left\lceil \log_p \dfrac{\alpha}{(1-\alpha)px} \right\rceil + 1, & \text{if } \dfrac{\alpha}{(1-\alpha)p} > x > 0; \end{cases} \tag{3.20}$$

defined for $0 < x \leq 1$. (Actually, we only use χ for $x \leq \alpha'$.) This function plays a role analogous to casts in a biased 2-3 tree. We say a tree is *completely cast* if it satisfies the following invariant.

Chip Invariant. Any subitem node n with balance βn has at least $\chi(\beta n)$ chips piled on

x	$\tau(x)$	$\chi(\frac{1}{x})$
$1 \leq x \leq \frac{1-\alpha}{\alpha}$	1	1
$\frac{1-\alpha}{\alpha} < x \leq \frac{1-\alpha}{\alpha}p$	4	1
$\frac{1-\alpha}{\alpha}p < x \leq \frac{1-\alpha}{\alpha}p^2$	7	4
$\frac{1-\alpha}{\alpha}p^2 < x \leq \frac{1-\alpha}{\alpha}p^3$	10	7

Table 3.1.
Some values of τ and χ.

it.

Table 3.1 shows the first few values of τ and χ. The next lemma establishes some of their elementary properties.

Lemma 3.11. As x increases, $\tau(x)$ increases and $\chi(x)$ decreases. Furthermore,

$$\begin{array}{llll}
\text{(a)} & \tau(x) & \geq \tau(x/q) + 3, & \text{if } p \leq q \text{ and } x > \frac{1-\alpha}{\alpha}. \\
\text{(b)} & \chi(x/q) & \geq \chi(x) + 3, & \text{if } p \leq q \text{ and } x < \frac{\alpha}{1-\alpha}. \\
\text{(c)} & \tau(x) & = \chi(1/x) + 3, & \text{if } x > \frac{1-\alpha}{\alpha}. \\
\text{(d)} & \tau(x) & \leq \chi(1/x) + 3, & \text{if } x \geq 1. \\
\text{(e)} & \chi(\frac{1}{qx}) & \geq \tau(x), & \text{if } p \leq q \text{ and } x \geq 1.
\end{array}$$

Proof. Monotonicity and properties (a)–(c) follow immediately from Equation (3.19) and Equation (3.20). Property (d) follows from (c), since $\tau(x) = \chi(1/x)$ for $1 \leq x \leq (1-\alpha)/\alpha$. For property (e) there are two cases. If $x \leq (1-\alpha)/\alpha$, then $\tau(x) = 1 \leq \chi(1/(qx))$. If $x > (1-\alpha)/\alpha$, then $\chi(1/(qx)) = \tau(qx) - 3 \geq \tau(x)$, by properties (c) and (a).

Our goal in this section is to prove the following theorem.

Theorem 3.12. The JOIN algorithm uses $\tau(W(S), W(T))$ chips to JOIN S and T, leaving the result completely cast, assuming both S and T were completely cast, provided that certain relations among α, α', and p hold. (These relations will be discussed in Section 3.9.)

Proof. Under the assumptions of the algorithm, the weight ratio x is $1/z$, and we must show that $\tau(x)$ chips suffice. Before we begin the case analysis, a word about overhead and certain subitem nodes. The algorithm factors rather fortuitously into the "odd" cases (1, 3, 5, and 7), and the "even" cases (2, 4, and 6). In the odd cases, we do a bounded amount of work rotating S before finally doing useful work in Case 1 or 3. One chip will pay for all this work (except the recursive calls); in other words, the overhead for Cases 5 and 7 is paid by Case 3.

The even cases are a little more complicated. They do a bounded amount of rebalancing (modulo recursive calls) before ending up in Case 2 or 3. However, the chips needed to satisfy the chip invariant on node a' in Case 4b and on node g' in Case 6c must come out of the supply from the cashier. Assuming that one chip apiece is needed for these nodes, by the time we get to Case 2 or 3 our supply of chips has possibly shrunk from $\tau(x)$ to $\tau(x) - 2$. So the overhead for the even cases is paid by Case 2 or 3, including two extra chips for casting α' and g'. Of course, we must prove that one chip apiece is enough to cast them.

Now we will go through the JOIN algorithm case by case to prove each case has enough chips, assuming certain relations among α, α', and p. For brevity we say that a table (or a line of a table) is *good* if the sum of the "Given" column exceeds the sum of the "Needed" column.

Case 1.

Given		Needed	
$\tau(1/z)$	from cashier	1	overhead

Since $x = 1/z \geq 1$, the table is good because $\tau(x) \geq 1$ for $x \geq 1$.

Case 2.

Given		Needed	
$\tau(1/z)$	from cashier	$1 + 2$	overhead
$\chi(\beta f)$	on f	$\tau(f, z)$	recursive call
		$\chi(\beta z')$	to cast z' (if necessary)

If $f \geq x$, then let $x = 1/z$; since $z' \geq f$, then $\beta z' \geq \beta f$ and so $\chi(\beta f) \geq \chi(\beta z')$. Since

$f \le \alpha'/(1+\alpha')$ by Lemma 3.5(a), and since $z < \alpha/(1-\alpha)$, the table is good if

$$\tau(x) \ge 3 + \tau(\frac{\alpha'}{1+\alpha'}x), \qquad \text{for } x > \frac{1-\alpha}{\alpha},$$

which is true by Lemma 3.11(a), assuming

$$p \le \frac{1+\alpha'}{\alpha'}. \tag{3.21}$$

Otherwise if $z \ge f$, let $x = 1/z$ and let $y = z/f$. Since $z < \alpha/(1-\alpha)$ and $a \le \alpha' m(b)$, we have $m(b) = 1 - a - f > 1 - \alpha' m(b) - \alpha/(1-\alpha)$, so

$$\frac{z}{m(b)} < \frac{\alpha}{1-\alpha}\frac{(1-\alpha)(1+\alpha')}{1-2\alpha}.$$

Since $\beta f = z/(m(b)y)$ and $\beta z' = (f+z)/(1-a-f) \ge z = 1/x$, the second line is good if

$$\chi(\frac{\alpha(1+\alpha')}{1-2\alpha}\frac{1}{y}) \ge \tau(y), \qquad \text{for } y \ge 1;$$

the table is good if, in addition,

$$\tau(x) \ge 3 + \chi(\frac{1}{x}), \qquad \text{for } x > \frac{1-\alpha}{\alpha}.$$

The second inequality is true by Lemma 3.11(c); the first only applies if $f > 0$ (if $f = 0$ the second line of the table is vacuously good), and it is true by Lemma 3.11(c), assuming

$$p \le \frac{1-2\alpha}{\alpha}\frac{1}{1+\alpha'}. \tag{3.22}$$

Case 3.

Given		Needed	
$\tau(1/z)$	from cashier	$1+2$	overhead
		$\tau(f,z)$	recursive call

If $f \geq z$, then let $x = 1/z$; since $f = 1 - a \leq (1-2\alpha)/(1-\alpha)$ and $z < \alpha/(1-\alpha)$, the table is good if

$$\tau(x) \geq 3 + \tau(\frac{1-2\alpha}{1-\alpha}x), \qquad \text{for } x > \frac{1-\alpha}{\alpha},$$

which is true by Lemma 3.11(a), assuming

$$p \leq \frac{1-\alpha}{1-2\alpha}. \tag{3.23}$$

Otherwise if $z \geq f$, then since $z < \alpha/(1-\alpha)$, so $\tau(1/z) \geq 4$, and since $f \geq \alpha$ by balance, the table is good if

$$4 \geq 3 + \tau(\frac{1}{1-\alpha}),$$

which is true by Equation (3.19), assuming

$$\frac{1}{1-\alpha} \leq \frac{1-\alpha}{\alpha}. \tag{3.24}$$

Case 4.

Given		Needed	
$\chi(\beta d)$	on d	$\tau(a,d)$	recursive call
1	from overhead	$\chi(\beta a')$	to cast a' (if necessary)

First note that if Case 4b applies, then $1 \geq \chi(\beta a')$, since $\beta a' = (a+d)/(f-d-g) \geq a/f = \rho a \geq \alpha/(1-\alpha)$, so the second line is good. If $d = 0$, the first line is vacuously good. Now if $a \geq d > 0$, then let $x = a/d$; since $\beta d = a/(m(f)x)$, and since $a/m(f) \leq a(1+2\alpha')/f = a(1+2\alpha')/(1-a) < \alpha(1+2\alpha')/(1-2\alpha)$ by Lemma 3.5(c), the first line is good if

$$\chi(\frac{\alpha(1+2\alpha')}{1-2\alpha}\frac{1}{x}) \geq \tau(x), \qquad \text{for } x \geq 1,$$

which is true by Lemma 3.11(e), assuming

$$p \leq \frac{1-2\alpha}{\alpha}\frac{1}{1+2\alpha'}. \tag{3.25}$$

Otherwise if $d \geq a$, then since $a \geq \alpha$, $\beta d \leq \alpha'$, and $d \leq \alpha'(1-a)/(1+\alpha')$, by balance and Lemma 3.5(a), the first line is good because

$$\tau(\frac{d}{a}) \leq \tau(\frac{\alpha'}{1+\alpha'}\frac{1-a}{a}) \leq \tau(\frac{1-a}{a}) \leq \tau(\frac{1-\alpha}{\alpha}) = 1 \leq \chi(\beta d).$$

<u>Case 5</u>. This case is paid for by Case 3.

<u>Case 6</u>.

<u>Given</u>		<u>Needed</u>	
$\chi(\beta c)$	on c	$\tau(a,c)$	recursive call
$\chi(\beta e)$	on e	$\tau(e,g)$	recursive call
1	from overhead	$\chi(\beta g')$	to cast g' (if necessary)

First note that if Case 6c applies, then $1 \geq \chi(\beta g')$, since $\beta g' \geq g/m(d) \geq g/d = \rho g \geq \alpha/(1-\alpha)$ by balance, so the third line is good. If $c = 0$, the first line is vacuously good. Now if $a \geq c > 0$, then let $x = a/c$; since $\beta c = a/(m(d)x)$, and since

$$\frac{a}{m(d)} < \frac{\alpha}{1-\alpha}\frac{(1-\alpha)(1+2\alpha')}{1-3\alpha+\alpha^2}$$

by Equation (3.10), the first line is good if

$$\chi(\frac{\alpha(1+2\alpha')}{1-3\alpha+\alpha^2}\frac{1}{x}) \geq \tau(x), \qquad \text{for } x \geq 1,$$

which is true by Lemma 3.11(e), assuming

$$p \leq \frac{1-3\alpha+\alpha^2}{\alpha}\frac{1}{1+2\alpha'}. \tag{3.26}$$

If $c \geq a$, then since $a \geq \alpha$ and $c \leq \alpha'(1-\alpha)^2/(1+\alpha')$ by balance and Lemma 3.5(a), the first line is good because

$$\tau(\frac{c}{a}) \leq \tau(\frac{\alpha'}{1+\alpha'}\frac{(1-\alpha)^2}{\alpha}) < \tau(\frac{1-\alpha}{\alpha}) = 1 \leq \chi(\beta c).$$

If $e = 0$, the second line is vacuously good. Next, if $g \geq e > 0$, then let $x = g/e$; since $\beta e = g/(m(d)x)$, and since

$$\frac{g}{m(d)} < \alpha\frac{(1-\alpha)(1+2\alpha')}{1-3\alpha+\alpha^2}$$

by Equation (3.10), the second line is good if

$$\chi(\frac{\alpha(1-\alpha)(1+2\alpha')}{1-3\alpha+\alpha^2}\frac{1}{x}) \geq \tau(x), \qquad \text{for } x \geq 1,$$

which is true by Lemma 3.11(e), assuming

$$p \leq \frac{1-3\alpha+\alpha^2}{\alpha(1-\alpha)}\frac{1}{1+2\alpha'}. \tag{3.27}$$

Finally, if $e \geq g$, then since $g \geq \alpha f$ and $e \leq \alpha'(1-\alpha)f/(1+\alpha')$ by balance and Lemma 3.5(a), the second line is good because

$$\tau(\frac{e}{g}) \leq \tau(\frac{\alpha'}{1+\alpha'}\frac{1-\alpha}{\alpha}) \leq \tau(\frac{1-\alpha}{\alpha}) = 1 \leq \chi(\beta e).$$

Case 7. This case is paid for by Case 3.

This completes the proof.

3.7. The SPLIT operation.

The SPLIT operation takes a tree S and a key K, and returns an item node I storing the item of K, as well as two trees L and R storing the left- and right-items of K. The trees may be null if the corresponding set of items is empty. For simplicity we assume that S contains the item of K at rank k; a simple change to Case 3 of the algorithm allows the SPLIT operation to work when K is not in S, but its running time becomes more difficult to analyze due to the uncertainty about the length of the path to the "missing" key. See Section 1.2, where this problem also arises with the INSERT operation.

The SPLIT operation works as follows:

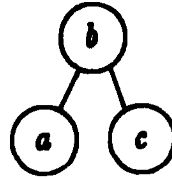


Figure 3.13.
A biased weight-balanced tree about to be split.

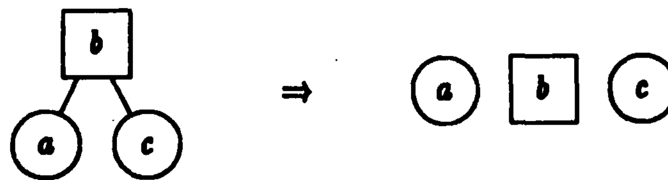


Figure 3.14.
Case 1, b is an item node containing the search key.

Algorithm. SPLIT S at K.

Input: A tree S and a key K.

Preconditions: S contains the item of K. $W(S) = W$, $W(K) = w$.

Output: A tree L (or null), an item node I, and a tree R (or null).

Postconditions: L stores the left-items of K, I stores the item of K, and R stores the right-items of K. There are no subitem nodes under I.

There are three cases; the algorithm uses the one that applies. Figure 3.13 shows a biased weight-balanced tree about to be SPLIT.

Case 1. [b is an item node whose key is K.] Detach the subtrees a and c from b, set $L \leftarrow a$, $I \leftarrow b$, and $R \leftarrow c$, and return. See Figure 3.14.

Case 2. [b is a non-item node.] The search key K occurs in one of the two subtrees a and c; assume that it occurs in c (the other case is symmetric). Recursively SPLIT the subtree c at K to form L', I, and R; JOIN a and L' to form L; and return.

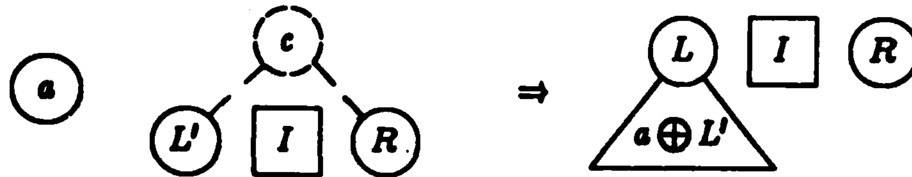


Figure 3.15.
Case 2, b is a non-item node.

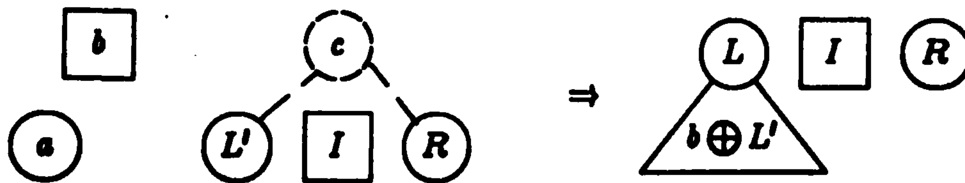


Figure 3.16.
Case 3, b is an item node.

See Figure 3.15.

Case 3. [b is an item node whose key is not K.] The search key K occurs in one of the two subtrees a and c; assume that it occurs in c (the other case is symmetric). Detach c from b and recursively SPLIT c to form L', I, and R; JOIN b and L' to form L; and return. See Figure 3.16.

Proposition 3.13. The SPLIT algorithm is correct.

Proof. Immediate from the correctness of the JOIN algorithm.

3.8. Analysis of the SPLIT algorithm.

As usual, we account for time spent in the SPLIT algorithm with poker chips. Let

$$\lambda = \max(3\lceil \log_p \frac{1-\alpha}{\alpha} \rceil + 2, 3\lceil \log_p \frac{1}{\alpha} \rceil) \tag{3.28}$$

be a constant (depending on α and p). Then let

$$\sigma(x) = \begin{cases} 7, & \text{if } x = 1; \\ (2+3\lambda)\lceil \log_{p'} x \rceil + (5-3\lambda), & \text{if } x > 1. \end{cases} \tag{3.29}$$

x	$\sigma(x)$
$1 \le x \le p'$	7
$p' < x \le p'^2$	$9 + 3\lambda$
$p'^2 < x \le p'^3$	$11 + 6\lambda$
$p'^3 < x \le p'^4$	$13 + 9\lambda$

Table 3.2.
Some values of σ.

This is the running-time function; that is, $\sigma(x)$ is the number of chips the cashier allots to SPLIT a tree of weight W at a node of weight w, where the weight ratio W/w is x. The constant p' is some number greater than 1 whose value will be determined in Section 3.9. Table 3.2 gives the first few values of σ.

Lemma 3.14. The function σ is monotonically increasing. Furthermore,

$$\sigma(qx) \ge \sigma(x) + (2 + 3\lambda), \qquad \text{if } p' \le q \text{ and } x > 1.$$

Proof. Immediate from Equation (3.29).

After we SPLIT a biased 2-3 tree, we left the resulting trees cast to the rank of the original tree; the chips in the cast helped pay to JOIN the trees to other trees in the forest. The analogous idea for biased weight-balanced trees is to leave the resulting trees completely cast (so that their subitem nodes have chips), and to add enough chips to the roots to satisfy the following invariant.

SPLIT Invariant. After a tree of weight W is SPLIT at an item with weight w, yielding two trees L and R with weights l and r, there will be $\tau(W/l)$ chips on the root of L and $\tau(W/r)$ chips on the root of R.

Here τ is the function describing the running time of the JOIN algorithm, defined in Equation (3.19). We will need two more simple facts about τ.

Lemma 3.15. If x and y are at least 1, then

$$\tau(xy) \leq \tau(x) + \tau(y) + \lambda - 3, \tag{3.30}$$

and

$$\tau(x/\alpha) \leq \tau(x) + \lambda. \tag{3.31}$$

Proof. The proof uses an even simpler fact, namely $\lceil a+b \rceil \leq \lceil a \rceil + \lceil b \rceil$. If $x, y \leq (1-\alpha)/\alpha$, then $\tau(x) = \tau(y) = 1$, so

$$\tau(xy) \leq \tau((\frac{1-\alpha}{\alpha})^2) = 3\lceil \log_p \frac{1-\alpha}{\alpha} \rceil + 1 \leq \lambda - 1 = \tau(x) + \tau(y) + \lambda - 3;$$

if $x \leq (1-\alpha)/\alpha < y$, then $\tau(x) = 1$, so

$$\tau(xy) = 3\lceil \log_p \frac{\alpha y}{1-\alpha} + \log_p x \rceil + 1 \leq \tau(y) + (\lambda - 2) + (\tau(x) - 1);$$

and if $x, y > (1-\alpha)/\alpha$, then

$$\tau(xy) = 3\left\lceil \log_p \frac{\alpha x}{1-\alpha} + \log_p \frac{\alpha y}{1-\alpha} + \log_p \frac{1-\alpha}{\alpha} \right\rceil + 1 \leq \tau(x) + (\tau(y) - 1) + (\lambda - 2).$$

This proves Equation (3.30). Finally,

$$\tau(x/\alpha) = 3\left\lceil \log_p \frac{\alpha x}{1-\alpha} + \log_p \frac{1}{\alpha} \right\rceil + 1 \leq \tau(x) + \lambda.$$

Theorem 3.16. The SPLIT algorithm uses $\sigma(W/w)$ chips to SPLIT S at K, and to leave its result satisfying both the Chip and Split Invariants, assuming that its input satisfied the Chip Invariant.

Proof. By normalising, we may assume that $W = b = 1$ (see Figure 3.13). As usual, we will go through the algorithm case by case, deriving chip tables and proving that they are good.

Case 1.

Given		Needed	
$\sigma(1/w)$	from cashier	1	overhead
$\chi(\beta a)$	on a	$\tau(1/a)$	to cast a
$\chi(\beta c)$	on c	$\tau(1/c)$	to cast c

(See Figure 3.14.) Let $x = 1/w = 1/m(b)$. Now $\tau(1/a) \le \chi(a)+3 \le \chi(ax)+3 = \chi(\beta a)+3$ by Lemma 3.11(d); similarly $\tau(1/c) \le \chi(\beta c)+3$, so the table is good because $\sigma(x) \ge 7$ for $x \ge 1$.

Case 2.

Given		Needed	
$\sigma(1/w)$	from cashier	1	overhead
$\tau(c/l')$	on L'	$\sigma(c/w)$	recursive call
$\tau(c/r)$	on R	$\tau(a,l')$	JOIN
		$\tau(1/(a+l'))$	to cast L
		$\tau(1/r)$	to cast R

(See Figure 3.15.) Let $x = c/w$. Since $c \ge \alpha$, by Equation (3.31) we have $\tau(1/r) \le \tau(c/r) + \lambda$. Also $\tau(1/(a+l')) \le \tau(1/a) \le \tau(1/\alpha) \le 1 + \lambda$, by Equation (3.31). Now if $a \ge l'$, then by balance and Equation (3.31) we have

$$\tau(c/l') \le \tau(\frac{1-\alpha}{\alpha}\frac{a}{l'}) \le \tau(\frac{1}{\alpha}\frac{a}{l'}) \le \tau(a/l') + \lambda;$$

and if $l' > a$ then $\tau(c/l') = \tau(a/l') = 1$ because $a < l' < c$ implies that c/l' and l'/a are both less than $c/a = \rho c \le (1-\alpha)/\alpha$. By these facts and the fact that $c \le 1-\alpha$, the table is good if

$$\sigma(\frac{1}{1-\alpha}x) \ge \sigma(x) + 2 + 3\lambda,$$

which is true by Lemma 3.14, assuming

$$p' \le \frac{1}{1-\alpha}. \tag{3.32}$$

Case 3.

Given		Needed	
$\sigma(1/w)$	from cashier	1	overhead
$\chi(\beta c)$	on c	$\sigma(c/w)$	recursive call
$\tau(c/l')$	on L'	$\tau(1-c, l')$	JOIN
$\tau(c/r)$	on R	$\tau(1/(1-c+l'))$	to cast L
		$\tau(1/r)$	to cast R

(See Figure 3.16.) Let $x = c/w$. As in Case 2, $\tau(1/r) \leq \tau(c/r) + \lambda$. Since $c \leq \alpha'/(1+\alpha')$ by Lemma 3.5(a), $\tau(1/(1-c+l')) \leq \tau(1/(1-c)) \leq \tau(1+\alpha') = 1$, assuming

$$1 + \alpha' \leq \frac{1-\alpha}{\alpha}. \tag{3.33}$$

Now assuming

$$\frac{\alpha'}{1+\alpha'} \leq \frac{1}{2}, \tag{3.34}$$

then $c \leq 1/2$, so $1 - c \geq c > l'$ and the JOIN has L' as the smaller tree. But now

$$\begin{aligned}
\tau(\frac{1-c}{l'}) &= \tau(\frac{1-c}{m(b)}\frac{m(b)}{l'}) \\
&\leq \tau((1+\alpha')\frac{m(b)}{l'}) \\
&\leq \tau(\frac{1}{\alpha}\frac{m(b)}{l'}) \\
&\leq \tau(\frac{m(b)}{c}\frac{c}{l'}) + \lambda \\
&\leq \tau(\frac{1}{\beta c}) + \tau(\frac{c}{l'}) + 2\lambda - 3 \\
&\leq \chi(\beta c) + \tau(\frac{c}{l'}) + 2\lambda
\end{aligned}$$

by Lemma 3.5(d), Equation (3.33), Equation (3.31), Equation (3.30), and Lemma 3.11(c). Thus the table is good if

$$\sigma(\frac{1+\alpha'}{\alpha'}x) \geq \sigma(x) + 3\lambda + 2,$$

which is true by Lemma 3.14, assuming

$$p' \leq \frac{1+\alpha'}{\alpha'}. \tag{3.35}$$

This completes the proof.

3.9. Tuning the algorithms.

Recall the running times for the various operations:

ACCESS (worst case): $\frac{1}{\lg \kappa} \lg \frac{W}{w} + O(1),$

ACCESS (average case): $\frac{1}{H_\alpha} \lg \frac{W}{w} + O(1),$

JOIN: $\frac{3}{\lg p} \lg \frac{W}{w} + O(1),$

SPLIT: $\frac{2 + 3\lambda}{\lg p'} \lg \frac{W}{w} + O(1),$

where

$$\kappa = \min\left(\frac{1}{1-\alpha}, \frac{1+\alpha'}{\alpha'}\right) \text{ and } \lambda = \max\left(3\left\lceil \log_p \frac{1-\alpha}{\alpha} \right\rceil + 2, 3\left\lceil \log_p \frac{1}{\alpha} \right\rceil\right).$$

(1) $\alpha \le \frac{1}{2}$

(2) $\frac{1}{1+\alpha'} \ge \frac{\alpha}{1-\alpha}$

(3) $\frac{\alpha'}{1+\alpha'} \ge \frac{\alpha}{1-\alpha}$

(4) $\frac{1-2\alpha}{\alpha(1-\alpha)} \ge 1+\alpha'$

(5) $\frac{\alpha'}{1+\alpha'} \ge \frac{\alpha}{1-2\alpha}$

(6) $1-2\alpha \ge \frac{\alpha}{1-\alpha}$

(7) $2\alpha-\alpha^2 \ge \frac{\alpha}{1-\alpha}$

(8) $\frac{1-3\alpha+\alpha^2}{\alpha(1-\alpha)+\alpha'(1-\alpha-\alpha^2)} \ge \frac{\alpha}{1-\alpha}$

(9) $\frac{1+\alpha\alpha'}{1+\alpha'} \ge \frac{\alpha(1-\alpha)}{1-2\alpha}$

(10) $\alpha' \ge \frac{\alpha(1-\alpha)}{1-3\alpha+\alpha^2}$

(11) $1-3\alpha+\alpha^2 \ge \frac{\alpha}{1-\alpha}$

(12) $1-3\alpha+\alpha^2 \ge \alpha$

(13) $(1-2\alpha)(2-\alpha) \ge 1-\alpha$

(14) $2-4\alpha+\alpha^2 \ge 1$

(15) $p \le \frac{1+\alpha'}{\alpha'}$

(16) $p \le \frac{1-2\alpha}{\alpha}\frac{1}{1+\alpha'}$

(17) $p \le \frac{1-\alpha}{1-2\alpha}$

(18) $\frac{1}{1-\alpha} \le \frac{1-\alpha}{\alpha}$

(19) $p \le \frac{1-2\alpha}{\alpha}\frac{1}{1+2\alpha'}$

(20) $p \le \frac{1-3\alpha+\alpha^2}{\alpha}\frac{1}{1+2\alpha'}$

(21) $p \le \frac{1-3\alpha+\alpha^2}{\alpha(1-\alpha)}\frac{1}{1+2\alpha'}$

(22) $p' \le \frac{1}{1-\alpha}$

(23) $1+\alpha' \le \frac{1-\alpha}{\alpha}$

(24) $\frac{\alpha'}{1+\alpha'} \le \frac{1}{2}$

(25) $p' \le \frac{1+\alpha'}{\alpha'}$

Table 3.3.
Relations among α, α', p, and p'.

We should choose the parameters α, α', p, and p' to minimize the leading coefficients, subject to the constraints given by Equations (3.3), (3.4), (3.5), (3.6), (3.7), (3.8), (3.9), (3.11), (3.12), (3.13), (3.14), (3.15), (3.16), (3.17), (3.21), (3.22), (3.23), (3.24), (3.25), (3.26), (3.27), (3.32), (3.33), (3.34), and (3.35). These equations are collected in Table 3.3.

In Table 3.4 we have simplified the inequalities of Table 3.3 and set $\alpha' = \alpha/(1-3\alpha)$ in the inequailities involving p and p'. This will be justified very soon.

In light of (1), it is tediously checked that inequalities (6), (7), (12), (13), (14), and (18) are weaker than (11). This means we are required to choose $\alpha \le \alpha_0$, where $\alpha_0 = 0.24512^+$ is the root of $1-5\alpha+4\alpha^2-\alpha^3$. Assuming this, we check that (2), (4), (8), (9), and (23)

(1)	$\alpha \le \frac{1}{2}$	(14)	$1 - 4\alpha + \alpha^2 \ge 0$
(2)	$\alpha' \le \frac{1-2\alpha}{\alpha}$	(15)	$p \le \frac{1-2\alpha}{\alpha}$
(3)	$\alpha' \ge \frac{\alpha}{1-2\alpha}$	(16)	$p \le \frac{1-3\alpha}{\alpha}$
(4)	$\alpha' \le \frac{1-3\alpha+\alpha^2}{\alpha(1-\alpha)}$	(17)	$p \le \frac{1-\alpha}{1-2\alpha}$
(5)	$\alpha' \ge \frac{\alpha}{1-3\alpha}$	(18)	$1 - 3\alpha + \alpha^3 \ge 0$
(6)	$1 - 4\alpha + 2\alpha^2 \ge 0$	(19)	$p \le \frac{1-5\alpha+6\alpha^2}{\alpha(1-\alpha)}$
(7)	$2 - 3\alpha + \alpha^2 \ge 0$	(20)	$p \le \frac{1-6\alpha+10\alpha^2-3\alpha^3}{\alpha(1-\alpha)}$
(8)	$\alpha' \le \frac{1-4\alpha+3\alpha^2}{\alpha(1-\alpha-\alpha^2)}$	(21)	$p \le \frac{1-6\alpha+10\alpha^2-3\alpha^3}{\alpha(1-\alpha)^2}$
(9)	$\alpha' \le \frac{1-3\alpha+\alpha^2}{\alpha^2}$	(22)	$p' \le \frac{1}{1-\alpha}$
(10)	$\alpha' \ge \frac{\alpha(1-\alpha)}{1-3\alpha+\alpha^2}$	(23)	$\alpha' \le \frac{1-2\alpha}{\alpha}$
(11)	$1 - 5\alpha + 4\alpha^2 - \alpha^3 \ge 0$	(24)	$\alpha' \le 1$
(12)	$1 - 4\alpha + \alpha^2 \ge 0$	(25)	$p' \le \frac{1-2\alpha}{\alpha}$
(13)	$1 - 4\alpha + 2\alpha^2 \ge 0$		

Table 3.4.

Simplified relations among α, α', p, and p'.

---

are weaker than (24), and that (3) and (10) are weaker than (5).

It is always advantageous to choose α' as small as possible — this discourages subitem nodes and allows more liberal choices of p and p'. Thus we should set $\alpha' = \alpha/(1-3\alpha)$. Now we check that (15), (16), (19), and (21) are weaker than (20), and that (25) is weaker than (22). Hence (22) governs the choice of p' and either (17) or (20) governs the choice of p. Clearly we should set $p' = 1/(1-\alpha)$. For p, we check that (20) is weaker for (17) for $0 < \alpha \le \alpha_1$, where $\alpha_1 = .18983^+$ is the smaller root of $1 - 9\alpha + 24\alpha^2 - 24\alpha^3 + 6\alpha^4$, but that (17) is weaker than (20) for $\alpha_1 < \alpha \le \alpha_0$. So to choose p, we compare α to α_1 and use (17) or (20) accordingly.

In order for the running time analyses to be valid, we must also have p and p' greater than 1. The right-hand sides of (17) and (22) always exceed 1, but the right-hand side of (20) does so only for $\alpha < \alpha_2$, where $\alpha_2 = .20550^-$ is the smallest root of $1 - 7\alpha + 11\alpha^2 - 3\alpha^3$. Note that $\alpha_2 < \alpha_0$, so this further restricts the feasible range for α. (It is curious that for $\alpha_2 < \alpha < \alpha_0$ the algorithm works correctly, but not provably efficiently, at least not by

this technique.)

To summarize, all the results in this chapter are valid if we choose

$$\begin{aligned} \alpha &< \alpha_2, \\ \alpha' &= \frac{\alpha}{1-3\alpha}, \\ p' &= \frac{1}{1-\alpha}, \\ \text{and } p &= \begin{cases} \dfrac{1-\alpha}{1-2\alpha}, & \text{if } 0 < \alpha \leq \alpha_1, \\ \dfrac{1-6\alpha+10\alpha^2-3\alpha^3}{\alpha(1-\alpha)}, & \text{if } \alpha_1 < \alpha < \alpha_2, \end{cases} \end{aligned}$$

where

$$\begin{aligned} \alpha_1 &= 0.18983^- \text{ satisfies } 1-9\alpha+24\alpha^2-24\alpha^3+6\alpha^4, \\ \text{and } \alpha_2 &= 0.20550^- \text{ satisfies } 1-7\alpha+11\alpha^2-3\alpha^3. \end{aligned}$$

We get the absolute maximum value for p by setting $\alpha = \alpha_1$; by increasing α, we increase the ACCESS time (since κ and H_α both increase) at the expense of the JOIN time (since p decreases). These formulas can by used to determine a compromise setting for α, assuming a decision about the relative frequency of ACCESS and JOIN operations. Some sample values

α	α'	p	p'	$\frac{1}{\lg \kappa}$	$\frac{1}{\lg H_\alpha}$	$\frac{3}{\lg p}$	$\frac{2+3\lambda}{\lg p'}$
.05	0.05882^+	1.0556^-	1.0526^+	13.513^+	3.4917^-	38.460^+	6837.8^-
.10	0.14286^-	1.125	1.1111^+	6.5788^+	2.1322^+	17.655^-	1197.3^+
.15	0.27273^-	1.2143^-	1.1765^-	4.2650^+	1.6398^-	10.710^+	392.38^+
.18	0.39130^+	1.2812^+	1.2195^+	3.4928^-	1.4704^+	8.3904^-	248.99^-
α_1	0.44094^-	1.3060^+	1.2343^+	3.2927^+	1.4263^+	7.7889^+	214.03^-
.19	0.44186^+	1.3023^-	1.2346^-	3.2894^-	1.4256^-	7.8729^-	213.81^+
.20	0.5	1.1	1.25	3.1063^-	1.3852^-	21.818^-	481.47^+
.205	0.53247^-	1.0088^-	1.2579^-	3.0214^-	1.3665^-	238.09^+	4955.1^-

Table 3.5.
Sample values for the parameters.

are given in Table 3.5.

Chapter 4

Conclusions and Future Directions

Biased 2-3 trees and biased weight-balanced trees are two examples of trees that implement dynamic dictionaries with logarithmic performance in the worst case for ACCESS and FIND operations, and with logarithmic performance for all dynamic operations, provided the cost is amortized over a sequence; the only technical exception is that the JOIN operation for biased 2-3 trees runs in time proportional to the difference in rank between two trees instead of the logarithm of the ratio of their weights. Even this "problem" (which we saw was not crucial due to the telescoping property of rank differences), biased 2-3 trees may be preferable to biased weight-balanced trees because the algorithms are simpler and because internal nodes need only store integer ranks, rather than real-number weights. Both trees are powerful enough to support the network flow and self-organizing data structure applications.

A number of questions involving dynamic weighted data structures were suggested but not answered in this thesis. We list a few here.

1. Special operations. The dynamic operations DELETE, PROMOTE, DEMOTE, and INSERT were implemented in terms of JOIN and SPLIT. Naturally, it should be possible to reduce the running time of these operations by writing direct algorithms. For instance, there is no reason to do a complete SPLIT at some node, essentially pulling it up to the root of the tree, if the weight added to it by PROMOTE would merely cause it to move up a shorter distance. Especially if PROMOTE and DEMOTE are only used with small values of δ (as in the self-orgainsing data structure application, where $\delta = 1$), a direct algorithm should achieve substantial savings over an indirect one.

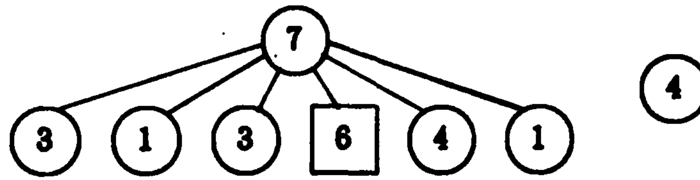


Figure 4.1.
A biased B-tree.

2. Discretization. Biased weight-balanced trees achieve true logarithmic performance, but they manipulate real numbers (weights) at every node. For practical purposes it would be desirable to make most of the arithmetic involve only integers, except for the conversion necessary from the weights of the items themselves. This was the case for biased 2-3 trees, but had the consequence that the running time of the JOIN operation depended on the rank difference and not on the weight ratio. The two measures are related, but not closely enough, because the upper and lower bounds on the weight of a node with a given rank do not match within a constant factor.

It would be interesting to find a data structure that could close the gap between these bounds without manipulating real numbers. For instance, by limiting the number of 3-nodes appearing near each other it may be possible to avoid the bad case of a tree with rank r having weight about 3^r.

3. Other trees. The general approach of both the data structures described here is to allow heavy item nodes to appear high in the tree, but to leave chips on the light nodes near it to pay for possible later operations involving these nodes. The chips make up for the fact that the light nodes really belong farther down in the tree; they can be thought of as counting the number of dummy nodes it would be necessary to introduce to form a path down to the level where the light nodes belong.

This idea is probably applicable to a wide variety of trees. It would be instructive to carry out the application for many kinds of trees, not only to have a large repertoire of dynamic weighted data structures, but also to determine how the structure constraints of various unweighted structures interact with the balance constraints of weighted ones. They do not always merge well; for instance it is not completely obvious how to make a biased B-tree with large branching factor. One problem is illustrated in Figure 4.1. An item node of rank 6 is surrounded by minor nodes in a biased 3-6 tree (each internal node except the root has between 3 and 6 children). If we JOIN another tree with rank 4, there are now

seven nodes with rank 6 or less, but we cannot make two trees with rank 7 from them. On the one hand we ought to combine some of the smaller nodes together into one subtree of rank 5, but on the other hand we cannot do that too often or we would reduce the number of children too much.

Nevertheless, it seems reasonable to expect that "biased" versions of B-trees and RB-trees exist. The balance conditions will just need to be subtler. If it is possible to exhibit a large repertoire of good structures, it would be interesting to examine general conditions under which an unweighted structure has a weighted counterpart. These conditions might mention the flexibility of substructures in the host structure, or the distribution of information-bearing and bookkeping nodes, or other properties.

4. The INSERT *problem.* It is difficult to INSERT into a biased 2-3 tree due to the uncertainty of how far down in the tree one has to look in order to find the correct place to insert the new item (see Section 1.2). Two different approaches might alleviate this problem.

First, if the probability that an INSERT will occur in each interval between currently present keys is known in advance, items corresponding to these gaps could be stored along with the real items, and given appropriate weights.

Second, if the search for the gap goes too far down in the tree, it might be possible to stop prematurely and insert the item, leaving a note that the INSERT never finished. Then a later operation which had occasion to go farther down the tree could carry the item along with it. Of course we would leave chips along with the note to pay for carrying on the INSERT. Two problems with this are how to keep track of many items waiting for their INSERTs to continue through the same node, and how to SPLIT at an incompletely INSERTed node.

5. Amortization. Of course, the main question is whether dynamic weighted data structures exist which do not require amortization, and which do not hide it in some other way (such as creating a lot of dummy nodes to intervene between light nodes and heavy item nodes.) Kriegel and others have recently worked on this problem [15, 16, 27].

But actually there is a sense in which amortization should not be considered a problem with this approach, but rather a feature. After all, if a light node joins the tree at less cost than its fair share, why should we pay more? We may eventually have to pay the remainder, but for now why not save our chips whenever possible. Only if the worst-case response of each operation is more important than the overall response of all operations together should we be searching for an unamortized solution to the dynamic weighted data structure problem.

References

[1] G. M. Adel'son-Vel'skiĭ and Y. M. Landis.
"An algorithm for the organization of information,"
Soviet Mathematics Doklady 3 (1962), 1259–1263.

[2] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley: Reading, Mass. (1974).

[3] J. L. Baer.
"Weight-balanced trees,"
Proceedings of the AFIPS National Conference 44 (1975), 467–472.

[4] Paul. J. Bayer.
"Improved bounds on the costs of optimal and balanced binary search trees,"
MAC Technical Memorandum 69.
Massachusetts Institute of Technology (1975).

[5] R. Bayer and E. McCreight.
"Organization and maintainance of large ordered indexes,"
Acta Informatica 1 (1972), 173–189.

[6] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan.
"Biased 2-3 trees,"
Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science (1980), 248–254.

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

[8] Mark R. Brown and Robert E. Tarjan.
"Design and analysis of a data structure for representing sorted lists,"
SIAM Journal on Computing 9 (1980), 594–614.

[9] J. Bruno and E. G. Coffman, Jr.
"Nearly optimal binary search trees,"
Infoımation Processing 71, North-Holland (1972), 99–103.

[10] Clark Allan Crane.
Linear Lists and Priority Queues as Balanced Binary Trees.
Ph.D. Thesis, Stanford University (1972).

[11] Michael L. Fredman.
"Two applications of a probabilistic search technique: sorting $x + y$ and building balanced search trees,"
Proceedings of the Seventh Annual ACM Symposium on Theory of Computing (1975), 240–244.

[12] Robert G. Gallager.
Information Theory and Reliable Communication.
Wiley: New York (1968).

[13] Adriano M. Garsia and Michelle L. Wachs.
"A new algorithm for minimum cost binary trees,"
SIAM Journal on Computing **6** (1977), 622–642.

[14] Leo J. Guibas and Robert Sedgewick.
"A dichromatic framework for balanced trees,"
Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science (1978), 8–21.

[15] H. Güting and H. P. Kriegel.
Dynamic k-dimensional multiway search under time-varying access frequencies,"
In *Theoretical Compter Science, 5th GI Conference* (P. Deussen, ed.).
Lecture Notes in Computer Science, Springer-Verlag, **104** (1981), 135–145.

[16] H. Güting and H. P. Kriegel.
"Multidimensional B-trees: An efficient dynamic file structure for exact match queries,"
Proceedings of the 10th Annual GI Conference;
Informatik Fachberichte **33** (1980), 375–388.

[17] T. C. Hu and A. C. Tucker.
"Optimal computer search trees and variable length alphabetic codes,"
SIAM Journal of Applied Math **21** (1971), 514–532.

[18] Scott Huddleston and Kurt Mehlhorn.
"Robust balancing in B-trees,"
In *Theoretical Compter Science, 5th GI Conference* (P. Deussen, ed.).
Lecture Notes in Computer Science, Springer-Verlag, **104** (1981), 234–244.

[19] Scott Huddleston and Kurt Mehlhorn.
"A new data structure for sorted lists,"
Acta Informatica, to appear.

[20] David A. Huffman.
"A method for the construction of minimum redundancy codes,"
Proc. IRE 40 (1951), 1098–1101.

[21] Donald E. Knuth.
The Art of Computer Programming, Vol. I: *Fundamental Algorithms*.
Addison-Wesley: Reading, Mass. (1968).

[22] Donald E. Knuth.
The Art of Computer Programming, Vol. III: *Sorting and Searching*.
Addison-Wesley: Reading, Mass. (1973).

[23] Donald E. Knuth.
"Dynamic Huffman coding,"
Journal of Algorithms, to appear.

[24] Donald E. Knuth.
"Huffman codes via algebra,"
Stanford Computer Science Dept. Technical Report STAN-CS-81-841 (1981).

[25] Donald E. Knuth.
"Optimum binary search trees,"
Acta Informatica 1 (1971), 14–25, 270.

[26] S. Rao Kosaraju.
"Localized search in sorted lists,"
Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (1981), 62–69.

[27] H. P. Kriegel and V. K. Vaishnavi.
"Weighted nultimensional B-trees used as nearly optimal dynamic dictionaries,"
Unpublished.

[28] David Maier and Sharon C. Salveter.
"Hysterical B-trees,"
Information Processing Letters bf 12 (1980), 199–202.

[29] Kurt Mehlhorn.
"Dynamic binary search,"
SIAM Journal on Computing 8 (1979), 175–198.

[30] Kurt Mehlhorn.
"Arbitrary weight changes in dynamic trees,"
Bericht 78/04, Universität des Saarlandes (1978).

[31] J. Nievergelt and E. M. Reingold.
"Binary search trees of bounded balance,"
SIAM Journal on Computing 2 (1973), 33–43.

[32] J. Nievergelt and C. K. Wong.
"On binary search trees,"
Information Processing 71, North-Holland (1972), 91–98.

[33] D. Stott Parker, Jr.
"Conditions for optimality of the Huffman algorithm,"
SIAM Journal on Computing **9** (1980), 470–489.

[34] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo.
Combinatorial Algorithms: Theory and Practice.
Prentice-Hall (1977).

[35] Ronald Rivest.
"On self-organizing sequential search heuristics,"
Communications of the ACM **19** (1976), 63–72.

[36] Daniel Dominic Kaplan Sleator.
An $O(mn \log n)$ Algorithm for Maximum Network Flow.
Ph.D. Thesis, Stanford University (1980).

[37] Daniel D. Sleator and Robert Endre Tarjan.
"A data structure for dynamic trees,"
Proceedings of the 13th Annual ACM Symposium on Theory of Computing (1981), 114–122.

[38] Robert Endre Tarjan.
"Efficiency of a good but not linear set union algorithm."
JACM **22** (1975), 215–225.

[39] K. Unterauer.
"Dynamic weighted binary search trees,"
Acta Informatica **11** (1979), 341–362.

[40] W. A. Walker and C. C. Gottlieb.
"A top-down algorithm for constructing nearly-optimal lexicographic trees,"
In *Graph Theory and Computing* (R. C. Read, ed.), Academic Press (1972), 303–323.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER STAN-CS-82-916	2. GOVT ACCESSION NO. AD-A122 046	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Dynamic Weighted Data Structures		5. TYPE OF REPORT & PERIOD COVERED technical, June 1982
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-82-916
7. AUTHOR(s) Samuel W. Bent, Ph.D.		8. CONTRACT OR GRANT NUMBER(s) N00014-76-K-0330 (ONR)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, CA 94305 USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ONR Branch Office 1030 East Green Street Pasadena, California 91101		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 80
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative - Mr. Robin Simpson Durand Aeronautics Building, Room 165 Stanford University Stanford, California 94305		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis discusses implementations of an abstract data structure called a 'dynamic dictionary'. Such a data structure stores a collection of items, each of which is equipped with a 'key' and a 'weight'. Among the operations we might wish to perform on such a collection are:

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

(a) accessing an item, given its key
(b) inserting a new item
(c) deleting an item
(d) joining two collections into one
(e) splitting a collection into two
(f) changing the weight of an item.

Operations (b)-(f) provide the dynamic nature of the data structure.

In addition we want the implementation to respect the weights, so that accessing a heavy item is quicker than accessing a light one. In an optimal binary tree, the path length to an item of weight w in a collection of total weight W is proportional to log(W/w) . By relaxing the optimality constraint and considering different kinds of trees, it is possible to retain this logarithmic access time (with a larger constant factor), and simultaneously achieve similar logarithmic times for the dynamic operations.

Two new data structures are proposed, 'biased 2-3 trees' and 'biased weight-balanced trees'. They achieve the logarithmic time bounds provided the cost is amortized over a sequence of operations. These data structures have applications to the network flow problem and to the design of "self-organizing" data structures.