Rationale for Adding Hash Tables to the C++ Standard Template Library

David R. Musser
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

DRAFT
February 20, 1995

Abstract

In *Hash Tables for the Standard Template Library*, Barreiro, Fraley, and Musser propose a restructuring and extension of the STL requirements for associative containers to accommodate hash table implementations. This paper gives the rationale for the proposed changes.

Contents

1 Introduction

The requirements for the C++ Standard Template Library [6] specify that associative containers have logarithmic worst case time bounds for inserting, erasing, and searching, and have iterators that efficiently iterate through the containers in sorted order. These requirements can be satisfied with balanced binary search trees, such as AVL trees or the red-black trees used in the Hewlett Packard reference implementation of STL, but an implementation using hash tables is ruled out. Storage and retrieval operations using hash tables have better *expected time* behavior—constant time—than balanced trees, but can take linear time in the worst case. And since hash tables do not keep entries organized according to any prescribed order, they do not support efficient ordered iteration. In *Hash Tables for the Standard Template Library* [3], Barreiro, Fraley, and Musser propose a restructuring and extension of the STL requirements for associative containers to accommodate hash table implementations. The proposal makes minimal changes to the existing requirements—the complexity requirements are relaxed from worst case bounds to be expected case bounds, which has the side-benefit of also permitting alternative balanced tree implementations that have expected or amortized logarithmic bounds, such as randomized search trees or splay trees. No change is made that affects any existing balanced binary tree implementation. This paper gives the rationale for the proposed changes.

The proposal restructures the existing STL associative container requirements into two parts: (1) order-independent requirements for all associative containers, and (2) an additional set of requirements for ordering. Together these parts are equivalent to the existing requirements except for the relaxation of complexity requirements from worst case to expected case requirements. The proposal then introduces (3) an additional set of requirements that together with (1) form a set of requirements for hash tables. The resulting set of requirements also serves as a significant example of the extensibility of the STL framework.

It is the overall STL framework and its potential for serving the many varying needs of all C++ programmers that motivates the hash table proposal itself and the detailed rationale given in this paper. Although all the basic facts about different varieties of both balanced trees and hash tables are well-known, they have never previously been subjected to the constraints imposed by a comprehensive and consistent framework for generic software components, such as STL now provides. In particular STL's requirements for efficient iteration and for supporting both unique and multi-key storage lead to some issues that have been only lightly treated in the literature, if at all. These issues are addressed in later sections.

2 How hash tables work

In general, "associative containers provide an ability for fast retrieval of data based on keys" [6]. The basic idea of hash tables is to shorten the searches required during data storage or retrieval by dividing up the set of keys into small subsets in such a way that each search can be confined to one of the subsets. A subset is determined as being all keys that are mapped to a particular integer by a function that has three properties:

1. it behaves as a mathematical function; i.e., it always produces the same integer from a given key;
2. it is easy (fast) to compute; and
3. for typical distributions of key occurrences it distributes the keys evenly over a large range of integers.

The first of these properties of the function is essential for correct storage and retrieval, and the second and third properties are essential for fast performance. Achieving 2 and 3 simultaneously is usually most feasible using a function that appears to associate integers randomly with keys (though because of 1 it must actually be only be pseudo-random), and for this reason the function is usually called a *hash* function.

2.1 Table organizations

Hash tables use the integer computed by a hash function from a key as an index to a random-access sequence container and store the key (and its associated value, if any) in the indexed position (which is usually called a *bucket* or *bin*) either as the entire contents of that position or as an item in a list stored in that position. In the former case, called *open addressing*, other keys that map into the same position must be stored in other positions, as determined by some collision resolution strategy, such as linear or quadratic probing or double hashing [5, Ch. 12]. In the case that lists are used, which is called a *separate chaining* method, the list stored at an index is searched with a simple linear search, which is acceptable if the lists are kept short by having sufficiently many buckets and an evenly distributing hash function.

2.2 Uniform hashing assumption

Whenever expected case performance requirements are discussed it is necessary to specify the distribution of inputs over which the expected times are to be computed. Here we make the simplest, most direct assumption possible for predicting or analyzing hash table performance, that each key is equally likely to hash into any of the buckets of the hash

table, independently of where any other key hashes to. This is called the *uniform hashing* assumption [5]. In practice this assumption is usually only imperfectly satisfied; occasionally it is violated severely, leading to much worse behavior than the ideal case, as discussed in the next section.

One case in which the uniform hashing assumption can be violated is that in which multiple entries with equal keys are stored (*hash_multiset* or *hash_multimap*) and there are a large number of entries with equal keys. All elements with equal keys must hash to the same bucket, so if a large number of elements with the same key are stored the search times for any key that hashes to that bucket will be long. If the number of repetitions of some key is proportional to the size of the table, the expected time for key searches may be linear rather than constant. If this situation exists, one should use the corresponding sorted associative container (*multiset* or *multimap*) instead, since they have an expected logarithmic time bound even if there are a large number of repetitions of a key.

2.3 Intermittent or gradual resizing

In order to achieve expected constant time bounds for search and retrieval operations, it is necessary to expand the number of buckets in the hash table as the number of entries grows. A simple method of expansion, called *intermittent resizing*, is to allocate another, larger, table of buckets; rehash all items in the old table into the new table; and then deallocate the old table. The resizing operation can be very time-consuming, perhaps prohibitively so in some applications requiring very fast response-times. This potential bottleneck can be avoided by expanding the table on a gradual basis (*gradual resizing*) using techniques such as those described in [7]. The bottleneck may also be overcome in certain applications by explicitly resizing the table when there is no time criticality. Gradual resizing is used in the reference implementation provided by Barreiro and Musser [2].

3 Advantages and disadvantages of hash tables

Some of the reasons commonly given for using hash tables in preference to other means of implementing associative containers include: they are faster, they use less memory, they do not require an ordering on the keys, and they are simpler to program. In each case, there is some truth to these claims but the real story is more complicated.

Speed. Perhaps the most often cited advantage of hash tables is that they can store and retrieve information in constant time, compared to logarithmic time bounds of balanced trees or linear time bounds of the crudest methods (such as linear lists). The main problem here is that the constant time bound for hash tables is an *expected* time bound, not a worst

case bound. If the hash function is not well-matched to the set of keys that occurs in a particular application, some buckets can receive many more than their share of keys. In the worst case, all keys map into a single bucket and the storage/retrieval times are a linear function of the number of elements stored. While it is not hard to avoid such a bad hash function, it may not be that easy in some situations to find a really good hash function (one that makes keys equally likely to be hashed into each bucket). This is a complication of using hash tables that does not arise with balanced trees.

The fact that hash tables cannot meet useful worst-case bounds is the main motivation for the one change the proposal makes in the existing associative container requirements: replacing worst-case time bounds with expected time bounds. But this change also seems desirable because it also opens the door to a wider variety of balanced tree implementations. For example, *randomized search trees* [1] have expected logarithmic time bounds for storage and retrieval, and splay trees [8] have amortized logarithmic time bounds; both are ruled out by the existing STL standard but would be permitted under the proposed change. Implementations based on AVL trees, red-black trees, or other balanced tree representations with worst case logarithmic bounds would be unaffected by the change.

Reduced memory requirements. In balanced trees each entry carries, in addition to the key and (with maps and multimaps) an associated value, at least three link fields (parent, left, and right child links). Some balanced tree algorithms can get along without parent links, by using a stack of nodes visited or by modifying structures as they descend into them, but this is not possible under the STL requirements on iterators (it must be possible to save iterators and start traversing the tree from where ever they point to). Red-black tree nodes have an additional field for the color; though it only requires one bit, in the C++ memory model at least one byte is necessarily allocated. Thus at least $M + 3A + 1$ bytes of memory are required for each entry, where M is the number of bytes required for the key and associated value and A is the number of bytes for an address.

With hash tables the amount of extra memory required depends on the organization of the table and on the *load factor* (whose definition also depends on the organization). The simplest case is the organization called open addressing in which all entries are stored in a single random-access table. No link fields are used and the load factor α is the portion of the table entries occupied. In this case the amount of memory used per entry is M/α, which can be much greater than $M + 3A + 1$ when α is small, but substantially less for $\alpha \approx .9$, say. (For $\alpha > .9$, the expected search times become intolerably large.)

With a separate chaining organization of the hash table, the table consists of a random-access table whose entries (called the buckets) are linked lists whose data fields are the keys or key/value pairs. The load factor α in this case is defined to be the number of entries divided by the number of buckets N (thus α can in this case be greater than 1).

With singly-linked lists, one link is required for each list node, and at least AN bytes are required for the bucket table, so the total amount of memory required per entry is at least $M + A(1 + 1/\alpha)$ bytes. Since α will generally be maintained at a value > 1, this amount can be substantially less than the $M + 3A + 1$ bytes required by red-black trees. Even if doubly linked lists are used the number of bytes required, $M + A(2 + 1/\alpha)$, will still be generally less than that for red-black trees.

The main difference between the balanced tree and hash table representations in terms of memory usage is the fact that for balanced trees the amount of memory in use at a given time is directly proportional to the number of entries stored, whereas for hash tables the amount in use also depends how the table was initialized and what strategy is being used to keep it from being too full for good performance.

No ordering needed. Balanced trees require an efficiently computable order relation on the keys, while hash tables only require an equality relation. This is probably not as big a difference as it might seem, since in many cases it is possible to construct a suitable order relation artificially (such as by considering all bits of the key representation to be the bits of an unlimited precision integer). Having to come up with such an ordering, though, in cases where sorted-order iteration is not needed may seem like an unnecessary burden on the user.

Simplicity of programming. Finally, balanced tree implementations are generally more complex than hash tables, but when the code is obtained from a library, difficulty of programming is not a consideration. The only issue might be the size of the code, particularly if several different instances of the template classes are necessary in an application. This is a factor that needs further study.

In summary, hash tables offer *potential* advantages over balanced trees in terms of expected time performance, lower memory use, independence of order relations, and smaller code size, but users of library versions of these containers must be aware of possible pitfalls. The major pitfall is the possibility of bad performance due to hash functions that are ill-suited to the particular distribution of keys encountered. Any user who does not have the time or inclination to guard against this possibility should stick to sorted associative containers and their more predictable balanced tree implementations.

4 Differences in associative container interfaces

In general the goal of the proposed restructuring and extension of the STL requirements for associative containers is to leave the interfaces for the existing (sorted) associative containers

completely unchanged, except for loosening the complexity requirements, and to provide as much of the same interface for hash tables as makes sense. By doing so, the proposal minimizes the effect on existing implementations (there are in fact no changes required) and maintains as far as possible the uniformity of interfaces that is one of the hallmarks of the STL framework. This section examines point by point the similarities and essential differences between sorted associative containers and the proposed unsorted associative containers (hash tables):

1. The template parameters for the existing *set*, *multiset*, *map* and *multimap* containers include a *Compare* function, used to compare keys and thus determine their ordering. The proposed *hash_set*, *hash_multiset*, *hash_map* and *hash_multimap* containers include instead a *KeyEqual* function parameter, used to compare keys for equality, and a *Hasher* parameter, which is type of functions that map keys pseudo-randomly to integers.

2. The existing containers provide iterators both for the normal direction of traversal from beginning (the smallest key among those in the container according to the ordering) to the end (the largest key), and the reverse order from end to beginning (largest to smallest). Both the normal and reverse direction iterators are *bidirectional* (they define both $++$ for forward traversal and $--$ for backward traversal). The proposed unsorted containers are only required to provide *forward* iterators (only $++$ is defined, not $--$) from beginning to end, producing the keys in no particular order (seemingly random order). Bidirectional iterators and reverse iterators could have been provided, but they do not seem as useful in the absence of sorted ordering of the keys, and limiting the requirement on iterators to forward-traversal-only allows a separate-chaining implementation to use singly-linked lists, which save space and time in comparison with the double linking required for bidirectional traversal.

3. For each of the new hashed container types, the same distinction holds as for the existing sorted-order containers between constant and non-constant iterator types, which is a generalization of the usual distinction between C++ pointer-to-const and pointer-to-nonconst types. Non-constant iterators are those that permit the item referred to by the iterator to be changed, whereas constant iterators permit no changes. The latter can thus be used on containers declared constant (usually those passed to a function by const reference). The *hash_set* and *hash_multiset* containers, in analogy to *set* and *multiset*, only provide a constant iterator type, since a non-constant iterator could be used to change a key, upsetting the internal retrieval mechanism. *Hash_map* and *hash_multimap*, in analogy to *map* and *multimap*, provide both constant and non-constant iterator types, the latter being useful for changing the value associated with a key. (The keys themselves are still protected from change by another mechanism.)

4. The constructors for sorted associative containers permit the key comparison function to be specified at construction time. The proposed hash table constructors permit the key-equality function and the hash function to be specified at construction time.

5. The *insert* functions for sorted associative containers include a version that takes an iterator p as an argument and treats it as a hint about where to begin searching for the given key. This version is useful in conjunction with STL's generic set algorithms (*set_union*, *set_difference*, *set_intersection*, *set_symmetric_difference*) since it allows these algorithms to operate in linear time when applied to associative containers. Since these algorithms depend on having sorted order, they are not applicable to hash tables, and thus there is no need in their interfaces for a version of *insert* with a hint argument.

6. The only other omissions from the hash table interfaces are operations whose meaning depends on sorted order: member functions *key_comp*, *value_comp*, *lower_bound*, and *upper_bound*; and *operator<*. All other insertion, erasure, search, and counting operations provided for sorted associative containers are also provided, with exactly the same interface, in the proposed unsorted associative containers. Included is *equal_range*, which returns a pair of iterators defining the range of container entries with keys equal to a given key; although the existing specification of *equal_range* is in terms of *lower_bound* and *upper_bound*, it is redefined in the proposal without reference to those operations. The definition in the new base requirements is that *equal_range*(k) returns a pair of iterators *first* and *last* such an iterator i refers to a value with key equal to k if and only if i is in the range [*first*, *last*). For sorted associative containers, the stronger definition is used, that *equal_range*(k) returns the pair *first* == *lower_bound*(k) and *last* == *upper_bound*(k). This definition implies the one in the base requirements but is stronger because it also means that *last*, if it is not equal to *end*(), must point to an element with key greater than k.

7. The present requirements for *lower_bound* and *upper_bound* in Table 12 (Associate container requirements) in [6] are not stated quite correctly; for example they say that *upper_bound* "returns an iterator pointing to the first element with key greater than k," but there will not be such an iterator if all elements have keys less than k. In the proposal these definitions are corrected to say that *a.lower_bound*(k) returns the iterator i such that all the elements in the container with keys equal to or greater than k are in [i, *a.end*()), and *a.upper_bound*(k) returns the iterator j such that all the elements in the container with keys greater than k are in [j, *a.end*()).

8. Operations added by hash table requirements to the common associative container requirements that are not in the requirements for sorted containers include the con-

structors already mentioned, and member functions *hash_funct*, *key_eq*, *bucket_count*, and *resize*. The *resize* operation, which takes an argument suggesting the number of buckets to use, is required even of implementations that do gradual resizing, so that users can exercise an extra degree of control over the number of buckets used.[1]

9. Equality (==) between two associative containers is defined by the requirements for all container types (Table 8 in [6]): $a == b$ means

   $$a.size() == b.size() \;\&\&\; equal(a.begin(),\ a.end(),\ b.begin());$$

   which means equality of the two sequences obtained by iterating through the containers, where equality of items in corresponding positions is checked with the == operator of the item type. The proposal retains this definition for hash tables as well as sorted associative containers. This is another case in which hash tables are less set-like than sorted associative containers. To see this, note that with sorted associative containers, equality of keys is not determined with ==; instead two keys *k1* and *k2* are considered equal if for the key comparison object *comp*, $comp(k1,\ k2) == false \;\&\&\; comp(k2,\ k1) == false$. But often these two definitions of key equality coincide, and in that case the above definition of equality of two sorted associative containers amounts to set equality (in the case of *set* or *map* containers) or multiset equality (in the case of *multiset* or *multimap* containers).

   With hash tables, though, $a == b$ can be false even if the key equality comparison object is the same as the key type's == operator and the two containers contain same set (or multiset) of elements, This happens if the sequences of their elements obtained by iteration are ordered differently, which of course can easily be the case if the elements have been inserted in different orders (or even if they were inserted in the same order but the two tables have different numbers of buckets).

10. The general requirements on all containers (Table 8 in [6]) also include requirements on the copy constructor and assignment, which are specified in terms of equality (==). Thus, for example, after an assignment $a = b$ it must be true that $a == b$. This implies that for hash tables assignment must be implemented by copying b structurally, not by hashing the elements of b into a, since that might produce an iteration sequence in a that differs from that of b.

[1] Under current consideration is a suggestion to replace *resize* by a *reserve* operation whose argument is the user's estimate of the maximum number of elements to be stored.

5 Using hash tables: a sample program

5.1 A program using a set

This subsection gives a small sample program that illustrates the use of the *set* sorted associative container. It is followed in the next subsection by a modification of the program to use use a *hash_set* to do essentially the same thing. First, here is the version using a *set*:

```
// Use an STL set to store a dictionary and look up a few words

#include <iostream.h>
#include <fstream.h>
#include <bstring.h>
#include <set.h>

typedef set<string, less<string> > set_1;

void lookup(const set_1& hs, const string& word)                    10
  // Look up word in the dictionary stored in hash_set hs
  // and report whether or not it was found.  If the word was found
  // and was not the last word in the dictionary, report the next
  // word following it (the next word in alphabetical order)
{
  cout << "Looking for " << word << " in the dictionary... ";
  set_1::iterator i = hs.find(word);

  if (i != hs.end())
    cout << "Found it." << endl;                                     20
  else {
    cout << "Didn't find it." << endl;
    return;
  }

  if (++i != hs.end())
    cout << "The next word after " << word << " in the dictionary is "
         << *i << endl;
  else
    cout << word << " is the last word in the dictionary.\n" << endl;  30
}
```

```
int main()
{
  string name("/usr/dict/words");

  ifstream ifs(name.c_str());
  typedef istream_iterator<string, ptrdiff_t> string_input;
  string_input i(ifs), eos;

  set_1 hs;

  cout << "Reading file " << name << endl;

  while (i != eos)
      hs.insert(*i++);

  cout << "The dictionary has " << hs.size() << " entries\n";

  lookup(hs, "hash");
  lookup(hs, "table");
  lookup(hs, "mispelled");
  lookup(hs, "zygote");

}
```

This program, compiled with the IBM xlC compiler and the February 7, 1995 release of the Hewlett-Packard reference implementation of STL, produces the following output:

```
Reading file /usr/dict/words
The dictionary has 26444 entries
Looking for hash in the dictionary... Found it.
The next word after hash in the dictionary is hashish
Looking for table in the dictionary... Found it.
The next word after table in the dictionary is tableau
Looking for mispelled in the dictionary... Didn't find it.
Looking for zygote in the dictionary... Found it.
zygote is the last word in the dictionary.
```

5.2 The same program, using a hash_set

Now let's modify the program to use a *hash_set* instead of the *set*. We use the reference implementation of hash tables of Barreiro and Musser [2], which uses a separate chaining table organization with gradual resizing. Resizing is controlled by two parameters, the minimum and maximum *load factors*. The load factor for a hash table is de-

fined to be the number of entries stored divided by the number of buckets. Facilities for specifying load factors are not included in the proposed standard for STL hash tables, as they may differ based on the type of representation (open addressing or separate chaining). The Barreiro and Musser implementation currently provides member functions **void** *set_loadfactors*(**float** *minLF*, **float** *maxLF*) for setting the load factors and **float** *minloadfactor*() and **float** *maxloadfactor*() for retrieving them, but this interface may be revised.

To modify the program of the previous subsection to use a *hash_set* we change

```
#include <set.h>
```

to

```
#include <hashset.h>
#include <hashfun.h>
```

and

```
typedef set<string, less<string> > set_1;
```

to

```
typedef hash_set<string, hash_fun1, equal_to<string> > set_1;
```

Here we have used the *hash_fun1* function object type, one of three sample hash function object types provided in *hashfun.h*. Although we could stick with the defaults for initial size (1009) and maximum load factor (2.0), we illustrate how these can be changed, by replacing

```
set_1 hs;
```

with

```
set_1 hs(10007);
hs.set_loadfactors(0.5, 1.5);
```

The initial size is set to 10007 and the minimum and maximum load factors are set to 0.5 and 1.5 (the minimum load factor was already 0.5 by default). Here is the resulting program, in which we have also modified the comments and added a line to report the

number of hash table buckets used.

```
// Use a hash_set to store a dictionary and look up a few words

#include <iostream.h>
#include <fstream.h>
#include <bstring.h>
#include <hashset.h>
#include <hashfun.h>

typedef hash_set<string, hash_fun1, equal_to<string> > set_1;
```

```
void lookup(const set_1& hs, const string& word)
  // Look up word in the dictionary stored in hash_set hs
  // and report whether or not it was found.  If the word was found
  // and was not the last word in the dictionary, report the next
  // word following it ("next" in no particular order)

{
  cout << "Looking for " << word << " in the dictionary... ";
  set_1::iterator i = hs.find(word);

  if (i != hs.end())
    cout << "Found it." << endl;
  else {
    cout << "Didn't find it." << endl;
    return;
  }

  if (++i != hs.end())
    cout << "The next word after " << word << " in the dictionary is "
         << *i << endl;
  else
    cout << word << " is the last word in the dictionary.\n" << endl;
}

int main()
{
  string name("/usr/dict/words");

  ifstream ifs(name.c_str());
  typedef istream_iterator<string, ptrdiff_t> string_input;
  string_input i(ifs), eos;

  set_1 hs(10007);
  hs.set_loadfactors(0.5, 1.5);

  cout << "Reading file " << name << endl;

  while (i != eos)
      hs.insert(*i++);

  cout << "The dictionary has " << hs.size() << " entries\n";
  cout << "There are " << hs.bucket_count() << " hash table buckets.\n\n";

  lookup(hs, "hash");
  lookup(hs, "table");
  lookup(hs, "mispelled");
  lookup(hs, "zygote");

}
```

When compiled with the IBM xlC compiler, the February 7, 1995 release of the Hewlett-Packard reference implementation of STL, and the Barreiro and Musser reference implementation of hash tables [2], this program produces the following output:

```
Reading file /usr/dict/words
The dictionary has 26444 entries
There are 17630 hash table buckets.

Looking for hash in the dictionary... Found it.
The next word after hash in the dictionary is fish
Looking for table in the dictionary... Found it.
The next word after table in the dictionary is exalt
Looking for mispelled in the dictionary... Didn't find it.
Looking for zygote in the dictionary... Found it.
The next word after zygote in the dictionary is expectorate
```

Note that the number of buckets has been automatically increased from the initial 10007 to 17630 (so that the load factor is maintained below the requested maximum of 1.5). Note also that, as expected, now the words following the retrieved words are no longer those that follow in alphabetical order.

6 Remaining issues

The main remaining issue in specifying the requirements for STL hash tables is how to provide facilities for controlling automatic resizing. At present such facilities are provided, but in somewhat different form, in two reference implementations of the proposal [2, 4]. These facilities are still being subjected to experimentation and are not yet included in the proposed requirements.

One other issue is the organization of `include` files, which is not specified in the proposal and which is somewhat different in the two reference implementations.

Acknowledgments. The rationale developed in this paper benefitted from many discussions with Javier Barreiro and Bob Fraley. Bob Fraley, Meng Lee, and Alex Stepanov suggested a number of corrections and improvements.

References

[1] Cecilia R. Aragon and Raimund G. Seidel, "Randomized Search Trees," Proc. of IEEE Conference on Foundations of Computer Science, 1989.

[2] Javier Barreiro and David R. Musser, *An STL Hash Table Implementation with Gradual Resizing*, February 20, 1995, available by anonymous ftp from ftp.cs.rpi.edu as pub/stl/hashimp2.ps.Z

[3] Javier Barreiro, Robert Fraley, and David R. Musser, *Hash Tables for the Standard Template Library*, Doc. No. X3J16/94-0218, WG21/N0605, January 30, 1995, revised February 20, 1995, available by anonymous ftp from ftp.cs.rpi.edu as pub/stl/hashdoc.ps.

[4] Bob Fraley, *An STL Hash Table Implementation*, February 17, 1995, available by anonymous ftp from butler.hpl.hp.com as stl/bfhash.Z.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1991.

[6] Alexander A. Stepanov and Meng Lee, *The Standard Template Library*, Technical Report, Hewlett-Packard Laboratories, September 20, 1994, revised February 7, 1995, available by anonymous ftp from ftp.cs.rpi.edu as pub/stl/doc.ps.Z or from butler.hpl.hp.com as part of stl/sharfile.Z.

[7] Per-Ake Larson, *CACM*, Vol. 31, Number 4, April 1988.

[8] D.D. Sleator and R.E. Tarjan, "Self-adjusting binary search trees," sl JACM 32 (1985), 652-686.