An STL Hash Table Implementation With Gradual Resizing

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

DRAFT
February 20, 1995

Abstract

This paper describes an implementation of hash tables that conforms to the proposed requirements in *Hash Tables for the Standard Template Library*, by Barreiro, Fraley, and Musser. The main characteristics of the implementation are its separate chaining table organization and its use of gradual resizing to maintain expected constant time performance.

1 Introduction

In *Hash Tables for the Standard Template Library* [2], Barreiro, Fraley, and Musser propose a restructuring and extension of the STL requirements [5] for associative containers to accommodate hash table implementations. A rationale for the proposal is given in [1]. In this paper we briefly describe a reference implementation of hash tables that conforms to the requirements. A separate chaining table organization is used, and *gradual resizing* of the table is used to maintain a number of hash buckets proportional to the number of elements stored. That is, new buckets are added one at a time to the existing table and some elements in old buckets are rehashed into the new ones, as opposed to resizing by the more conventional means of allocating an entirely new table and rehashing all elements into it. Resizing helps ensure that all storage and retrieval operations are performed in expected constant time, and doing it gradually avoids the significant delays that can occur if an entire table has to be rehashed. The gradual resizing algorithm is based mainly on a method of Larson [6], which is described more fully in Section 2.

This implementation has been tested mainly by storing and retrieving words from dictionaries of various sizes, but it is still experimental and subject to change. It still contains some code whose only purpose is for debugging and which will be removed in a later version. One operation required by the proposed standard, @resize@, for explicit resizing of the table, is not yet implemented (it does nothing). A future version of this paper will describe the implementation in more detail and will report the results of extensive performance measurements and comparisons to another hash table implementation [3] and to balanced tree implementations of STL sorted associative containers.

The current code may be obtained by anonymous ftp from ftp.cs.rpi.edu in directory pub/stl or from butler.hpl.hp.com in directory stl.

2 How gradual resizing is done

The basic idea of Larson's method of gradual hash table expansion [6] is to add one bucket at a time at the end of the table, slightly adjusting the hash function accordingly (the actual hash function in use at a given time is an adaptation of an initially-given hash function). Some elements (ones whose keys hash to the new bucket using the new hash function) are moved from an existing bucket to the new one. The existing bucket, the "buddy" of the new one, can be chosen simply as the middle bucket; this choice simplifies the transition from the existing hash function to the new one. Similarly, contraction can be handled by deleting a bucket at the end after moving its contents to its buddy.

Inserting a new position at the end of a random-access sequence is an operation that is already supported in constant (amortized) time in STL by its vector and deque containers. (Larson described a representation that is essentially equivalent to that used for deques in the Hewlett-Packard reference implementation of STL). Although reallocations of the vector or deque may sometimes be necessary, it is then only necessary to copy the existing entries to the new space rather than having to rehash all existing entries, which would be much more time-consuming than just copying. Thus with gradual expansion rehashing does occur but its cost is distributed over all the insertions instead of being concentrated in a single insertion or resize operation.

3 Singly-linked lists

This hash table implementation uses a singly-linked list class, @slist@, which is based on the standard STL @list@ class (which uses double linking). The main reason for using singly-linked lists in the hash table implementation is to reduce space requirements; some time is saved also since only one link field has to be maintained for each entry. At present the implementation of @slist@ is incomplete (member functions @remove@, @unique@,

@merge@, @reverse@, and @sort@ are not implemented), and some changes to the interface are planned. Once these revisions are made and the implementation is completed, @slist@ should be independently useful and will be described in a separate report [4].

References

[1] David R. Musser, *Rationale for Adding Hash Tables to the C++ Standard Template Library*, February 20, 1995, available by anonymous ftp from ftp.cs.rpi.edu as pub/stl/hashrationale.ps.

[2] 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.

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

[4] David R. Musser, *Generic Singly-linked Lists Compatible with the C++ Standard Template Library*, in preparation.

[5] 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.

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