Doc No: X3J16/94-0218
WG21/N0605
Date: February 20, 1995
Project: Programming Language C++
Reply to: musser@cs.rpi.edu
fraley@hpl.hp.com

Hash Tables for the Standard Template Library

Javier Barreiro, Robert Fraley**, David R. Musser**

**Rensselaer Polytechnic Institute*
Computer Science Department
Troy, NY 12180

***Hewlett-Packard Laboratories*
1501 Page Mill Road
Palo Alto, CA 94304

Hash Tables for the Standard Template Library

1 Introduction

The Standard Template Library (STL), which has been approved for inclusion in the C++ Standard Library by the ANSI/ISO C++ Standards Committee, supplies a set of associative container classes (set, multiset, map, multimap) that support fast storage retrieval of data based on keys. As specified in the STL standard these containers could be more precisely described as *sorted* associative containers, since in addition to fast storage and retrieval they are required to support efficient iteration through the entries in sorted order---according to the order determined by a boolean-valued function supplied as a parameter when the container is created. The sorted iteration capability is not always needed. If the requirement for fast sorted iteration is dropped, a wider range of implementations is possible; in particular, an attractive choice is a hash table implementation. In this report we first show how to restructure the existing STL associative container requirements into two parts: (1) base requirements for all associative containers, and (2) an additional set of requirements for sorting capabilities.We then introduce a set of requirements (3) that together with (1) form a set of requirements for hash tables. These changes and additions are fully consistent with the STL framework and also serve as a significant example of the extensibility of the framework.

We have developed and experimented with two independent hash table implementations, which are described in detail in separate reports. These implementations not only demonstrate the maturity and consistency of the requirements described here, but should also be of immediate practical use to C++ programmers who need associative containers but can do without sorted order of the keys. These implementations are available by anonymous ftp from butler.hpl.hp.com in directory stl or from ftp.cs.rpi.edu in directory pub/stl. A paper giving the rationale for this proposal is also available from either site.

Part of the material in the following section is adapted directly from *The Standard Template Library*, by Alexander Stepanov and Meng Lee, Hewlett-Packard Report, December 6, 1994. This section is the restructuring and extension of the section on associative containers (Section 8) from that document. The intended changes are the following:

- What were called “associative containers” are now called “sorted associative containers,” and the name “associative containers” now applies to both sorted and unsorted associative containers.
- The previous requirements have been divided into a base set of requirements for all associative containers and an additional set that define requirements for sorted associative containers.
- The complexity requirements for storage and retrieval operations on associative containers are relaxed by specifying *expected time* bounds rather than worst-case bounds or amortized-cost bounds.
- A new table has been added to define the requirements for hash tables.

The use of expected time bounds instead of worst case bounds is necessary for the hash table requirements, since hash tables cannot satisfy useful worst case bounds (hash tables have *linear* worst-case behavior; their advantage is in the average case). Using expected time bounds for *all* associative containers also opens the door to 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 tree implementation.

The interfaces of container classes set, multiset, map, and multimap have been included for completeness; no changes are being proposed to these classes.

2 Associative containers

Associative containers provide an ability for fast retrieval of data based on keys. There are two major categories of associative containers: *sorted associative containers* and *hashed associative containers*; the latter are also called *hash tables*. These categories share many requirements, which we call the *base requirements for associative containers*.

The library provides four basic kinds of sorted associative containers: `set`, `multiset`, `map` and `multimap`, and four basic kinds of hash tables: `hash_set`, `hash_multiset`, `hash_map` and `hash_multimap`.

All of the sorted associative containers are parameterized on `Key` and an ordering relation `Compare` that induces a total ordering on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary type `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container.

All of the hash tables are parameterized on `Key`, a function type `Hasher` that maps elements of `Key` into integers, and a function type `KeyEqual` that induces an equivalence relation on elements of `Key`. In addition, `hash_map` and `hash_multimap` associate an arbitrary type `T` with the `Key`. The object of type `KeyEqual` is called the *key-equivalence object* of a container.

With sorted associative containers, when we talk about equality of keys we mean the equivalence relation imposed by the comparison and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equal if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`.

With hash tables, when we talk about equality of keys we mean the equivalence relation imposed by the key-equivalence object and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equal if for the key equivalence object `equal`, `equal(k1, k2) == true`.

An associative container (either sorted or hashed) supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equal keys*. `set`, `map`, `hash_set`, and `hash_map` support unique keys. `multiset`, `multimap`, `hash_multiset`, and `hash_multimap` support equal keys.

For `set`, `multiset`, `hash_set`, and `hash_multiset`, the value type is the same as the key type. For `map`, `multimap`, `hash_map`, and `hash_multimap`, it is equal to `pair<const Key, T>`.

`The iterator` type of a sorted associative container is of the bidirectional iterator category, while that of hash tables is of the forward iterator category. `insert` does not affect the validity of iterators and references to the container, and `erase` invalidates only the iterators and references to the erased elements.

In the following table, `X` is an associative container class, `a` is a value of `X`, `a_uniq` is a value of `X` when `X` supports unique keys, and `a_eq` is a value of `X` when `X` supports multiple keys, `i` and `j` satisfy input iterator requirements and refer to elements of `value_type`, `[i, j)` is a valid range, `p` is a valid iterator to `a`, `q`, `q1`, `q2` are valid dereferenceable iterators to `a`, `[q1, q2)` is a valid range, `t` is a value of `X::value_type` and `k` is a value of `X::key_type`. The table entries in the complexity column are in some cases qualified by the

Table 1: Base requirements for associative container (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
`X::key_type`	`Key`		compile time
`a_uniq.insert(t)`	`pair<iterator, bool>`	inserts `t` if and only if there is no element in the container with key equal to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place and the `iterator` component of the pair points to the element with key equal to the key of `t`.	expected logarithmic

Table 1: Base requirements for associative container (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
`a_eq.insert(t)`	`iterator`	inserts `t` and returns the iterator pointing to the newly inserted element.	expected logarithmic
`a.insert(i, j)`	result is not used	inserts the elements from the range `[i, j)` into the container.	expected `O(Nlog(size()+N)` (`N` is the distance from `i` to `j`) in general; expected `O(N)` if `[i, j)` is sorted according to `value_comp()`
`a.erase(k)`	`size_type`	erases all the elements in the container with key equal to `k`. returns the number of erased elements.	expected `O(log(size()) + count(k))`
`a.erase(q)`	result is not used	erases the element pointed to by `q`.	expected constant
`a.erase(q1, q2)`	result is not used	erases all the elements in the range `[q1, q2)`.	`O(log(size())+ N)` where `N` is the distance from `q1` to `q2`.
`a.find(k)`	`iterator; const_iterator` for constant `a`	returns an iterator pointing to an element with the key equal to `k`, or `a.end()` if such an element is not found.	expected logarithmic
`a.count(k)`	`size_type`	returns the number of elements with key equal to `k`.	expected `O(log(size())+ count(k))`
`a.equal_range(k)`	`pair<iterator, iterator>; pair<const_iterator, const_iterator>` for constant `a`	returns iterators `i` and `j` such that all elements with keys equal to `k` are in the range `[i, j)`. This range is empty if no elements have key `k`.	expected logarithmic

word *expected*, meaning that the expected (or average) time is required to satisfy the indicated bound, rather than the worst-case time. For example expected logarithmic means that the expected time must be O(log size()), which is a looser requirement than requiring the worst-case time to be logarithmic. Note that an expected time requirement is also met by an implementation for which the *amortized* time meets the bound; i.e., the stated expected time requirements allow implementations with either expected or amortized time bounds. Finally, note that many of the complexity requirements given in this base requirements table are replaced in Table 3 (Additional requirements for hash tables) by stronger complexity requirements (e.g., expected logarithmic is replaced by expected constant).

2.1 Sorted Associative Containers

The additional requirements for sorted associative containers are given in the following table:

Table 2: Sorted associative container requirements (in addition to base requirements)

expression	return type	assertion/note pre/post-condition	complexity
X::key_compare	Compare	defaults to less<key_type>.	compile time
X::value_compare	a binary predicate type	is the same as key_compare for set and multiset; is an ordering relation on pairs induced by the first component (i.e. Key) for map and multimap.	compile time
X(c) X a(c);		constructs an empty container; uses c as a comparison object.	constant
X() X a;		constructs an empty container; uses Compare() as a comparison object.	constant
X(i, j, c) X a(i, j, c);		constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.	expected O(NlogN) in general (N is the distance from i to j); expected O(N) if [i, j) is sorted with value_comp()
X(i, j) X a(i, j);		same as above, but uses Compare() as a comparison object.	same as above
a.key_comp()	X::key_compare	returns the comparison object out of which a was constructed.	constant
a.value_comp()	X::value_compare	returns an object of value_compare constructed out of the comparison object.	constant
a.insert(p, t)	iterator	inserts t if and only if there is no element with key equal to the key of t in containers with unique keys; always inserts t in containers with equal keys. always returns the iterator pointing to the element with key equal to the key of t. iterator p is a hint pointing to where the insert should start to search.	expected logarithmic in general, but expected constant if t is inserted right after p.
a.lower_bound(k)	iterator; const_iterator for constant a	returns an iterator i such that all elements with keys equal to or greater than k are in the range [i, a.end()).	expected logarithmic
a.upper_bound(k)	iterator; const_iterator for constant a	returns an iterator j such that all elements with keys greater than k are in the range [j, a.end()).	expected logarithmic

Table 2: Sorted associative container requirements (in addition to base requirements)

expression	return type	assertion/note pre/post-condition	complexity
`a.equal_range(k)`	same as base requirements	equivalent to `make_pair(a.lower_bound(k), a.upper_bound(k))`.	expected logarithmic

The fundamental property of iterators of sorted associative containers is that they iterate through the containers in the non-descending order of keys, where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive,

```
value_comp(*j, *i) == false
```

For sorted associative containers with unique keys the stronger condition holds,

```
value_comp(*i, *j) == true.
```

2.1.1 Set

`set` is a kind of sorted associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves.

```
template <class Key, class Compare = less<Key>,
             template <class U> class Allocator = allocator>
class set {
public:

// typedefs:

      typedef Key key_type;
      typedef Key value_type;
      typedef Allocator<Key>::pointer pointer;
      typedef Allocator<Key>::reference reference;
      typedef Allocator<Key>::const_reference const_reference;
      typedef Compare key_compare;
      typedef Compare value_compare;
      typedef iterator;
      typedef iterator const_iterator;
      typedef size_type;
      typedef difference_type;
      typedef reverse_iterator;
      typedef const_reverse_iterator;

// allocation/deallocation:

      set(const Compare& comp = Compare());
      template <class InputIterator>
      set(InputIterator first, InputIterator last,
            const Compare& comp = Compare());
      set(const set<Key, Compare, Allocator>& x);
      ~set();
      set<Key, Compare, Allocator>& operator=(const set<Key, Compare,
            Allocator>& x);
      void swap(set<Key, Compare, Allocator>& x);
```

```
// accessors:

      key_compare key_comp() const;
      value_compare value_comp() const;
      iterator begin() const;
      iterator end() const;
      reverse_iterator rbegin();
      reverse_iterator rend();
      bool empty() const;
      size_type size() const;
      size_type max_size() const;

// insert/erase:

      pair<iterator, bool> insert(const value_type& x);
      iterator insert(iterator position, const value_type& x);
      template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
      void erase(iterator position);
      size_type erase(const key_type& x);
      void erase(iterator first, iterator last);

// set operations:

      iterator find(const key_type& x) const;
      size_type count(const key_type& x) const;
      iterator lower_bound(const key_type& x) const;
      iterator upper_bound(const key_type& x) const;
      pair<iterator, iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(const set<Key, Compare, Allocator>& x,
             const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<(const set<Key, Compare, Allocator>& x,
             const set<Key, Compare, Allocator>& y);
```

`iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the same type as `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

2.1.2 Multiset

`multiset` is a kind of sorted associative container that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves.

```
template <class Key, class Compare = less<Key>,
             template <class U> class Allocator = allocator>
class multiset {
public:
```

```
// typedefs:

       typedef Key key_type;
       typedef Key value_type;
       typedef Allocator<Key>::pointer pointer;
       typedef Allocator<Key>::reference reference;
       typedef Allocator<Key>::const_reference const_reference;
       typedef Compare key_compare;
       typedef Compare value_compare;
       typedef iterator;
       typedef iterator const_iterator;
       typedef size_type;
       typedef difference_type;
       typedef reverse_iterator;
       typedef const_reverse_iterator;

// allocation/deallocation:

       multiset(const Compare& comp = Compare());
       template <class InputIterator>
       multiset(InputIterator first, InputIterator last,
              const Compare& comp = Compare());
       multiset(const multiset<Key, Compare, Allocator>& x);
       ~multiset();
       multiset<Key, Compare, Allocator>& operator=(const multiset<Key, Compare,
              Allocator>& x);
       void swap(multiset<Key, Compare, Allocator>& x);

// accessors:

       key_compare key_comp() const;
       value_compare value_comp() const;
       iterator begin() const;
       iterator end() const;
       reverse_iterator rbegin();
       reverse_iterator rend();
       bool empty() const;
       size_type size() const;
       size_type max_size() const;

// insert/erase:

       iterator insert(const value_type& x);
       iterator insert(iterator position, const value_type& x);
       template <class InputIterator>
       void insert(InputIterator first, InputIterator last);
       void erase(iterator position);
       size_type erase(const key_type& x);
       void erase(iterator first, iterator last);

// multiset operations:

       iterator find(const key_type& x) const;
       size_type count(const key_type& x) const;
       iterator lower_bound(const key_type& x) const;
       iterator upper_bound(const key_type& x) const;
       pair<iterator, iterator> equal_range(const key_type& x) const;
};
```

```
template <class Key, class Compare, class Allocator>
bool operator==(const multiset<Key, Compare, Allocator>& x,
                const multiset<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<(const multiset<Key, Compare, Allocator>& x,
               const multiset<Key, Compare, Allocator>& y);
```

`iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the same type as `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

2.1.3 Map

`map` is a kind of sorted associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type `T` based on the keys.

```
template <class Key, class T, class Compare = less<Key>,
          template <class U> class Allocator = allocator>
class map {
public:

// typedefs:

    typedef Key key_type;
    typedef pair<const Key, T> value_type;
    typedef Compare key_compare;
    class value_compare
        : public binary_function<value_type, value_type, bool> {
    friend class map;
    protected:
        Compare comp;
        value_compare(Compare c) : comp(c) {}
    public:
        bool operator()(const value_type& x, const value_type& y) {
            return comp(x.first, y.first);
        }
    };
    typedef iterator;
    typedef const_iterator;
    typedef Allocator<value_type>::pointer pointer;
    typedef Allocator<value_type>::reference reference;
    typedef Allocator<value_type>::const_reference const_reference;
    typedef size_type;
    typedef difference_type;
    typedef reverse_iterator;
    typedef const_reverse_iterator;

// allocation/deallocation:

    map(const Compare& comp = Compare());
    template <class InputIterator>
    map(InputIterator first, InputIterator last,
```

```
            const Compare& comp = Compare());
      map(const map<Key, T, Compare, Allocator>& x);
      ~map();
      map<Key, T, Compare, Allocator>&
            operator=(const map<Key, T, Compare, Allocator>& x);
      void swap(map<Key, T, Compare, Allocator>& x);

// accessors:

      key_compare key_comp() const;
      value_compare value_comp() const;
      iterator begin();
      const_iterator begin() const;
      iterator end();
      const_iterator end() const;
      reverse_iterator rbegin();
      reverse_iterator rend();
      bool empty() const;
      size_type size() const;
      size_type max_size() const;
      Allocator<T>::reference operator[](const key_type& x);

// insert/erase:

      pair<iterator, bool> insert(const value_type& x);
      iterator insert(iterator position, const value_type& x);
      template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
      void erase(iterator position);
      size_type erase(const key_type& x);
      void erase(iterator first, iterator last);

// map operations:

      iterator find(const key_type& x);
      const_iterator find(const key_type& x) const;
      size_type count(const key_type& x) const;
      iterator lower_bound(const key_type& x);
      const_iterator lower_bound(const key_type& x) const;
      iterator upper_bound(const key_type& x);
      const_iterator upper_bound(const key_type& x) const;
      pair<iterator, iterator> equal_range(const key_type& x);
      pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const map<Key, T, Compare, Allocator>& x,
             const map<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const map<Key, T, Compare, Allocator>& x,
             const map<Key, T, Compare, Allocator>& y);
```

`iterator` is a bidirectional iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

In addition to the standard set of member functions of sorted associative containers, `map` provides `reference operator[](const key_type&)`. For a map `m` and key `k`, `m[k]` is semantically equivalent to `(*((m.insert(make_pair(k, T()))).first)).second`.

2.1.4 Multimap

`multimap` is a kind of sorted associative container that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys.

```
template <class Key, class T, class Compare = less<Key>,
             template <class U> class Allocator = allocator>
class multimap {
public:

// typedefs:

       typedef Key key_type;
       typedef pair<const Key, T> value_type;
       typedef Compare key_compare;
       class value_compare
              : public binary_function<value_type, value_type, bool> {
       friend class multimap;
       protected:
              Compare comp;
              value_compare(Compare c) : comp(c) {}
       public:
              bool operator()(const value_type& x, const value_type& y) {
                     return comp(x.first, y.first);
              }
       };
       typedef iterator;
       typedef const_iterator;
       typedef Allocator<value_type>::pointer pointer;
       typedef Allocator<value_type>::reference reference;
       typedef Allocator<value_type>::const_reference const_reference;
       typedef size_type;
       typedef difference_type;
       typedef reverse_iterator;
       typedef const_reverse_iterator;

// allocation/deallocation:

       multimap(const Compare& comp = Compare());
       template <class InputIterator>
       multimap(InputIterator first, InputIterator last,
              const Compare& comp = Compare());
       multimap(const multimap<Key, T, Compare, Allocator>& x);
       ~multimap();
       multimap<Key, T, Compare, Allocator>&
              operator=(const multimap<Key, T, Compare, Allocator>& x);
       void swap(multimap<Key, T, Compare, Allocator>& x);

// accessors:
```

```
        key_compare key_comp() const;
        value_compare value_comp() const;
        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        reverse_iterator rend();
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

// insert/erase:

        iterator insert(const value_type& x);
        iterator insert(iterator position, const value_type& x);
        template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
        void erase(iterator position);
        size_type erase(const key_type& x);
        void erase(iterator first, iterator last);

// multimap operations:

        iterator find(const key_type& x);
        const_iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        iterator lower_bound(const key_type& x);
        const_iterator lower_bound(const key_type& x) const;
        iterator upper_bound(const key_type& x);
        const_iterator upper_bound(const key_type& x) const;
        pair<iterator, iterator> equal_range(const key_type& x);
        pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const multimap<Key, T, Compare, Allocator>& x,
             const multimap<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const multimap<Key, T, Compare, Allocator>& x,
             const multimap<Key, T, Compare, Allocator>& y);
```

`iterator` is a bidirectional iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

2.2 Hash Tables

The following table gives the requirements beyond the base requirements of Table 1 for hashed associative containers (hash tables). In this table b is an integer of type size_type, and h is a hash function of type X::hash_type.

Table 3: Hashed associative container requirements (in addition to base requirements)

expression	return type	assertion/note pre/post-condition	complexity
`X::key_equal`	`KeyEqual`	Type of the key equality function. Defaults to `equal_to<key_type>`.	compile time
`X::hasher`	`Hasher`	Type of the hash function. Defaults to `hash<Key>`	compile time
`X()` `X a;`		constructs an empty container; uses HashFunction() as the hash function and Key_Equal() as the key comparison; builds at least 1000 buckets.	constant
`X(b)` `X a(b);`		constructs an empty container using above defaults, except having space for `b` buckets.	constant
`X(b, h)` `X a(b, h);`		as above, but uses h as the hash function.	constant
`X(b, h, c)` `X a(b, h, c)`		constructs an empty container; uses h as the hash function and c as the key comparison	constant

`X(i, j)`
`X(i, j, b)`
`X(i, j, b, h)`
`X(i, j, b, h, c)`
`X a(i,j);`
`X a(i, j, b);`
`X a(i, j, b, h);`
`X a(i,j,b,h,c);`
		constructs an empty container and inserts elements from the range `[i,j)` into it. Uses `b, h, c` as described above, or defaults when they are not provided.	expected `O(N)`, where `N` the distance from `i` to `j`.

`a::hash_funct()`	`X::hasher`	returns the hashing function	compile time
`a::key_eq()`	`X::key_equal`	returns the key comparison function	compile time
`a_uniq.insert(t)`	`pair<iterator, bool>`	same as base requirements	expected constant
`a_eq.insert(t)`	`iterator`	same as base requirements	expected constant
`a.insert(i,j)`	result is not used	same as base requirements	expected `O(N)`, where `N` is the distance from `i` to `j`.
`a.erase(k)`	`size_type`	same as base requirements	expected `O(count(k))`

Table 3: Hashed associative container requirements (in addition to base requirements)

expression	return type	assertion/note pre/post-condition	complexity
`a.erase(q1, q2)`	result is not used	erases all the elements in the range `[q1, q2)`	expected `O(N)` (`N` is the distance from `q1` to `q2`).
`a.find(k)`	as in base requirements	as in base requirements	expected constant
`a.equal_range(k)`	as inbase requirements	as in base requirements	expected `O(N)`, where `N` is the number of elements having key `k`.
`a.count(k)`	`size_type`	returns the number of elements with key equal to `k`.	expected `O(count(k))`
`bucket_count()`	`size_type`	returns the current number of hash buckets	constant
`resize(b)`	result is not used	restructures the table to contain `b` buckets.	linear

For hash tables, iteration through the table presents the keys in no particular order; the actual order depends on the order of insertion, the number of buckets, and on the hash function. Table entries which have equal keys will appear consecutively when iterating through the table. Iterators and references remain valid after all operations, including `resize`, but ranges such as the result of `equal_range` may no longer be valid.

The bucket parameters to the constructors and the resize operation guide the implementation in selecting the actual number of buckets being used. An implementation is free to choose more. The number of buckets in use for a table may change any time elements are inserted or deleted from the table. The resize(buckets) member function can be issued to change the bucket count at a time when the program can accommodate the overhead of table restructuring.

The hash function is expected to accept a single parameter, the key, and return a number in the range defined by size_type. The ability of the implementation to achieve expected constant time performance depends on the ability of the hashing function to hash key values evenly across a large range of integers.

2.2.1 Hash Function

Each implementation will provide a default hash function, a subclass of `unary_function<T, size_type>`, where T is the key type.

```
template <class T> class hash: unary_function<T, size_type>{
        size_type operator() (const T & key);
};
```

This shall be defined at least for the numeric types and `string`.

2.2.2 Hash_set

hash_set is a kind of hash table that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves.

```
template <class Key, class Hasher=hash<Key>, class KeyEqual = equal_to<Key>,
          template <class U> class Allocator = allocator>
class hash_set {
public:

// typedefs:

    typedef Key key_type;
    typedef Key value_type;
    typedef Hasher hasher;
    typedef KeyEqual key_equal;

    typedef pointer;
    typedef reference;
    typedef const_reference;
    typedef iterator;
    typedef iterator const_iterator;
    typedef size_type;
    typedef difference_type;

// allocation/deallocation

    hash_set(size_type table_size = 1009,
             const Hasher& hf = Hasher(),
             const KeyEqual& equal = KeyEqual());
    template <class InputIterator>
    hash_set(InputIterator first, InputIterator last,
             size_type table_size = 1009,
             const Hasher& hf = Hasher(),
             const KeyEqual& equal = KeyEqual());
    ~hash_set();

// copying, assignment, swap

    hash_set(const hash_set<Key, Hasher, KeyEqual, Allocator>& x);
    hash_set<Key, Hasher, KeyEqual, Allocator>&
       operator=(const hash_set<Key, Hasher, KeyEqual, Allocator>& x);
    void swap(hash_set<Key, Hasher, KeyEqual, Allocator>& x);

// accessors:

    key_equal key_eq() const;
    hasher hash_funct() const;
    iterator begin() const;
    iterator end() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

// insert/erase

    pair<iterator, bool> insert(const value_type& x);
    template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// search operations:
```

```
        iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        pair<iterator, iterator> equal_range(const key_type& x) const;

    // hash table size operations

        size_type bucket_count() const;
        void resize(size_type buckets);
    };

    template <class Key, class Hasher, class KeyEqual, Allocator>
    bool operator==(const hash_set<Key, Hasher, KeyEqual, Allocator>& x,
                   const hash_set<Key, Hasher, KeyEqual, Allocator>& y);
```

`iterator` is a constant forward iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the same type as `iterator`.

2.2.3 Hash_multiset

`hash_multiset` is a kind of hash table that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves.

```
    template <class Key, class Hasher=hash<Key>, class KeyEqual = equal_to<Key>,
              template <class U> class Allocator = allocator>
    class hash_multiset {
    public:

    // typedefs:

        typedef Key key_type;
        typedef Key value_type;
        typedef Hasher hasher;
        typedef KeyEqual key_equal;

        typedef pointer;
        typedef reference;
        typedef const_reference;
        typedef iterator;
        typedef iterator const_iterator;
        typedef size_type;
        typedef difference_type;

    // allocation/deallocation

        hash_multiset(size_type table_size = 1009,
                      const Hasher& hf = Hasher(),
                      const KeyEqual& equal = KeyEqual());
        template <class InputIterator>
        hash_multiset(InputIterator first, InputIterator last,
                      size_type table_size = 1009,
                      const Hasher& hf = Hasher(),
                      const KeyEqual& equal = KeyEqual());
        ~hash_multiset();

    // copying, assignment, swap
```

```
    hash_multiset(const hash_multiset<Key, Hasher, KeyEqual, Allocator>& x);
    hash_multiset<Key, Hasher, KeyEqual, Allocator>&
       operator=(const hash_multiset<Key, Hasher, KeyEqual, Allocator>& x);
    void swap(hash_multiset<Key, Hasher, KeyEqual, Allocator>& x);

// accessors:

    key_equal key_eq() const;
    hasher hash_funct() const;
    iterator begin() const;
    iterator end() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

// insert/erase

    iterator insert(const value_type& x);
    template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// search operations:

    iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x) const;

// hash table size operations

    size_type bucket_count() const;
    void resize(size_type buckets);
};

template <class Key, class Hasher, class KeyEqual, Allocator>
bool operator==(const hash_multiset<Key, Hasher, KeyEqual, Allocator>& x,
                const hash_multiset<Key, Hasher, KeyEqual, Allocator>& y);
```

`iterator` is a constant forward iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the same type as `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

2.2.4 Hash_map

hash_map is a kind of hash table that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys.

```
template <class Key, class Hasher=hash<Key>, class KeyEqual = equal_to<Key>,
          template <class U> class Allocator = allocator>
```

```
class hash_map {
public:

// typedefs:

    typedef Key key_type;
    typedef pair<const Key, T> value_type;
    typedef Hasher hasher;
    typedef KeyEqual key_equal;

    typedef pointer;
    typedef reference;
    typedef const_reference;
    typedef iterator;
    typedef const_iterator;
    typedef size_type;
    typedef difference_type;

// allocation/deallocation

    hash_map(size_type table_size = 1009,
             const Hasher& hf = Hasher(),
             const KeyEqual& equal = KeyEqual());
    template <class InputIterator>
    hash_map(InputIterator first, InputIterator last,
             size_type table_size = 1009,
             const Hasher& hf = Hasher());
    ~hash_map();

// copying, assignment, swap

    hash_map(const hash_map<Key, T, Hasher, KeyEqual, Allocator>& x);
    hash_map<Key,T,Hasher,KeyEqual, Allocator>&
      operator=(const hash_map<Key, T, Hasher, KeyEqual, Allocator>& x);
    void swap(hash_map<Key, T, Hasher, KeyEqual, Allocator>& x);

// accessors:

    key_equal key_eq() const;
    hasher hash_funct() const;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    Allocator<T>::reference operator[](const key_type& k);

// insert/erase

    pair<iterator, bool> insert(const value_type& x);
    template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// search operations:
```

```
    iterator find(const key_type& x);
    const_iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x);
    pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

// hash table size operations

    size_type bucket_count() const;
    void resize(size_type buckets);
};

template <class Key, class T, class Hasher, class KeyEqual, Allocator>
bool operator==(const hash_map<Key, T, Hasher, KeyEqual, Allocator>& x,
               const hash_map<Key, T, Hasher, KeyEqual, Allocator>& y);
```

`iterator` is a forward iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is a constant forward iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

In addition to the standard set of member functions of hash tables, `hash_map` provides `Allocator<T>::reference operator[](const key_type&)`. For a `hash_map` `m` and key `k`, `m[k]` is semantically equivalent to `(*((m.insert(make_pair(k, T()))).first)).second`.

2.2.5 Hash_multimap

`hash_multimap` is a kind of hash table that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys.

```
template <class Key, class Hasher=hash<Key>, class KeyEqual = equal_to<Key>,
          template <class U> class Allocator = allocator>
class hash_multimap {
public:

// typedefs:

    typedef Key key_type;
    typedef pair<const Key, T> value_type;
    typedef Hasher hasher;
    typedef KeyEqual key_equal;

    typedef pointer;
    typedef reference;
    typedef const_reference;
    typedef iterator;
    typedef const_iterator;
    typedef size_type;
    typedef difference_type;

// allocation/deallocation
```

```
    hash_multimap(size_type table_size = 1009,
                  const Hasher& hf = Hasher(),
                  const KeyEqual& equal = KeyEqual());
    template <class InputIterator>
    hash_multimap(InputIterator first, InputIterator last,
                  size_type table_size = 1009,
                  const Hasher& hf = Hasher(),
                  const KeyEqual& equal = KeyEqual());
    ~hash_multimap();

// copying, assignment, swap

    hash_multimap(const hash_multimap<Key, T, Hasher, KeyEqual, Allocator>& x);
    hash_multimap<Key, T, Hasher, KeyEqual, Allocator>&
      operator=(const hash_multimap<Key, T, Hasher,KeyEqual, Allocator>& x);
    void swap(hash_multimap<Key, T, Hasher, KeyEqual, Allocator>& x);

// accessors:

    key_equal key_eq() const;
    hasher hash_funct() const;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

// insert/erase

    iterator insert(const value_type& x);
    template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// search operations:

    iterator find(const key_type& x);
    const_iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x);
    pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

// hash table size operations

    size_type bucket_count() const;
    void resize(size_type size);
};

template <class Key, class T, class Hasher, class KeyEqual, Allocator>
bool operator==(const hash_multimap<Key, T, Hasher, KeyEqual, Allocator>& x,
                const hash_multimap<Key, T, Hasher, KeyEqual, Allocator>& y);
```

`iterator` is a forward iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the a constant forward iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

3 Hash Table Implementations

We have developed two separate reference hash table implementations that meet all of the above requirements. Both use the method of separate chaining, but with somewhat different ways of storing lists in the hash buckets. The major difference between the two implementations is the way they handle expansion. More detail about these implementations, including performance comparisons, are given in separate reports, as mentioned in the introduction.

In order to realize the expected constant time bounds for insertion, searching, and erasure that are required for hash tables (Time Complexity entries in Table 3), it is necessary that hash tables expand dynamically (increasing the number of buckets) as the number of items stored grows. A simple method of expansion is often used: allocate another, larger, table of buckets; rehash all items in the old table into the new table; and deallocate the old table. This method, which we call *intermittent resizing*, can be very time-consuming, perhaps prohibitively so in some applications requiring very fast response-times. A different implementation avoids this potential bottleneck by expanding (or contracting) the table on a gradual basis; it could be called *gradual resizing*. The method used is based mainly on one described in Per-Ake Larson, *CACM*, Vol. 31, Number 4, April 1988.

4 Acknowledgements

Alex Stepanov and Meng Lee provided helpful comments on the drafts of this document and suggested several significant improvements.