

# Tables in Practice

---

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, April 28, 2008

Glenn G. Chappell

Department of Computer Science  
University of Alaska Fairbanks

[CHAPPELLG@member.ams.org](mailto:CHAPPELLG@member.ams.org)

© 2005–2008 Glenn G. Chappell

# Unit Overview

## Tables & Priority Queues

---

### Major Topics

- ✓ • Introduction to Tables
- ✓ • Priority Queues
- ✓ • Heap algorithms
- ✓ • Heaps and Priority Queues in practice
- ✓ • 2-3 Trees
  - Other balanced search trees
  - Hash Tables
  - Prefix Trees
  - Tables in practice

# Tables in Practice

## Overview

---

We now take a brief look at Table usage in various languages, beginning with C++.

- C++ STL
  - Sets: `std::set`
  - Maps: `std::map`
  - Other Tables
  - Algorithms for sets
- Other Languages
  - Python
  - Perl
  - Lisp

## Tables in Practice

### Aside: `std::pair` [1/2]

---

The C++ STL contains a “pair” template: `std::pair`, in `<utility>`. It acts as if it is declared roughly like this:

- Note the public data members.

```
template<typename T, typename U>
struct pair {
    T first;
    U second;
};
```

Other members, including `operator<` and `operator==`, exist.

- So you can put a `std::pair` into a sorted container, as long as types `T` and `U` have `operator<`.
- You can also do `std::find` (Sequential Search) to look for a `std::pair`, as long as types `T` and `U` have `operator==`.

# Tables in Practice

## Aside: `std::pair` [2/2]

---

Example:

```
std::pair<int, double> p;  
p.first = 3;  
p.second = 4.5;  
// Or we can do the above using a ctor:  
std::pair<int, double> p2(3, 4.5);
```

What do we use a `std::pair` for?

- To return more than one value from a function.
  - Remember `fibonacci3.cpp`?
- To return a range as two iterators.
  - As in `std::equal_range`, a variation of Binary Search.
- To store a key-data pair.
  - For use in key-based look-up.

# Tables in Practice

## `std::set` — Introduction

---

The simplest STL Table implementation is `std::set`, in `<set>`.

- The key type and value type are the same.
  - That is, the key is the whole data item.
- Duplicate (equivalent) keys are not allowed.
  - Thus, all you can say about a value is whether or not it is in the structure.
  - In short, it is just what it says it is: a set.
- The specification was put together with a balanced search tree in mind. Most implementations will probably use a Red-Black Tree.

Declare a set as follows:

```
std::set<valuetype> s;
```

An optional template parameter specifies the comparison used.

- This is done just as for sorting, Heap algorithms, etc.
- The default is to use `operator<`.

```
std::set<valuetype, comparison> s;
```

# Tables in Practice

## `std::set` — Iterators

---

A `std::set::iterator` is a bidirectional iterator.

- Items appear in sorted order.

A `std::set::iterator` is not a **mutable iterator**, that is, one cannot do `*iter = v;`.

- Why not?
  - Because items are stored in sorted order. Changing an item might break this invariant.

Iterators and references are valid until the item is erased.

- How would this be implemented?
  - Clearly, *references* stay valid, because this is a node-based structure.
  - A Red-Black tree can be reorganized by an insertion or deletion. Thus, *iterators* must *not* store information about the structure of the tree (as they do in a `std::deque`).
  - So we must be able to find our way around the tree starting at a leaf. This means the tree must have **parent pointers**.
  - Conclusion: Give the tree parent pointers, and make the iterator a wrapper around a pointer.

# Tables in Practice

## `std::set` — Major Operations [1/2]

---

### Insert

- Given an item.
- Inserts given item into the set. Does nothing if an equivalent item (key) is already in the set.
- Returns a `std::pair<iterator, bool>`. The iterator points to the inserted item or the already present item. The `bool` is `true` if the insertion happened.
- Example:

```
std::set<int> s;  
s.insert(3);  
if (!s.insert(4).second)  
    cout << "4 was already present" << endl;
```

# Tables in Practice

## `std::set` — Major Operations [2/2]

---

### Erase

- Given a key **or** an iterator.
- Removes the proper item (if any) from the set.
- Examples:

```
s.erase(3);
```

```
s.erase(s.begin());
```

### Find

- Given a key.
- Returns an iterator, which either points to the item or is `end()`.
- Example:

```
If (s.find(3) != s.end())
```

```
    cout << "3 was found" << endl;
```

- Why not just use `std::find` or `std::binary_search`?
  - The former is always linear time. The latter is only efficient on random-access data.

## Tables in Practice

### `std::set` — Other

---

There are many other members in `std::set`, including `range insert` & `erase`, etc.

One interesting member function is “insert with hint”.

- This works like regular `insert`, but it is given an iterator, too. It returns an iterator to the item.
- The second parameter (iterator) is a “hint” as to where the item should be inserted.
- The code *may* ignore the hint, but it probably uses it.
- How do you think this is typically implemented?
  - Probably the inorder traversal property of a Red-Black Tree is used to look for locations “near” the given one.
- What is a good hint to give?
  - If you are inserted items in sorted order, a good hint is the location of the last item inserted.
- What is the likely effect of giving a bad hint?
  - Slower behavior [but still  $O(\log n)$ , as the Standard requires].

# Tables in Practice

## `std::map` — Introduction

---

The other main Table available in C++ is `std::map`, in `<map>`.

- The key and data types are specified separately.
- The value type is a pair: `std::pair<keytype, datatype>`.
- As with `std::set`:
  - Duplicate (equivalent) keys are not allowed.
  - The specification was put together with a balanced search tree in mind. The implementation is usually a Red-Black Tree.
  - An optional comparison can be specified. It defaults to using `operator<`.

Declaration:

```
std::map<keytype, datatype> m;
```

or

```
std::map<keytype, datatype, comparison> m;
```

## Tables in Practice

### `std::map` — Bracket Operator [1/3]

---

Major operations in `std::map` are much the same as for `std::set`.  
A very convenient operation is: *data* & `operator[]`(*key*)

- This allows a map to be used like an array. Examples:

```
std::map<std::string, int> m;  
m["abc"] = 7;  
cout << m["abc"] << endl;  
m["abc"] += 2;
```

- This is usually defined as follows ("*k*" is the given key):

```
(*((m.insert(value_type(k, data_type()))).first)).second
```

## Tables in Practice

### `std::map` — Bracket Operator [2/3]

---

More `operator[]` examples:

```
std::map<int, int> m2;
m[0] = 34;
m[123456789] = 28; // Very little memory used!

std::map<std::string, std::string> id;
id["Hubert Gump"] = "abc";
cout << id["Fred Smurg"] << endl;
// The above line inserts
//     std::pair<std::string, std::string>
//     ("Fred Smurg", std::string())
// into the map. (Right?)
```

# Tables in Practice

## std::map — Bracket Operator [3/3]

---

A map's operator[] is very convenient and useful. However ...

- This operator[] **always inserts**. Thus, it has no const version.

```
void printEntry(const std::map<std::string, int> & m1)
{
    cout << m1["abc"] << endl; // DOES NOT COMPILE!
}
```

- Because of this insertion, operator[] is generally *not* a good way to check whether a given key is already in the map. Instead, use map::find.

```
std::map<Foo, Bar> m2;
Foo theKey;
// I want to test whether theKey lies in m2

if (m2.find(theKey) != m2.end()) // GOOD way to test
if (m2[theKey] == ...)          // BAD way to test
```

# Tables in Practice

## `std::map` — Iterators

---

Iterators are much as for `std::set`.

- They are bidirectional iterators.
- Items appear in sorted order, by key.
- They are not **mutable**.
  - Thus, no `*iter = v;`.

However, one can do `(*iter).second = d;`.

- This makes sense, because the data part does not affect the organization of the structure.
- But how can we disallow the former, while allowing the latter?
- Answer: The value type is `std::pair<const key_type, data_type>`.



Note: This is okay, but we usually write `iter->second = d;`

# Tables in Practice

## Other Tables in the C++ STL

---

The C++ STL also has `std::multiset` and `std::multimap`.

- These are much the same as `std::set` and `std::map`, except that duplicate (equivalent) keys are allowed.
- There is no `operator[]` in `std::multimap`.
- Retrieve operations typically involve either returning a range or else counting the number of matching items.
- See the doc's.

The current (1998) C++ Standard includes no Hash Tables.

- However, many STL implementations include `std::hash_set`, `std::hash_map`, `std::hash_multiset`, `std::hash_multimap`.
- These are nonstandard and (sadly) their interfaces vary a bit from one implementation to another.
- The upcoming revised C++ standard is expected to include Hash-Table-based classes.

## Tables in Practice

### C++ STL Set Algorithms — Introduction

---

The C++ STL contains special algorithms that deal with **sorted sequences** that may not be random-access: the **set algorithms**.

- These are `includes`, `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference`.

Each of these takes two sorted sequences, each specified with two input iterators.

- Function `includes` returns a `bool`.
- For the others, the output is another sequence, written to a given output iterator (the 5<sup>th</sup> parameter).

How can we find the intersection of two **sets** and store the result as another **set**?

- It helps to use a special kind of iterator that does insertion.

# Tables in Practice

## Aside: Inserters

---

The C++ STL offers several iterators that insert into containers.

- All are defined in the header `<iterator>`.

A **back insertion iterator** calls the `push_back` member function.

- Function `std::back_inserter` returns such an iterator. This takes one parameter: the container, which must have a `push_back` member.
- Say `v` is a `vector<int>`. Then `std::back_inserter(v)` returns an iterator.
  - Doing `*iter++ = item;` with this iterator does `v.push_back(item);`

```
std::copy(iter1, iter2, std::back_inserter(v));
```

- The above **inserts** a copy of range `[iter1, iter2)` at the end of `v`.

There is also `std::front_inserter`, for doing `push_front`.

And there is `std::inserter`, which takes two parameters: a container `c` and an iterator `pos` into container `c`. It returns an iterator that does `pos = c.insert(item, pos);`

- Note: `insert-with-hint` for sets & maps looks like the above.

# Tables in Practice

## C++ STL Set Algorithms — Usage

---

Again, how can we find the intersection of two `sets`, and store the result as another `set`?

- Use `std::set_intersection`. This takes two sorted sequences and writes the result, as a sorted sequence, to a given output iterator.
- For the inputs, use the `begin` & `end` member functions of the two given sets.
- For the output, use an inserter (special iterator) for a third set.

```
#include <set>           // for std::set
#include <algorithm>     // for std::set_intersection
#include <iterator>      // for std::inserter
std::set<Foo> s1, s2, s3;
// Here, we initialize sets s1 and s2.
// Now, we want s3 to be the set intersection of s1, s2.
std::set_intersection(s1.begin(), s1.end(),
                     s2.begin(), s2.end(),
                     std::inserter(s3, s3.begin()));
```

# Tables in Practice

## C++ STL Set Algorithms — Do It

---

### TO DO

- Write a program that uses STL set algorithms.
  - Perhaps `std::set_intersection`?

*Done. See `usesetalg.cpp`,  
on the web page.*

# Tables in Practice

## Other Languages — Python

---

Python includes a Table type, called a “dictionary” or “dict”.

- Dictionaries are used for many things in Python.
  - For example, they are used for member-function look-up, which is always done at runtime.
- Dictionaries are Hash-Table based.
  - Built-in types have hash functions provided.
  - User-defined types can specify their own hash functions.
- Example:

```
d = { 1:"one", "hi":"ho", "two":2 } # d is a dict
x = d[1] # x should now be "one"
if 1 in d:
    print "1 was found"
for k in d: # Loop over keys
    print "key:", k, "data:", d[k]
```

# Tables in Practice

## Other Languages — Perl

---

Perl has had Tables for a long time.

- A Perl Table implementation is called a “hash”.
  - These are Hash-Table based, of course.
  - Example:

```
$H{1} = "one";           # H is a hash
$H{"hi"} = "ho";
print $H{"hi"}, "\n";   # Prints "ho"
@A = keys %H;           # Array of keys of hash H
foreach $K (keys %H)    # Loop over keys
{
    print "key: ", $K, " data: ", $H{$K}, "\n"
}
```

Note: This syntax is slated to change in Perl version 6.

## Tables in Practice

### Other Languages — Lisp

---

Lisp has had Tables for a long, long time.

- Remember that Lisp uses Binary-Tree-based lists of lists for **everything**.
  - Thus, we cannot expect Lisp to have a special Table type.
  - There are, however conventions for expressing Tables as lists.
- One kind: **property list**.
  - Constructed as "( *key1 data1 key2 data2 key3 data3* )".
  - Example: "( 1 2 hi ho )".
- Another kind : **association list**.
  - Something like this: "( ( *key1 data1* ) ( *key2 data2* ) )".
    - I'm told that the implementation uses memory a bit more efficiently, replacing pointers with values.
- Of course, these do not have very efficient operations.
  - But they also predate much of the intense research on key-based look-up in the 1960's. So we'll forgive them. **Maybe.**

# Review

## Introduction to Tables

---

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant (?)	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

### Idea #1: Restricted Table

- Perhaps we can do better if we do not implement a Table in its full generality.

### Idea #2: Keep a Tree Balanced

- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

### Idea #3: "Magic Functions"

- Use an unsorted array of key-data pairs. Allow array items to be marked as "empty".
- Have a "magic function" that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:

- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables