Tables in Various Languages

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, December 2, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

Review
Where Are We? — The Big Problem

Our problem for much of the rest of the semester:

- Store: a collection of data items, all of the same type.
- Operations:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

A solution to this problem is a **container**.

Generic containers: those in which client code can specify the type of data stored.

Unit Overview
Tables & Priority Queues

Major Topics

- ✓ Introduction to Tables ← Lots of lousy implementations
- ✓ Priority Queues
- ✓ Binary Heap algorithms
- ✓ Heaps & Priority Queues in the C++ STL

  } Idea #1: Restricted Table

- ✓ 2-3 Trees
- ✓ Other balanced search trees

  } Idea #2: Keep a Tree Balanced

- ✓ Hash Tables

  } Idea #3: "Magic Functions"

- ✓ Prefix Trees
- Tables in various languages

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

Overview of Advanced Table Implementations

We will cover the following advanced Table implementations.

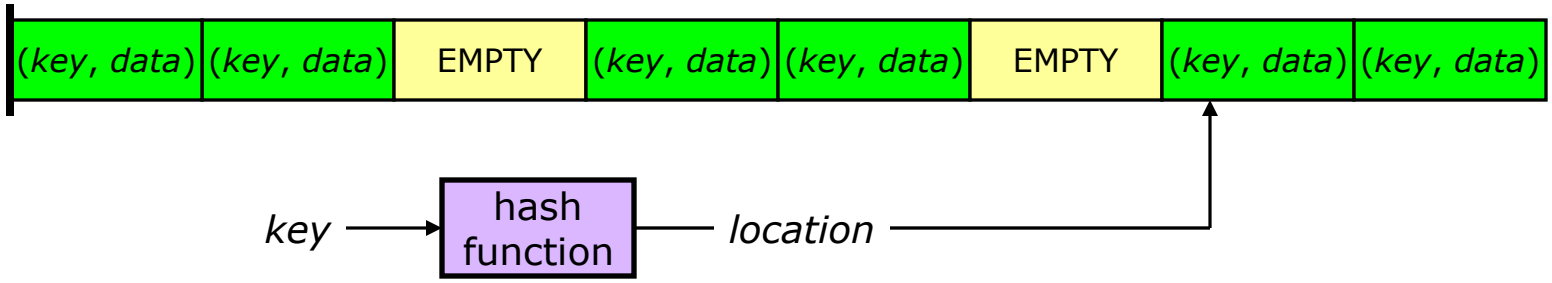
- Balanced Search Trees
  - Binary Search Trees are hard to keep balanced, so to make things easier we allow more than 2 children:
    - ✓ **2-3 Tree**
      - Up to 3 children
    - ✓ **2-3-4 Tree**
      - Up to 4 children
    - ✓ **Red-Black Tree**
      - A binary tree representation of a 2-3-4 tree
  - Or back up and try a balanced Binary Tree again:
    - ✓ **AVL Tree**
- Alternatively, forget about trees entirely:
  - ✓ **Hash Tables**
- Finally, "the Radix Sort of Table implementations":
  - ✓ **Prefix Tree**

DONE

Hash Tables — Introduction

A **Hash Table** is a Table implementation that uses a **hash function** for key-based look-up.

- A Hash Table is generally implemented as an array. The index used is the output of the hash function.



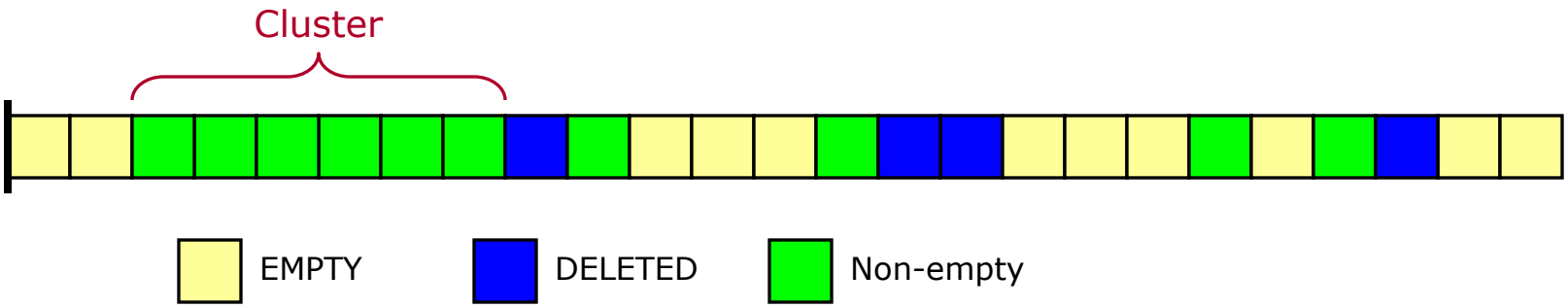
Needed:

- **Hash function**.
- **Collision resolution** method.
  - **Collision**: hash function gives same output for different keys.
  - This is the primary design decision involved in a Hash Table.

Hash Tables — Collision Resolution [1/2]

Collision Resolution Methods — Type 1: **Open Addressing**

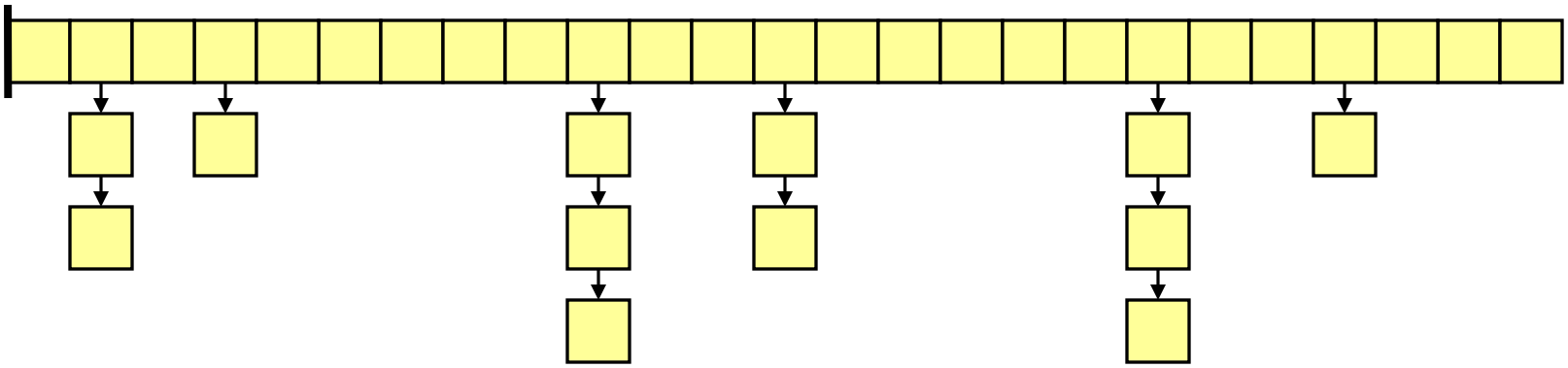
- Hash Table is an array. Each location holds one key-data pair, "empty", or "deleted".
- Search in a sequence of locations (the **probe sequence**), beginning at the location given by the hashed key.
- **Linear probing**: t, $t+1$, $t+2$, etc.
  - Tends to form **clusters**.
- **Quadratic probing**: t, $t+1^2$, $t+2^2$, etc.
- **Double hashing**: Use another hash function to help determine the probe sequence.



Hash Tables — Collision Resolution [2/2]

Collision Resolution Methods — Type 2: **"Buckets"**

- Hash Table is an array of data structures, each of which can hold multiple key-data pairs.
- Array locations are **buckets**.
- **Separate chaining**: Each bucket is a Linked List.
  - This is very common.



Review
Hash Tables — Efficiency

Idea #1 Idea #2 Idea #3

	Priority Queue using Heap	Red-Black Tree	Hash Table: average case	Hash Table: worst case
Retrieve	Constant*	Logarithmic	Constant	Linear
Insert	(Amortized)** logarithmic	Logarithmic	Amortized constant***	Linear****
Delete	Logarithmic*	Logarithmic	Constant	Linear

*Priority Queue retrieve & delete are not Table operations in their full generality. Only the item with the highest priority can be retrieved/deleted.

**This is logarithmic if (1) the PQ does not manage its own memory, or (2) enough memory is preallocated. Otherwise, occasional linear-time reallocate-and-copy may be required. Time per-operation, averaged over many consecutive operations, will be logarithmic. Thus, "amortized logarithmic".

***Hash Table insert is constant time in a "double average" sense: when averaged *both* over all possible inputs *and* over a large number of consecutive operations.

****This is amortized constant time if *both* of the following are true: (1) separate chaining is used, and (2) duplicate keys are allowed.

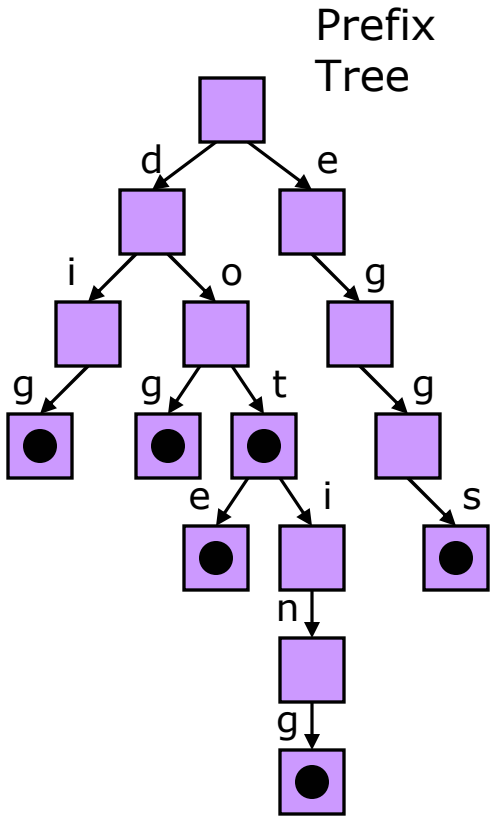
Review
Prefix Trees

A **Prefix Tree** (or **Trie**) is a Table implementation in which the keys are strings.

- We use "string" in a general sense, as in our discussion of Radix Sort.
  - A nonnegative integer is a string of digits.
- The quintessential key type is **words**, as in the previous slide.
- A Prefix Tree is space-efficient when keys tend to share prefixes.

A Prefix Tree is a kind of tree.

- Each node can have one child for each possible character.
- Each node also contains a Boolean value, indicating whether it represents a stored key.
  - Duplicate keys are not allowed.
- Lastly, each node can hold the data associated with a key.



Tables in Various Languages
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.
  - Set algorithms.
- Other Languages
  - Python.
  - Perl.
  - Lisp.

C++ STL: 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==**.

C++ STL: 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 **`fibo3.cpp`**?
- To specify a range, using two iterators.
  - As in the return value of **`std::equal_range`**, a Binary Search variation.
- To store a key-data pair.
  - For use in a Table implementation.

Tables in Various Languages
C++ STL: `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 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 Various Languages
C++ STL: `std::set` — Iterators

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

- Items appear in sorted order.
- So a **`std::set`** is pretty nearly a SortedSequence, right?

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.

- What does this tell us about the implementation?
  - 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.

C++ STL: `std::set` — Major Operations [1/2]

Table Insert (**`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.

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

Tables in Various Languages
C++ STL: `std::set` — Major Operations [2/2]

Table Delete (**`erase`**)

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

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

Table Retrieve (**`find`**)

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

```
If (s.find(3) != s.end())
   cout << "3 was found" << endl;
```

- Why not just use **`std::find`** or **`std::binary_search`** (or a variation)?
  - They both work.
  - Both are linear time, the former because it always is, the latter because this is not random-access data. However, **`set<K>::find`** is logarithmic time.

C++ STL: `std::set` — Other Operations

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 a second parameter: an iterator. 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 inserting items in sorted order, a good hint is an iterator to 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].

C++ STL: `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;
std::map<keytype, datatype, comparison> m2;
```

Major operations in **`std::map`** are much the same as for **`std::set`**.

- Insert operation: member function **`insert`**, given item.
- Delete operation: member function **`erase`**, given key or iterator.
- Retrieve operation: member function **`find`**, given key.

Different!

Tables in Various Languages
C++ STL: `std::map` — Bracket Operator [1/3]

A very convenient operation is: *datatype* **`& 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 can be defined as follows ("**k**" is the given key):

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

  - "Make sure key **k** is in the map, and give me the associated data."

C++ STL: `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 Various Languages
C++ STL: `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<K,D>::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
```

C++ STL: `std::map` — Iterators

Iterators for **`std::map`** 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;`**

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.

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 5th parameter).

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

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

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.

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()));
```

TO DO

- Write a program that uses an STL set algorithm.
  - Perhaps **`std::set_intersection`**?

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

Tables in Various Languages
TO BE CONTINUED …

Tables in Various Languages will be continued next time.