Thoughts on Assignment 8
Tables in the C++ STL & Elsewhere

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, November 22, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Thoughts on Assignment 8

Thoughts on Assignment 8
Overview

Assignment 8 is the last assignment this semester.

It includes two exercises:

- In Exercise A, you will write a complete C++ program (including `main`!) that uses an STL Table implementation.
- In Exercise B, you will write a test program, using the *doctest* framework, for a simple class.

Thoughts on Assignment 8
Exercise A [1/3]

In Exercise A you will write a program that is given a filename. It reads the file with that name and breaks it into words. Then it prints certain information about those words.

You are to choose an appropriate STL Table implementation and use it in your program.

The program needs to *work*, of course. But your choice of Table implementation, and proper use of it, will also be factor in the grading.

Thoughts on Assignment 8
Exercise A [2/3]

A **word** is a sequence of non-space characters.

For example, suppose your program is given the name of a file containing the following text.

```
dog dog? dog
dog dog?   cat
```

The above file contains 6 words having 3 distinct values. In lexicographic order, these values are the following:

- `cat`
- `dog`
- `dog?`

Thoughts on Assignment 8
Exercise A [3/3]

If it can read the file, then your program should do the following.

- Print a message indicating the number of *distinct* words in the file.
- Go through these words in lexicographic order. For each, print, on one line, the word, a colon, a blank, and then the number of times that word appears in the file.

For the given file, the following should be printed.

```
Number of distinct words: 3

cat: 1
dog: 3
dog?: 2
```

Text in file:

```
dog dog? dog
dog dog?   cat
```

Thoughts on Assignment 8 Exercise B [1/9]

In Exercise B you will write a test program for a class called `Reverser`. Your test program will use the *doctest* framework, just like the test programs for previous Assignments.

Objects of class `Reverser` are function objects. The function is a template that takes a range specified by two bidirectional iterators and reverses the order of the values in it.

So `Reverser` should be usable as follows.

```
Reverser rr;
vector<int> vv { 1, 9, 2 };
rr(begin(vv), end(vv));  // Now values in vv are 2, 9, 1
```

The idea is that class `Reverser` would be defined in a header `reverser.hpp`, with no associated source file. Your test program will be in file `reverser_test.cpp`.

Thoughts on Assignment 8 Exercise B [2/9]

We could write class `Reverser` as follows.

```
// class Reverser.
// Class invariants: None.
class Reverser {
public:
    // operator(). Given a range specified by two bidirectional
    // iterators. Reverses the order of the values in the range.
    // Throws what & when value type ops throw.
    // Basic guarantee
    // Exception neutral
    template<typename BDIter>
    void operator()(BDIer first, BDIter last) const
    { std::reverse(first, last); }

    // Default ctor, copy ctor, move ctor, copy =, move =, dctor:
    // automatically generated versions used.
};
```

This code would go in file `reverser.hpp`*.*

Thoughts on Assignment 8
Exercise B [3/9]

At the beginning of your test program (file `reverser_test.cpp`):

- `#include` the header for the code you wish to test.
- `#define` the symbol `DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN`, which tells *doctest* to write function `main`. (And then do *not* write `main`!)
- `#include` the *doctest* header (`doctest.h`).

```
// reverser_test.cpp
// By Moonface Malaprop
// 2024-11-22 (I do my work early!)
// Test program for class Reverser

#include "reverser.hpp"  // For class Reverser
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
                      // doctest writes main for us
#include "doctest.h"  // For doctest framework
```

Code something like this would go in your test program: file `reverser_test.cpp`*.*

Thoughts on Assignment 8
Exercise B [4/9]

After the initial includes, your test program should contain a number of **test cases**. Each will look something like this:

```
TEST_CASE("Reverser: ranges of size 2")
{
    …
}
```

A printable message identifying the test case.

Code something like this would go in your test program: file `reverser_test.cpp`*.*

Thoughts on Assignment 8
Exercise B [5/9]

Within a test case are one or more **tests**, which are done with a `REQUIRE` directive. Inside parentheses after `REQUIRE` is an expression of type `bool`, which is `true` if the test passes, and `false` otherwise.

Other code may be included in a test case.

```
TEST_CASE("Reverser: ranges of size 2")
{
    Reverser rr;
    deque<double> dd { 1.2, 5.2 };
    deque<double> dd_rev { 5.2, 1.2 };
    rr(begin(dd), end(dd));
    REQUIRE(dd == dd_rev);
    …
}
```

Code something like this would go in your test program: file `reverser_test.cpp`*.*

Thoughts on Assignment 8
Exercise B [6/9]

You should also include `INFO` directives. Each of these takes a string, which is printed if any following test fails.

```
TEST_CASE("Reverser: ranges of size 2")
{
    Reverser rr;
    deque<double> dd { 1.2, 5.2 };
    deque<double> dd_rev { 5.2, 1.2 };
    rr(begin(dd), end(dd));
    INFO("Reversing deque of size 2");
    REQUIRE(dd == dd_rev);
    …
}
```

Code something like this would go in your test program: file `reverser_test.cpp`*.*

Thoughts on Assignment 8
Exercise B [7/9]

Stream insertion (`<<`) may be used in an `INFO` directive.

```
TEST_CASE("Reverser: ranges of size 2")
{
    const size_t deque_size = 2;
    …
    INFO("Reversing deque of size " << deque_size);
    …
}
```

Code something like this would go in your test program: file `reverser_test.cpp`*.*

Thoughts on Assignment 8
Exercise B [8/9]

If you do not want the `INFO` string to be printed for *all* tests that follow in the test case, then you can put it, and the associated `REQUIRE`, in a **subcase**. The `INFO` string will go away at the end of the subcase.

```
TEST_CASE("Reverser: ranges of size 2")
{
    Reverser rr;
    SUBCASE("vector") {
        INFO("Reversing vector of size 2");
        …
    }
    SUBCASE("deque") {
        INFO("Reversing deque of size 2");
        …
    }
    …
```

Code something like this would go in your test program: file `reverser_test.cpp`*.*

Thoughts on Assignment 8
Exercise B [9/9]

Issues to consider when writing your test program:

- Does `Reverser` work for different data structures? `std::list`? `std::deque`? `std::vector`? `std::array`? `std::string`?
- Does `Reverser` work for empty ranges? For ranges of size 1?
- Does `Reverser` work for very large ranges?
- Is it clear that `Reverser` does not modify items just before or just after the range it is given?
- When a test fails, is the message printed both correct and helpful?

doctest has other features that you may wish to use: `CHECK` directives, `CAPTURE` directives, etc. You may use these if you wish; however, you are not required to use them.

I have provided a skeleton source file for a test program.

See `reverser_test.cpp`*.*

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL
- ✓ 2-3 Trees
- ✓ Other self-balancing search trees
- ✓ Hash Tables
- ✓ Prefix Trees
- Tables in the C++ STL & Elsewhere

Review

Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

Note the three primary single-item operations: **retrieve**, **insert**, **delete**. We will see these over & over again.

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

In a **generic container**, client code can specify the value type.

Review
Introduction to Tables

A **Table** allows for arbitrary key-based look-up.

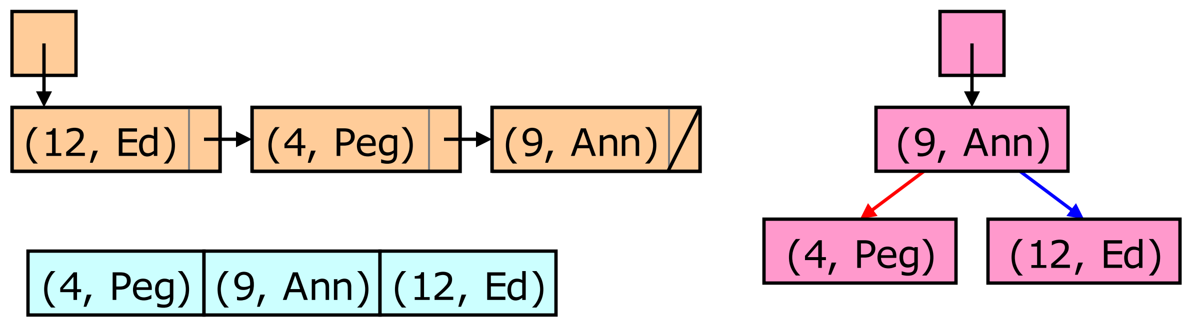
Three single-item operations: retrieve, insert, delete by key.

A Table implementation typically holds **key-value pairs**.

Table

Key	**Value**
12	Ed
4	Peg
9	Ann

Inefficient Implementations



Three ideas for improving efficiency:

1. Restricted Table → Priority Queues
2. Keep a tree balanced → Self-balancing search trees
3. Magic functions → Hash Tables

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables — Several lousy implementations
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL

Idea #1: Restricted Table

- ✓ 2-3 Trees
- ✓ Other self-balancing search trees

Idea #2: Keep a tree balanced

- ✓ Hash Tables

Idea #3: Magic functions

- ✓ Prefix Trees — A special-purpose implementation: "the Radix Sort of Table implementations"
- Tables in the C++ STL & Elsewhere

Overview of Advanced Table Implementations

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more children (?):
    - ✓ **2-3 Tree**
      - Up to 3 children
    - ✓ **2-3-4 Tree**
      - Up to 4 children
    - ✓ **Red-Black Tree**
      - Binary Tree representation of a 2-3-4 Tree
  - Or back up and try for a strongly balanced Binary Search Tree again:
    - ✓ **AVL Tree**
- Alternatively, forget about trees entirely:
  - ✓ **Hash Table**
- Finally, "the Radix Sort of Table implementations":
  - ✓ **Prefix Tree**

Idea #2:
Keep a tree balanced

Idea #3:
Magic functions

Later, we cover other self-balancing search trees: B-Trees, B+ Trees.

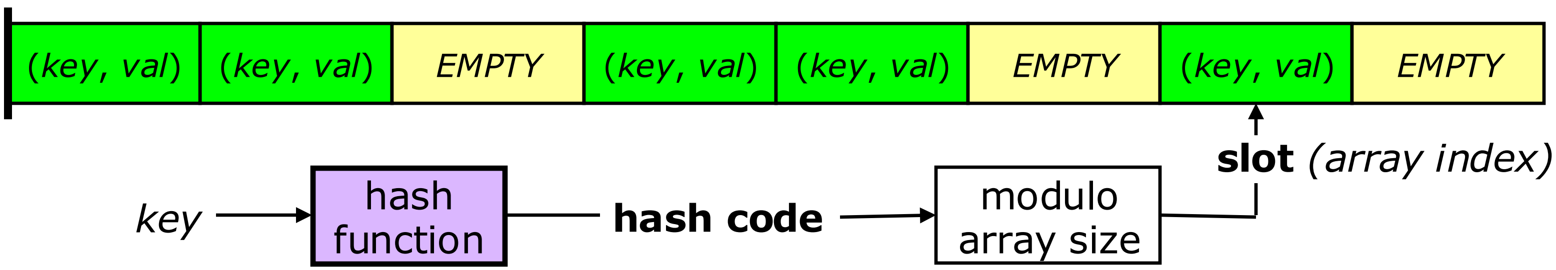
DONE

Review
Hash Tables [1/5]

A **Hash Table** is a Table implementation that stores key-value pairs in an unsorted array. Array indices are **slots**.

- A key's slot is computed using a **hash function**.
- An array location can be *EMPTY*.

Or just keys, if there are no associated values.

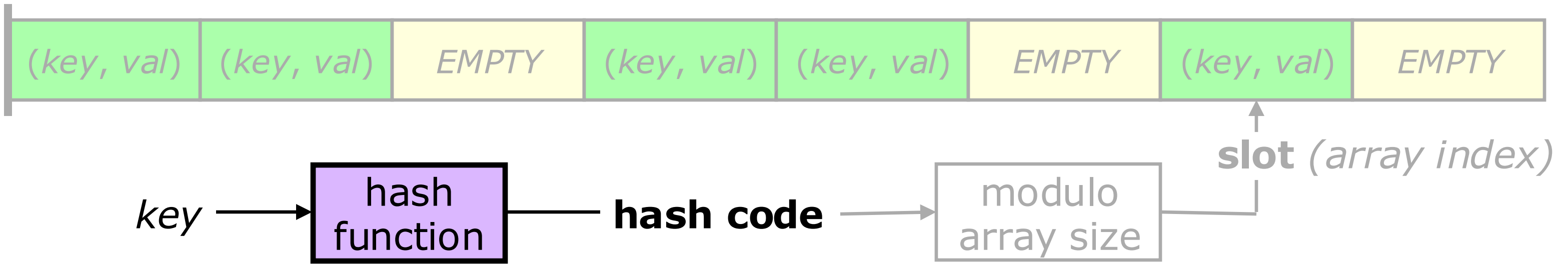


Collision: when an item gets a slot that already holds an item. The *possibility* of collisions is typically an unavoidable problem; there are often far more possible keys than slots.

Needed

- Hash function (typically separate from Hash Table implementation).
- **Collision-resolution** method.

Hash Tables [2/5]

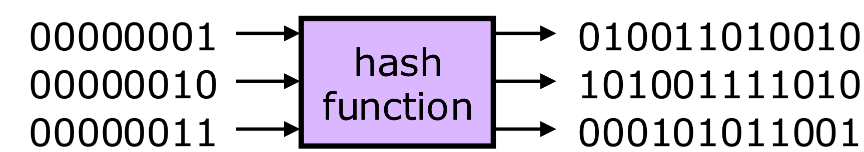


A hash function *must*:

- Take a key and return a nonnegative integer (**hash code**).
- Be **deterministic**: output depends only on input.
  A particular key always gives the same hash code.
- Return the same hash code for equal (==) keys.

Consistency requirement

A *good* hash function:

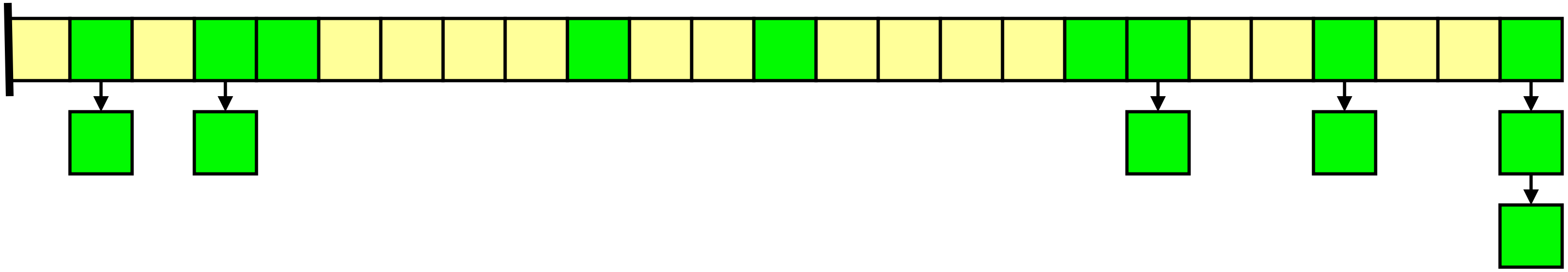


- Is fast.
- Spreads its results evenly
  over the possible output values.
- Turns patterns in its input into unpatterned output.

Hash Tables [3/5]

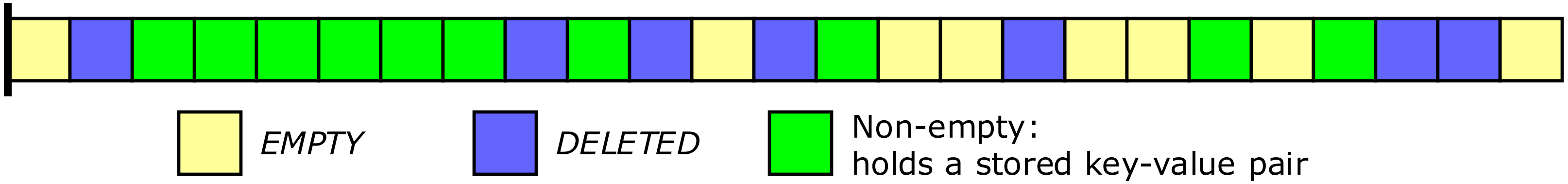
Collision resolution methods, category #1: **Open Hashing**

- An array item (**bucket**) can store multiple key-value pairs.
- Buckets are virtually always Singly Linked Lists.
- To find a key, determine which bucket to look in based on the hash code. Do a Sequential Search on that bucket.



Collision resolution methods, category #2: **Closed Hashing**

- An array item holds one key-value pair, or is *EMPTY* or *DELETED*.
- To find a key, begin at the slot given by the hash code, and **probe** in a sequence of slots: the **probe sequence**.

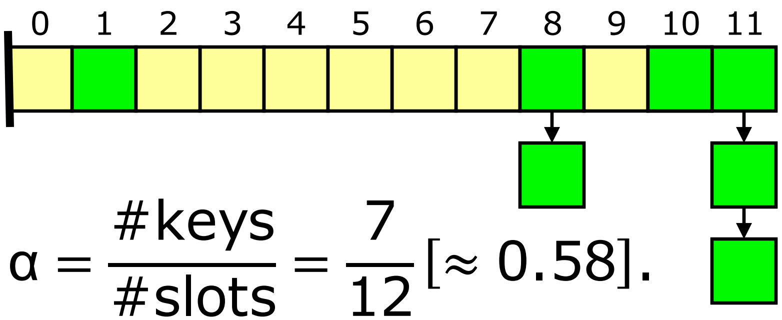


Review
Hash Tables [4/5]

Worst-case time for all of the usual operations is linear time.

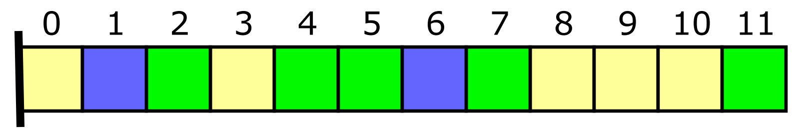
Average-case performance of a Hash Table can be analyzed based on its **load factor**: α = (# of keys present) / (# of slots).

Open Hashing



$$\alpha = \frac{\#\text{keys}}{\#\text{slots}} = \frac{7}{12} [\approx 0.58].$$

Closed Hashing



$$\alpha = \frac{\#\text{keys}}{\#\text{slots}} = \frac{5}{12} [\approx 0.42].$$

The load factor is kept small—usually well below 1.

Average-case time for *retrieve* and *delete* is constant time.

When the load factor gets too high, **rehash**: rebuild the Hash Table in a larger array. So average-case time for *insert* is amortized constant.

Review
Hash Tables [5/5]

Efficiency Comparison (duplicate keys not allowed)

	Idea #1	Idea #2	Idea #3	
	Priority Queue using Heap	Self-Balancing Search Tree	Hash Table: worst case	Hash Table: average case
Retrieve	Constant*	Logarithmic	Linear	Constant
Insert	Amortized** logarithmic	Logarithmic	Linear	Amortized constant***
Delete	Logarithmic*	Logarithmic	Linear	Constant

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

**Logarithmic if enough memory is preallocated. Otherwise, occasional reallocate-and-copy—linear time—may be required. Time per *insert*, averaged over many consecutive *inserts*, will be logarithmic. Thus, *amortized logarithmic time* (which is not a term I expect you to know).

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

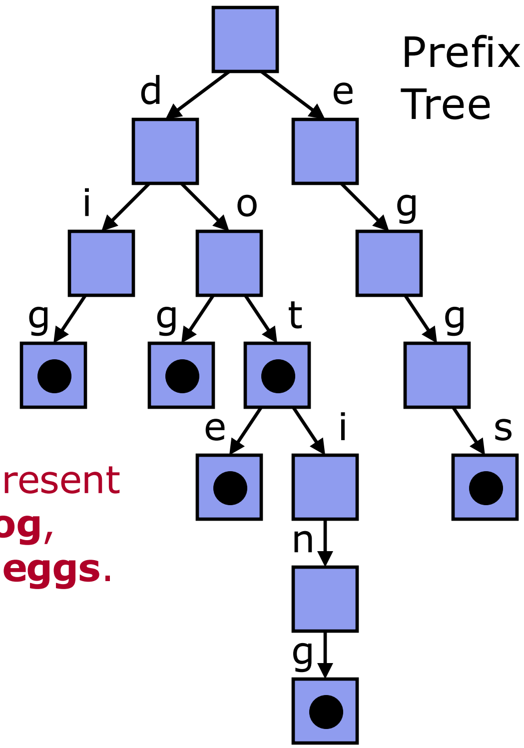
Review
Prefix Trees [1/2]

A **Prefix Tree** (a.k.a. **Trie**) is a Table implementation in which the keys are **strings**—in a general sense, as for Radix Sort.

A Prefix Tree is a kind of tree.

- A node has:
  - A Boolean—whether it represents a stored key.
  - Child pointers—one for each possible character.
  - The value associated with a key, if needed.

Time for operations is something like the length of a key. So constant-time *if length is considered fixed*.



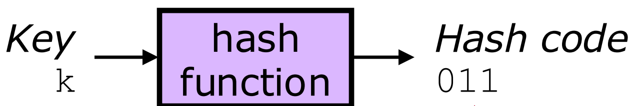
Nodes with dots represent stored keys: **dig**, **dog**, **dot**, **dote**, **doting**, **eggs**.

Prefix Trees are not difficult to implement well!

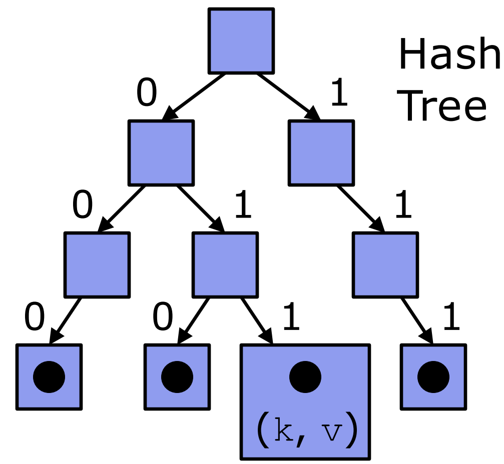
Review
Prefix Trees [2/2]

A **Hash Tree** is a Table implementation in which hash codes are used as keys for a Prefix Tree.

This allows some of the benefits of a Prefix Tree for a much more general class of keys—anything hashable.



This is a toy example. In practice, hash codes are longer, and this tree has greater height.



Tables in the C++ STL & Elsewhere

Tables in the C++ STL & Elsewhere
Overview

Now we look briefly at Table implementations as they exist in the C++ STL and also in the broader world of programming.

- C++ STL
  - Set: `std::set`
  - Key-value structure: `std::map`
  - Hash Table versions: `std::unordered_set`, `std::unordered_map`
  - Tables allowing duplicate keys
- Other Programming Languages
  - Perl
  - JavaScript
  - Python

C++ STL: std::set — Introduction

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

- An item is simply a key; there is no associated value.
- Duplicate (equivalent) keys are not allowed.
- The spec. was written with a self-balancing search tree in mind. Implementations will typically use a Red-Black Tree.

Declare a `std::set` as follows:

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

The comparison is specified as for `std::priority_queue`.

- Default: use `operator<`.
- Optional second template parameter: the *type* of the comparison.
- When writing the comparison as a lambda, pass the comparison itself as a constructor argument.

C++ STL: std::set — Iterators [1/2]

`std::set` has bidirectional iterators. `begin`/`end` work as usual.

Items appear in sorted order; `set` is basically a SortedSequence.

`set` does not have **mutable** iterators. We cannot do “`*iter = v;`”, and `begin`/`end` cannot be used to modify items.

Q. Why not?

A. Items are sorted. Changing an item could break this invariant.

Range-based for-loops can be used.

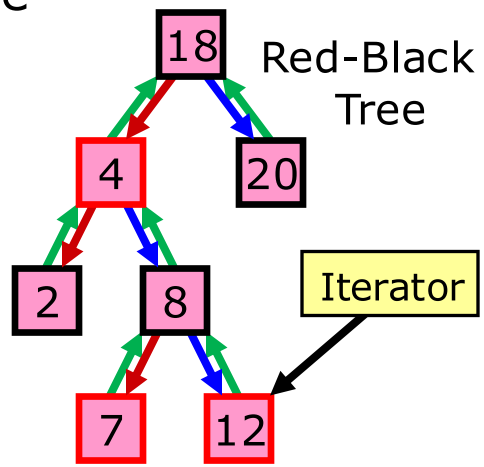
```
for (const auto & k : s)
{
    cout << "Key: " << k << endl;
}
```

C++ STL: std::set — Iterators [2/2]

`std::set` iterators and references are valid until the referenced item is destroyed.

What does this tell us about the implementation?

- A Red-Black tree may be reorganized by an insertion or deletion. So iterators must not store information about the structure of the tree.
- But we must be able to navigate around the tree efficiently, starting at a leaf node.
- So the tree must have **parent pointers**.
- An iterator can simply be a wrapper around a pointer to a node—together with algorithms for navigating the tree.



C++ STL: std::set — Operations [1/2]

Table Insert: member function `insert`

- Given an item (same as a key, for `set`). Inserts this into the set.
- **Does nothing** if an equivalent item (key) is already in the set.
- Returns `pair<`*iterator*`, bool>`. Iterator points to inserted item or already present item. The `bool` is `true` if the insertion happened.

```
set<int> s;
auto p = s.insert(3);
if (!p.second) cout << "3 was already present";
```

Table Delete: member function `erase`

- Given key *or* iterator. Deletes the proper item, if any, from the set.

```
s.erase(5);
s.erase(p.first());
```

C++ STL: std::set — Operations [2/2]

Table Retrieve #1: member function `find`

- Given a key. Returns iterator to the item, or `end()` if not found.

```
auto iter = s.find(3);
if (iter != s.end()) cout << "3 found";
```

Table Retrieve #2: member function `count`

- Given a key. Returns number of times key occurs in set (`0` or `1`).

```
if (s.count(3) != 0) cout << "3 found";
```

Why not use `std::find` or `std::binary_search` (or a variant)?

- Both work!
- But both are $\Theta(n)$: `find` because it does Sequential Search, `binary_search` because `std::set` is not random-access.
- However, member function `find` is $\Theta(\log n)$ [Red-Black Tree].

C++ STL: std::map — Introduction

The other main STL Table is `std::map`, in `<map>`.

- An item is a key along with associated data.
  - The key type & **data type** are both specified.
  - The value type is `pair<const` *keytype*`,` *datatype*`>`.
- Duplicate (equivalent) keys are not allowed.
- The spec. was written with a self-balancing search tree in mind.

STL-speak for *type of the associated value*.

Declare a `std::map` as follows:

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

An optional comparison can be specified. The default is `operator<`.

Most `map` operations are much the same as for `set`.

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

For `map`, these two are different!

C++ STL: std::map — Bracket Operator [1/3]

A very convenient operation: *datatype* `& operator[](`*key*`)`
  This allows a `map` to be used like an array. Examples:

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

`operator[]` is defined as follows (`k` is the given key):

```
(*((m.insert(make_pair(k, datatype()))).first)).second
```

Make sure key `k` *is in the* `map`*, and give me the associated value.*

C++ STL: std::map — Bracket Operator [2/3]

More `operator[]` examples:

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

map<string, string> id;
id["Cuthbert Gump"] = "abc";
cout << id["Frederica Murg"] << endl;
// The above line inserts
//     pair<string, string> ("Frederica Murg", string())
// into the map.
```

C++ STL: std::map — Bracket Operator [3/3]

`operator[]` for `map` is useful and convenient. However, it *always* calls `insert`. So it has no const version.

```
void printAbcValue(const map<string, int> & mm)
{
    cout << mm["abc"] << "\n";   // DOES NOT COMPILE!
}
```

Due to the insertion, `operator[]` is a poor way to check whether a key is already in the `map`. Use member function `count`.

```
map<Foo, Bar> m3;
Foo theKey;  // We want to check whether theKey is in m3
if (m3.count(theKey) != 0)   // GOOD way to check
if (m3[theKey] == …)         // BAD way to check
```

C++ STL: std::map — Iterators

Iterators for `map` are much as for `set`.

- They are bidirectional iterators.
- Items appear in sorted order, by key.
- They are not **mutable**. We cannot do "`*iter = v;`".

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

This is legal, but we would normally write `iter->second = d;`

Q. How is this possible?

A. The value type is `pair<const` *keytype*`,` *datatype*`>`.

Remember that a `std::map` item is a key-value pair.

```
for (const auto & kvpair : m)
{
    cout << "Key: "   << kvpair.first  << " "
         << "value: " << kvpair.second << endl;
}
```

Tables in the C++ STL & Elsewhere
C++ STL: Hash Tables

The 2011 C++ Standard added Hash Table versions of `set` & `map`:

- `std::unordered_set`, in header `<unordered_set>`.
- `std::unordered_map`, in header `<unordered_map>`.

These are very similar to `set` & `map`, respectively.

- Value types are identical.
- Member functions `insert`, `erase`, `find`, & `count` work the same.
- `unordered_map` has `operator[]`—which inserts.

Primary differences:

- Efficiency is as for a Hash Table, not a self-balancing search tree.
- Table traverse is not sorted—thus "unordered".
- No ordering is used. A custom **hash function** and **equality comparison** can be specified.

The interfaces were written with open hashing in mind. In particular, iteration through a single bucket is supported. (Why? I could not say.)

C++ STL: Tables Allowing Duplicate Keys

The STL also has Tables that allow duplicate keys:

- `std::multiset`
- `std::multimap`
- `std::unordered_multiset`
- `std::unordered_multimap`

Each is declared in the same header as its non-multi version.

- E.g., `std::multiset` is declared in header `<set>`.

Each is similar to its non-multi version. Important differences:

- The `count` member function may return values greater than `1`.
- `multimap` & `unordered_multimap` have no `operator[]`.
- In practice, when using these containers, we might deal with a *range* of items having equivalent/equal keys. Relevant member functions include `equal_range`, `lower_bound`, `upper_bound`.

Other Programming Languages: Perl

Possibly the first major general-purpose programming language to have built-in Tables was **Perl**.

A Perl Table is called a **hash**: Hash Table using open hashing.

One can optionally switch to a Red-Black Tree implementation.

This is Perl 5.
Perl 6 (renamed *Raku*) uses different syntax.

```
$H{1} = "one";               # H is a hash
$H{"hi"} = "ho";             # Multiple key types are allowed
$H{"two"} = 2;               # Similarly for associated values
print $H{"hi"}, "\n";        # Prints "ho"
@A = keys %H;                # A: array of the keys of hash H
foreach $K (keys %H)         # Loop over keys
{
    print "Key: ", $K, " data: ", $H{$K}, "\n"
}
```

Other Programming Languages: JavaScript

The main programming language for web scripting is **JavaScript**.

A JavaScript object *is* a Hash Table. Implementations vary.

- Keys are strings. Numbers may be used as keys, but they are converted to strings. Associated values may be of any type.
- Different associated-value types may be included in a single Table.

```
var ob = { 1:"one", "hi":"ho", "two":2 };
```

Lookup by key uses the bracket operator. When a key looks like an identifier, the dot operator may be used.

```
var a = ob["two"];  // a is 2
var b = ob.two;     // b is 2
var c = ob[1];      // c is "one"
var d = ob["1"];    // d is "one"
```

Python has several standard Table types. The main two:

- **Dictionary**: `dict`. Hash Table of key-value pairs.
- **Set**: `set`. Hash Table of keys.

```
dd = { 1:"one", "hi":"ho", "two":2 }  # dd is a dict
x = dd[1]  # x should now be "one"
if 1 in dd:
    print("1 was found")
for k in dd:                           # Loop over keys
    print("Key:", k, "value:", dd[k])
ss = { 34, "hello" }                   # ss is a set
```

As in Perl, different key types can be included in a single Table.

Dictionaries are used for *many* things in Python, including function & member look-up, which occurs at runtime.

Tables in the C++ STL & Elsewhere
Other Programming Languages: Python [2/2]

Several Python implementations exist. The standard: **CPython**.

- CPython built-in Hash Tables use closed hashing.
- The array size is a power of 2. The load factor is kept under 2/3.
- The probe sequence is illustrated by the following C code.

```
size_t hash_code, array_size;  // Hash code, # of slots

size_t perturb = hash_code;
size_t i = hash_code % array_size;  // A slot number
while (probe(i)) // Probe @ i; true: different key found
{
    i = (5*i + perturb + 1) % array_size;
    perturb /= 32;
}
// Now i is the slot where the key is (to be) stored.
```

Simplified version of part of the source code for CPython 3.14.0a2, file `dictobject.c`.