Hashing

1. Hash Functions
   - mapping data to tables
   - hashing integers and strings
2. Open Addressing
   - a class `Hash_Table`
   - inserting and locating strings
3. Chaining
   - a class `Hash_Table`
   - inserting and locating strings

MCS 360 Lecture 28
Introduction to Data Structures
Jan Verschelde, 27 October 2010

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing

1. Hash Functions
   - mapping data to tables
   - hashing integers and strings
2. Open Addressing
   - a class Hash_Table
   - inserting and locating strings
3. Chaining
   - a class Hash_Table
   - inserting and locating strings

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

mapping data to tables

We covered STL sets and maps for a frequency table.
The STL implementation uses a balanced search tree.

An alternative to using trees is hashing.
As running application, consider frequency tables.

We map counted objects first to a positive integer: the key,
and then we take `key % n`, for tables of size `n`.

Using a hash function h to store objects in table of size `n`:

The `index` is used in an array of `n` elements.

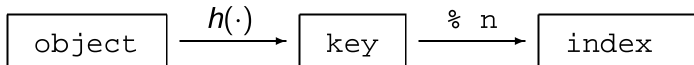
mapping data to tables

We covered STL sets and maps for a frequency table.
The STL implementation uses a balanced search tree.

An alternative to using trees is hashing.
As running application, consider frequency tables.

We map counted objects first to a positive integer: the key, and then we take `key % n`, for tables of size `n`.

Using a hash function h to store objects in table of size `n`:



The `index` is used in an array of `n` elements.

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hash Functions

The mapping of an object (e.g.: integer or string) to a key is done by a hash function h.

Desirable properties:

1. h is fast and easy to evaluate;
2. few collisions: $x \neq y \Rightarrow h(x) \neq h(y)$.

Some similarity with uniform random number generator, although distribution of data is often not uniform, we want distribution of the keys to be uniform.

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

Hash Functions

The mapping of an object (e.g.: integer or string) to a key is done by a hash function h.

Desirable properties:

1. h is fast and easy to evaluate;
2. few collisions: $x \neq y \Rightarrow h(x) \neq h(y)$.

Some similarity with uniform random number generator, although distribution of data is often not uniform, we want distribution of the keys to be uniform.

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

unsigned int

The type used to index arrays is `size_t`, which is 8 bytes long on 64-bit configurations, or otherwise 4 bytes long.

Including `<cmath>` allows

```
int n = 8*sizeof(unsigned int);
unsigned int m = pow(2.0,n-1);
cout << "  m = " << m << endl;
m = 2*m;
cout << "2*m = " << m << endl;
m = m - 1;
cout << "2*m - 1 = " << m << endl;
```

shows the modular arithmetic (modulo 2^{32})

```
  m = 2147483648
2*m = 0
2*m - 1 = 4294967295
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

unsigned int

The type used to index arrays is `size_t`, which is 8 bytes long on 64-bit configurations, or otherwise 4 bytes long.

Including `<cmath>` allows

```
int n = 8*sizeof(unsigned int);
unsigned int m = pow(2.0,n-1);
cout << "  m = " << m << endl;
m = 2*m;
cout << "2*m = " << m << endl;
m = m - 1;
cout << "2*m - 1 = " << m << endl;
```

shows the modular arithmetic (modulo 2^{32})

```
  m = 2147483648
2*m = 0
2*m - 1 = 4294967295
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

unsigned int

The type used to index arrays is `size_t`, which is 8 bytes long on 64-bit configurations, or otherwise 4 bytes long.

Including `<cmath>` allows

```
int n = 8*sizeof(unsigned int);
unsigned int m = pow(2.0,n-1);
cout << "  m = " << m << endl;
m = 2*m;
cout << "2*m = " << m << endl;
m = m - 1;
cout << "2*m - 1 = " << m << endl;
```

shows the modular arithmetic (modulo 2^{32})

```
  m = 2147483648
2*m = 0
2*m - 1 = 4294967295
```

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing

1. Hash Functions
   - mapping data to tables
   - hashing integers and strings
2. Open Addressing
   - a class `Hash_Table`
   - inserting and locating strings
3. Chaining
   - a class `Hash_Table`
   - inserting and locating strings

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing Integers

Take a prime p, $h(i) = (p \times i) \mod 2^{32}$.
Start with $i = 1$, find n so $p^n \mod 2^{32} = 1$?

```
unsigned int x = 1;
unsigned int i;
while(true)
{
    x = prime*x; i++;
    if(x == 1)
    {
        cout << "cycle detected at i = "
             << i << endl;
        break;
    }
}
```

```
cycle detected at i = 536870912
```

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing Integers

Take a prime p, $h(i) = (p \times i) \mod 2^{32}$.
Start with $i = 1$, find n so $p^n \mod 2^{32} = 1$?

```
unsigned int x = 1;
unsigned int i;
while(true)
{
   x = prime*x; i++;
   if(x == 1)
   {
       cout << "cycle detected at i = "
            << i << endl;
       break;
   }
}
```

```
cycle detected at i = 536870912
```

Hashing Integers

Take a prime p, $h(i) = (p \times i) \mod 2^{32}$.
Start with $i = 1$, find n so $p^n \mod 2^{32} = 1$?

```
unsigned int x = 1;
unsigned int i;
while(true)
{
   x = prime*x; i++;
   if(x == 1)
   {
       cout << "cycle detected at i = "
            << i << endl;
       break;
   }
}
```

```
cycle detected at i = 536870912
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

hashing strings

For a string $s = s_0 s_1 s_2 \cdots s_{n-1}$, compute

$$\begin{aligned} k &= s_0 31^{n-1} + s_1 31^{n-2} + s_2 31^{n-3} + \cdots + s_{n-1} \\ &= (\cdots((s_0 31 + s_1)31 + s_2)31 + \cdots)31 + s_{n-1}. \end{aligned}$$

Characters are 8 bits long; 31 is prime.

```
size_t hash ( string s )
{
   size_t r = 0;

   for(size_t i=0; i<s.length(); i++)
      r = r*31 + s[i];

   return r;
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

hashing strings

For a string $s = s_0 s_1 s_2 \cdots s_{n-1}$, compute

$$\begin{aligned} k &= s_0 31^{n-1} + s_1 31^{n-2} + s_2 31^{n-3} + \cdots + s_{n-1} \\ &= (\cdots((s_0 31 + s_1)31 + s_2)31 + \cdots)31 + s_{n-1}. \end{aligned}$$

Characters are 8 bits long; 31 is prime.

```
size_t hash ( string s )
{
   size_t r = 0;

   for(size_t i=0; i<s.length(); i++)
      r = r*31 + s[i];

   return r;
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

open addressing and chaining

The two ways to organize hash tables
differ in the ways of resolving collisions.

For some `index`, we consider

1. open addressing: if `index` occupied take next one;
   Locating an element is called *probing*.
2. chaining: keep list of elements at position `index`.
   The list is called a bucket, we speak of *bucket hashing*.

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing

1 Hash Functions
mapping data to tables
hashing integers and strings

2 Open Addressing
a class Hash_Table
inserting and locating strings

3 Chaining
a class Hash_Table
inserting and locating strings

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

a class Hash_Table

We define a class Hash_Table in the namespace mcs360_open_hash_table.

The hash table stores a frequency table:

- type of the key is a string,
- value type is an integer.

```
typedef std::pair<std::string,int> entry;
```

We create hash table with number of elements n, the capacity of the table.

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

mcs360_open_hash_table.h

```
#ifndef MCS360_OPEN_HASH_TABLE_H
#define MCS360_OPEN_HASH_TABLE_H
#include <utility>
#include <string>
#include <vector>
namespace mcs360_open_hash_table
{
   class Hash_Table
   {
      public:
         typedef std::pair<std::string,int> entry;
         Hash_Table( size_t n );

         bool insert( const entry& e );
         // returns true if e is inserted well
         int value( std::string s );
         // returns -1 if s is not found
```

private members

```
private:

   size_t hash ( std::string s );
   // returns an index in the table
   // applying a hash function to the string s

   std::vector<entry*> table;
   // data member
```

Our hash table will have a fixed capacity, determined at instantiation.

Enlarging the hash table requires *rehashing*.

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

private members

```
private:

   size_t hash ( std::string s );
   // returns an index in the table
   // applying a hash function to the string s

   std::vector<entry*> table;
   // data member
```

Our hash table will have a fixed capacity, determined at instantiation.

Enlarging the hash table requires *rehashing*.

test code snippets

```
Hash_Table T(n);

while(true){                    // inserting
   string w; cin >> w;
   Hash_Table::entry e;
   e.first = w;
   cin >> e.second;
   if(T.insert(e))    // rest omitted
}

while(true){                    // searching
   string w; cin >> w;
   int v = T.value(w);
   if(v == -1)            // rest omitted
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

test code snippets

```
Hash_Table T(n);

while(true){                // inserting
   string w; cin >> w;
   Hash_Table::entry e;
   e.first = w;
   cin >> e.second;
   if(T.insert(e))    // rest omitted
}

while(true){                // searching
   string w; cin >> w;
   int v = T.value(w);
   if(v == -1)          // rest omitted
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

test code snippets

```
Hash_Table T(n);

while(true){                // inserting
   string w; cin >> w;
   Hash_Table::entry e;
   e.first = w;
   cin >> e.second;
   if(T.insert(e))    // rest omitted
}

while(true){                // searching
   string w; cin >> w;
   int v = T.value(w);
   if(v == -1)             // rest omitted
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

constructor and hash function

```
Hash_Table::Hash_Table( size_t n )
{
   for(int i=0; i<n; i++)
      table.push_back(NULL);
}

size_t Hash_Table::hash ( std::string s )
{
   size_t r = 0;

   for(size_t i=0; i<s.length(); i++)
      r = r*31 + s[i];

   return (r % table.size());
}
```

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing

1. Hash Functions
   - mapping data to tables
   - hashing integers and strings
2. Open Addressing
   - a class `Hash_Table`
   - inserting and locating strings
3. Chaining
   - a class `Hash_Table`
   - inserting and locating strings

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

insert() method

```
bool Hash_Table::insert( const entry& e )
{
   size_t i = hash(e.first);
   if(table[i] == NULL)
   {
      table[i] = new entry(e);
      return true;
   }
   else if(table[i]->first == e.first)
   {
      table[i]->second = e.second;
      return true;
   }
```

insert() continued

```
    else
    {
       for(int j=1; j<table.size(); j++)
       {
          size_t k = (i+j) % table.size();
          if(table[k] == NULL)
          {
             table[k] = new entry(e);
             return true;
          }
          else if(table[k]->first == e.first)
          {
             table[k]->second = e.second;
             return true;
          }
       }
       return false;
   }
}
```

search the value

```
int Hash_Table::value( std::string s ) {
   size_t i = hash(s);
   if(table[i] == NULL)
      return -1;
   else if(table[i]->first == s)
      return table[i]->second;
   else {
      for(int j=1; j<table.size(); j++) {
         size_t k = (i+j) % table.size();
         if(table[k] == NULL)
            return -1;
         else if(table[k]->first == s)
            return table[k]->second;
         }
         return -1;
      }
   }
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

evaluation of open addressing

Advantage: no other data structure needed to deal with collisions.

Disadvantages:

- clusters may overlap;
  alternative to linear probing: quadratic probing
  in quadratic probing, increments form quadratic series:

$$1^2 + 2^2 + 3^2 + \cdots + i^2$$

- deleted items must be marked and cannot be deallocated because of open addressing.

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

evaluation of open addressing

Advantage: no other data structure needed
to deal with collisions.

Disadvantages:

- clusters may overlap;
  alternative to linear probing: quadratic probing
  in quadratic probing, increments form quadratic series:

$$1^2 + 2^2 + 3^2 + \cdots + i^2$$

- deleted items must be marked and cannot be
  deallocated because of open addressing.

evaluation of open addressing

Advantage: no other data structure needed to deal with collisions.

Disadvantages:

- clusters may overlap;
  alternative to linear probing: quadratic probing
  in quadratic probing, increments form quadratic series:

$$1^2 + 2^2 + 3^2 + \cdots + i^2$$

- deleted items must be marked and cannot be deallocated because of open addressing.

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing

1. Hash Functions
   - mapping data to tables
   - hashing integers and strings
2. Open Addressing
   - a class `Hash_Table`
   - inserting and locating strings
3. Chaining
   - a class `Hash_Table`
   - inserting and locating strings

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

a class Hash_Table

In the namespace mcs360_chain_hash_table
we define a Hash_Table.

The hash table stores a frequency table:

- type of the key is a string,
- value type is an integer.

```
typedef std::pair<std::string,int> entry;
```

We create a hash table with number of elements n,
the capacity of the table.

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

mcs360_chain_hash_table.h

```
#ifndef MCS360_CHAIN_HASH_TABLE_H
#define MCS360_CHAIN_HASH_TABLE_H
#include <utility>
#include <string>
#include <vector>
#include <list>
namespace mcs360_chain_hash_table
{
   class Hash_Table
   {
      public:
         typedef std::pair<std::string,int> entry;
         Hash_Table( size_t n );

         bool insert( const entry& e );
         // returns true if e is inserted well
         int value( std::string s );
         // returns -1 if s is not found
```

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

private members

The public part of this `Hash_Table` is the same as with open addressing

$\Rightarrow$ use the same interactive test

```
private:

   size_t hash ( std::string s );
   // returns an index in the table
   // applying a hash function to the string s

   std::vector< std::list<entry> > table;
   // data member
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

private members

The public part of this `Hash_Table` is the same as with open addressing
$\Rightarrow$ use the same interactive test

```
private:

   size_t hash ( std::string s );
   // returns an index in the table
   // applying a hash function to the string s

   std::vector< std::list<entry> > table;
   // data member
```

constructor and hash function

```
Hash_Table::Hash_Table( size_t n )
{
   for(int i=0; i<n; i++)
   {
      std::list<entry> L;
      table.push_back(L);
   }
}

size_t Hash_Table::hash ( std::string s )
{
   size_t r = 0;

   for(size_t i=0; i<s.length(); i++)
      r = r*31 + s[i];

   return (r % table.size());
}
```

Hash Functions
mapping data to tables
hashing integers and strings

Open Addressing
a class Hash_Table
inserting and locating strings

Chaining
a class Hash_Table
inserting and locating strings

Hashing

1. Hash Functions
   - mapping data to tables
   - hashing integers and strings
2. Open Addressing
   - a class `Hash_Table`
   - inserting and locating strings
3. Chaining
   - a class `Hash_Table`
   - inserting and locating strings

the insert() method

```
bool Hash_Table::insert( const entry& e )
{
   size_t i = hash(e.first);

   table[i].push_back(e);

   return true;
}
```

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

search a value

```
int Hash_Table::value( std::string s )
{
   size_t i = hash(s);

   if(table[i].size() == 0)
      return -1;
   else
   {
      std::list<entry> L = table[i];

      for(std::list<entry>::const_iterator
          k = L.begin();
          k != L.end(); k++)
         if(k->first == s) return k->second;

      return -1;
   }
}
```

evaluation of hashing

Resolving collisions with chaining:

- collisions affect only collided keys,
- flexible insert and effective delete,
- at expensive of `list` data structure.

The STL list takes care of bucket management.

We covered very basic implementations, left to do:

1. write delete methods
2. templates for key and value type
3. iterators on the tables

Hash Functions
mapping data to tables
hashing integers and strings
Open Addressing
a class Hash_Table
inserting and locating strings
Chaining
a class Hash_Table
inserting and locating strings

evaluation of hashing

Resolving collisions with chaining:

- collisions affect only collided keys,
- flexible insert and effective delete,
- at expensive of `list` data structure.

The STL list takes care of bucket management.

We covered very basic implementations, left to do:

1. write delete methods
2. templates for key and value type
3. iterators on the tables

Summary + Assignments

Covered more of chapter 9, elements of §9.3, 9.4, and 9.5.

Assignments:

1. Download a text from `www.gutenberg.org` and write code to read the words from file. Use the STL `map` to make a frequency table of the keys applying on the strings the function `hash` of this lecture. Study the number of collisions for various sizes of the table.
2. To hash strings, would it help to apply the hash function $h(i) = (p \times i) \mod 2^{32}$ on the key returned by the hash function on strings? Justify your answer, eventually referring to your answer to the previous exercise.

more assignments

3. Run the code on a small table with 3 entries to simulate collisions. Add printing statements and sketch the evolution of the hash table as more elements are inserted. Compare open addressing with chaining.
4. For the hash table with open addressing, modify the code of `insert()` to use quadratic probing. Illustrate with an example, choosing appropriate table size and sequence of insertions, how quadratic probing may reduce the effect of clustering.

Homework collection on Friday 29 October, at noon:
#1,2 of L-21; #1,2 of L-22; and #4 of L-23.