Data Structures We've Discussed

- array/vector: refer to element by numerical position
- linked list: refer to element starting from the head of the list
- queues and stack: refer to element relative to top/front
- trees: refer to element starting from the root (or min node) and by its ancestors.

data structure	build	insert	find
vector	O(n)	O(1) amortized	O(n)
sorted vector	O(n log n)	O(n)	O(log n)
set or map	O(n log n)	O(log n)	O(log n)
list	O(n)	O(1)	O(n)
sorted list	O(n log n)	O(n)	O(n)

Faster Search than a Balanced Binary Search tree?

n, O(Log(n)), O(1)?

Hash Tables!

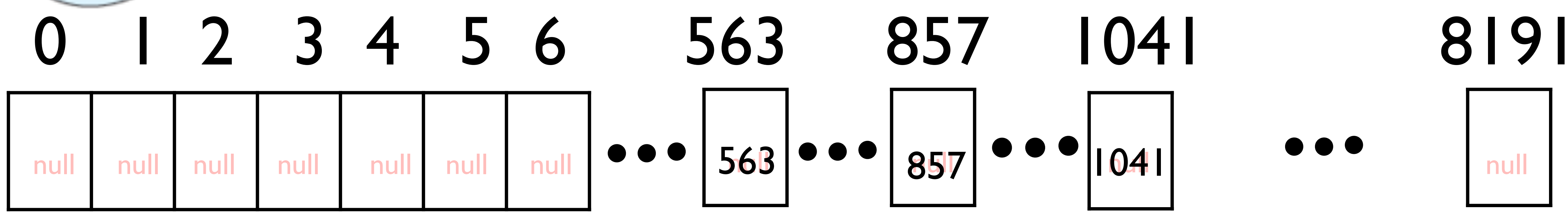
Items are ***not*** inserted in sorted order

Operations such as findMin, findMax, or printing all the items in sorted order cannot be done efficiently

The general idea…

This technique is called **direct addressing** and is useful if the items stored have a small range of possible values

How about putting the items in an array?



Universe of keys = Student id #'s (range 0 to 8191)

Actual keys: Class list: 563, 857, 1041,...

Advantages:
Constant time insert, delete, lookup!

Problem: Size of array is MUCH larger than the number of items.

Time-Space Tradeoff

Hash Function:
rule to associate key to index

How about computing a function that determines the index into the array?

Hash Function: h(Student ID#)= (Student ID#) mod 61

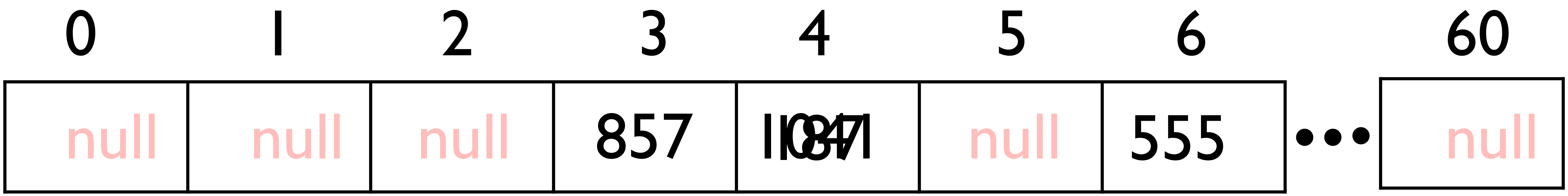
0	1	2	3	4	5	6		60
null	null	null	857	null	null	555	•••	null

857?

h(857)=3

Keys: Class list: 555, 857, ~~1041~~,...

What could go wrong?????



h(187)=4 !!!!!!!!

Keys: Class list: 555, 857, 1041, 187...

Definitions

hash function h: a function used to determine where the items belong in an array

hash table T: an array to store the items

Collision: when two keys map to the same value

Solutions:

Time-space tradeoff!
- no memory limitation... have a huge array and no collisions!
- no time limitation... use sequential search and no wasted memory

1) Choose a better *hash function* - the book goes into detail on how to choose a good hash function.

Hashing time memory tradeoff changes by adjusting the parameters.

2) Resolve the collision
   a) Linear probing
   b) Quadratic probing
   c) Double hashing
   d) Chaining

3) Resize array if load factor is close to one (costly option.) load factor= (#keys stored)/(size of array)

Linear Probing:

- Insert k:
  - Compute h(k).
  - If position h(k) of table is occupied, try position h(k)+1, h(k)+2, h(k)+3, etc. (wrapping around to beginning if necessary), until find empty slot. Insert k in empty slot.
- Find k:
  - Compute h(k)
  - If position h(k) of table is occupied, check h(k)+1, h(k)+2, etc. until either find k or find unoccupied slot. If the latter, then report “k not found”

0	1	2	3	4	5	6	7	8	…	60
243	null	null	857	1041	187	555	309	null	…	365

426?

h(426)=60

Keys: Class list: 555, 857, 1041, 187, 309, 365, 243,...

A REALLY BAD IDEA!!

0	1	2	3	4	5	6	7	8		60
243	null	null	857	null	187	555	309	null	•••	365

309?

h(309)=4

Keys: Class list: 555, 857, 1~~04~~1, 187, 309, 365, 243, ...

More on linear probing

- Deletion is a bit problematic. Can’t just make the slot unoccupied since this could mess up future finds. Could mark the slot as “formerly occupied”, and treat it as occupied during finds (but allow insertion into it).
- Find and Insert (and Delete) take time O(n) in the worst case.

Instead mark location as formerly occupied!

0	1	2	3	4	5	6	7	8		60
Formerly Occupied	null	null	857	Formerly Occupied	187	555	Formerly Occupied	null	•••	null

309?

$h(309)=4$

Keys: Class list: 555, 857, 1~~0~~41, 187, 3~~0~~9, 2~~4~~4,...

Clustering

- Find and insert are slow when elements in the hash table are "clustered" – that is, many occupy contiguous slots
- Will get big clusters when "load factor" of table approaches 1

  load factor = (# keys stored)/(size of array)
- Clustering can also be problem if hash function and key values together cause
  1. lots of collisions and/or
  2. keys to be assigned to contiguous slots.

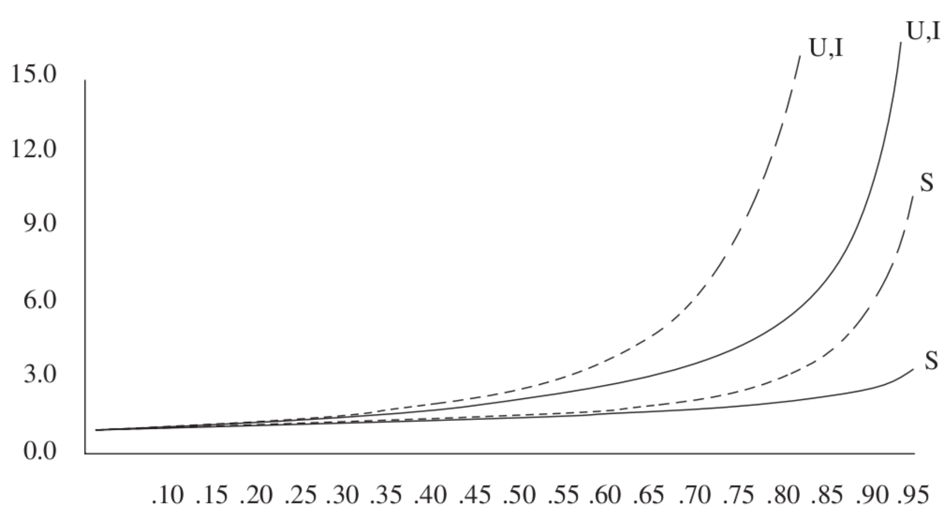


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

Weiss, Mark A. (2013-06-11). Data Structures and Algorithm Analysis in C++ (4th Edition)

Expected #Probes for insertion

- Under the *simple uniform hashing assumption,* the expected number probes to insert is approximately $1/2(1+1/(1-\lambda)^2)$ where λ=load factor.
  - If λ=0.5 then approx. 2.5 probes are need on average
  - If λ=0.75 then approx. 8.5 probes are need on average
  - If λ=0.9 then approx. 50 probes are need on average

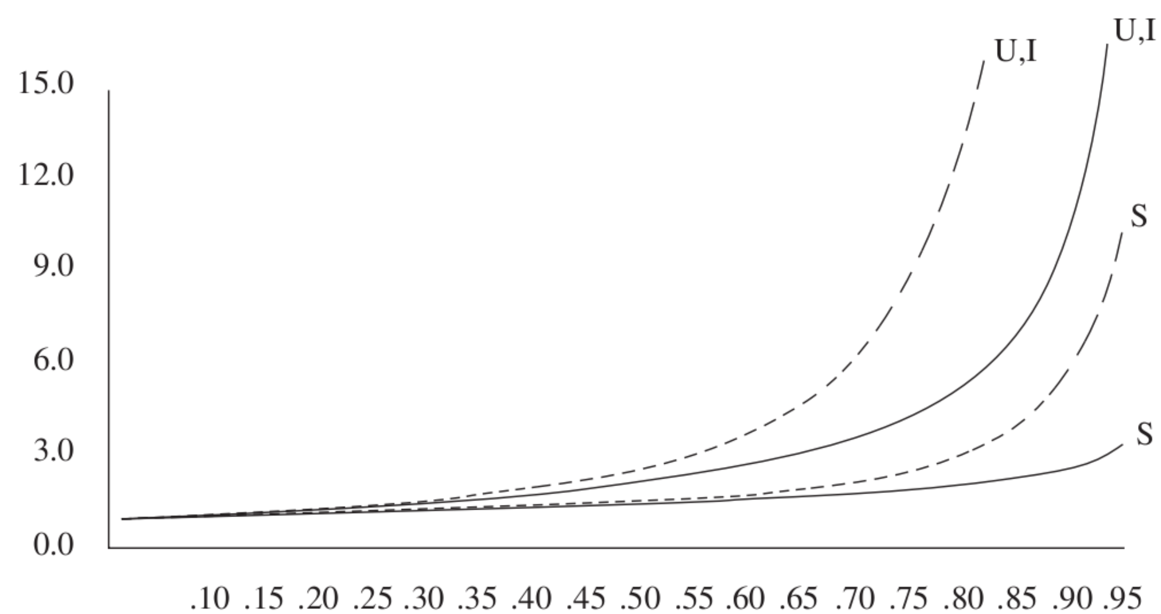
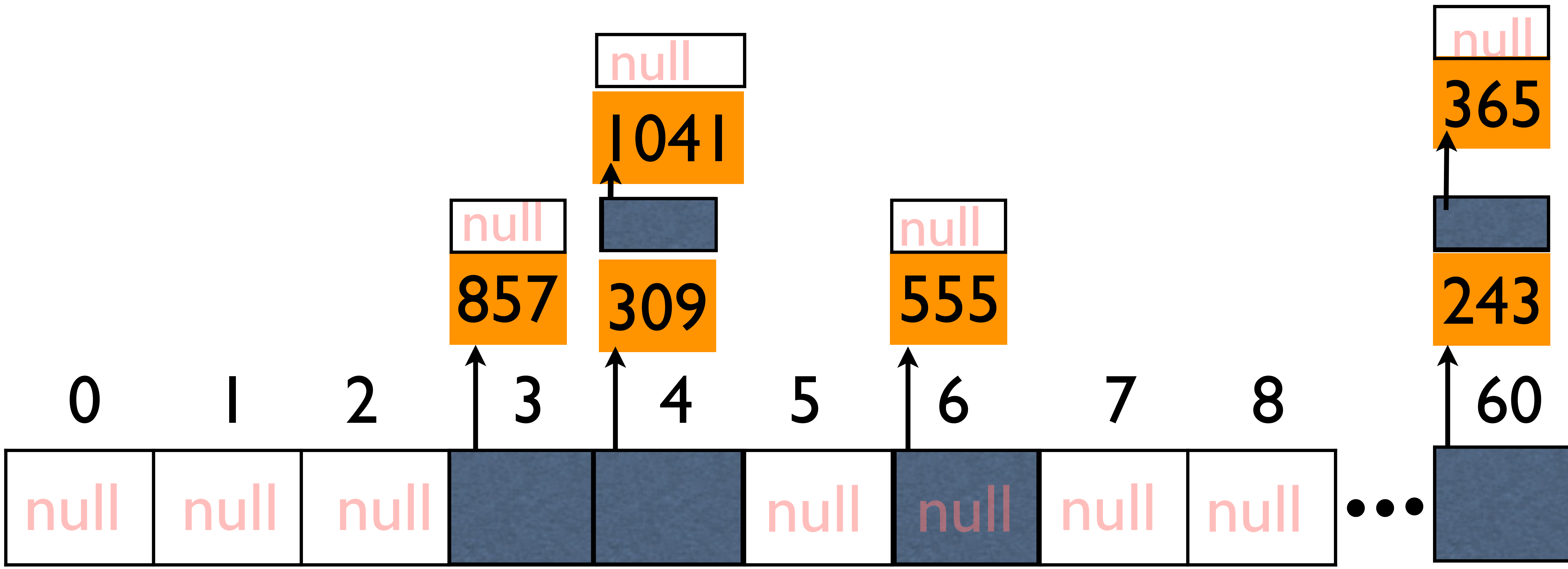


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

Weiss, Mark A. (2013-06-11). Data Structures and Algorithm Analysis in C++ (4th Edition)

Chaining

- In linear probing, store all keys in the array
- Alternative is chaining. Use an array of lists. Store elements with same hash value j in linked list at position j.
- Insertion, deletion, find are easy
  - Calculate h(k)
  - Do insert, delete, or find in list at position h(k)
- Worst-case is when all elements hash to same value, end up in same chain (list)
  - Search, Delete, Find take time O(n) worst case
  - But if chains are constant length, operations take constant time, O(1)
- Resize if too many keys in table
  - Average length of chain = load factor



426? h(426)=60

370? h(370)=4

Keys: Class list: ~~555~~, 857, 1041, ~~187~~, 309, 365, 243,...

3 MAIN CHOICES

Search performed on the part of the data called the key.

The size of the array is called the table size

The hash function should be:
- computable in constant time (fast)
- distribute the items uniformly among the array slots
- equal keys produce same value

- Array size - depends on the data
- Hash Function: a function that maps keys to indices in the array
- Collision Strategy: linear probing, quadratic probing, double hashing, chaining

A Different Hash Function

Hash Function: h(Student name)= (Student name) mod 11

0	1	2	3	4	5	6	...	10
null	null	null	formerly occupied	Lincoln	Jefferson	null	•••	null

If the table is large, and the size of the key is at most 9, the keys map to 0 through 9*127=1143. Thus only part of the table would be used

Not a good hash functions!! Used because it is easy to understand

Lincoln?

h("Lincoln")=(76+105+110+99+111+108+110)mod 11=4

Keys: Class list: Abraham Lincoln, Thomas Jefferson, John Ken~~n~~edy,...

Yet Another Hash Function

Hash Function: h(Student id#)= (Student id#)^3 mod 10

0	1	2	3	4	5	6	7	8	9
null	21	null	7	null	formerly occupied	null	null	null	null

21?

$h(21)=(21)^3 \bmod 10=1$

Keys: 7, ~~5~~, 21

Some Hash functions

Hash function should ideally be simple to compute and should map different keys to different values.

Modular hashing: h(k) = k mod m

Folding hashing(shift and boundary): Divide key into parts, and then the parts are combined.

- *Fold Shift*: the parts are added together.
- *Fold Boundary*: the parts are folded (reversed) and added together.

Mid Square hashing: h(k) = midvalue(k*k). Note that most all values contribute to the result. As a variation, select only a portion of the key to square.

Extraction hashing: Selected digits are of the key are used as the address.

Hash functions gone bad...

keys are social security numbers e.g. 123-45-6789

- hash function uses first three digits.
- first three digits are geographic location....

keys are english words at least 3 characters long

conation
concatenate
concave
concavities
conceal
concede
conceit
conceivable
concentration
...

```
// assumes strings have at least 3 characters
unsigned int hash( const string & key, int tableSize )
{
    return ( key[0] + 27 * key[1] + 729*key[2] ) % tableSize;
            //aka (1*key[0] + 27*key[1] + 27^2*key[2])%tableSize;
}
```

27 equals the number of alphabetic letters (26) + blank space (1)

- table size, M = 10,007
- 26^3 = 17,576 possible combinations
- English is not random....
- A large dictionary hashed to only 2,851 keys...

Only 28% of the table can be hashed to!

Code and second example from:
Weiss, Mark A. (2013-06-10). Data Structures and Algorithm Analysis in C++

A Hash Function for strings

Converting a hash function into an integral type. (Later we ensure it is an array index.)

This hash function would be too slow if the string was long (e.g. a complete address. it would be better to choose a couple of characters from each part of the address)

Our hash function doesn't know the table size

```
//hash function, in global scope
size_t hash( const string & key)
//This hash function is simple, reasonably fast.
{
    // uses Horner's rule to efficiently compute
    // the sum of key[i]*37^i
    size_t hashVal = 0;
    for (int  j=0; j < key.size(); j++)
        hashVal = 37*hashVal + key[j];
    return hashVal;
}
```

If the string is long - this would take too much time - an alternative is to just use part of the string

Unsigned int so when overflow happens the number doesn't turn negative!

The STL has a hash function object class (#include <functional>) It returns a size_t value. "Other function object types can be used as Hash for unordered containers provided they … are at least copy-constructible, destructible function objects."

Implementing Separate Chaining Hashing

Creating a Hash Table Using Separate Chaining

```
template <class Object>
class HashTable
{
public:
    explicit HashTable( int size = 101, int loadFactor = 1 ):array(size),num(0), MAXLOADFACTOR(loadFactor) {}

    bool contains( const Object & x ) const;
    bool insert( const Object & x );
    bool insert( Object && x );
    bool remove( const Object & x );
    void makeEmpty( );

private:
    int num;
    double MAXLOADFACTOR;
    vector< list<Object> > array;
    std::hash<Object> my_hash;

    void rehash( );
};
```

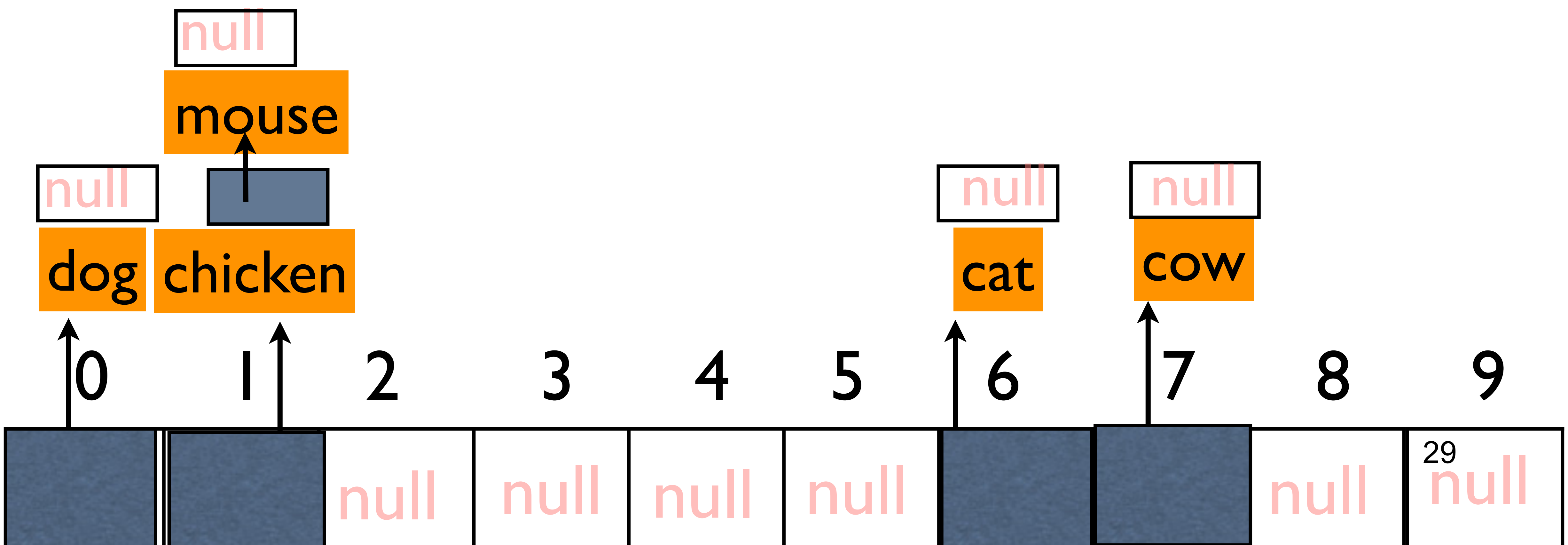
Hash Tables often assume the items stored in the hash table have == or != defined (or both)

Our implementation does not have the same interface as the STL unordered_map class.

Finding a Object into the Hash Table

```
template <class Object>
bool HashTable<Object>::contains(const Object & x)const
{

        size_t i = my_hash(x) % array.size();

        typename list<Object>::const_iterator itr;

        itr = std::find( array[i].begin(), array[i].end(), x );

        return ( itr == array[i].end() ) ? false: true;

      // return std::find( array[i].begin(), array[i].end(), x ) != array[i].end();

}
```

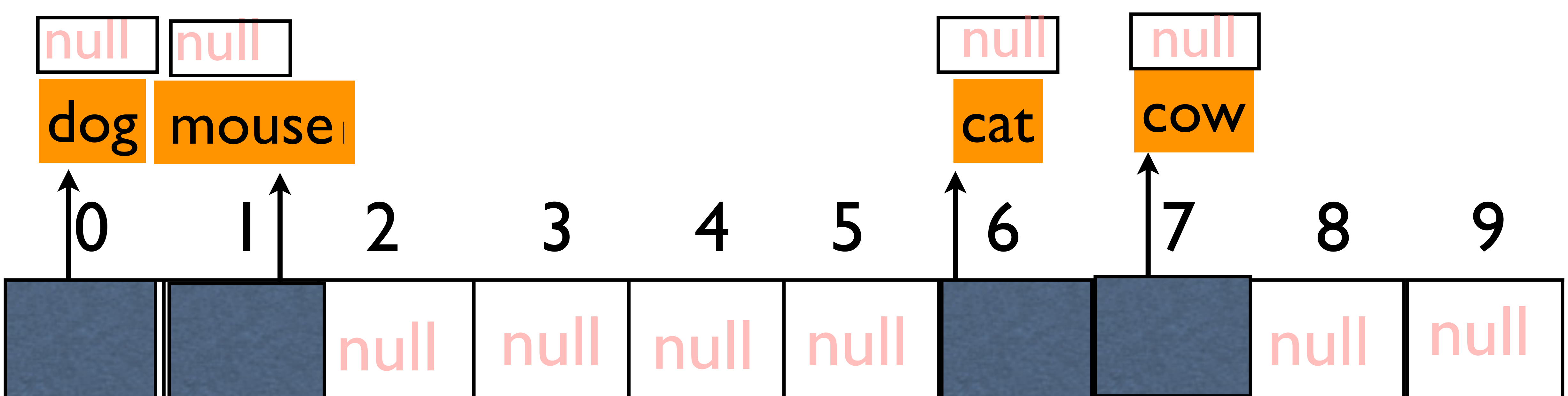
could be replaced by one line…



Inserting a Object into the Hash Table

```
template <class Object>
bool HashTable<Object>::insert(const Object & x)
{       if ( contains(x) )
              return false;

        size_t i = my_hash(x) % array.size();

         array[i].push_front( x );

         if ( ++num >= array.size() * MAXLOADFACTOR )
             rehash();

         return true;



}
```

Insert “chicken”



Increasing the size of the Hash Table

The STL The implementation of the rehab in the STL ensures that all iterators remain valid. Our implementation is easier to understand, but not as efficient and does not maintain iterator validity.

```
template<class Object>
void HashTable<Object>::rehash()
{

   vector< list<Object> > oldarray = std::move( array );
    array.clear();
    num = 0;

    array.resize(2*oldarray.size()+1);
   // better to resize to a prime size
    for (int i=0; i < oldarray.size(); i++)
    {
        for ( typename list<Object>::iterator itr = oldarray[i].begin(); itr != oldarray[i].end(); itr++ )
              insert( std:: move(*itr) );
    }

}
```

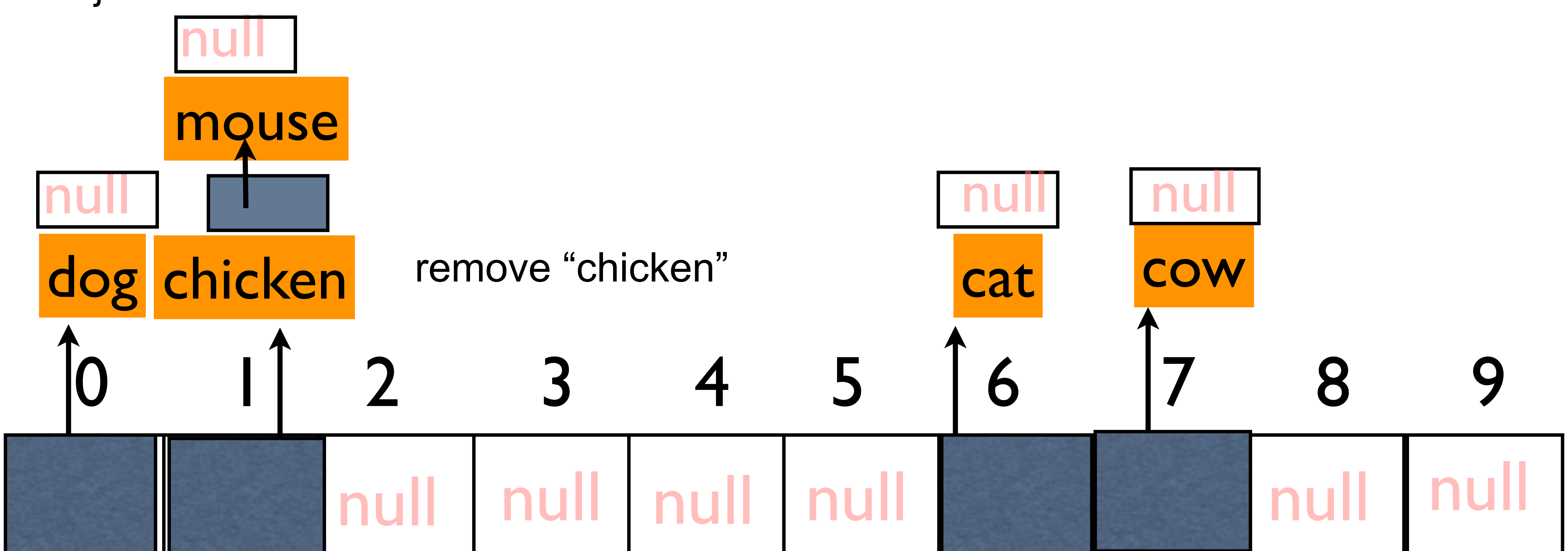
After we move the items resources from the old vector, we don't know what state the new vector is in. So, to make sure it is empty we call the clear method

Ideas on how to implement an iterator for the hash table: http://bannalia.blogspot.com/2013/10/implementation-of-c-unordered.html

Removing an object in the Hash Table

```
template <class Object>
bool HashTable<Object>::remove(const Object & x)
{
        size_t i = my_hash(x) % array.size();
        typename list<Object>::iterator itr;

        itr = std::find( array[ i ].begin(), array[i].end(), x );
        if ( itr == array[ i] .end() )
            return false;

        array[ i ].erase( itr );
        - - num;
        return true;
}
```

Uses the STL find algorithm, and the STL erase method from the list class



Separate Chaining Analysis

- Under the *simple uniform hashing assumption,* the expected number items in a cell of the array is n/m = λ=load factor.
- If the hash function takes O(1) time to compute, then the average time to insert is O(1 + λ)
  - If λ=1 then the expected time is O(1)
- If the hash function takes O(1) time to compute, then the average time to unsuccessfully search is O(1 + λ)
  - If λ=1 then the expected time is O(1)
- If the hash function takes O(1) time to compute, then the average time* to successfully search is O(1 + λ)
  - If λ=1 then the expected time is O(1)
- If the hash function takes O(1) time to compute, then the average time to delete an item is O(1 + λ)
  - If λ=1 then the expected time is O(1)

* A tight analysis is more difficult because we have to think about when the item we are searching for was inserted

Hash Table

- Advantages:
  - Very fast* for insert, delete, find (especially if the amount of data is known in advance)

- Disadvantages:
  - Need to find a “good” hash function
  - Cannot efficiently find “nearest” neighbor
  - Time consuming to resize hash table

* Unless too many keys are hashed to the same value

Choosing a poor hash function...

- Invalidate *simple uniform hashing assumptions*
- which means we don't get constant running time...

Difficult to prove a hash function is good

- Some choices clearly don't work:
  - h(name) = |name|
  - h(phone number) = |number of digits|

Cryptographic hash functions take too much time

Adversary….

Solution?

Randomness

☺

Universal Hashing

Choosing the hash function independently of the keys to be stored!

choose h randomly from:

$\mathcal{H} = \{h_1, h_2, \ldots, h_w\}$ a finite collection of hash functions with certain mathematical properties

Universal Class of Hash Functions ☺

$\mathcal{H} = \{h_1, h_2, \ldots, h_w\}$ is a universal hash function family if

for all $k, k' \in U$ $\quad \Pr_{h \in \mathcal{H}}\left[h(k) = h(k')\right] \leq \frac{1}{m}$

where m is the table size

HOW CAN WE CONSTRUCT A UNIVERSAL HASH FUNCTION FAMILY

Universal Family $\mathcal{H}$

p(p-1) choices for a and b

Theorem

p is larger than any key

Table size

$$\mathcal{H} = \{ H_{a,b}(x) = ((ax+b) \bmod p) \bmod M,$$

$$1 \le a \le p-1, 0 \le b \le p-1 \}$$ is universal

Examples:

$$H_{3,9}(x) = ((3x+9) \bmod p) \bmod M$$

$$H_{7,3}(x) = ((7x+3) \bmod p) \bmod M$$

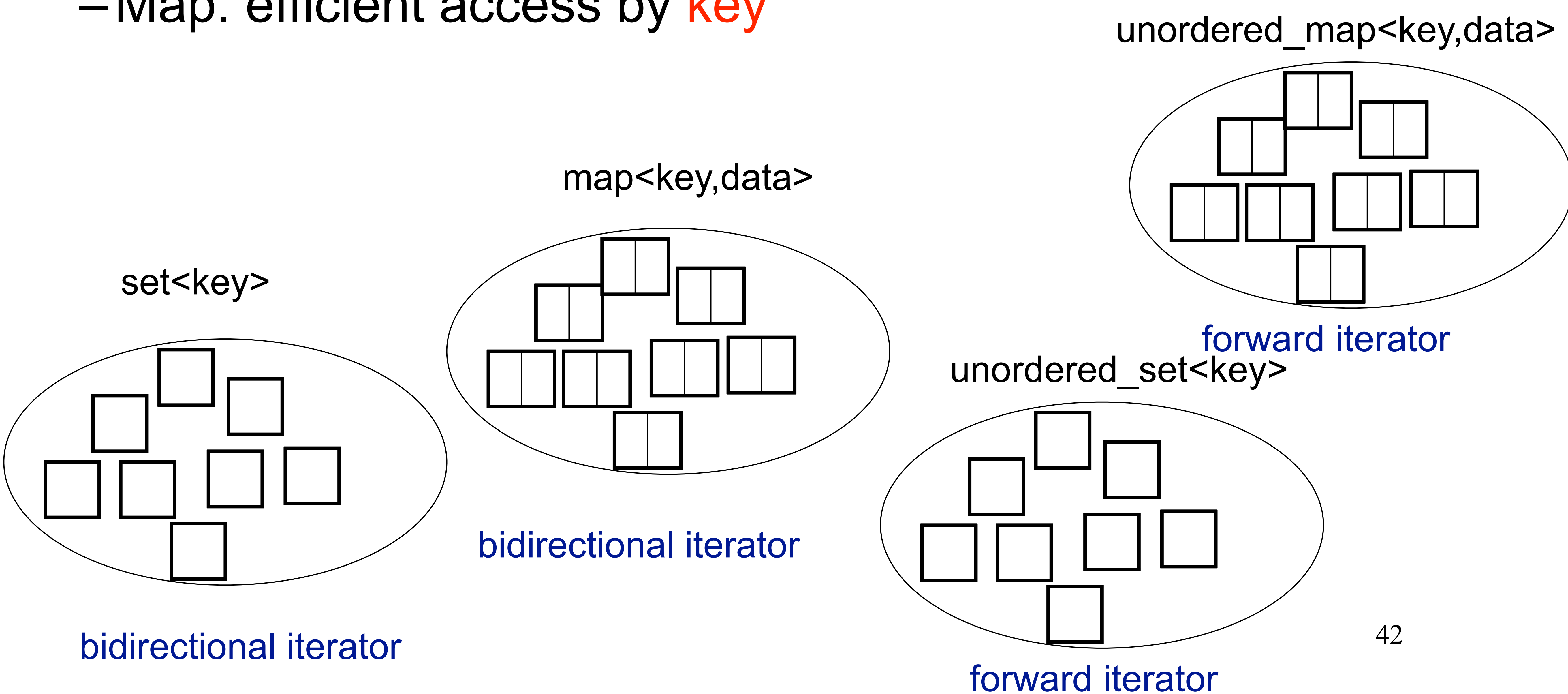
$$H_{4,0}(x) = ((4x) \bmod p) \bmod M$$

Hashing Applications

- counting number of times something occurs
- keeping track of declared variable in source code. The data structure is called a *symbol table* (compiler application)
- dispatch table (“if see this do that”)
- online spell checkers
- transposition table used in game programming to determine if a game position has occurred again

Associative containers

– Can't insert element into particular position
– Elements stored have key and (maybe) value, access by key
  • E.g. Key is Social Security Number, value is employee data
– Set: Elements stored by key, but no value, access by key
– Map: efficient access by key



This list is not complete. Check expert-level resource for more info.

#include<unordered_map>
Objects must have overloaded operator==

unordered_map
forward iterator

*Template class, so can store arbitrary elements. (char, int, custom classes).
*No order to the elements (stored by hash value.)
*When using a custom class, a custom hash function needs to be provided.

	average case:	worst case:
• u.insert(pair)	O(1)	O(n) iterators remain valid if a rehash is done
• u.find(key)	O(1)	O(n)
• u.size()	O(1)	O(1)
• u.begin()	O(1)	O(1)
• u.end()	O(1)	O(1)
• u[key]	O(1)	O(n) iterators remain valid if a rehash is done
• u.clear()	O(n)	O(n)
• u.erase(key) & u.erase(iterator)	O(1) & O(1)	O(n)

unordered_map example

```
// unordered_map::insert
#include <iostream>
#include <string>
#include <unordered_map>
using namespaces std;

int main ()
{
  unordered_map<string,double>
              myrecipe,
              mypantry = {{"milk",2.0},{"flour",1.5}};

  pair<string,double> myshopping ("baking powder",0.3);

  myrecipe.insert (myshopping);                        // copy insertion
  myrecipe.insert (mypantry.begin(), mypantry.end());     // range insertion
  myrecipe.insert ( {{"sugar",0.8},{"salt",0.1}} );      // initializer list insertion

  return 0;
}
```

Example modified from http://www.cplusplus.com/reference/unordered_map/unordered_map/insert/

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

int main()
{
    std::unordered_map<std::string, int> months;
    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;
    std::cout << "september -> " << months["september"] << std::endl;
    std::cout << "april     -> " << months["april"] << std::endl;
    std::cout << "december  -> " << months["december"] << std::endl;
    std::cout << "february  -> " << months["february"] << std::endl;
    return 0;
}
```

https://en.wikipedia.org/wiki/Unordered_associative_containers_(C++)

Using unordered_map with a user defined key

```
struct X{int i,j,k;};

struct hash_X{
  size_t operator()(const X &x) const{
    return hash<int>()(x.i) ^ hash<int>()(x.j) ^ hash<int>()(x.k);
  }
};

 unordered_map<X,int,hash_X> my_map;
```

https://en.wikipedia.org/wiki/Unordered_associative_containers_(C++)

^ XOR Bitwise exclusive OR

data structure	build (n items)	insert	find
vector	O(n)	O(1) amortized	O(n)
sorted vector	O(n log n)	O(n)	O(log n)
set or map	O(n log n)	O(log n)	O(log n)
list	O(n)	O(1)	O(n)
sorted list	O(n log n)	O(n)	O(n)
hash table	O(n) ave. O(n^2) worst	O(1) ave O(n) worst	O(1) ave O(n) worst