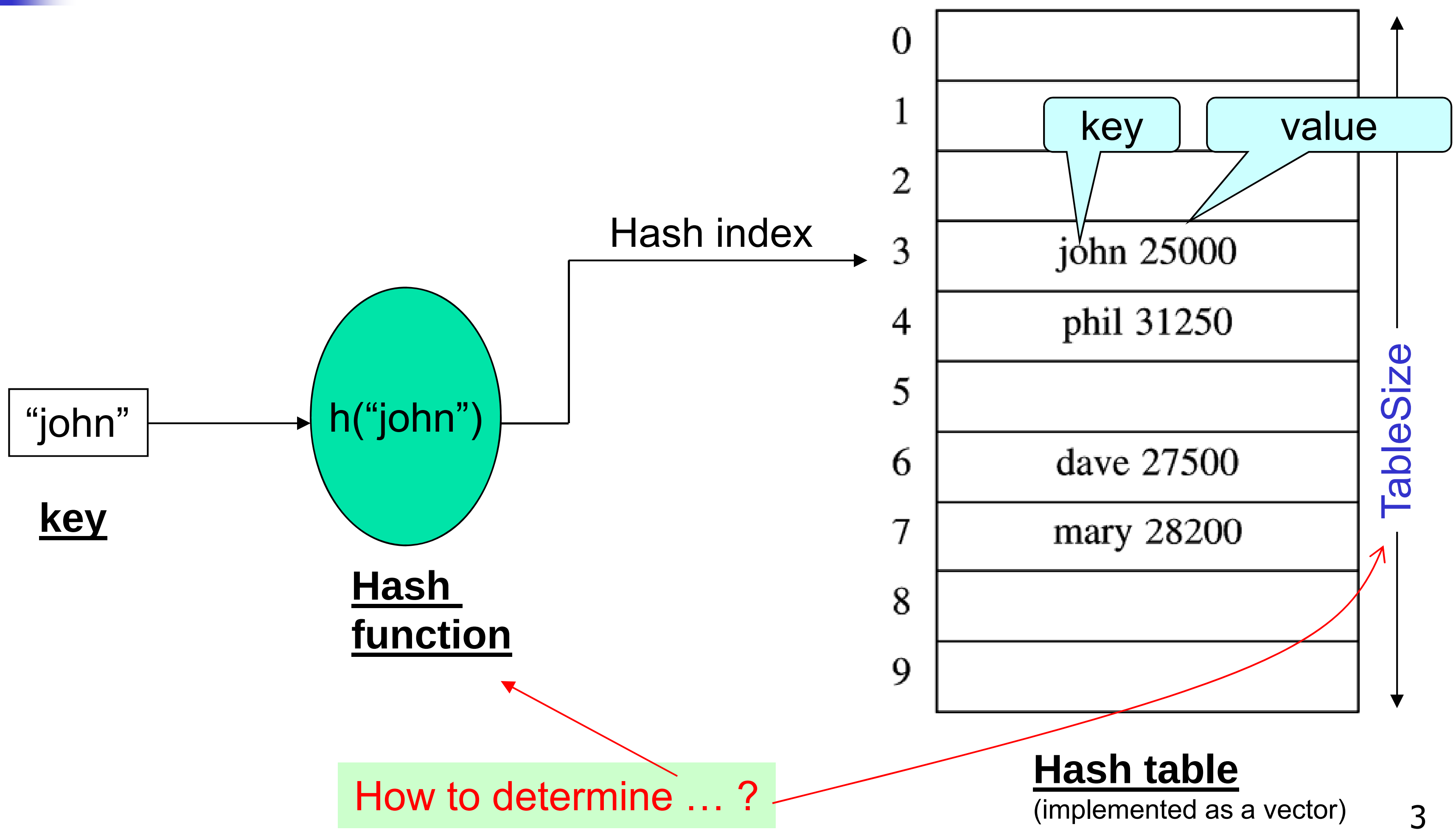
Hashing & Hash Tables

Overview

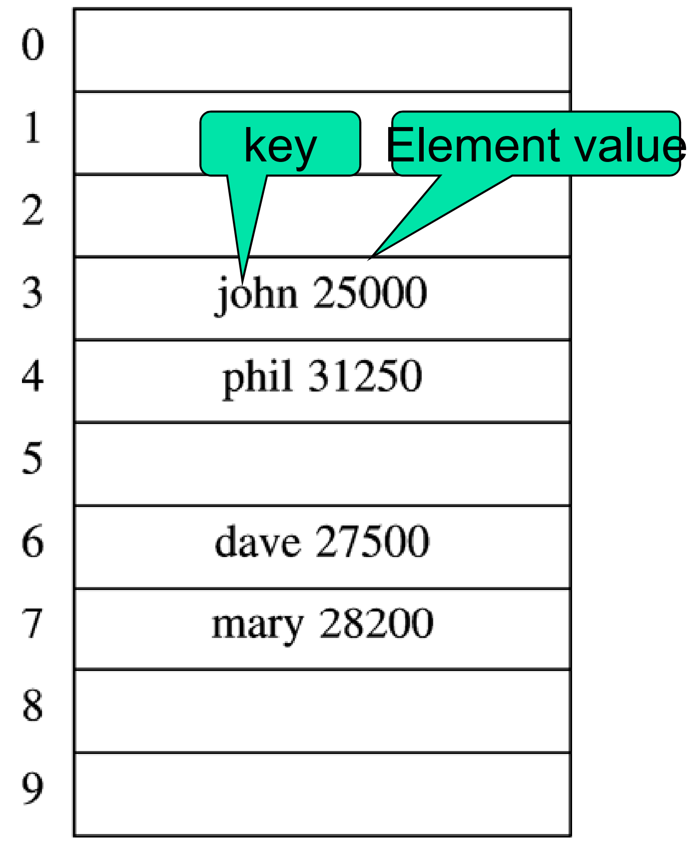
- Hash Table Data Structure : Purpose
  - *To support insertion, deletion and search in average-case constant time*
    - Assumption: Order of elements irrelevant
    - ==> data structure *not* useful for if you want to maintain and retrieve some kind of an order of the elements
- Hash function
  - *Hash["string key"] ==> integer value*
- Hash table ADT
  - Implementations, Analysis, Applications

Hash table: Main components



Hash Table

- Hash table is an array of fixed size TableSize
- Array elements indexed by a key, which is mapped to an array index (0…TableSize-1)
- Mapping (hash function) h from key to index
  - E.g., h("john") = 3



Hash Table Operations

Hash function

Hash key

- <u>Insert</u>
  - T [h("john")] = <"john",25000>

Data record

- <u>Delete</u>
  - T [h("john")] = NULL
- <u>Search</u>
  - T [h("john")] returns the element hashed for "john"

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

What happens if h("john") == h("joe") ?

"***<u>collision</u>***"

Factors affecting Hash Table Design

- Hash function

- Table size
  - Usually fixed at the start

- Collision handling scheme

Hash Function

- A hash function is one which maps an element's key into a valid hash table index
  - h(key) => hash table index

Note that this is (slightly) different from saying:
h(string) => int

- Because the key can be of any type
  - E.g., "h(int) => int" is also a hash function!
- But also note that any type can be converted into an equivalent string form

h(key) ==> hash table index

Hash Function Properties

- A hash function maps key to integer
  - Constraint: Integer should be between [0, TableSize-1]
- A hash function can result in a many-to-one mapping (causing collision)
  - Collision occurs when hash function maps two or more keys to same array index
- Collisions cannot be avoided but its chances can be reduced using a "good" hash function

h(key) ==> hash table index

Hash Function Properties

- A "good" hash function should have the properties:
  1. Reduced chance of collision
     - Different keys should ideally map to different indices
     - Distribute keys uniformly over table
  2. Should be fast to compute

Hash Function - Effective use of table size

- Simple hash function (assume integer keys)
  - h(Key) = Key *mod* TableSize

- For random keys, h() distributes keys evenly over table
  - What if TableSize = 100 and keys are ALL multiples of 10?
  - Better if TableSize is a prime number

Different Ways to Design a Hash Function for String Keys

A very simple function to map strings to integers:

- Add up character ASCII values (0-255) to produce integer keys
  - E.g., "abcd" = 97+98+99+100 = 394
  - ==> h("abcd") = 394 % TableSize

Potential problems:

- Anagrams will map to the same index
  - h("abcd") == h("dbac")
- Small strings may not use all of table
  - Strlen(S) * 255 < TableSize
- Time proportional to length of the string

Different Ways to Design a Hash Function for String Keys

- Approach 2
  - Treat first 3 characters of string as base-27 integer (26 letters plus space)
    - Key = S[0] + (27 * S[1]) + (27^2 * S[2])
  - Better than approach 1 because … ?

Potential problems:

- Assumes first 3 characters randomly distributed
  - Not true of English

Apple
Apply
Appointment
Apricot

collision

Different Ways to Design a Hash Function for String Keys

- <u>Approach 3</u>

  Use all N characters of string as an N-digit base-K number

  - Choose K to be prime number larger than number of different digits (characters)
    - I.e., K = 29, 31, 37
  - If L = length of string S, then

$$h(S) = \left[\sum_{i=0}^{L-1} S[L-i-1] * 37^i \right] \mod TableSize$$

  - Use Horner's rule to compute h(S)
  - Limit L for long strings

```
/**
 * A hash routine for string objects.
 */
int hash( const string & key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key[ i ];

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}
```

<u>Problems:</u>

potential overflow

larger runtime

“Collision resolution techniques”

Techniques to Deal with Collisions

Chaining

Open addressing

Double hashing

Etc.

Resolving Collisions

- What happens when $h(k_1) = h(k_2)$?
  - ==> collision !
- Collision resolution strategies
  - ***Chaining***
    - Store colliding keys in a linked list at the same hash table index
  - ***Open addressing***
    - Store colliding keys elsewhere in the table

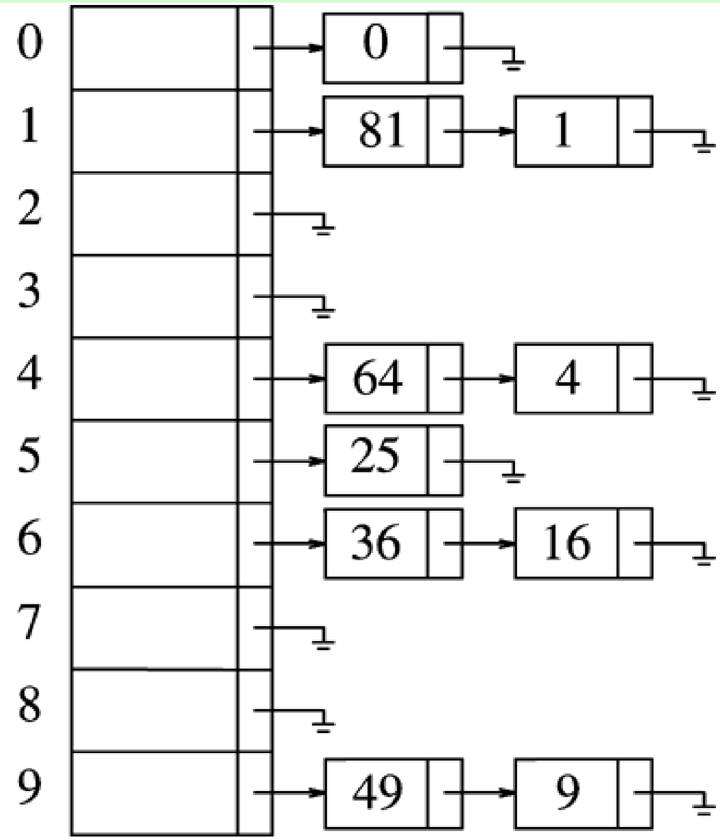
Chaining

Collision resolution technique #1

Chaining strategy: maintains a linked list at every hash index for collided elements

Insertion sequence: { 0 1 4 9 16 25 36 49 64 81 }

- Hash table T is a vector of linked lists
  - Insert element at the head (as shown here) or at the tail
- Key k is stored in list at T[h(k)]
- E.g., TableSize = 10
  - h(k) = k mod 10
  - Insert first 10 perfect squares



Implementation of Chaining Hash Table

```
template <typename HashedObj>
class HashTable
{
  public:
    explicit HashTable( int size = 101 );

    bool contains( const HashedObj & x ) const;

    void makeEmpty( );
    void insert( const HashedObj & x );
    void remove( const HashedObj & x );

  private:
    vector<list<HashedObj> > theLists;   // The array of Lists
    int  currentSize;

    void rehash( );
    int myhash( const HashedObj & x ) const;
};

int hash( const string & key );
int hash( int key );
```

Vector of linked lists (this is the main hashtable)

Current #elements in the hashtable

Hash functions for integers and string keys

Implementation of Chaining Hash Table

```
int myhash( const HashedObj & x ) const
{
    int hashVal = hash( x );

    hashVal %= theLists.size( );
    if( hashVal < 0 )
        hashVal += theLists.size( );

    return hashVal;
}
```

This is the hashtable's current capacity (aka. "table size")

This is the hash table index for the element x

```
bool insert( const HashedObj & x )
{
    list<HashedObj> & whichList = theLists[ myhash( x ) ];
    if( find( whichList.begin( ), whichList.end( ), x ) != whichList.end( ) )
        return false;
    whichList.push_back( x );

        // Rehash; see Section 5.5
    if( ++currentSize > theLists.size( ) )
        rehash( );

    return true;
}
```

Duplicate check

Later, but essentially resizes the hashtable if its getting crowded

```cpp
void makeEmpty( )
{
    for( int i = 0; i < theLists.size( ); i++ )
        theLists[ i ].clear( );
}

bool contains( const HashedObj & x ) const
{
    const list<HashedObj> & whichList = theLists[ myhash( x ) ];
    return find( whichList.begin( ), whichList.end( ), x ) != whichList.end( );
}

bool remove( const HashedObj & x )
{
    list<HashedObj> & whichList = theLists[ myhash( x ) ];
    list<HashedObj>::iterator itr = find( whichList.begin( ), whichList.end( ), x );

    if( itr == whichList.end( ) )
        return false;

    whichList.erase( itr );
    --currentSize;
    return true;
}
```

Each of these operations takes time linear in the length of the list at the hashed index location

```
// Example of an Employee class
class Employee
{
  public:
    const string & getName( ) const
      { return name; }

    bool operator==( const Employee & rhs ) const
      { return getName( ) == rhs.getName( ); }
    bool operator!=( const Employee & rhs ) const
      { return !( *this == rhs; }

      // Additional public members not shown

  private:
    string name;
    double salary;
    int    seniority;

      // Additional private members not shown
};

int hash( const Employee & item )
{
    return hash( item.getName( ) );
}
```

All hash objects must define == and != operators.

Hash function to handle Employee object type

Collision Resolution by Chaining: Analysis

- Load factor λ of a hash table T is defined as follows:
  - N = number of elements in T (“current size”)
  - M = size of T (“table size”)
  - λ = N/M (“ load factor”)
    - i.e., λ is the average length of a chain
- Unsuccessful search time: O(λ)
  - Same for insert time
- Successful search time: O(λ/2)
- Ideally, want $\lambda \leq 1$ (not a function of N)

Potential disadvantages of Chaining

Linked lists could get long

- Especially when N approaches M
- Longer linked lists could negatively impact performance

More memory because of pointers

Absolute worst-case (even if N << M):

- All N elements in one linked list!
- Typically the result of a bad hash function

Open Addressing

Collision resolution technique #2

An "inplace" approach

Collision Resolution by Open Addressing

When a collision occurs, look elsewhere in the table for an empty slot

- Advantages over chaining
  - No need for list structures
  - No need to allocate/deallocate memory during insertion/deletion (slow)
- Disadvantages
  - Slower insertion – May need several attempts to find an empty slot
  - Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance
    - Load factor $\lambda \approx 0.5$

Collision Resolution by Open Addressing

- A <u>"Probe sequence"</u> is a sequence of slots in hash table while searching for an element x
  - $h_0(x)$, $h_1(x)$, $h_2(x)$, ...
  - Needs to visit each slot exactly once
  - Needs to be repeatable (so we can find/delete what we've inserted)

- <u>Hash function</u>
  - ***$h_i(x)$ = (h(x) + f(i)) mod TableSize***
  - f(0) = 0 ==> position for the 0^{th} probe
  - f(i) is "*the distance to be traveled relative to the 0^{th} probe position, during the i^{th} probe*".

Linear Probing

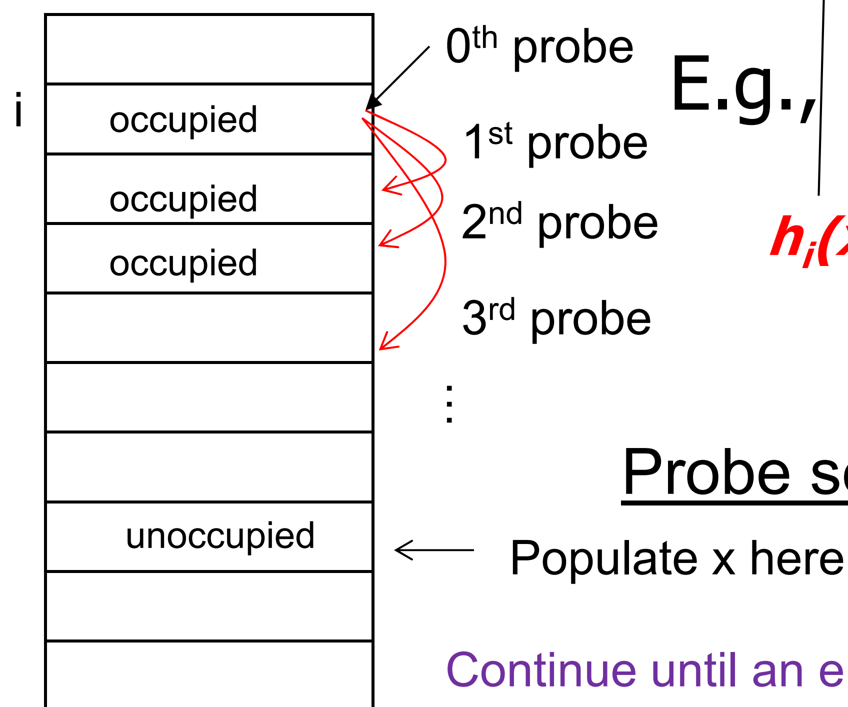
i^{th} probe index = 0^{th} probe index + i

- f(i) = is a linear function of i,

E.g., $f(i) = i$

$$h_i(x) = (h(x) + i) \bmod TableSize$$

Linear probing:



Probe sequence: +0, +1, +2, +3, +4, …

Continue until an empty slot is found

#failed probes is a measure of performance

i^{th} probe index =

0^{th} probe index

+ i

Linear Probing

- f(i) = is a linear function of i, e.g., *f(i) = i*
  - ***$h_i(x)$ = (h(x) + i) mod TableSize***
  - Probe sequence: +0, +1, +2, +3, +4, …
- Example: h(x) = x mod TableSize
  - $h_0(89)$ = (h(89)+f(0)) mod 10 = 9
  - $h_0(18)$ = (h(18)+f(0)) mod 10 = 8
  - $h_0(49)$ = (h(49)+f(0)) mod 10 = 9 (X)
  - $h_1(49)$ = (h(49)+f(1)) mod 10

    = (h(49)+ 1) mod 10 = 0

Linear Probing Example

Insert sequence: 89, 18, 49, 58, 69

time →

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

#unsuccessful probes:	0	0	1	3	3	7 total

Linear Probing: Issues

Probe sequences can get longer with time

Primary clustering

- Keys tend to cluster in one part of table
- Keys that hash into cluster will be added to the end of the cluster (making it even bigger)
- Side effect: Other keys could also get affected if mapping to a crowded neighborhood

Linear Probing: Analysis

- Expected number of probes for insertion or unsuccessful search

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- Expected number of probes for successful search

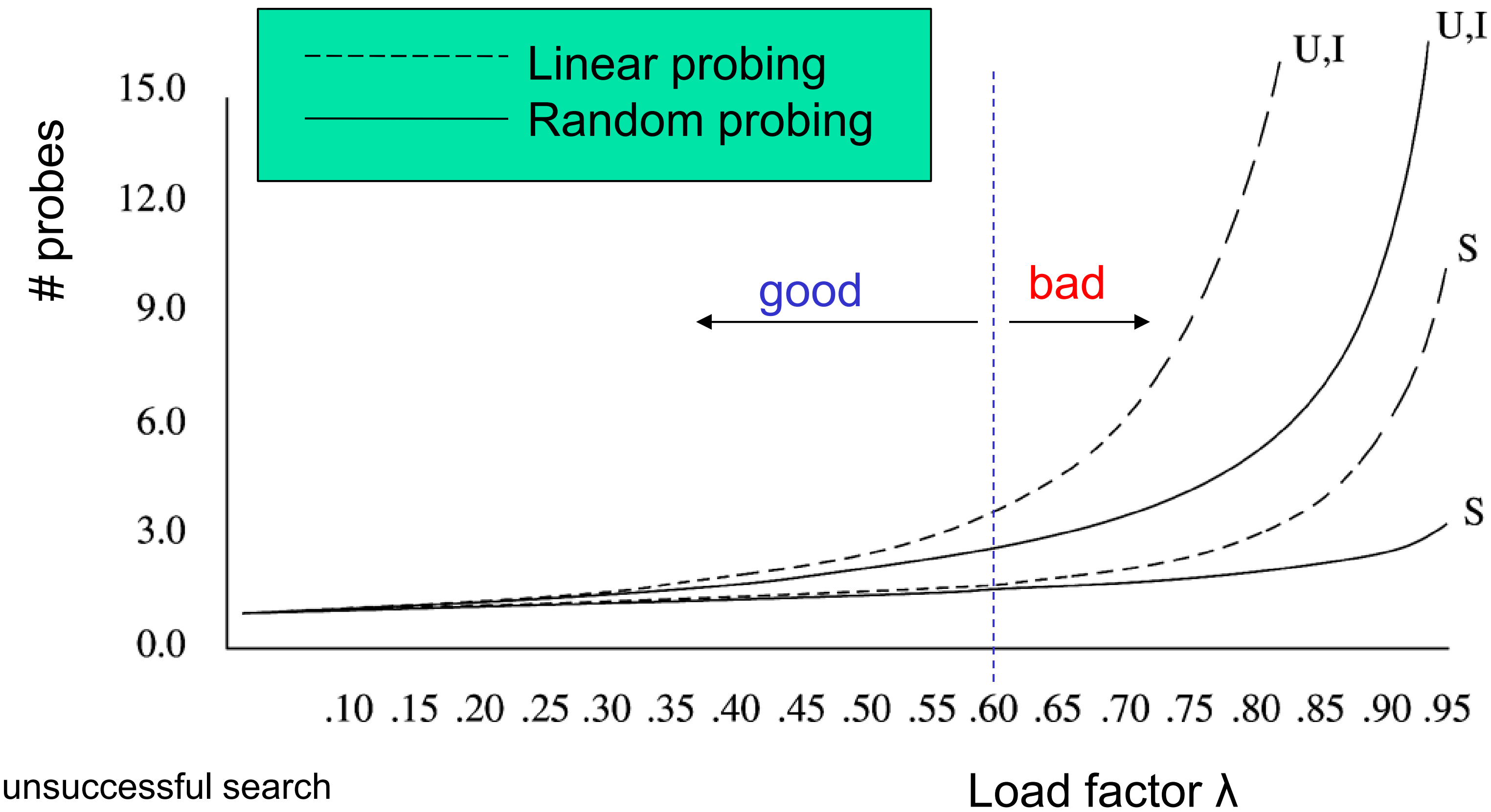
$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

- Example (λ = 0.5)
  - Insert / unsuccessful search
    - 2.5 probes
  - Successful search
    - 1.5 probes
- Example (λ = 0.9)
  - Insert / unsuccessful search
    - 50.5 probes
  - Successful search
    - 5.5 probes

Random Probing: Analysis

- Random probing does not suffer from clustering
- Expected number of probes for insertion or unsuccessful search: $$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$
- Example
  - $\lambda = 0.5$: 1.4 probes
  - $\lambda = 0.9$: 2.6 probes

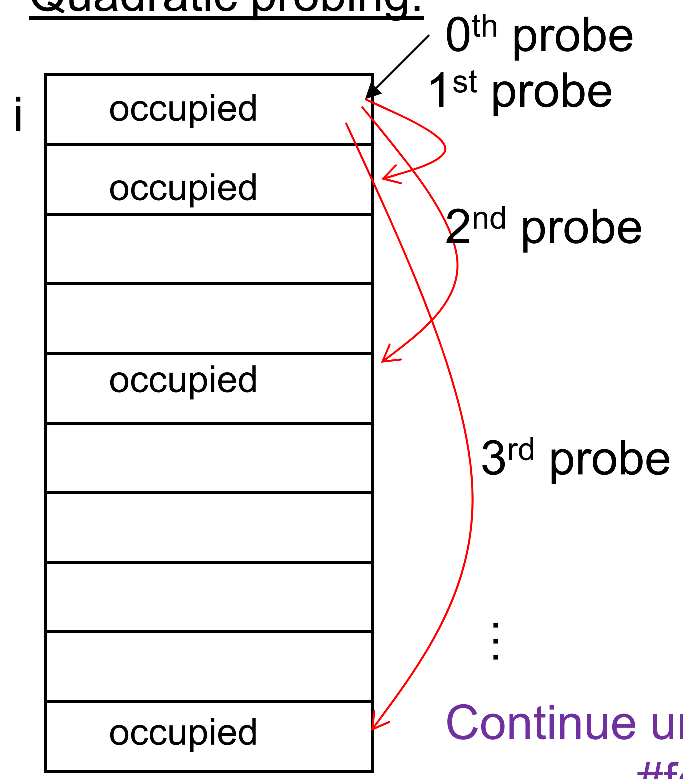
Linear vs. Random Probing



U - unsuccessful search
S - successful search
I - insert

Quadratic Probing

Quadratic probing:



- Avoids primary clustering
- f(i) is quadratic in i

  e.g., $f(i) = i^2$

  $h_i(x) = (h(x) + i^2) \bmod TableSize$

  - <u>Probe sequence:</u>

    +0, +1, +4, +9, +16, …

Continue until an empty slot is found

#failed probes is a measure of performance

Quadratic Probing

- Avoids primary clustering
- f(i) is quadratic in I, e.g., *$f(i) = i^2$*
  - ***$h_i(x) = (h(x) + i^2)$ mod TableSize***
  - Probe sequence: +0, +1, +4, +9, +16, …
- Example:
  - $h_0(58) = (h(58)+f(0)) \bmod 10 = 8$ (X)
  - $h_1(58) = (h(58)+f(1)) \bmod 10 = 9$ (X)
  - $h_2(58) = (h(58)+f(2)) \bmod 10 = 2$

Q) Delete(49), Find(69) - is there a problem?

Quadratic Probing Example

Insert sequence: 89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49 $+1^2$	49	49 $+1^2$
1						
2					58 $+2^2$	58
3						69 $+2^2$
4						
5						
6						
7						
8			18 $+0^2$	18	18 $+0^2$	18
9		89 $+0^2$	89	89 $+0^2$	89 $+1^2$	89 $+0^2$

#unsuccessful probes:	0	0	1	2	2	5 total

Quadratic Probing: Analysis

- Difficult to analyze
- Theorem 5.1
  - New element can always be inserted into a table that is at least half empty and TableSize is prime
- Otherwise, may never find an empty slot, even is one exists
- Ensure table never gets half full
  - If close, then expand it

Quadratic Probing

- May cause "secondary clustering"

- Deletion
  - Emptying slots can break probe sequence and could cause find stop prematurely
  - Lazy deletion
    - Differentiate between empty and deleted slot
    - When finding skip and continue beyond deleted slots
      - If you hit a non-deleted empty slot, then stop find procedure returning "not found"
  - May need compaction at some time

Quadratic Probing: Implementation

```
template <typename HashedObj>
class HashTable
{
  public:
    explicit HashTable( int size = 101 );

    bool contains( const HashedObj & x ) const;

    void makeEmpty( );
    bool insert( const HashedObj & x );
    bool remove( const HashedObj & x );

```

Quadratic Probing: Implementation

```
    enum EntryType { ACTIVE, EMPTY, DELETED };

  private:
    struct HashEntry
    {
        HashedObj element;
        EntryType info;

        HashEntry( const HashedObj & e = HashedObj( ), EntryType i = EMPTY )
          : element( e ), info( i ) { }
    };

    vector<HashEntry> array;
    int currentSize;

    bool isActive( int currentPos ) const;
    int findPos( const HashedObj & x ) const;
    void rehash( );
    int myhash( const HashedObj & x ) const;
};
```

Lazy deletion

Quadratic Probing: Implementation

```
explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
  { makeEmpty( ); }

void makeEmpty( )
{
    currentSize = 0;
    for( int i = 0; i < array.size( ); i++ )
        array[ i ].info = EMPTY;
}
```

Ensure table size is prime

Quadratic Probing: Implementation

```
bool contains( const HashedObj & x ) const
  { return isActive( findPos( x ) ); }

int findPos( const HashedObj & x ) const
{
    int offset = 1;
    int currentPos = myhash( x );

    while( array[ currentPos ].info != EMPTY &&
            array[ currentPos ].element != x )
    {
        currentPos += offset;  // Compute ith probe
        offset += 2;
        if( currentPos >= array.size( ) )
            currentPos -= array.size( );
    }

    return currentPos;
}

bool isActive( int currentPos ) const
  { return array[ currentPos ].info == ACTIVE; }
```

Find

Skip DELETED;
No duplicates

Quadratic probe
sequence (really)

Quadratic Probing: Implementation

```
bool insert( const HashedObj & x )
{
        // Insert x as active
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return false;

    array[ currentPos ] = HashEntry( x, ACTIVE );

        // Rehash; see Section 5.5
    if( ++currentSize > array.size( ) / 2 )
        rehash( );

    return true;
}

bool remove( const HashedObj & x )
{
    int currentPos = findPos( x );
    if( !isActive( currentPos ) )
        return false;

    array[ currentPos ].info = DELETED;
    return true;
}
```

Insert

No duplicates

Remove

No deallocation needed

Double Hashing: keep two hash functions h_1 and h_2

- Use a second hash function for all tries I other than 0: ***$f(i) = i * h_2(x)$***
- Good choices for $h_2(x)$?
  - Should never evaluate to 0
  - $h_2(x) = R – (x \bmod R)$
    - R is prime number less than TableSize
- Previous example with R=7
  - $h_0(49) = (h(49)+f(0)) \bmod 10 = 9$ (X)
  - $h_1(49) = (h(49)+\underbrace{1*(7 – 49 \bmod 7)}_{f(1)}) \bmod 10 = 6$

Double Hashing Example

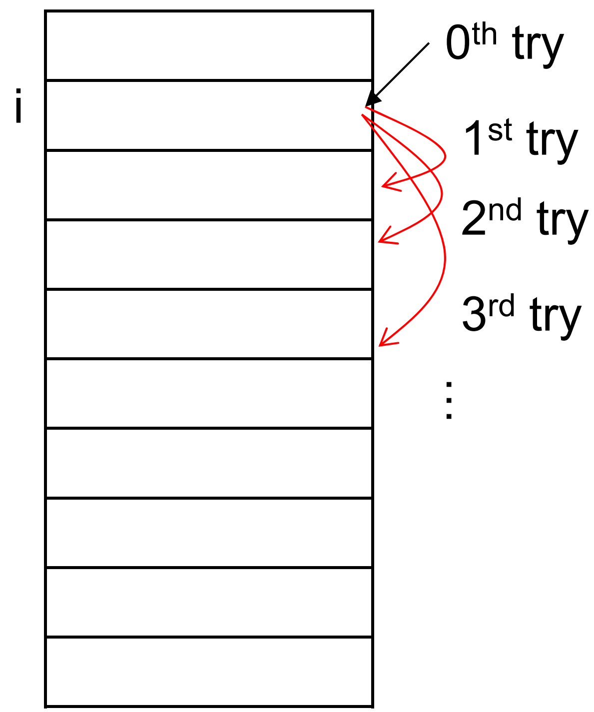
	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing: Analysis

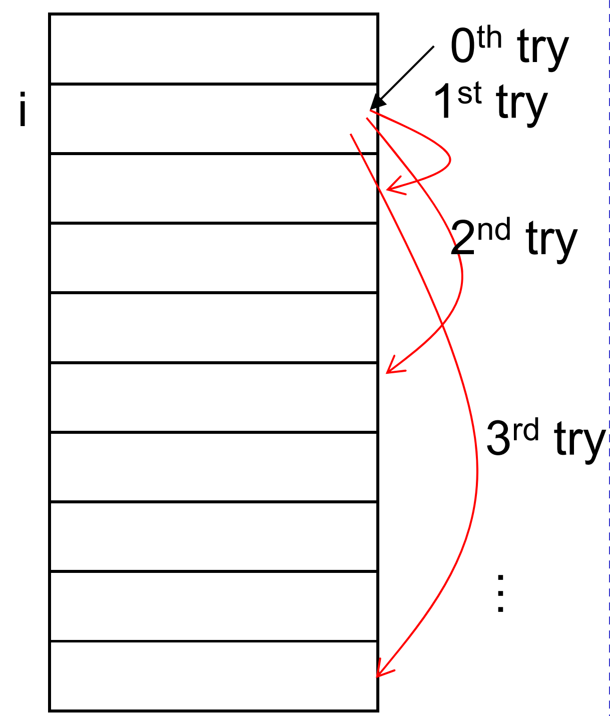
- Imperative that TableSize is prime
  - E.g., insert 23 into previous table
- Empirical tests show double hashing close to random hashing
- Extra hash function takes extra time to compute

Probing Techniques - review

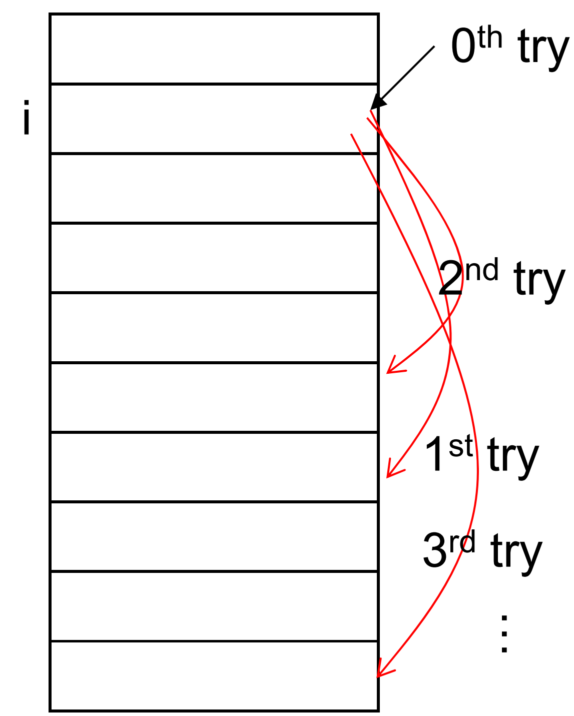
Linear probing:



Quadratic probing:



Double hashing*:



*(determined by a second hash function)

Rehashing

- Increases the size of the hash table when load factor becomes “too high” (defined by a cutoff)
  - Anticipating that prob(collisions) would become higher
- Typically expand the table to twice its size (but still prime)
- Need to reinsert all existing elements into new hash table

Rehashing Example

$h(x) = x \bmod 7$

$\lambda = 0.57$

0	6
1	15
2	
3	24
4	
5	
6	13

Insert 23

$\lambda = 0.71$

0	6
1	15
2	23
3	24
4	
5	
6	13

Rehashing

$h(x) = x \bmod 17$

$\lambda = 0.29$

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing Analysis

- Rehashing takes time to do N insertions
- Therefore should do it infrequently
- Specifically
  - Must have been N/2 insertions since last rehash
  - Amortizing the O(N) cost over the N/2 prior insertions yields only constant additional time per insertion

Rehashing Implementation

- When to rehash
  - When load factor reaches some threshold (e.g,. $\lambda \geq 0.5$), OR
  - When an insertion fails

- Applies across collision handling schemes

Rehashing for Chaining

```
/**
 * Rehashing for separate chaining hash table.
 */
void rehash( )
{
    vector<list<HashedObj> > oldLists = theLists;

        // Create new double-sized, empty table
    theLists.resize( nextPrime( 2 * theLists.size( ) ) );
    for( int j = 0; j < theLists.size( ); j++ )
        theLists[ j ].clear( );

        // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldLists.size( ); i++ )
    {
        list<HashedObj>::iterator itr = oldLists[ i ].begin( );
        while( itr != oldLists[ i ].end( ) )
            insert( *itr++ );
    }
}
```

Rehashing for Quadratic Probing

```
/**
 * Rehashing for quadratic probing hash table.
 */
void rehash( )
{
    vector<HashEntry> oldArray = array;

        // Create new double-sized, empty table
    array.resize( nextPrime( 2 * oldArray.size( ) ) );
    for( int j = 0; j < array.size( ); j++ )
        array[ j ].info = EMPTY;

        // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldArray.size( ); i++ )
        if( oldArray[ i ].info == ACTIVE )
            insert( oldArray[ i ].element );
}
```

Hash Tables in C++ STL

- Hash tables not part of the C++ Standard Library
- Some implementations of STL have hash tables (e.g., SGI's STL)
  - **`hash_set`**
  - **`hash_map`**

Hash Set in STL

```
#include <hash_set>

struct eqstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

void lookup(const hash_set<const char*, hash<const char*>, eqstr>& Set,
            const char* word)
{
  hash_set<const char*, hash<const char*>, eqstr>::const_iterator it
    = Set.find(word);
  cout << word << ": "
       << (it != Set.end() ? "present" : "not present")
       << endl;
}

int main()
{
  hash_set<const char*, hash<const char*>, eqstr> Set;
  Set.insert("kiwi");
  lookup(Set, “kiwi");
}
```

Key

Hash fn

Key equality test

Hash Map in STL

```
#include <hash_map>

struct eqstr
{
  bool operator() (const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

int main()                 Key       Data      Hash fn          Key equality test
{
  hash_map<const char*, int, hash<const char*>, eqstr> months;
  months["january"] = 31;
  months["february"] = 28;
  …
  months["december"] = 31;
  cout << “january -> " << months[“january"] << endl;
}
```

Internally treated like insert (or overwrite if key already present)

Problem with Large Tables

- What if hash table is too large to store in main memory?
- Solution: Store hash table on disk
  - Minimize disk accesses
- But…
  - Collisions require disk accesses
  - Rehashing requires a lot of disk accesses

Solution: **Extendible Hashing**

Hash Table Applications

- Symbol table in compilers
- Accessing tree or graph nodes by name
  - E.g., city names in Google maps
- Maintaining a transposition table in games
  - Remember previous game situations and the move taken (avoid re-computation)
- Dictionary lookups
  - Spelling checkers
  - Natural language understanding (word sense)
- Heavily used in text processing languages
  - E.g., Perl, Python, etc.

Summary

- Hash tables support fast insert and search
  - O(1) average case performance
  - Deletion possible, but degrades performance
- Not suited if ordering of elements is important
- Many applications

Points to remember - Hash tables

- Table size prime
- Table size much larger than number of inputs (to maintain λ closer to 0 or < 0.5)
- Tradeoffs between chaining vs. probing
- Collision chances decrease in this order: linear probing => quadratic probing => {random probing, double hashing}
- Rehashing required to resize hash table at a time when λ exceeds 0.5
- Good for searching. Not good if there is some order implied by data.