Data Structures:
Hashing & Hash Tables

1. Hashing & Hash Table
2. Collision
3. Rehashing
4. Coding
   - Using list in STL
   - Using unordered_map in STL

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

자기 아들을 아끼지 아니하시고 우리 모든 사람을 위하여 내주신 이가 어찌 그 아들과 함께 모든 것을 우리에게 주시지 아니하겠느냐 (로마서 8:32)

우리가 알거니와 하나님을 사랑하는 자 곧 그의 뜻대로 부르심을 입은 자들에게는 모든 것이 합력하여 선을 이루느니라 (로마서 8:28)

Overview

- Hash Table Data Structure: Purpose

- Implementations So Far

Array of size n	unsorted list	sorted array	Trees BST – average AVL – worst	Heap, Priority Queue	Hash Table
`insert`	`find+O(1)`	`O(n)`	`O(log n)`	`O(log n)`	
`find`	`O(n)`	`O(log n)`	`O(log n)`	`O(log n)`	
`remove`	`find+O(1)`	`O(n)`	`O(log n)`	`O(log n)`	

Overview

- Hash Table Data Structure: Purpose
  support insertion, deletion and search in average case constant time O(1)

- Implementations So Far

Array of size n	unsorted list	sorted array	Trees BST – average AVL – worst	Heap, Priority Queue	Hash Table
insert	find+O(1)	O(n)	O(log n)	O(log n)	*O(1)*
find	O(n)	O(log n)	O(log n)	O(log n)	*O(1)*
remove	find+O(1)	O(n)	O(log n)	O(log n)	*O(1)*

Overview

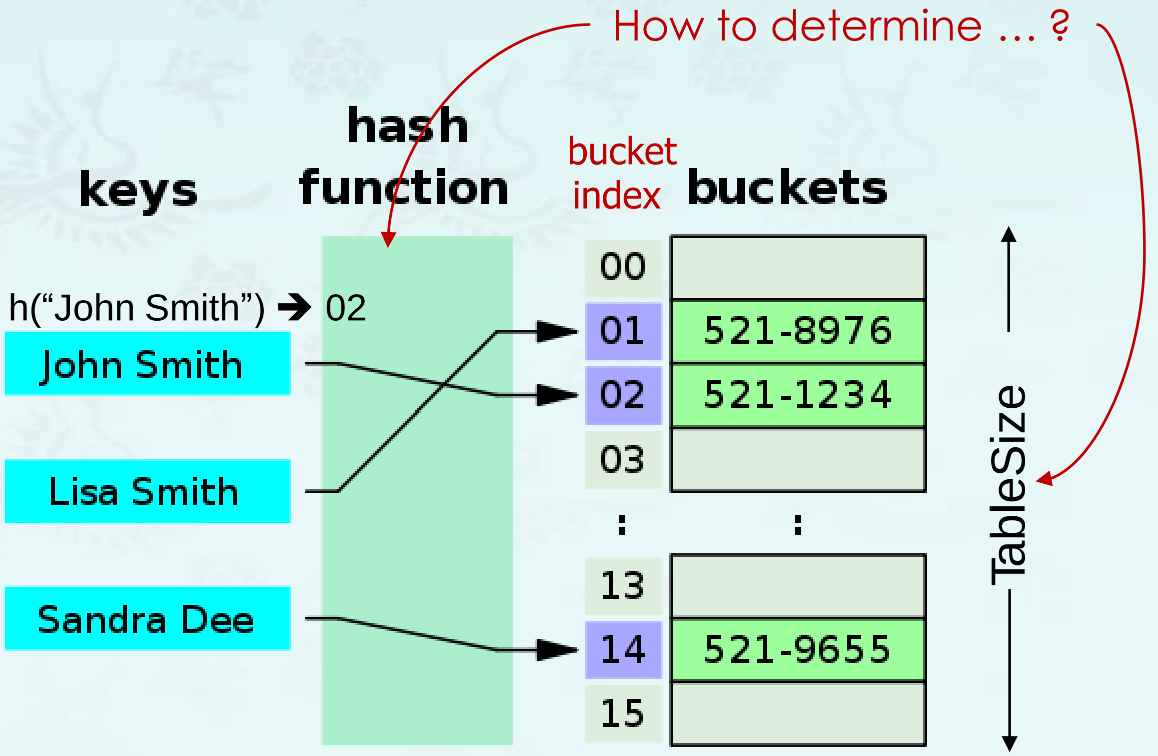
- Hash Table Data Structure: Purpose
  support insertion, deletion and search in average case constant time O(1)

- Hash function
  - Hash[int key]➔ integer value
  - Hash["string key"] ➔ integer value

- Hash table ADT
  - Implementations, Analysis, Applications

Hash Table Main components

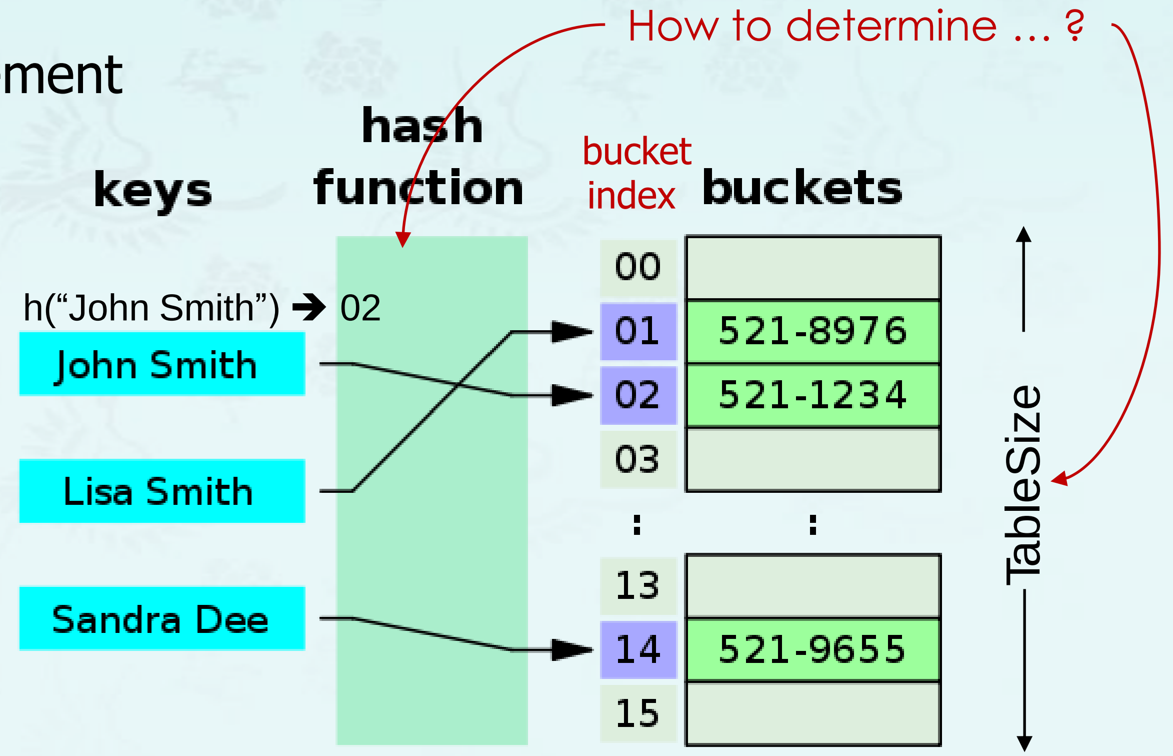
- Hash table is an array of fixed size elements
- Array elements indexed by a key mapped to an bucket index (0 ... TableSize-1)
- Mapping (hash function) h from key to index
  - e.g., h("John Smith") ➔ 02



Hash Table Operations

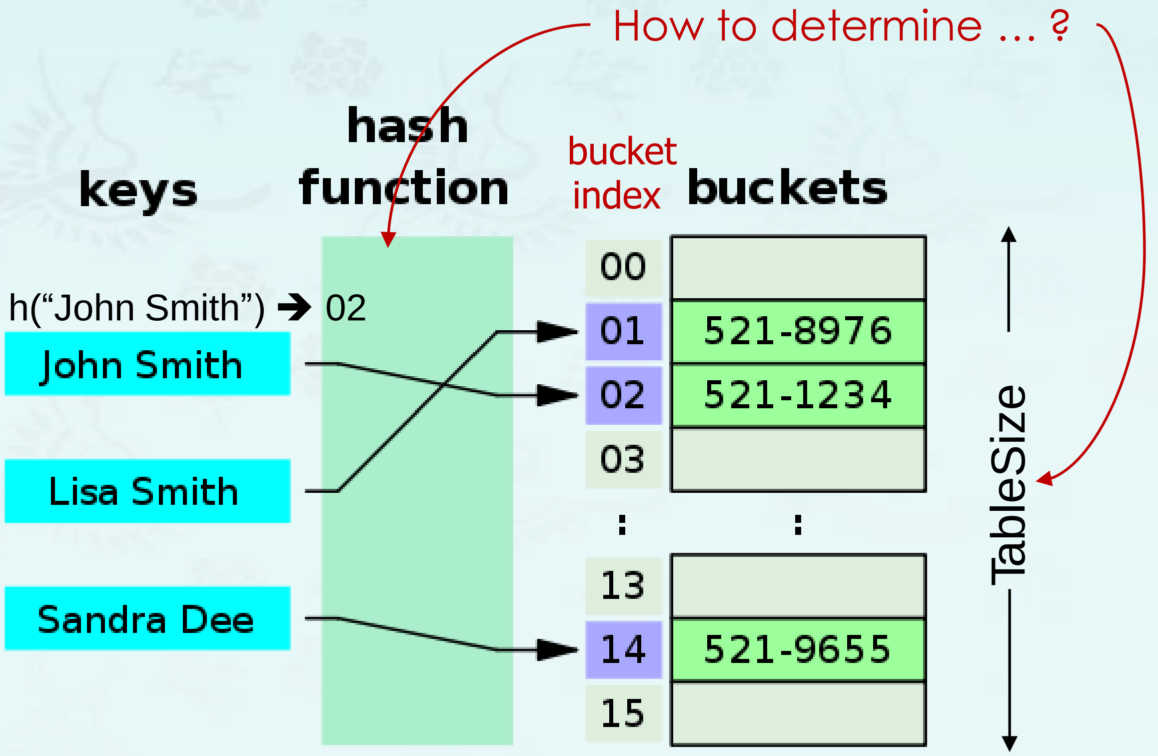
- insert
  - HashTable[h("John Smith")] = <"John Smith", 521-1234>
- remove
  - HashTable[h("John Smith")] = NULL
- find
  - HashTable[h("John Smith")] returns the element hashed for "John Smith"

- What happens if h("John Smith") == h("Joe Blow")
  - "Collision"



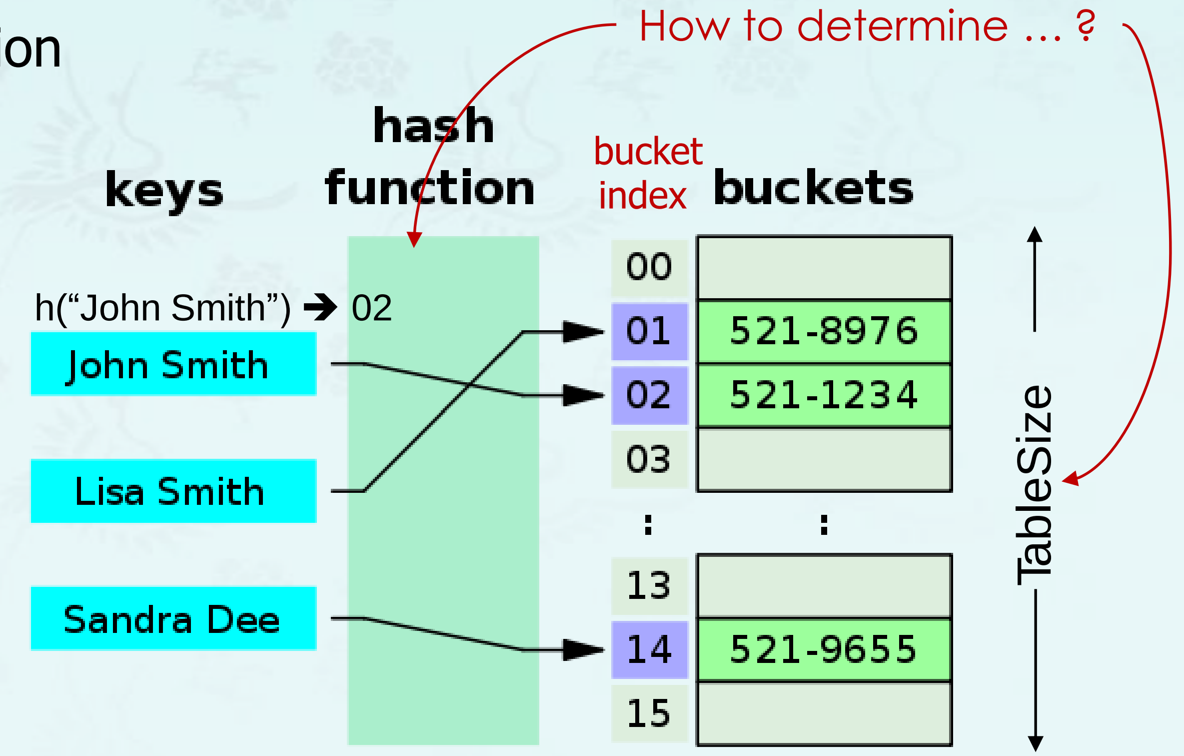
Factors affecting Hash Table Design

- Hash function
- Table size or
  - Usually fixed at the start
- Collision handling schemes



Hash Function

- It 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



Hash Function Properties

- It maps an element's key into a valid hash table index
  - h(key) → hash table index
- It 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

Hash Function - Effective use of table size

- Simple hash function (assume integer keys)
  - h(Key) = Key % 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

- Another approach:
  - Treat first 3 characters of string as base-27 integer (26 letters plus space)
    - e.g., Key = s[0] + (27^1 * s[1]) + (27^2 * s[2])
    - Better than previous approach because ...

- Potential problems:
  - Assumes first 3 characters randomly distributed
    - This is not true in English

Apple
Apply
Appointment
Apricot

→ collision

Different Ways to Design a Hash Function for String Keys

- Last approach:
  - *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

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

    - Use Horner's rule to compute h(S).
    - Limit L for long strings
  - Potential problems
    - Overflow
    - Larger runtime

```
// a hash function for strings
int hash(const string& key, int tablesize) {
    int value = 0;
    for (auto x : key)
        value = value * 37 + x;
    value %= tablesize;
    if (value < 0) value += tablesize;
    return value;
}
```

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

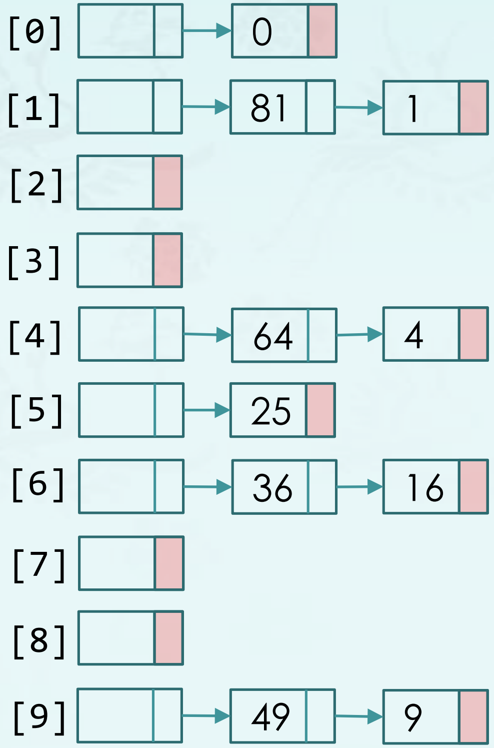
Chaining:

Collison resolution technique 1

Collision Resolution by Chaining: Analysis

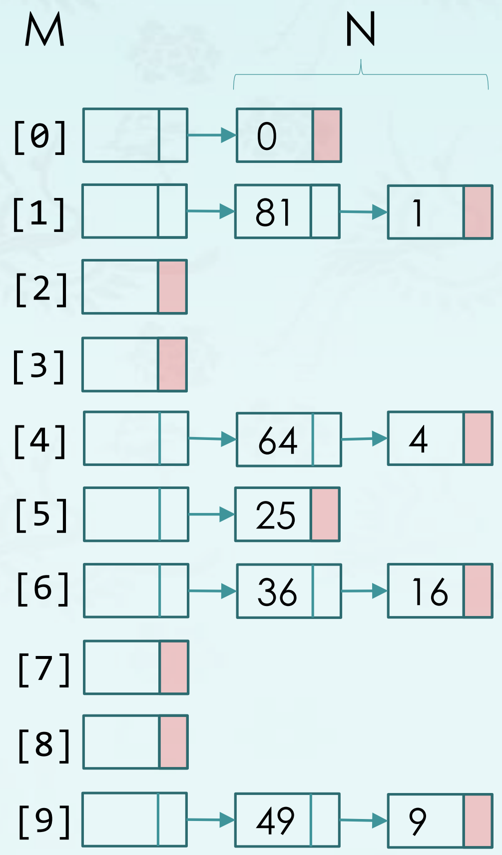
- Chaining strategy: maintains a linked list at every hash index for collided elements
  - Hash table T is a vector of linked lists
    - Insert element at the head (as shown here) or at the tail
  - Key k is stored in list a HashTable[h(k)]
  - e.g., TableSize = 10
    - h(k) = k % 10
    - insert first 10 perfect squares

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



Collision Resolution by Chaining: Analysis

- Load factor λ of a hash table T is defined as follows:
  - Element size: N = number of elements in T
  - Table size: M = size of T
  - Load factor: λ = N/M (적재율)
    - i.e., λ is the average length of a chain



- Unsuccessful search time: O(λ)
  - Same for insert time
- Successful search time average: O(λ/2)
- Ideally, want λ ≤ 1 (then, 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:
Collison resolution technique 2

- Linear Probing선형조사법
- Quadratic Probing이차조사법
- Double Hashing이중해싱법

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 "Probe sequence" 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)

- Hash function
  - $h_i(x) = (h(x) + f(i))$ % 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". It can be linear, quadratic etc.

Linear Probing선형조사법

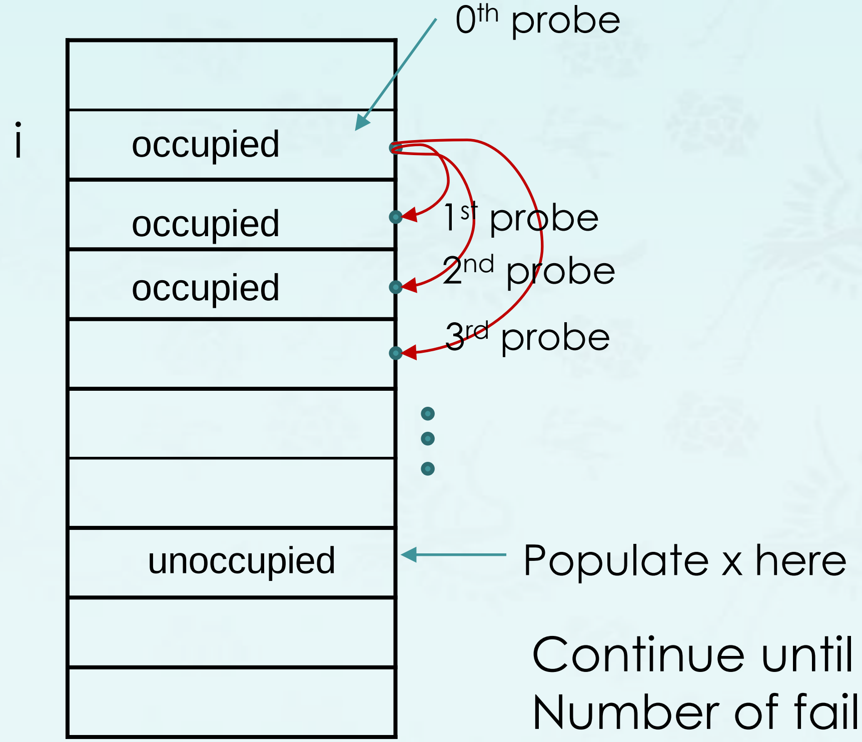
- f(i) = is a linear function of i, e.g., **f(i) = i**

$$h_i(x) = (\ h(x) + i\ )\ \%\ \text{TableSize}$$

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

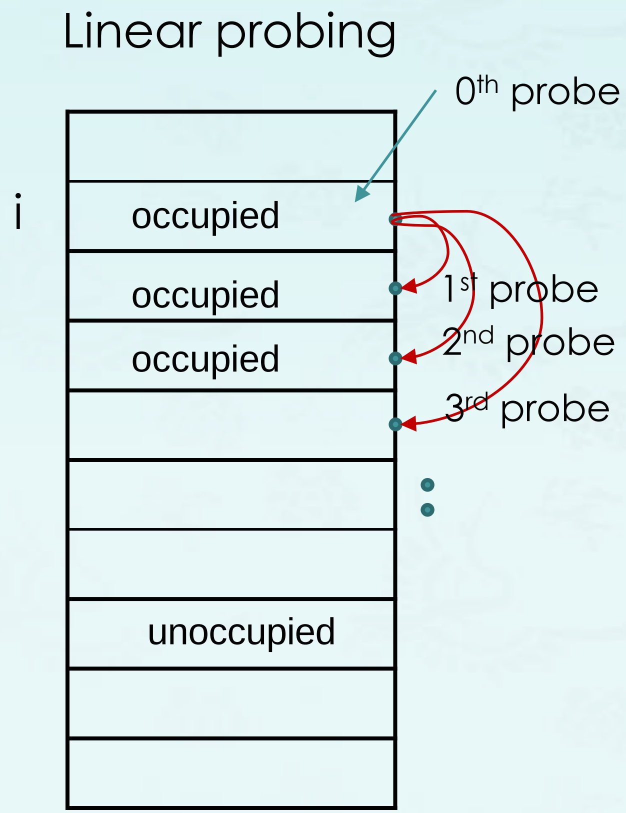
Probe sequence: +0, +1, +2, +3, +4, …

Linear probing



Continue until an empty slot is found
Number of failed probes is a measure of performance

Linear Probing Example



- f(i) = is a linear function of i, e.g., **f(i) = i**

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

  ith probe index　　0th probe index

  Probe sequence: +0, +1, +2, +3, +4, …

- Example: h(x) = x % TableSize
  - $h_0(89) = (h(89) + 0)\ \%\ 10 = 9$
  - $h_0(18) = (h(18) + 0)\ \%\ 10 = 8$
  - $h_0(49) = (h(49) + 0)\ \%\ 10 = 9$ (collision)

    $h_1(49) = (h(49) + 1)\ \%\ 10 = (h(49) + 1)\ \%\ 10 = 0$

Linear Probing Example

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

h(x) = k % 10

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

For example, linear probing for 58

$h_0(58) = (h(58)+f(0))$ % 10
$= (\quad 8 + 0)$ % 10 = 8 (collision)

$h_1(58) = (h(58)+ 1)$ % 10 = 0 (collision)

$h_2(58) = (h(58)+ 2)$ % 10 = 2

0	**0**	**1**	**3**	**3**	7
Unsucessful no. of probes					**Total**

Linear Probing Example

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

h(x) = k % 10

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

0	**0**	**1**	**3**	**3**	7
Unsucessful no. of probes					**Total**

For example, linear probing for 58

$h_0(58) = (h(58)+f(0))$ % 10
$= (\ \ 8 + 0)$ % 10 = 8 (collision)

$h_1(58) = (h(58)+\ 1)$ % 10 = 0 (collision)

$h_2(58) = (h(58)+\ 2)$ % 10 = 2

For example, linear probing for 69

$h_0(69) = (h(69)+f(0))$ % 10
=

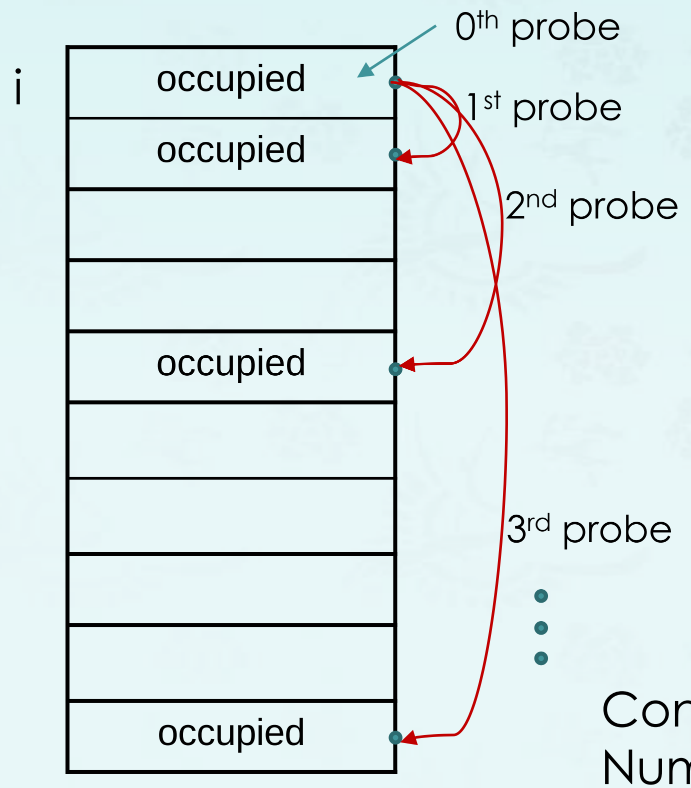
Linear Probing Issues

- 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

Quadratic Probing이차조사법

- Avoids primary clustering
- f(i) is quadratic in i, e.g., $f(i) = i^2$

Linear probing



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

ith probe index   0th probe index

Probe sequence: +0, +1, +4, +9, +16, ...

Continue until an empty slot is found
Number of failed probes is a measure of performance

Quadratic Probing이차조사법

- Avoids primary clustering
- f(i) is quadratic in i, e.g., $f(i) = i^2$

$$h_i(x) = (\ h(x) + i^2\ )\ \%\ \text{TableSize}$$

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

Probe sequence: +0, +1, +4, +9, +16, …

- Example:

$h_0(58) = (h(58) + 0^2)\ \%\ 10 = 8$ (collision)

$h_1(58) = (h(58) + 1^2)\ \%\ 10 = 9$ (collision)

$h_2(58) = (h(58) + 2^2)\ \%\ 10 = 2$

Quadratic Probing Example

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

h(x) = k % 10

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49 $+1^2$	49	49
1						
2					58 $+2^2$	58
3						
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	**0**	**1**	**2**	**2**	**5**
Unsucessful no. of probes					**Total**

For example, linear probing for 58

$h_0(58) = (h(58)+f(0))$ % 10

= (8 + 0) % 10 = 8 (collision)

$h_1(58) = (h(58)+ 1)$ % 10 = 9 (collision)

$h_2(58) = (h(58)+ 4)$ % 10 = 2

Quadratic Probing Example

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

h(x) = k % 10

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49 $+1^2$	49	49
1						
2					58 $+2^2$	58
3						
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
Unsucessful no. of probes		**0**	**0**	**1**	**2**	**2**

5

Total

For example, linear probing for 58

$h_0(58) = (h(58)+f(0))$ % 10

$= (\quad 8 + 0)$ % 10 = 8 (collision)

$h_1(58) = (h(58)+ 1)$ % 10 = 9 (collision)

$h_2(58) = (h(58)+ 4)$ % 10 = 2

For example, linear probing for 69

$h_0(69) =$

Quadratic Probing Analysis

- Difficult to analyze
- Theorem:
  - 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
- May cause "secondary clustering"

Double Hashing이중해싱법

- 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'(x)**

- Good choices for h(x)?
  - Should never evaluate to 0
  - h'(x) = R – (x % R)
    - R is prime number less than TableSize
- Previous example with R = 7

  $h_0(49) = (h(49) + f(0))$ % 10 = 9 (collision)

  $h_1(49) = (h(49) + 1 * \underbrace{(7 - 49 \% 7)}_{h'(x)})$ % 10 = 6

f(1)

i

Double Hashing Example

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

$h(x) = x \% 10$

	Empty Table	After 89	After 18	After 49	After 58	After 69	After 23
0							
1							
2							
3							
4							
5							
6				49	49	49	49
7							
8			18	18	18	18	18
9		89	89	89	89	89	89
		0	**0**	**1**	**2**	**2**	

Unsucessful no. of probes

$h_0(49) = (h(49)+f(0)) \% 10 = 9$ (collison)

$h_1(49) = (h(49)+1*(7 - 49 \% 7)) \% 10 = 6$

$h_0(58) =$

$h_1(58) =$

$h_0(69) =$

$h_1(69) =$

$h_0(23) =$

$h_1(23) =$

:

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

Rehashing

- Rehashing is the reconstruction of the hash table:
  - All the elements in the container are rearranged according to their hash value into the new set of buckets. This may alter the order of iteration of elements within the container.

- Increases the size of the hash table when load factor becomes "too high" (defined by a cutoff)
  - Anticipating that collisions would become higher
- Typically expand the table to twice its size (but still prime)
  - $TableSize_{new}$ = nextprime(2 * $TableSize_{old}$)
  - e.g., 2 → 5, 5 → 11, 11 → 23
- Need to reinsert all existing elements into new hash table

Rehashing Example

h(x) = x % 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	
3	24
4	
5	
6	13

TableSize = 7

$\lambda_{max} = 0.5$

Rehashing
since $\lambda > \lambda_{max}$

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

TableSize = 17
nextprime(TableSize * 2)

h(x) = x % 17

$\lambda = 0.29$

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

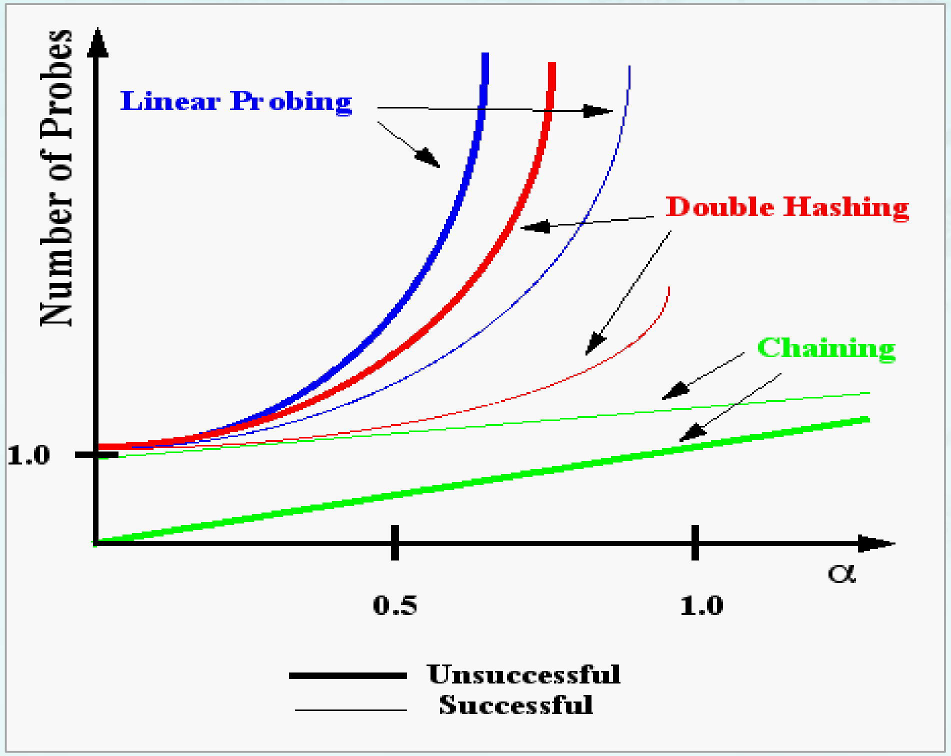
Rehashing Implementation

- When to rehash
  - When load factor reaches some threshold (e.g.,. $\lambda \geq 1.0$), OR
  - When an insertion fails (open addressing cases)

- Applies across collision handling schemes

Summary(1/2)

- Hash tables support fast insert and search
  - O(1) average case performance
  - Deletion possible, but degrades performance (but, not in chaining)
- Not suited if ordering of elements is important
- Expected number of probe vs. Load factor



Summary(2/2)

- Many 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.

Points to remember - Hash tables

- Table size prime
- Table size larger than number of inputs (to maintain λ << 1.0)
- Tradeoffs between chaining vs. probing
- Collision chances decrease in this order:
  linear probing선형조사법 ➔ quadratic probing이차조사법 ➔ double hashing이중해싱법
- Rehashing recommended 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

Hash Table(1) Using **list** in STL

```
struct Hash {
  int           tablesize;                     // hash table size
  list<string>* hashtable;                     // pointer to an array of buckets M
  int           nelements;                     // number of elements in table N
  double        threshold;                     // max_loadfactor N/M

  Hash(int size = 2, double lf = 1.0) {        // a magic number, use a small prime
    tablesize = size;                          // using list<string>* for a pedagogical purpose
    hashtable = new list<string>[size];        // but vector<list<string>> may be used
    nelements = 0;
    threshold = lf;                            // rehashes if loadfactor >= threshold
  }
  ~Hash() {
    delete[] hashtable;
  }
};
```

Hash Table(1) Using **list** in STL

```
// Notice that ht is passed by reference when its pointer may be changed inside.
// Passing by reference has not been used for speed-up the code.

int hashfunction(Hash* ht, int key);          // hash function for int key
int hashfunction(Hash* ht, string key);       // hash for string key

void rehash(Hash*& ht);                       // rehashes - doubles its tablesize
bool insert(Hash*& ht, string key);           // inserts key
                                              // rehashes if loadfactor >= threshold
bool erase(Hash*& ht, string key);            // erases key, returns true if successful
list<string> find(Hash* ht, string key);      // returns its list if found

void clear(Hash*& ht);                     // clear the table
void show(Hash* ht, bool show_empty);      // show the table
int nextprime(int x);                      // returns the next prime
int tablesize(Hash* ht);                   // returns hash table size
int nelements(Hash* ht);                   // returns number of elements in table
double loadfactor(Hash* ht);               // returns nelements/tablesize
double threshold(Hash* ht);                // returns threshold(or max_loadfactor)
void threshold(Hash* &ht, double th);      // sets threshold and rehashes if needed
```

Hash Table(2) Using **list** in STL

```
// std::pair is a STL container or class which a pair of public members called
// 'first' and 'second'. The types of two members may be the same or different.

typedef std::pair<string, int> wordcount;

struct Hash {
  int                tablesize;                // hash table size or bucket_count()
  list<wordcount>* hashtable;                  // pointer to an array of buckets
  int                nelements;                // number of elements in table or size()
  double             threshold;                // max_loadfactor

  Hash(int size = 2, double lf = 1.0) {       // a magic number, use a small prime
    tablesize = size;                          // using list<wordcount> for pedagogical purpose
    hashtable = new list<wordcount>[size];    // but vector<list<wordcount>> may be used
    nelements = 0;
    threshold = lf;                            // rehashes if loadfactor >= threshold
  }
  ~Hash() {
    delete[] hashtable;
  }
};
```

Hash Table(2) Using **list** in STL

```
// Notice that ht is passed by reference only when its pointer may be changed inside.
// Passing by reference has not been used for speed-up the code.

int hashfunction(Hash* ht, int key);          // hash function for int key
int hashfunction(Hash* ht, string key);       // hash function for string key

void rehash(Hash*& ht);                       // rehashes - doubles its tablesize
void rehash(Hash*& ht, int usersize);         // rehashes using user-specified tablesize
bool insert(Hash*& ht, string key);           // inserts key & rehashes if loadfactor>=threshold
bool erase(Hash* ht, string key);             // erases key and returns true if successful
list<wordcount> find(Hash* ht, string key);   // returns its bucket list if found
void clear(Hash* &ht);                        // clear the table

void show(Hash* ht, bool show_empty=false, int show_n=0); // show the table

int nextprime(int x);                         // returns the next prime
int tablesize(Hash* ht);                      // returns the table size
int nelements(Hash* ht);                      // returns number of elements in the table
double loadfactor(Hash* ht);                  // returns nelements/tablesize
double threshold(Hash* ht);                   // returns threshold(or max_loadfactor)
void threshold(Hash*& ht, double th);         // sets threshold and rehashes if needed
```

Hash Table(3) Using **unordered_map** in STL

- Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.

- The key value is generally used to uniquely **identify the element**, while the mapped value is an object with the content associated to this key.

- Internally, the elements are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets **depending on their** hash values **to allow for fast access to individual elements directly by their key values** (with a **constant average time complexity** on average).

- It is faster than **map** containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.
- It implements **the direct access operator** (operator[]) which allows for direct access of the mapped value using its key value as argument.
- Iterators in the container are at least forward iterators.

unordered_map::begin/end example

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main () {
  unordered_map<string, int> mymap;
  mymap = {{"all", 3}, {"the", 2}, {"time", 5}};

  cout << "mymap contains:";
  for (auto it = mymap.begin(); it != mymap.end(); ++it)
    cout << " " << it->first << ":" << it->second;
  cout << endl;

  cout << "mymap's buckets contain:\n";
  for (unsigned i = 0; i < mymap.bucket_count(); ++i) {
    cout << "bucket #" << i << " contains:";
    for (auto ir = mymap.begin(i); ir!= mymap.end(i); ++ir)
      cout << " " << ir->first << ":" << ir->second;
    cout << endl;
  }
  return 0;
}
```

```
mymap contains: time:5 the:2 all:3
mymap's buckets contain:
bucket #0 contains:
bucket #1 contains:
bucket #2 contains:
bucket #3 contains:
bucket #4 contains: the:2
bucket #5 contains:
bucket #6 contains:
bucket #7 contains: time:5
bucket #8 contains:
bucket #9 contains:
bucket #10 contains: all:3
bucket #11 contains:
bucket #12 contains:
```

unordered_map::begin/end example

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main () {
  unordered_map<string, int> mymap;
  mymap = {{"all", 3}, {"the", 2}, {"time", 5}};

  cout << "mymap contains:";
  for (auto it = mymap.begin(); it != mymap.end(); ++it)
    cout << " " << it->first << ":" << it->second;
  cout << endl;

  cout << "mymap's buckets contain:\n";
  for (unsigned i = 0; i < mymap.bucket_count(); ++i) {
    cout << "bucket #" << i << " contains:";
    for (auto ir = mymap.begin(i); ir!= mymap.end(i); ++ir)
      cout << " " << ir->first << ":" << ir->second;
    cout << endl;
  }
  return 0;
}
```

```
for (auto x : mymap)
    cout << " " << x.first << ":" << x.second;
```

Data Structures:
Hashing & Hash Tables

1. Hashing & Hash Table
2. Collision
3. Rehashing
4. Coding
   - Using list in STL
   - Using unordered_map in STL

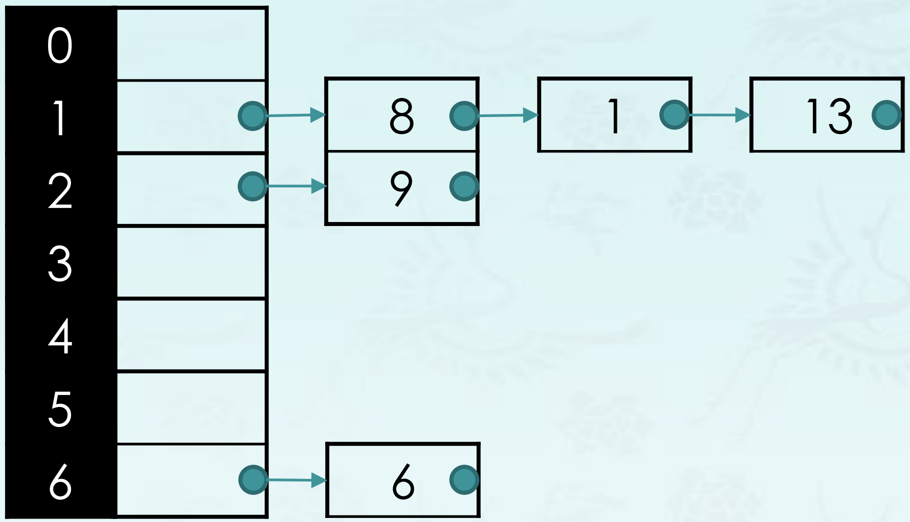
Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

Chain Hashing Example

Insert sequence: **8, 1, 9, 6, 15**

h(x) = k mod 7

Empty Table



$h_0(8)$ = 8 mod 7 = 1
$h_0(1)$ = 1 mod 7 = 1 (collision) new node
$h_0(9)$ = 9 mod 7 = 2
$h_0(6)$ = 6 mod 7 = 6
$h_0(15)$ = 15 mod 7 = 1 (collision) new node

Linear Hashing Example

Insert sequence: **8, 1, 9, 6, 15**

h(x) = x % 7

	Empty Table	After 8	After 1	After 9	After 6	After 15
0						
1		8	8	8	8	8
2			1	1	1	1
3				9	9	9
4						15
5						
6					6	6

$h_0(8) = 8 \bmod 7 = 1$

$h_0(1) = 1 \bmod 7 = 1$
$h_1(1) = (h(1)+1) \bmod 7 = 2$

$h_0(9) = 9 \bmod 7 = 2$
$h_1(9) = (h(9)+1) \bmod 7 = 3$

$h_0(6) = 6 \bmod 7 = 6$

$h_0(15) = 15 \bmod 7 = 1$
$h_1(15) = (h(15)+1) \bmod 7 = 2$
$h_2(15) = (h(15)+2) \bmod 7 = 3$
$h_3(15) = (h(15)+3) \bmod 7 = 4$

Double Hashing Example

Insert sequence: **8, 1, 9, 6, 13**

$h(x) = x \% 10$

$h'(x) = x \% 7$

	Empty Table	After 8	After 1	After 9	After 6	After 13
0						
1		8	8	8	8	8
2				9	9	9
3						13
4						
5			1	1	1	
6					6	

$h_0(8) = 8 \bmod 7 = 1$

$h_0(1) = 1 \bmod 7 = 1$
$h_0(1) = (h(1)+h'(1)) \bmod 7 = (1+5-(1 \bmod 5)) \bmod 7 = 5$

$h_0(9) = 9 \bmod 7 = 2$

$h_0(6) = 6 \bmod 7 = 6$

$h_0(13) = 15 \bmod 7 = 1$
$h_0(13) = (h(13)+ 1*h'(13)) \bmod 7 = (6+1*(5-(13 \bmod 5))) \bmod 7= 1$
$h_0(13) = (h(13)+ 2*h'(13)) \bmod 7 = (6+2*2) \bmod 7= 3$