Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

Hash Tables

Last updated: **01/03/2011**

Tang Wai Chung, Matthew

Introduction

- Many applications require a dynamic set that supports only the **dictionary** operations insert, search and delete
  - The set is often called a **symbol table** (ST)
- A hash table is an **effective** data structure for implementing dictionaries.
- Though searching an element in a hash table can take as long as $\Theta(N)$ in the **worst** case, the **expected** time to search for an element is only $O(1)$.

Simple Example of Hashing

We want to build a symbol table for the list of words

onion	**potato**	**salt**
tomato	**mushroom**	**chicken**

To convert the characters into array indices, encode the characters as follows:

$$a = 1,\ b = 2,\ \ldots,\ z = 26$$

Then sum up the values in each word

$h(\text{onion}) = 15 + 14 + 9 + 15 + 14 = 67$

$h(\text{chicken}) = 53$

$h(\text{potato}) = 16 + 15 + 20 + 1 + 20 + 15 = 87$

$h(\text{tomato}) = 84$

$h(\text{salt}) = 19 + 1 + 12 + 20 = 52$

$h(\text{mushroom}) = 122$

Simple Example of Hashing

Divide all sums by 6 and use the remainder as the array index:

h(onion) mod 6	= 67 mod 6 = 1
h(potato) mod 6	= 87 mod 6 = 3
h(salt) mod 6	= 52 mod 6 = 4
h(tomato) mod 6	= 84 mod 6 = 0
h(mushroom) mod 6	= 122 mod 6 = 2
h(chicken) mod 6	= 53 mod 6 = 5

tomato
onion
mushroom
potato
salt
chicken

To check whether a word belongs to the symbol table:

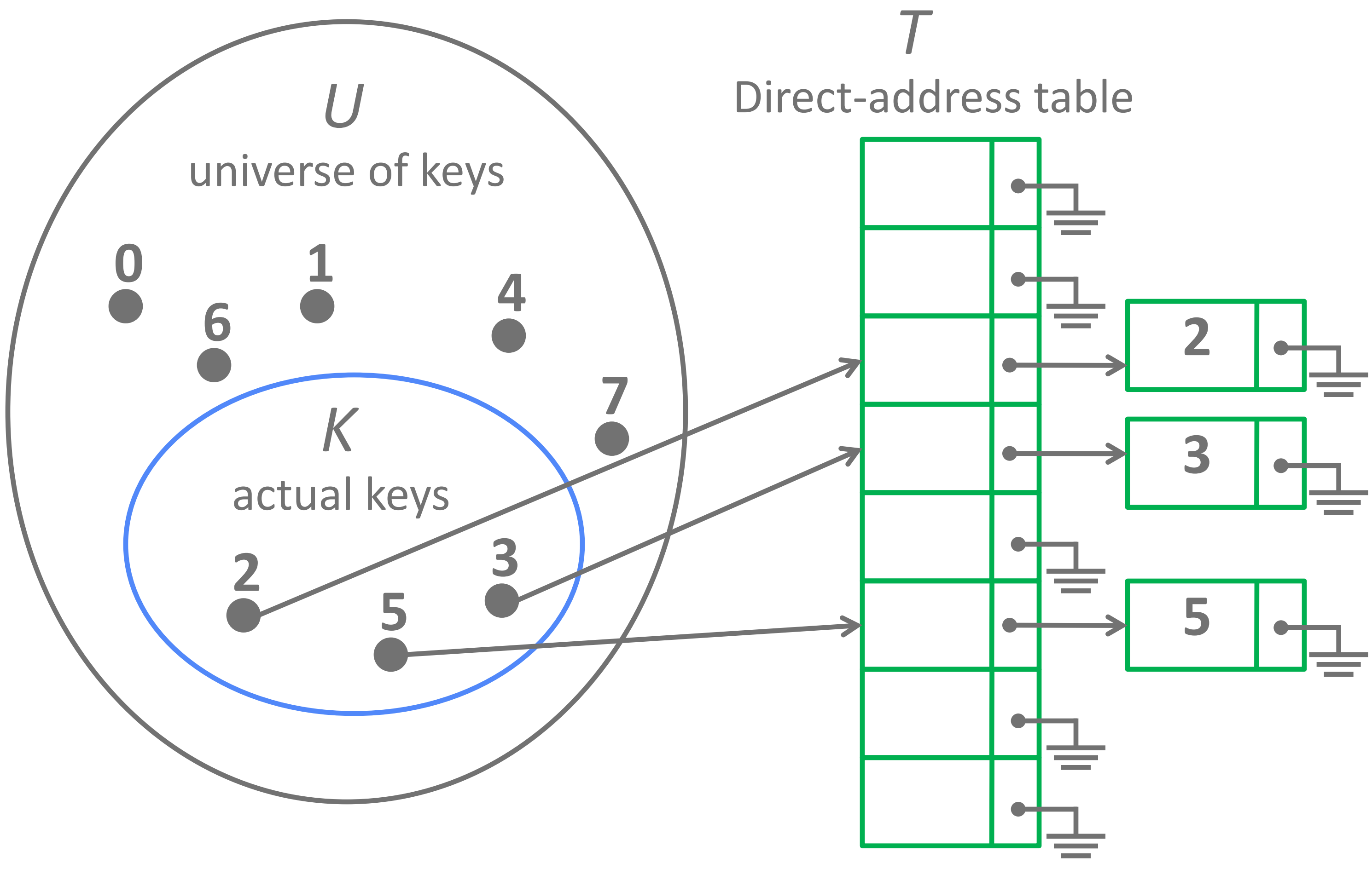
Identify the slot by first summing the character values and then modulo 6.

If it is not empty, the word is present.

Direct-Address Tables

- Direct addressing is a simple technique that works well when the universe U (the complete set) of keys is reasonably **small**.
- Suppose that an application needs a **dynamic** set in which each element has a **key** drawn from $U = \{0, 1, ..., M - 1\}$ and M is not too large.
- If we assume no two elements have the same key, then we can use an array, or **direct-address table**, in which each position (slot) corresponds to a key in U.

Direct-Address Tables: Example



Dictionary Operations

- If we use an array with size ***M***, then
  - ***search***: return T[k]
  - ***insert***: T[key(x)] = x
  - ***delete***: T[key(x)] = **NULL**
- Each of the above operations is fast: ***O*(1)**
- **Storage** of the elements:
  1. with the direct-address table itself
  2. externally with a pointer from a slot in the table

Hash Tables

- The difficulty with direct addressing is obvious: if the universe ***U*** is large, storage a table of size **|*U*|** is impractical.
  - **e.g. ASCII string with 10 chars: 26^{10}**
- The set ***K*** of keys actually stored may be so small relative to ***U***.
  - **e.g. actually words with 10 chars**
- When the set ***K*** is small, a hash table requires much less storage than a direct address table. Memory required: **$\Theta(|K|)$**

Hash Function

- With hashing, an element with key ***k*** is stored in slot ***h*(*k*)**.
- We use a hash function to compute the slot from a key *k*.
  - Let ***M*** be the size of the hash table, then

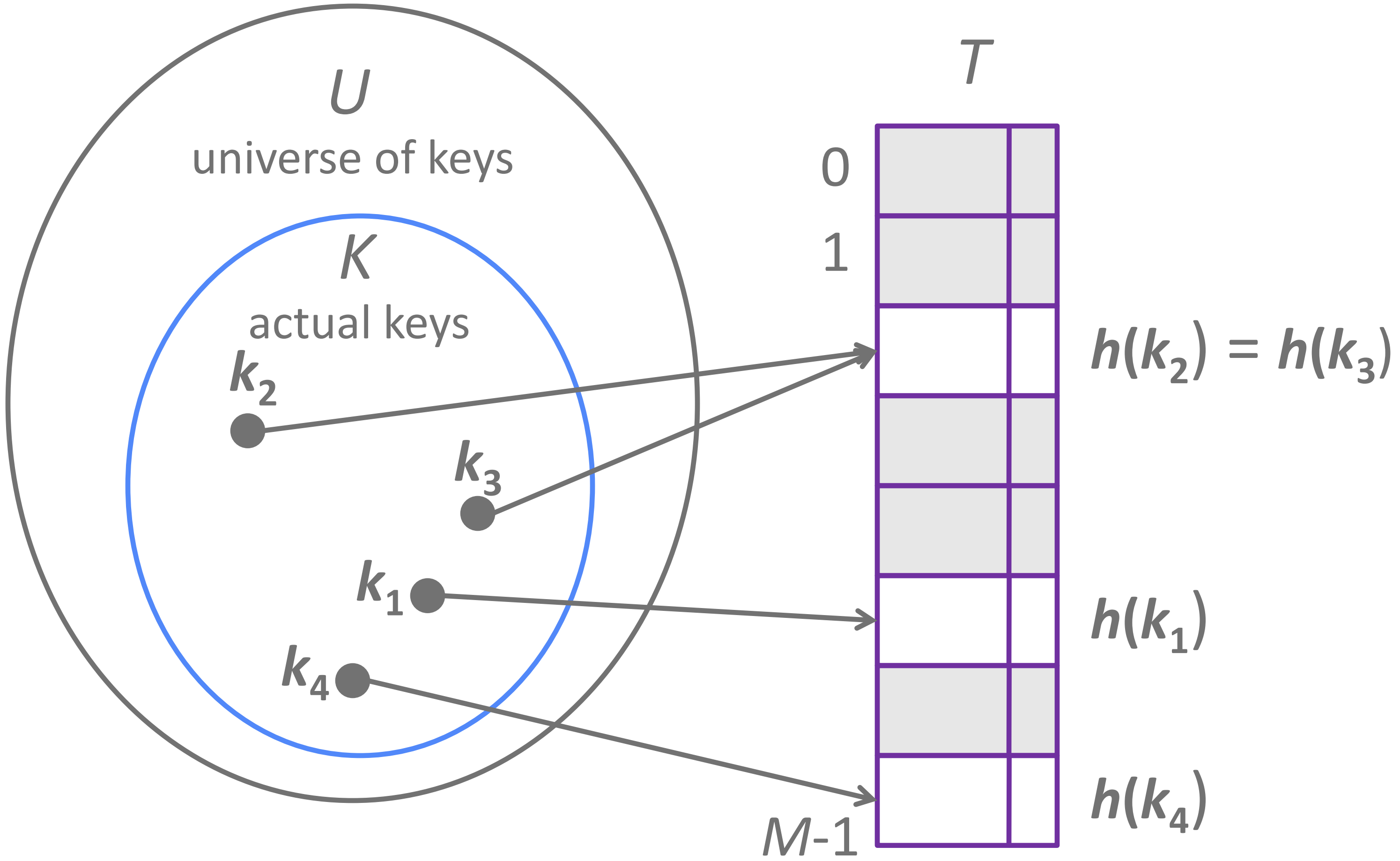
$$h: U \rightarrow \{0, 1, \ldots, M-1\}$$

- We say that an element with key *k* **hashes** to slot *h*(*k*) while *h*(*k*) is the **hash value** of key *k*.
- ☑ Hash function **reduces** the range of indices needed to be handled: ***M*** instead of **|*U*|**.

Hash Function (2)

- The performance of the hashing depends on **how well** the hash function **distribute** the keys to the *m* slots.
- **Simple uniform hashing**: any given element is **equally likely** to be hash into any of the *M* slots.
- **Hitch** 💣
  Collisions: two keys may hash to the **same** slot
  - We need to find methods to handle collisions – **collision solution** (handling).

Hash Tables: Example



Choosing a Hash Function

- The hash function *h* should be:
  - be **easy** and **quick** to be computed.
  - achieve an **even** distribution of the keys.
  - **deterministic** that a given input k always produce same output h(k).
- It is a good idea to ensure that the table size is **prime**. **WHY?**

Hash Function for Integers

If the input keys are integers, then simply returning ***K* mod *M*** is generally a reasonable strategy.

When the keys are random integers, this function is simple to compute and can distribute keys evenly.

This applies to data that interpreted as integers (e.g. floating point numbers considered as bytes)

A Simple Hash Function for Strings

- Strings are common keys as well.
- A Simple hash function for strings:
  add up the ASCII values of the characters and mod the sum.

```
unsigned int hash(char *key, unsigned int m){
    unsigned int hash_val = 0;
    while (*key != '\0')
        hash_val += *key++;
    return (hash_val % m);
}
```

☹ **Problem**: this hash function **does not hash evenly**.

Consider strings shorter than 8 in a hash table with M = 10007, the function can only return values up to **127 x 8 = 1016**.

A Better Hash Function

- Examines the three letters of the keys
- The magic number **27** comes from the total number of English alphabets and the space (you may have to adjust it).
- If all 3 letters are **random**, then we have $26^3 = 17,576$ combinations that can fit quite well on a table with M = 10,007.
- Unfortunately, English words are **not** random. A large on-line dictionary reveals only 2,851 combinations with the first 3 letters.

```
unsigned int hash(char *key, unsigned int m){
    return (key[0] + 27 * key[1] + 729 * key[2]) % m;
}
```

A Good Hash Function

- This hash function includes all characters in the key.
- It computes $$\sum_{i=0}^{N-1} k[N-i-1] \times 27^i \Rightarrow \sum_{i=0}^{N-1} k[N-i-1] \times 32^i$$
  where *N* is the length of the key.
- It computes the polynomial function using **Horner's rule**.
- **More efficiency**: the multiplication of 32 can be achieved by shifting 5 bits to the left.

```
unsigned int hash(char *key, unsigned int m){
   unsigned int hash_value = 0;
   while (*key != '\0')
        hash_val = (hash_val << 5) + *key++;
    return (hash_val % m);
}
```

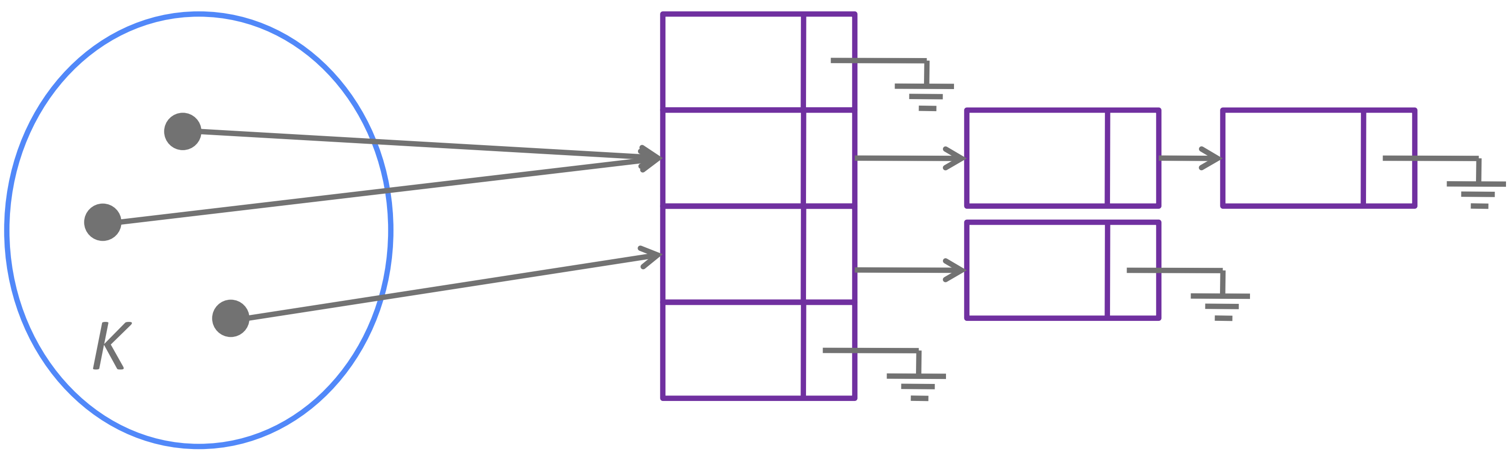
Horner's rule:
$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)\cdots))$$

More About Hash Functions

- It is common not to use all characters – the length and properties of the keys would influence the choice.
- The hash function might include a couple characters from each field.
- **Truncation**: ignore part of the key and use the remaining part directly as the index.
  - e.g. 62,538,194 hash to 394 by taking 1st, 2nd and 5th digits.
- **Folding**: partition the key into several parts and combine parts in a **convenient** way.
  - e.g. 62,538,194 maps to 625 + 381 + 94 = 1100 and is truncated to 100.

Collision Resolution by Chaining

- In **chaining**, we put all elements that hash to the same slot in a **linked list** (or normal list).
- Slot *j* contains a **header** node of the list that stores ALL elements that are hashed to *j*.
  - contains a **NULL** link if it is empty.



Hash Table: Implementation

- Suppose we reuse our list ADT in the implementation of the hash table, the table is actually an array of lists.

```
typedef struct hashtbl_s {
    int m;                      /* hash table size */
    list_t *slots;              /* array of lists */
} hashtbl_t;
```

```
hashtbl_t *hash_init_tbl(int m){
    int i;
    hashtbl_t *ht = (hashtbl_t *)malloc(sizeof(hashtbl_t));
    ht->m = next_prime(m);
    ht->slots = (list_t *)malloc(ht->m * sizeof(list_t));
    for (i = 0; i < ht->m; i++)
        ht->slots[i] = list_create();

    return ht;
}
```

Dictionary Operations: *Insert*

- *insert*(T, x): insert a new element x to the hash table T where the **key** of x is ***key*(*x*)**
- We need to hash *key*(x) to get the list stored at the slot and insert x to the list.

```
void hash_insert(hashtbl_t *ht, int key){
    list_t list = ht->slots[hash(key, ht->m)];
    if (list_find(list, key))
        return; /* key found: do nothing */
    list_insert(list, list, key);
}
```

Complexity = $\Theta(1 + \alpha)$

since list insertion is $O(1)$.

α is the average length of the list

Dictionary Op.: *Delete* & *Find*

- The two operations are similar: first hash to find the slot, get the list and then continue with the basic list search/delete operations.
- Complexity = $\Theta(1 + \alpha)$ *average length of the list*

```
void hash_delete(hashtbl_t *ht, int key){
    list_t list = ht->slots[hash(key, ht->m)];
    list_delete(list, key);
}
```

```
pos_t hash_find(hashtbl_t *ht, int key){
    list_t list = ht->slots[hash(key, ht->m)];
    return list_find(list, key);
}
```

Analysis of Chaining

Load Factor α

Given a hash table T with M slots that stores N elements, we define the load factor $\alpha = N/M$.

- The load factor depends on how well the hash function distributes the keys among m slots:
  - **Worse-case**:
    Horrible. The hash table is the same as a linked list with all N elements hash to the same slot. $\alpha = N$.
  - **Average-case**:
    Assume simple uniform hashing, any given element is **equally likely** to hash into any of the M slots. $\alpha = N/M$.
- If we choose M s.t. $N = O(M)$ then $\alpha = O(1)$. Both ***find*** and ***delete*** are $O(1)$ on **average**.

Average No. of Probes in Chaining

When chaining is used, the average number of probes required to search in a hash table of size M that contains $N = \alpha M$ keys is about

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

for hits

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

for misses

load factor (α)	1/2	2/3	3/4	9/10
search hit	1.5	2.0	3.0	5.5
search miss	2.5	5.0	8.5	55.5

Remarks:
Chaining performs badly when the hash table is filling up.

Open Addressing

- In **open addressing**, all elements are stored in the hash table itself.
  - Each slot contains either **an element** or **NULL**.
- The hash table can ***fill up*** so that no further insertions can be made → α can never exceed 1.
- **Advantage**: pointers are avoided.
- We computer the **sequence** of slots to be examined.
- ***insert***: successively prove the hash table until we find an empty slot to save the key.
- Probing in fixed order 0, 1, ..., $M - 1$ is **unwise** since it requires $\mathbf{\Theta}(N)$ search time.

Open Addressing (Cont')

- The **probe sequence** depends upon the key being inserted.
- We extend the hash function to include the probe number as a **second input**.

$$h: U \times \{0, 1, \ldots, M - 1\} \rightarrow \{0, 1, \ldots, M - 1\}$$

- With open addressing, for every key *k*, the probe sequence is:

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1)\rangle$$

as an permutation of $\{0, 1, \ldots, M - 1\}$

Open Addressing: *insert*

- We assume the element only contains the key itself
- Each slot contains either a key or a value for NULL (**HASH_NULL**).

```
unsigned int hash_insert(hashtbl_t *ht, int key){
    int i;
    unsigned int j;

    for (i = 0; i != ht->m; i++){
        j = oahash(key, i, ht->m);
        if (ht->slots[j] == HASH_NULL){
            ht->slots[j] = key;
            return j;
        }
    }
    perror("oahash.c: hash table overflow\n");
    return HASH_NULL;
}
```

Open Addressing: *find*

- The search **terminates** when it finds an empty slot, since the ***key*** would have been inserted there.
- **Assumption**:
  the key is **not** deleted from the table once before

```
unsigned int hash_find(hashtbl_t *ht, int key){
    int i = 0;
    unsigned int j;

    do {
        j = oahash(key, i, ht->m);
        if (ht->slots[j] == key)
            return j;
    } while (ht->slots[j] != HASH_NULL && i++ != ht->m);

    return HASH_NULL;
}
```

Open Addressing: *delete*

- Delete from an open-addressing hash table is **difficult**.
- When we delete a key from slot *i*, we cannot simply mark that slot as empty by storing a NULL.
  - then we fail to retrieve any key whose probe sequence contains slot *i.*
- **Solution**: mark the slot deleted instead. (you need to modify insert accordingly)
  - **Drawback**: the search time is no longer dependent on the load factor α
- When keys have to be deleted, **chaining** is more preferred.

Linear Probing

- Given an ordinary hash function h', the method of linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \bmod M$$

- If the slot $T[h'(k)]$ is occupied, we continue to probe the next one $T[h'(k) + 1]$ and so on, until we have an empty slot.
- Easy to implement, but suffers from **primary clustering**: long runs of occupied slots build up, increasing average search time.

```
static unsigned int oahash(int key, unsigned int i, unsigned int m){
    return (hash(key, m) + i) % m;
}
```

Linear Probing: Example

- $h'(k) = k \bmod 10$, $M = 10$
- Insert 89, 18, 49, 58, 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

Quadratic Probing

- Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod B$$

- The initial position probed is $T[h'(k)]$; late positions probe are offset by an amount quadratic to the probe number i.
- Works much better than linear probing but parameters c_1, c_2 and m have to be selected carefully to make full use of the hash table.
- **secondary clustering**: if $h(k_1, 0) = h(k_2, 0)$, then $h(k_1, i) = h(k_2, i)$

```
static unsigned int oahash(int key, unsigned int i, unsigned int m){
    return (hash(key, m) + i * i) % m;
}
```

Quadratic Probing: Example

- $h'(k) = k \bmod 10$, $M = 10$
  $c_1 = 0$, $c_2 = 1$
- Insert 89, 18, 49, 58, 69

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

Double Hashing

- Double hashing is one of the **best** methods for open addressing because the permutations produced is quite random.
- It uses hash functions of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod M$$

- Initial probe position is $h_1(k)$ and successive probe positions are offset by $h_2(k)$ (modulo m)
- The value $h_2(k)$ must be relatively **prime** to the size M so that the whole table can be searched.
- **Method 1**: choose $M = 2N$ and design h_2 to return only odd numbers.

Double Hashing (Cont')

Method 2: choose *M* prime and design h_2 to return a positive integer < *M*

e.g. $h_1(k) = k \bmod M$, $h_2(k) = 1 + (k \bmod (M - 1))$

```
static unsigned int hash2(int key, unsigned int m){
    return (1 + (key % (m - 1)));
}

static unsigned int oahash(int key, unsigned int i, unsigned int m){
    return (hash(key, m) + i * hash2(key, m)) % m;
}
```

- Double hashing improves over linear or quadratic probing in that $\Theta(M^2)$ sequences are used.
- It appears to be very close to the **ideal performance**.

Double Hashing: Example

- $h_1(k) = k \bmod 10$,
  $M = 10$
  $h_2(k) = 7 - (k \bmod 7)$
- Insert
  89, 18, 49 (collide),
  58(collide), 69(collide)
- $h_1(49) + h_2(49)$
  $= (9 + (7 - 0))$ % $10 = 6$

  $h_1(58) + h_2(58)$
  $= (8 + (7 - 2))$ % $10 = 3$

  $h_1(69) + h_2(69)$
  $= (9 + (7 - 6))$ % $10 = 0$

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

Analysis of Open Addressing

Given an open-address hash table with load factor $\alpha = N/M < 1$, the expected number of probes in an **unsuccessful** search is at most $1/(1 - \alpha)$, assuming uniform hashing.

Intuitive Interpretation

1 probe is always made.

With probability approx. α, the first probe finds an occupied slot and we need another probe.

With probability approx. α^2, the first two slots are occupied and we need the third probe.

$$\text{No. of probes} = 1 + \alpha + \alpha^2 + \ldots = 1/(1 - \alpha)$$

The no. of probes needed for insertion is the same as that in an unsuccessful search.

Analysis of Open Addressing (2)

Given an open-address hash table with load factor $\alpha = N/M < 1$, assuming uniform hashing, the expected number of probes in a **successful** search is at most

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

PROOF

If ***k*** was the $(i+1)^{th}$ key inserted, the expected number of probes made in a search for ***k*** is at most $\dfrac{1}{1-i/M} = \dfrac{M}{M-i}$

Average over all *n* keys gives us the average no .of probes:

$$\frac{1}{N}\sum_{i=0}^{N-1}\frac{M}{M-i} = \frac{M}{N}\sum_{i=0}^{N-1}\frac{1}{M-i} = \frac{1}{\alpha}\sum_{k=M-N+1}^{M}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{M-N}^{M}\frac{1}{x}dx = \frac{1}{\alpha}\ln\frac{M}{M-N} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Avg. # of Probes in Double Hashing

We can ensure that the average cost of all searches is less than ***t*** probes by keeping the load factor less than

$$1 - 1/\sqrt{t}$$ **for linear probing**

$$1 - 1/t$$ **for double hashing**

load factor (α)	1/2	2/3	3/4	9/10
search hit	1.4	1.6	1.8	2.6
search miss	1.5	2.0	3.0	5.5

Summary

- Direct-address table vs Hash table
- Concept of **simple uniform hashing**
- Choice of hash functions for **integers** and **strings**
- **Collision handling**
  - **Chaining**
    - Implementation
    - Performance analysis
  - **Open addressing**: linear & quadratic probing, double hashing
    - Implementation
    - Performance analysis