CIS2520

7. Hashing

Reading suggestion: Chapter 11 of the textbook

TABLE ADT

TABLE ADT: Introductory Examples (1/2) 7.3

SIN: 522-145-678
Last name: Smith
First name: John
Title: Associate Professor
......

SIN: **501-997-403**
Last name: Roe
First name: Jane
Title: Professor
......

SIN: 593-021-844
Last name: **Major**
First name: **Mary**
Title: **Assistant Professor**
......

Insert
Update
Remove
Search
Retrieve

SIN: 570-133-981
Last name: Smith
First name: John
Title: **Professor**
......

SIN: 537-702-556
Last name: Miles
First name: Richard
Title: Assistant Professor
......

TABLE ADT: Introductory Examples (2/2) 7.4

Code: PHL
Country: Pennsylvania
City: Philadelphia
Volume: 31 million
......

Code: AKL
Country: New Zealand
City: Auckland
Volume: 13 million
......

Code: FRA
Country: Germany
City: Frankfurt
Volume: 53 million
......

Insert
Update
Remove
Search
Retrieve

Code: ORY
Country: France
City: Paris
Volume: 25 million
......

Code: HKG
Country: China
City: Hong Kong
Volume: 50 million
......

TABLE ADT: Definition 7.5

$\mathbb{N}$: The set of nonnegative integers
I: A nonempty set
K: A nonempty set
$\preccurlyeq$: A total order relation on K

Table of **items** of type I and **keys** of type K:

A set $\{(item_1,key_1),(item_2,key_2),\ldots,(item_n,key_n)\}$ such that
- $\forall i \in 1..n,\ (item_i \in I \wedge key_i \in K)$
- $\forall i \in 1..n,\ \forall j \in 1..n,\ (key_i = key_j \rightarrow item_i = item_j)$

Each pair (item,key) is a table **entry**

Table[I,K]:

The set of all tables of items of type I and keys of type K

TABLE ADT: Operations 7.6

$\left.\begin{array}{l}\textbf{Create}: \varnothing \rightarrow \text{Table[I,K]}\end{array}\right\}$ **constructor**

$\left.\begin{array}{l}\textbf{Insert}: \text{IxKxTable[I,K]} \rightarrow \text{Table[I,K]}\\ \textbf{Update}: \text{IxKxTable[I,K]} \rightarrow \text{Table[I,K]}\\ \textbf{Remove}: \text{KxTable[I,K]} \rightarrow \text{Table[I,K]}\end{array}\right\}$ **mutators**

$\left.\begin{array}{l}\textbf{Full}: \text{Table[I,K]} \rightarrow \text{Boolean}\\ \textbf{Empty}: \text{Table[I,K]} \rightarrow \text{Boolean}\\ \textbf{Size}: \text{Table[I,K]} \rightarrow \mathbb{N}\\ \textbf{Entry}: \text{KxTable[I,K]} \rightarrow \text{Boolean}\\ \textbf{Retrieve}: \text{KxTable[I,K]} \rightarrow \text{I}\end{array}\right\}$ **accessors**

$\left.\begin{array}{l}\{\neg Full(t) \wedge \neg Entry(k,t)\}\ Insert(i,k,t)\\ \{Entry(k,t)\}\ Update(i,k,t)\end{array}\right\}$ **preconditions**

$\left.\begin{array}{l}Remove(k,t)\ \{\neg Full(t) \wedge \neg Entry(k,t)\}\\ Insert(i,k,t)\ \{Retrieve(k,t)=i\}\end{array}\right\}$ **postconditions**

TABLE ADT: Representations (1/2) 7.7

✧ A table entry (item,key) can be represented by a C struct.

✧ A table can be represented by an **array** of (pointers to) table entries stored in ascending order of their keys.

PHL
ORY
GCM
HKG
GLA
AKL
FRA
LAX
DCA

AKL	DCA	FRA	GCM	GLA	HKG	LAX	ORY	PHL		

Traversal	Search	Insertion	Removal
O(n)	O(log n)	O(n)	O(n)

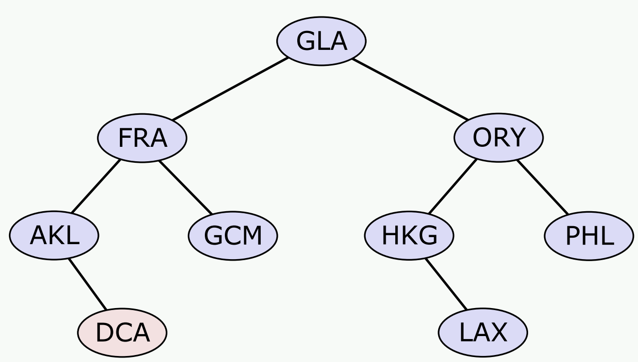
Reading suggestion: Chapter 11 of the textbook

TABLE ADT: Representations (2/2) 7.8

✧ A table entry (item,key) can be represented by a C struct.

✧ A table can be represented by an **AVL tree**, where each tree node stores a (pointer to a) table entry and pointers to other tree nodes.

PHL
ORY
GCM
HKG
GLA
AKL
FRA
LAX
DCA



Traversal	Search	Insertion	Removal
O(n)	O(log n)	O(log n)	O(log n)

Reading suggestion: Chapter 11 of the textbook

Table ADT

HASH TABLES

Collision Resolution

Algorithms

Hash and Probe Functions

Conclusion

HASH TABLES: Introductory Example (1/2) 7.10

Name: Neptune
Diameter: 50,000 km
Orbit: 30 AU
Moons: 13
......

Name: Mars
Diameter: 6,800 km
Orbit: 1.5 AU
Moons: 2
......

Name: Mercury
Diameter: 4,900 km
Orbit: 0.4 AU
Moons: 0
......

Name: Earth
Diameter: 12,800 km
Orbit: 1 AU
Moons: 1
......

Name: Jupiter
Diameter: 143,000 km
Orbit: 5.2 AU
Moons: 65
......

HASH TABLES: Introductory Example (2/2) 7.11

	A	B	…	Z	a	b	…	z
ASCII	**65**	**66**	**…**	**90**	**97**	**98**	**…**	**122**

$h(c_0c_1\ldots) = [\mathrm{ASCII}(c_0)+\mathrm{ASCII}(c_1)] \bmod 15$

h("Earth") = (**69**+**97**) mod 15 = 1
h("Jupiter") = 11
h("Mars") = 9
h("Mercury") = 13
h("Neptune") = 14
h("Pluto") = 8
h("Saturn") = 0
h("Uranus") = 4
h("Venus") = 7

0	Saturn
1	Earth
2	
3	
4	Uranus
5	
6	
7	Venus
8	Pluto
9	Mars
10	
11	Jupiter
12	
13	Mercury
14	Neptune

HASH TABLES: Perfect Hash Function 7.12

array: able to store up to m (pointers to) table entries
h: total function from the set of keys to the integer interval 0..m−1;
an arithmetic calculation transforms each key into an array index

perfect hash function **hash table**

To search for an item with key k, just look in slot h(k) of the array.
The h(k) values lie in a relatively small range.
The h(k) values are dispersed in that range.
The h(k) values are all different.

hash address

Pros: worst-case running time for search, insertion and removal is O(1).
Cons: traversal is O(n log n), and lost space in array.

Note that in the previous example,
the number of keys is equal to the number of entries.

HASH TABLES: General Case 7.13

array: able to store up to m (pointers to) table entries
h: total function from the set of keys to the integer interval 0..m−1;
an arithmetic calculation transforms each key into an array index

hash function **hash table**

To search for an item with key k, **look first** in slot h(k) of the array.
The h(k) values lie in a relatively small range.
The h(k) values are dispersed in that range.
Most h(k) values are different.

hash address

Pros: **expected** running time for search, insertion and removal is O(1).
Cons: traversal is O(n log n), and lost space in array.

Note that in the general case,
the number of keys is much greater than the number of entries.

Reading suggestion: Chapter 11 of the textbook

HASH TABLES: Collisions 7.14

0		
1		
2		
3		
4	PHL	HKG
5		
6	GCM	
7		
8	ORY	
9		
10		

key	h(key)
PHL	**4**
ORY	**8**
GCM	6
HKG	4
GLA	8
AKL	7
FRA	5
LAX	1
DCA	1

A	B	C	...	Z
0	**1**	**2**	**...**	**25**

h(PHL)
= (**15**×26²+**7**×26¹+**11**) mod 11
= 10333 mod 11 = **4**

h(ORY)
= (**14**×26²+**17**×26¹+**24**) mod 11
= 9930 mod 11 = **8**

Collision when $h(k_1)=h(k_2)$ and $k_1 \neq k_2$
Need for a **collision resolution policy**

Reading suggestion: Chapter 11 of the textbook

Table ADT
Hash Tables
COLLISION RESOLUTION
Algorithms
Hash and Probe Functions
Conclusion

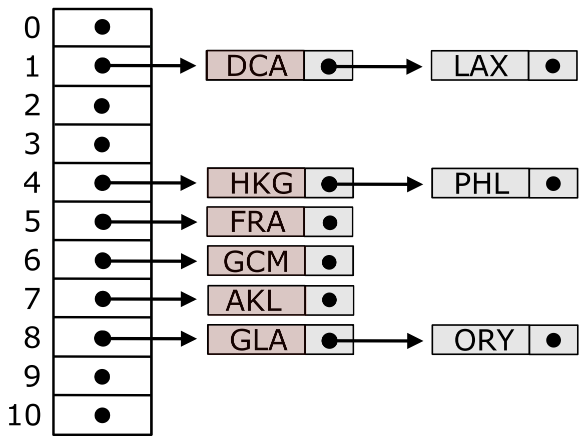
COLLISION RESOLUTION: Birthday Problem 7.16

collisions are relatively frequent
even in sparsely occupied hash tables

Consider hash table with m=365 slots:

- ✧ if n=366 entries inserted randomly
  probability of collision 100%
- ✧ if n=57 entries inserted randomly
  probability of collision greater than 99%
  (and table 16% full)
- ✧ if n=23 entries inserted randomly
  probability of collision greater than 50%
  (and table 6% full)

COLLISION RESOLUTION: Chaining 7.17

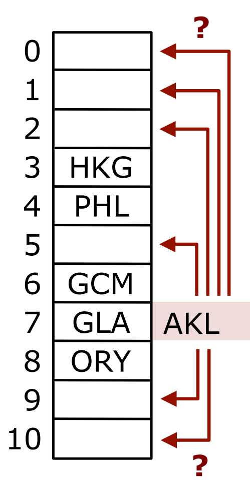


key	h(key)
PHL	4
ORY	8
GCM	6
HKG	4
GLA	8
AKL	7
FRA	5
LAX	1
DCA	1

chaining

one linked list per slot; two colliding entries placed on the same linked list

COLLISION RESOLUTION: Open Addressing 7.18



open addressing
colliding entries are inserted into empty slots

probe decrement function
total function p | K→ 1..+∞

probe sequence
sequence of slots probed when, e.g., inserting a new entry

cluster
sequence of adjacent occupied slots

key	h(key)
PHL	4
ORY	8
GCM	6
HKG	4
GLA	8
AKL	7
FRA	5
LAX	1
DCA	1

COLLISION RESOLUTION: Linear Probing 7.19

0	DCA
1	LAX
2	FRA
3	HKG
4	PHL
5	AKL
6	GCM
7	GLA
8	ORY
9	
10	

linear probing
if slot i is occupied try i−p(key)
(or m+i−p(key) if i−p(key)<0)
where p(key)=1 for any key

primary clustering
clusters tend to merge
and grow faster and faster

key	h(key)
PHL	4
ORY	8
GCM	6
HKG	4
GLA	8
AKL	7
FRA	5
LAX	1
DCA	1

5 collisions and up to 3 clusters

COLLISION RESOLUTION: Double Hashing (1/2) 7.20

0	
1	HKG
2	DCA
3	
4	PHL
5	FRA
6	GCM
7	AKL
8	ORY
9	LAX
10	GLA

double hashing
if slot i is occupied try i−p(key)
(or m+i−p(key) if i−p(key)<0)
where p(key) depends on key
very much like h(key)

double hashing
avoids primary
clustering

key	h(key)	p(key)
PHL	4	4
ORY	8	1
GCM	6	1
HKG	4	3
GLA	8	9
AKL	7	2
FRA	5	6
LAX	1	7
DCA	1	2

4 collisions and up to 5 clusters

COLLISION RESOLUTION: Double Hashing (2/2) 7.21

A	B	C	...	Z
0	**1**	**2**	**...**	**25**

h(PHL) = $\mathbf{15}\times26^2+\mathbf{7}\times26^1+\mathbf{11} = \mathbf{10333} \bmod 11 = 4$

h(ORY) = $\mathbf{14}\times26^2+\mathbf{17}\times26^1+\mathbf{24} = \mathbf{9930} \bmod 11 = 8$

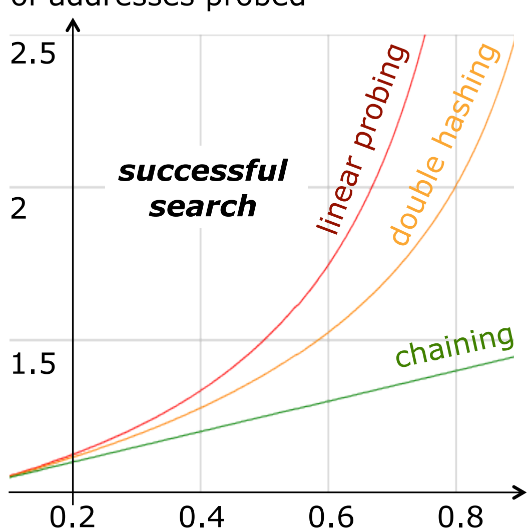
p(PHL) = max { 1 , (**10333** div 11) mod 11} = **4**

p(ORY) = max { 1 , (**9930** div 11) mod 11} = **1**

key	p(key)
PHL	**4**
ORY	**1**
GCM	1
HKG	3
GLA	9
AKL	2
FRA	6
LAX	7
DCA	2

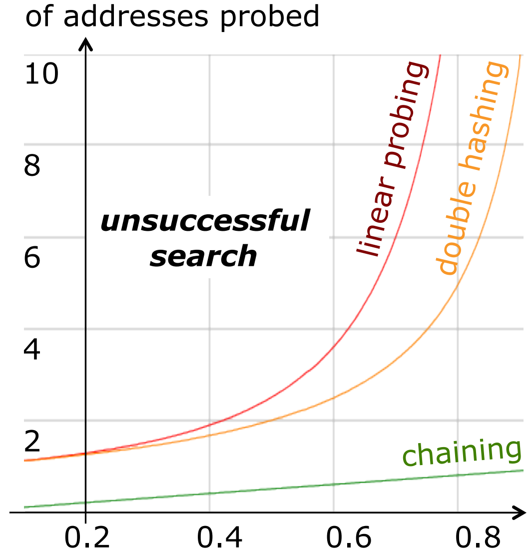
COLLISION RESOLUTION: Efficiency 7.22

average number
of addresses probed



load factor (i.e., n/m)

average number
of addresses probed



load factor (i.e., n/m)

ALGORITHMS: Initialize() and Size() 7.24

```
function Initialize (table)
        table.entries=0
        table.freeSlots=table.slots
        for i=0 to table.slots-1
                table[i]=nil

function Size (table)
        return table.entries
```

Reading suggestion: Chapter 11 of the textbook

ALGORITHMS: Full() and Empty() 7.25

```
function Full (table)
       if table.freeSlots=1
       then return true
       else return false

function Empty (table)
       if table.entries=0
       then return true
       else return false
```


ALGORITHMS: Remove() 7.26

```
function Remove (key, table)
       i=h(key)
       decrement=p(key)
       while table[i].key≠key
              i=i-decrement
              if i<0 then i=i+table.slots
       table[i].available=true
       table.entries=table.entries-1
```

ALGORITHMS: Insert() 7.27

```
function Insert (item, key, table)
       i=h(key)
       decrement=p(key)
       while table[i]≠nil and table[i].available=false
              i=i-decrement
              if i<0 then i=i+table.slots
       if table[i]=nil
       then table.freeSlots=table.freeSlots-1
       table[i].key=key
       table[i].item=item
       table[i].available=false
       table.entries=table.entries+1
```


ALGORITHMS: Search() 7.28

```
function Search (key, table)
       i=h(key)
       decrement=p(key)
       while table[i]≠nil and table[i].key≠key
              i=i-decrement
              if i<0 then i=i+table.slots
       if table[i]=nil or table[i].available=true
       then return -1
       else return i
```

ALGORITHMS: Update() and Retrieve() 7.29

```
function Update (item, i, table)
        table[i].item=item

function Retrieve (i, table)
        return table[i].item
```

Reading suggestion: Chapter 11 of the textbook

HASH FUNCTION: Ideal Case 7.31

A good hash function maps keys uniformly and randomly onto the full range of possible table locations. Ideally:

$$\forall i, \forall j, \left|\{k \in K \mid h(k) = i\}\right| = \left|\{k \in K \mid h(k) = j\}\right|$$

HASH FUNCTION: Division Method 7.32

Assume keys are strings of symbols from some alphabet.
The symbols are seen as base b digits.

Consider a key $s_t s_{t-1} \ldots s_1 s_0$. It is seen as the base b expansion of

$$i = s_t\, b^t + s_{t-1}\, b^{t-1} + \ldots + s_1\, b + s_0$$

Let m be the number of slots in the hash table.
Choose m prime, but do not choose it too close to a small power of b.

$$h(s_t s_{t-1} \ldots s_1 s_0) = i \bmod m$$

h(PHL)
$= (\mathbf{15} \times 26^2 + \mathbf{7} \times 26^1 + \mathbf{11}) \bmod 11$
$= 10333 \bmod 11 = \mathbf{4}$

A	B	C	...	Z
0	**1**	**2**	**...**	**25**

HASH FUNCTION: Other Methods 7.33

folding
divide sequence of digits into subsequences; combine them

k=512-678-890
h(k)=512+678+890=2080

middle squaring
take middle digits; square them; take middle digits again if necessary

k=512-678-890
678^2=459684
h(k)=5968

truncation
delete part of the key; use the remaining digits

k=512-678-890
h(k)=8890

CIS2520 Hashing

PROBE DECREMENT FUNCTION 7.34

0	6
1	2
2	9
3	5
4	1
5	8
6	4
7	11
8	7
9	3
10	10

h(k)=4
p(k)=3

A slot should not appear twice in a probe sequence.
At worst, a probe sequence should cover all slots.

p(k) must be relatively prime to the number m of slots
(e.g., m prime and p(k) in 1..m−1,
m power of 2 and p(k) odd).

division method
h(k) = i mod m
p(k) = max { 1 , (i div m) mod m}

Reading suggestion: Chapter 11 of the textbook

Table ADT
Hash Tables
Collision Resolution
Algorithms
Hash and Probe Functions
CONCLUSION

CONCLUSION 7.36

Best representation for a table
(e.g., sorted array, AVL tree, hash table)
depends on the frequency of the operations to be performed.

HASH TABLE

In the worst case,
searches, insertions and removals
take O(n) time.

In practice,*
searches, insertions and removals
are extremely fast and take O(1) time.

* *for collision resolution with open addressing*
keep the load factor below some threshold;
the lower the load factor the better

Reading suggestion: Chapter 11 of the textbook