Part 8. Hashing

CS 200 Algorithms and Data Structures

1

Outline

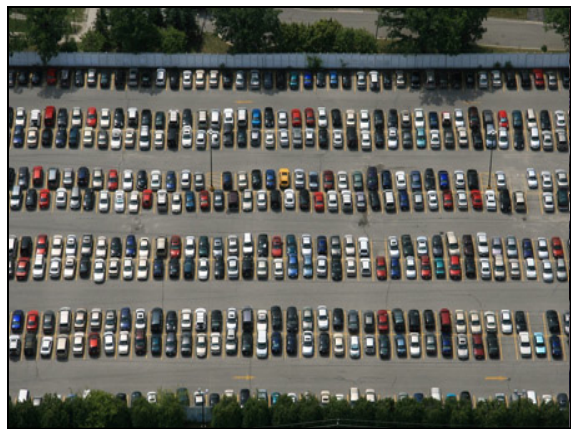
- **Hashing**
- Hash Functions
- Resolving Collisions
- Efficiency of Hashing
- Java Hashtable and HashMap

2

Table Implementations

	Search	Add	Remove
Sorted array-based	O(log n)	O(n)	O(n)
Unsorted array-based	O(n)	O(1)	O(n)
Balanced Search Trees	O(log n)	O(log n)	O(log n)

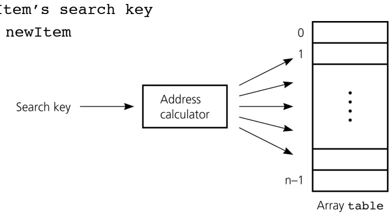
Can we build a faster data structure?



Tables in O(1)

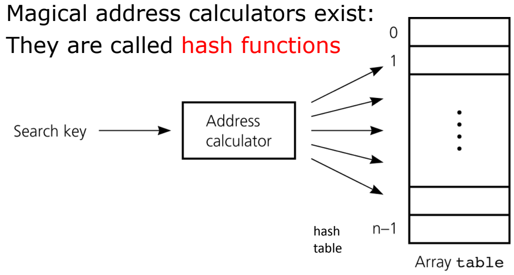
Suppose we have a magical address calculator…

```
tableInsert(in: newItem:TableItemType)
   i = index that the address calculator gives you
   for    newItem’s search key
   table[i] = newItem
```



Hash Functions and Hash Tables

Magical address calculators exist:
They are called hash functions



Outline

- Hashing
- **Hash Functions**
- Resolving Collisions
- Efficiency of Hashing
- Java Hashtable and HashMap

7

Simple Hash Functions

Credit card numbers

- 3: travel/entertainment cards (e.g. American Express and Diners Club)
  - Digits three and four are type and currency
  - Digits five through 11 are the account number
- 4: Visa
  - Digits two through six are the bank number
  - Digits seven through 12 or seven through 15 are the account number
- 5: Mastercard
  - Digits two and three, two through four, two through five or two through six are bank number
  - Till digits 15 are the account number
  - Digit 16 is a check digit

- To design a system to find an account based on the account number, we don't need 16 or more digits of numbers on the credit card.

Other simple example

- Phone exchange: need quick access to record corresponding to phone #.
  h(123-4567) = 34567

9

Requirements (1/2)

- In the previous examples:
  - The hash function mapped each x to a **unique** integer h(x)
  - There was no empty space in the table
- We used domain knowledge to design the hash function
- We want general purpose hash functions!

Requirements (2/2)

Desired properties:

- Easy and fast to compute
- Values evenly distributed
  - Within array size range

Hash function: Selecting digits

- h(001364825) = 35
  - Select the fourth and last digits
- Simple and fast
  - Does not evenly distribute items

12

Hash function: Folding

- Suppose the search key is a 9-digit ID.
- Sum-of-digits:
  h(001364825) = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5

  satisfies: 0 <= h(key) <= 81
- Grouping digits: 001 + 364 + 825 = 1190
  0 <= h(search key) <=3*999=2997

Modulo arithmetic

- Modulo arithmetic for Hash function

  $h(x) = x \bmod tableSize$

  tableSize is usually chosen as prime

Hash function: Converting Strings (1/4)

- First step: convert characters to integers (e.g. using ASCII values)
- Example: "NOTE"

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

E:69
N:78
O:79
T:84

Source: www.LookupTables.com

16

Hash function: Converting Strings (3/4)

- Hashing the sequence of integers:
  - Sum the values representing the characters
  - Write the numeric values in binary and concatenate.

    h("NOTE") = 100111 101000 101101 1000101

    $= 78 * 64^3 + 79 * 64^2 + 84 * 64^1 + 69 * 64^0$

    $= 20{,}776{,}261$
  - Using only 1 through 26 to the letters A through Z

    h("NOTE") = 01110 01111 10100 00101

    $= 14 * 32^3 + 15 * 32^2 + 20 * 32^1 + 5 * 32^0$

    $= 474{,}757$

Can now apply $x \bmod tableSize$

Hash function: Converting Strings (4/4)

$h(\text{“NOTE”}) = 14 * 32^3 + 15 * 32^2 + 20 * 32^1 + 5 * 32^0$

- Overflow can occur for long strings.
- Horner's Rule:
  - Hash function can be expressed as:

  $h(\text{“NOTE”})$

  $= 14 * 32^3 + 15 * 32^2 + 20 * 32^1 + 5 * 32^0$

  $= ((14 * 32 + 15) * 32 + 20) * 32 + 5$
- Prevent overflow by applying the modulo operation at each step

  If A = B (mod N), then for any C, A + C = B + C (mod N)

  If A = B (mod N), then for any D, AD = BD (mod N)

In practice it is better to use a prime number instead of 32

Outline

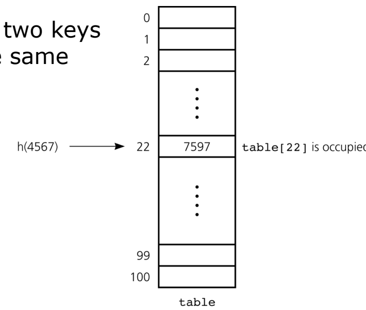
- Hashing
- Hash Functions
- **Resolving Collisions**
- Efficiency of Hashing
- Java Hashtable and HashMap

19

Collisions

Collision: two keys map to the same index

WHY?



The Birthday Problem (1/3)

- What is the minimum number of people so that the probability that at least two of them have the same birthday?
- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely

The Birthday Problem (2/3)

- p'_n – the probability that all people have different birthdays

$$p'_n = 1 \text{ x } (1 - 1/365) \text{ x } (1 - 2/365) \text{ x } \ldots \text{ x}(1 - (n-1)/365)$$

The event of at least two of the n persons having the same birthday:

$$p_n = 1 - p'_n$$

The Birthday Problem (3/3)

N (# of people)	P n (Probability that at least two of the n persons have the same birthday)
10	11.7 %
20	41.1 %
23	50.7 %
30	70.6 %
50	97. 0 %
57	99.0%
100	99.99997%
200	99.99999999999999999999999999998%
366	100%

Probability of Collision

- How many items do you need to have in a hash table so that the probability of collision is greater than ½?
- For a table of size 1,000,000 you only need 1178 items for this to happen!

Methods for Handling Collisions

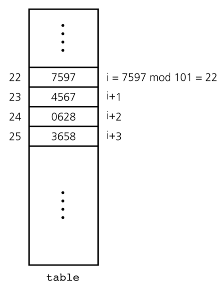
- Approach 1: Open addressing
  - probe for an empty slot in the hash table
- Approach 2: Restructuring the hash table
  - Change the structure of the array table

Approach 1: Open addressing

- A location in the hash table that is already occupied
  - Probe for some other empty, open, location in which to place the item.
  - Probe sequence
    - The sequence of locations that you examine

Open addressing: Linear Probing (1/3)

- If `table[h(key)]` is occupied check
  `h(key) + 1, h(key) + 2,…`
  until we find an available position
- Retrieval?
- Works until you need to delete.



Open addressing: Linear Probing (2/3)

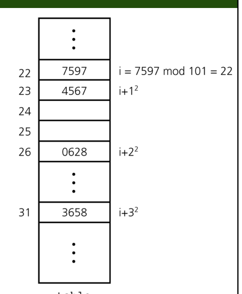
- Deletion: The empty positions created along a probe sequence could cause the retrieve method to stop, incorrectly indicating failure.
- Resolution: Each position can be in one of three states occupied, empty, or deleted. Retrieve then continue probing when encountering a deleted position. Insert into empty or deleted positions.

Open addressing: Linear Probing (3/3)

- Primary clustering: Items tend to cluster in the hash table.
- Large clusters tend to get larger.
- Decreases the efficiency of hashing.

Open Addressing: Quadratic Probing

- check
  `h(key) + 1², h(key) + 2², h(key) + 3²,…`
- Eliminates the primary clustering phenomenon
- Secondary clustering: two items that hash to the same location have the same probe sequence



Open Addressing: Double Hashing

Use two hash functions:

- h_1(key) – determines the position
- h_2(key) – determines the step size for probing
  - the secondary hash h_2 needs to satisfy:
    - $h_2(key) \neq 0$
    - $h_2 \neq h_1$ (why?)
- Rehashing
  Using more than one hash functions

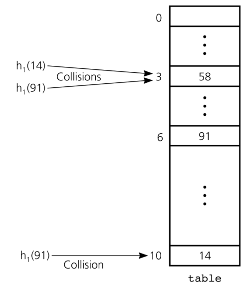
Double Hashing

Example:

$h_1(key) = key \bmod 11$

$h_2(key) = 7 - (key \bmod 7)$

Insert 58, 14, 91



Open Addressing: Increasing the size

- Increasing the size of the table: as the table fills the likelihood of a collision increases.
  - Cannot simply increase the size of the table – need to run the hash function again

Approach 2: Restructuring the Hash table

- Change the structure of the hash table to resolve collisions.
  - The hash table can accommodate more than one item in the same location

34

Restructuring: Buckets

- Each location `table[i]` is itself an array called a **bucket**.
  - Store items that hash into `table[i]` in this array.
- If the bucket size is too small?
  - Collisions will happen soon
- If the bucket size is too large?
  - Waste of storage

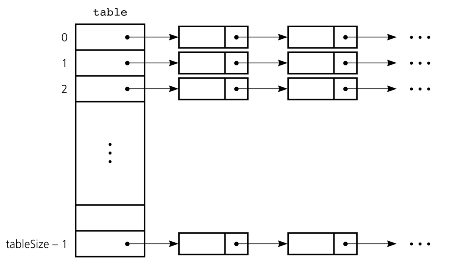
35

Restructuring: Separate Chaining(1/3)

- Separate chaining:
  - Design the hash table as **an array of linked lists.**

36

Restructuring: Separate Chaining(2/3)



Each location of the hash table contains a reference to a linked list

Restructuring: Separate Chaining(3/3)

- Does not need special care (`deleted`) for removal as open addressing does
- How do find, add and delete work?

Outline

- Hashing
- Hash Functions
- Resolving Collisions
- **Efficiency of Hashing**
- Java Hashtable and HashMap

39

The Efficiency of Hashing

- Consider a hash table with n items
  - Load factor $\alpha = n / tableSize$
  - n: current number of items in the table
  - tableSize: maximum size of array
  - α : a measure of how full the hash table is.
    - measures difficulty of finding empty slots
- Efficiency decreases as n increases

Size of Table

- Determining the size of Hash table
  - Estimate the largest possible n
  - Select the size of the table to get the load factor small.
  - Load factor should not exceed 2/3.

41

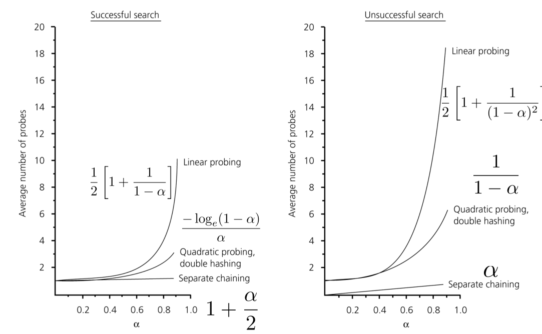
Hashing: Length of Probe Sequence

- Average number of comparisons that a search requires,
  - Linear Probing
    - successful $\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$
    - unsuccessful $\frac{1}{2}\left[1+\frac{1}{(1-\alpha^2)}\right]$
  - Quadratic Probing and Double Hashing
    - successful $\frac{-\log_e(1-\alpha)}{\alpha}$
    - unsuccessful $\frac{1}{1-\alpha}$

Hashing: Length of Probe Sequence

- Chaining
  - successful: 1 + α/2
  - unsuccessful: α
  - Note that α can be > 1

Comparison of Collision Resolution Methods



Good Hash Function?

- Easy and fast to compute
- Scatter the data evenly
  - Perfect hash function: Impractical to construct
  - Each chain should contain approximately the same number of items

45

How well does the HF scatter random data?

- Compare
  - h(x) = (first two digits of x) mod 40
  - h(x) = x mod 101

46

How well does the hash function scatter nonrandom data?

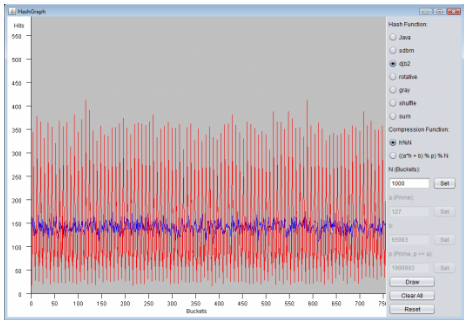
- Data can have patterns.
- Example
  - Table[0..99]
  - h(x) = first two digits of x
  - Emplyee IDs are according to department:
    - 10xxxxx Sales
    - 20xxxxx Customer Relations
    - 90xxxxx Data Processing
- Only 9 entries will be used (because all of the second digits are 0)
- Larger departments will have more crowded entries.

47

- Calculation of the hash function should involve the entire search key.
  - Comparing a modulo of the entire ID number is much safer than using only its first two digits.
- If a hash function uses modulo arithmetic, the base should be prime.
  - H(x) = x modulo tableSize
  - tableSize should be a prime number
  - This can avoid subtle types of patterns in the data.

48

Impact of using prime number



Traversal of Hash Tables

- If you need to traverse your tables by the sorted order of keys – hash tables may not be the appropriate data structure.

Outline

- Hashing
- Hash Functions
- Resolving Collisions
- Efficiency of Hashing
- **Java Hashtable and HashMap**

Hash Tables in Java

```
public class Hashtable<K,V> extends
  Dictionary<K,V> implements Map<K,V>

public class HashMap<K,V> extends
  AbstractMap<K,V> implements Map<K,V>

  public HashMap(int initialCapacity, float
  loadFactor)

  public HashMap(int initialCapacity)  //
  default loadFactor: 0.75
```

- HashMap is a newer implementation, and is the recommended one to use