HASH TABLES

Written by Tai Do

(Edited by Sarah Buchanan)

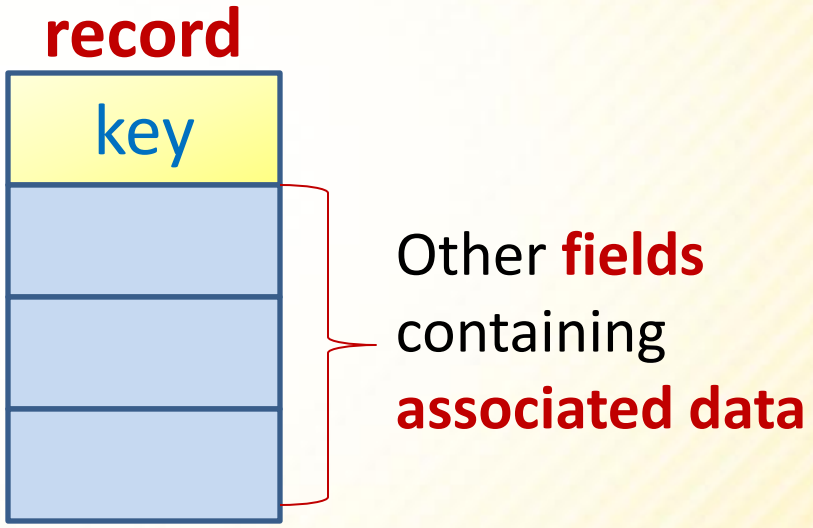
COP 3502

Outline

- Hash Table:
  - Motivation
  - Direct Access Table
  - Hash Table
- Solutions for Collision Problem:
  - Open Addressing:
    - Linear Probing
    - Quadratic Probing
    - Dynamic Table Expansion
  - Separate Chaining

Motivation

- We have to store some records and perform the following:
  - add new records
  - delete records
  - search a record by key



- Find a way to do these efficiently!

Record Example

pid (key)	name	score
0012345	andy	81.5
0033333	betty	90
0056789	david	56.8

...

9903030	tom	73
9908080	bill	49

...

Consider this problem. We want to store 1,000 student records and search them by student id.

Existing Data Structures

- Use an array to store the records, in unsorted order
  - add - add the records as the last entry, very fast *O(1)*
  - delete a target - slow at finding the target, fast at filling the hole (just take the last entry) *O(n)*
  - search - sequential search, slow *O(n)*
- Use an array to store the records, keeping them in sorted order
  - add - insert the record in proper position, much record movement, slow *O(n)*
  - delete a target - how to handle the hole after deletion? Much record movement, slow *O(n)*
  - search - binary search, fast *O(log n)*

Existing Data Structures

- Binary Search Tree:
  - add: insert the record in proper position, fast *O(logn)*
  - delete a target: fast *O(logn)*
  - search: fast *O(logn)*

Direct Access Table

	name	score
0		
:	:	:
12345	andy	81.5
:	:	:
33333	betty	90
:	:	:
56789	david	56.8
:	:	:
:	:	:
9908080	bill	49
:	:	:
9999999		

One way is to store the records in a huge array (index 0..9999999)
The index is used as the student id, i.e. the record of the student with pid 0012345 is stored at A[12345]

Direct Access Table

- Pros:
  - add- very fast *O(1)*
  - delete – very fast *O(1)*
  - search – very fast *O(1)*
- Cons:
  - Waste a lot of memory.
  - Use a table of TEN MILLION entries to store ONE THOUSAND records.

Hash Function

function **Hash**(key): integer;

Imagine that we have such a magic function **Hash**. It maps the key (sid) of the 1000 records into the integers 0..999, one to one. No two different keys maps to the same number.

H('0012345') = 134
H('0033333') = 67
H('0056789') = 764
…
H('9908080') = 3

hash code

Hash Table

To store a record, we compute Hash(pid) for the record and store it at the location Hash(pid) of the array.

To search for a student, we only need to peek at the location Hash(target sid).

H('0012345') = 134
H('0033333') = 67
H('0056789') = 764
…
H('9908080') = 3

0			
	:	:	:
3	9908080	bill	49
	:	:	:
67	0033333	betty	90
	:	:	:
134	0012345	andy	81.5
	:	:	:
764	0056789	david	56.8
	:	:	:
999	:	:	:

Hash Table with Perfect Hash

- Such magic function is called perfect hash
  - add – very fast *O(1)*
  - delete – very fast *O(1)*
  - search – very fast *O(1)*
- But it is generally difficult to design perfect hash. (e.g. when the potential key space is large)

Cost Summary

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted Array	log N	N	N	log N	N/2	N/2
Unsorted Array	N	1	N	N/2	1	N/2
Binary Search Tree	N	N	N	log N	log N	log N
Hash Table w/ Perfect Hash	1	1	1	1	1	1

Issues in hashing

- Each hash should generate a unique number. If two different items produce the same hash code we have a collision in the data structure. Then what?
- To deal with collisions, two issues must be addressed:
  1. Hash functions must minimize collisions (there are strategies to do this).
  2. When collisions do occur, we must know how to handle them.

Collision Resolution

- Focus on issue #2 (collision resolution):
  - Assume the following hash function is a reasonably good one:

h(k) = k%1000 (hash code = last 3 digits)

- Two ways to resolve collisions:
  - **Open Addressing**: every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key.
    - **Linear Probing**
    - **Quadratic Probing**
  - **Separate Chaining**: every hash table entry contains a pointer to a linked list of keys that hash to the same entry.

Open Addressing

- Store all keys in the hash table itself.
- Each slot contains either a key or NULL.
- To search for key *k*:
  - Compute *h(k)* and examine slot *h(k)*.

    Examining a slot is known as a ***probe***.
  - Case 1: If slot *h(k)* contains key *k*, the search is successful.

    Case 2: If this slot contains NULLL, the search is unsuccessful.
  - Case 3: There's a third possibility, slot *h(k)* contains a key that is not *k*.

    We compute the index of some other slot, based on *k* and on which probe (count from 0: 0th, 1st, 2nd, etc.) we're on. Keep probing until we either find key *k* (successful search) or we find a slot holding NULL (unsuccessful search).

How to compute probe sequences

- ***Linear probing:*** Given auxiliary hash function *h*, the probe sequence starts at slot *h(k)* and continues sequentially through the table, wrapping after slot *m* − 1 to slot 0. Given key *k* and probe number *i* ($0 \le i < m$),

  $h(k, i) = (h(k) + i) \bmod m$, m is the size of the table.

- ***Quadratic probing:*** As in linear probing, the probe sequence starts at *h(k)*. Unlike linear probing, it examines cells 1,4,9, and so on, away from the original probe point:

  $h(k, i) = (h(k) + i^2) \bmod m$

Open Addressing Example

- Three students:
  - <0000001, A, 81.3>
  - <0001001, B, 92.5>
  - <0002001, C, 99.0>
- Hash codes:
  - h(0000001) = 1%1000 = 1
  - h(0001001) = 1001%1000 = 1
  - h(0002001) = 2001%1000 = 1

Linear Probing: $h(k, i) = (h(k) + i) \bmod m$.

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.

0			
1	0000001	A	81.3
2	0001001	B	92.5
3	0002001	C	99.0
	…	…	…
999			

$h(k) = 1$

i	$h(k, i)$
0	1
1	2
2	3

Action	*# probe*
Store A	1
Store B	2
Store C	3

Linear Probing Example

Index	0	1	2	3	4	5	6	7	8	9
Value				173	281	**352**	461			

- Let's say the next value to store, 352, hashed to location 3.
- We see a value is stored there, so what do we do?
- Now, if we want to search for 352?
- When do we know that we can stop searching for a value with this method?
  - When we hit an empty location.

Linear Probing Example

- Linear Probing Example (Shown on the board)

Quadratic Probing: $h(k, i) = (h(k) + i^2) \bmod m$

- Quadratic probing eliminates the primary clustering problem of linear probing by examining certain cells away from the original probe point.

0			
1	0000001	A	81.3
2	0001001	B	92.5
3			
4			
5	0002001	C	99.0
	…	…	…
999			

h(k) = 1

i	*h(k, i)*
0	1
1	2
2	5

Action	*# probe*
Store A	1
Store B	2
Store C	3

Quadratic Probing Example

Index	0	1	2	3	4	5	6	7	8	9
Value				173	281		461	**352**		

- Let's say the next value to store, 352, hashed to location 3.
- We see a value is stored there, so what do we do?
- Now, if we want to search for 352?
- When do we know that we can stop searching for a value with this method?
  - When we hit an empty location.

An Issue with Quadratic Probing

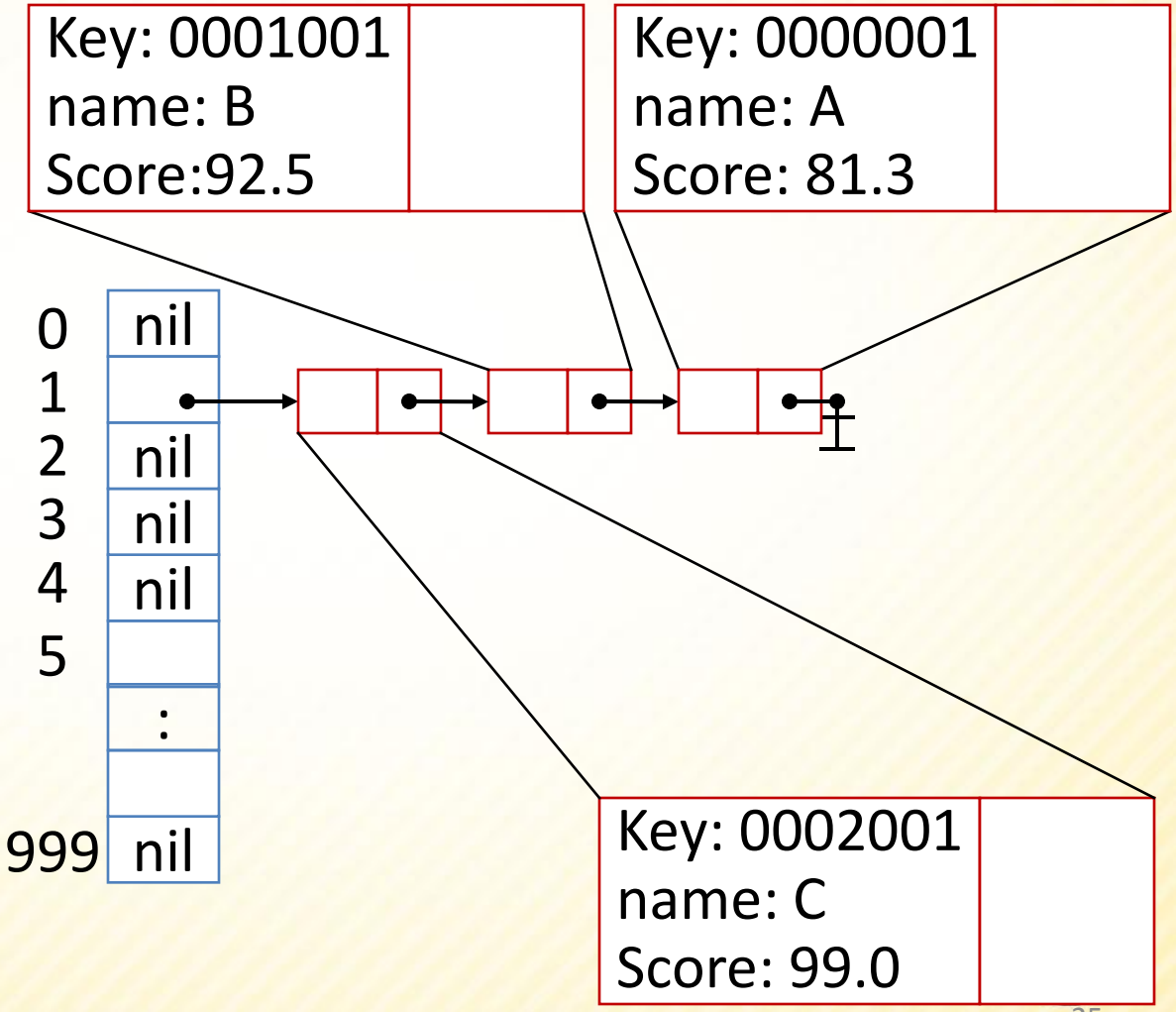
- For a hash table of size *m*, after *m* probes, all array elements should have been examined.
- This is true for Linear Probing, but NOT always true for Quadratic Probing (Why?)
- Insertion in Quadratic Probing: How do we know that eventually we will find a "free" location in the array, instead of looping around all filled locations?
  - if the table size is prime, AND the table is at least half empty, quadratic probing will always find an empty location.

Dynamic Table Expansion

- What if we don’t know how many records we'll have to store in a hash table before we set it up?
- Expand the hash table:
  1. Pick a prime number that is approximately twice as large as the current table size.
  2. Use this number to change the hash function.
  3. Rehash ALL the values already stored in the table.
  4. Now, hash the value to be stored in the table.

Separate Chaining

An array of linked lists. Insert new items to the front of the corresponding linked list.



Separate Chaining

- Good hash function, appropriate hash size:
  - Few collisions. Add, delete, search very fast *O(1)*
- Otherwise…
  - some hash value has a long list of collided records
  - add - just insert at the head fast *O(1)*
  - delete a target - delete from unsorted linked list slow *O(n)*
  - search - sequential search slow *O(n)*

Summary

- A data structure to support (add, delete, search) operations efficiently.
- Hash table includes an array and a hash function.
- Properties of a good hash function:
  - simple and quick to calculate
  - even distribution, avoid collision as much as possible
- Collision Resolution:
  - Open Addressing:
    - Linear Probing
    - Quadratic Probing
  - Separate Chaining

The Hash Function

- Designing a good hash function
- Let's say we were storing Strings, we want the hash function to map an arbitrary String to an integer in the range of the hash table array.

- Example:
  - F(w) = ascii value of the first character of w
- Why is this a poor choice?
  1) It's designed for an array of only size 26. (Or maybe bigger if we allow non-alphabetic)
  2) More words start with certain letters than others.

The Hash Function

- What if we used the following function:
  - $f(c_0c_1…c_n) = (ascii(c_0) + ascii(c_1) +…+ ascii(c_n))$
  - The problem is even if the table size is big, even 10,000, then the highest value an 8 letter string could hash to is 8*127 = 1016.
  - Then you would NOT use nearly 90% of the hash locations at all.
  - Resulting in many collisions.

The Hash Function

- Another idea in the book:
  - Each character has an ascii value less than 128. Thus, a string could be a representation of a number in base 128.
  - For example the string "dog" would hash to:
    - ascii('d')*1280 + ascii('o')1281 + ascii('g')*1282 =
    - 100*1 + 111*128 + 103*1282 = 1701860

- What are the problems with this technique?
  1) small strings map to HUGE integers
  2) Just computing this function may cause an overflow.

The Hash Function

- How can we deal with these problems?
  - Using the mod operator

$f(c_0c_1\ldots c_n) = (ascii(c_0)*128^0 + ascii(c_1)*128^1 +\ldots+ ascii(c_n)*128^n) \bmod tablesize$

  - We can still get an overflow if we mod at the end.
  - So we can use Horner's rule:
    - Specifies how to evaluate a polynomial without calculating xn in that polynomial directly:
    - $c_nx^n + c_{n-1}x^{n-1} + \ldots c_1x + c_0 = c_0 + x(c_1 + x(c_2 + \ldots + x(c_{n-1} + xc_n)\ldots))$

$(ascii(c_0)*128^0 + ascii(c_1)*128^1 +\ldots+ ascii(c_n)*128^n) =$

$ascii(c_0) + 128(ascii(c_1) + 128(ascii(c_2) + \ldots+(128(ascii(c_{n-1}) + 128\ ascii(c_n))\ldots))$

Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
64	40	100	@	@	96	60	140	`	`
65	41	101	A	A	97	61	141	a	a
66	42	102	B	B	98	62	142	b	b
67	43	103	C	C	99	63	143	c	c
68	44	104	D	D	100	64	144	d	d
69	45	105	E	E	101	65	145	e	e
70	46	106	F	F	102	66	146	f	f
71	47	107	G	G	103	67	147	g	g
72	48	110	H	H	104	68	150	h	h
73	49	111	I	I	105	69	151	i	i
74	4A	112	J	J	106	6A	152	j	j
75	4B	113	K	K	107	6B	153	k	k
76	4C	114	L	L	108	6C	154	l	l
77	4D	115	M	M	109	6D	155	m	m
78	4E	116	N	N	110	6E	156	n	n
79	4F	117	O	O	111	6F	157	o	o
80	50	120	P	P	112	70	160	p	p
81	51	121	Q	Q	113	71	161	q	q
82	52	122	R	R	114	72	162	r	r
83	53	123	S	S	115	73	163	s	s
84	54	124	T	T	116	74	164	t	t
85	55	125	U	U	117	75	165	u	u
86	56	126	V	V	118	76	166	v	v
87	57	127	W	W	119	77	167	w	w
88	58	130	X	X	120	78	170	x	x
89	59	131	Y	Y	121	79	171	y	y
90	5A	132	Z	Z	122	7A	172	z	z
91	5B	133	[	[	123	7B	173	{	{
92	5C	134	\	\	124	7C	174			|
93	5D	135	]	]	125	7D	175	}	}
94	5E	136	^	^	126	7E	176	~	~
95	5F	137	_	_	127	7F	177	&#127;	DEL

$$(ascii(c_0)*128^0 + ascii(c_1)*128^1 + \ldots + ascii(c_n)*128^n) =$$

$$ascii(c_0) + 128(ascii(c_1) + 128(ascii(c_2) + \ldots + (128(ascii(c_{n-1}) + 128\ ascii(c_n))\ldots))$$