Hash tables

Comp Sci 1575 Data Structures

Missouri S&T | Computer Science

Outline

Outline

Larger context

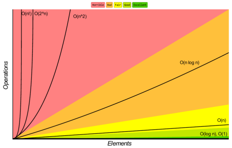
```
                              COLLECTION
                             /          \
UNORDERED_COLLECTION      ORDERED_COLLECTION
          |                     |          |
         BAG                  LINEAR     NON_LINEAR
          |                     |          |          |
         SET              LINKED_LIST    TREE   DIRECTED_GRAPH
          |                     |          |          |
      HASH_TABLE              DEQUE     BIN_TREE  UNDIRECTED_GRAPH
                             /    \      /    \
                         STACK  QUEUE  BST   HEAP
```

- **Hash tables** are un-ordered data structure which implements an **associative array** abstract data type, mapping keys to values.
- Use a hash function to compute an index into an array of buckets or slots, in which the value can be found.
- The second most common non-trivial data structure (besides the list)

Data structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Color key:



Hash tables: please don't do this!

Outline

Problems

How to store a large keyspace in a smaller structure?

- Which data type allows constant time access to store integers?
- With unlimited space, how can we design a very simple data structure to add, remove, and find integers in a data structure in constant time?
- How can we design a data structure with a max of 100 elements to store 50 random numbers between 25 and 100,000, e.g.,
  4123
  42
  99,999
  34,004
  ...

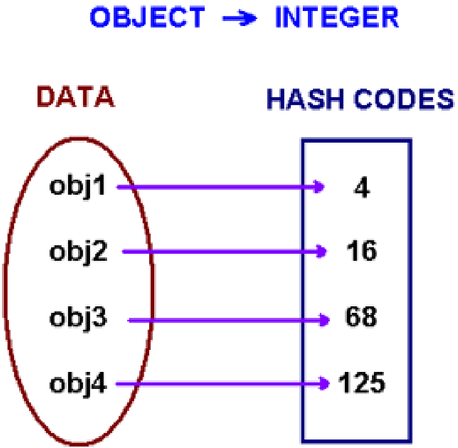
Problems

With unlimited space, how can we design a very simple data structure to add, remove, and find non-numeric keys to a data structure in constant time?

- How can we design a data structure with a max of 50 elements to store 20 random characters, e.g.,
  'H'
  'A'
  'S'
  'H'
  ...

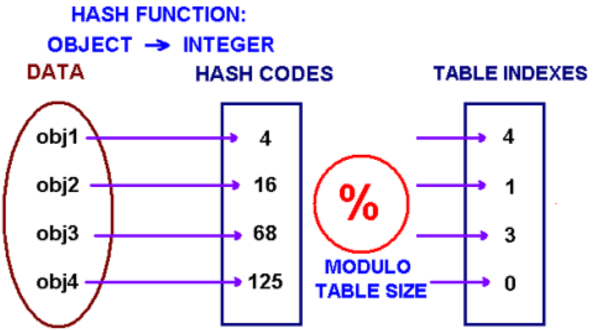
Outline

Hash functions map keys to hash codes



- Many types of hash function exist.

Hash codes of keys serve as indices

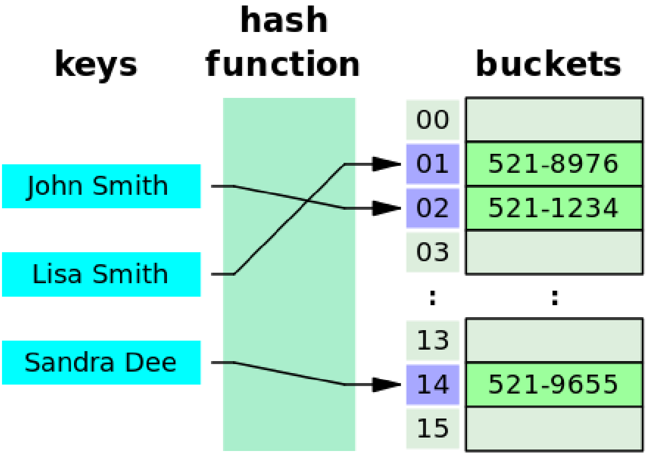


HASH TABLE

OBJ4	OBJ2		OBJ3	OBJ1
0	1	2	3	4

- % by table size squishes the codes into the array
- Process of finding a record by mapping its key value to a position in the array involves hashing.

Hash table dictionary



- Names (keys) map to hash codes which serve as indices
- Phone numbers (values) are merely data entries here
- Array that holds the records is called the hash table

Outline

What are hash tables good for?

- Good for: "What record, if any, has key value K?"
- Main advantage of hash tables over other table data structures is speed, especially with large dictionaries
- Hashing is not good for applications where multiple records with the same key value are permitted.
- Can't easily find ranges, the record with the minimum or maximum key value, or visit the records in key order

Outline

Hash functions

- **Problem**: Typically, there are many more values in the key range than there are slots in the hash table.
- **Solution**: A hash function can be used to deterministically map data of arbitrary size to data of fixed size.
- Hash function should be computable in constant time
- The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.
- Hash functions have much broader uses besides hash tables (e.g., A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or equivalent alternatives) by knowing the stored hash value.)

Simple example

Example: map set of 4-digit keys into a length 10 array

- simple hash function (h) could be:
  h(key) = key mod 10
  Alternative notation: $h(key) = key\%10$
- Keys: 9431, 9643, 3624, 9315, 6427

Index	0	1	2	3	4	5	6	7	8	9
Hash table		9431		9643	3624	9315		6427		

Mod is often the last step of hashing

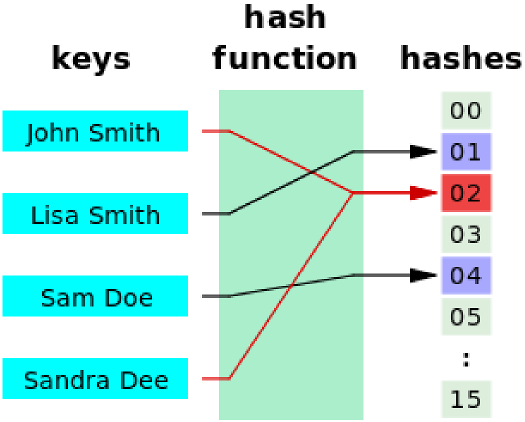
Hash function requirements

To use hash tables for these types of data, we must map these data types to w-bit hash codes. Hash code mappings should have the following properties:

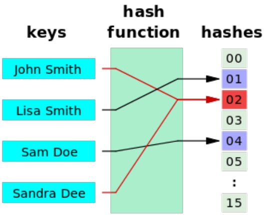
- If x and y are equal, then x.hashCode() and y.hashCode() are equal.
  The first property ensures that if we store x in a hash table and later look up a value y equal to x, then we will find x–as we should.
- If x and y are not equal, then the probability that x.hashCode() = y.hashCode() should be small (close to $1/2^{w}$).
  The second property minimizes the loss from converting our objects to integers. It ensures that unequal objects usually have different hash codes and so are likely to be stored at different locations in our hash table.

Outline

Collisions


keys
hash function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

Collisions



- Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key.
- Given a hash function h and two keys k_1 and k_2, if $h(k_1) = \beta = h(k_2)$ where β is a slot in the table, then we say that k_1 and k_2 have a collision at slot β under hash function h

Key distributions impact hash function design

People round numbers to 5:

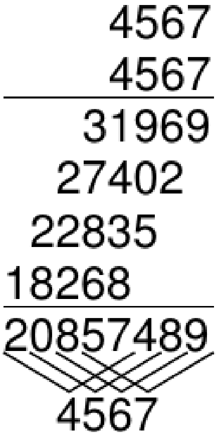
- simple hash function (h) could be: h(key) = key mod 10
- Keys: 9430, 96435 3620, 9315, 6425

Index	0	1	2	3	4	5	6	7	8	9
Hash table										

What happens here?

One fix for number hashing: mid-square method

Goal is to hash these key values to a table of size 100. Example Key value 4567.

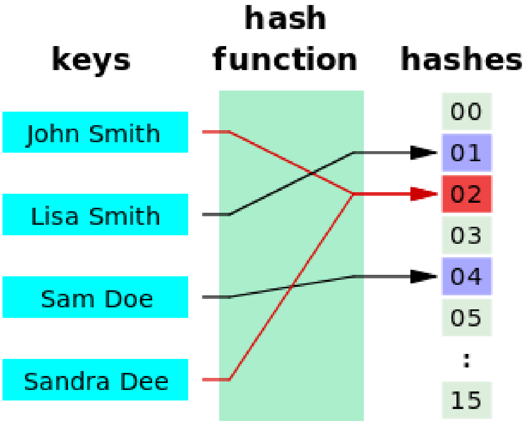


- Square the key value, and then take the middle r bits of the result, giving a value in the range 0 to $2^r - 1$.
- Most or all bits of the key value contribute to the result.
- Range (0-99) is equivalent to two digits in base 10. That is, r = 2.
- Middle two digits of square result are 57.
- Note: just one of many methods to accomplish the same goal

Outline

Hashing characters and strings

How might we hash the names below?



Hashing characters and strings

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Hashing characters and strings

```
int h(char *x)
{
    int i, sum;
    for(sum=0, i=0; x[i] != '\0'; i++)
        sum += (int)x[i];
    return sum % M;
}
```

Above function sums the ASCII values of the letters in a string.
Note: this is just one of many methods to do the same.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Data distribution

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

- Letter frequencies from corpus of English language text
- Frequency distributions can cause collisions for some hash functions
- Solutions?

Outline

Collision management

While one goal of a hash function is to minimize collisions, some collisions are unavoidable in practice.

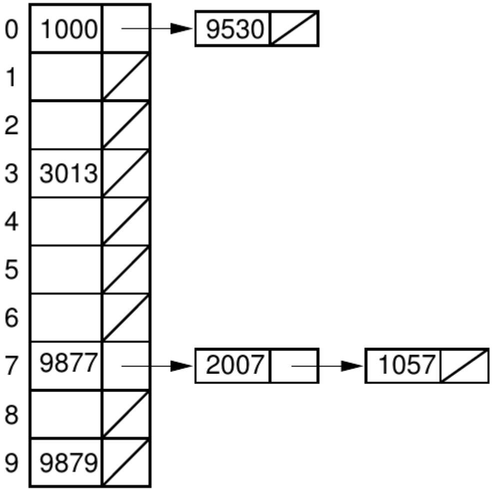
Collision resolution techniques can be broken into two primary classes:

1. **Open hashing** (also called separate chaining) : collisions result in storing one of the records outside the table
2. **Closed hashing** (also called open addressing) : collisions result in storing one of the records at another slot in the table

Hybrids are also possible

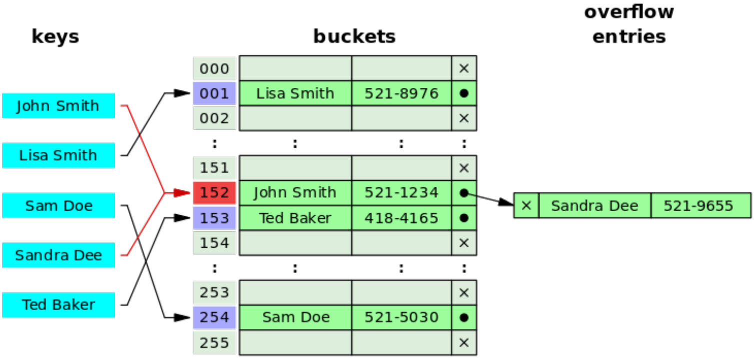
Outline

Open hashing



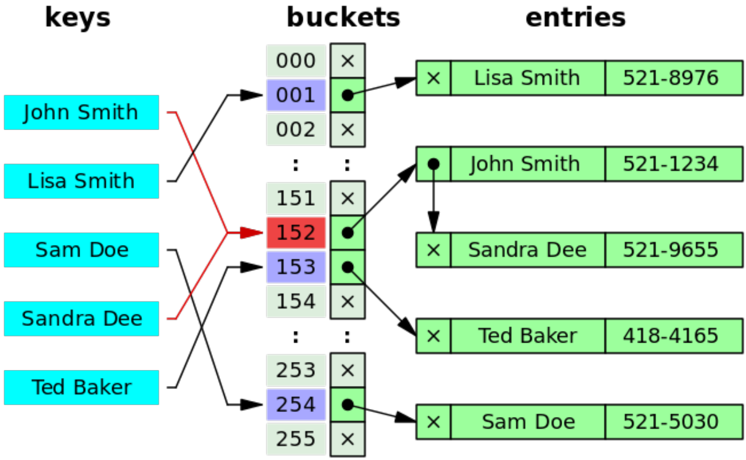
- The simplest form of open hashing defines each slot in the hash table to be the head of a linked list.

Open hashing



- Only the first item is stored in the array
- Collisions are stored in linked list

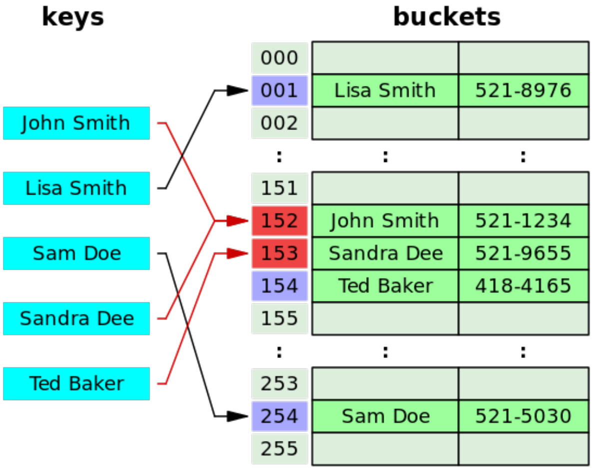
Open hashing



- Alternatively, all items can be stored externally

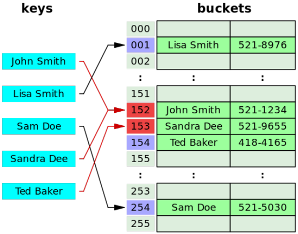
Outline

Closed hashing



- Closed hashing stores all records directly in the hash table

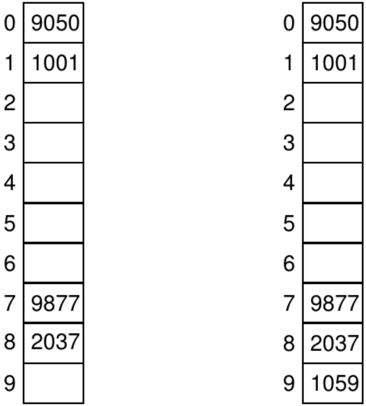
Closed hashing



- Each record R with key value k_R has a home position that is $h(k_R)$, the slot computed by the hash function
- If R is to be inserted and another record already occupies R's home position, collision policy systematically picks another index in the table

Linear probing

h(K) = K mod 10



(a) (b)

- First slot in the sequence will be the home position for the key.
- If the home position is occupied, then try the next slot in a pre-defined order, the probe sequence
- Probe sequence is generated by some function, p
- $pos = (home + p(k, i))\%M$ where p(K, i) = i

Linear probing


```
function find_slot(key)
  i = hash(key) mod num_slots
  // search until we either find the key,
  // or find an empty slot.
  while((slot[i] is full) and (slot[i].key != key))
      i = (i + 1) % num_slots
  return i
```

Alternative linear probe sequences

h(K) = K mod 10

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

(a)

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

(b)

$pos = (home + p(k, i))\%M$
where:

- $p(K, i) = ci$
- Will this visit all slots before returning back to home:
  for c=1?
  for c=2?
- Constant c must be relatively prime to M to generate a linear probing sequence that visits all slots in the table (c and M must share no factors).

Quadratic hashing

$pos = (home + p(k, i))\%M$
where:

- $p(K, i) = c_1 i^2 + c_2 i + c_3$
- Simple case: $p(K, i) = i^2$
- Draw?

Double hashing

$pos = (home + p(k, i))\%M$
where:

- h_2 is a second hash function and
  $p(K, i) = i * h_2(K)$
- Can be combined with other methods like pseudo-random or quadratic, e.g.,
  $p(K, i) = i^2 * h_2(K)$
- Draw?

Robin Hood hashing

- During double hashing, a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position.
- Reduces worst case search times in the table.
- What else do we need to store?
- Draw?

Outline

Search

Finding a record with key value K in a database organized by hashing follows a two-step procedure:

1. Compute the table location h(K).
2. Starting with slot h(K), locate the record containing key K using (if necessary) a collision resolution policy.

How to search with collision resolution?

h(K) = K mod 10

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

$index = (home + p(k, i))\%M$

- What should search do if looking for 9877?
- What should search do if looking for 2037?

Outline

How to delete?

$h(K) = K \bmod 10$

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- What is the process to:
  Delete only 9877?
  Delete only 2037?
  Delete 9877 then 2037?
- What is a general solution?

How to delete?

$h(K) = K \bmod 10$

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- If 9877 is deleted from the table, a search for 2037 must still pass through Slot 7 as it probes to slot 8

Delete: set flag or tombstone (aka lazy delete)

h(K) = K mod 10

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- Indicates that a record once occupied the slot but does so no longer.
- During search, if a tombstone is encountered during a probe sequence, search continues.
- When does search end?
- What about during insertion?
- Problems?

Delete: set flag or tombstone (aka lazy delete)

h(K) = K mod 10

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- During insertion, tombstone slots can be used to store the new record.
- To avoid inserting duplicate keys, follow the probe sequence until a truly empty position has been found, to verify that a duplicate is not in the table.
- Problems?

Delete: set flag or tombstone (aka lazy delete)

h(K) = K mod 10

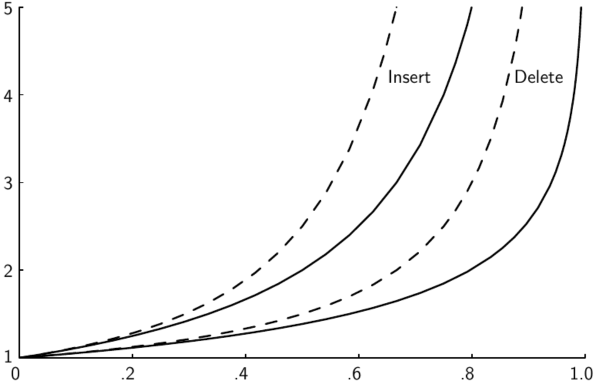
0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- Degrades HT over time: as number of delete/insert operations increases, cost of a successful search increases.
- Fix 1: During later search, when found, an element can be relocated to the first location marked for deletion that was probed during the search.
- Fix 2: Periodically rehash by reinserting all records into a new hash table.

Outline

Load factor: cost goes up for full tables

$\alpha = N/M$ where
N is the number of records currently in the table and
M is the size of the hash table

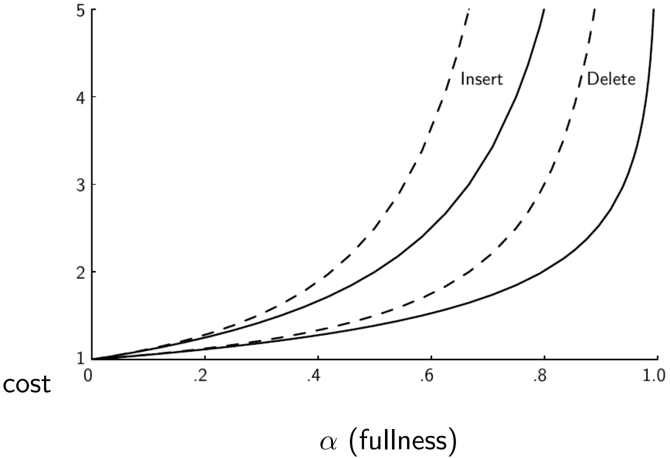


cost

α (fullness)

Is there a solution to this slowdown?

Load factor: re-hashing



Build another table that is about twice as big (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each element and inserting it in the new table.

Outline

Asymptotic comparison of dictionary DS options

ADT	Lookup		Insertion		Deletion		Ordered
	Average	Worst	Average	Worst	Average	Worst	
Sequential container: key-value pairs	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	No
Sequential container: key-value pairs	O(log n)	O(n)	O(1)	O(1)	O(n)	O(n)	Yes
Hash table	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	No
Self-balancing binary search tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Unbalanced binary search tree	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)	Yes

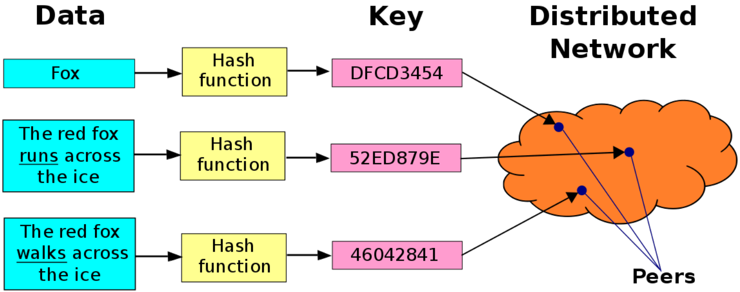
- Reminder: BST is also a decent data structure for a dictionary.
- How does the BST compare in the average and worst cases?

Outline

Problem: distributed database

- To download a file from someone, knowing their IP address is one way to initiate a peer to peer (p2p) connection.
- How to store a database of pairings between IP addresses and torrents without a central server?
- Keys could be content names (e.g., names of books and software), and the value could be the IP address at which the content is stored; in this case, an example key-value pair is the tuple:
  (ComputerNetworkingEssentials.pdf, 128.17.123.38).
  Ask: Which is the key and which is the value?
- Building such a database is straightforward with a client-server architecture that stores all the (key, value) pairs in one central server. You just ask the central server (at its known IP) for the IP of the people with your file of interest.

DHT


Data
Key
Distributed Network
Fox
Hash function
DFCD3454
The red fox runs across the ice
Hash function
52ED879E
The red fox walks across the ice
Hash function
46042841
Peers

Solution: distributed hash table (DHT)

- n users
- Each user identifier is an integer in the range $[0,\ 2^n - 1]$
- Hash the key (author/book name) into a number, mod $2^n - 1$
- The user that has the closest value after the hashed key stores the item

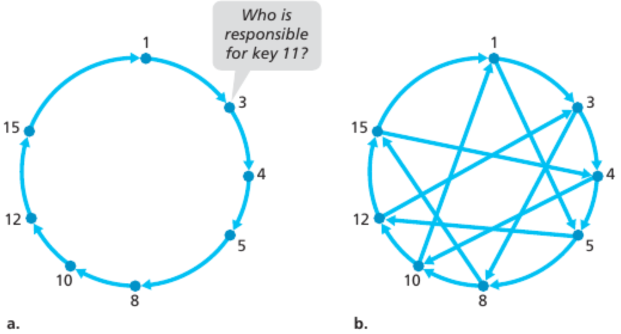
Problem

- How to lookup which user is storing a particular hashed key?
- How to find the other user who "knows" about that user?
- Should we store the location of all "neighbors"?

Solution: circular DHT

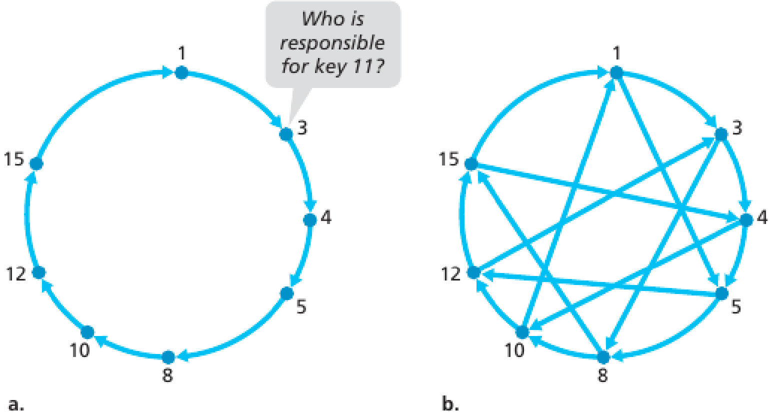
The user that has the closest value after the hashed key stores the item.
Each user stores the IP of users with immediately larger keys.
“Join” and “Leave” protocols are needed.

(a) Only index forward neighbors; number of messages is n/2



(b) Storing indices of more neighbors increases messaging efficiency, and increases storage overhead

A balance of connections: space versus time



DHT can be designed so that both the number of neighbors per peer as well as the average number of messages per query is $O(\log N)$, where N is the number of peers.