

# Hashing

Chapter 5 of textbook 1

## Simple problem

Let's try a simpler problem

- How do I store your examination grades so that I can access (or find) your grades in  $\Theta(1)$  time?

Recall that each student is issued an 8-digit number

- How do I store your examination grades so that I can access your grades in  $\Theta(1)$  time?
- Suppose student An has the student ID 10421118
- I can't create an array of size  $10^8 \approx 1.5 \times 2^{26}$

## Simple problem

Suppose that I could create an array of size 1000

- How could you convert an 8-digit number into a 3-digit number?
- First three digits might cause a problem: almost all students start with 104. [CS Student ID](#)
- The last three digits, however, seems to be ok.

Therefore, I could store Jane's examination grade into an array with size 1000:

```
grade[118] = 86;
```

## Simple problem

Consequently, I have a function that maps a student  $\vdots$   $\vdots$   
onto a 3-digit number

- I can store something in that location
- Storing it, accessing it, and erasing it is  $\Theta(1)$
- The table is called hash table
- Problem: two or more students may map to the same number:
  - An has ID 10421118 and scored 85
  - Vy has ID 10465118 and scored 87

454	
455	
456	86
457	
458	
459	
460	
461	
462	
463	79
464	
465	

$\vdots$   $\vdots$

## The hashing problem

The process of mapping an object (or a number) onto an integer in a given range is called *hashing*

Problem: multiple objects may hash to the same value

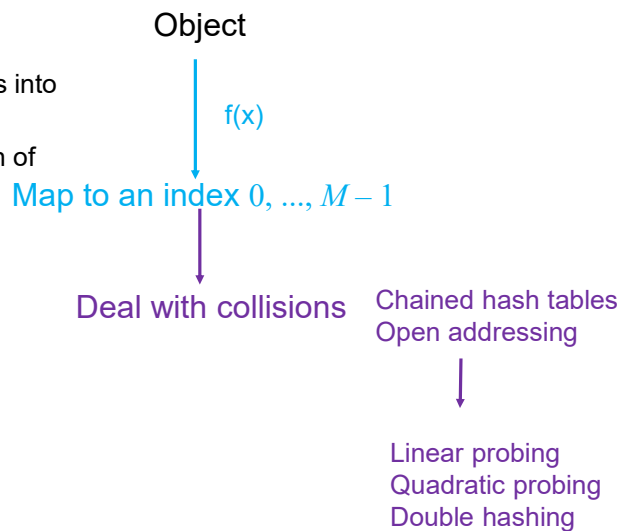
- Such an event is termed a *collision*

Hash tables use a hash function together with a mechanism for dealing with collisions

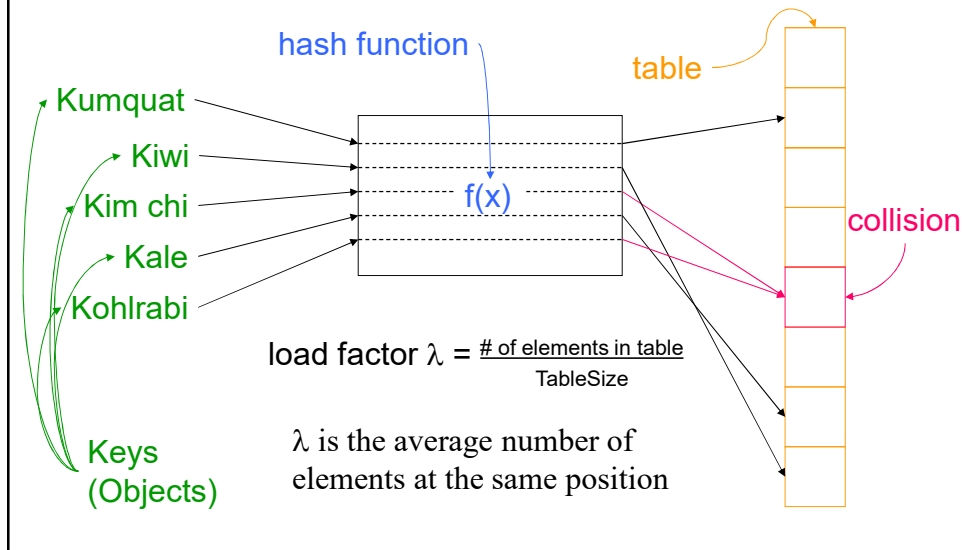
## The hash process

We will break the process into two **independent** steps:

- We will try to get each of these down to  $\Theta(1)$



## Hash Table Terminology



## The hash function

Assume that the hash table  $H$  has size  $M$

There is a hash function  $f(x)$  which maps an object (or a key  $k$ ) to a value  $p$  in  $0, \dots, M-1$ , and the element is placed in position  $p$  in the hash table. In the hash function we usually have two small steps:

- Step1: convert an object to an integer
- Step2: map this integer to a value  $p$  in range  $[0, M-1]$

## An example of Hash function for strings

```

Index
Hash( const char *Key, int TableSize )
{
    unsigned int HashVal = 0;

    /* 1*/ while( *Key != '\0' )
    /* 2*/     HashVal += *Key++;

    /* 3*/ return HashVal % TableSize;
}

```

Step 1 points to the while loop condition.

Step 2 points to the HashVal += \*Key++; line.

Figure 5.3 A simple hash function

This Hash function is not good:

- Slow running time ( $O(n)$ )
- TableSize is 10000 and Index in range from 0 to 1016 ( $127 \times 8$ ) => poor distribution.

## Another example of hash function

Figure 5.4 Another possible hash function—  
not too good

```

Index
Hash( const char *Key, int TableSize )
{
    return ( Key[ 0 ] + 27 * Key[ 1 ] + 729 * Key[ 2 ] )
           % TableSize;
}

```

We use first three characters of a string for hashing. Choosing a number  $r=27$  (number of English letters), hash value for “abcde” is:

$$\text{ASCII}(c) \cdot r^2 + \text{ASCII}(b) \cdot r + \text{ASCII}(a)$$

This function is better but not too good: poor distribution. There are  $27^3$  possible combinations of three characters but actually we only have 2851 real combinations.

## What is a good hash function?

Necessary properties of such a hash function  $h$  are:

- Should be fast: ideally  $\Theta(1)$
- The hash value must be *deterministic*
  - It must always return the same integer each time
- Equal objects hash to equal values
  - $x = y \Rightarrow h(x) = h(y)$
- Using the whole hash table (for all  $0 \leq k < \text{size}$ , there's an  $i$  such that  $\text{hash}(i) \% \text{size} = k$ )
- A good distribution on hash table.

## A good hash function

```
Index Hash(char Key[], int TableSize ) {
    unsigned int hash_value = 0;

    for(int k = 1; k <= strlen(Key); k*= 2)
        hash_value = (hash_value << 5) + key[k - 1];

    return hash_value % TableSize;
}
```

<< : Bitwise operator

- Don't use all characters: ex: **A**\_E**l**ber**e**th Gil**t**h**o**niel
  - Running time:  $O(\ln(n))$
- Expected to distribute well

## How to Design a Hash Function

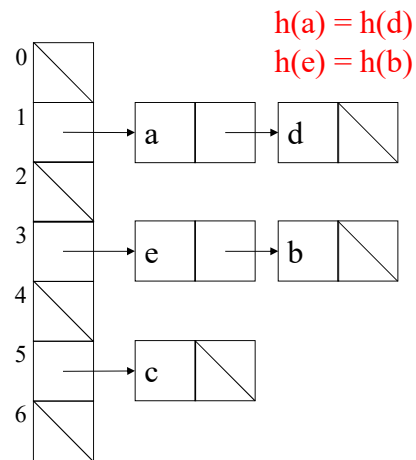
- Know what your keys are
- Study how your keys are distributed
- Try to include all important information in a key in the construction of its hash
- `c t b t` Prune the features used to create the hash until it runs “fast enough” (very application dependent)

## Collisions

- *Pigeonhole principle* says we can't avoid all collisions
  - try to hash without collision  $m$  keys into  $n$  slots with  $m > n$
  - try to put 6 pigeons into 5 holes
- What do we do when two keys hash to the same entry?
  - open hashing: put a linked list in each entry
  - closed hashing: pick a next entry to try
  - Double hashing: use two hash functions

## Open Hashing or Separate Chaining

- Put a linked list at each entry
  - choose type as appropriate
  - common case is unordered linked list (chain)
- Properties
  - $\lambda$  can be greater than 1
  - performance degrades with length of chains



## Another example of Separate Chaining

Hash function  $f(X) = X \bmod 10$

H is a hash table

To find an element  $j$ , compute  $f(j)$ . Let  $f(j) = k$ . Then search in link list  $H[k]$

To insert an element  $j$ , compute  $h(j)$ . Let  $h(j) = k$ . Then insert in link list  $H[k]$

To delete an element, delete from the link list.

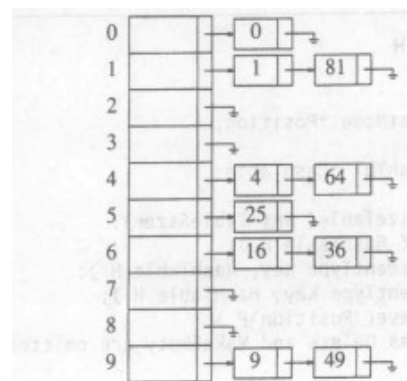


Figure 5.6 A separate chaining hash table



## Another example of Separate Chaining

Search for 7, look in position 7 in the array, find it empty, conclude 7 not there

Search for 13, look in position 3 in the array, search the link list, 13 not found, conclude that 13 is not there

Insert 14, add it at the head of the link list starting at position 4

## Run time Analysis

Insertion is  $O(1)$ .

Successful Search requires  $1 + (\lambda/2)$  on average.

Unsuccessful Search requires  $\lambda$  on average

Worst case searching complexity depends on the maximum length of a list  $H[p] \Rightarrow O(q)$  if  $q$  is the maximum length.

Want  $\lambda$  to be approximately 1. To reduce worst case complexity we choose hash functions which distribute the elements evenly in the list.

## Quiz

We will store strings and the hash value of a string will be the last 3 bits of the first character in the host name

Ex: The hash value of "optimal" is based on "o" => 111

0	→ 0	a	01100001	n	01101110
1	→ 0	b	01100010	o	01101111
2	→ 0	c	01100011	p	01110000
3	→ 0	d	01100100	q	01110001
4	→ 0	e	01100101	r	01110010
5	→ 0	f	01100110	s	01110011
6	→ 0	g	01100111	t	01110100
7	→ 0	h	01101000	u	01110101
		i	01101001	v	01110110
		j	01101010	w	01110111
		k	01101011	x	01111000
		l	01101100	y	01111001
		m	01101101	z	01111010

Using the above hash function to enter the following strings into the hash table: optimal, cheetah, wellington, augustin, lowpower, ashok, vlach, ims, jab, cad.

Please calculate the load factor 10 / 7

## Implementation of Separate Chaining

```
/* Place in the implementation file */
struct ListNode
{
    ElementType Element;
    Position Next;
};

typedef Position List;

/* List *TheList will be an array of lists, allocated later */
/* The lists use headers (for simplicity), */
/* though this wastes space */
struct HashTbl
{
    int TableSize;
    List *TheLists;
};
```

Basic Operations:

InitializeTable  
Search  
Insert  
**Delete**

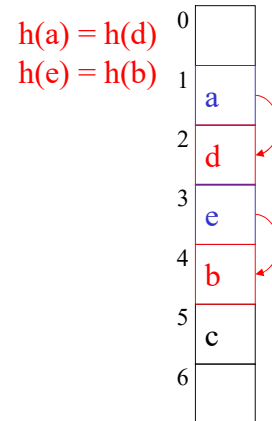
## Closed Hashing / Open Addressing

What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

- Properties

- $\lambda \leq 1$ 
  - Number of elements  $\leq$  TableSize
- performance degrades with difficulty of finding right spot



## Probing

- Probing how to:
  - First probe - given a key  $k$ , hash to  $h(k)$
  - Second probe - if  $h(k)$  is occupied, try  $h(k) + f(1)$
  - Third probe - if  $h(k) + f(1)$  is occupied, try  $h(k) + f(2)$
  - And so forth
- Probing properties
  - we force  $f(0) = 0$
  - the  $i^{\text{th}}$  probe is to  $(h(k) + f(i)) \bmod \text{size}$
  - if  $i$  reaches size - 1, the probe has failed
  - depending on  $f()$ , the probe may fail sooner
  - long sequences of probes are costly!

# Linear Probing

- Probe sequence is
  - $h(k) \bmod \text{size}$
  - $h(k) + 1 \bmod \text{size}$
  - $h(k) + 2 \bmod \text{size}$
  - ...

$$f(i) = i$$

## Linear Probing Example

	insert(76) $76\%7 = 6$	insert(93) $93\%7 = 2$	insert(40) $40\%7 = 5$	insert(47) $47\%7 = 5$	insert(10) $10\%7 = 3$	insert(55) $55\%7 = 6$
0				47	47	47
1						55
2		93	93	93	93	93
3					10	10
4						
5			40	40	40	40
6	76	76	76	76	76	76
Number of probes	1	1	1	3	1	3

## Quiz

Insert 3-digit hexadecimal number into this initially empty hash table:

19A, 207, 3AD, 488, 5BA, 680, 74C, 826, 946, ACD, B32, C8B,  
DBE, E9C

With the least-significant digit is the primary hash function

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	DBE	B32	E9C			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

## Searching

Start at the appropriate position, and  
searching forward until

1. The item is found,
2. An empty item is found, or
3. We have traversed the entire array

The third case will only occur if the hash  
table is full (load factor of 1)

# Searching

Searching for C8B

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Searching

Searching for C8**B**

- Examine bins B, C, D, E, F
- The value is found in Bin F

0	1	2	3	4	5	6	7	8	9	A	<b>B</b>	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	<b>C8B</b>

# Searching

Searching for 23E

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Searching

Searching for 23E

- Search bins E, F, 0, 1, 2, 3, 4
- The last bin is empty; therefore, 23E is not in the table

0	1	2	3	4	5	6	7	8	9	A	B	C	D	<b>E</b>	F
680	D59	B32	E93	×		826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

## Erasing

We cannot simply remove elements from the hash table

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

## Erasing

We cannot simply remove elements from the hash table  
– For example, consider erasing 3AD

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B



## Erasing

We cannot simply remove elements from the hash table

- For example, consider erasing 3AD
- If we just erase it, it is now an empty bin
  - By our algorithm, we cannot find ACD, C8B and D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C		ACD	C8B

## Erasing methods

Two methods:

- Real deletion
- Lazy deletion

Now focusing on real deletion: we must attempt to fill the empty element

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C		ACD	C8B

## Erasing

We must attempt to fill the empty bin  
– We can move ACD into the location

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	ACD	C8B

## Erasing

Now we have another empty element to fill

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD		C8B

## Erasing

Now we have another empty element to fill  
 – We can move ACD into the location

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	<del>C8B</del>	C8B

## Erasing

Now we must attempt to fill the empty  
 element at F  
 – We cannot move 680


0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	

## Erasing

Now we must attempt to fill the empty element at F

- We cannot move 680
- We can, however, move D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	<b>D59</b>



## Erasing

At this point, we cannot move B32 or E93 and the next bin is empty

- We are finished

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	D59

## Erasing

Suppose we delete 207

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	D59

## Erasing

Suppose we delete 207

– Cannot move 488

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826		488	946	19A	5BA	74C	ACD	C8B	D59

## Erasing

Suppose we delete 207

– We could move 946 into Bin 7

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488	946	19A	5BA	74C	ACD	C8B	D59

## Erasing

Suppose we delete 207

– We cannot move either the next five entries

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488		19A	5BA	74C	ACD	C8B	D59

## Erasing

Suppose we delete 207

- We cannot move either the next five entries

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488	<del>D59</del>	<del>19A</del>	<del>5BA</del>	<del>74C</del>	<del>ACD</del>	<del>C8B</del>	D59

## Erasing

Suppose we delete 207

- We cannot fill this bin with 680, and the next bin is empty
- We are finished

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488	D59	19A	5BA	74C	ACD	C8B	

## Quiz

Using the last digit as our hash function—insert these nine numbers into a hash table of size  $M = 10$

31, 15, 79, 55, 42, 99, 60, 80, 23

Then, remove 79, 31, 42, and 60, in that order

## Primary clustering

- Primary Clustering is when different keys collide to form one big group.

47	6	14	8		40	76
----	---	----	---	--	----	----

- Think of this as “clusters of many colors”. Even though these keys are all different, they end up in a giant cluster.
- Primary clustering reduces the performance

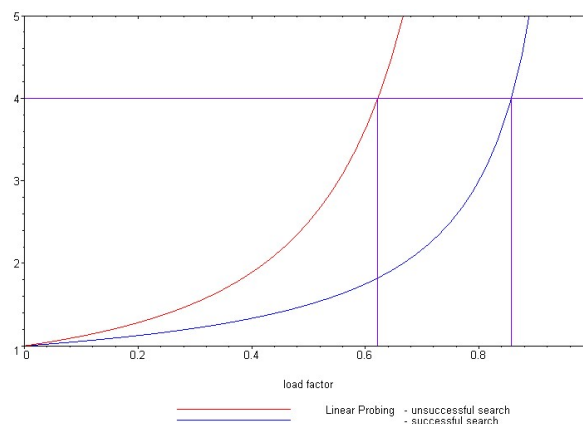


## Load Factor in Linear Probing

- For *any*  $\lambda < 1$ , linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search:  $\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right)$
  - unsuccessful search:  $\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right)$
- Linear probing suffers from *primary clustering*
- Performance quickly degrades for  $\lambda > 1/2$

## Run-time analysis

The following plot shows how the number of required probes increases



## Run-time analysis

Our goal was to keep all operations  $\Theta(1)$   
Unfortunate, as  $\lambda$  grows, so does the run time

One solution is to keep the load factor under a given bound

If we choose  $\lambda = 2/3$ , then the number of probes for either a successful or unsuccessful search is 2 and 5, respectively

## Linear Probing Summary

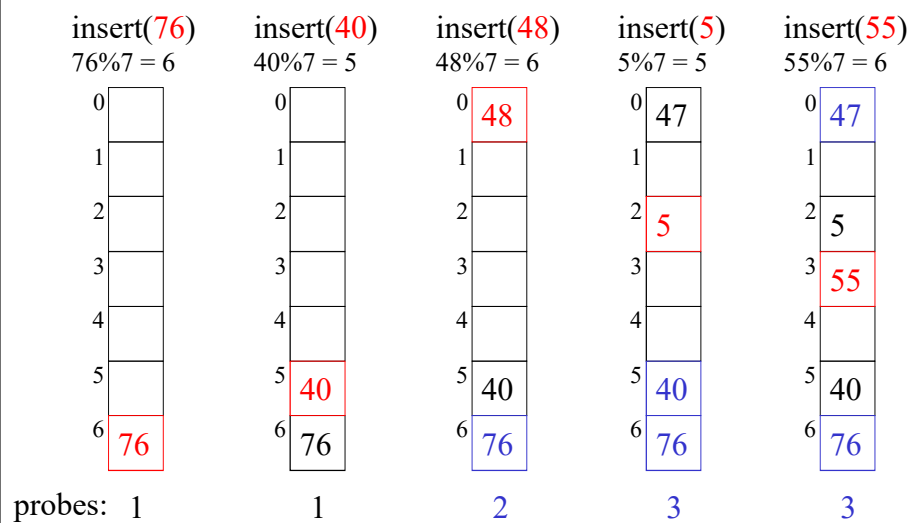
- Continue looking forward until an empty cell is found
- Searching follows the same rule
- Removing an object is more difficult
- Primary clustering is an issue
- Keep the load factor  $\lambda \leq 2/3$

## Quadratic Probing

- Probe sequence is
  - $h(k) \bmod \text{size}$
  - $(h(k) + 1) \bmod \text{size}$
  - $(h(k) + 4) \bmod \text{size}$
  - $(h(k) + 9) \bmod \text{size}$
  - ...
- We try to spread elements far away from the hash position of keys
- Quadratic probing gets rid of primary clustering.

$$f(i) = i^2$$

## Quadratic Probing Example



## Quadratic Probing Example

insert( <b>76</b> ) $76\%7 = 6$	insert( <b>93</b> ) $93\%7 = 2$	insert( <b>40</b> ) $40\%7 = 5$	insert( <b>35</b> ) $35\%7 = 0$	insert( <b>47</b> ) $47\%7 = 5$
0 1 2 3 4 5 6 <b>76</b>	0 1 2 <b>93</b> 3 4 5 6 76	0 1 2 93 3 4 5 <b>40</b> 6 76	0 <b>35</b> 1 2 93 3 4 5 40 6 76	0 <b>35</b> 1 2 <b>93</b> 3 4 5 <b>40</b> 6 <b>76</b>
probes: 1	1	1	1	$\infty$

## Quadratic Probing Succeeds (for $\lambda \leq \frac{1}{2}$ )

- If table size is prime and  $\lambda \leq \frac{1}{2}$ , then quadratic probing always find an empty slot for a new element

## Quiz

Insert these numbers into this initially empty hash table

9A, 07, AD, 88, BA, 80, 4C, 26, 46, C9, 32, 7A, BF, 9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80		32	BF		9C	26	07	88	C9	9A	BA	4C	AD	7A	46

## Erase

We will use the concept of *lazy deletion*

– Mark an element as ERASED; not really delete

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	21		43			76				9A					50

### Implementation

– Storing three states for each element

UNOCCUPIED,  
OCCUPIED,  
ERASED

## Erase

If we erase AD, we must mark that element as erased

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	<del>AD</del>		C9

## Search

When searching, it is necessary to skip over this bin

– For example, find AD: D, E

find 5C: C, D, F, 2, 5, 9, F, 6, E

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	<del>AD</del>		C9

## Modified insertion

We must modify insert, as we may place new items into either

- Unoccupied items
- Erased items

## Multiple insertions and erases

One problem which may occur after multiple insertions and removals is that numerous elements may be marked as ERASED

- In calculating the load factor, an ERASED element is equivalent to an OCCUPIED element

This will increase our run times...

## Multiple insertions and erases

If the load factor  $\lambda$  grows too large, we have two choices:

- If the load factor due to occupied bins is too large, double the table size
- Otherwise, rehash all of the objects currently in the hash table

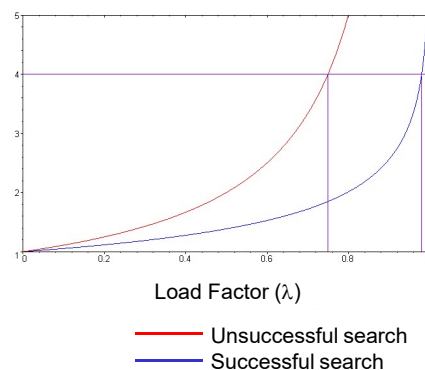
## Expected number of probes

It is possible to calculate the expected number of probes for quadratic probing, again, based on the load factor:

- Successful searches:  $\frac{\ln\left(\frac{1}{1-\lambda}\right)}{\lambda}$
- Unsuccessful searches:  $\frac{1}{1-\lambda}$

When  $\lambda = 2/3$ , we require  
1.65 and 3 probes, respectively

- Linear probing required  
3 and 5 probes, respectively



Reference: Knuth, The Art of Computer Programming, Vol. 3, 2<sup>nd</sup> Ed., 1998, Addison Wesley, p. 530.



## Quadratic probing versus linear probing

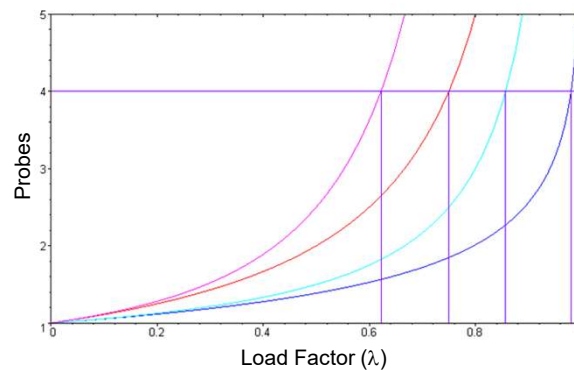
Comparing the two:

### Linear probing

Unsuccessful search ———  
Successful search ———

### Quadratic probing

Unsuccessful search ———  
Successful search ———



## Quadratic probing summary

- An open addressing technique
- Steps forward by a quadratically growing steps
- Insertions and searching are straight forward
- Removing objects is more complicated: use lazy deletion
- Still subject to secondary clustering

## Double Hashing

- Probe sequence is
    - $h_1(k) \bmod \text{size}$
    - $(h_1(k) + 1 \cdot h_2(k)) \bmod \text{size}$
    - $(h_1(k) + 2 \cdot h_2(k)) \bmod \text{size}$
    - ...
- $f(i) = i \cdot \text{hash}_2(k)$

## A Good Double Hash Function...

- ...is quick to evaluate.
  - ...differs from the original hash function.
  - ...never evaluates to 0 (mod size).
- Ex:  $X \bmod 9$  is useless

One good choice is to choose prime  $R < \text{size}$  and:

$$\text{hash}_2(x) = R - (x \bmod R)$$

## Double Hashing Example (R=5)

insert(76) $76\%7 = 6$	insert(93) $93\%7 = 2$	insert(40) $40\%7 = 5$	insert(47) $47\%7 = 5$ $5 - (47\%5) = 3$	insert(10) $10\%7 = 3$	insert(55) $55\%7 = 6$ $5 - (55\%5) = 5$
0 1 2 3 4 5 6	0 1 2 3 4 5 6	0 1 2 3 4 5 6	0 1 2 3 4 5 6	0 1 2 3 4 5 6	0 1 2 3 4 5 6
			47	47	47
	93	93	93	93	93
				10	10
					55
		40	40	40	40
76	76	76	76	76	76
probes: 1	1	1	2	1	2

## Load Factor in Double Hashing

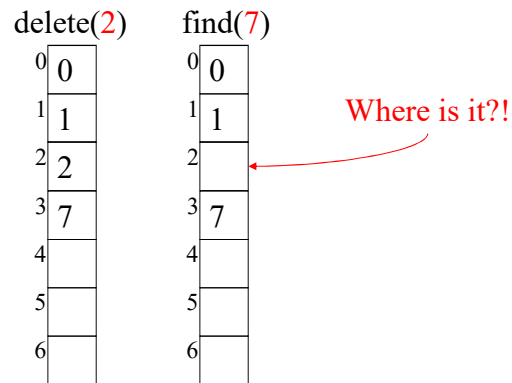
- For *any*  $\lambda < 1$ , double hashing will find an empty slot (given appropriate table size and hash<sub>2</sub>)
- Search cost appears to approach optimal (random hash):

– successful search:  $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$

– unsuccessful search:  $\frac{1}{1-\lambda}$

- No primary clustering
- One extra hash calculation

## Deletion in Double Hashing



- Use lazy deletion

## The Squished Pigeon Principle

- An insert using **closed hashing** *cannot* work with a load factor of 1 or more.
- An insert using **closed hashing** with quadratic probing may not work with a load factor of  $\frac{1}{2}$  or more.
- Whether you use open or closed hashing, large load factors lead to poor performance!
- How can we decrease the load factor?

## Rehashing

- When the load factor gets “too large” (over a constant threshold on  $\lambda$ ), rehash all the elements into a new, larger table:
  - takes  $O(n)$ , but amortized  $O(1)$  as long as we (just about) double table size on the resize
  - spreads keys back out, may drastically improve performance
  - gives us a chance to retune parameterized hash functions
  - avoids failure for closed hashing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items

## Example of Rehashing

Figure 5.19 Open addressing hash table with linear probing with input 13, 15, 6, 24

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 5.20 Open addressing hash table with linear probing after 23 is inserted

0	6
1	15
2	23
3	24
4	
5	
6	13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 5.21 Open addressing hash table after rehashing

## Next week

- Online assignment 3: Trees, Priority Queues, and Hashing
- Duration: 1h
- Graph