HASHING

COL 106

Shweta Agrawal, Amit Kumar

Slide Courtesy : Linda Shapiro, Uwash
Douglas W. Harder, UWaterloo

The Need for Speed

- Data structures we have looked at so far
  - Use comparison operations to find items
  - Need O(log N) time for Find and Insert
- In real world applications, N is typically between 100 and 100,000 (or more)
  - log N is between 6.6 and 16.6
- Hash tables are an abstract data type designed for **O(1)** Find and Inserts

Fewer Functions Faster

- compare trees and hash tables
  - trees provide for known ordering of all elements
  - hash tables just let you (quickly) find an element

Limited Set of Hash Operations

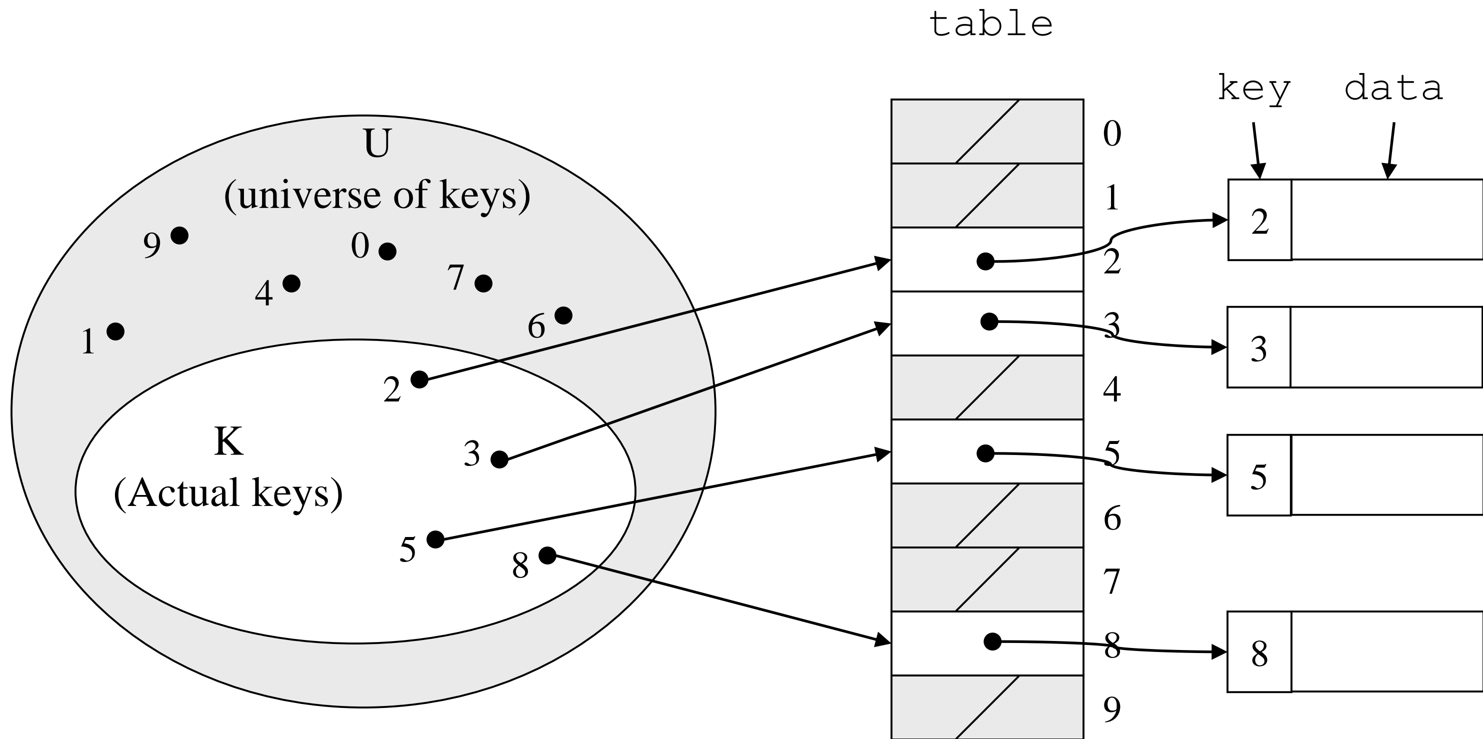
- For many applications, a limited set of operations is all that is needed
  - Insert, Find, and Delete
  - Note that no ordering of elements is implied
- For example, a compiler needs to maintain information about the symbols in a program
  - user defined
  - language keywords

Say that our data has format (key, value). How should we store it for efficient insert, find, delete?

Direct Address Tables

- Direct addressing using an array is very fast
- Assume
  - keys are integers in the set U={0,1,...m-1}
  - m is small
  - no two elements have the same key
- Then just store each element at the array location array[key]
  - search, insert, and delete are trivial

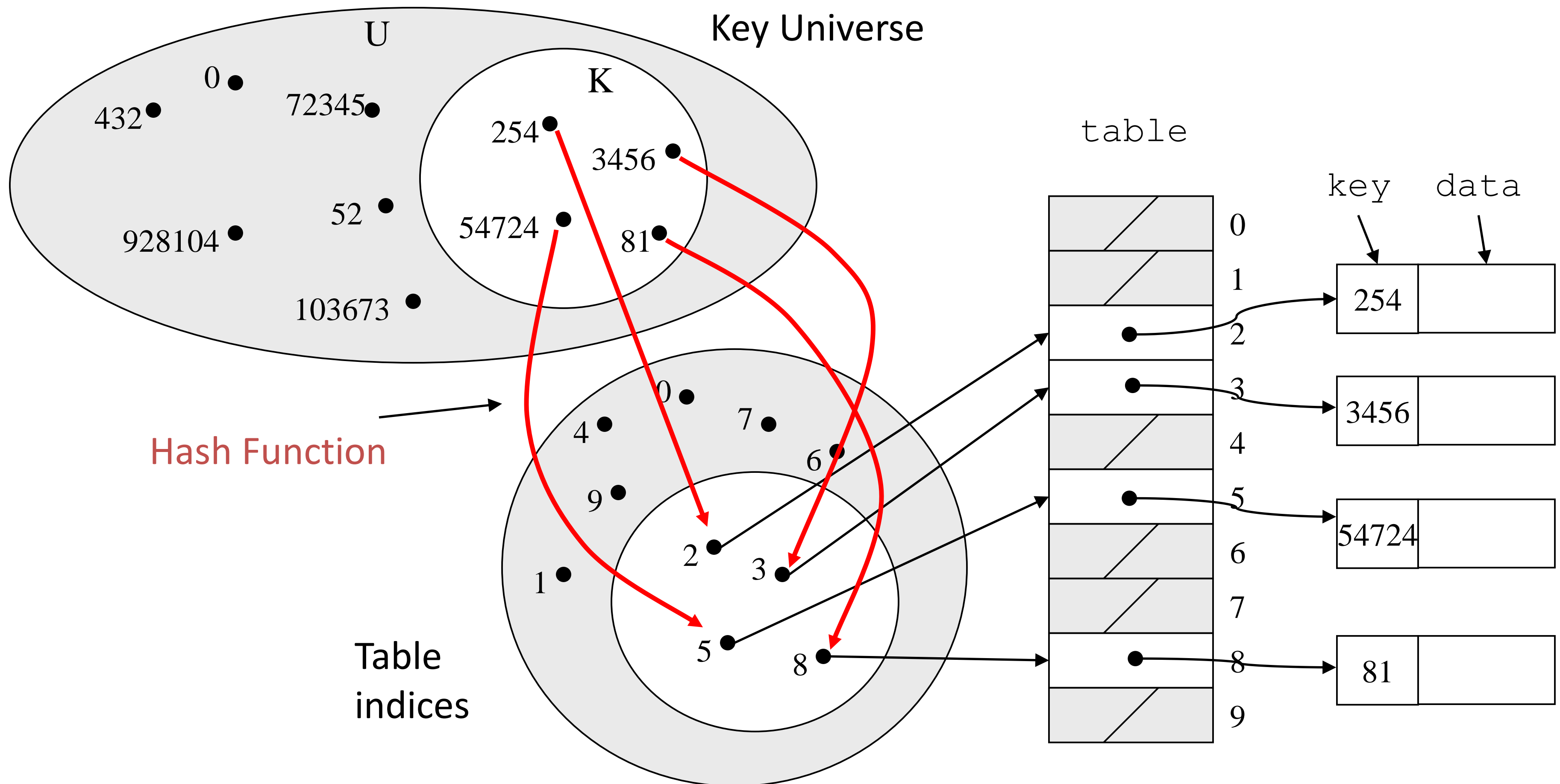
Direct Access Table



An Issue

- If most keys in U are used
  - direct addressing can work very well (m small)
- The largest possible key in U , say m, may be much larger than the number of elements actually stored (|U| much greater than |K|)
  - the table is very sparse and wastes space
  - in worst case, table too large to have in memory
- If most keys in U are not used
  - need to map U to a smaller set closer in size to K

Mapping the Keys



Hashing Schemes

- We want to store N items in a table of size M, at a location computed from the key K (which may not be numeric!)
- Hash function
  - Method for computing table index from key
- Need of a collision resolution strategy
  - How to handle two keys that hash to the same index

"Find" an Element in an Array

Key

element

- Data records can be stored in arrays.
  - A[0] = {"CHEM 110", Size 89}
  - A[3] = {"CSE 142", Size 251}
  - A[17] = {"CSE 373", Size 85}
- Class size for CSE 373?
  - Linear search the array – O(N) worst case time
  - Binary search - O(log N) worst case

Go Directly to the Element

- What if we could directly index into the array using the key?
  - A["CSE 373"] = {Size 85}
- Main idea behind hash tables
  - Use a key based on some aspect of the data to index directly into an array
  - O(1) time to access records

Indexing into Hash Table

- Need a fast *hash function* to convert the element key (string or number) to an integer (the *hash value*) (i.e, map from U to index)
  - Then use this value to index into an array
  - Hash(“CSE 373”) = 157, Hash(“CSE 143”) = 101
- Output of the hash function
  - must always be less than size of array
  - should be as evenly distributed as possible

Choosing the Hash Function

- What properties do we want from a hash function?
  - Want universe of hash values to be distributed randomly to minimize collisions
  - Don't want systematic nonrandom pattern in selection of keys to lead to systematic collisions
  - Want hash value to depend on all values in entire key and their positions

The Key Values are Important

- Notice that one issue with all the hash functions is that the actual content of the key set matters
- The elements in K (the keys that are used) are quite possibly a restricted subset of U, not just a random collection
  - variable names, words in the English language, reserved keywords, telephone numbers, etc, etc

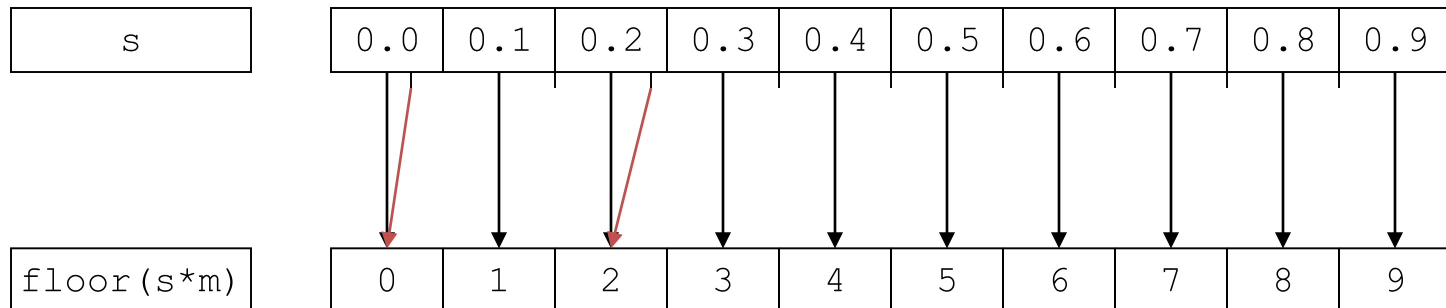
Simple Hashes

- It's possible to have very simple hash functions if you are certain of your keys
- For example,
  - suppose we know that the keys s will be real numbers uniformly distributed over $0 \le s < 1$
  - Then a very fast, very good hash function is
    - $hash(s) = floor(s \cdot m)$
    - where m is the size of the table

Example of a Very Simple Mapping

- hash(s) = floor($s \cdot m$) maps from $0 \leq s < 1$ to 0..m-1

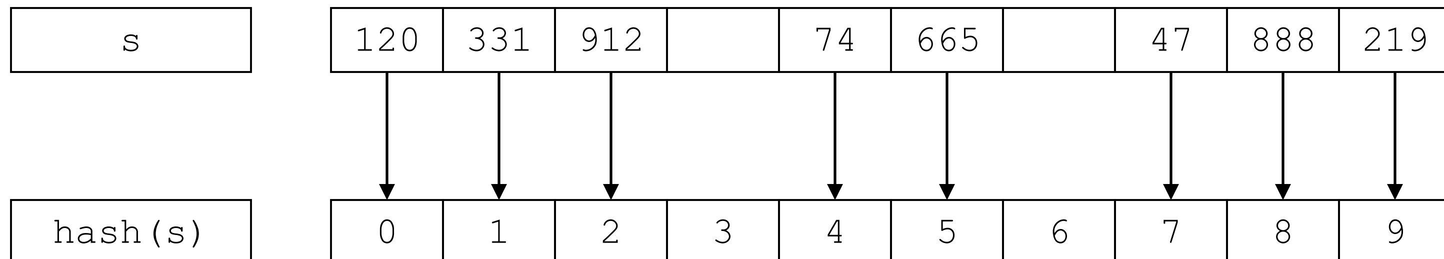
  Example m = 10



Note the even distribution. There are collisions, but we will deal with them later.

Perfect Hashing

- In some cases it's possible to map a known set of keys uniquely to a set of index values
- You must know every single key beforehand and be able to derive a function that works *one-to-one*

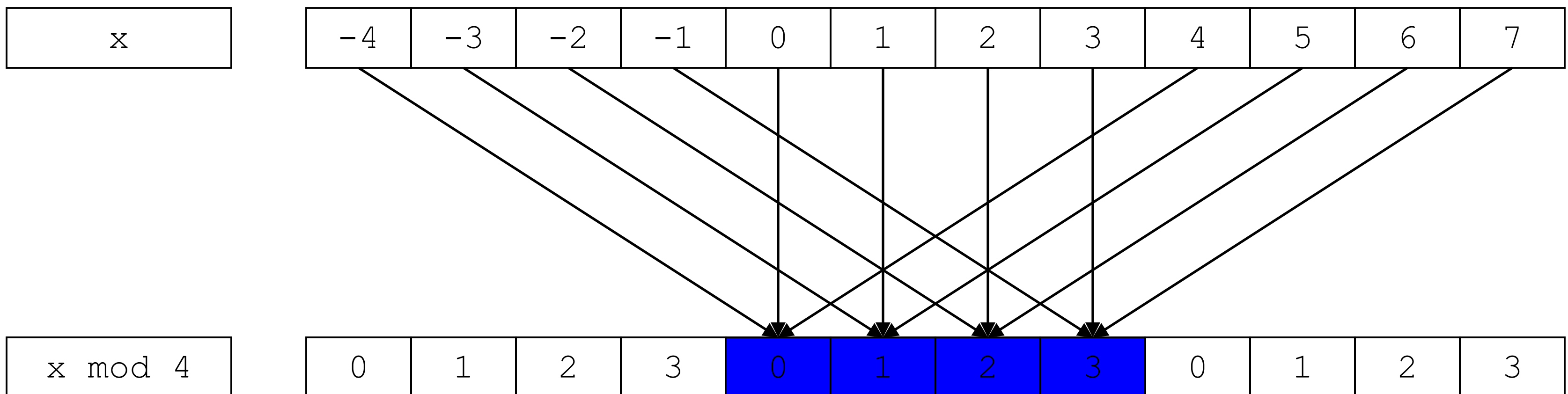


Mod Hash Function

- One solution for a less constrained key set
  - modular arithmetic
- `a` **`mod`** `size`
  - remainder when "a" is divided by "size"
  - in C or Java this is written as **`r = a % size;`**
  - If TableSize = 251
    - 408 mod 251 = 157
    - 352 mod 251 = 101

Modulo Mapping

- *a* mod *m* maps from integers to 0..m-1
  - one to one? no
  - onto? yes



Hashing Integers

- If keys are integers, we can use the hash function:
  - Hash(key) = key mod TableSize
- Problem 1: What if TableSize is 11 and all keys are 2 repeated digits? (eg, 22, 33, …)
  - all keys map to the same index
  - Need to pick TableSize carefully: often, a prime number

Nonnumerical Keys

- Many hash functions assume that the universe of keys is the natural numbers **N**={0,1,…}
- Need to find a function to convert the actual key to a natural number quickly and effectively before or during the hash calculation
- Generally work with the ASCII character codes when converting strings to numbers

Characters to Integers

- If keys are strings can get an integer by adding up ASCII values of characters in *key*
- We are converting a very large string $c_0c_1c_2 \ldots c_n$ to a relatively small number $c_0+c_1+c_2+\ldots+c_n$ mod size.

character →	C	S	E		3	7	3	<0>
ASCII value →	67	83	69	32	51	55	51	0

Hash Must be Onto Table

- Problem 2: What if *TableSize* is 10,000 and all keys are 8 or less characters long?
  - chars have values between 0 and 127
  - Keys will hash only to positions 0 through 8*127 = 1016
- Need to distribute keys over the entire table or the extra space is wasted

Problems with Adding Characters

- Problems with adding up character values for string keys
  - If string keys are short, will not hash evenly to all of the hash table
  - Different character combinations hash to same value
    - “abc”, “bca”, and “cab” all add up to the same value (recall this was Problem 1)

Characters as Integers

- A character string can be thought of as a base 256 number. The string $c_1c_2 \ldots c_n$ can be thought of as the number $c_n + 256c_{n-1} + 256^2c_{n-2} + \ldots + 256^{n-1} c_1$
- Use Horner's Rule to Hash!

```
r= 0;
for i = 1 to n do
r := (c[i] + 256*r) mod TableSize
```

Collisions

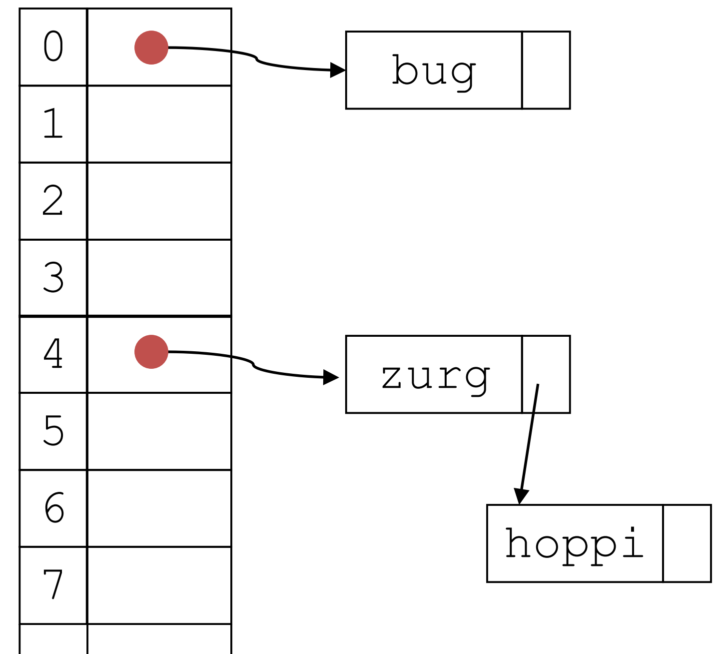
- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value for the mod17 hash function
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!

Collision Resolution

- Separate Chaining
  - Use data structure (such as a linked list) to store multiple items that hash to the same slot
- Open addressing (or probing)
  - search for empty slots using a second function and store item in first empty slot that is found

Resolution by Chaining

- Each hash table cell holds pointer to linked list of records with same hash value
- Collision: Insert item into linked list
- To Find an item: compute hash value, then do Find on linked list
- Note that there are potentially as many as TableSize lists



Why Lists?

- Can use List ADT for Find/Insert/Delete in linked list
  - O(N) runtime where N is the number of elements in the particular chain
- Can also use Binary Search Trees
  - O(log N) time instead of O(N)
  - But the number of elements to search through should be small (otherwise the hashing function is bad or the table is too small)
  - generally not worth the overhead of BSTs

Load Factor of a Hash Table

- Let N = number of items to be stored
- Load factor L = N/TableSize
  - TableSize = 101 and N =505, then L = 5
  - TableSize = 101 and N = 10, then L = 0.1
- Average length of chained list = L and so average time for accessing an item =

  O(1) + O(L)
  - Want L to be smaller than 1 but close to 1 if good hashing function (i.e. TableSize ≈ N)
  - With chaining hashing continues to work for L > 1

Resolution by Open Addressing

- All keys are in the table - no links
  - Reduced overhead saves space
- Cell Full? Keep Looking
- A *probe sequence*: $h_1(k), h_2(k), h_3(k), \ldots$
- Searching/inserting ***k*:** check locations $h_1(k)$, $h_2(k)$, $h_3(k)$
- Deletion ***k*:** *Lazy deletion* needed – mark a cell that was deleted
- Various flavors of open addressing differ in which probe sequence they use

Cell Full? Keep Looking.

- `h`$_i$`(X)=(Hash(X)+F(i)) mod TableSize`
  - Define F(0) = 0
- F is the collision resolution function. Some possibilities:
  - Linear: F(i) = i
  - Quadratic: F(i) = i^2
  - Double Hashing: F(i) = $i \cdot Hash_2(X)$

Linear Probing

- When searching for `K`, check locations `h(K)`, `h(K)+1`, `h(K)+2`, … mod TableSize until either
  - `K` is found; or
  - we find an empty location (`K` not present)
- If table is very sparse, almost like separate chaining.
- When table starts filling, we get clustering but still constant average search time.
- Full table => infinite loop.

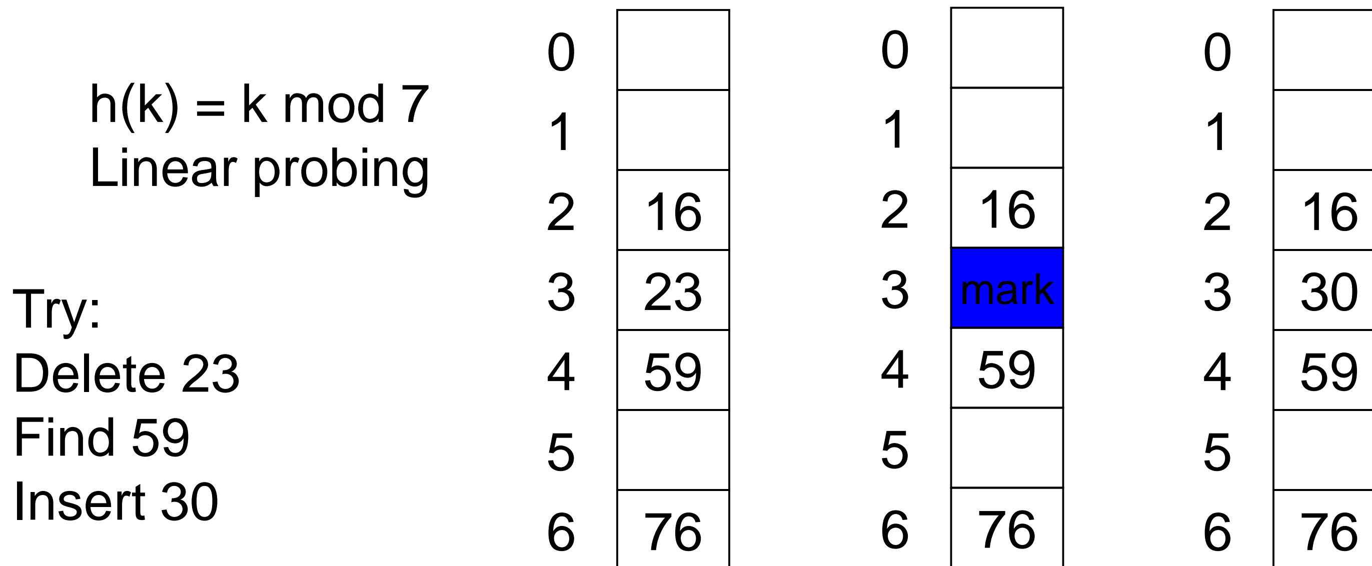
Linear Probing Example

	Insert(76) (6)	Insert(93) (2)	Insert(40) (5)	Insert(47) (5)	Insert(10) (3)	Insert(55) (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
Probes	1	1	1	3	1	3

H(x)= x mod 7

Deletion: Open Addressing

- Must do lazy deletion: Deleted keys are marked as deleted
  - Find: done normally
  - Insert: treat marked slot as an empty slot and fill it



Linear Probing Example:

		delete(40) (5)	search(47) (5)
0	47	47	47
1	55	55	55
2	93	93	93
3	10	10	10
4			
5	40	x	x
6	76	76	76
Probes		1	3

H(k)= k mod 7

Another example

Insert these numbers into this initially empty hash table:

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

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Example

Start with the first four values:

19A, 207, 3AD, 488

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Example

Start with the first four values:

19**A**, 20**7**, 3A**D**, 48**8**

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	**488**		**19A**			**3AD**		

Example

Next we must insert 5BA

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

Example

Next we must insert 5B**A**

- Bin **A** is occupied
- We search forward for the next empty bin

0	1	2	3	4	5	6	7	8	9	**A**	B	C	D	E	F
							207	488		19A	**5BA**		3AD		

Example

Next we are adding 680, 74C, 826

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A	5BA		3AD		

Example

- Next we are adding 68**0**, 74**C**, 82**6**
  - All the bins are empty—simply insert them

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						**826**	207	488		19A	5BA	**74C**	3AD		

Example

Next, we must insert 946

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

Example

Next, we must insert 94**6**

- Bin **6** is occupied
- The next empty bin is 9

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

Example

Next, we must insert ACD

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

Example

Next, we must insert AC**D**

- Bin **D** is occupied
- The next empty bin is E

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

Example

Next, we insert B32

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

Example

Next, we insert B3**2**

– Bin **2** is unoccupied

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

Example

Next, we insert C8B

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

Example

Next, we insert C8**B**

- Bin **B** is occupied
- The next empty bin is F

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

Example

Next, we insert D59

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

Example

Next, we insert D59

- Bin **9** is occupied
- The next empty bin is 1

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

Example

Finally, insert E9C

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

Example

Finally, insert E9C

- Bin **C** is occupied
- The next empty bin is 3

0	1	2	3	4	5	6	7	8	9	A	B	**C**	D	E	F
680	D59	B32	**E9C**			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

Example

Having completed these insertions:

- The load factor is $\lambda = 14/16 = 0.875$
- The average number of probes is $38/14 \approx 2.71$

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 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**	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	**C8B**

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	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

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

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**

– If we just erase it, it is now an empty bin

- By our algorithm, we cannot find ACD, C8B and D59

Erasing

Instead, we must attempt to fill the empty bin

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

- Instead, 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 bin 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 bin to fill
  - We can move 38B 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	**C8B** ←	C8B

Erasing

Now we must attempt to fill the bin 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 bin 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	**D59**

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	**D59**	19A	5BA	74C	ACD	C8B	D59

Erasing

Suppose we delete 207

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

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	

- We are finished

Primary Clustering

We have already observed the following phenomenon:

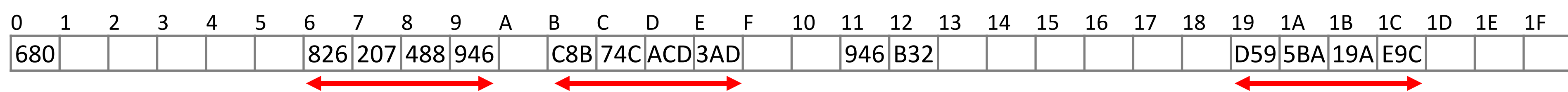
– With more insertions, the contiguous regions (or *clusters*) get larger

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			

This results in longer search times

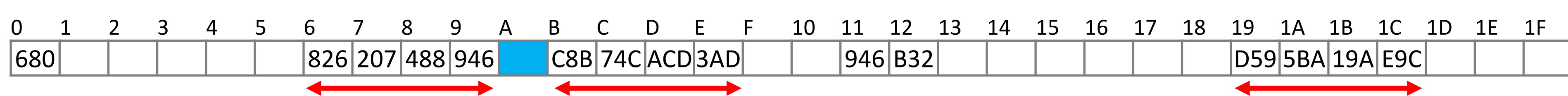
Primary Clustering

We currently have three clusters of length four



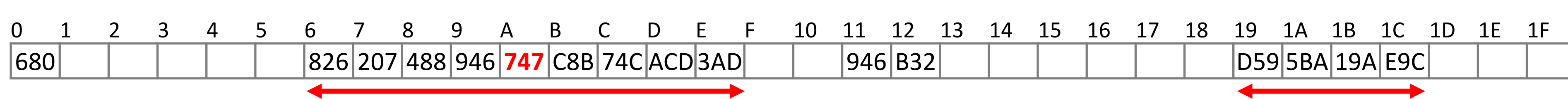
Primary Clustering

There is a $5/32 \approx 16\ \%$ chance that an insertion will fill Bin A



Primary Clustering

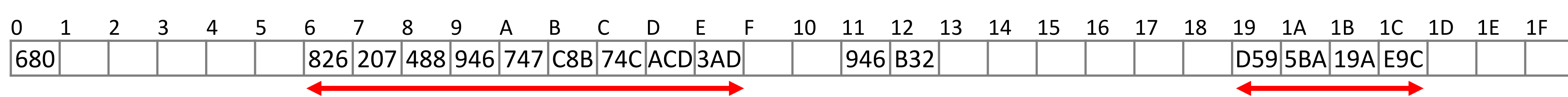
There is a $5/32 \approx 16\ \%$ chance that an insertion will fill Bin A



– This causes two clusters to *coalesce* into one larger cluster of length 9

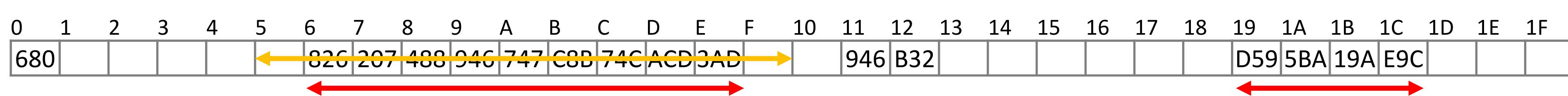
Primary Clustering

There is now a $11/32 \approx 34\ \%$ chance that the next insertion will increase the length of this cluster



Primary Clustering

As the cluster length increases, the probability of further increasing the length increases



In general:

- Suppose that a cluster is of length ℓ
- An insertion either into any bin occupied by the chain or into the locations immediately before or after it will increase the length of the chain
- This gives a probability of $\frac{\ell+2}{M}$

Quadratic Probing

- When searching for `X`, check locations $h_1(X)$, $h_1(X) + 1^2$, $h_1(X) + 2^2$, … `mod TableSize` until either
  - `X` is found; or
  - we find an empty location (`X` not present)

Quadratic Probing

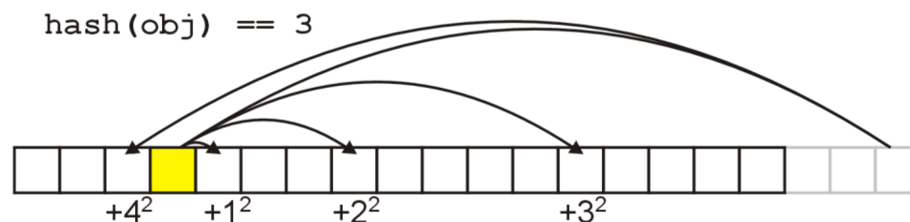
Suppose that an element should appear in bin h:

- if bin h is occupied, then check the following sequence of bins:

$$h+1^2,\ h+2^2,\ h+3^2,\ h+4^2,\ \ h+5^2, \ldots$$

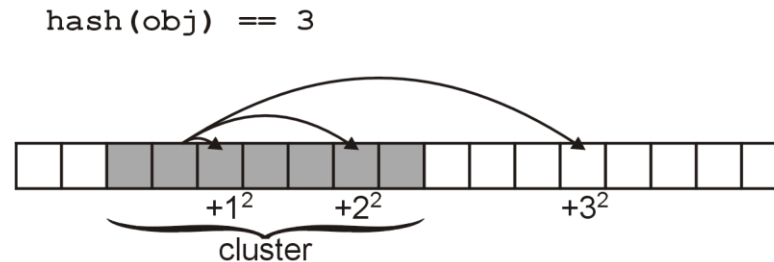
$$h+1,\ \ h+4,\ \ h+9,\ \ h+16,\ h+25, \ldots$$

For example, with $M = 17$:



Quadratic Probing

If one of $h + i^2$ falls into a cluster, this does not imply the next one to map into position i will



Quadratic Probing

For example, suppose an element was to be inserted in bin 23 in a hash table with 31 bins

The sequence in which the bins would be checked is:

23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic Probing

Even if two bins are initially close, the sequence in which subsequent bins are checked varies greatly

Again, with $M = 31$ bins, compare the first 16 bins which are checked starting with 22 and 23:

22 22, 23, 26, 0, 7, 16, 27, 9, 24, 10, 29, 19, 11, 5, 1, 30

23 23, 24, 27, 1, 8, 17, 28, 10, 25, 11, 30, 20, 12, 6, 2, 0

Quadratic Probing

Thus, quadratic probing solves the problem of primary clustering

Unfortunately, there is a second problem which must be dealt with

- Suppose we have $\mathrm{M} = 8$ bins:

  $$1^2 \equiv 1,\ 2^2 \equiv 4,\ 3^2 \equiv 1$$

- In this case, we are checking bin $h + 1$ twice having checked only one other bin

Quadratic Probing

Unfortunately, there is no guarantee that

$$h + i^2 \bmod M$$

will cycle through $0, 1, \ldots, M - 1$

What if :

- Require that M be prime
- In this case, $h + i^2 \bmod M$ for $i = 0, \ldots, (M - 1)/2$ will cycle through exactly $(M + 1)/2$ values before repeating

Quadratic Probing

Example

$M = 11:$

$0, 1, 4, 9, 16 \equiv 5, 25 \equiv 3, 36 \equiv 3$

$M = 13:$

$0, 1, 4, 9, 16 \equiv 3, 25 \equiv 12, 36 \equiv 10, 49 \equiv 10$

$M = 17:$

$0, 1, 4, 9, 16, 25 \equiv 8, 36 \equiv 2, 49 \equiv 15, 64 \equiv 13, 81 \equiv 13$

Quadratic Probing

Thus, quadratic probing avoids primary clustering

- Unfortunately, we are not guaranteed that we will use all the bins

In practice, if the hash function is reasonable, this is not a significant problem

Secondary Clustering

The phenomenon of primary clustering does not occur with quadratic probing

However, if multiple items all hash to the same initial bin, the same sequence of numbers will be followed

- This is termed *secondary clustering*
- The effect is less significant than that of primary clustering

Secondary Clustering

Secondary clustering may be a problem if the hash function does not produce an even distribution of entries

One solution to secondary is double hashing: associating with each element an initial bin (defined by one hash function) and a skip (defined by a second hash function)

Double Hashing

- When searching for X, check locations $h_1(X)$, $h_1(X)+h_2(X)$, $h_1(X)+2*h_2(X)$,... mod Tablesize until either
  - X is found; or
  - we find an empty location (X not present)
- Must be careful about $h_2(X)$
  - Not 0 and not a divisor of M
  - eg, $h_1(k) = k \bmod m_1$, $h_2(k)=1+(k \bmod m_2)$
    where m_2 is slightly less than m_1

Rules of Thumb

- Separate chaining is simple but wastes space…
- Linear probing uses space better, is fast when tables are sparse
- Double hashing is space efficient, fast (get initial hash and increment at the same time), needs careful implementation

Caveats

- Hash functions are very often the cause of performance bugs.
- Hash functions often make the code not portable.
- If a particular hash function behaves badly on your data, then pick another.