1. Divide into teams of 4-5 students.

2. Repeat until your group has completed 100 experiments:

   Flip a coin until it comes up heads and then record the number of coin flips it took to get heads.

3. Fill out a chart and use it to compute the average number of flips.

Number of flips for heads	Number of trials
1	
2	
3	
…	

Final exam tutorial time:
Thursday Dec. 5: from 11am until all questions are answered. ECS 116.

.

Written Assignment #5: due Wed. Dec. 4, beginning of class.

Please take the time to give some feedback to your lab instructor.

Hashing

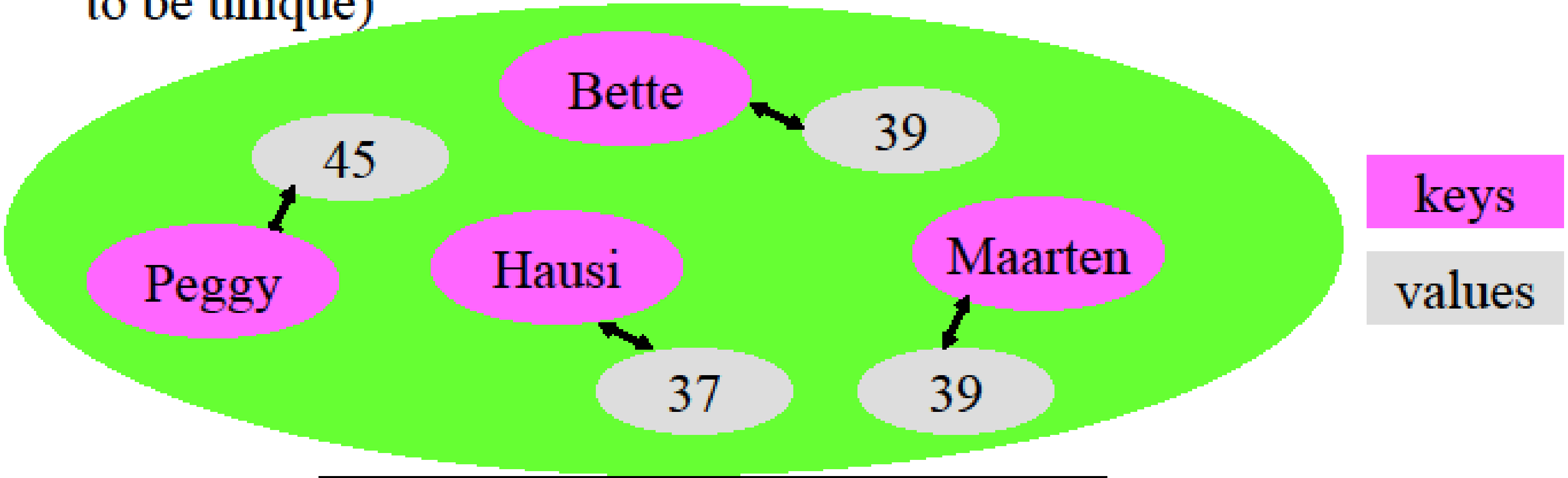
Just as radix sort can beat the

$\Omega(n \log n)$ barrier for the comparison model, hashing can be faster in practice (although not always in the worst case) than the log n time used in a binary search.

Slides today from Spring 2007 were created by Ulrike Stege and/or Hausi Muller (blue boxes on top).

Dictionary

- A dictionary is an unordered container that contains key-value pairs
- The keys are unique, but the values can be anything (e.g., don’t have to be unique)



Bette

39

45

Peggy

Hausi

Maarten

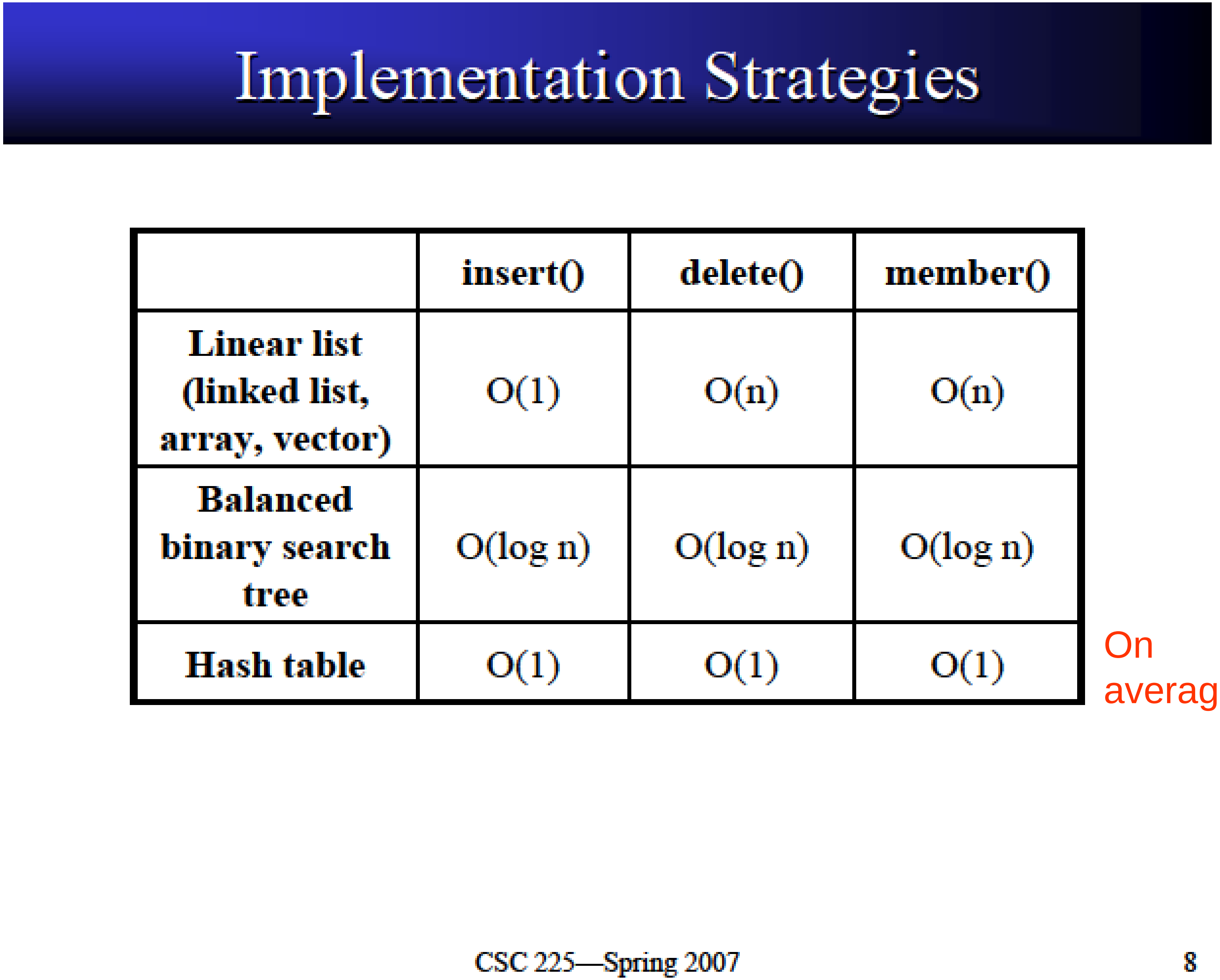
37

39

keys

values

Search key	Associated value
Bette	39
Hausi	37
Peggy	45
Maarten	39



Implementation Strategies

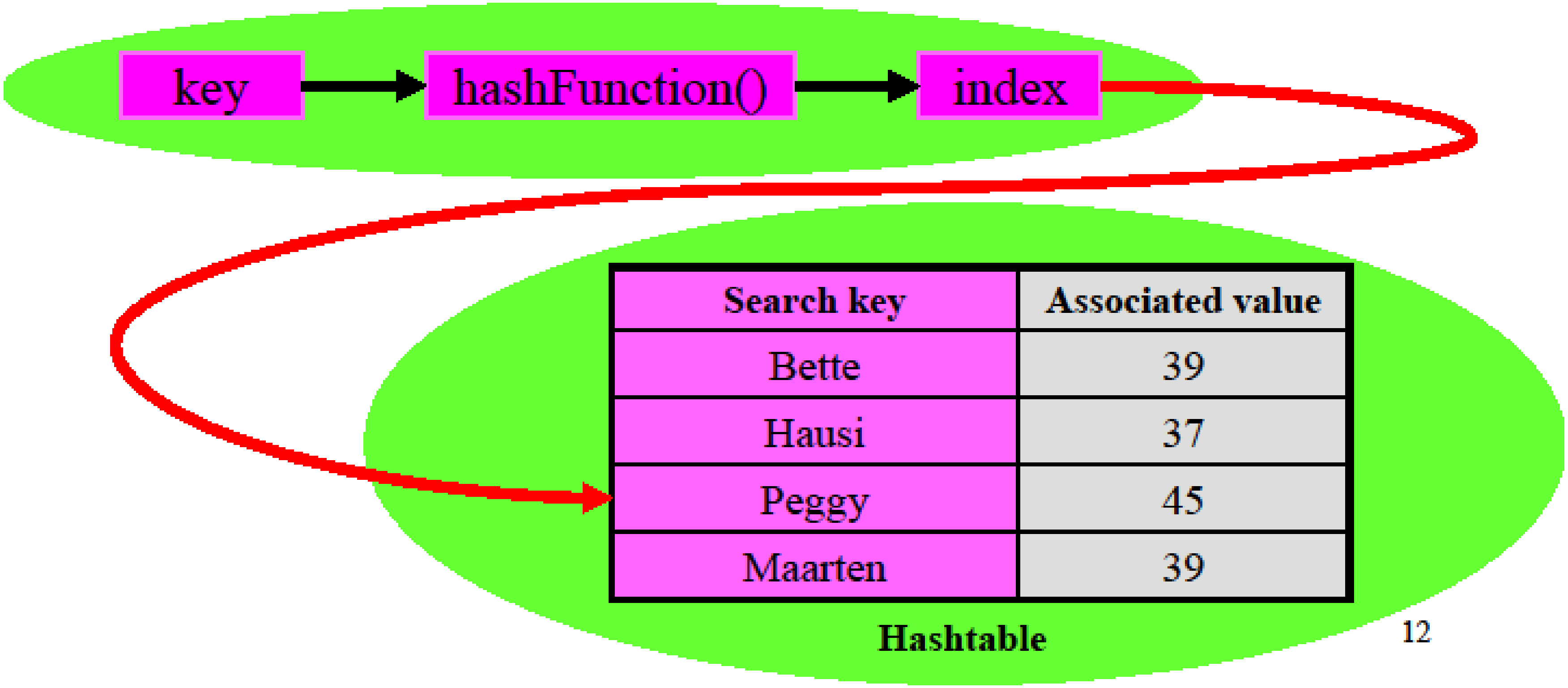
	insert()	delete()	member()
Linear list (linked list, array, vector)	O(1)	O(n)	O(n)
Balanced binary search tree	O(log n)	O(log n)	O(log n)
Hash table	O(1)	O(1)	O(1)

On average

How does hashing work?

On average

- How can we find an element in a hash table in constant time O(1)?
- Given a key, we compute an index into the hash table using a *hash function* of a *hash code*



Search key	Associated value
Bette	39
Hausi	37
Peggy	45
Maarten	39

Hashtable

Hash Functions

- A *hash function* or *hash code* maps keys to indices
  - It should map keys uniformly across all possible indices
  - It should be fast to compute
  - It should be applicable to all objects
- Hash table size
  - The hash table size should be a prime number
  - The hash table size should **not** be a power of two; however most advanced hash functions use a power of two because division is much faster than with a prime
- When two keys map to the same index, we have a *hash collision*
- When a collision occurs, a *collision resolution algorithm* is used to establish the locations of the colliding keys
- In some cases when we know all of the key values in advance we can construct a perfect hash function that maps each key to a different index (i.e., with no collisions)

Excellent article: Designing a good hash function is an art

http://burtleburtle.net/bob/hash/doobs.html

Collision: two data items have the same hash value

Collision resolution scheme: strategy for dealing with collisions.

Open addressing: use extra space, for example a linked list of items hashing to the same place.

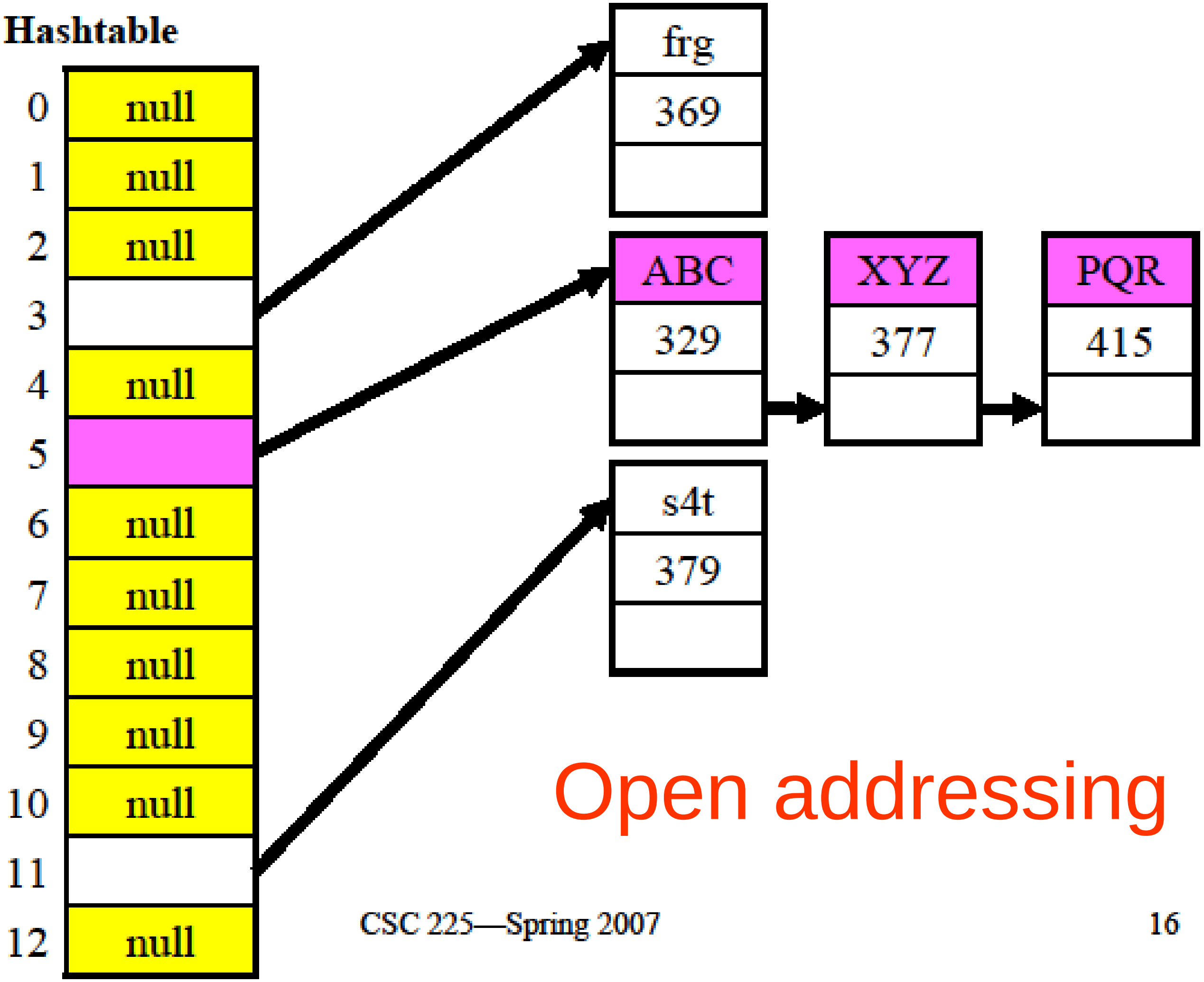
Closed addressing: resolve collisions by looking for an empty spot elsewhere in the hash table.

String Hash Function: An Example

- Let
  - **`s`** be a key of type **`String`**
  - **`sum`** be the sum of the ordinal values of all the characters in **`s`**
  - **`N`** be the hash table size
- Then the hash table index **`k`** is
  - **`k = sum % N`**
    - where **%** is the modulo operator
    - Thus, **`k`** is in the range **`0`** to **`N-1`**
- Example

```
s = "ABC"
N = 59 (prime number)
sum = ord('A') + ord('B') + ord('C')
    = 60 + 61 + 62 = 183
k = sum % N = 183 % 59 = 6
```

Separate Chaining



Hashtable

0	null
1	null
2	null
3	
4	null
5	
6	null
7	null
8	null
9	null
10	null
11	
12	null

frg
369

ABC
329

XYZ
377

PQR
415

s4t
379

Open addressing

Linear Probing

- Linear probing is an open addressing algorithm
- Locations are checked from the hash location **`k`** to the end of the table and the element is placed in the first empty slot
  - ➢ If the bottom of the table is reached, checking “wraps around” to the start of the table (i.e., modulo hashtable size)
- Collision resolution factors into **`member()`**, **`insert()`**, **`delete()`**
- Thus, if linear probing is used, these routines must continue down the table until a match or empty location is found
- Even though the hash table size is a prime number (i.e., 13), this is probably not an appropriate size; the size should be at least 30-40% larger than the maximum number of elements ever to be stored in the table

0	empty
1	empty
2	deleted
3	frg
4	empty
5	ABC
6	XYZ
7	PQR
8	empty
9	deleted
10	empty
11	s4t
12	empty

Quadratic Probing

- Quadratic probing is another open addressing algorithm
- Locations are checked from the hash location to the end of the table and the element is placed in the first computed empty slot
  - ➢ Instead of probing consecutive location, we probe the 1st, 4th, 9th, 16th, etc. — this is called quadratic probing
  - ➢ If the bottom of the table is reached, checking "wraps around" to the start of the table (i.e., modulo hash table size)

0	empty
1	empty
2	deleted
3	frg
4	empty
5	ABC
6	XYZ
7	empty
8	empty
9	PQR
10	empty
11	s4t
12	empty

If p is a prime number and k is an integer, $1 \leq k \leq p-1$, then

$$x, x+k, x+2k, x+3k, \ldots, x + (p-1)k$$

are all different numbers modulo p.

So this is a probe sequence that can be used to visit all locations of the hash table.

This is why primes a good choices for a hash table size.

A small example:

Assume integral key values.
Simple hash function: hash(key)= key modulo p, where p, a prime number, is the hash table size.

Suppose p=11.
The hash table is an array H:
int H[0…10];

In practice it best to choose p so that the number of data items you expect to have is not more than 60% of p.

NULL_DATA= some value that is illegal for a key.

If any integer is permitted, a separate True/False array can be used to indicate if a table entry is being used or not.

To initialize the hash table:

```
for (i=0; i < P; i++) H[i]= NULL_DATA;
```

To determine if a key is in the table:

Use probe sequence until either the key or an empty table location is found.

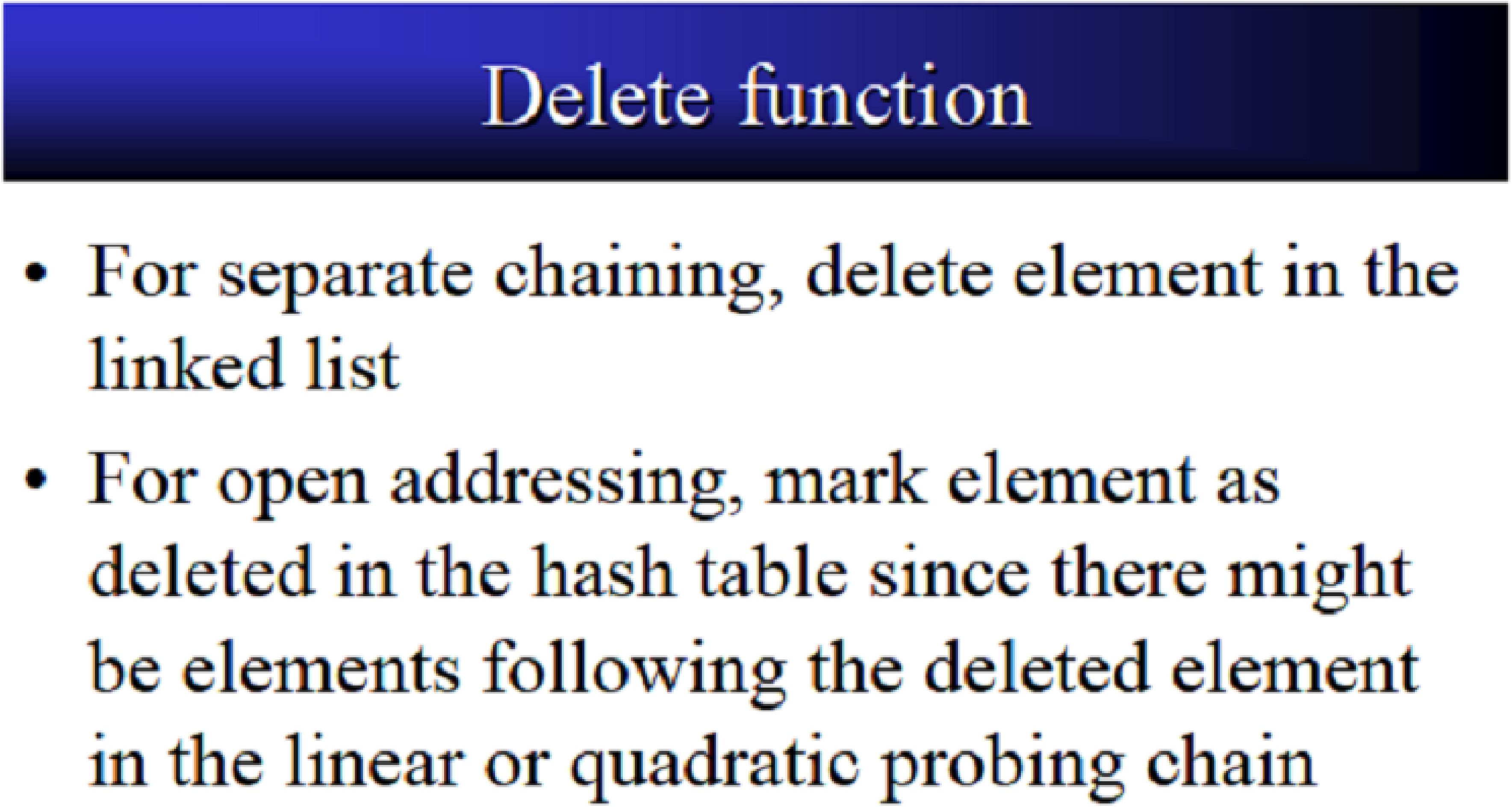
```
int find(int [] H,  int key)

r= hash(key)

while (H[ r] != NULL_DATA && H[r]!= key)

{     r= (r+1) % P;  }

return(r) // Empty spot or position of key
```



Delete function

- For separate chaining, delete element in the linked list
- For open addressing, mark element as deleted in the hash table since there might be elements following the deleted element in the linear or quadratic probing chain

Analysis of Hash table Access

- If the number of collisions is small, searching, inserting, and deleting elements in a hash table takes O(1) time
- To reduce the number of collisions, in addition to using a good hash function, we should make sure that the table does not get too full
- The **load factor** of a hash table is the ratio of occupied slots to total slots
- For best results, the load factor should not be above 0.6
- If it gets higher, we should extend the hash table and re-hash all of its elements

hash(x) = x mod 11,

Using linear probing, insert:

11, 32, 52, 30, 15, 31, 49, 9, 19

x	hash(x)	Probe sequence
…	…	…

Linear probing means that we try location k=hash(x) first and then k+1, k+2, k+3, k+4, … until an empty spot is found.

What happens when we try to find 8?

Summary

- Dictionary
  - Member, insert, delete; associations: keys, values
- Hash table
  - Array of hash entries
  - Hash function
    - Compute index from key by 'hashing' the key
    - Distribute indeces over entire index space: 0..htSize
  - Collisions
    - Different keys map to the same index
    - Open addressing: linear, quadratic probing
    - Separate chaining
  - Hash table implementation
    - Hash table size should be a prime number
    - 3-state hash table entry (empty, valid, deleted)
  - Time complexity
    - Member, insert, delete take O(1) time (i.e., constant time)

Worst case time:

O(n)

A run is a consecutive sequence of cells in the array (treated as a cyclic array) that are non-empty.

When using linear probing, long runs are likely to form and then after they do, the probability of hashing into a run is high. Then the search must continue to the end of the run before the key can be inserted and the run grows in length after the insertion.

Quadratic probing is an attempt to deal with this.

Recall, the jump sequence is obtained by adding 1, 4, 9, 16, 25, …
= 1^2, 2^2, 3^2, 4^2, 5^2, …

This helps but we still get “runs” when keys hash to the same space but the “runs” are spaced out in the array.

One solution to this problem:

After hashing to choose an initial table location, use a second hash function to choose the jump amount.

This helps to ensure the probe sequence is likely to hit an empty spot with a probability that corresponds to the percent of open spots in the table.

Worst case examples of O(n) however can still be created (chose keys all having the same hash value and same jump value).

$\mathrm{hash}(x) = x \bmod 11$,

$j = [(x/10) \bmod 10\ ] + 1$ // jump for probes

Insert: 11, 32, 52, 30, 15, 31, 49, 9, 19

x	hash(x)	j	Probe sequence
…	…	…	…

What happens when we try to find 27?