Hashing 1

EECS 233

Hashing

- We have learned several data structures that allow us to store and search for data items using their keys fields.
  - Lists and arrays - linear search complexity
  - Sorted arrays - log complexity for search; linear for inserts
  - Various trees
    - ✓ Generally logarithmic time complexity
    - ✓ Rich operations:
      - Insert, remove, search, find max/min, top-k
- Can we do better than logarithmic complexity?

Taking Stock…

Taking Stock…

- The efficiency of the different data structures and operations:

Data structure	Search for an item	Insert an item
List (unsorted array)	**O(N)**	**O(N)**
List (sorted array)	**O(log N)**	**O(N)**
List (linked list)	**O(N)**	**O(1)**
Binary search tree	**O(h)** (h = log N if balanced)	**O(h)**
Balanced search tree	**O(log N)**	**O(log N)**

Taking Stock…

- The efficiency of the different data structures and operations:

Data structure	Search for an item	Insert an item
List (unsorted array)	**O(N)**	**O(N)**
List (sorted array)	**O(log N)**	**O(N)**
List (linked list)	**O(N)**	**O(1)**
Binary search tree	**O(h)** (h = log N if balanced)	**O(h)**
Balanced search tree	**O(log N)**	**O(log N)**

- Hash tables and hashing techniques may allow us to search, insert, and delete an item in sub-logarithmic time, e.g., O(1)

Storing Objects in an Array: Keys and Indexes

Storing Objects in an Array: Keys and Indexes

- Key: a (unique) attribute of object
- Index: a position of object in an array

Storing Objects in an Array: Keys and Indexes

- Key: a (unique) attribute of object
- Index: a position of object in an array
- Unsorted arrays:
  - Indexes are independent of keys
  - Searching is dumb and inefficient
- Sorted arrays:
  - Indexes are related to keys
  - Searching is smart (no longer blind) and efficient (logarithmic)

Storing Objects in an Array: Keys and Indexes

- Key: a (unique) attribute of object
- Index: a position of object in an array
- Unsorted arrays:
  - Indexes are independent of keys
  - Searching is dumb and inefficient
- Sorted arrays:
  - Indexes are related to keys
  - Searching is smart (no longer blind) and efficient (logarithmic)
- The hashing idea: treat the key as an index!
  - Searching is simple and takes almost no time (constant) = genius

Storing Objects in an Array: Keys and Indexes

- Key: a (unique) attribute of object
- Index: a position of object in an array
- Unsorted arrays:
  - Indexes are independent of keys
  - Searching is dumb and inefficient
- Sorted arrays:
  - Indexes are related to keys
  - Searching is smart (no longer blind) and efficient (logarithmic)
- The hashing idea: treat the key as an index!
  - Searching is simple and takes almost no time (constant) = genius
- Example: storing grades about students in this class
  - Give each student a unique key (integer from 1-80).
  - Store the student records in an array, in the position determined by the key
  - We can perform both search/update and insertion in $O(1)$ time (for up to 80 students).

Hash Functions

Hash Functions

- Problem with student grades:
  - A student will have different keys in different classes
  - What if we used a social security number?
  - Nice, but a humongous array!

Hash Functions

- Problem with student grades:
  - A student will have different keys in different classes
  - What if we used a social security number?
  - Nice, but a humongous array!
- In many real-world problems, the key attribute has semantic meaning
  - Cannot be arbitrarily assigned
  - Phone book: key is person's name, not a unique number
  - Web cache: key is a URL

Hash Functions

- Problem with student grades:
  - A student will have different keys in different classes
  - What if we used a social security number?
  - Nice, but a humongous array!
- In many real-world problems, the key attribute has semantic meaning
  - Cannot be arbitrarily assigned
  - Phone book: key is person's name, not a unique number
  - Web cache: key is a URL
- To handle these problems, we use a *hash function*
  - Convert ("map") keys into array indices
  - Domain: the keys; range: integers in [0, size-of-array)
  - The word "hash": to chop into small pieces (Merriam-Webster)
    - ✓ Chopping large domain space into small number of array cells

Hash Tables

- Example: student list
  - Index = SSN mod 80
- Problem: multiple keys mapping to the same index
  - Two students with SSN 511-00-0001 and 511-00-0161
    - ✓ 511000001 mod 80 = 511000161 mod 80 = 1
  - We need techniques to handle such cases called *collisions*
- The resulting data structure is known as a *hash table*
  - Hash function
  - Array
  - Procedures and data structures to handle collisions
- Operations:
  - Insert
  - Remove
  - Contains (search)
  - isEmpty, makeEmpty, etc.
  - No findMax!

Hash Functions Desiderata

Hash Functions Desiderata

- Example 1: Salary as key
  - 10-workers shop
  - Keep employees record in hash table with (salary mod 10) hash function
  - What if all salaries are multiple of 10K? - Bad function!
  - What if all salaries are multiple of 2K? - Better function!
  - Random salaries - Good function!

Hash Functions Desiderata

- Example 1: Salary as key
  - 10-workers shop
  - Keep employees record in hash table with (salary mod 10) hash function
  - What if all salaries are multiple of 10K? - Bad function!
  - What if all salaries are multiple of 2K? - Better function!
  - Random salaries - Good function!

- Example 2: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - ✓ h(key) = (the byte sum of all characters) mod table-size
    - ✓ example: h("cat") = ('c' + 'a' + 't') mod 100000
    - ✓ But: assume words are mostly up to 20 characters-long
    - ✓ Max sum is 127*20 = 2540; any larger table is of no use!

Hash Functions Desiderata

- Example 1: Salary as key
  - 10-workers shop
  - Keep employees record in hash table with (salary mod 10) hash function
  - What if all salaries are multiple of 10K? - Bad function!
  - What if all salaries are multiple of 2K? - Better function!
  - Random salaries - Good function!

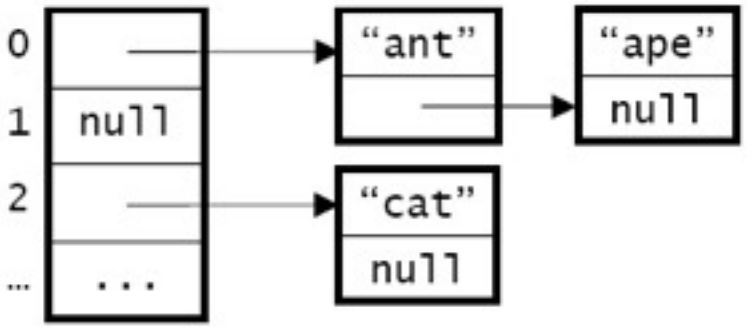
- Example 2: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - ✓ h(key) = (the byte sum of all characters) mod table-size
    - ✓ example: h("cat") = ('c' + 'a' + 't') mod 100000
    - ✓ But: assume words are mostly up to 20 characters-long
    - ✓ Max sum is 127*20 = 2540; any larger table is of no use!

- Requirements for good hash functions:
  - Full table size utilization
  - Even ("uniform") key mapping throughout the table
    - ✓ Utilizing known key distribution in the objects
    - ✓ Making hash function "random" - the distribution of keys is independent of distribution of indexes they map to

Handling Collisions with Chaining

- If multiple items are assigned the same hash code, we "chain" them together. Each position in the hash table serves as a *bucket* that is able to store multiple data items.

- Two implementations:
  - each bucket is itself an array (or points to an array)
    - ✓ disadvantages: (1) large buckets can waste memory, (2) a bucket may become full; *overflow* occurs when we try to add an item to a full bucket
  - a linked list
    - ✓ disadvantage: memory overhead for the references



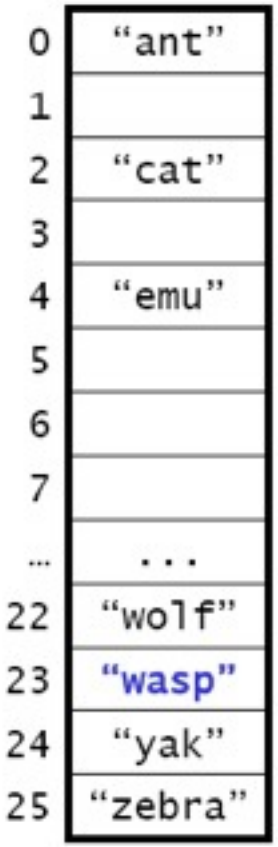
Operations with Chaining

- Search for an item
  - Need to traverse the corresponding linked list or array
  - To guarantee short lists, the number of hash table slots should be of the same order as the total number of items
  - *Load factor:* ratio of the number of items to the number of hash table slots
- Inserting an item
  - Insertion in a linked list or array
  - New list node must be allocated
  - Array size need to be dynamically adjusted
- Removing an item
  - What if a large array is almost empty (wasting storage)?
  - List node must be garbage-collected

Handling Collisions with Open Addressing

- When the position assigned by the hash function is occupied, find another open position - a process called *probing*.
  - Example: h(key) = <character encoding of first char> – <encoding of 'a'>
  - "wasp" has a hash code of 22, but it ends up in position 23, because position 22 is occupied.

- The hash table also performs probing to search for an item.
  - example: when searching for "wasp", we look in position 22 and then look in position 23
  - we can only stop a search when we reach an empty position

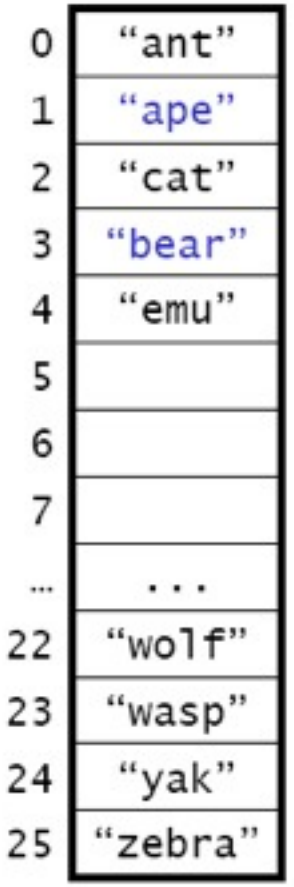


Linear Probing for Open Addressing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.
  - Example:
    - ✓ “ape” (h = 0) would be placed in position 1, because position 0 is already full.
    - ✓ “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
    - ✓ where would “whale” end up?

- Advantage: if there is an open position, linear probing will eventually find it.

- Disadvantage: “clusters” of occupied positions develop
  - Increases the lengths of subsequent probes.
  - As load factor increases, both search and insert times look increasingly linear!



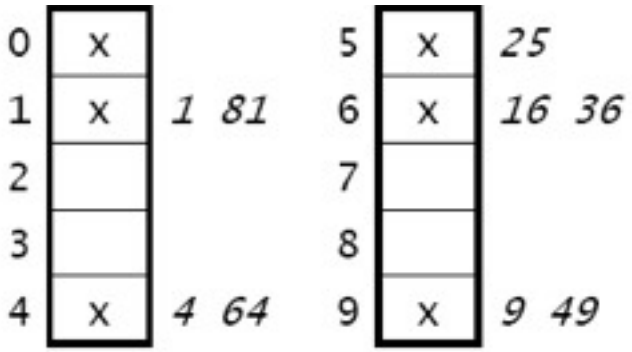
Quadratic Probing for Open Addressing

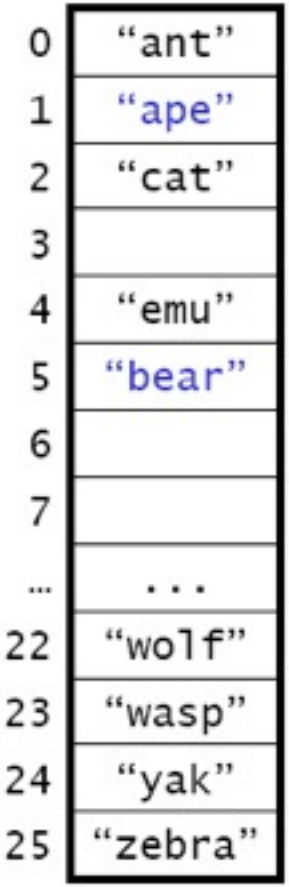
- Probe sequence: h(key), h(key) + 1, h(key) + 4, h(key) + 9, …, wrapping around as necessary.
  - the offsets are perfect squares: h + 1^2, h + 2^2, h + 3^2, …
  - Example:
    - ✓ “ape” (h = 0): try 0, 0 + 1 – open!
    - ✓ “bear” (h = 1): try 1, 1 + 1, 1 + 4 – open!
    - ✓ “whale”?

- Advantage: reduces clustering

- Disadvantage: it may fail to find an existing open position.

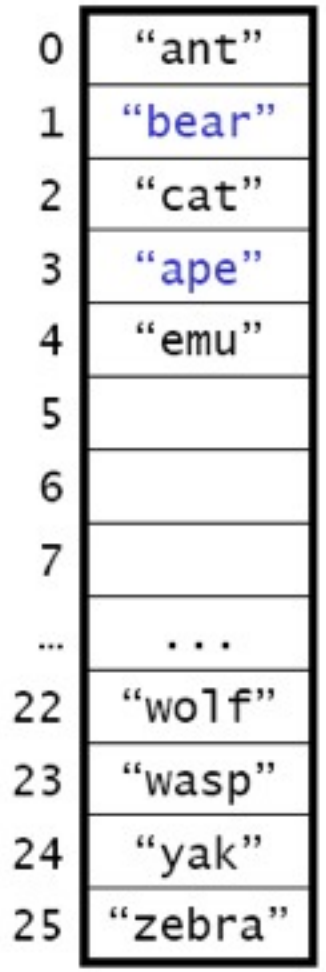
Example:
- table size = 10
- x = occupied
- trying to insert a key with h(key) = 0



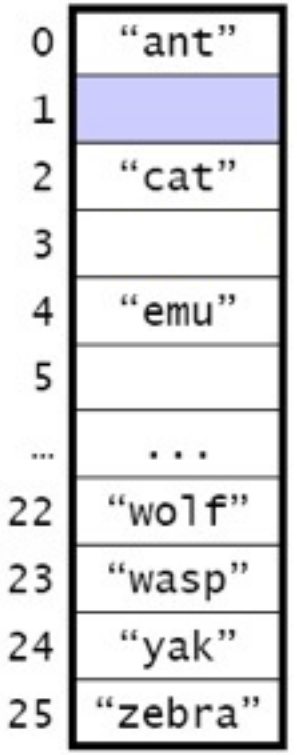


Double Hashing for Open Addressing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - Probe sequence: h1, h1 + h2, h1 + 2*h2, …
- Example:
  - h1 = our previous function h
  - h2 = number of characters in the string
  - “ape” (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - “bear” (h1 = 1, h2 = 4): try 1 – open!
  - “whale”?
- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - Theorem: will find an open position if there is one, provided the table size is a prime number.
  - Disadvantage: the need for two hash functions

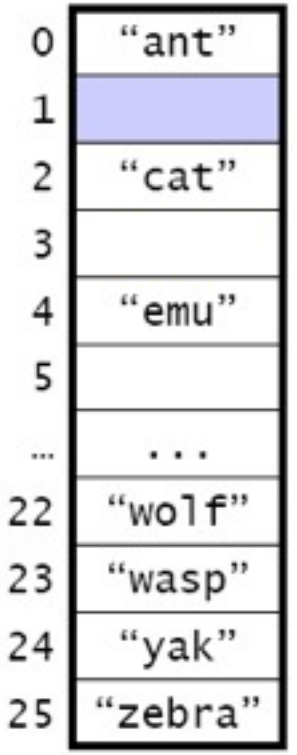


Removing Items with Open Addressing



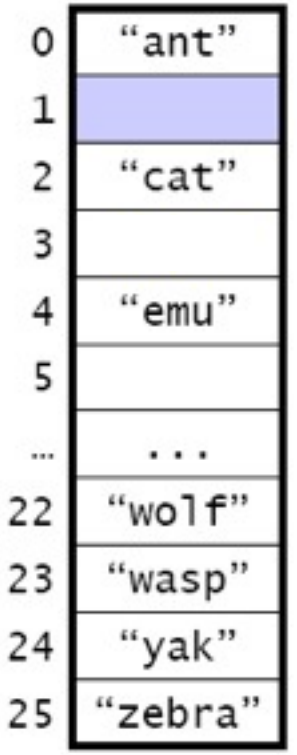
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing



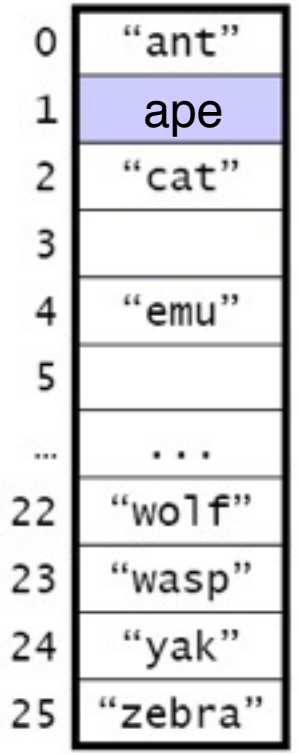
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!



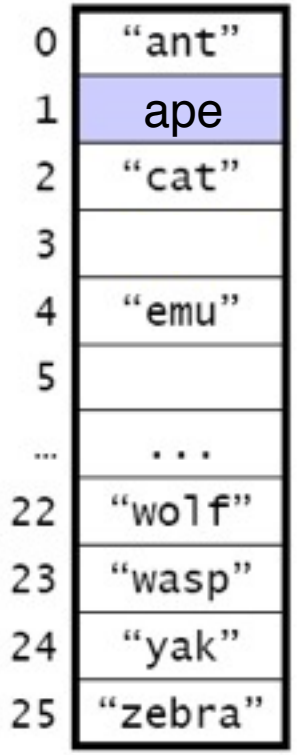
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!



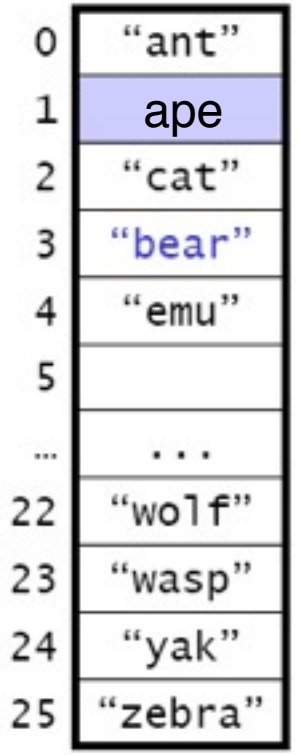
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!



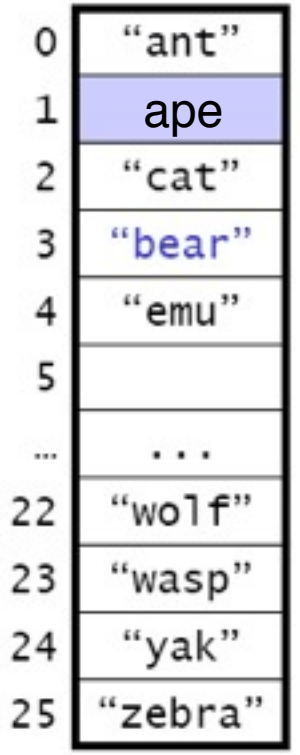
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!



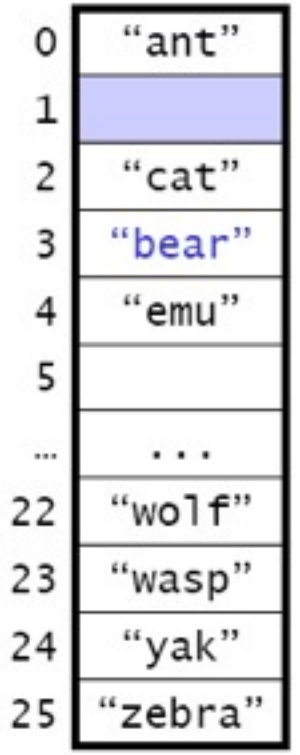
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”



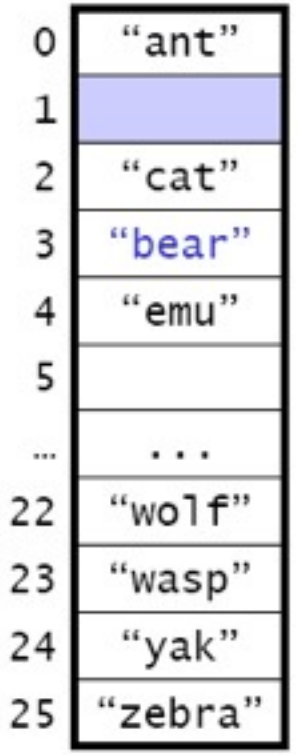
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”



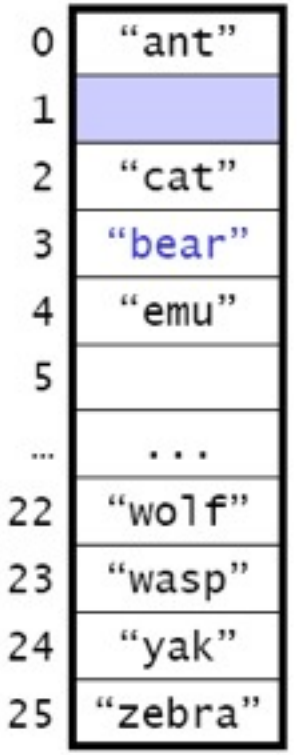
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item



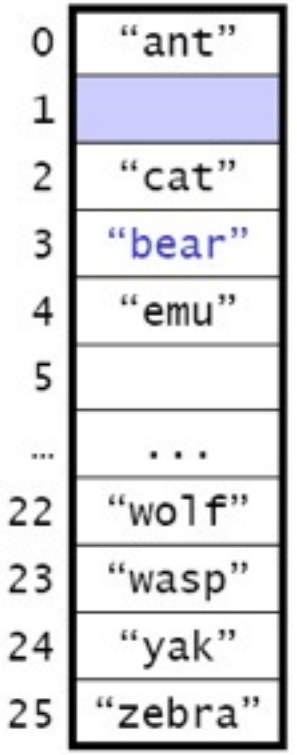
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,



Removing Items with Open Addressing

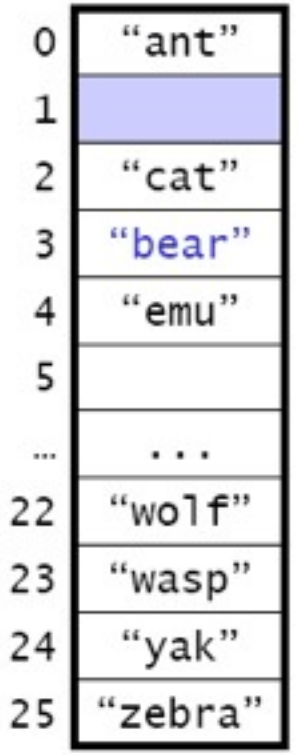
- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!



Removing Items with Open Addressing

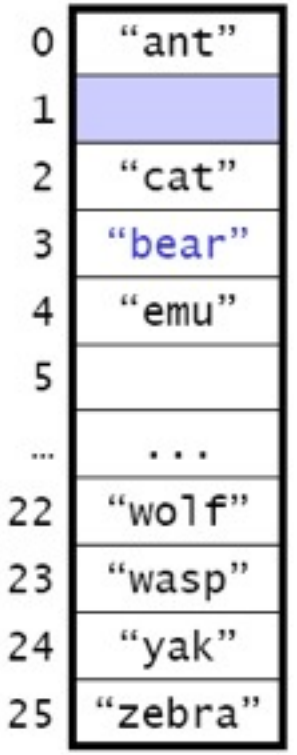
- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!

  - Cannot tell if it is not in the table



Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!
  - Cannot tell if it is not in the table



- Any difference with quadratic probing or double hashing?

Removing Items with Open Addressing

- When we remove an item from a position, we need to leave a special value in that position to indicate that an item was removed.
- Three types of positions: occupied, empty, “removed”.
- When we search for a key, we stop probing when we encounter an empty position, but not when we encounter a removed position.
  - How to search for “ape”?
  - How to search for “bear”?
- We can insert items in either empty or removed positions.

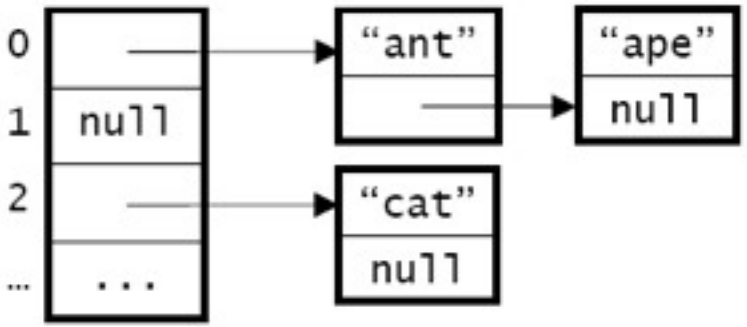
Previous Lecture

- Hash functions desiderata
- Handling collisions with open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Need to distinguish between “removed” and “empty” positions

Handling Collisions with Chaining

- If multiple items are assigned the same hash code, we "chain" them together. Each position in the hash table serves as a *bucket* that is able to store multiple data items.

- Two implementations:
  - each bucket is itself an array (or points to an array)
    - ✓ disadvantages: (1) large buckets can waste memory, (2) a bucket may become full; *overflow* occurs when we try to add an item to a full bucket
  - a linked list
    - ✓ disadvantage: memory overhead for the references

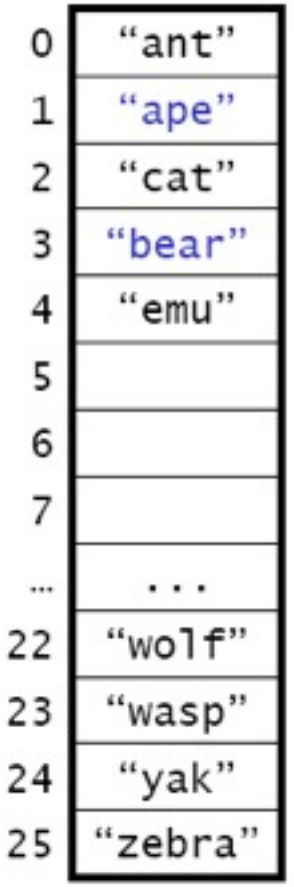


Linear Probing for Open Addressing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.
  - Example:
    - ✓ “ape” (h = 0) would be placed in position 1, because position 0 is already full.
    - ✓ “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
    - ✓ where would “whale” end up?

- Advantage: if there is an open position, linear probing will eventually find it.

- Disadvantage: “clusters” of occupied positions develop
  - Increases the lengths of subsequent probes.
  - As load factor increases, both search and insert times look increasingly linear!



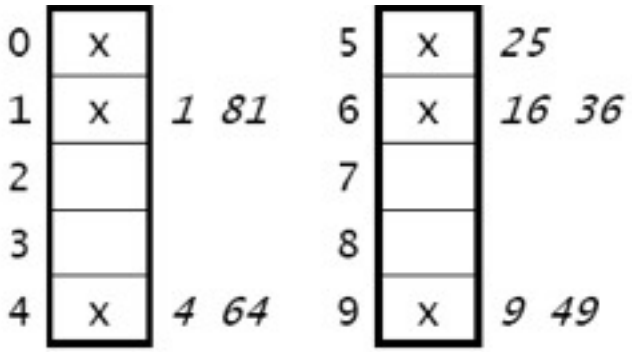
Quadratic Probing for Open Addressing

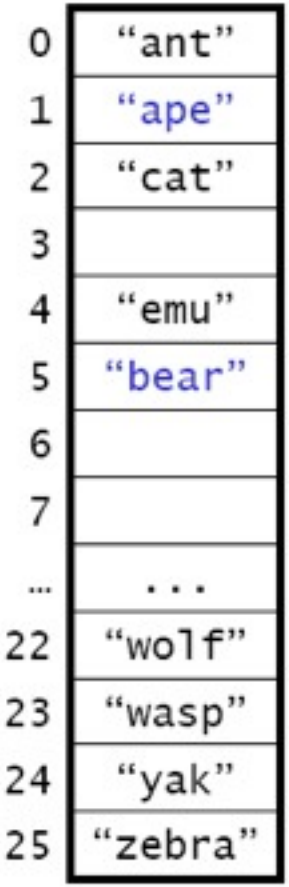
- Probe sequence: h(key), h(key) + 1, h(key) + 4, h(key) + 9, …, wrapping around as necessary.
  - the offsets are perfect squares: h + 1^2, h + 2^2, h + 3^2, …
  - Example:
    - ✓ “ape” (h = 0): try 0, 0 + 1 – open!
    - ✓ “bear” (h = 1): try 1, 1 + 1, 1 + 4 – open!
    - ✓ “whale”?

- Advantage: reduces clustering

- Disadvantage: it may fail to find an existing open position.

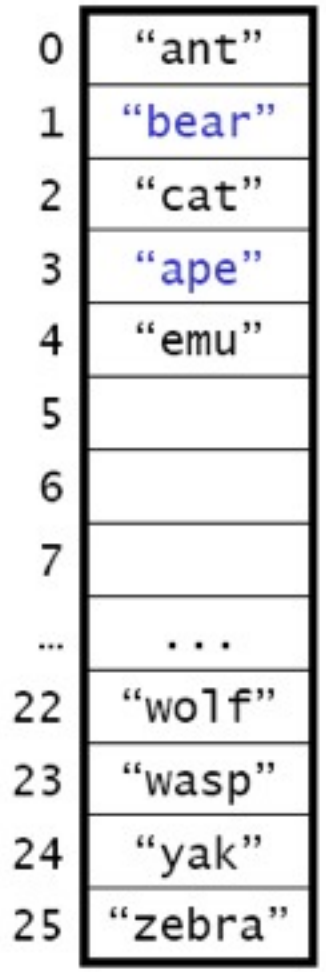
Example:
- table size = 10
- x = occupied
- trying to insert a key with h(key) = 0



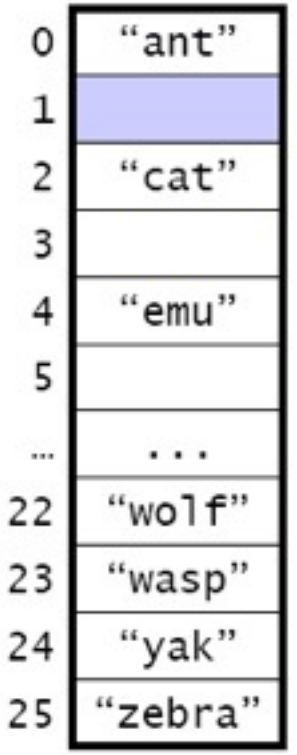


Double Hashing for Open Addressing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - Probe sequence: h1, h1 + h2, h1 + 2*h2, …
- Example:
  - h1 = our previous function h
  - h2 = number of characters in the string
  - “ape” (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - “bear” (h1 = 1, h2 = 4): try 1 – open!
  - “whale”?
- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - Theorem: will find an open position if there is one, provided the table size is a prime number.
  - Disadvantage: the need for two hash functions

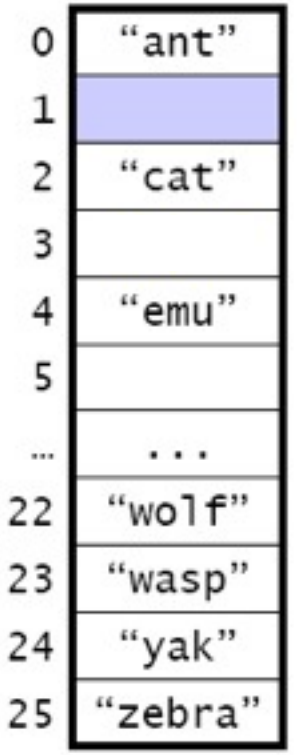


Removing Items with Open Addressing



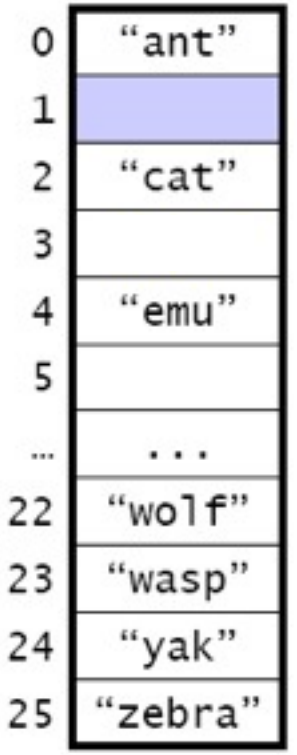
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing



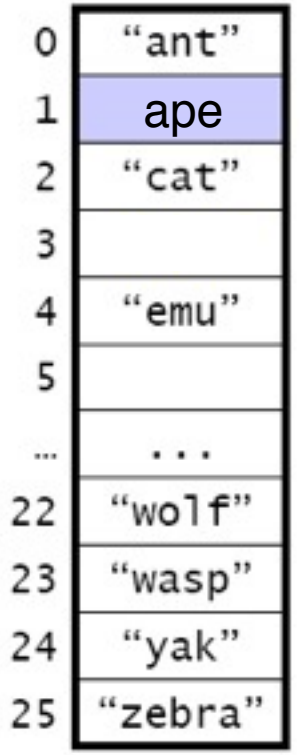
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!



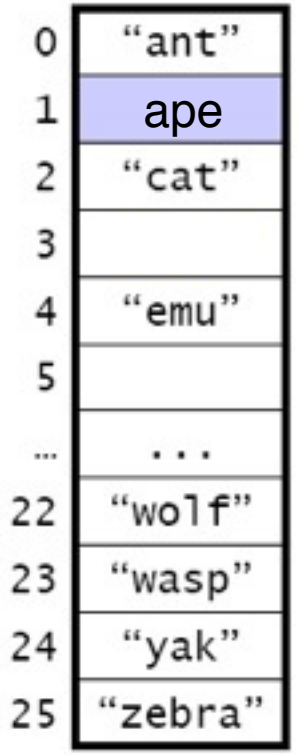
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!



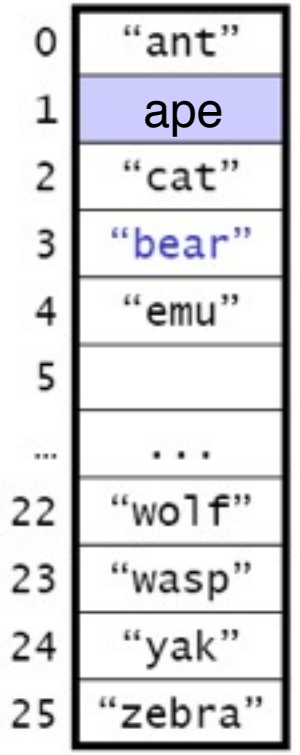
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!



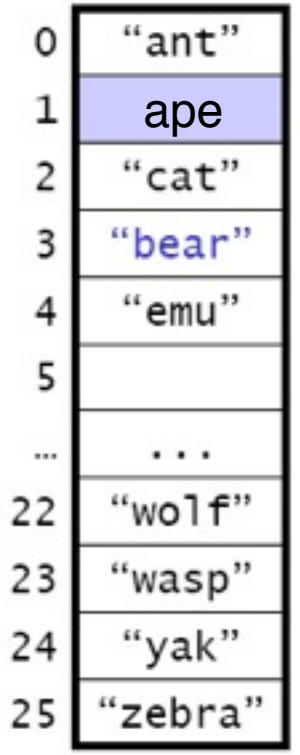
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!



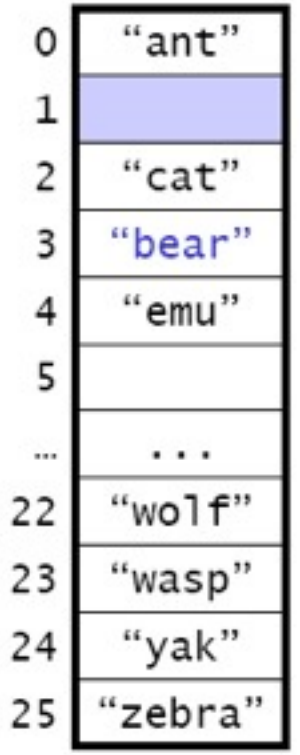
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”



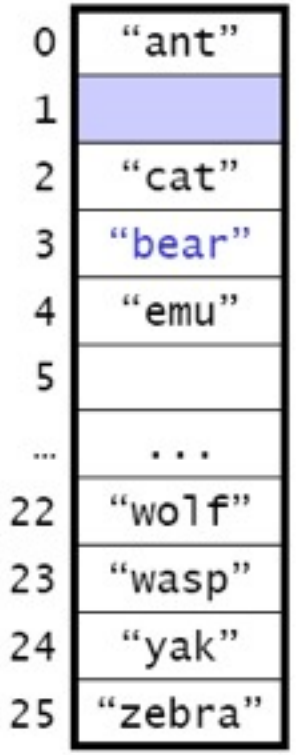
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”



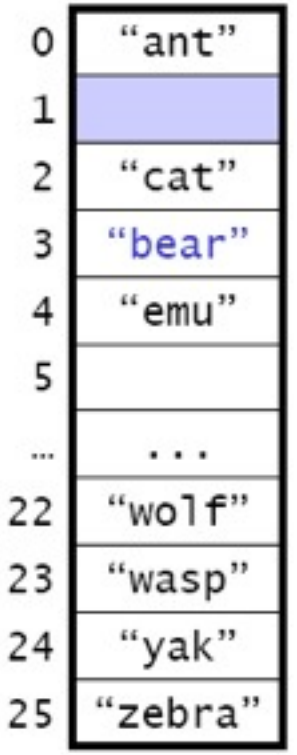
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item



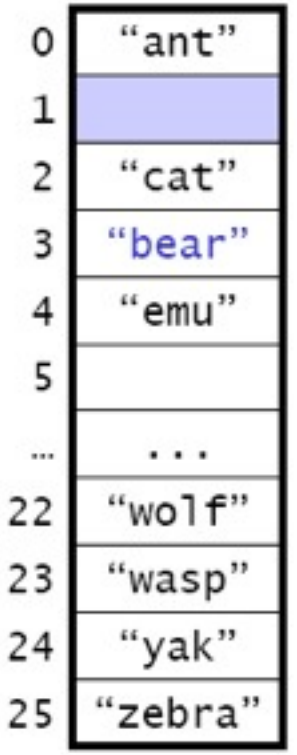
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,



Removing Items with Open Addressing

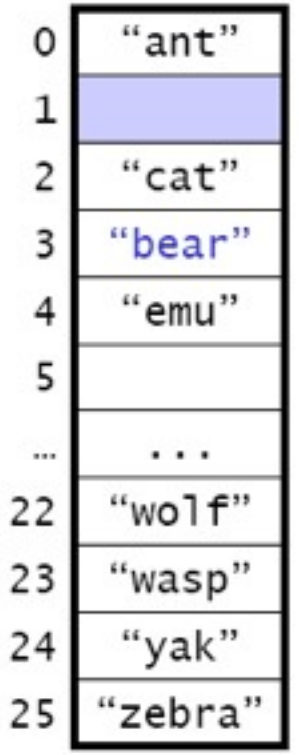
- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!



Removing Items with Open Addressing

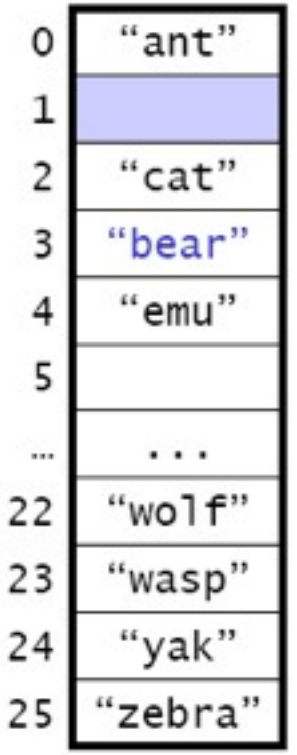
- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!

  - Cannot tell if it is not in the table



Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!
  - Cannot tell if it is not in the table



- Any difference with quadratic probing or double hashing?

Implementation of Hash Tables

- A simple hash table with open addressing.

```
public class HashTable {
        private class Entry {
                private String key;
                private boolean removed;
                private String etymology;
                …
        }
        private Entry[] table;
        private int tableSize;
        ...
}
```

- for an empty position, table[i] will equal null
- for a removed position, table[i] will refer to an Entry object whose *removed* field equals true
- for an occupied position, table[i] will refer to an Entry object whose *removed* field equals false

Constructors

- Initializing the hash table

```
public HashTable(int size) {
        table = new Entry[size];
        tableSize = size;
}
```

- Initializing an entry (before insertion)

```
private Entry(String key, String etymology) {
        this.key = key;
        this.etymology = etymology;
        removed = false;
        …
}
```

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
           removed)
        while ( ?                                                    )

              ?

        return i;
}
```

- Does it always terminate?

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
            removed)
        while (   table[i] != null && table[i].removed==false   )

                ?

        return i;
}
```

- Does it always terminate?

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
            removed)
        while (   table[i] != null && table[i].removed==false   )

                    i = (i + j) % tableSize;

        return i;
}
```

- Does it always terminate?

Finding An Open Position: Infinite Loops

- The loop in our probe method could become infinite.
  - When would this happen?

- To avoid infinite loops, we can stop probing after checking n positions (n = table size) because the probe sequence will just repeat after that point. Why?
  - for double hashing:
    - ✓ (h1 + n*h2) % n = h1 % n
    - ✓ (h1 + (n+1)*h2) % n = (h1 + n*h2 + h2) % n = (h1 + h2)%n
    - ✓ (h1 + (n+2)*h2) % n = (h1 + n*h2 + 2*h2) % n = (h1 + 2*h2)%n
    - ✓ …
  - for quadratic probing:
    - ✓ $(h1 + n^2) \% n = h1 \% n$
    - ✓ $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1)\%n$
    - ✓ $(h1 + (n+2)^2) \% n = (h1 + n^2 + 4n + 4) \% n = (h1 + 4)\%n$
    - ✓ …

Finding An Open Position: Infinite Loop Protection

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing until we get an empty or removed position
        while (table[i] != null && table[i].removed==false) {
                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return i;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while ( ?                                          ) {
                // return if key is found, otherwise continue
                ?


                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                          ) {
                // return if key is found, otherwise continue
                ?


                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                              ) {
              // return if key is found, otherwise continue
             if (table[i].key == key)
                 return i;

              i = (i + j) % tableSize;
              iterations++;
              if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                          ) {
                // return if key is found, otherwise continue
                ?


                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                              ) {
              // return if key is found, otherwise continue
             if (table[i].removed==false && table[i].key.equals(key))
                 return i;

              i = (i + j) % tableSize;
              iterations++;
              if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Search() Method

- Search for the entry with the key, and return the Etymology (in other implementation, we may return other fields instead)

```
public String search(String key) {
        int i = findKey(key);
        if (i == -1)
                return null;
        else
                return table[i].etymology ;
}
```

- It calls the helper method findKey() to locate the position of the key.

Remove() Method

- Search the hash table and delete the key if found

```
public void remove(String key) {
        int i = findKey(key);
        if (i == -1)
                return;
        table[i].removed = true;
}
```

- It also uses findKey().

Etc.

- “Hashing” to be continued!