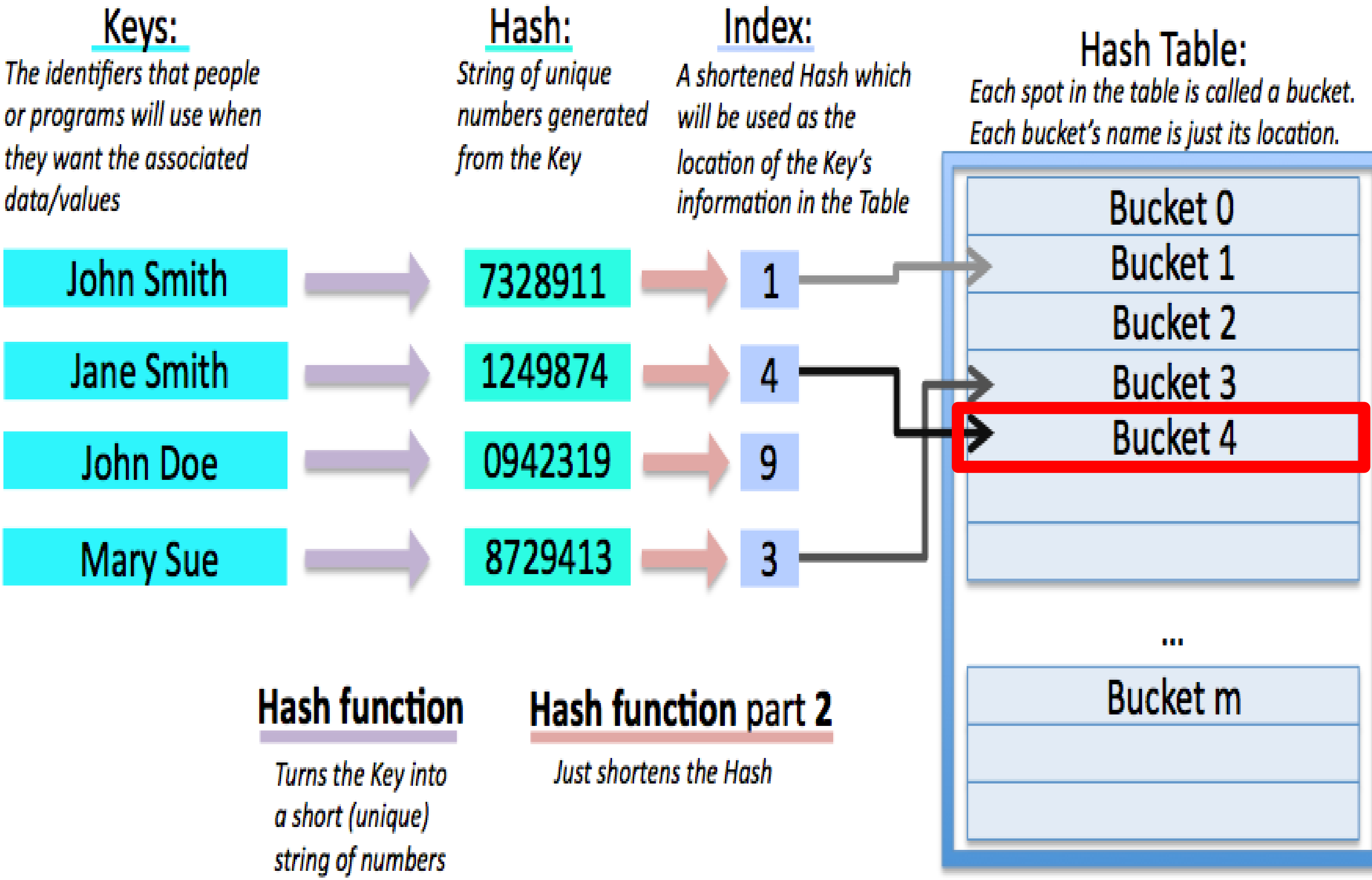
Hash Tables:
in a nutshell



Hash Functions

- A hash function defines a mapping from the set of possible keys to the set of integers.
- We then use the modulus operator to get a valid array index.

key value ⇨ **hash function** ⇨ integer $\xrightarrow{\%}$ integer in [0, n – 1]
(n = array length)

- Here's a very simple hash function for keys of lower-case letters:

  h(key) = ASCII value of first char – ASCII value of 'a'

  - examples:

    h("ant") = ASCII for 'a' – ASCII for 'a' = 0

    h("cat") = ASCII for 'c' – ASCII for 'a' = 2

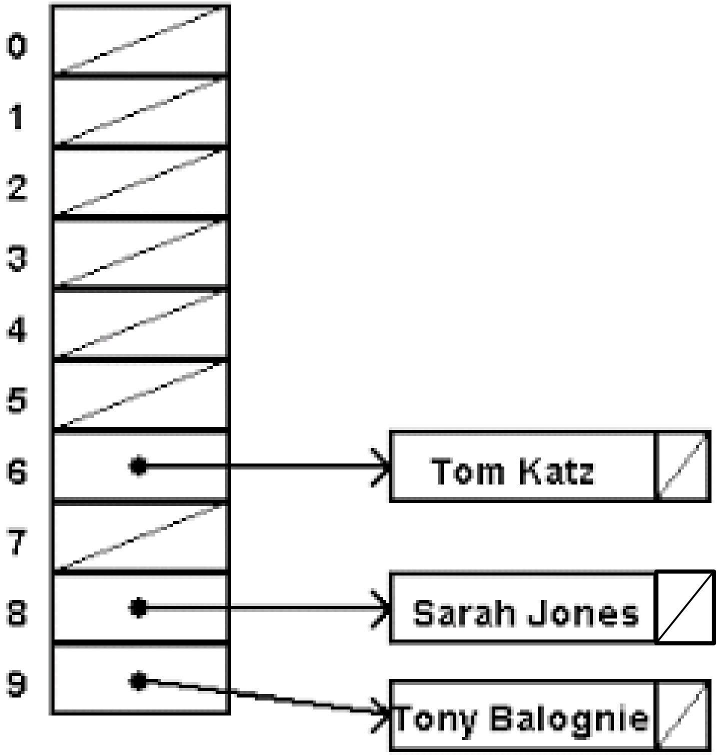
- h(key) is known as the key's *hash code.*
- A *collision* occurs when items with different keys are assigned the same hash code.

Removing Items Under 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 – conclude not in table
  - search for "bear": try 1 – conclude not in table, but "bear" is further down in the table!
- When we remove an item from a position, we need to leave something in that position to indicate that an item was removed.
- Three types of positions: occupied, empty, *removed*.
- We stop probing when we encounter an empty position, but not when we encounter a removed position.
  ex: search for "bear": try 1 (removed), 1 + 1, 1 + 2 – found!
- We can insert items in either empty or removed positions.

0	"ant"
1	
2	"cat"
3	"bear"
4	"emu"
5	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Hash Table with
Open Addressing



Operations on a Hash Table

- What operations should a Hash Table support?
  - insert
  - remove
  - search

- How can we ensure that our implementation supports these operations?

Interface

An Interface For Hash Tables

```
public interface HashTable {
    boolean insert(Object key, Object value);
    Queue<Object> search(Object key);
    Queue<Object> remove(Object key);
}
```

- `insert()` returns:
  - `true` if the key-value pair can be added
  - `false` if there is overflow and the pair cannot be added

- `search()` and `remove()` both return a queue containing all of the values associated with the specified key.
  - example: an index for a book
    - key = word
    - values = the pages on which that word appears
  - return `null` if the key is not found

An Interface For Hash Tables

```
public interface HashTable {
    boolean insert(Object key, Object value);
    Queue<Object> search(Object key);
    Queue<Object> remove(Object key);
}
```

- `insert()` returns:
  - `true` if the key-value pair can be added
  - `false` if there is overflow and the pair cannot be added
- `search()` and `remove()` both return a queue containing all of the values associated with the specified key.
  - example: an index for a book
    - key = word
    - values = the pages on which that word appears
  - return `null` if the key is not found

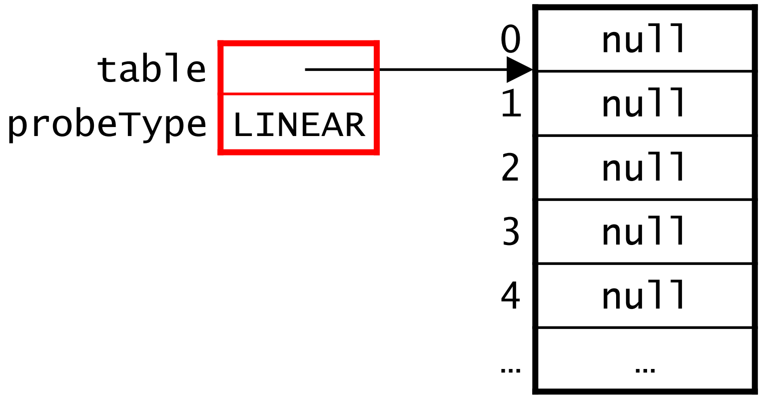
An Interface For Hash Tables

```
public interface HashTable {
    boolean insert(Object key, Object value);
    Queue<Object> search(Object key);
    Queue<Object> remove(Object key);
}
```

- **`insert`**`()` returns:
  - `true` if the key-value pair can be added
  - `false` if there is overflow and the pair cannot be added
- **`search`**`()` and **`remove`**`()` both return a queue containing all of the values associated with the specified key.
  - example: an index for a book
    - key = word
    - values = the pages on which that word appears
  - return `null` if the key is not found

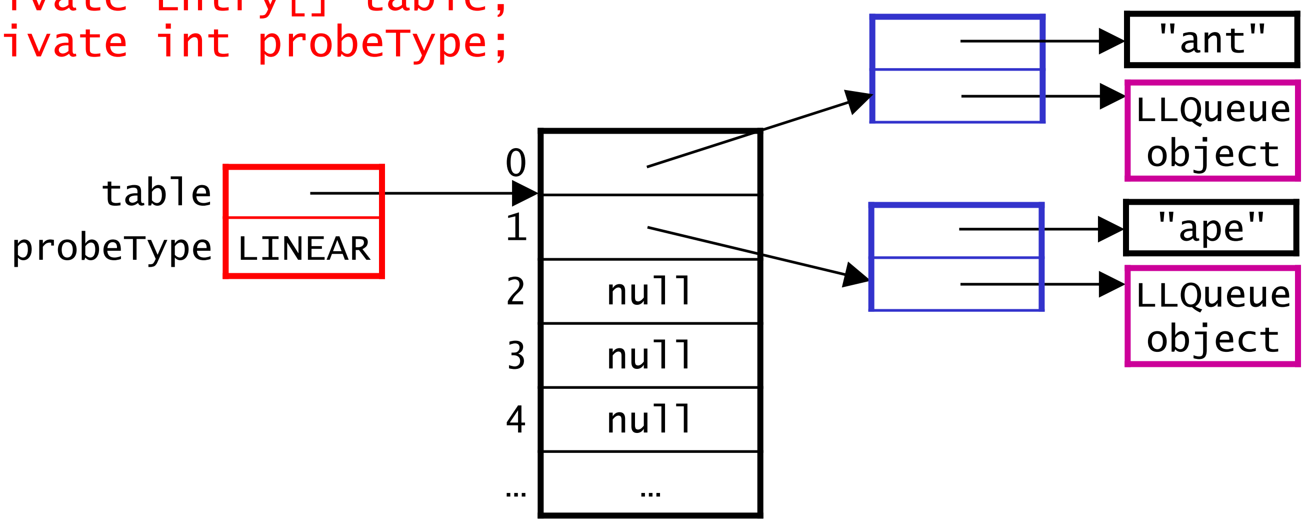
An Implementation Using Open Addressing

```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



An Implementation Using Open Addressing

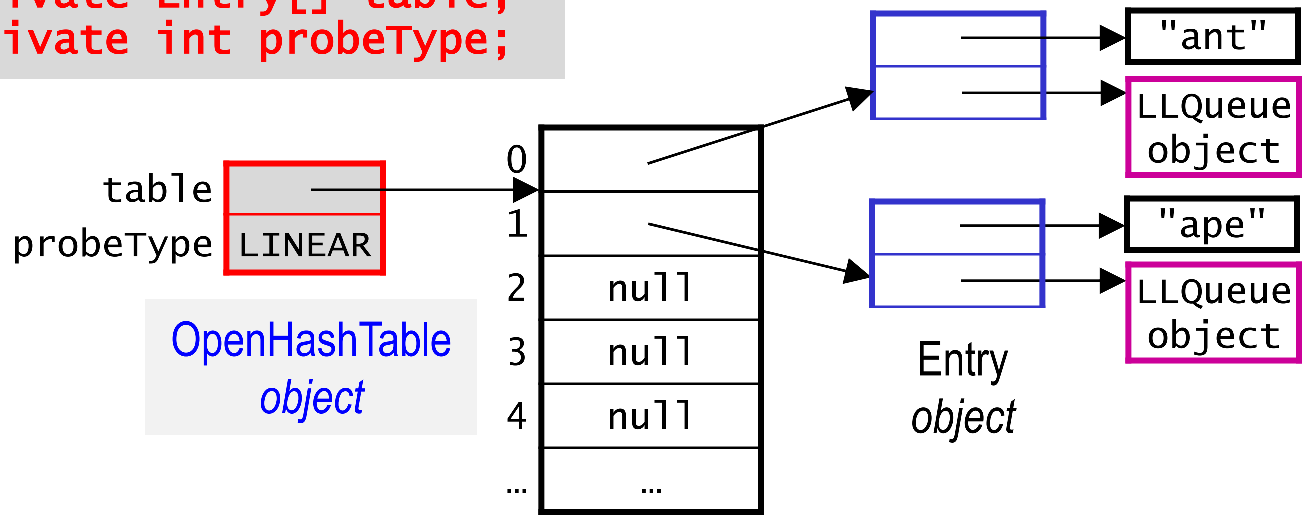
```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;                      // key
        private LLQueue<Object> values;          // value
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



- We use a private inner class for the entries in the hash table. Composed of:
  - a *reference to a* key
  - a *reference to* the value(s) associated with the key

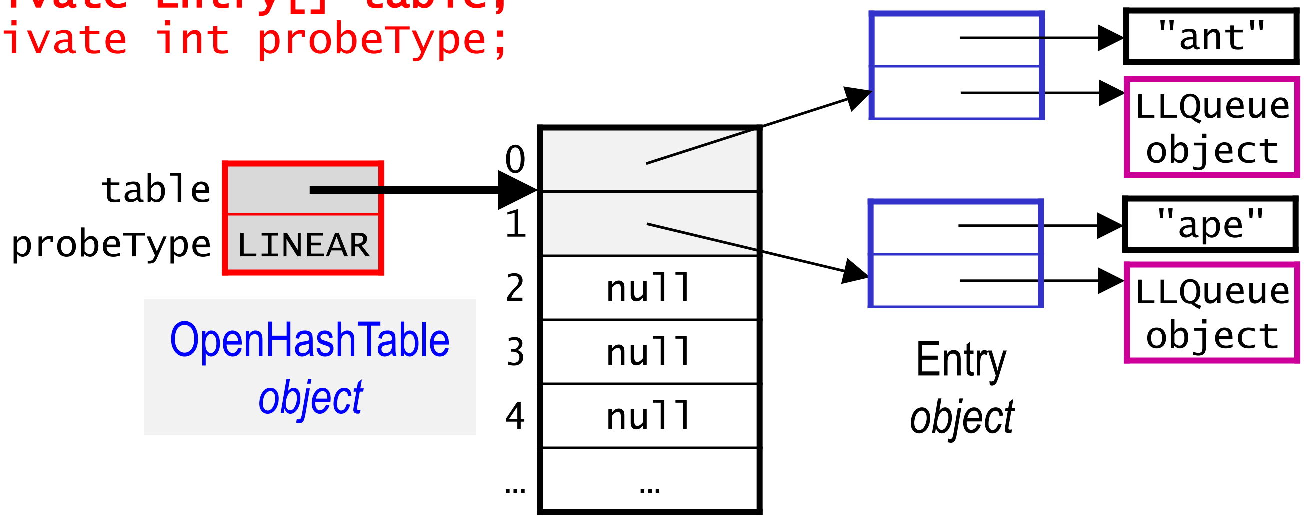
An Implementation Using Open Addressing

```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



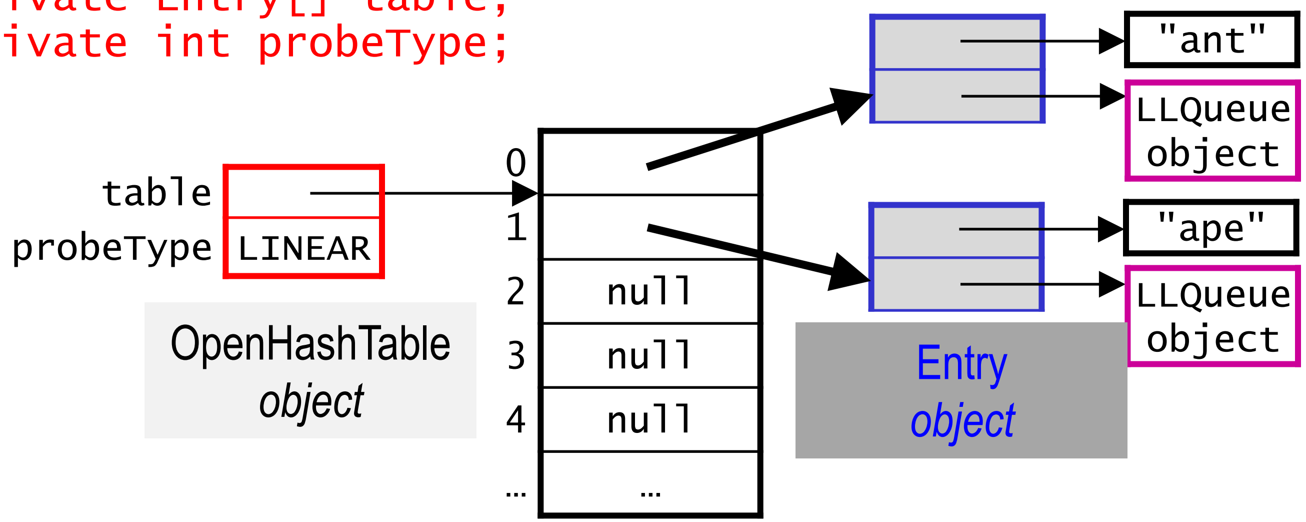
An Implementation Using Open Addressing

```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



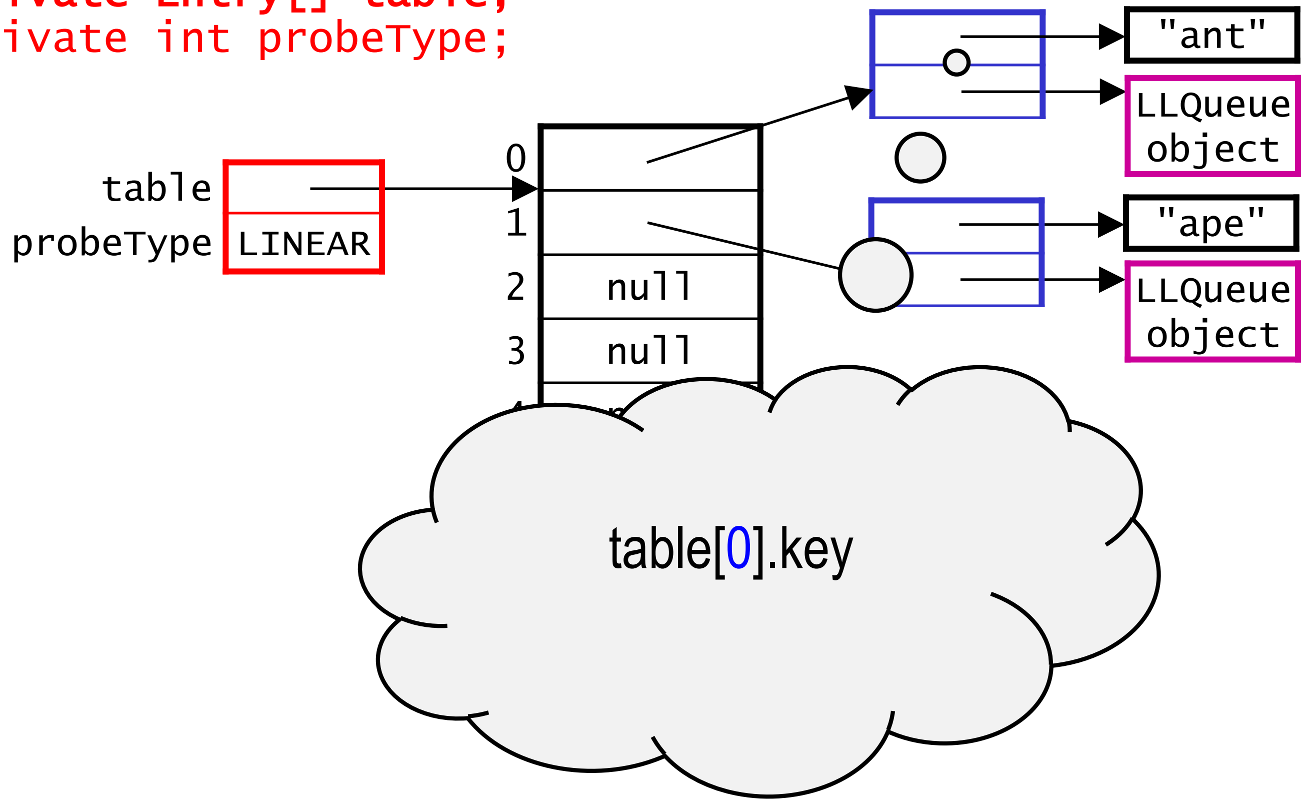
An Implementation Using Open Addressing

```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



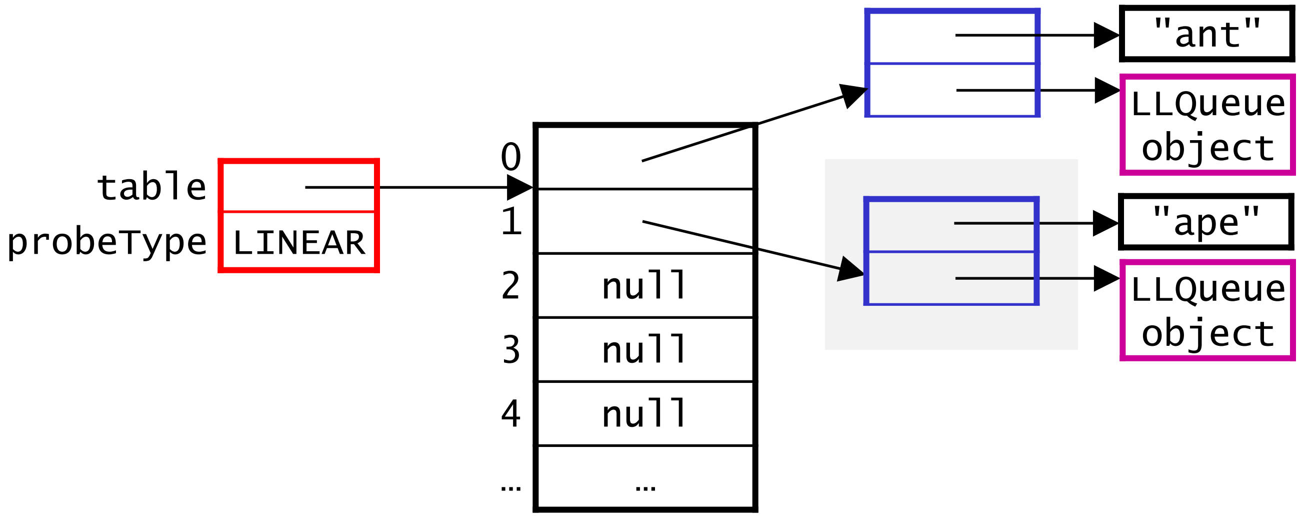
An Implementation Using Open Addressing

```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



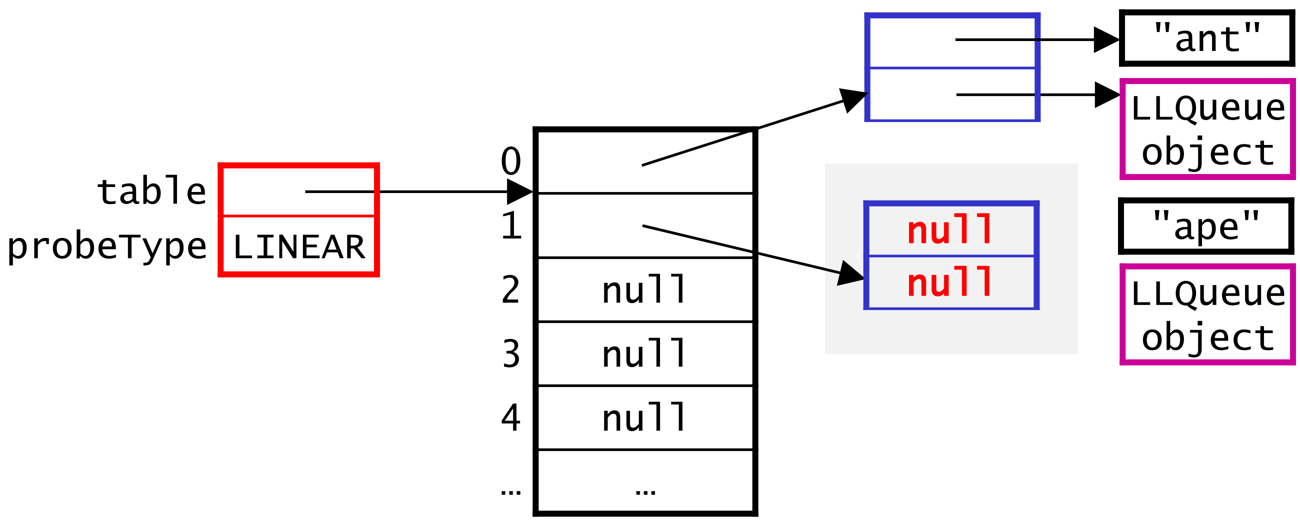
Empty vs. Removed

- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: `remove("ape")`:



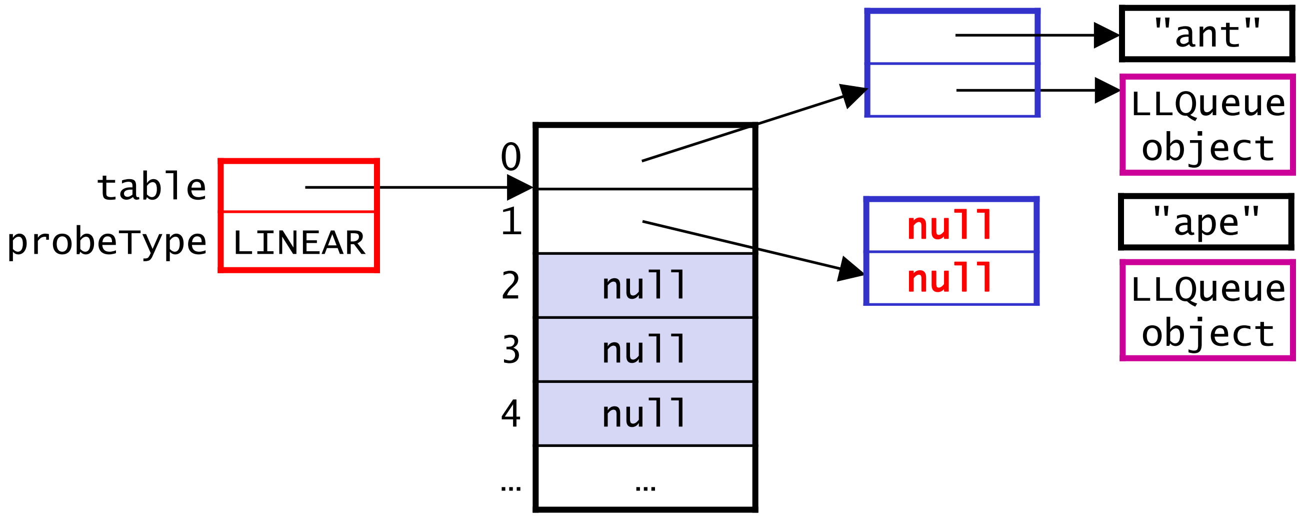
Empty vs. Removed

- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: **after** `remove("ape")`:



Empty vs. Removed

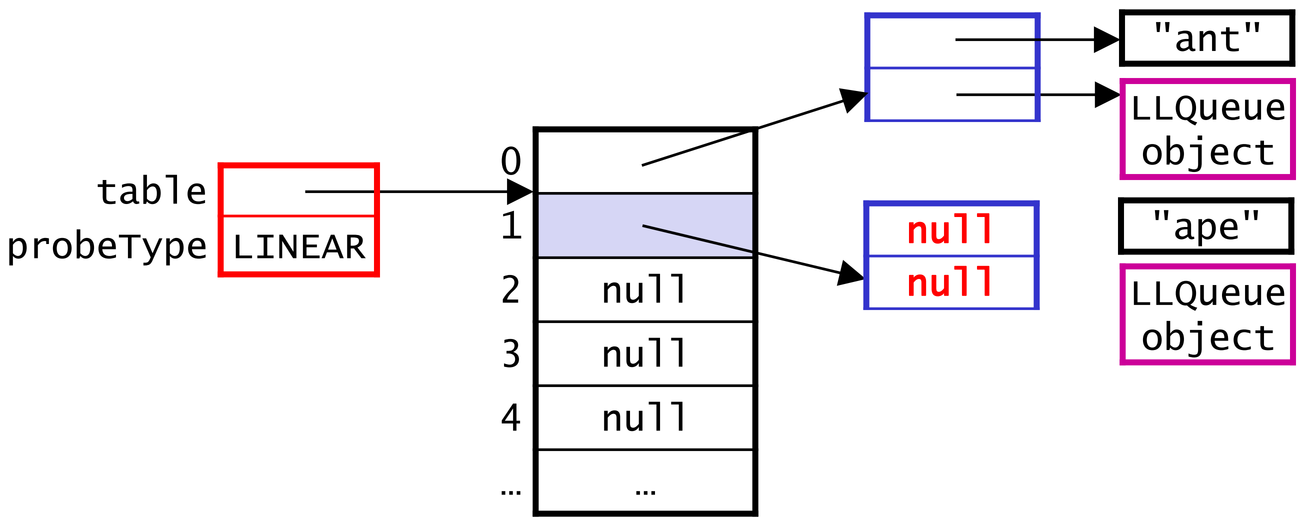
- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: after `remove("ape")`:



- Note the difference:
  - a truly empty position has a value of `null` in the table (example: positions 2, 3 and 4 above)

Empty vs. Removed

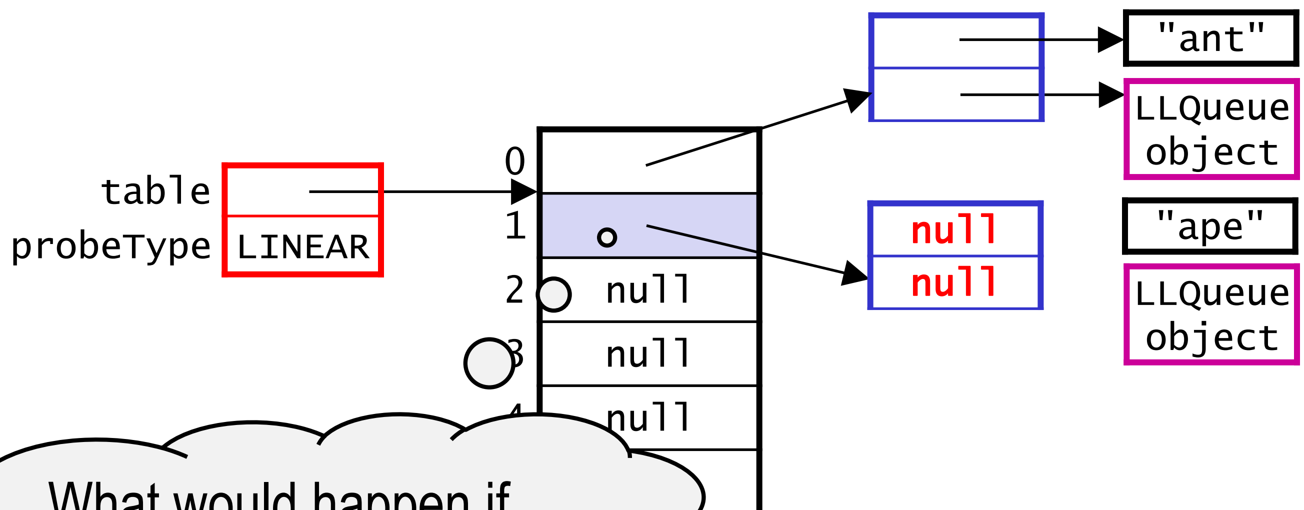
- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: after `remove("ape")`:

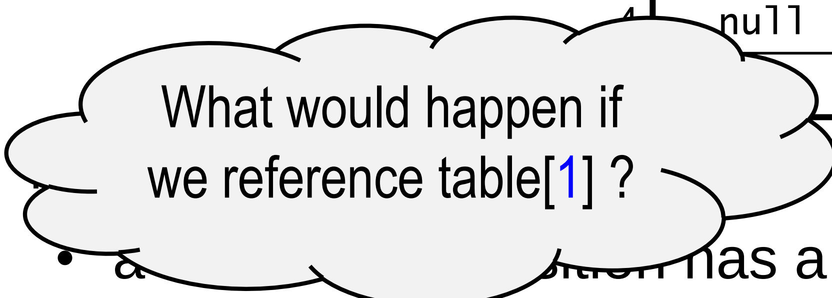


- Note the difference:
  - a truly empty position has a value of `null` in the table (example: positions 2, 3 and 4 above)
  - a removed position refers to an `Entry` object whose `key` and `values` fields are `null` (example: position 1 above)

Empty vs. Removed

- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: after `remove("ape")`:





-
  - a [illegible] has a value of `null` in the table (example: positions 2, 3 and 4 above)
  - a removed position refers to an `Entry` object whose `key` and `values` fields are `null` (example: position 1 above)

Probing Using Double Hashing:

instance method of OpenHashTable

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

    // keep probing until we get an empty position or match
    // (write this together)



    return i;
}
```

where `i` is the *index* into the hash table!

Probin… …shing

… where entry refers to an entry object in the hash table!

```
private int prob…
    int i = h1(k…                                   …tion
    int h2 = h2(key)…                                …function

    // keep probing until we get an empty position or match
    while (entry is not empty and
           search key does not equal entry key )) {
        probe to find the next open position
    }


    return i;
}
```

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }



    return i;
}
```

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }



    return i;
}
```

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }



    return i;
}
```

- It is essential that we:
  - check for `table[i] != null` first. why?
    if `table[i]` is `null`, the && operator short-circuits, which prevents a null pointer exception on `table[i].key`

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);      // first hash function
    int h2 = h2(key);     // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }

    return i;
}
```

- It is essential that we:
  - check for `table[i] != null` first. why?
    if `table[i]` is `null`, the && operator short-circuits,
    which prevents a null pointer exception on `table[i].key`
  - call the `equals` method on **`key`**, not on `table[i].key`. **why?**

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }



    return i;
}
```

- It is essential that we:
  - check for `table[i] != null` first. why?
    if `table[i]` is `null`, the && operator short-circuits, which prevents a null pointer exception on `table[i].key`
  - call the `equals` method on `key`, not `table[i].key`. why?
    because `table[i].key` may be `null` (for removed cells)

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;   // probe
    }



    return i;
}
```

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);      // first hash function
    int h2 = h2(key);     // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;   // probe
    }

    return i;
}
```

Note the mathematical simplification of:

- h1 + h2
- h1 + 2*h2 ….

by the use of i as an accumulator variable.

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function for offset

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }



    return i;
}
```

Any other potential problem
with our code?

Avoiding an Infinite Loop

- The while loop in our probe method could lead to an infinite loop.

```
while (table[i] != null && !key.equals(table[i].key)) {
    i = (i + h2) % table.length;
}
```

- When would this happen?
  if the key isn't in the table, and there are no empty positions

Avoiding an Infinite Loop

- The while loop in our probe method could lead to an infinite loop.

```
while (table[i] != null && !key.equals(table[i].key)) {
    i = (i + h2) % table.length;
}
```

- When would this happen?
  if the key isn't in the table, and there are no empty positions

- We can stop probing after checking n positions (n = table size), because the probe sequence will just repeat after that point.
  - for quadratic probing:
    $(h1 + n^2) \% n = h1 \% n$
    $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$
  - for double hashing:
    $(h1 + n*h2) \% n = h1 \% n$
    $(h1 + (n+1)*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2) \% n$

Avoiding an Infinite Loop (cont.)

```
private int probe(Object key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + h2) % table.length;
        numChecked++;
    }

    return i;
}
```

Avoiding an Infinite Loop (cont.)

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + h2) % table.
        numChecked++;
    }

    return i;
}
```

…or assign -1 to variable i and break out of the loop!

Handling the Other Types of Probing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + h2) % table.length;
        numChecked++;
    }

    return i;
}
```

Note that this limits the probe to double hashing. We can generalize by…

Handling the Other Types of Probing

```
private int probe(Object key) {
    int i = h1(key);      // first hash function
    int h2 = h2(key);     // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }

    return i;
}
```

calling a private method that calculates the probe increment.

Handling the Other Types of Probing (cont.)

- The `probeIncr()` method bases the increment on the type of probing:

```
private int probeIncr(int numChecked, int h2) {
    int increment;

    if (probeType == LINEAR) {
        increment = 1;
    } else if (probeType == QUADRATIC) {
        increment = (2*numChecked - 1);
    } else {   //  DOUBLE_HASHING:
        increment = h2;
    }

     return( increment );
}
```

Handling the Other Types of Probing (cont.)

- The `probeIncr()` method bases the increment on the type of probing:

```
private int probeIncr(int numChecked, int h2) {
    int increment;

    if (probeType == LINEAR) {
        increment = 1;
    } else if (probeType == QUADRATIC) {
        increment = (2*numChecked - 1);
    } else {   //  DOUBLE_HASHING:
        increment = h2;
    }

     return( increment );
}
```

Handling the Other Types of Probing (cont.)

- The `probeIncr()` method bases the increment on the type of probing:

```
private int probeIncr(int numChecked, int h2) {
    int increment;

    if (probeType == LINEAR) {
        increment = 1;
    } else if (probeType == QUADRATIC) {
        increment = (2*numChecked - 1);
    } else {   //  DOUBLE_HASHING:
        increment = h2;
    }

     return( increment );
}
```

Handling the Other Types of Probing (cont.)

- Why is `2*numChecked - 1`
  the correct increment for quadratic probing?

- Recall that for quadratic probing:
  - probe sequence = h1, h1 + 1^2, h1 + 2^2, …
  - jth index in the sequence = h1 + j^2 (note: j = numChecked)

- The increment used to compute the jth index
  = jth index – (j – 1)st index
  = (h1 + j^2) – (h1 + $(j – 1)^2$)
  = $j^2 – (j – 1)^2$
  = $j^2 – (j^2 – 2j + 1)$
  = 2**j** – 1

`numChecked`

Handling the Other Types of Probing (cont.)

- The `probeIncr()` method bases the increment on the type of probing:

```
private int probeIncr(int numChecked, int h2) {
    int increment;

    if (probeType == LINEAR) {
        increment = 1;
    } else if (probeType == QUADRATIC) {
        increment = (2*numChecked - 1);
    } else {   //  DOUBLE_HASHING:
        increment = h2;
    }

     return( increment );
}
```

Handling the Return

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }

    return i;
}
```

Note the possible values can ***be returned***.

Handling the Return

```java
private int probe(Object key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }

    return i;
}
```

If `i` is valid, what possible values can `table[i]` contain?

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLQueue<Object> search(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}

public LLQueue<Object> remove(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLQueue<Object> search(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}
```

key was not found or entry is ***empty***!

```
public LLQueue<Object> remove(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLQueue<Object> search(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}
```

key was found and the reference to the queue of values associated with the key is returned.!

```
public LLQueue<Object> remove(Ob
    int i = probe(key);
    if (i == -1 || table[i] == null)
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLQueue<Object> search(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}
```

```
public LLQueue<Object> remove(Ob
    int i = probe(key);
    if (i == -1 || table[i] == nu
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Assign the queue of values to a local reference that is returned by the method.

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLQueue<Object> search(Object key) {
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}
```

```
public LLQueue<Object> remove(Ob
    int i = probe(key);
    if (i == -1 || table[i] == null)
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;       // mark it as a removed cell
    table[i].values = null;
    return removedVals;
}
```

We remove the entry by assigning a null value to the key:value references of the entry object.

Insertion

- We begin by probing for the key.
- Several cases:
  1. the key is already in the table (we're inserting a duplicate)
     → add the value to the values in the key's `Entry`
  2. the key is not in the table: three subcases:
     a. encountered 1 or more removed positions while probing
        → put the (key, value) pair in the *first* removed position seen during probing. why?
        we'll find it sooner in subsequent searches
     b. no removed position; reached an empty position
        → put the (key, value) pair in the empty position
     c. no removed position or empty position
        → overflow; return `false`

Insertion (cont.)

- To handle the special cases, we give this method its own implementation of probing:

```
public boolean insert(Object key, Object value) {
    int i = h1(key);
    int h2 = h2(key);
    int numChecked = 1;
    int firstRemoved = -1; // saves first removed position

    while (table[i] != null && !key.equals(table[i].key)) {
        if (table[i].key == null && firstRemoved == -1) {
            firstRemoved = i;
        }
        if (numChecked == table.length) {
            break;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }
    // deal with the different cases (see next slide)
}
```

- ***firstRemoved*** remembers the first removed position that we see

Insertion (cont.)

- To handle the special cases, we give this method its own implementation of probing:

```
public boolean insert(Object key, Object value) {
    int i = h1(key);
    int h2 = h2(key);
    int numChecked = 1;
    int firstRemoved = -1; // saves first removed position

    while (table[i] != null && !key.equals(table[i].key)) {
        if (table[i].key == null && firstRemoved == -1) {
            firstRemoved = i;
        }
        if (numChecked == table.length) {
            break;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }
    // deal with the different cases (see next slide)
}
```

- *firstRemoved* remembers the first removed position that we see

Insertion (cont.)

- To handle the special cases, we give this method its own implementation of probing:

```
public boolean insert(Object key, Object value) {
    int i = h1(key);
    int h2 = h2(key);
    int numChecked = 1;
    int firstRemoved = -1; // saves first removed position

    while (table[i] != null && !key.equals(table[i].key)) {
        if (table[i].key == null && firstRemoved == -1) {
            firstRemoved = i;
        }
        if (numChecked == table.length) {
            break;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }
    // deal with the different cases (see next slide)
}
```

Insertion (cont.)

- To handle the special cases, we give this method its own implementation of probing:

```
public boolean insert(Object key, Object value) {
    int i = h1(key);
    int h2 = h2(key);
    int numChecked = 1;
    int firstRemoved = -1;

    while (table[i] != null && !key.equals(table[i].key)) {
        if (table[i].key == null && firstRemoved == -1) {
            firstRemoved = i;
        }
        if (numChecked == table.length) {
            break;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }
    // deal with the different cases (see next slide)
}
```

Insertion (cont.)

```
public boolean insert(Object key, Object value) {
    ...
    int firstRemoved = -1;
    while (table[i] != null && !key.equals(table[i].key) {
        if (table[i].key == null && firstRemoved == -1) {
            firstRemoved = i;
        }
        if (numChecked == table.length) {
            break;
        }
        i = (i + probeIncr(numChecked, h2)) % table.length;
        numChecked++;
    }
    boolean inserted = true;

    if (table[i] != null && key.equals(table[i].key)) { // 1
        table[i].values.insert(value);
    } else if (firstRemoved != -1) {                    // 2a
        table[firstRemoved] = new Entry(key, value);
    } else if (table[i] == null) {                      // 2b
        table[i] = new Entry(key, value);
    } else {                                            // 2c
        inserted = false;
    }
    return inserted;
```

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison"

  - insert "cow"

0	"ant"
1	**"bear"**
2	"cat"
3	
4	"emu"
5	"fox"
6	
7	
8	
9	
10	

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, **(1 + 5) % 26 – open!**
  - insert "cow"

0	"ant"
1	"bear"
2	"cat"
3	
4	"emu"
5	"fox"
6	**"bison"**
7	
8	
9	
10	

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, **(2 + 2*3) % 26 – open!**

0	"ant"
1	"bear"
2	"cat"
3	
4	"emu"
5	"fox"
6	"bison"
7	
8	"cow"
9	
10	

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found!

0	"ant"
1	"bear"
2	"cat"
3	
4	"emu"
5	"fox"
6	"bison"
7	
8	"cow"
9	
10	

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2
- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found! **make it a removed cell**

0	"ant"
1	"bear"
2	"cat"
3	
4	
5	"fox"
6	"bison"
7	
8	"cow"
9	
10	

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5): try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3): try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3): try 4 – found! make it a removed cell
  - search "eel" (h1 = 4, h2 = 3): **try 4 (can't stop at removed)**

0	"ant"
1	"bear"
2	"cat"
3	
4	
5	"fox"
6	"bison"
7	
8	"cow"
9	
10	

Tracing Through Some Examples

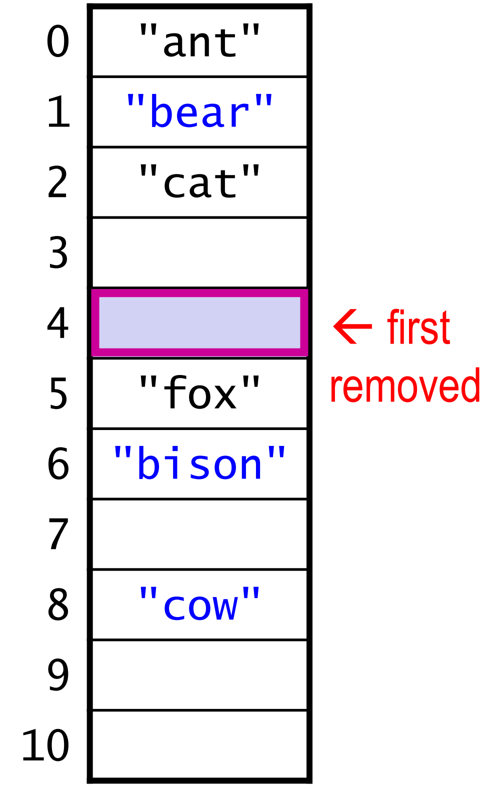
- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found! make it a removed cell
  - search "eel" (h1 = 4, h2 = 3):
    try 4 (can't stop at removed), **(4+3)%26–empty, so not found**

0	"ant"
1	"bear"
2	"cat"
3	
4	
5	"fox"
6	"bison"
7	
8	"cow"
9	
10	

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2



- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found! make it a removed cell
  - search "eel" (h1 = 4, h2 = 3):
    try 4 (can't stop at removed), (4 + 3) % 26 – empty, so not found
  - **insert "bee". in which position will it go?**
    (h1 = 1, h2 = 3): try 1, **(1 + 3) % 26**

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found! make it a removed cell
  - search "eel" (h1 = 4, h2 = 3):
    try 4 (can't stop at removed), (4 + 3) % 26 – empty, so not found
  - **insert "bee". in which position will it go?**
    (h1 = 1, h2 = 3): try 1, (1 + 3) % 26, **(1 + 2*3) % 26 (empty)**

0	"ant"	
1	"bear"	
2	"cat"	
3		
4		← first removed
5	"fox"	
6	"bison"	
7		
8	"cow"	
9		
10		

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found! make it a removed cell
  - search "eel" (h1 = 4, h2 = 3):
    try 4 (can't stop at removed), (4 + 3) % 26 – empty, so not found
  - **insert "bee". in which position will it go?**
    (h1 = 1, h2 = 3): try 1, (1 + 3) % 26, (1 + 2*3) % 26 (empty)

0	"ant"	
1	"bear"	
2	"cat"	
3		
4	"bee"	← first removed
5	"fox"	
6	"bison"	
7		
8	"cow"	
9		
10		

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2

- Perform the following operations:
  - insert "bear" (h1 = 1, h2 = 4): try 1 – open!
  - insert "bison" (h1 = 1, h2 = 5):
    try 1, (1 + 5) % 26 – open!
  - insert "cow" (h1 = 2, h2 = 3):
    try 2, (2 + 3) % 26, (2 + 2*3) % 26 – open!
  - delete "emu" (h1 = 4, h2 = 3):
    try 4 – found! make it a removed cell
  - search "eel" (h1 = 4, h2 = 3):
    try 4 (can't stop at removed), (4 + 3) % 26 – empty, so not found
  - **insert "bee". in which position will it go?**

    A. 1 B. 3 C. **4** D. 7

0	"ant"	
1	"bear"	
2	"cat"	
3		
4	**"bee"**	← **first removed**
5	"fox"	
6	"bison"	
7		
8	"cow"	
9		
10		

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing: when all positions are occupied
  - quadratic probing:
    - when all positions are occupied
    - when the probe sequence can't reach the unoccupied slots
  - double hashing:
    - if the table size is a prime number: same as linear
    - if the table size is not a prime number: same as quadratic
- To avoid overflow (*and reduce search times*), grow the hash table when the % of occupied positions gets too big.

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing: ... are occupied
  - quadratic pro...
    - when a...
    - when th... reach the unoccupied slots
  - double has...
    - if the table s... ...e numb... same as linear
    - if the table size is not a prime numb...: same as quadratic

Does this present a special challenge for hash tables?

- To avoid overflow (*and reduce search times*), grow the hash table when the % of occupied positions gets too big.

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing: ... are occupied
  - quadratic pro...
    - when a...
    - when th... reach the unoccupied slots
  - double hash...
    - if the table s... ...e numb... same as linear
    - if the table size is not a prime numb...: same as quadratic

Consider the probing sequence that we rely on?

- To avoid overflow (*and reduce search times*), grow the hash table when the % of occupied positions gets too big.

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing: when all positions a
  - quadratic probing:
    - when all positions are occupie
    - when the probe sequence can't rea upied slots
  - double hashing:
    - if the table size is a prime number: sa as linear
    - if the table size is not a prime number: same as quadratic
- To avoid overflow (*and reduce search times*), grow the hash table when the % of occupied positions gets too big.
  - problem: we need to rehash **all** of the existing items. why?
    items may end up in different positions in the larger table

Rehash all buckets into a new hash table!

The Hash Function

KEY

$$hash(C) = \left[\sum_{i=0}^{m-1} c_i \times 31^{(m-i-1)}\right] mod\ 2^{32}$$

$$= [c_0 \times 31^{m-1} + c_1 \times 31^{m-2} + \cdots + c_{m-1} \times 31^0] mod\ 2^{32}$$

Hash Table Index

Implementing the Hash Function

- Characteristics of a good hash function:
  1) efficient to compute
  2) uses the entire key
     - changing any char/digit/etc. should change the hash code
  3) distributes the keys more or less uniformly across the table
  4) must be a function!
     - a key must always get the same hash code

Implementing the Hash Function

- Characteristics of a good hash function:
  1) efficient to compute
  2) uses the entire key
     - changing any char/digit/etc. should change the hash code
  3) distributes the keys more or less uniformly across the table
  4) must be a function!
     - a key must always get the same hash code

- In Java, every object has a `hashCode()` method.
  - the version inherited from `Object` returns a value based on an object's memory location
  - classes can override this version with their own

Hash Functions in `OpenHashTable`

- Initial hash function: returns a value in `[0, table.length - 1]`

```
public int h1(Object key) {
    int h1 = key.hashCode() % table.length;
    if (h1 < 0) {
        h1 += table.length;
    }
    return h1;
}
```

Hash Functions in `OpenHashTable`

- Initial hash function: returns a value in `[0, table.length - 1]`

```
public int h1(Object key) {
    int h1 = key.hashCode() % table.length;
    if (h1 < 0) {
        h1 += table.length;
    }
    return h1;
}
```

- Second hash function (for double hashing):

```
public h2(Object key) {
    int h2 = key.hashCode() % 5;
    if (h2 < 0) {
        h2 += 11;
    }
    h2 += 5;
    return h2;
}
```

  - 5 and 11 are values that could be adjusted as needed
  - provide a range of possible increments >= 5

Hash Functions for Strings: version 1

- h_a = the sum of the characters' ASCII values
- Example: h_a("eat") = 101 + 97 + 116 = 314
- All permutations of a given set of characters get the same code.
  - example: h_a("tea") = h_a("eat")
  - could be useful in a Scrabble game
    - allow you to look up all words that can be formed from a given set of characters
- The range of possible hash codes is very limited.
  - example: hashing keys composed of 1-5 lower-case char's (padded with spaces)
  - 26*27*27*27*27 = over 13 million possible keys
  - smallest code = h_a("a   ") = 97 + 4*32 = 225
    largest code = h_a("zzzzz") = 5*122 = 610

610 – 225
= 385 codes

Hash Functions for Strings: version 2

- Compute a *weighted* sum of the ASCII values:

$$h_b = a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b + a_{n-1}$$

where a_i = ASCII value of the ith character
b = a constant
n = the number of characters

- Multiplying by powers of b allows the *positions* of the characters to affect the hash code.
  - different permutations get different codes
- We may get arithmetic overflow, and thus the code may be negative. We adjust it when this happens.
- Java uses this hash function with $b = 31$ in the `hashCode()` method of the `String` class.

Hash Table Efficiency

- In the best case, search and insertion are $O(1)$.
- In the worst case, search and insertion are linear.
  - open addressing: $O(m)$, where m = the size of the hash table
  - separate chaining: $O(n)$, where n = the number of keys
- With good choices of hash function and table size, complexity is generally better than $O(\log n)$ and approaches $O(1)$.
- ***load factor* = # keys in table / size of the table.** To prevent performance degradation:
  - open addressing: try to keep the load factor < 1/2
  - separate chaining: try to keep the load factor < 1
- Time-space tradeoff: bigger tables have better performance, but they use up more memory.

Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can't easily:
  - print the contents in sorted order
  - perform a range search
  - perform a rank search – get the kth largest item

  We *can* do all of these things with a search tree.

Dictionaries in Java's Class Library

- Java provides a generic interface for dictionaries:

```
public interface Map<K, V> {
    ...
}
```

  - `K` is the type of the keys
  - `V` is the type of the values

- It differs somewhat from our dictionary implementations:
  - `insert()` is called `put()`
  - `search()` is called `get()`
  - it does *not* support duplicates
    - to have multiple values for a given key, the client can use a list as the key's value

- The implementations of `Map<K, V>` include:
  - **`TreeMap<K, V>` - uses a balanced seach tree**
  - `HashMap<K, V>` - uses a hash table with separate chaining

Dictionaries in Java's Class Library

- Java provides a generic interface for dictionaries:

```
public interface Map<K, V> {
    ...
}
```

  - `K` is the type of the keys
  - `V` is the type of the values

- It differs somewhat from our dictionary implementations:
  - `insert()` is called `put()`
  - `search()` is called `get()`
  - it does *not* support duplicates
    - to have multiple values for a given key, the client can use a list as the key's value

- The implementations of `Map<K, V>` include:
  - `TreeMap<K, V>` - uses a balanced seach tree
  - **`HashMap<K, V>` - uses a hash table with separate chaining**

Play Time

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells
- **What is the probe sequence for "baboon"?**
  (the sequence of positions seen during probing)

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. 1, 7, 2, 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- **What is the probe sequence for "baboon"?**
  (h1 = 1, h2 = 6)

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. 1, 7, 2, 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

- **What is the probe sequence for "baboon"?**

(h1 = 1, h2 = 6)   try: 1 % 11 = 1

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. 1, 7, 2, 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- **What is the probe sequence for "baboon"?**

(h1 = 1, h2 = 6)   try: 1 % 11 = 1

(1 + 6) % 11 = 7

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. 1, 7, 2, 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- **What is the probe sequence for "baboon"?**

(h1 = 1, h2 = 6) try: 1 % 11 = 1

(1 + 6) % 11 = 7

(1 + 2*6) % 11 = 2

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. 1, 7, 2, 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- **What is the probe sequence for "baboon"?**

(h1 = 1, h2 = 6) try: 1 % 11 = 1
(1 + 6) % 11 = 7
(1 + 2*6) % 11 = 2
(1 + 3*6) % 11 = 8

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. 1, 7, 2, 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- **What is the probe sequence for "baboon"?**

(h1 = 1, h2 = 6) try: 1 % 11 = **1**

(1 + 6) % 11 = **7**

(1 + 2*6) % 11 = **2**

(1 + 3*6) % 11 = **8**

empty cell, so stop probing

A. 1, 2, 5

B. 1, 6

C. 1, 7, 2

D. 1, 7, 3

E. **1, 7, 2, 8**

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- What is the probe sequence for "baboon"?

(h1 = 1, h2 = 6) try: 1 % 11 = 1

(1 + 6) % 11 = 7

(1 + 2*6) % 11 = 2

(1 + 3*6) % 11 = 8

- **If we insert "baboon", in what position will it go?**

A. 1 B. 7 C. 2 D. 8

Another Example

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

0	"ant"
1	"baboon"
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- What is the probe sequence for "baboon"?

(h1 = 1, h2 = 6) try: 1 % 11 = 1
(1 + 6) % 11 = 7
(1 + 2*6) % 11 = 2
(1 + 3*6) % 11 = 8

- **If we insert "baboon", in what position will it go?**

A. **1** B. 7 C. 2 D. 8

the first *removed* position seen while probing