Implementing the Map ADT

Outline

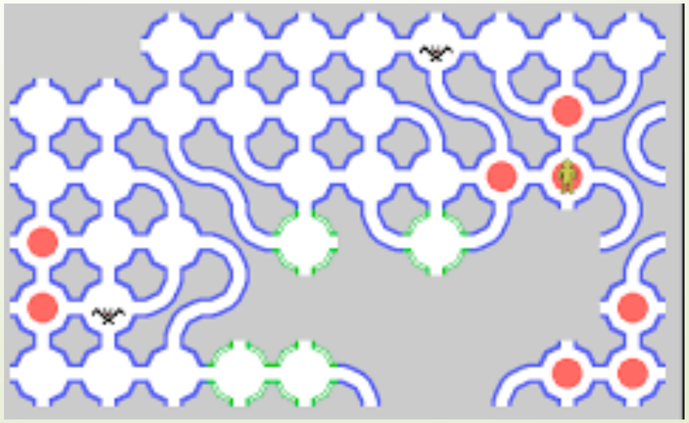
- The Map ADT
- Implementation with Java Generics
- A Hash Function
  - translation of a string key into an integer
- Consider a few strategies for implementing a hash table
  - linear probing
  - quadratic probing
  - separate chaining hashing
  - OrderedMap using a binary search tree

The Map ADT

- A Map models a searchable collection of key-value mappings
- A key is said to be “mapped” to a value
- Also known as: dictionary, associative array
- Main operations: insert, find, and delete

Applications

- Store large collections with fast operations
  - For a long time, Java only had Vector (think ArrayList), Stack, and Hashmap   (now there are about 67)
- Support certain algorithms
  - for example, probabilistic text generation in 127B
- Store certain associations in meaningful ways
  - For example, to store connected rooms in Hunt the Wumpus in 335



The Map ADT

- A value is "mapped" to a unique key
- Need a key *and* a value to insert new mappings
- Only need the key to find mappings
- Only need the key to remove mappings

Key and Value

- With Java generics, you need to specify
  - the type of key
  - the type of value
- Here the key type is **`String`** and the value type is **`BankAccount`**

```
Map<String, BankAccount> accounts
     = new HashMap<String, BankAccount>();
```

put(key, value)   get(key)

- Add new mappings (a key mapped to a value):

```
Map<String, BankAccount> accounts = new
                    TreeMap<String, BankAccount>();
accounts.put("M",);
accounts.put("G", new BankAcnew BankAccount("Michel",
111.11)count("Georgie", 222.22));
accounts.put("R", new BankAccount("Daniel", 333.33));

BankAccount current = accounts.get("M");
assertEquals(111.11, current.getBalance(), 0.001);
assertEquals("Michel", current.getID());
current = accounts.get("R");
  // What is current.getID()? _______________
  // What is current.getBalance()? __________
```

keys must be unique

- `put` returns replaced value if key existed
  - In this case, the mapping now has the same key mapped to a new value
  - or returns null if the key does not exist

```
assertNull(ranking.put("newKey",
                new BankAccount("newID", 444.44)));

// "newKey" is already in the map
assertNotNull(ranking.put("R", current));
// The account with "newID" is gone forever
```

- This method comes in handy: `containsKey()`

get returns null

- **`get`** will return **`null`** if the key is not found

```
assertNotNull(accounts.get("M"));
assertTrue(accounts.remove("M"));
assertNull(accounts.get("M"));
```

remove

- `remove` will return false if key is not found
  - return true if the mapping (the key-value pair) was successfully removed from the collection

```
assertTrue(accounts.remove("G"));

assertFalse(accounts.remove("Not Here"));
```

Which data structure?

- What data structures could we use to implement our own Map ADT?
  - ________ , _________ , _______ , _________

Big O using different data structures for a Map ADT?

Data Structure	put	get	remove
Unsorted Array			
Sorted Array			
Unsorted Linked List			
Sorted Linked List			
Binary Search Tree			

Where each node has a key and a value

Hash Tables

- Hash table: another data structure
- Provides virtually direct access to objects based on a key (a unique String or Integer)
  - key could be your SID, your telephone number, social security number, account number, …
  - Must have unique keys
  - Each key is associated with–mapped to–a value

Hashing

- Must convert keys such as "555-1234" into an integer index from 0 to some reasonable size
- Elements can be found, inserted, and removed using the integer index as an array index
- Insert (called put), find (get), and remove must use the same "address calculator"
  - which we call the Hash function

Hashing

- Ideally, every key has a unique hash value
- Then the hash value could be used as an array index
- However, ideal is impossible
  - Some keys "hash" to the same integer index
    - Known as a collision
  - Need a way to handle collisions!
    - "abc" may hash to the same integer array index as "cba"

Hashing

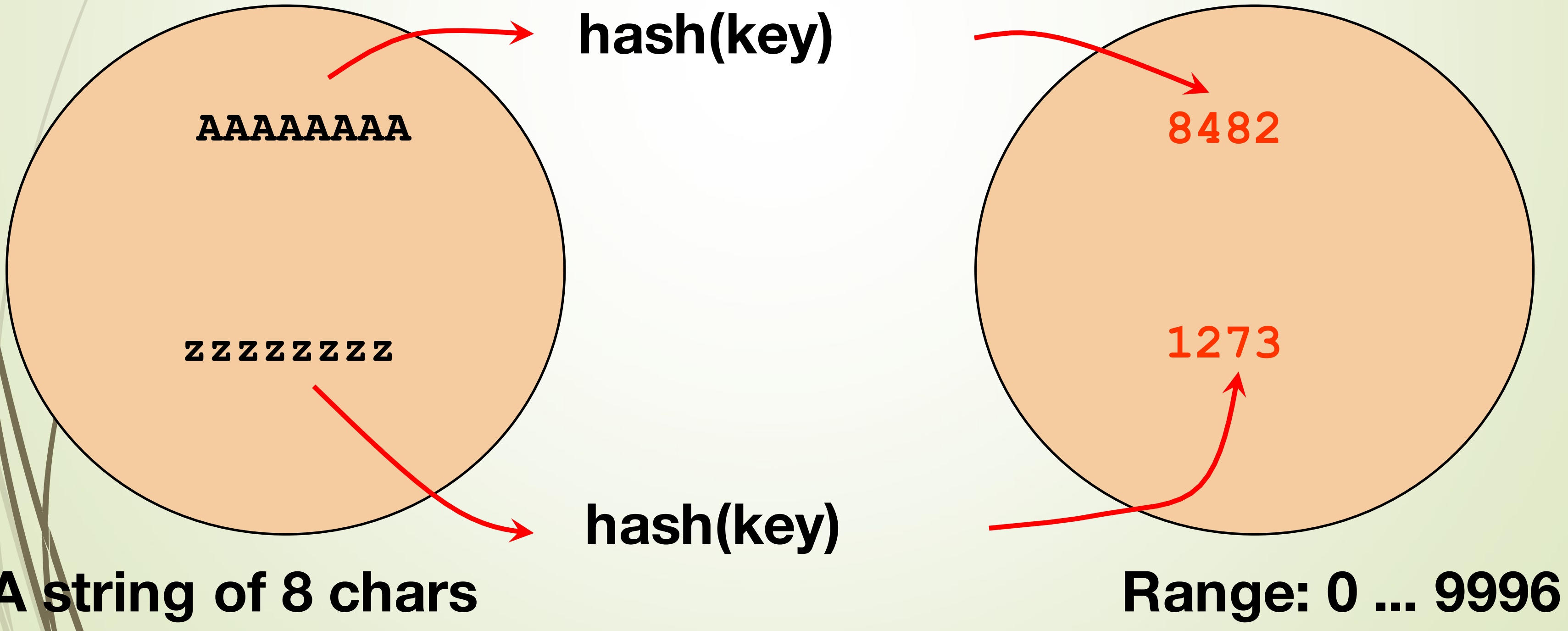
- Can make String or Integer keys into integer indexes by "hashing"
  - Need to take hashcode % array size
  - "S12345678" becomes an int **`0..array.length`**

Hash Tables: Runtime Efficient

- Lookup time doesn’t grow when n increases
- A hash table supports
  - fast insertion   O(1)
  - fast retrieval   O(1)
  - fast removal   O(1)
- Could use String keys each ASCII character equals some unique integer
  - "able" = 97 + 98 + 108 + 101 == 404

Hash method works something like…

Convert a String key into an integer that will be in the range of 0 through the maximum capacity-1 *Assume array capacity here is 9997*



Hash method

- What if the ASCII value of individual chars of the string key added up to a number from ("A") 65 to 488 ("zzzz") *4 chars max*
- If the array has size = 309, mod the sum

```
390 % TABLE_SIZE = 81
394 % TABLE_SIZE = 85
404 % TABLE_SIZE = 95
```

- These <u>array indices</u> index <u>these keys</u>

$$\begin{tikzcd}
81 \arrow[r] & \text{abba} \\
85 \arrow[r] & \text{abcd} \\
95 \arrow[r] & \text{able}
\end{tikzcd}$$

A too simple hash function

```
@Test
public void testHash() {
  assertEquals(81, hash("abba"));
  assertEquals(81, hash("baab"));
  assertEquals(85, hash("abcd"));
  assertEquals(86, hash("abce"));
  assertEquals(308, hash("IKLT"));
  assertEquals(308, hash("KLMP"));
}

private final int TABLE_SIZE = 309;

public int hash(String key) {
  // Return an int in the range of 0..TABLE_SIZE-1
  int result = 0;
  int n = key.length();
  for (int j = 0; j < n; j++)
    result += key.charAt(j); // add up the chars
  return result % TABLE_SIZE;
}
```

Collisions

- A good hash method
  - executes quickly
  - distributes keys equitably
- But you still have to handle collisions when two keys have the same hash value
  - the hash method is not guaranteed to return a unique integer for each key
    - example: simple hash method with "baab" and "abba"
- There are several ways to handle collisions
  - let us first examine linear probing

Linear Probing
Dealing with Collisions

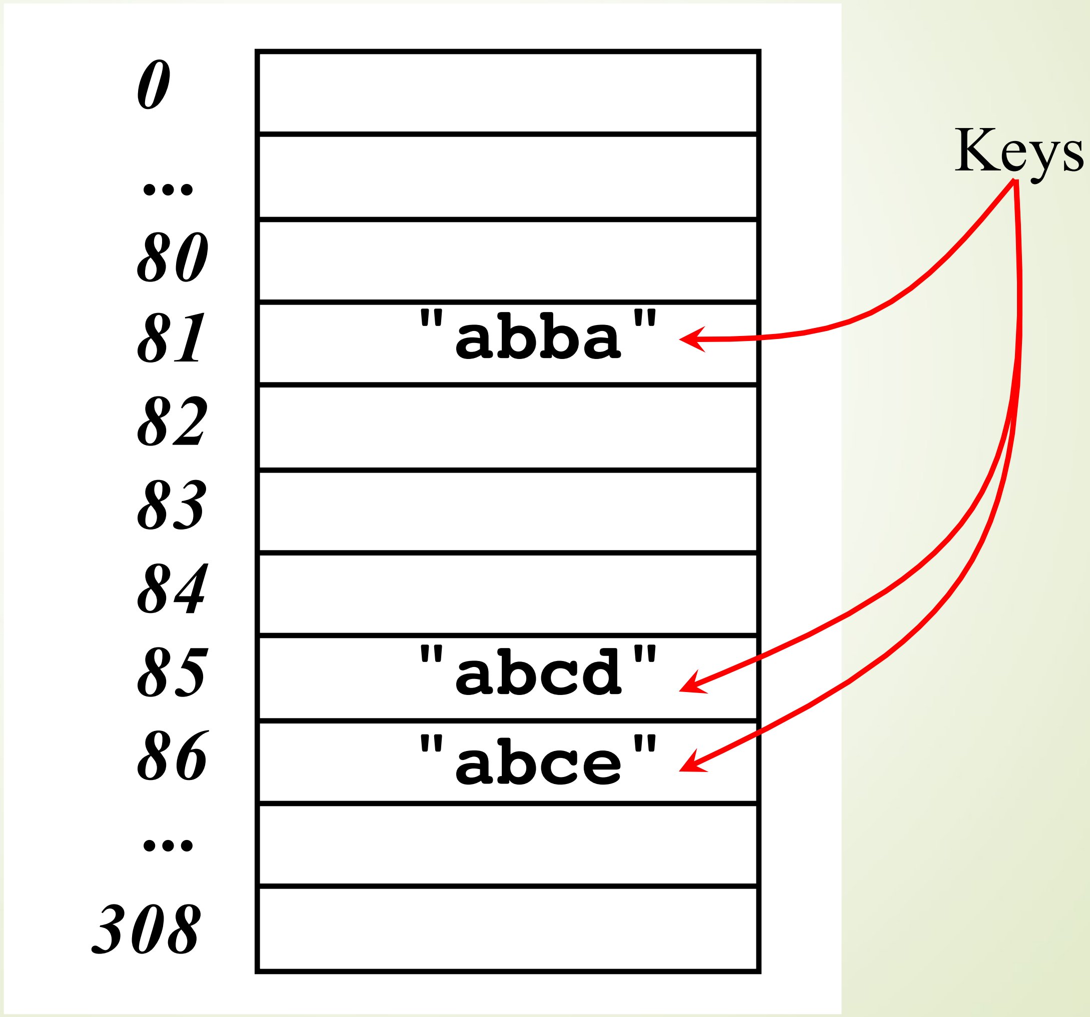
- ***Collision***: When an element to be inserted hashes out to be stored in an array position that is already occupied.
- ***Linear Probing***: search sequentially for an unoccupied position
  - uses a wraparound (circular) array

A hash table after three insertions

using the too simple (lousy) hash method

insert objects with these three keys:

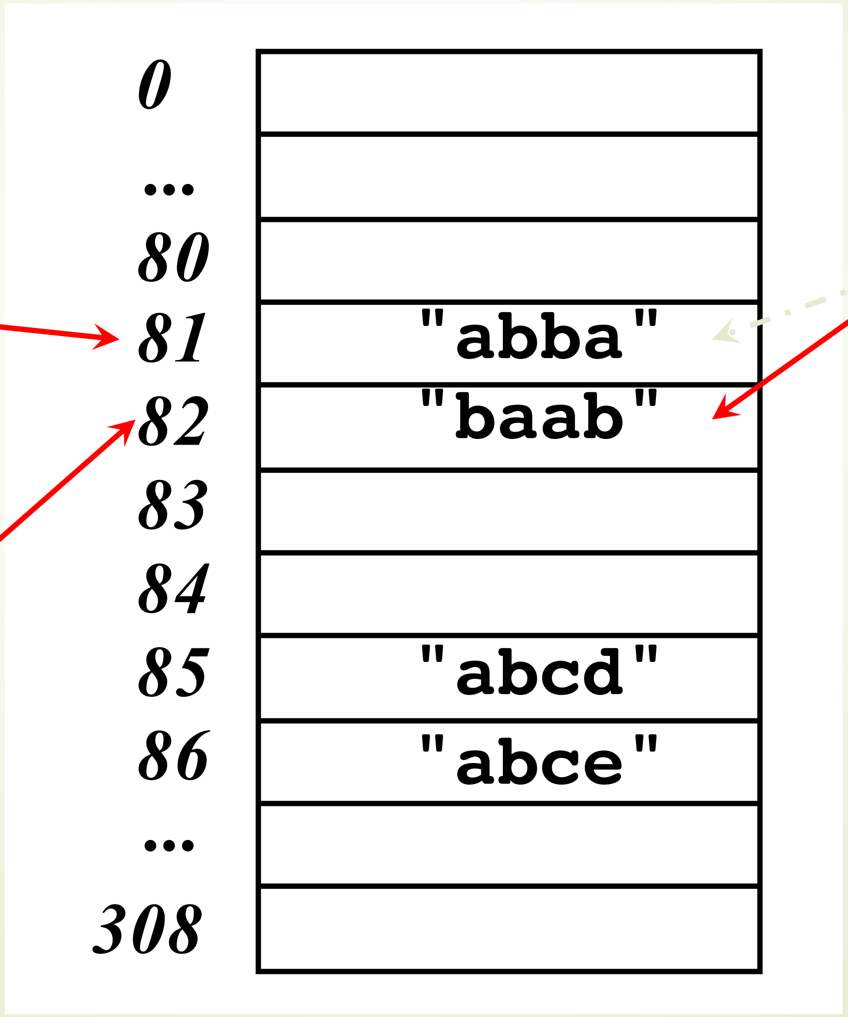
"abba"
"abcd"
"abce"



Collision occurs while inserting "baab"

can't insert "baab" where it hashes to same slot as "abba"

Linear probe forward by 1, inserting it at the next available slot



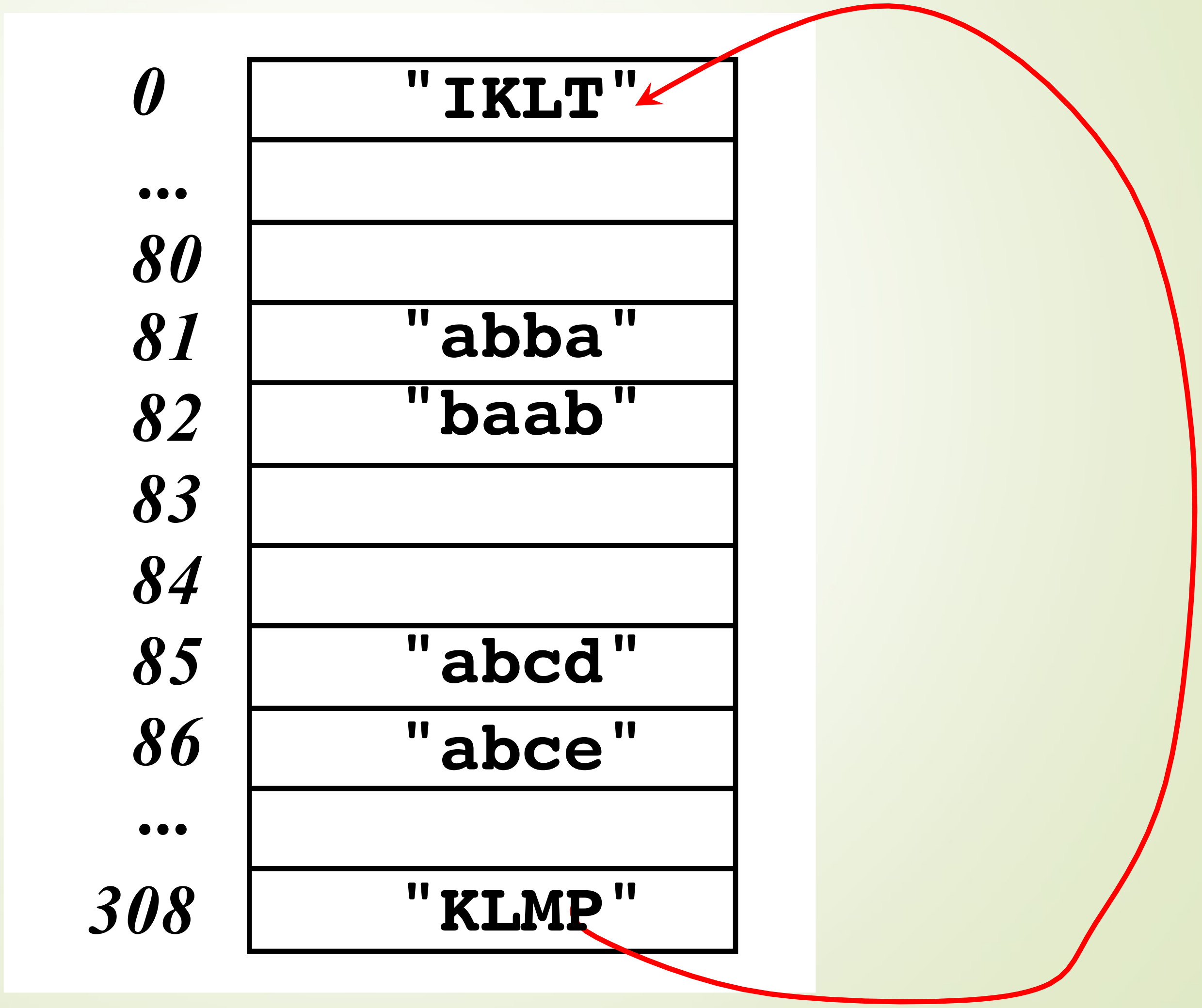
"baab"

Try [81]

Put in [82]

Wrap around when collision occurs at end

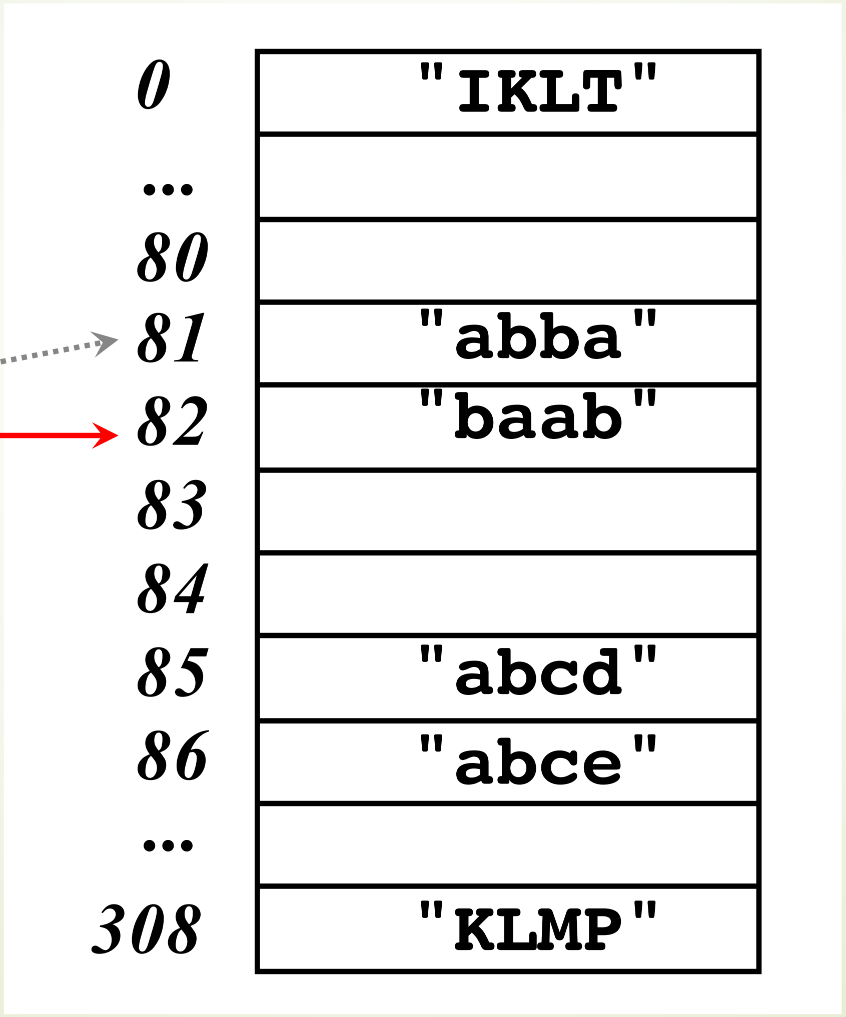
Insert "KLMP" and "IKLT" both of which have a hash value of 308



Find object with key "baab"

"baab" still hashes to 81, but since [81] is occupied, linear probe to [82]

At this point, you could return a reference or remove baab



HashMap put with linear probing

```
public class HashTable<Key, Value> {

  private class HashTableNode {
    private Key key;
    private Value value;
    private boolean active;
    private boolean tombstoned; // Allow reuse of removed slots

    public HashTableNode() {
      // All nodes in array will begin initialized this way
      key = null;
      value = null;
      active = false;
      tombstoned = false;
    }

    public HashTableNode(Key initKey, Value initData) {
      key = initKey;
      value = initData;
      active = true;
      tombstoned = false;
    }
  }
```

Constructor and beginning of put

```
private final static int TABLE_SIZE = 9;
private Object[] table;

public HashTable() {
  // Since HashNodeTable has generics, we can not have
  // a new HashNodeTable[], so use Object[]
  table = new Object[TABLE_SIZE];
  for (int j = 0; j < TABLE_SIZE; j++) {
    table[j] = new HashTableNode();
  }
}

public Value put(Key key, Value value) // . . .
```

put

- Four possible states when looking at slots
  - 1) the slot was never occupied, a new mapping
  - 2) the slot is occupied and the key equals argument
    - will wipe out old value
  - 3) the slot is occupied and key is not equal
    - proceed to next
  - 4) the slot was occupied, but nothing there now removed
    - We could call this a *tombStoned* slot
    - It can be reused

A better hash function

- This is the actual `hashCode()` algorithm of Java.lang.String (Integer’s is…well, the int)

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

Using int arithmetic, where s[i] is the *i*th character of the string, n is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

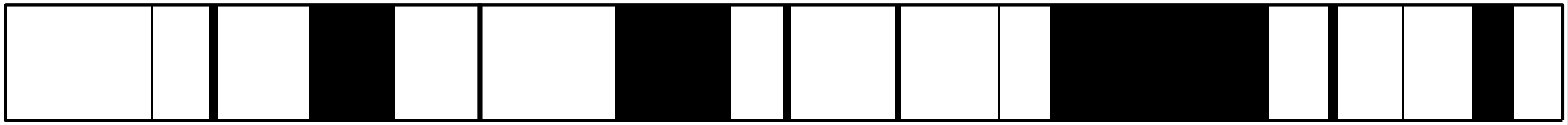
An implementation

```
private static int TABLE_SIZE = 309;

// s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
// With "baab", index will be 246.
// With "abba", index will be 0 (no collision).
public int hashCode(String s) {
  if(s.length() == 0)
    return 0;
  int sum = 0;
  int n = s.length();
  for(int i = 0; i < n-1; i++) {
    sum += s.charAt(i)*(int)Math.pow(31, n-i-1);
  }
  sum += s.charAt(n-1);
  return index = Math.abs(sum) % TABLE_SIZE;
}
```

Array based implementation has Clustering Problem

Used slots tend to cluster with linear probing



Black areas represent slots in use; white areas are empty slots

Quadratic Probing

- Quadratic probing eliminates the primary clustering problem
- Assume `hVal` is the value of the hash function
- Instead of linear probing which searches for an open slot in a linear fashion like this

```
hVal+1, hVal+2, hVal+3, hVal+4, ...
```

- add index values in increments of powers of 2

```
hVal+21, hVal+22, hVal+23, hVal+24, ...
```

Does it work?

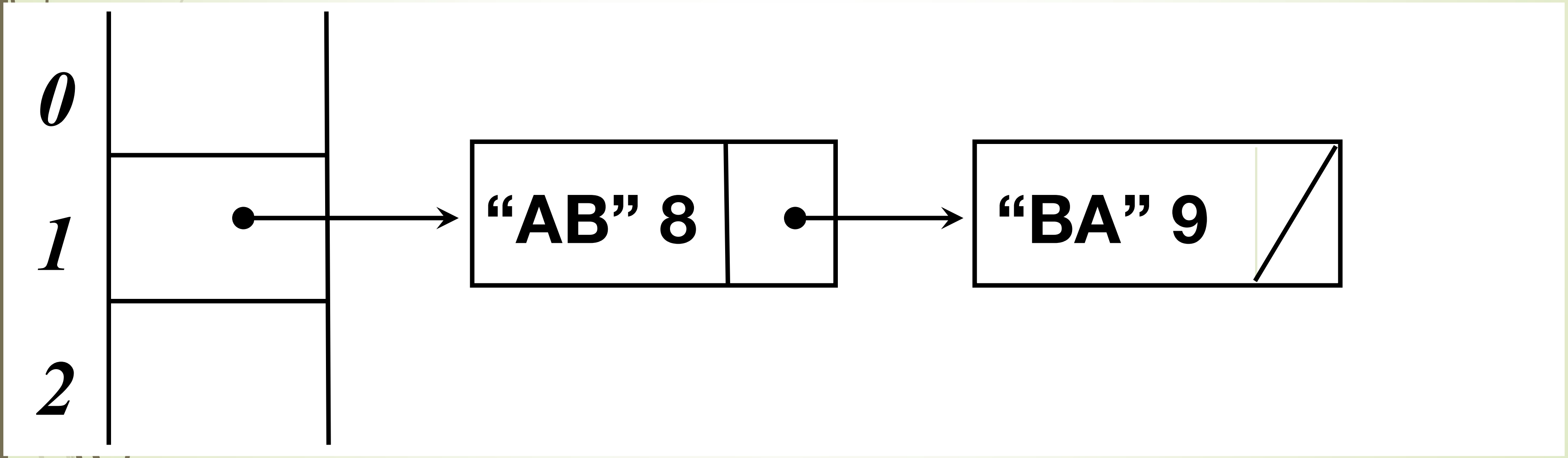
- Quadratic probing works well if
  - 1) table size is prime
    - studies show the prime numbered table size removes some of the non-randomness of hash functions
  - 2) table is never more than half full
- Make the table twice as big as needed
  - insert, find, remove are O(1)
  - A space (memory) tradeoff:
    - 4*n additional bytes required for unused array locations
    - es 1, 4, 9, 17, 33, 65, 129, ... slots away

Separate Chaining

- Separate Chaining is an alternative to probing
- How? Maintain an array of lists
- Hash to the same place always and insert at the beginning (or end) of the linked list
- The list must have add and remove methods,
  - Could use `LinkedList<E>` or `ArrayList<E>`

Array of LinkedLists Data Structure

- Each array element is a List



Insert Six Objects

```
@Test
public void testPutAndGet() {
  MyHashTable<String, BankAccount> h =
           new MyHashTable<String, BankAccount>();

  BankAccount a1 = new BankAccount("abba", 100.00);
  BankAccount a2 = new BankAccount("abcd", 200.00);
  BankAccount a3 = new BankAccount("abce", 300.00);
  BankAccount a4 = new BankAccount("baab", 400.00);
  BankAccount a5 = new BankAccount("KLMP", 500.00);
  BankAccount a6 = new BankAccount("IKLT", 600.00);

  // Insert BankAccount objects using ID as the key
  h.put(a1.getID(), a1);
  h.put(a2.getID(), a2);
  h.put(a3.getID(), a3);
  h.put(a4.getID(), a4);
  h.put(a5.getID(), a5);
  h.put(a6.getID(), a6);
  System.out.println(h.toString());
}
```

Lousy hash function and TABLE_SIZE==11

```
0. [IKLT=IKLT $600.00, KLMP=KLMP $500.00]
1. []
2. []
3. []
4. []
5. [baab=baab $400.00, abba=abba $100.00]
6. []
7. []
8. []
9. [abcd=abcd $200.00]
10. [abce=abce $300.00]
```

With Java's better hash method, collisions still happen

```
0. [IKLT=IKLT $600.00]
1. [abba=abba $100.00]
2. [abcd=abcd $200.00]
3. [baab=baab $400.00, abce=abce $300.00]
4. [KLMP=KLMP $500.00]
5. []
6. []
7. []
8. []
9. []
10. []
```

Experiment Rick v. Java

- Rick's linear probing implementation, Array size was 75,007
  - Time to construct an empty hashtable: 0.161 seconds
  - Time to build table of 50000 entries: 0.65 seconds
  - Time to lookup each table entry once: 0.19 seconds
- *8000 arrays of Linked lists*
  - Time to construct an empty hashtable: 0.04 seconds
  - Time to build table of 50000 entries: 0.741 seconds
  - Time to lookup each table entry once: 0.281 seconds
- *Java's HashMap<K, V>*
  - Time to construct an empty hashtable: 0.0 seconds
  - Time to build table of 50000 entries: 0.691 seconds
  - Time to lookup each table entry once: 0.11 seconds

Runtimes?

- What are the Big O runtimes for Hash Table using linear probing with an array of Linked Lists
- get ___________
- put _____________
- remove ______________

Hash Table Summary

- Hashing involves transforming a key to produce an integer in a fixed range (0..TABLE_SIZE-1)
- The function that transforms the key into an array index is known as the hash function
- When two data values produce the same hash value, you get a collision
  - it happens!
- Collision resolution may be done by searching for the next open slot at or after the position given by the hash function, wrapping around to the front of the table when you run off the end (known as linear probing)

Hash Table Summary

- Another common collision resolution technique is to store the table as an array of linked lists and to keep at each array index the list of values that yield that hash value *known as separate chaining*
- Most often the data stored in a hash table includes both a key field and a data field (e.g., social security number and student information).
- The key field determines where to store the value.
- A lookup on that key will then return the value associated with that key (if it is mapped in the table)