CMSC 132: Object-Oriented Programming II

Hash Tables

Key Value Map

- Red Black Tree: O(Log n)
- BST: O(n)
- 2-3-4 Tree: O(log n)

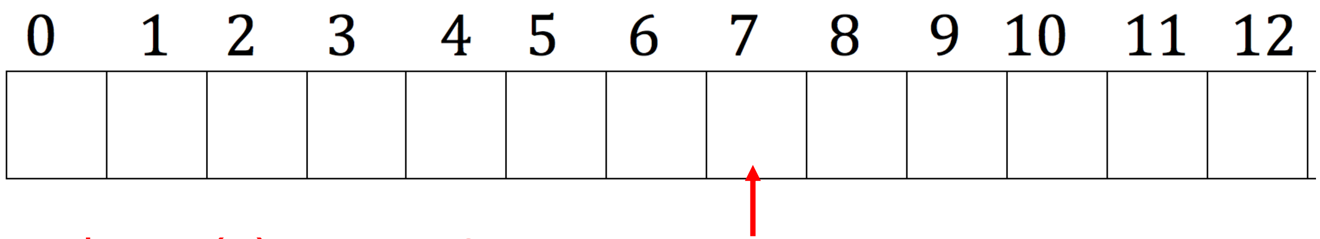
- Can we do better?

Hash Tables

- a data structure used to implement (a dictionary) an associative array, a structure that can map keys to values.
- O(1) best case. (Or average case).
- O(n) worst case. extremely unlikely to arise by chance, but a malicious adversary with knowledge of the hash function may be able to supply information to a hash that creates worst-case behavior by causing excessive collisions, resulting in very poor performance

Hash Table Example

- A hash table uses a **magic hash function** to compute an index into an array slots.
- An object can be found from the array using the index.



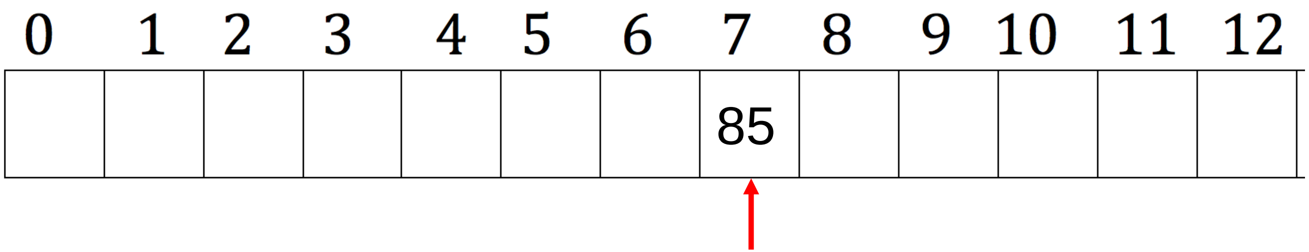
Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(85)= 7, H(91)=0, H(66)= 1, H(96)= 5, H(80)=2, H(88)=10, H(95)=4, H(87)=9, H(77)=12, H(63)=11, H(93)=2, H(82)=4

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.



Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(85)= 85 % 13 = 7

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91							85					

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(91)= 91 % 13 = 0

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66						85					

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(91)= 66 % 13 = 1

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66				96		85					

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(96)= 96 % 13 = 5

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80			96		85					

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(80)= 80 % 13 = 2

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80			96		85			88		

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(80)= 88 % 13 = 10

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80		95	96		85					

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(95)= 95 % 13 = 4

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80		95	96		85		87			

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(87)= 87 % 13 = 9

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80		95	96		85		87			77

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(77)= 77 % 13 = 12

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80		95	96		85		87		63	77

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(63)= 63 % 13 = 11

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80	93	95	96		85		87		63	77

Collision, find next available cell.

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(93)= 93 % 13 = 2

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12
91	66	80	93	95	96	82	85		87		63	77

Collision, find next available cell.

Hash function: H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

H(82)= 82 % 13 = 4

Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
91	66	80	93	95	96	82	85		87	88	63	77	

H(K) = K % 13

Grades: 85, 91, 66,96,80,88,95,87,77, 63, 93, 82

To find 10 numbers: 85, 91, 66,96,80,88,95,87,77, 63 need just 1 comparison,
93 needs 2 comparison,
82 needs 3 comparison

Average Search Times: (10 * 1 + 1 * 2 + 1 * 3)/ 12 = 1.25

Quiz 1

What is the time complexity to retrieve from a hash table if there are no collisions?

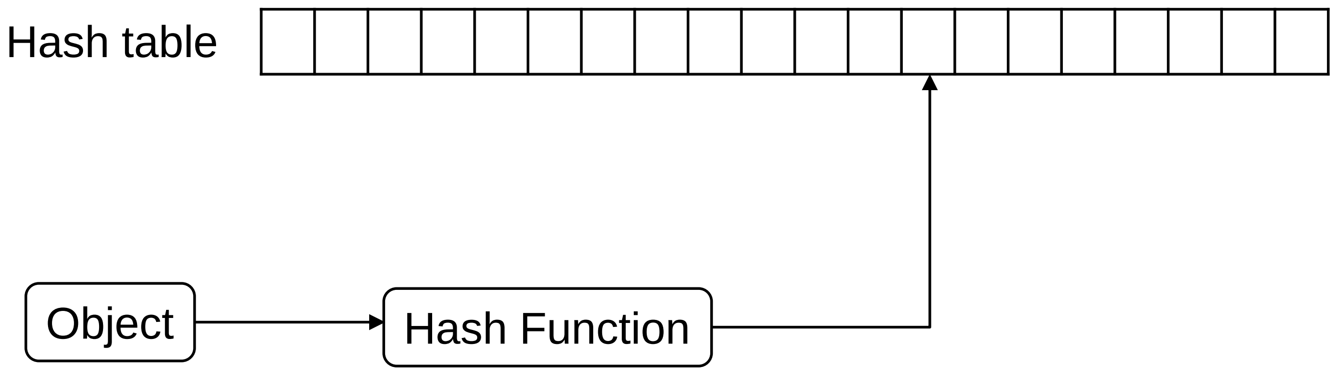
A. O(1)
B. O(n)
C. $O(n^2)$
D. O(log n)

Quiz 1

What is the time complexity to retrieve from a hash table if there are no collisions?

A. **O(1)**
B. O(n)
C. O(n^2)
D. O(log n)

Hash Functions



Given a value to search for, Hash function would tell us exactly where in the array to look:

- If it's in that location, it's in the array
- If it's not in that location, it's not in the array

Hash Functions

- When applied to an Object, returns a number
- When applied to *equal* Objects, returns the *same* number for each
- When applied to *unequal* Objects, is *very unlikely* to return the same number for each
- Hash functions is very important for searching.

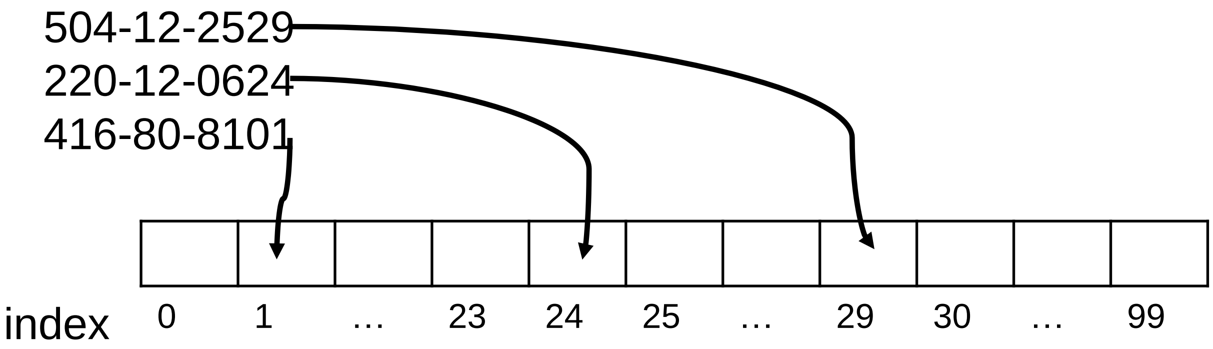
A.equals(B)==true ➔ A.hashCode()==B.hashcode()

Hash Functions

- Save items in a key-indexed table
- Array index is the function of the key.

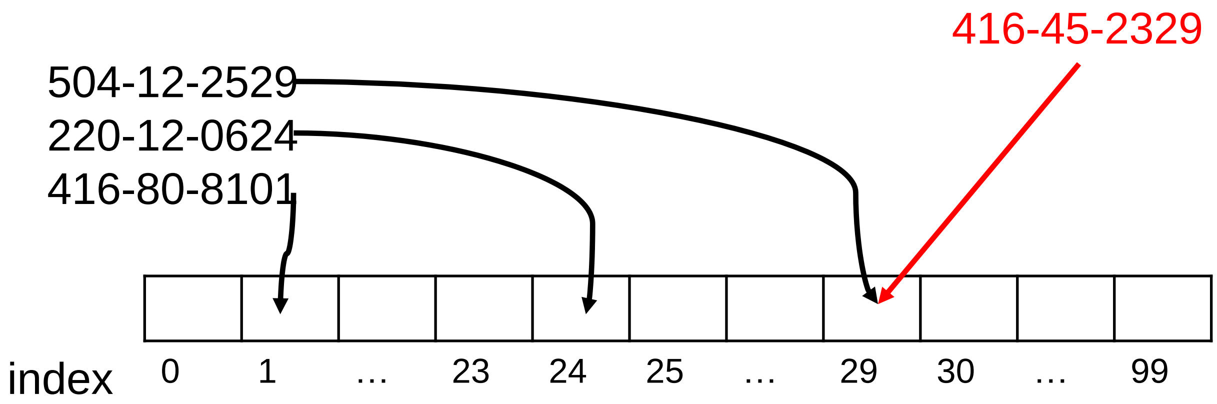
- Hash function: Method for computing array index from key

  H(key)= index
- Example:
  - Keys are social security numbers
  - Hash function: **`H(Key) = Key % 100`**



Hash Functions

- Issues:
  - Computing the hash function
  - Equality test: Method for checking whether two keys are equal
  - Collison resolution: Algorithm and data structure to handle two keys that hash to the same index



Quiz 2

A hash table of length 10 uses open addressing with hash function **h(k)=k mod 10**, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below. Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

A. 46, 42, 34, 52, 23, 33
B. 34, 42, 23, 52, 33, 46
C. 46, 34, 42, 23, 52, 33
D. 42, 46, 33, 23, 34, 52

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Quiz 2

A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below. Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

A. 46, 42, 34, 52, 23, 33
B. 34, 42, 23, 52, 33, 46
C. 46, 34, 42, 23, 52, 33
D. 42, 46, 33, 23, 34, 52

Quiz 3

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

A

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

B

Quiz 3

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

A

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

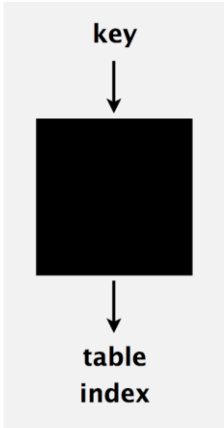
B

Hash Table: space and time limit

- Classic space-time tradeoff
  - No space limit: trivial hash function with key as index
  - No time limit: trivial collision resolution with sequential search
  - Space and time limit: hashing in the real world

Computing the Hash Function

- Idealistic goal: Scramble the keys uniformly to produce a table index.
  - Efficiently computable.
  - Each table index equally likely for each key.



  - Example 1. Phone numbers
    - Bad: first three digits.
    - Better: last three digits.
  - Example 2. Social Security numbers.
    - Bad: first three digits.
    - Better: last three digits.
  - Practical challenge. Need different approach for each key type.

Java's hash code conventions

- Java classes inherit a method hashCode(), which returns a 32-bit int.

- Requirement:

```
    If x.equals(y) then
     (x.hashCode() == y.hashCode()).
```

- Highly desirable:

```
     If !x.equals(y) then
           (x.hashCode() != y.hashCode())
```

Java's Hash Code Conventions

- Java hashCode() Default implementation:
  - Memory address of x.

- Legal (but poor) implementation:
  - Always return 17.

- Customized implementations:
  - Integer, Double, String, File, URL, Date,

- User-defined types:
  - Users are on their own.

Hash code: integers, and doubles

```
public final class Integer{
   private final int value;
     ...
   public int hashCode(){
   return value;  }
}

final class Double{
  private final double value;
   ...
  public int hashCode(){
    long bits = doubleToLongBits(value);
    return (int) (bits ^ (bits >>> 32));
   }
}
```

Hash code: booleans

```
public final class Boolean{
   private final boolean value;
      ...
   public int hashCode() {
      if (value)
            return 1231;
      else
            return 1237;
   }
}
```

Implementing hash code: Strings

```
public final class String{
    private final char[] s;
    ...
    public int hashCode(){
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

i^{th} character of s

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

(Horner's method)

Hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>{
   private final String  who;
   private final Date    when;
   private final double  amount;
   public Transaction(String who, Date when, double amount)
   {  /* as before */  }
      ...
   public boolean equals(Object y){/* as before */  }
   public int hashCode(){
          int hash = 17;
          hash = 31*hash + who.hashCode();
          hash = 31*hash + when.hashCode();
          hash = 31*hash + ((Double) amount).hashCode();
          return hash;
   }
}
```

Hash Code Design

- Standard" recipe for user-defined types:
  - Combine each significant field using the **`31x + y`** rule.
  - If field is a primitive type, use wrapper type hashCode().
  - If field is null, return 0.
  - If field is a reference type, use hashCode().
  - If field is an array, apply to each entry.

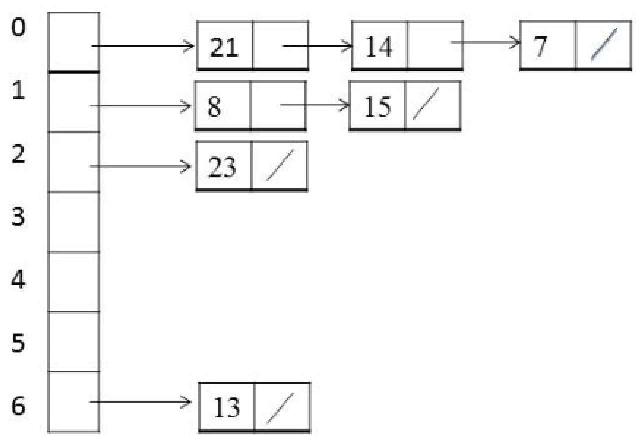
- In practice:
  - Recipe works reasonably well; used in Java libraries.
  - In theory. Keys are bitstring; "universal" hash functions exist.

Separate Chaining

- Use an array of M < N link lists
- Hash: map key to integer **`i`** between 0 and M-1
- Insert: put at front of **`i`**th chain (if not already there)
- Search: need to search only **`i`**th chain

H(k)= k %7

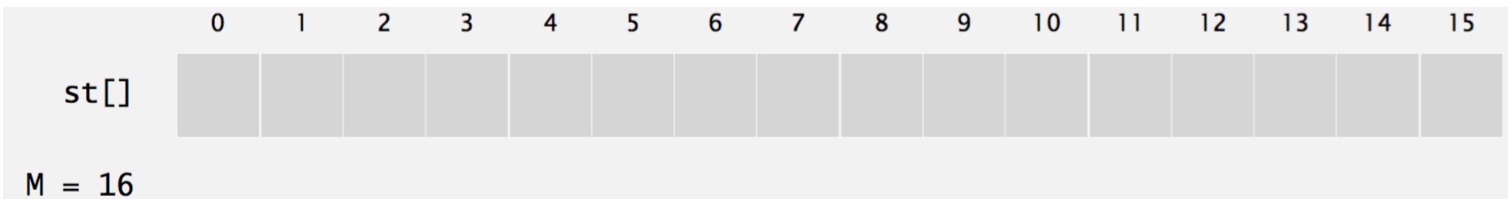
Insert: 8,21,23,14,13,7



Hash: Example 1

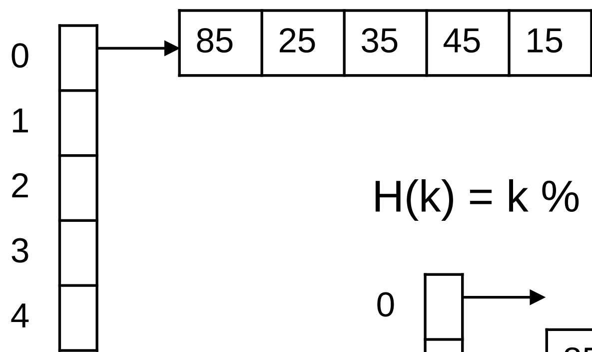
1. Linear Probing H(k) = k % 15

Insert 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82,

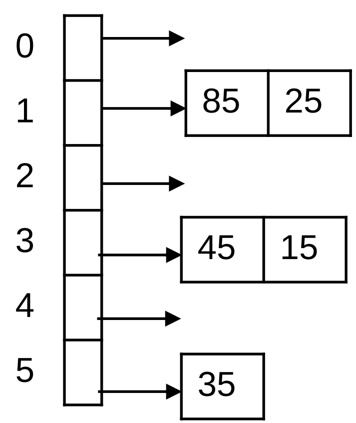


Hash: Example 2

1. Separate Chain H(k) = k % 5
   Insert 85, 25, 35, 45,15



H(k) = k % 6



Clustering

- Cluster. A contiguous block of items.
- Observation. New keys likely to hash into middle of big clusters.
- Solutions:

Separate chaining vs. linear probing

- Separate chaining.
  - Easier to implement delete.
  - Performance degrades gracefully.
  - Clustering less sensitive to poorly-designed hash function.
- Linear probing.
  - Less wasted space.
  - Better cache performance.

Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

Hash Tables

- Simpler to code
- No effective alternative fo
- Faster for simple keys (a f
- Better system support in Ja
- Balanced search trees
- Stronger performance guaran
- Support for ordered ST oper
- Easier to implement compare
- Red-black BSTs
- Hash tables:

Hash tables vs. balanced search trees

Hash Tables (HashMap, HashSet)	Balanced search trees (Red-black tree, treeMap, TreeSet)
• Simpler to code • No effective alternative for unordered keys • Faster for simple keys (a few arithmetic ops vs logN compares) • Better system support in Java for strings (e.g., cached hash code).	• Stronger performance guarantee. • Support for ordered ST operations • Easier to implement compareTo() correctly than equals() and hashCode().

Quiz 2

A hash function should have which properties?

A. Uniform distribution
B. Efficient hash code computation
C. Range is a subset of the integers
D. Equivalent objects produce equal hash codes

Quiz 2

A hash function should have which properties?

A. Uniform distribution
B. Efficient hash code computation
C. Range is a subset of the integers
D. Equivalent objects produce equal hash codes

Quiz 5

Hash table of size seven, with starting index zero, and a hash function **`(3x + 4) mod 7`**. Keys 1, 3, 8, 10 are inserted into an empty table.

Which of the following is the contents of the table when?

Index	0	1	2	3	4	5	6
A	8	1				3	10
B	1	8	10				3
C	1	10	8				3
D	1	10	8				3

Quiz 5

Hash table of size seven, with starting index zero, and a hash function **`(3x + 4) mod 7`**. Keys 1, 3, 8, 10 are inserted into an empty table.

Which of the following is the contents of the table when?

Index	0	1	2	3	4	5	6
A	8	1				3	10
B	**1**	**8**	**10**				**3**
C	1	10	8				3
D	1	10	8				3

Quiz 6

Hash table keys are ordered.

A. True
B. False

Quiz 6

Hash table keys are ordered.

A. True
B. False

Quiz 7

What is the worst-case time complexity to retrieve from a hash?

A. O(1)
B. O(n)
C. $O(n^2)$
D. O(log n)

Quiz 7

What is the worst-case time complexity to retrieve from a hash?

A. O(1)
B. O(n)
C. $O(n^2)$
D. O(log n)

ConcurrentHashMap

- A hash table supporting full concurrency of retrievals and high expected concurrency for updates.

- **`put(K key, V value)`**: Maps the specified key to the specified value in this table.

- **`get(Object key)`**: Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

- **`putIfAbsent(K key, V value)`**: If the specified key is not already associated with a value, associate it with the given value.