PROGRAMMING AND DATA STRUCTURES

HASH TABLES

HOURIA OUDGHIRI

FALL 2023

OUTLINE

- Hashing and Hash Tables
- Collisions in Hash Tables
- Solutions to the collision problem
    - Open Addressing and Separate Chaining
- Performance of Hash Tables
- Implementing a Hash Table (class HashMap)

STUDENT LEARNING OUTCOMES

At the end of this chapter, your should be able to:

- Describe how hashing and hash tables work
- Apply different solutions to handle collisions
- Implement and test the hash table data structure
- Evaluate the time complexity of the operations on a Hash Table

Search operation

- Array List: `O(n)`
- Linked List: `O(n)`
- BST: `O(log n)` (when kind of balanced)
- Can we perform search in less time?

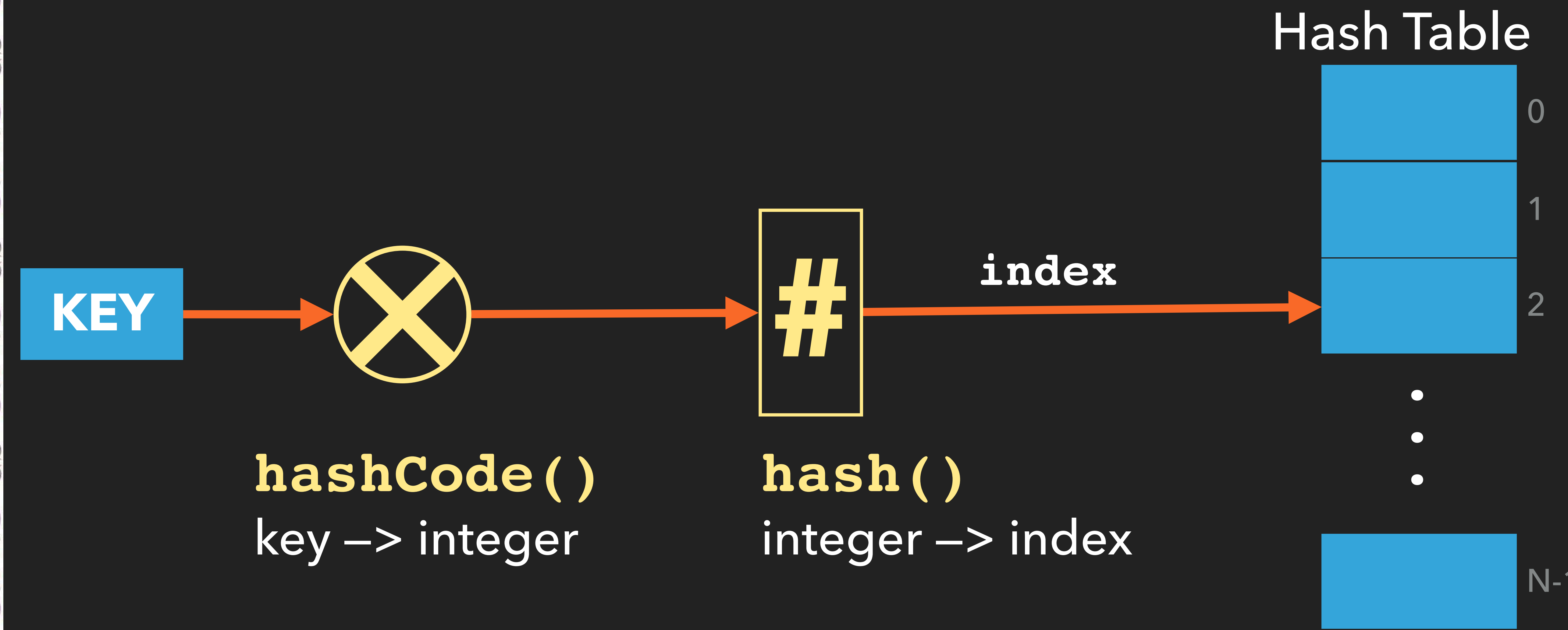
Hash tables

- Hash tables allow to perform search operations in `O(1)`
- Hash tables use associative access to data
- Associative memory: access data using the data itself instead of an address (index)

Hash tables

- Store the data in an array - Hash Table (**`HT`**)
- Access the **`HT`** elements using a hash function **`h()`** that returns an index in the **`HT`**

Hash tables


Hash Table
KEY
index
0
1
2
N-1
hashCode()
key –> integer
hash()
integer –> index

Hash tables

- If the size of **`HT`** is **`N`**, then

  `0 ≤ hash()≤ N-1` (valid index)

- Searching for a value **`key`** is performed using one comparison with

  `HT[hash(hashCode(key))]`

- How is data added/found in **`HT`** using **`hash()`**?

- How **`hashCode()`** and **`hash()`** are defined?

Hash tables

- Adding data to the table
    - Apply **`hash()`** to the data to determine the index where the data should be added
- Retrieving data from the table
    - Apply **`hash()`** to the data to find the index where it is in the hash table

Hash tables

Example

- Values `{11, 34, 57, 59, 72, 85, 91, 93}` to store in a `HT` of size 11
- Each value `v` is stored at an index `i` calculated by `hash(v)= v%(size of HT) [i = v % 11]`
- Searching for a value `v`: compare `v` to `HT[hash(v)]`

Hash tables

- Adding data to **HT**

```
hash(11) = 11 % 11 = 0
hash(34) = 34 % 11 = 1
hash(57) = 57 % 11 = 2
hash(59) = 59 % 11 =4
hash(72) = 72 % 11 = 6
hash(85) = 85 % 11 = 8
hash(91) = 91 % 11 = 3
hash(93) = 93 % 11 = 5
```

HT

HT	
11	0
34	1
57	2
91	3
59	4
93	5
72	6
-1	7
85	8
-1	9
-1	10

Hash tables

- Retrieving data from `HT`
  - `key = 91`

    `hash(key) = 91 % 11 = 3`

    key found

  - `key = 75`

    `hash(75) = 75 % 11 = 9`

    key not found

  - Search operation: `O(1)`

HT

HT	
11	0
34	1
57	2
91	3
60	4
93	5
72	6
-1	7
85	8
-1	9
-1	10

Collisions

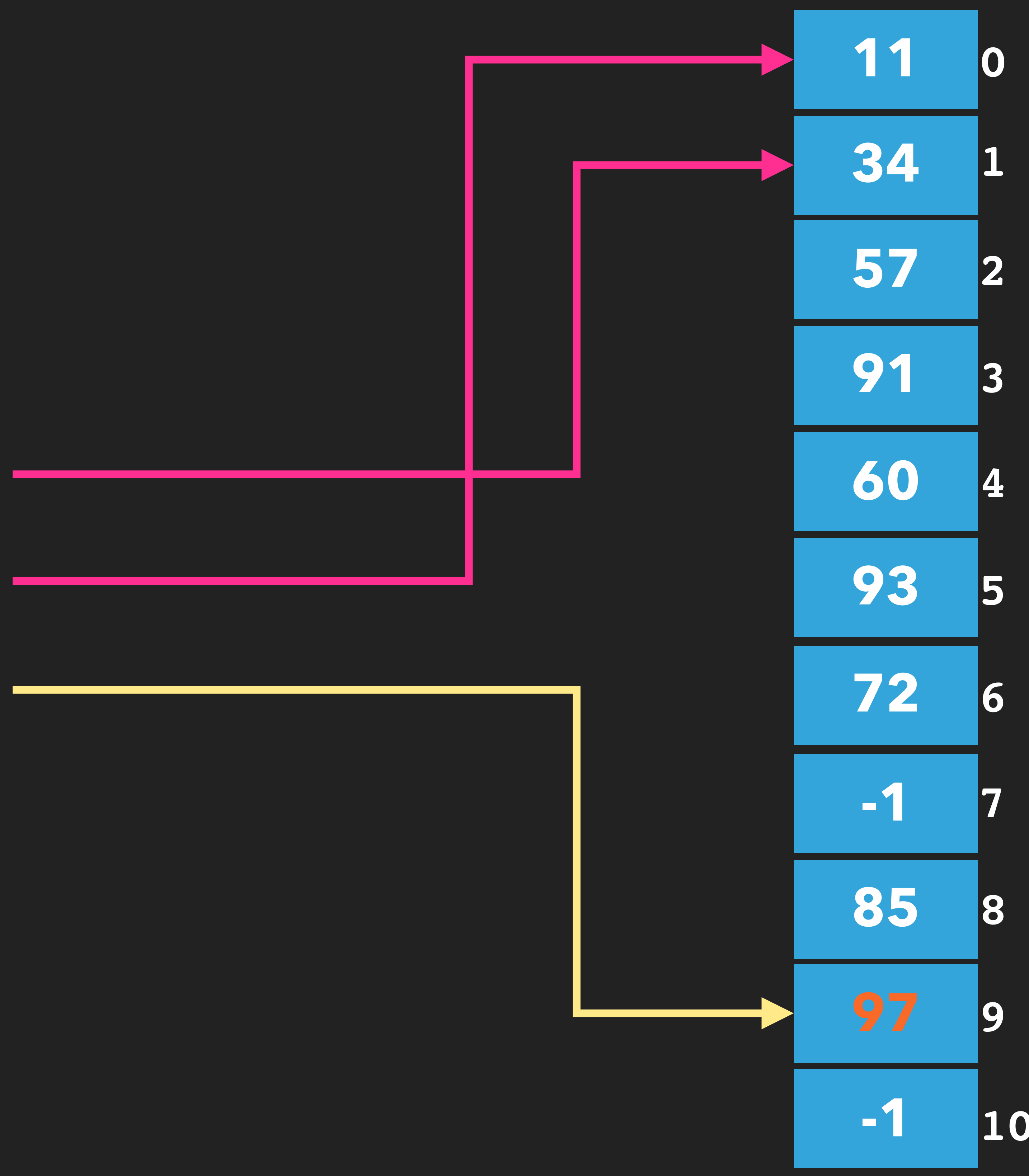
- Issue with the hash method
- Set of values may be hashed to the same index (23, 34, 56, 78, 89 all have the hash value = 1) for size = 11
- Collision: two or more values have the same hash function value

Collisions

h(45)=45%11

h(55)=55%11

h(97)=97%11



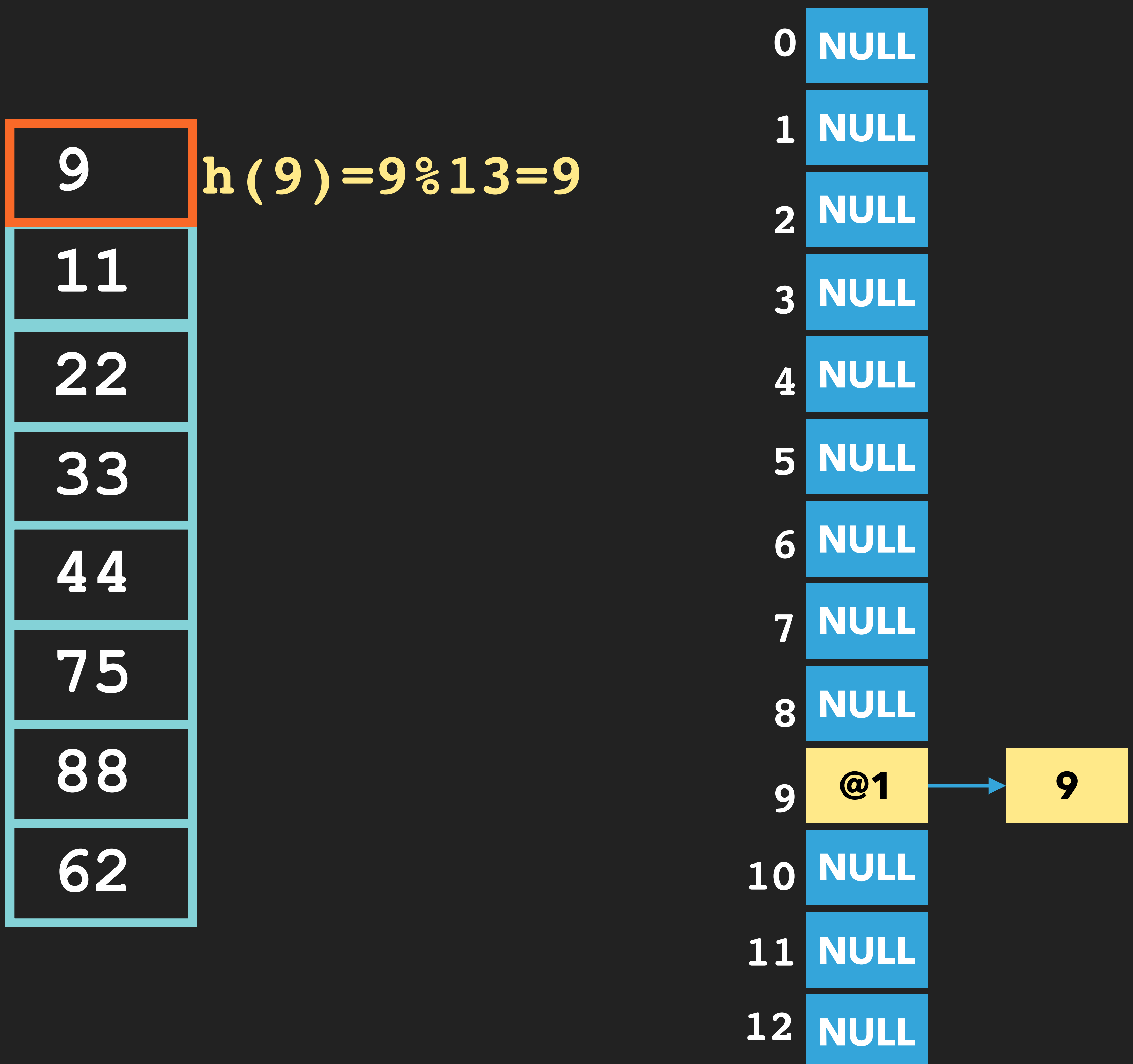
Collisions

- Collisions - two or more values have the same hash function value
- Two solutions for the collision problem
  - Separate Chaining (open hashing)
  - Open Addressing (closed hashing)

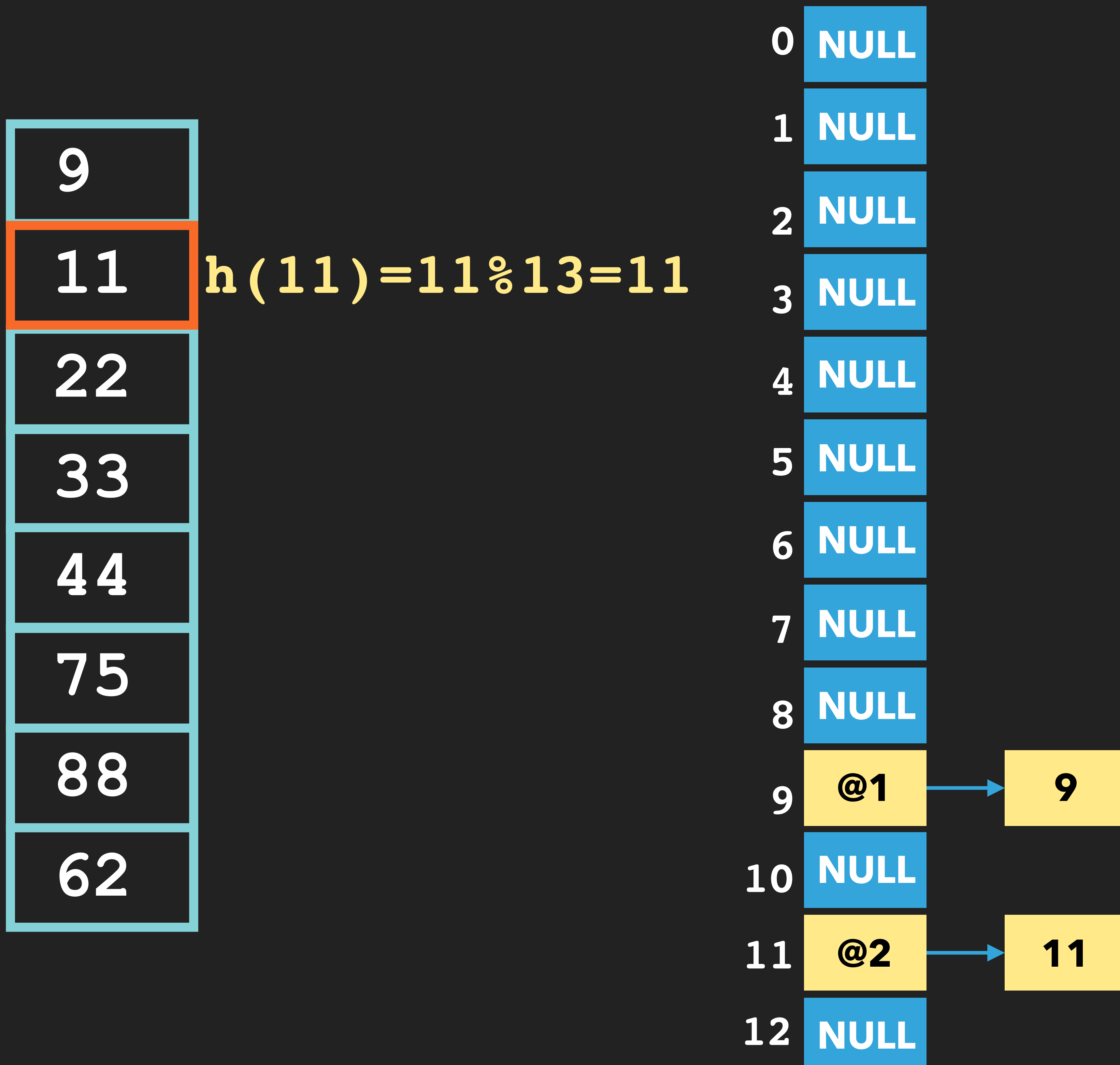
Collisions

- Solutions for the collision problem
  - **Separate Chaining** - collisions are stored outside the hash table in a list (array list or linked list, or even a tree)
  - **Open Addressing** - collisions are stored in the hash table itself at the next available index

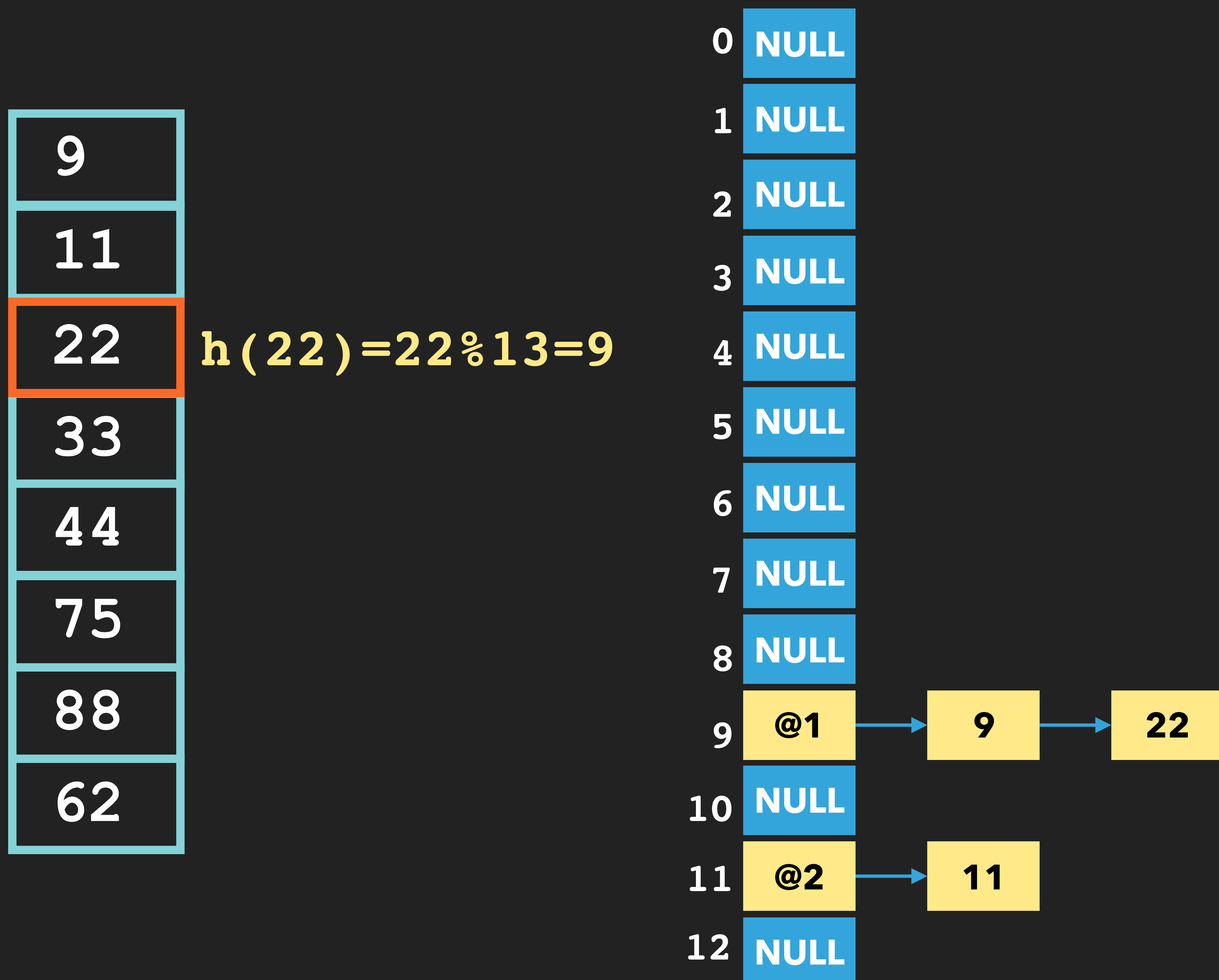
Collisions - Separate Chaining



Collisions - Separate Chaining



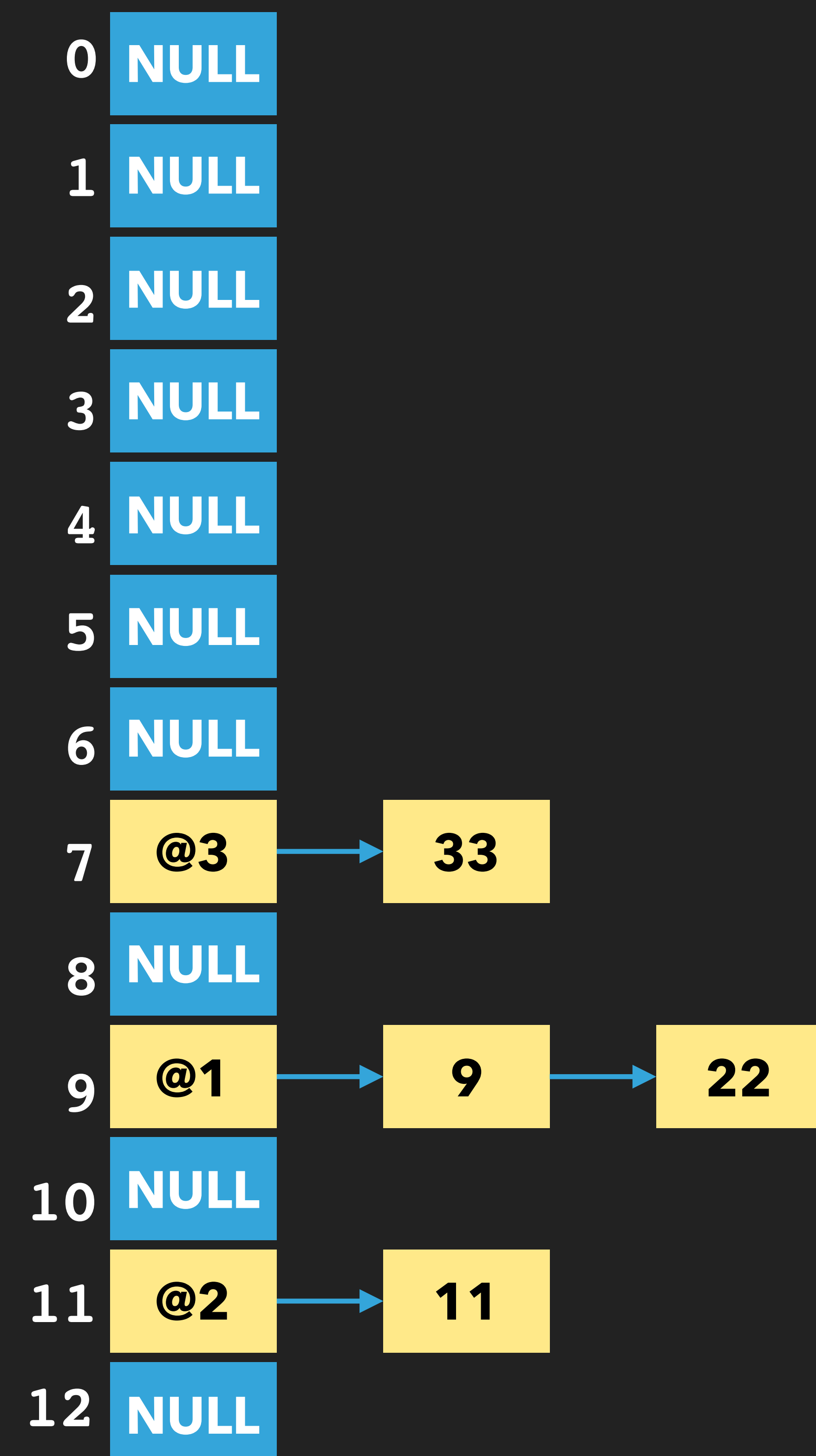
Collisions - Separate Chaining



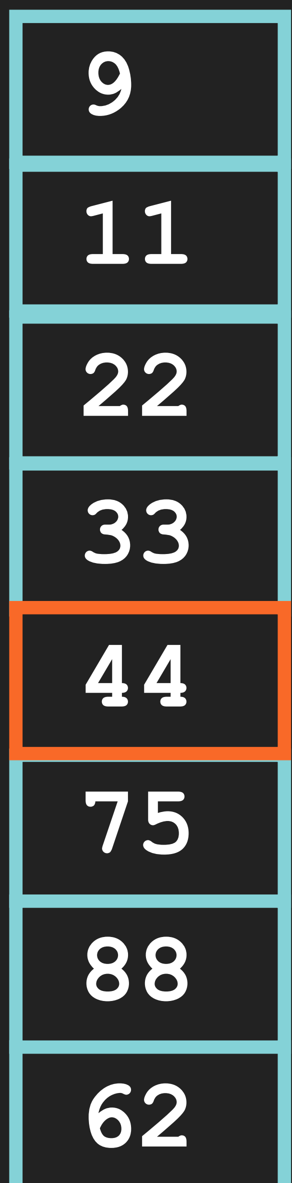
Collisions - Separate Chaining

9
11
22
33
44
75
88
62

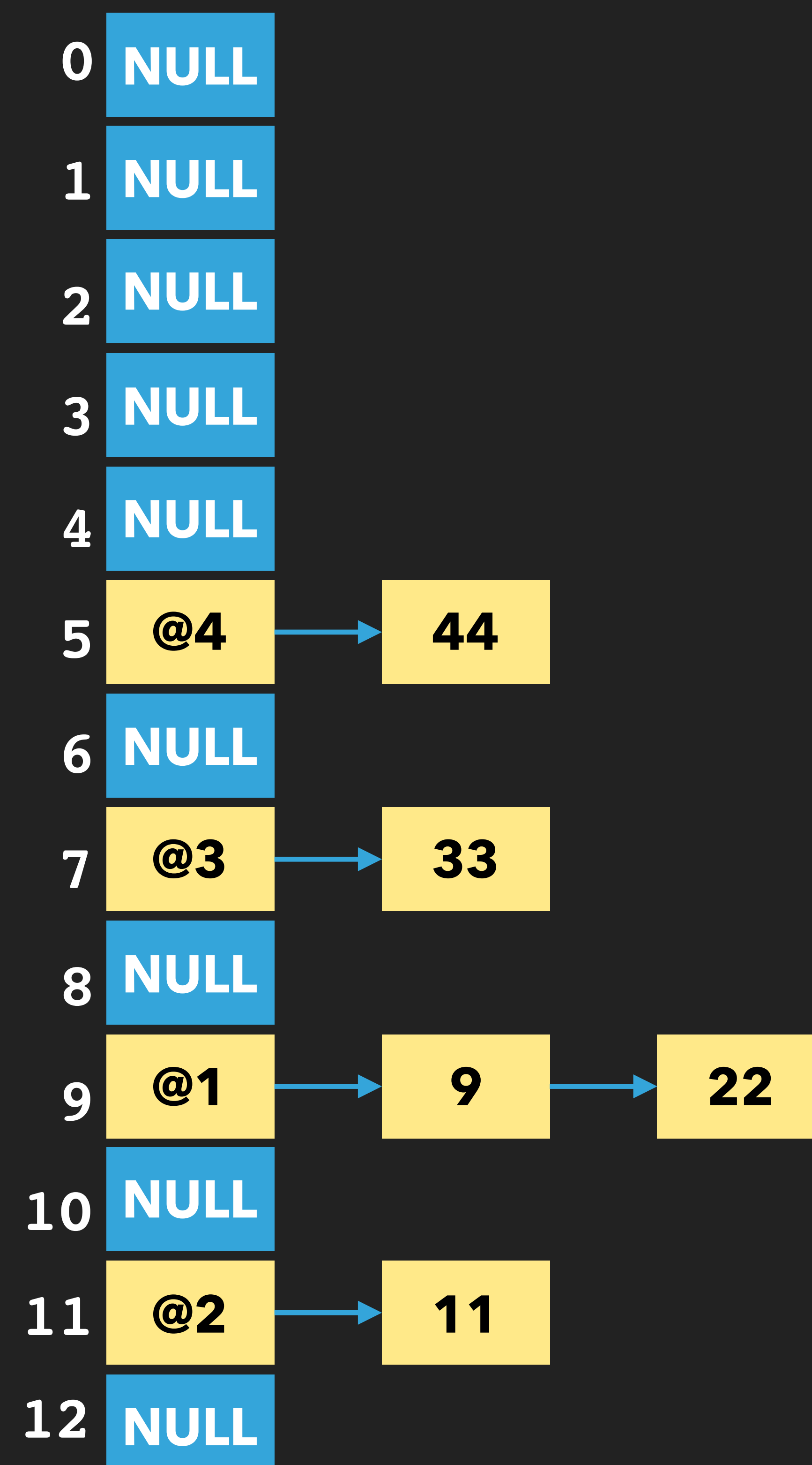
`h(33)=33%13=7`



Collisions - Separate Chaining



`h(44)=44%13=5`



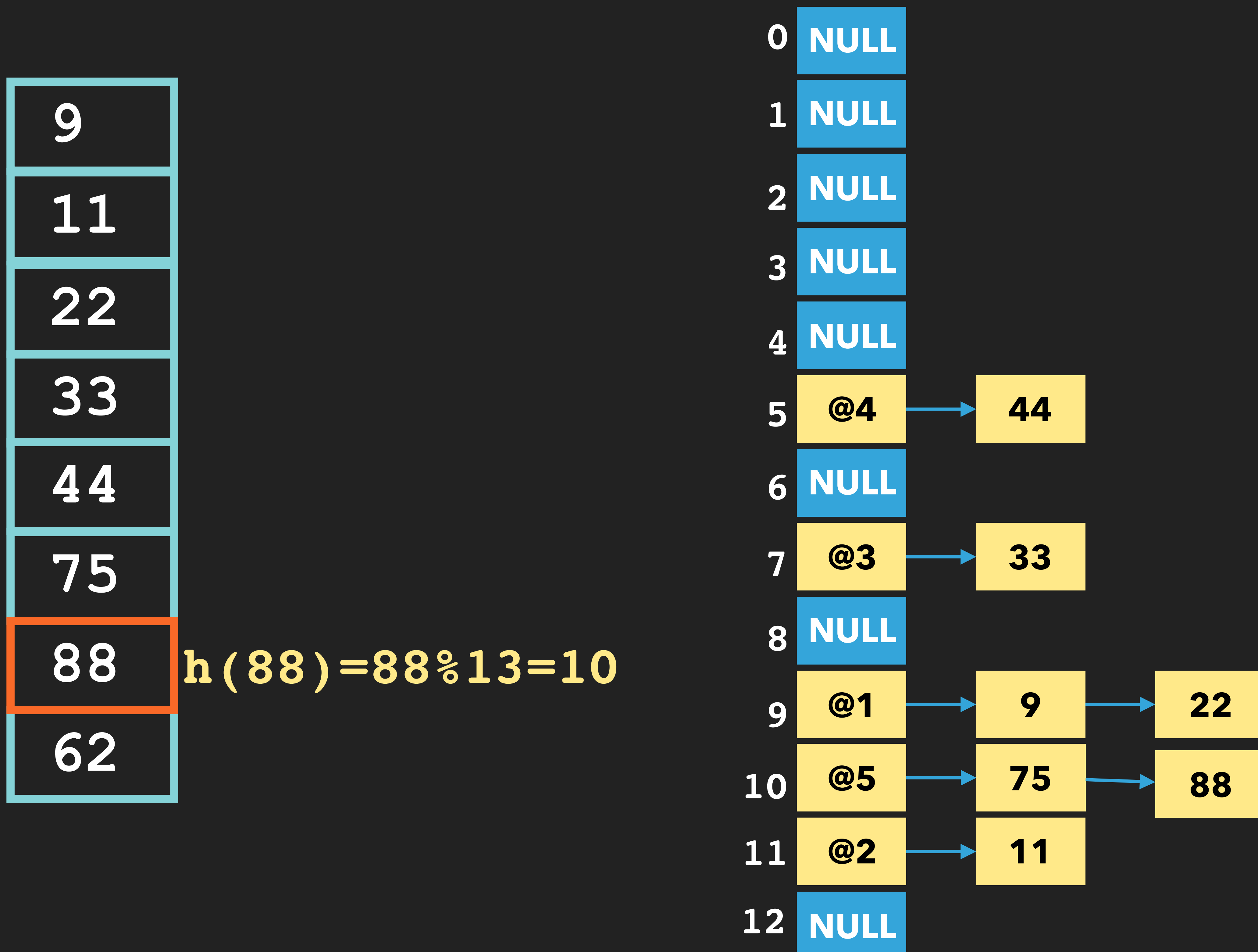
Collisions - Separate Chaining

9
11
22
33
44
75
88
62

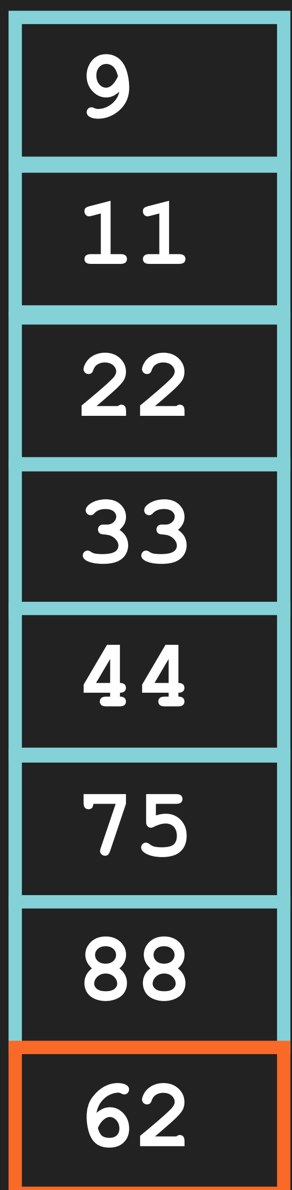
`h(75)=75%13=10`

0	NULL		
1	NULL		
2	NULL		
3	NULL		
4	NULL		
5	@4	→ 44	
6	NULL		
7	@3	→ 33	
8	NULL		
9	@1	→ 9	→ 22
10	@5	→ 75	
11	@2	→ 11	
12	NULL		

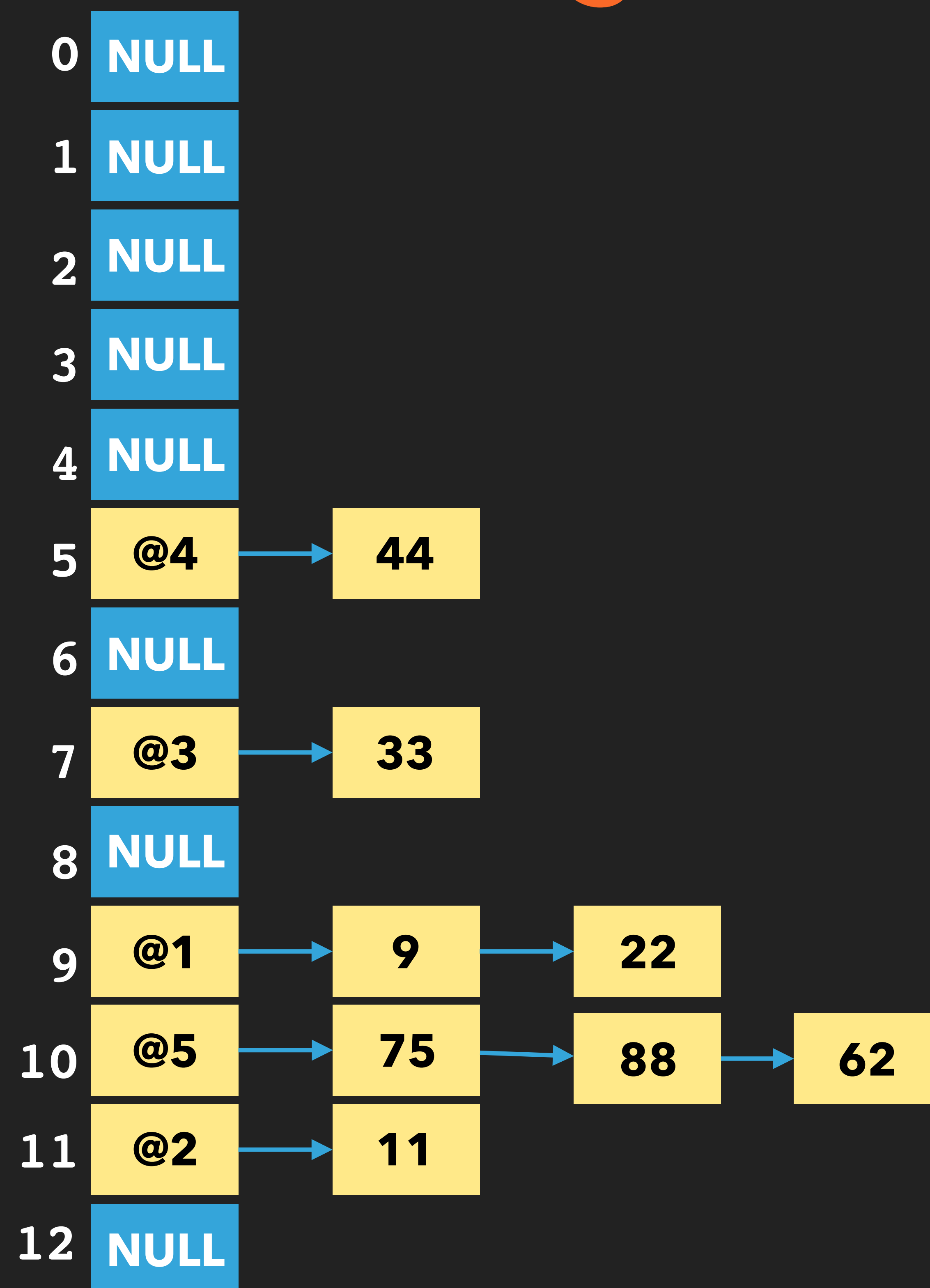
Collisions - Separate Chaining



Collisions - Separate Chaining



h(62)=62%13=10



Collisions - Open Addressing

- **Linear probing** - look for the next available slot using a linear function (`index+j` until an empty element is found)
- **Quadratic probing** - look for the next available slot using a quadratic function (`index+j`^{2} until an empty element is found)
- **Double hashing** - look for the next available slot at an index using another hash function (`index + hash2(v)`)

Collisions - Linear Probing

9
11
22
33
44
75
88
62

HT

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

Collisions - Linear Probing

9, 11, 22, 33, 44, 75, 88, 62

h(9)= 9 % 13 = 9

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

Collisions - Linear Probing

Keys
9
11
22
33
44
75
88
62

`h(11)= 11 % 13 = 11`

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

Collisions - Linear Probing

9
11
22
33
44
75
88
62

`h(22)=(9 + 1) % 13   10`

0	null	null
1	null	null
2	null	null
3	null	null
4	null	null
5	null	null
6	null	null
7	null	null
8	null	null
9	9	9
10	null	22
11	11	11
12	null	null

Collisions - Linear Probing

9
11
22
33
44
75
88
62

`h(33) = 33 % 13 = 7`

0	null	null
1	null	null
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	NULL	NULL
6	NULL	NULL
7	NULL	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions - Linear Probing

9
11
22
33
44
75
88
62

`h(44) = 44 % 13 = 5`

0	NULL	NULL
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	NULL	44
6	NULL	NULL
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions - Linear Probing

9
11
22
33
44
75
88
62

`h(75)=(10 + 2) % 13 = 12`

0	NULL	NULL
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	44	44
6	NULL	NULL
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	75

Collisions - Linear Probing

9
11
22
33
44
75
88
62

`h(88) =(10 + 3) % 13 = 0`

0	NULL	88
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	44	44
6	NULL	NULL
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	75	75

Collisions - Linear Probing

9
11
22
33
44
75
88
62

`h(62) = (10 + 4) % 13 = 1`

0	88	88
1	NULL	62
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	44	44
6	NULL	NULL
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	75	75

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

HT

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

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(9) = 9 % 13 = 9`

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

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(11) = 11 % 13 = 11`

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

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(22) = (9 + 1) % 13 = 10`

0	NULL	NULL
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	NULL	NULL
6	NULL	NULL
7	NULL	NULL
8	NULL	NULL
9	9	9
10	NULL	22
11	11	11
12	NULL	NULL

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

```
h(33) = 7 % 13 = 7
```

0	NULL	NULL
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	NULL	NULL
6	NULL	NULL
7	NULL	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(44) = 44 % 13 = 5`

0	NULL	NULL
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	NULL	44
6	NULL	NULL
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(75) = (10 + 4) % 13 1`

0	NULL	NULL
1	NULL	75
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	44	44
6	NULL	NULL
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(88) = (10 + 9) % 13 = 6`

0	NULL	NULL
1	75	75
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	44	44
6	NULL	88
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions - Quadratic Probing

9
11
22
33
44
75
88
62

`h(62) = (10 + 16) % 13 = 0`

0	NULL	62
1	75	75
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	44	44
6	88	88
7	33	33
8	NULL	NULL
9	9	9
10	22	22
11	11	11
12	NULL	NULL

Collisions

- The number of collisions may affect the performance of the search operation
- could result in linear search if the number of collisions is high

Practice

- We want to add the following data to a hash table of 15 elements

  57, 8, 19, 30, 20, 34, 37, 42, 35, 21, 27

- The hash function is `hash(x) = x % 15`

- Show the content of the hash table after adding the data using separate chaining, linear probing, and quadratic probing to resolve the collisions (Practice Sheet)

Collisions

- Linear Probing
  - Clusters are formed
  - Clusters can grow in size and slow down the (search, add, and remove) operations
  - Linear probing guarantees to find an empty element if the table is not full

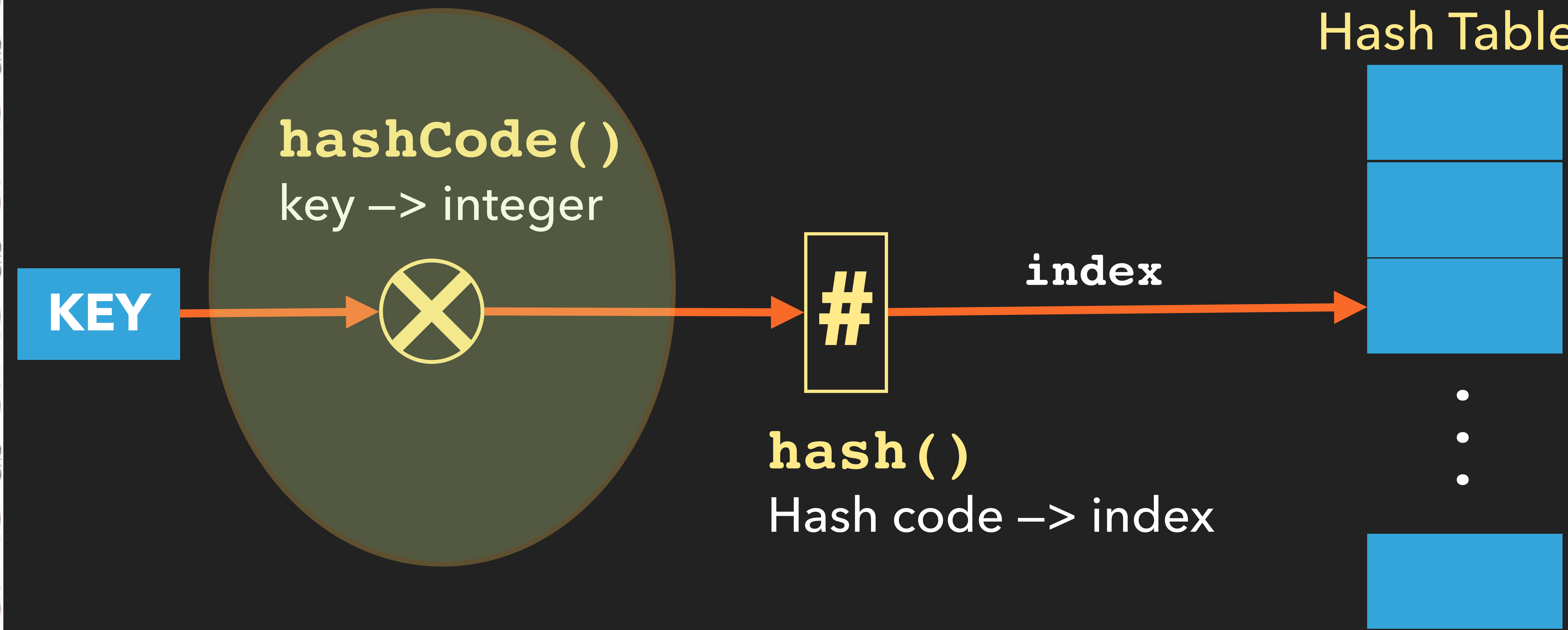
Collisions

- Quadratic Probing
  - Avoids clustering (linear probing)
  - Has its own clustering problem (secondary clustering: keys that collide with an occupied element use the same probing sequence)
  - does not guarantee finding an empty element

Hashtable Performance

- Search, Add, Remove - `O(1)`
- Three factors may cause more collisions - affect the constant time performance
    - HashCode function
    - Hash function
    - Size of the hash table

Hash Code function



Hash Code function

- `hashCode()` is simple for integers - return the integer itself
  - `hashCode()`? for double, strings, etc.
  - Class **`Object`** has a method `hashCode()` that returns the reference to the object
  - Override `hashCode()` to generate a hash code for the type you are using
  - Wrapper classes and class **`String`** override **`hashCode()`**

Hash Code function

- General guidelines for `hashCode()`
  - You should override `hashCode()` whenever you override `equals()` to ensure that two equal objects return the same hash code
  - Multiple calls to `hashCode` return the same integer provided that the object's data did not change
  - Two unequal objects may have the same hash code, but you should implement `hashCode()` to minimize such cases

Hash Code function

- `hashCode` for primitive types
  - **`byte`** (8 bits), **`short`** (16 bits), **`char`** (16 bits) - cast to **`int`** (32 bits)
  - **`float`** (32 bits)- covert to **`int`** using `Float.floatToIntBits(float f)` - returns an `integer` that corresponds to the bit representation of **`f`**

`f` = 15.25 = 1111.01 = 1.11101 x 2^3 , m=11101, e=3

0 00000011 11101000000000000000000

`floatToIntBits(f)` returns $2^{18} + 2^{20}+2^{21}+2^{22}+2^{23}+2^{24}$

Hash Code function

- `hashCode` for primitive types
  - **`long`** - fold 64 bits into 32 bits -

```
(int) (number^(number>>32))
```

```
x =        0111001010111010100000011001100111110011010101010101010101011101
x>>32=     0000000000000000000000000000000001110010101110101000000110011001
hashCode(x) =                                  10000001111011111010100110001000100
```

  - **`double`** - convert to **`long`** using `doubleToLongBits()` and fold into 32 bits

Hash Code function

✦ `hashCode` for strings

- Search keys are often strings - good `hashCode` method for strings
- Intuitive approach
  - Sum of the Unicode of all the characters in the string
  - Does not work well if two strings have the same letters

```
hashCode("tod") = hashCode("dot") = 0x0064 + 0x006F + 0x0074
```

- Better approach
  - Take the position of the character into consideration
  - Polynomial hashCode

$$hashCode(s) = s_0 \,.\, b^{n-1} + s_1 \,.\, b^{n-2} + \ldots + s_{n-1}, s_i = s\,.\,charAt(i)$$

Hash Code function

- `hashCode` for strings
  - Efficient polynomial hashCode evaluation:

$$hashCode(s) = s_{n-1} + b(s_{n-1} + b(\ldots + b(s_2 + b(s_1 + bs_0)))\ldots.)))$$

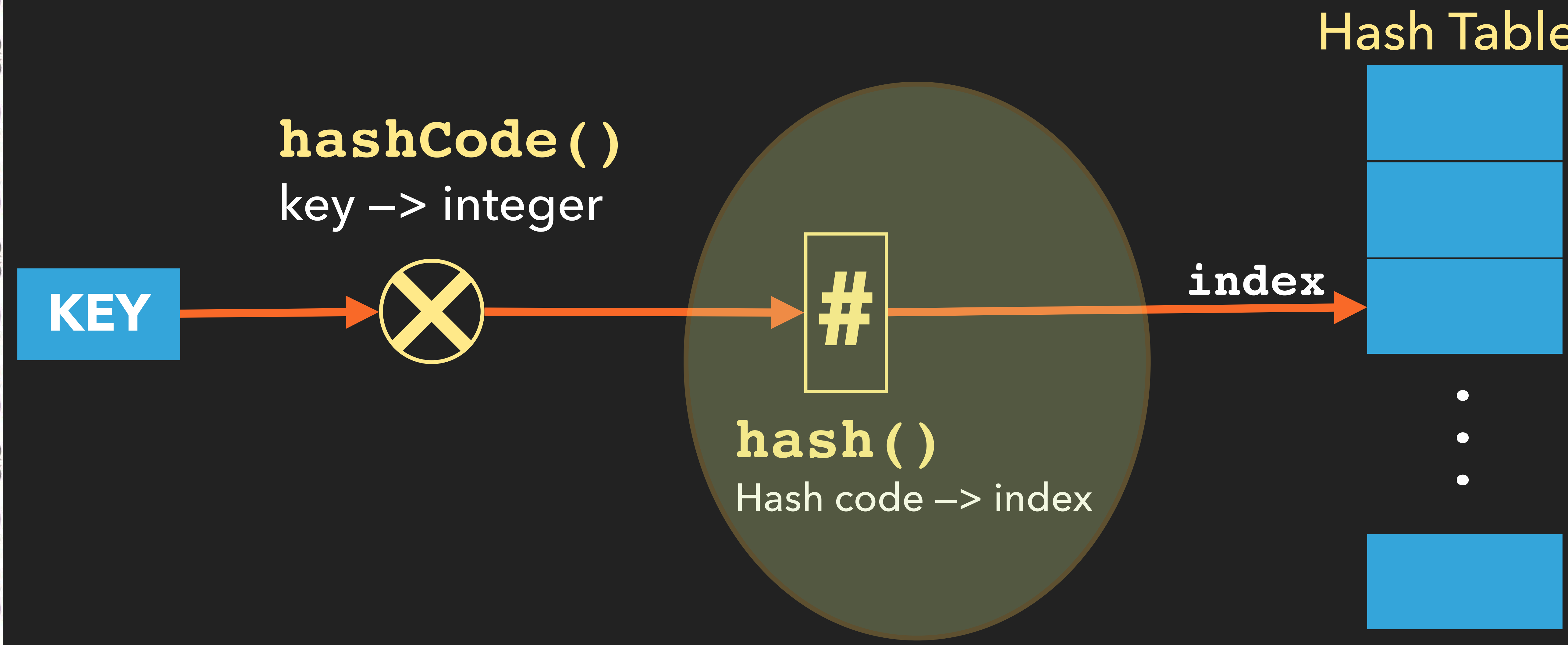
  - The weight `b` should be selected to minimize similar values for different strings. Experiments show that using `b` as `31`, `33`, `37`, `39`, and `41` minimize similar values
  - Class `String` overrides `hashCode()` method to use a polynomial hash code with `b=31`

Hash Code function

```
public static void main(String[] args){
        Integer i = 55000;
        Long l = 99123900555L;
        Double d = 5e200;
        Float fl = 2e-10f;
        Character c ='B';
        String s = "Lehigh University";
        System.out.printf("%-15s\t%-20d %-10s %d\n", "Integer i = ",
                          i,"hashCode = ",i.hashCode());
        System.out.printf("%-15s\t%-20d %-10s %d\n", "Long l = " ,
                          l ,"hashCode = ", l.hashCode());
        System.out.printf("%-15s\t%-20.0e %-10s %d\n", "Double d = " ,
                          d , "hashCode = ", d.hashCode());
        System.out.printf("%-15s\t%-20.0e %-10s %d\n", "Float f = ",
                          fl, "hashCode = ", fl.hashCode());
        System.out.printf("%-15s\t%-20c %-10s %d\n", "Character c = ",
                          c, "hashCode = ", c.hashCode());
        System.out.printf("%-15s\t%-20s %-10s %d\n", "String s = ",
                          s, "hashCode = ", s.hashCode());
    }
```

```
Integer i =      55000                 hashCode =   55000
Long l =         99123900555           hashCode =   339652764
Double d =       5e+200                hashCode =   1683429934
Float f =        2e-10                 hashCode =   794552063
Character c =    B                     hashCode =   66
String s =       Lehigh University     hashCode =   1269381427
```

Hash function



Hash Function

- **`hash()`** method
  - Transform return value of `hashCode()` (can be a large integer) into a valid index in the hash table
  - For a hash table of size N, the most common hash function is **`index = hashCode % N (0 to N-1)`**
  - **`N`** should be prime to spread the indices evenly - time consuming to find a large prime number

Hash Function

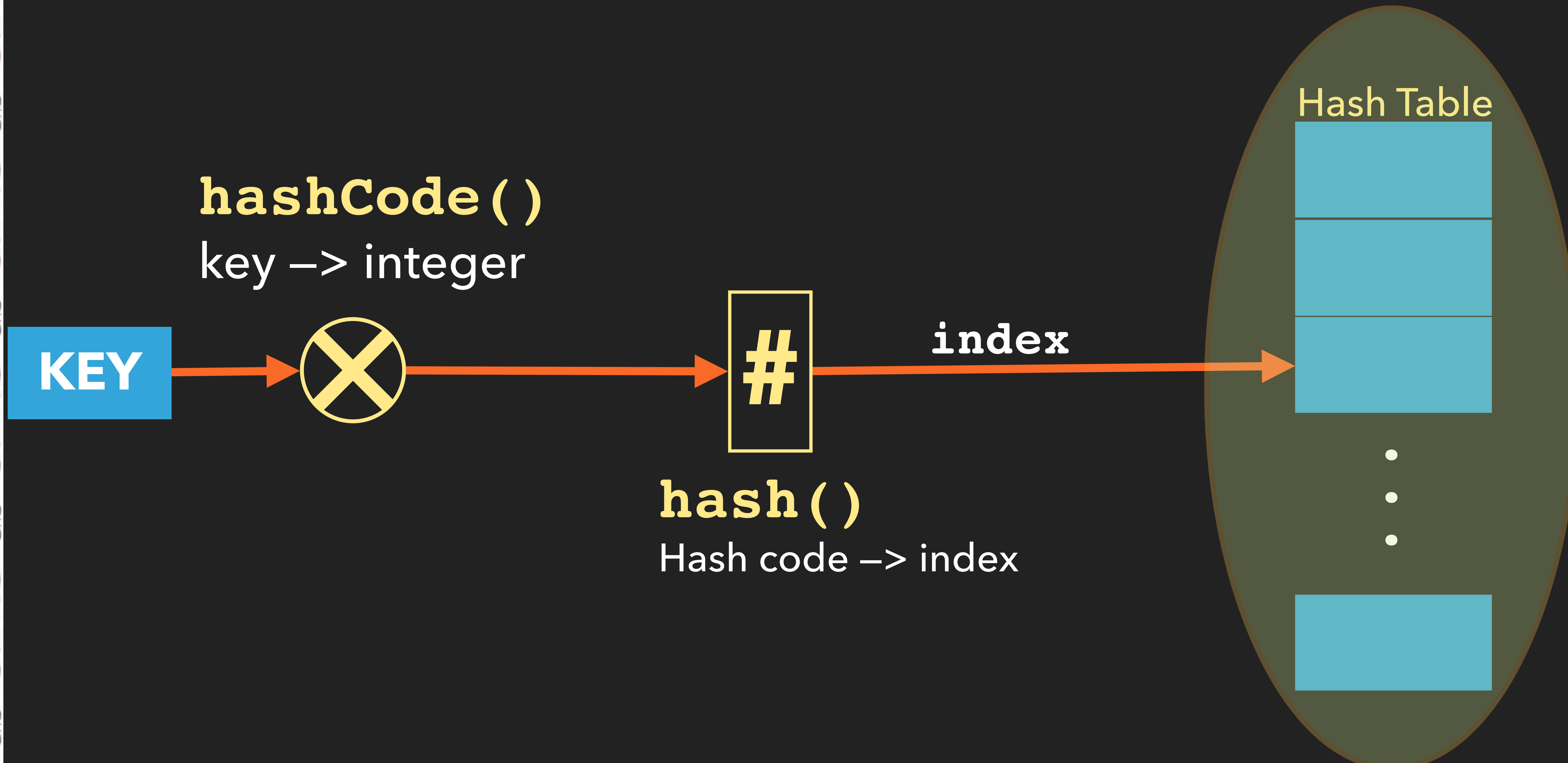
- In practice, the size of the hash table is set to an integer power of 2 to simplify the hash function operation

- `java.util.HashMap` sets the size of the hash table to a power of 2

```
hash(key) = hashCode(key) & (N-1)
```

- Bit-level operations `>>`, `^`, `&` are faster than arithmetic operations `*`, `/`, and `%`

Hash function



Size of the Hash Table

- Choosing the size of the table
  - A prime number larger than the size of the data set - may take time to find such number
  - Bigger table - less collisions - waste of memory space
  - Tradeoff - space vs. time - use power of 2 to simplify calculations

Size of the Hash Table

- Load Factor - How full is the hash table?
  - Load Factor = # of added elements / size of the HT
  - High load factor results in more collisions - requires rehashing

Size of the Hash Table

- **Rehashing** - Increase the size of the table and rehash all the data to add it to the new table
- `0.5 < load factor < 0.9` (0.5 for probing and 0.9 for chaining)

Implementation

- Hash Table with separate chaining
- Array of pointers to linked lists
- Each element in the hash table is a reference to a linked list

Implementation

```
HashMap<K, V>
-----------------------------------------------
-hashTable: LinkedList<HashMapEntry<K,V>>[]
-loadFactor: double
-size: int
-----------------------------------------------
+HashMap()
+HashMap(int capacity)
+HashMap(int capacity, double loadFactor)
-trimToPowerOf2(int capacity): int
-hash(int hashCode): int
-rehash(): void
+get(K key): V
+put(K key, V value): V
+remove(K key): void
+containsKey(K key): boolean
+size(): int
+isEmpty(): boolean
+clear(): void
+toString(): String
+toList(): ArrayList<HashMapEntry<K,V>>
```

```
HashMapEntry<K, V>
-----------------------------------------------
-key: K
-value: V
-----------------------------------------------
+HashMapEntry(K k, V v)
+getKey(): K
+getValue():V
+setKey(K k): void
+setValue(V v): void
+toString(): String
```

HashMap (Data members)

```
import java.util.ArrayList;
import java.util.LinkedList;
/**
 * Class HashMap: An implementation of the hash table
 * using separate chaining
 */
public class HashMap <K, V> {
 // data members
 private int size;
 private double loadFactor;
 private LinkedList<HashMapEntry<K,V>>[] hashTable;
```

HashMap (Constructors)

```
/**
 * Default constructor
 * default capacity: 100
 * default load factor: 0.9
 */
  public HashMap() {
    this(100, 0.9);
  }
/**
 * Constructor with one parameter
 * @param c for the capacity
 * default load factor: 0.9
 */
  public HashMap(int c) {
    this(c, 0.9);
  }
/**
 * Constructor with two parameters
 * @param c for the capacity
 * @param lf for the load factor
 */
  public HashMap(int c, double lf) {
    hashTable = new LinkedList[trimToPowerOf2(c)];
    loadFactor = lf;
    size = 0;
  }
```

HashMap (private methods)

```
/**
 * Method trimToPowerOf2
 * @param c the capacity of the hash table
 * @return the closest power of 2 to c
 */
  private int trimToPowerOf2(int c) {
    int capacity = 1;
    while (capacity < c)
       capacity  = capacity << 1;
    return capacity;
  }
/**
 * Method hash
 * @param hashCode
 * @return a valid index in the hashtable
 */
  private int hash(int hashCode) {
    return hashCode & (hashTable.length-1);
  }
```

HashMap (useful methods)

```
/**
 * Method size
 * @return the number of pairs (key, value) stored the hashtable
 */
  public int size() {
    return size;
  }
/**
 * Method clear to clear the hashtable
 */
  public void clear() {
    size = 0;
    for(int i=0; i<hashTable.length; i++)
      if(hashTable[i] != null)
         hashTable[i].clear();
  }
/**
 * Method isEmpty
 * @return true if the hashtable is empty, false otherwise
 */
  public boolean isEmpty() {
    return (size == 0);
  }
```

HashMap (search methods)

```
/**
 * Method contains to search for a key in the hashtable
 * @param key the value of the key being searched for
 * @return true if key was found, false otherwise
 */
  public boolean containsKey(K key) {
    if(get(key) != null)
      return true;
    return false;
  }
/**
 * Method get to find an entry in the hashtable
 * @param key the value of the key being searched for
 * @return the value associated with the key if key is found, null otherwise
 */
  public V get(K key) {
    int HTIndex = hash(key.hashCode());
    if(hashTable[HTIndex] != null) {
      LinkedList<HashMapEntry<K,V>> ll = hashTable[HTIndex];
      for(HashMapEntry<K,V> entry: ll) {
        if(entry.getKey().equals(key))
          return entry.getValue();
      }
    }
    return null;
```

HashMap (remove method)

```
/**
 * Method remove to remote an entry from the hashtable
 * @param key the key to be removed
 * if the key is found, the pair (key and it associated value)
 * will be removed from the hashtable
 * the hashtable is not modified if key is not found
 */
  public void remove(K key) {
    int HTIndex = hash(key.hashCode());
    if (hashTable[HTIndex]!=null) { //key is in the hash map
      LinkedList<HashMapEntry<K,V>> ll = hashTable[HTIndex];
      for(HashMapEntry<K,V> entry: ll) {
        if(entry.getKey().equals(key)) {
          ll.remove(entry);
          size--;
          break;
        }
      }
    }
  }
```

HashMap (put method)

```
/**
 * Method put to add a new entry to the hashtable
 * @param key the value of the key of the new entry
 * @param value the value associated with the key
 * @return the old value of the entry if an entry is found for key
 *         or the new value if a new entry was added to the hashtable
 */
 public V put(K key, V value) {
   // check if the key is already in the hashtable
   // modify its associated value if key is found
   if(get(key) != null) {
     int HTIndex = hash(key.hashCode());
      LinkedList<HashMapEntry<K,V>> ll = hashTable[HTIndex];
      for(HashMapEntry<K,V> entry: ll) {
        if(entry.getKey().equals(key)) {
           V old = entry.getValue();
           entry.setValue(value);
           return old;
        }
      }
   }
   // key was not found. Check if rehashing is needed before adding a new entry
   if(size >= hashTable.length * loadFactor)
      rehash();
   // Add a new entry to the hashtable
   int HTIndex = hash(key.hashCode());
   if(hashTable[HTIndex] == null){
      hashTable[HTIndex] = new LinkedList<>();
   }
   hashTable[HTIndex].add(new HashMapEntry<>(key, value));
   size++;
   return value;
 }
```

HashMap (rehash method)

```
/**
 * Method rehash
 * creates a new hashtable with double capacity
 * puts all the entries from the old hashtable into the new table
 */
  private void rehash() {
    ArrayList<HashMapEntry<K,V>> list = toList();
    hashTable = new LinkedList[hashTable.length << 1];
    size = 0;
    for(HashMapEntry<K,V> entry: list)
       put(entry.getKey(), entry.getValue());

  }
```

HashMap (toList and toString methods)

```
/**
 * Method toList
 * @return an arraylist with all the entries in the hashtable
 */
  public ArrayList<HashMapEntry<K,V>> toList(){
    ArrayList<HashMapEntry<K,V>> list = new ArrayList<>();
    for(int i=0; i< hashTable.length; i++) {
      if(hashTable[i]!= null) {
        LinkedList<HashMapEntry<K,V>> ll = hashTable[i];
        for(HashMapEntry<K,V> entry: ll)
           list.add(entry);
      }
    }
    return list;
  }
/**
 * Method toString
 * @return a formatted string with all the entries in the hashtable
 */
  public String toString() {
    String out = "[";
    for(int i=0; i<hashTable.length; i++) {
      if(hashTable[i]!=null) {
         for(HashMapEntry<K,V> entry: hashTable[i])
            out += entry.toString();
         out += "\n";
      }
    }
    out += "]";
    return out;
  }
```

Testing HashMap

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class TestHashMap{
    public static void main(String[] args){
        HashMap<String, String> states = new HashMap<>();
        readStates(states, "states.txt");
        System.out.println(states);

    }
    public static void readStates(HashMap<String,String> hm, String filename){
        try{
            Scanner read = new Scanner(new File(filename));
            while(read.hasNextLine()){
                String line = read.nextLine();
                String[] tokens = line.split("\\|");
                String state = tokens[0];
                String capital = tokens[1];
                hm.put(state, capital);
            }
            read.close();
        }
        catch(FileNotFoundException e){
            System.out.println("File not found");
        }
    }
}
```

Performance of the HashMap

Operation	Complexity	Operation	Complexity
`HashMap()`	**O(1)**	`isEmpty()`	**O(1)**
`HashMap(int)`	**O(log n)**	`containsKey(K)`	**O(1)**
`HashMap(int, double)`	**O(log n)**	`get(K)`	**O(1)**
`trimToPowerOf2(int)`	**O(log n)**	`put(K, V)`	**O(1)-O(n)**
`hash(int)`	**O(1)**	`remove(K)`	**O(1)**
`rehash()`	**O(n)**	`toList()`	**O(n)**
`size()`	**O(1)**	`toString()`	**O(n)**

Summary

- Hash Tables
  - Efficient search operation (`O(1)`)
  - Performance affected by:
    - Hash Code function - Hash function - size of the table (load Factor and rehashing)
    - Collisions
      - Chaining
      - Probing (linear, quadratic, double hashing)
- Implementation of the HashMap data structure