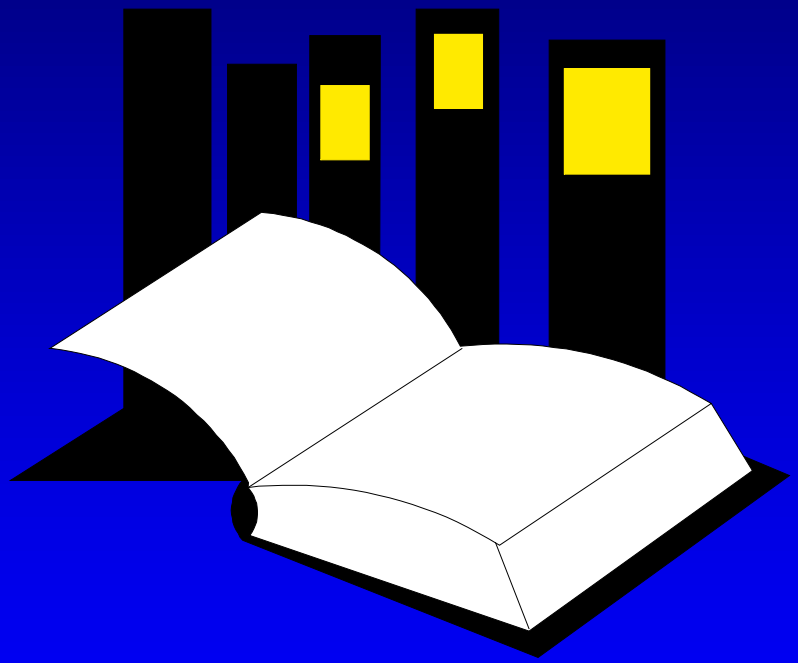
Hash Tables



Data Structures and Other Objects Using C++

This Chapter discusses several ways of storing information in an array, and later searching for the information.

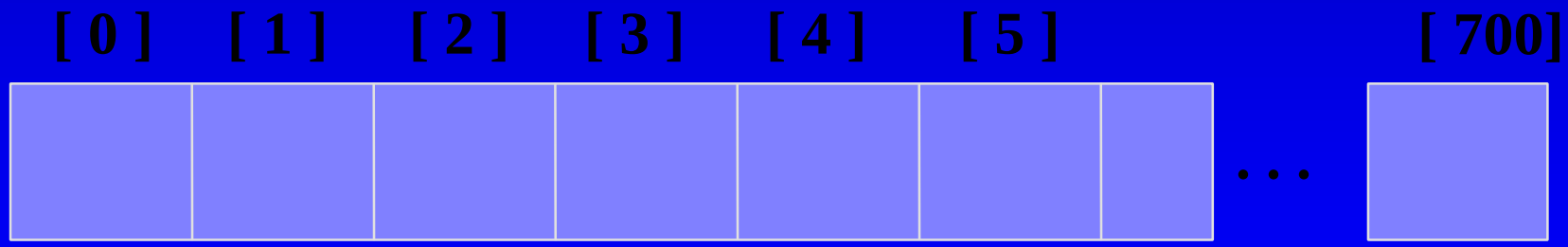
Hash tables are a common approach to the storing/searching problem.

This presentation introduces hash tables.

What is a Hash Table ?

The simplest kind of hash table is an array of records.

This example has 701 records.

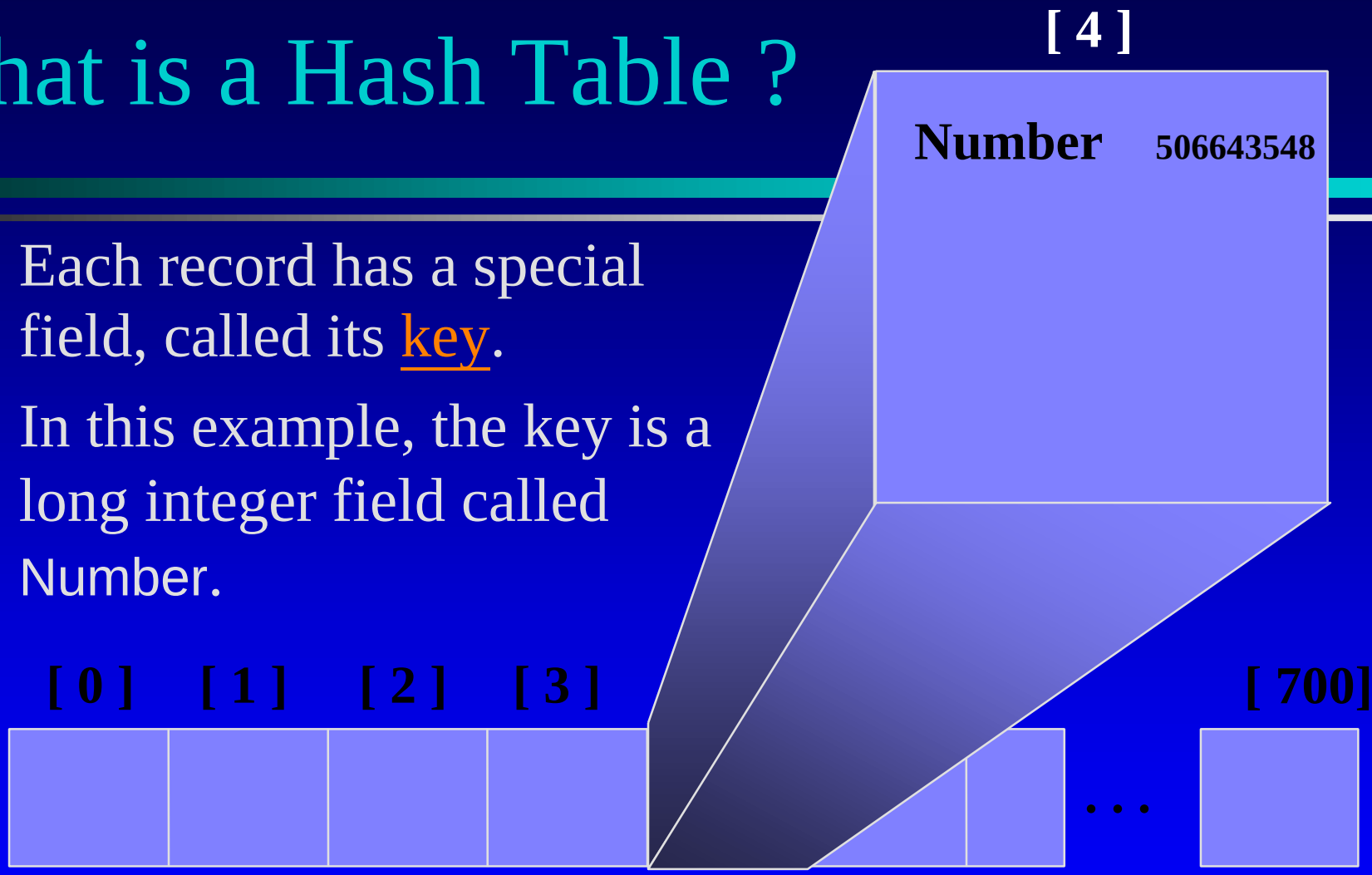


An array of records

What is a Hash Table ?

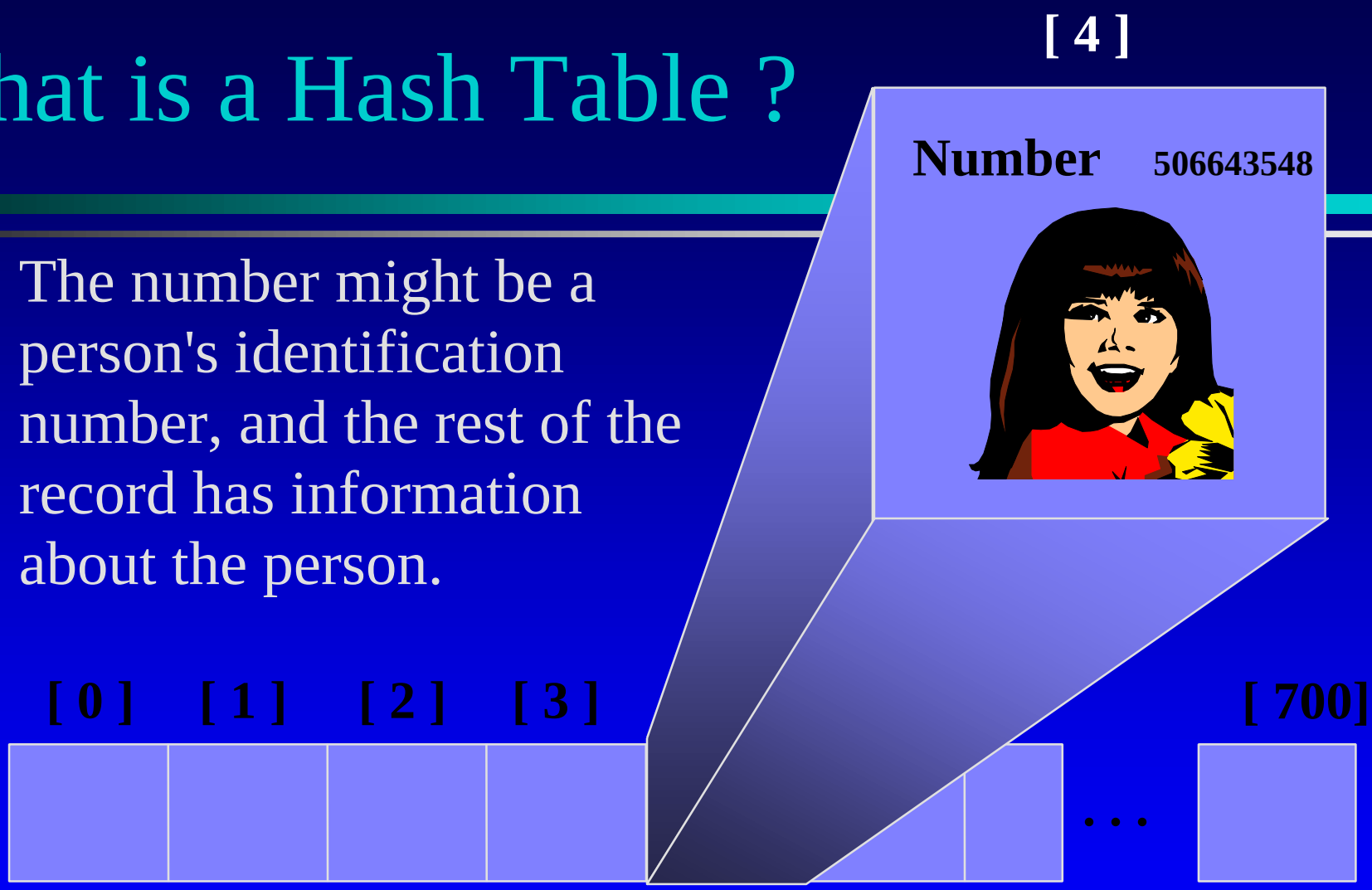
Each record has a special field, called its key.

In this example, the key is a long integer field called Number.



What is a Hash Table ?

The number might be a person's identification number, and the rest of the record has information about the person.



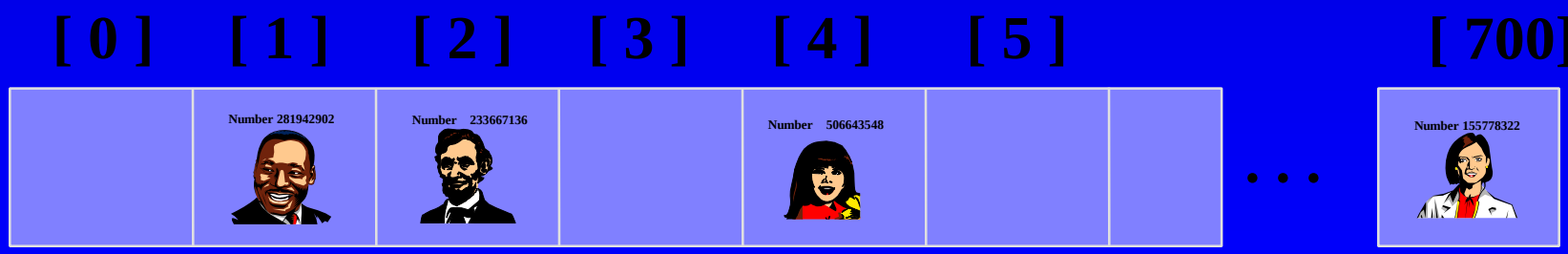
What is a Hash Table ?

When a hash table is in use, some spots contain valid records, and other spots are "empty".

Inserting a New Record

In order to insert a new record, the **key** must somehow be **converted to** an array **index**.

The index is called the **hash value** of the key.



Inserting a New Record

Typical way create a hash value:

(Number mod 701)

What is (580625685 mod 701) ?

Inserting a New Record

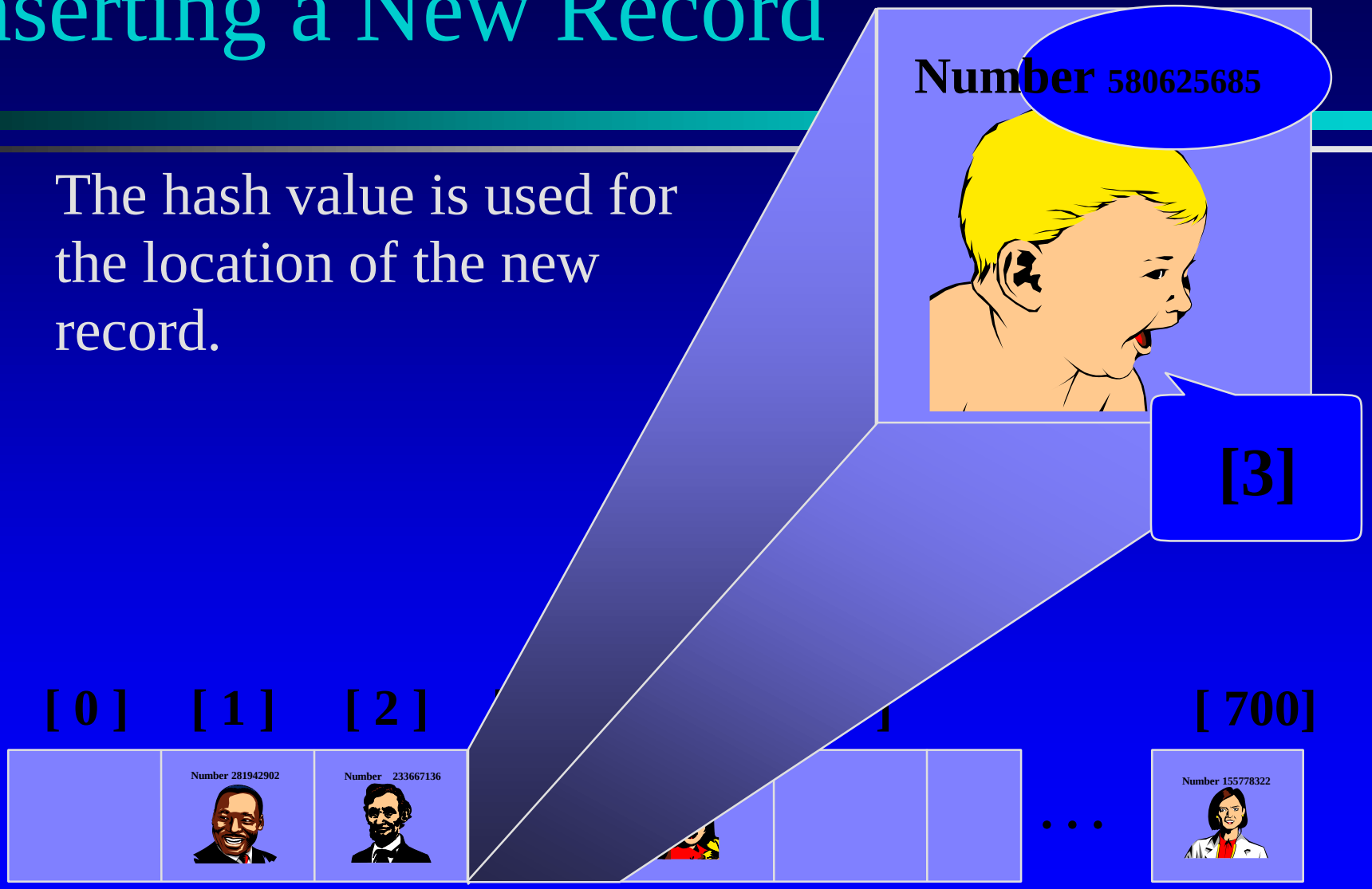
Typical way to create a hash value:

(Number mod 701)

What is (580625685 mod 701) ?

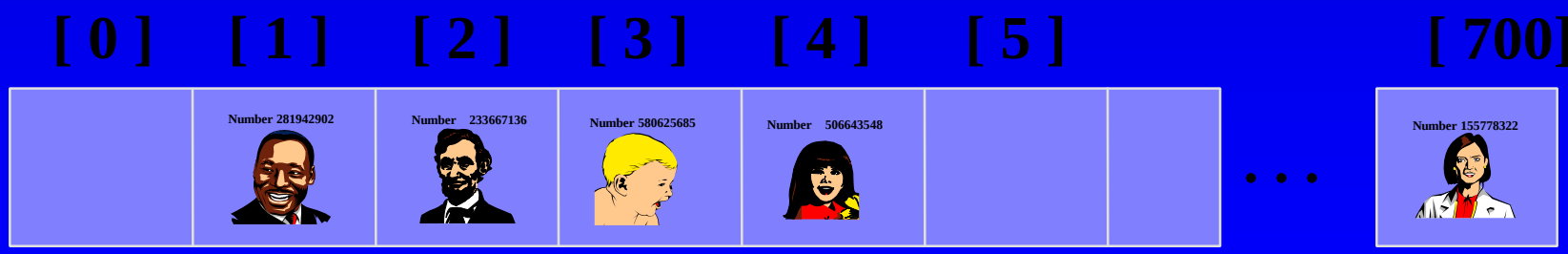
Inserting a New Record

The hash value is used for the location of the new record.



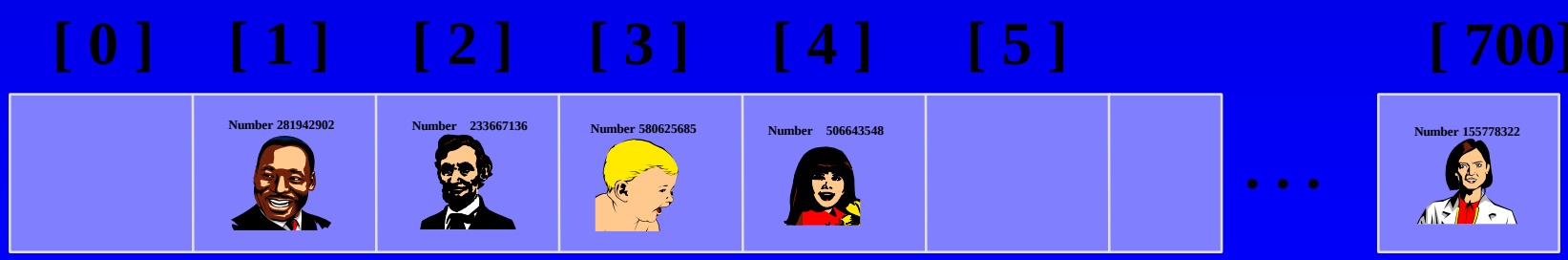
Inserting a New Record

The hash value is used for the location of the new record.



Collisions

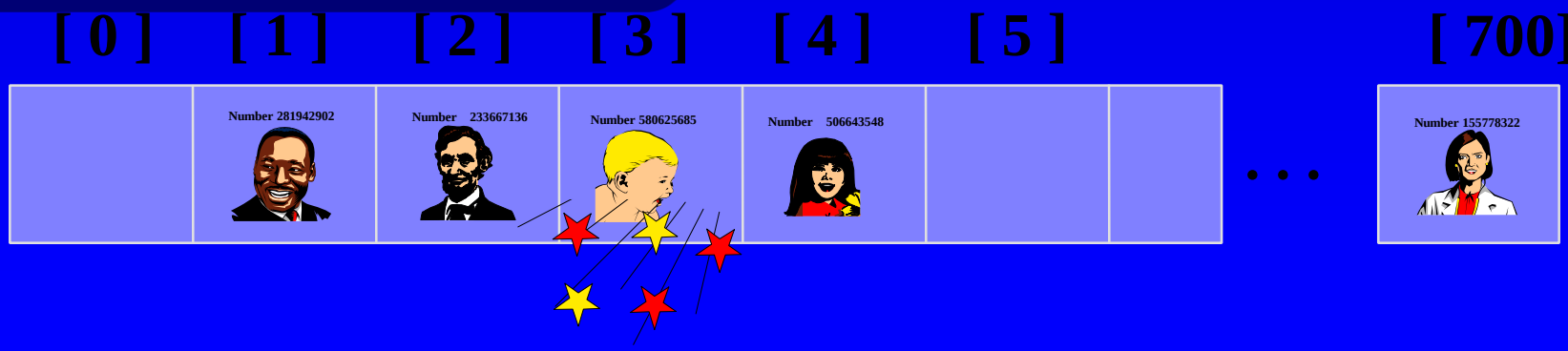
Here is another new record to insert, with a hash value of 2.



Collisions

This is called a **collision**, because there is already another valid record at [2].

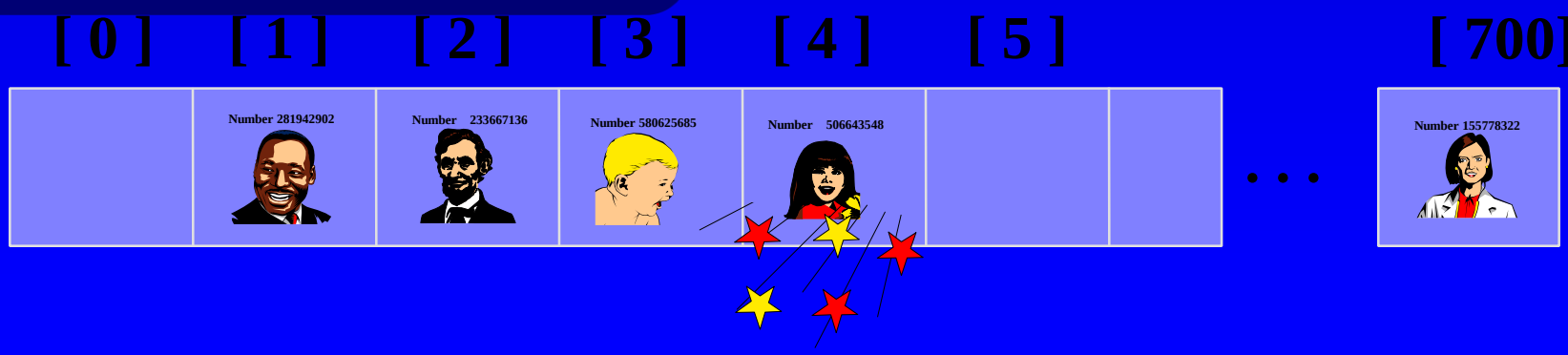
When a collision occurs, move forward until you find an empty spot.



Collisions

This is called a **collision**, because there is already another valid record at [2].

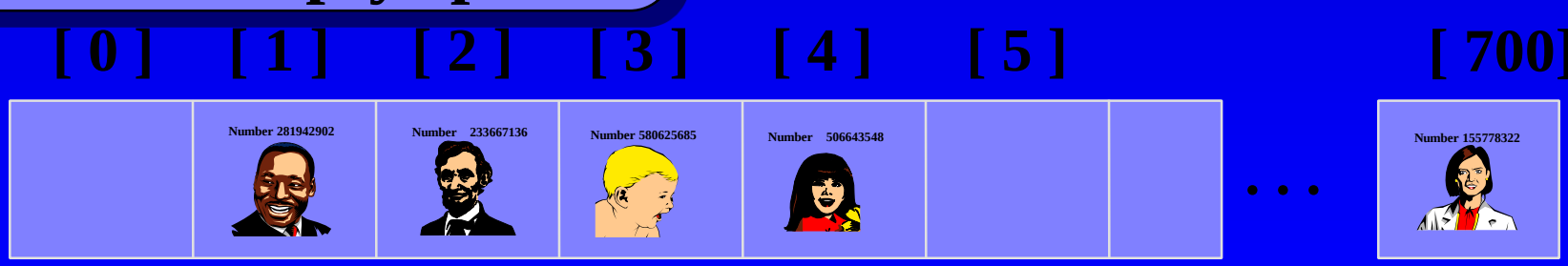
When a collision occurs, move forward until you find an empty spot.



Collisions

This is called a **collision**, because there is already another valid record at [2].

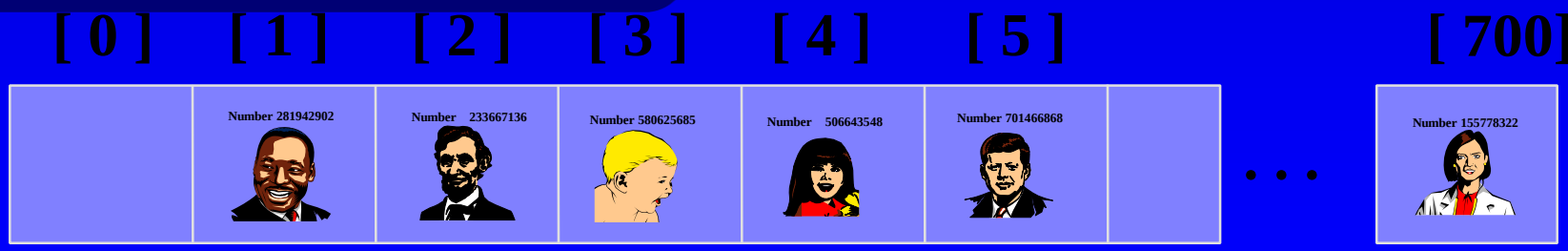
When a collision occurs, move forward until you find an empty spot.



Collisions

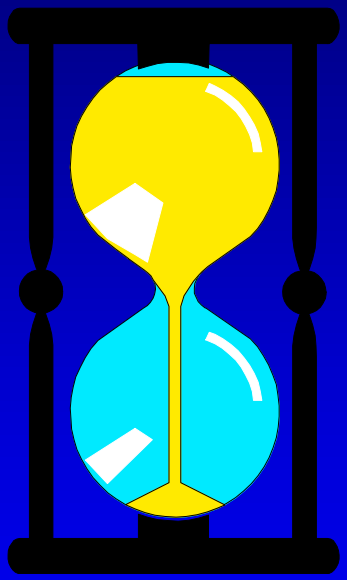
This is called a **collision**, because there is already another valid record at [2].

The new record goes in the empty spot.



A Quiz

Where would you be placed in this table, if there is no collision? Use your social security number or some other favorite number.



[0]	[1]	[2]	[3]	[4]	[5]		[700]
	Number 281942902	Number 233667136	Number 580625685	Number 506643548	Number 701466868		Number 155778322

Searching for a Key

The data that's attached to a key can be found fairly quickly.

Number 701466868

[0] [1] [2] [3] [4] [5] [700]

Searching for a Key

Calculate the hash value.

Check that location of the array for the key.

Number 701466868

My hash value is [2].

Not me.

[0] **[1]** **[2]** **[3]** **[4]** **[5]** **[700]**

Searching for a Key

Keep moving forward until you find the key, or you reach an empty spot.

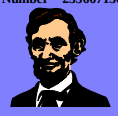
Number 701466868

My hash value is [2].

Not me.

[0] **[1]** **[2]** **[3]** **[4]** **[5]** **[700]**

	Number 281942902	Number 233667136	Number 580625685	Number 506643548	Number 701466868		. . .	Number 155778322



Searching for a Key

Keep moving forward until you find the key, or you reach an empty spot.

Searching for a Key

Keep moving forward until you find the key, or you reach an empty spot.

Number 701466868

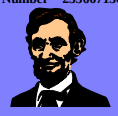
My hash value is [2].

Yes!

[0] **[1]** **[2]** **[3]** **[4]** **[5]** **[700]**

Number 281942902

Number 233667136



Number 580625685

Number 506643548

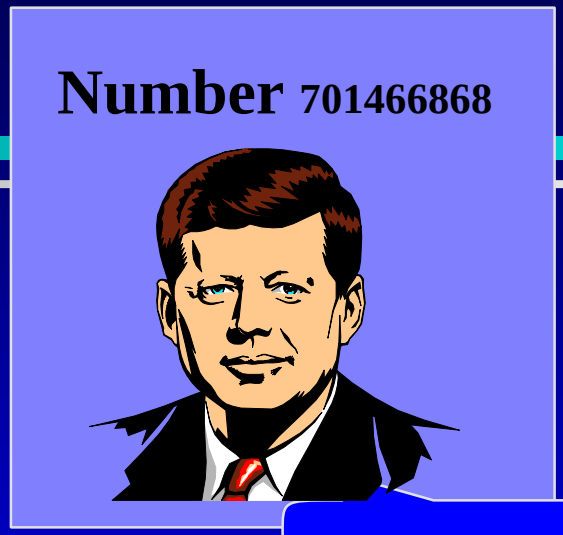
Number 701466868

. . .

Number 155778322

Searching for a Key

When the item is found, the information can be copied to the necessary location.



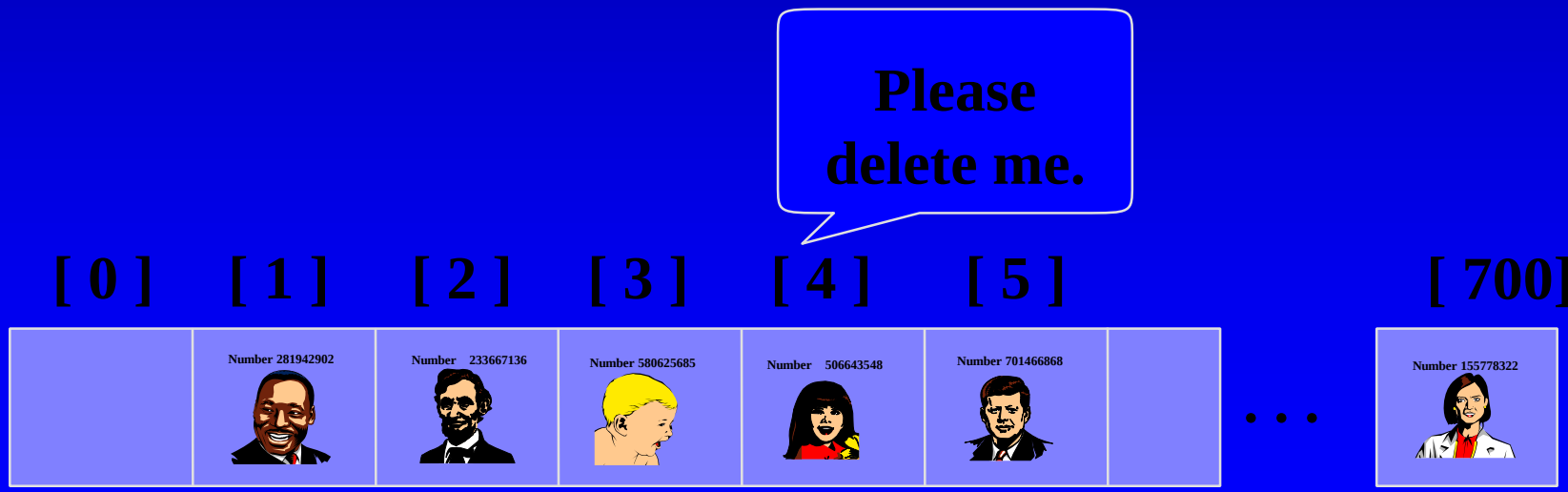
My hash value is [2].

Yes!

[0] [1] [2] [3] [4] [5] [700]

Deleting a Record

Records may also be deleted from a hash table.



Deleting a Record

Records may also be deleted from a hash table.

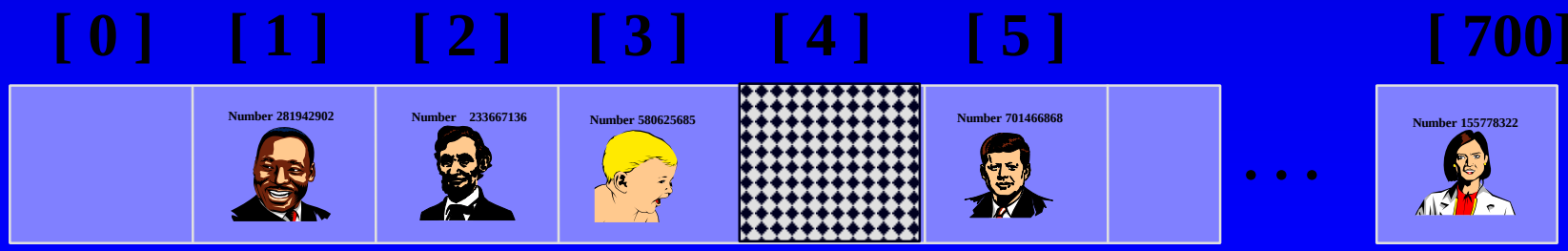
But the location must not be left as an ordinary "empty spot" since that could interfere with searches.

Deleting a Record

Records may also be deleted from a hash table.

But the location must not be left as an ordinary "empty spot" since that could interfere with searches.

The location must be marked in some special way so that a search can tell that the spot used to have something in it.



Summary

Hash tables store a collection of records with keys.

The location of a record depends on the hash value of the record's key.

When a collision occurs, the next available location is used.

Searching for a particular key is generally quick.

When an item is deleted, the location must be marked in a special way, so that the searches know that the spot used to be used.

Implementing Dynamic Dictionaries

- Want a data structure in which finds/searches are very fast
  - As close to O(1) as possible
  - minimum number of executed instructions per method
- Insert and Deletes should be fast too
- Objects in dictionary have unique keys
  - A key may be a single property/attribute value
  - Or may be created from multiple properties/values

Hash tables vs. Other Data Structures

- We want to implement the dictionary operations Insert(), Delete() and Search()/Find() efficiently.
- Arrays:
  - can accomplish in O(1) time
  - but are not space efficient (assumes we leave empty space for keys not currently in dictionary)
- Binary search trees
  - can accomplish in O(log n) time
  - are space efficient.
- Hash Tables:
  - A generalization of an array that under some reasonable assumptions is O(1) for Insert/Delete/Search of a key

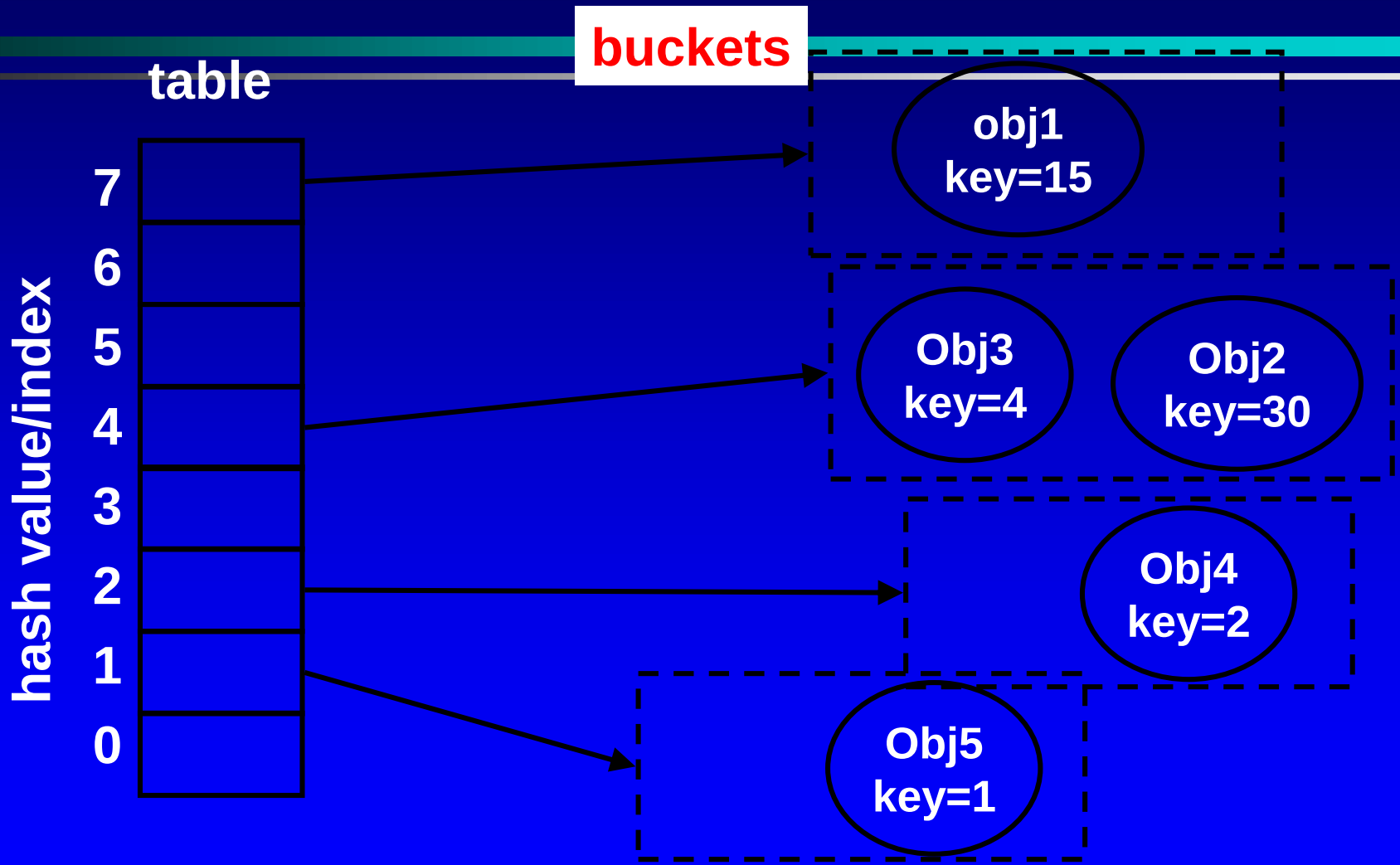
Array Approach – example

- A social security application keeping track of people where the primary search key is a person's social security number (SSN)
- You can use an array to hold references to all the person objects
  - Use an array with range 0 - 999,999,999
  - Using the SSN as a key, you have O(1) access to any person object
- Unfortunately, the number of active keys (Social Security Numbers) is much less than the array size (1 billion entries)
  - Est. US population, Oct. 20th 2004: 294,564,209
  - Over 60% of the array would be unused

Hash Tables

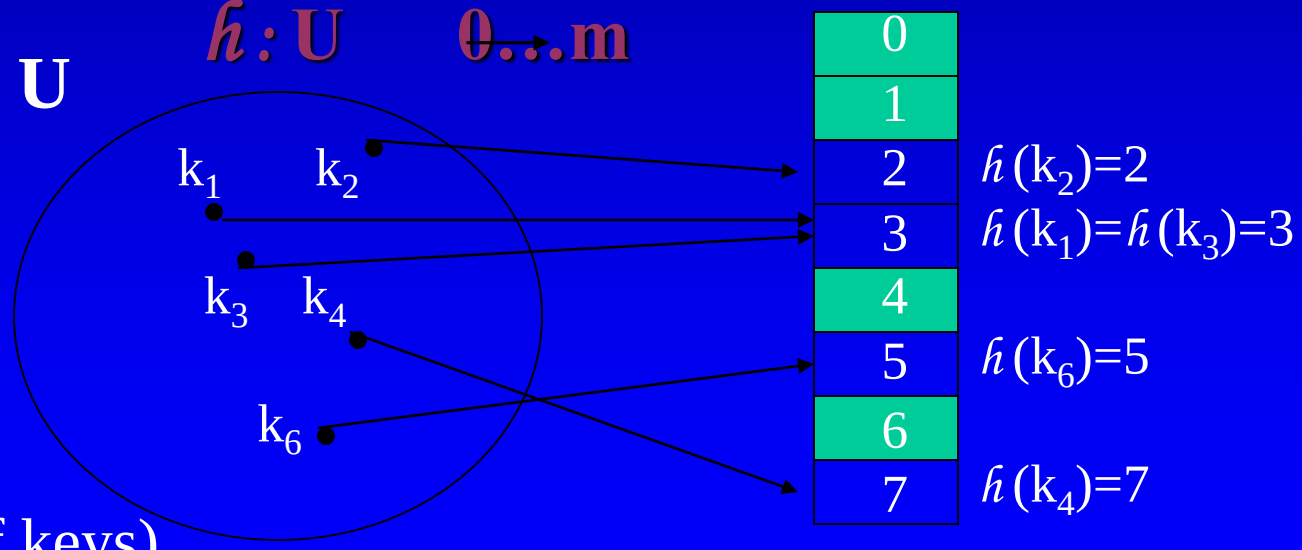
- Very useful data structure
  - Good for storing and retrieving key-value pairs
  - Not good for iterating through a list of items
- Example applications:
  - Storing objects according to ID numbers
    - When the ID numbers are widely spread out
    - When you don’t need to access items in ID order

Hash Tables – Conceptual View



Hash Tables

- **Hash Tables** solve these problems by using a much smaller array and mapping keys with a **hash function.**
- Let universe of keys U and an array of size m. A *hash function* h is a function from U to 0…m, that is:



(universe of keys)

Hash index/value

- A *hash value* or *hash index* is used to index the hash table (array)
- A hash function takes a *key* and returns a hash value/index
  - The hash index is a integer (to index an array)
- The key is specific value associated with a specific object being stored in the hash table
  - It is important that the key remain constant for the lifetime of the object

Hash Function

- You want a hash function/algorithm that is:
  - Fast
  - Creates a good distribution of hash values so that the items (based on their keys) are distributed evenly through the array
- Hash functions can use as input
  - Integer key values
  - String key values
  - Multipart key values
    - Multipart fields, and/or
    - Multiple fields

The mod function

- Stands for **modulo**
- When you divide x by y, you get a result and a remainder
- Mod is the remainder
  - 8 mod 5 = 3
  - 9 mod 5 = 4
  - 10 mod 5 = 0
  - 15 mod 5 = 0
- Thus for key-value mod M, multiples of M give the same result, 0
  - But multiples of other numbers do not give the same result
  - So what happens when M is a prime number where the keys are not multiples of M?

Hash Tables: Insert Example

For example, if we hash keys 0…1000 into a hash table with 5 entries and use ***h*(key) = key mod 5** , we get the following sequence of events:

Insert 2

	key	data
0		
1		
2	2	…
3		
4		

Insert 21

	key	data
0		
1	21	…
2	2	…
3		
4		

Insert 34

	key	data
0		
1	21	…
2	2	…
3		
4	34	…

Insert 54

There is a ***collision*** at array entry #4

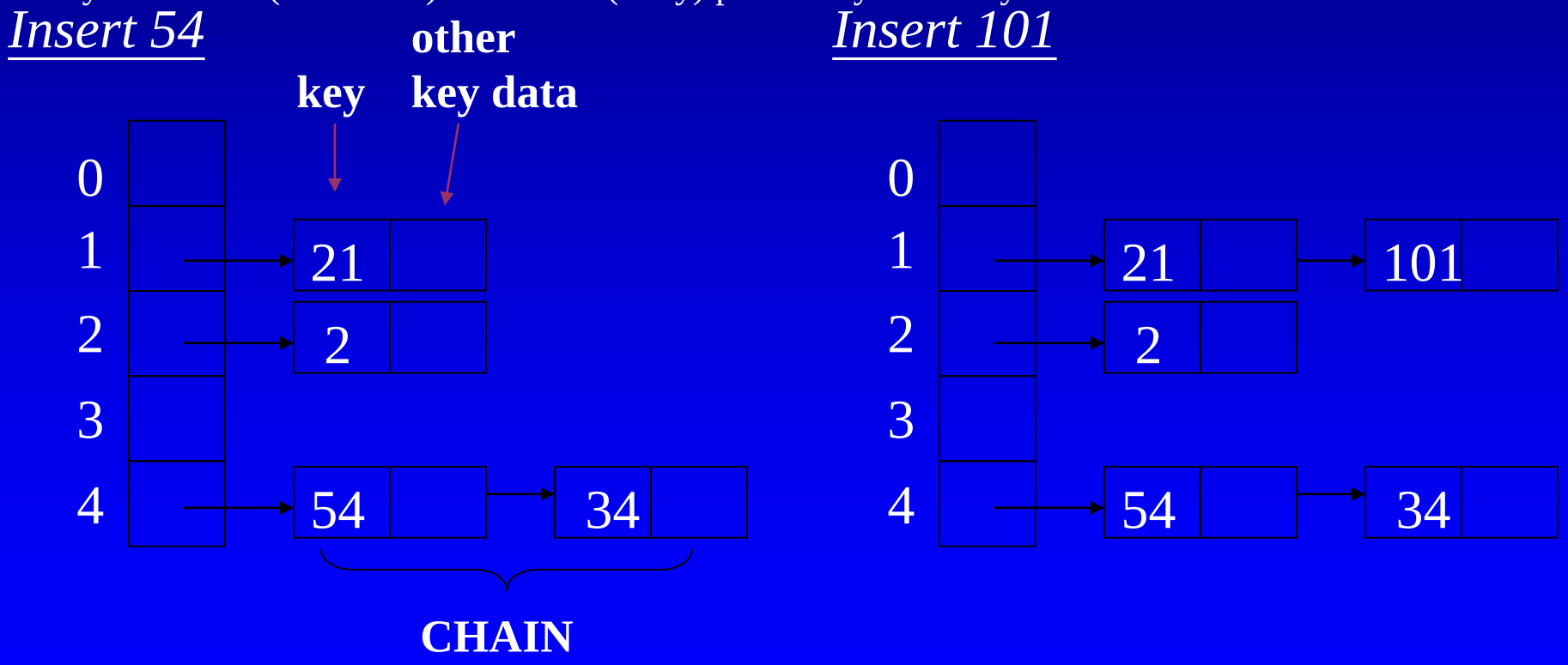
???

Dealing with Collisions

- A problem arises when we have two keys that hash in the same array entry – this is called a **collision**.
- There are two ways to resolve collision:
  - **Hashing with Chaining (a.k.a. "Separate Chaining"):** every hash table entry contains a pointer to a linked list of keys that hash in the same entry
  - **Hashing with Open Addressing:** every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key

Hashing with Chaining

The problem is that keys 34 and 54 hash in the same entry (4). We solve this *collision* by placing all keys that hash in the same hash table entry in a **chain** (linked list) **or bucket** (array) pointed by this entry:



Hashing with Chaining

- What is the running time to insert/search/delete?
  - **Insert:** It takes O(1) time to compute the hash function and insert at head of linked list
  - **Search:** It is proportional to max linked list length
  - **Delete:** Same as search
- Therefore, in the unfortunate event that we have a "bad" hash function all *n* keys may hash in the same table entry giving an O(*n*) run-time!

So how can we create a "good" hash function?

Choosing a Hash Function – 1

- The performance of the hash table depends on having a hash function that evenly distributes the keys: *uniform hashing* is the ideal target
- Choosing a good hash function requires taking into account the kind of data that will be used.
  - The statistics of the key distribution needs to be accounted for
  - E.g., Choosing the first letter of a last name will likely cause lots of collisions depending on the nationality of the population
- Most programming languages have hash functions built in

Choosing a Hash Function – 2

- Division/modulo method
  - key mod m
  - m is the array size; in general, it should be prime number
- Multiplication method
  - Floor ((key*someFraction mod 1)*arraySize)
  - Where some fraction is typically 0.618

Prime Number Distribution

- For example, assume
- Keys (key values) are multiples of 5
  - 5, 10, 15, 20, 25…
- The keys are evenly distributed 5 to 245
- An M (the divisor) of 7

- Then, the hash values will be evenly distributed from 0 to 6 for the keys
  - See table →
- If M was 5, then you would have what kind of distribution?

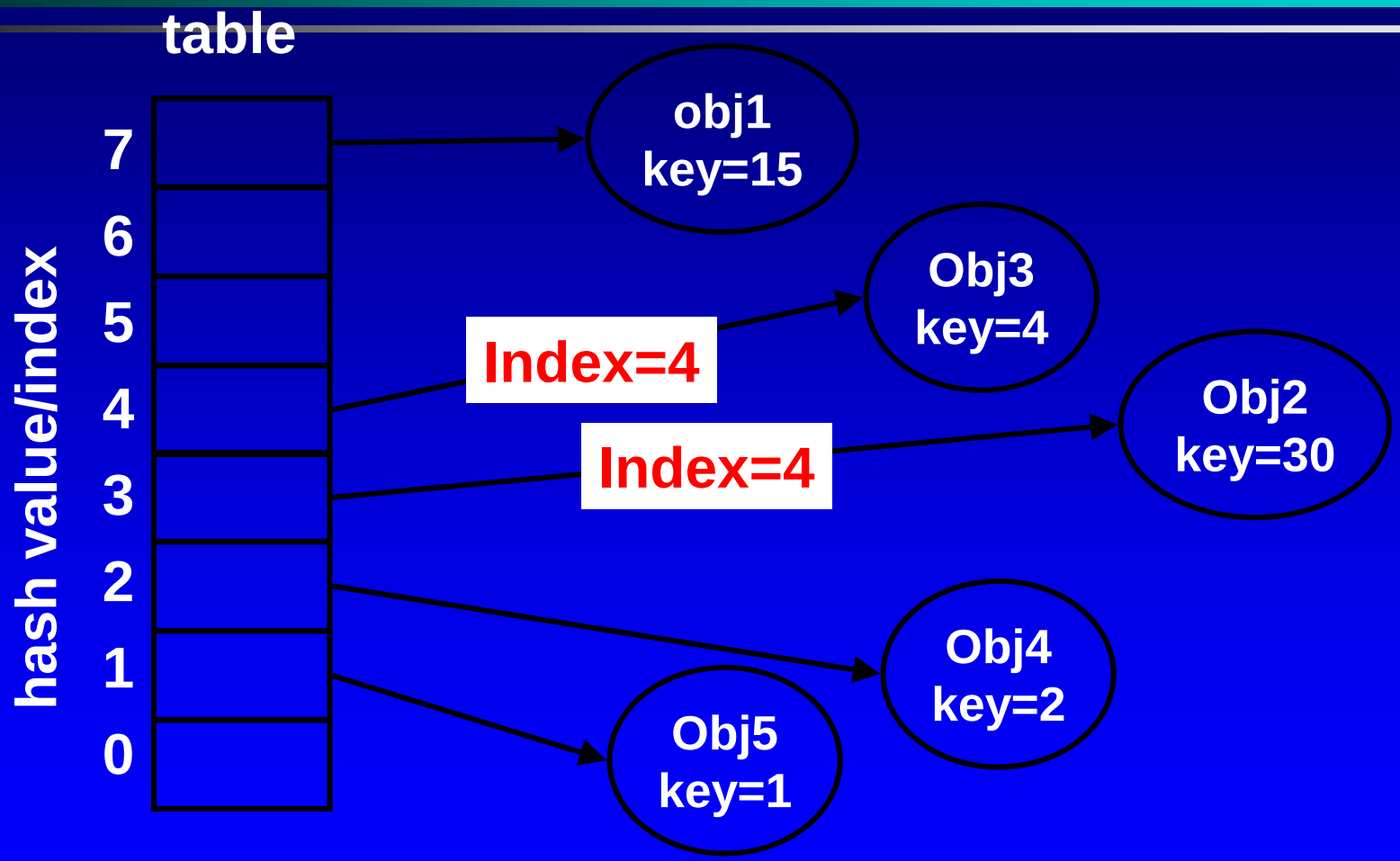
Key mod M	Total
0	7
1	7
2	7
3	7
4	7
5	7
6	7
(blank)	
Grand Total	49

hash value = key mod m
(m is typically the table size)

Choosing Hash Function – 3

- If keys are non-random – e.g. part numbers
  - Use all data to contribute to the hash function to get a better distribution
  - Consider *folding* – sum the natural (or arbitrary) groups of digits in key
  - Don't use redundant or non-data (.e.g. checksum values)
  - Do not use information that might change!
- ➔ Analyze your expected key values (or some representative subset) to make sure your hash function gives a good distribution!

Hash Tables – Open Addressing



Hashing with Open Addressing

- So far we have studies hashing with chaining, using a list to store the items that hash to the same location
- Another option is to store all the items (references to single items) directly in the table.
- **Open addressing**
  - collisions are resolved by systematically examining other table indexes, i_0 , i_1 , i_2 , … until an empty slot is located.

Open Addressing

- The key is first mapped to an array cell using the hash function (e.g. key % array-size)
- If there is a collision find an available array cell
- There are different algorithms to find (to probe for) the next array cell
  - Linear
  - Quadratic
  - Double Hashing

Probe Algorithms (Collision Resolution)

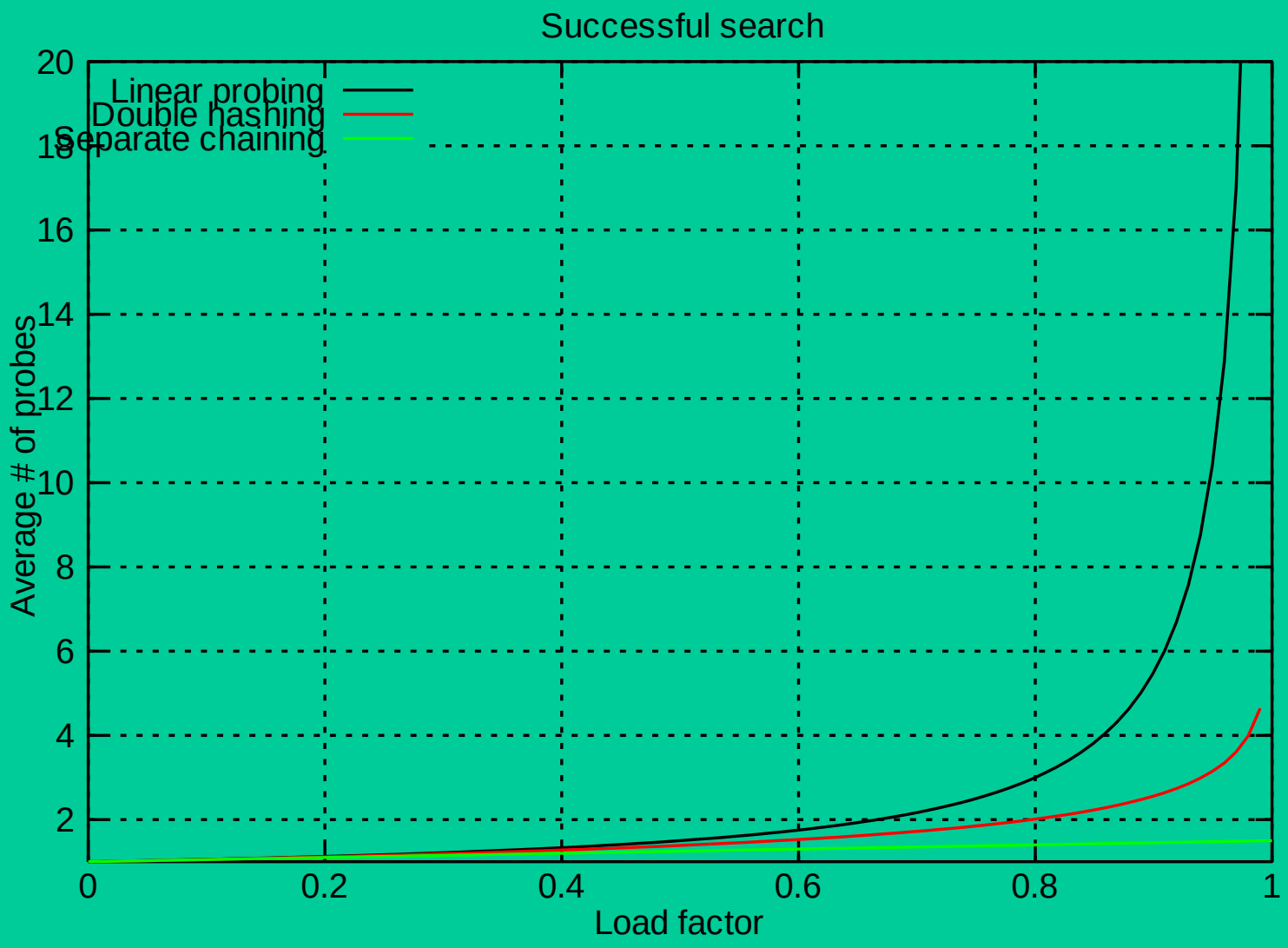
- Linear Probing
  - Choose the next available array cell
    - First try arrayIndex = hash value + 1
    - Then try arrayIndex = hash value + 2
    - Be sure to wrap around the end of the array!
    - arrayIndex = (arrayIndex + 1) % arraySize
    - Stop when you have tried all possible array indices
  - If the array is full, you need to throw an exception or, better yet, resize the array
- Quadratic Probing
  - Variation of linear probing that uses a more complex function to calculate the next cell to try

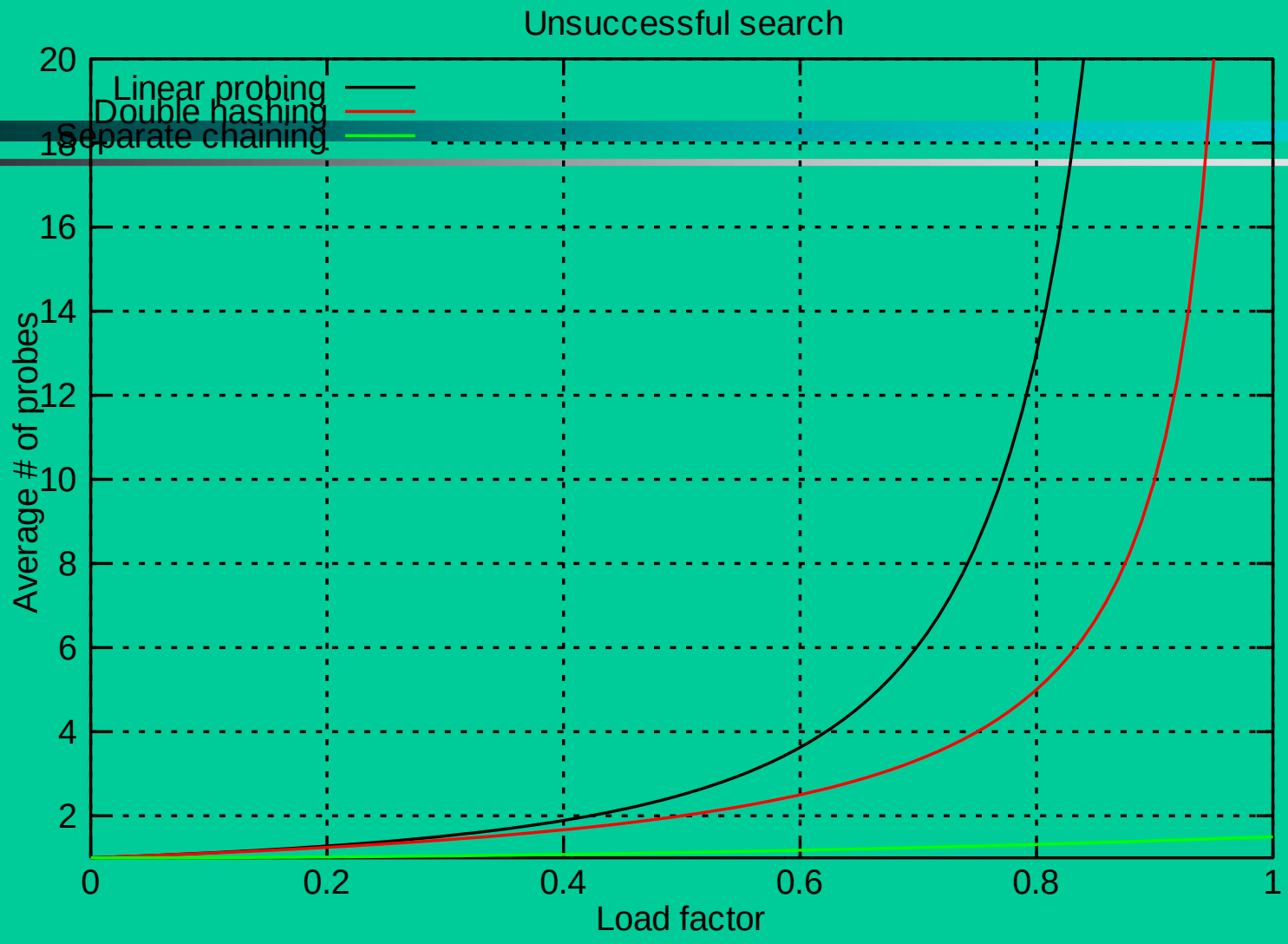
Double Hashing

- Apply a second hash function after the first
- The second hash function, like the first, is dependent on the key
- Secondary hash function must
  - Be different than the first
  - And, obviously, not generate a zero
- Good algorithm:
  - arrayIndex = (arrayIndex + stepSize) % arraySize;
  - Where stepSize = constant – (key % constant)
  - And constant is a prime less than the array size

Load Factor

- Understanding the expected load factor will help you determine the efficiency of you hash table implementation and hash functions
- Load factor = number of items in hash table / array size
- For Open Addressing:
  - If < 0.5, wasting space
  - If > 0.8, overflows significant
- For Chaining:
  - If < 1.0, wasting space
  - If > 2.0, then search time to find a specific item may factor in significantly to the [relative] performance


Successful search
Linear probing
Double hashing
Separate chaining
Average # of probes
Load factor
0
0.2
0.4
0.6
0.8
1
2
4
6
8
10
12
14
16
18
20


Unsuccessful search
Linear probing
Double hashing
Separate chaining
Average # of probes
Load factor
20
18
16
14
12
10
8
6
4
2
0
0.2
0.4
0.6
0.8
1

Open Addressing vs. Separate Chaining

- When should you be concerned about Open Addressing and Separate Chaining implementations?
- Note that there are Hash libraries… Java supports Hashtable, HashMap, LinkedHashMap, HashSet,…
- But, if you are implementing your own hash table consider:
  - Do you know the total number of items to be inserted into the table?
  - Do you have plenty of memory?
  - Do you know the expected load factor?