

Python Programming

Chapter 6 : Simple Algorithms and Data Structures

Introduction

- Successful professional computer scientists might invent maybe one algorithm during their whole career—if they are lucky.
- Most of us never invent a novel algorithm. What we do instead is learn to reduce the most complex aspects of the problems with which we are faced to previously solved problems.
- More specifically, we
 - Develop an understanding of the inherent complexity of the problem with which we are faced,
 - Think about how to break that problem up into sub problems, and
 - Relate those sub problems to other problems for which efficient algorithms already exist.

Search Algorithms

- A **search algorithm** is a method for finding an item or group of items with specific properties within a collection of items.
- We refer to the collection of items as a **search space**.
- The search space might be something concrete, such as a set of electronic medical records, or something abstract, such as the set of all integers.

```
def search(L, e):  
    """Assumes L is a list.  
    Returns True if e is in L and False otherwise"""
```

Linear Search and Using Indirection to Access Elements

```
def search(L, e):  
    for i in range(len(L)):  
        if L[i] == e:  
            return True  
    return False
```

- If the element e is not in the list, the algorithm will perform $O(\text{len}(L))$ tests
- i.e. the complexity is at best linear in length of L .

Linear Search and Using Indirection to Access Elements...

- Element of the list is an integer.
- This implies that each element of the list is the same size, e.g., four units of memory (four eight-bit bytes).
- In this case the address in memory of the i th element of the list is simply $\text{start} + 4i$, where start is the address of the start of the list.
- Therefore we can assume that Python could compute the address of the i th element of a list of integers in constant time.

Linear Search and Using Indirection to Access Elements...

- In Python, a list is represented as a length (the number of objects in the list) and a sequence of fixed-size pointers to objects.
- The shaded region represents a list containing four elements.
- The leftmost shaded box contains a pointer to an integer indicating the length of the list.
- Each of the other shaded boxes contains a pointer to an object in the list.

Linear Search and Using Indirection to Access Elements...

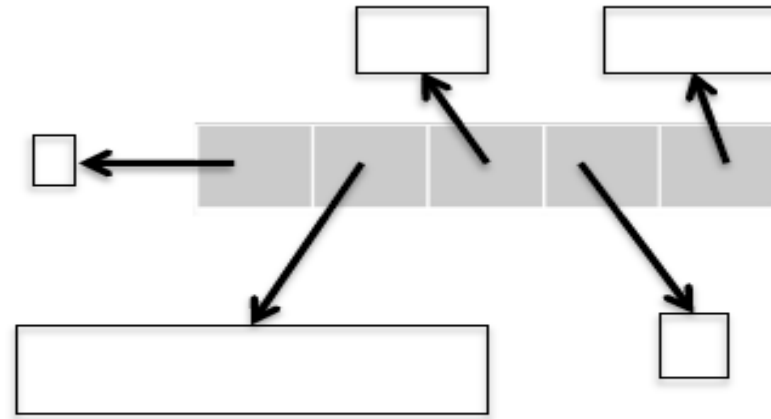


Figure 10.1 Implementing lists

- If the length field is four units of memory, and each pointer (address) occupies four units of memory, the address of the i th element of the list is stored at the address $\text{start} + 4 + 4i$.
- Again, this address can be found in constant time, and then the value stored at that address can be used to access the i th element.
- This access too is a constant-time operation.

Linear Search and Using Indirection to Access Elements...

- This example illustrates one of the most important implementation techniques
- used in computing: **indirection**.
- Generally speaking, indirection involves accessing something by first accessing something else that contains a reference to the thing initially sought.
- This is what happens each time we use a variable to refer to the object to which that variable is bound.
- When we use a variable to access a list and then a reference stored in that list to access another object, we are going through two levels of indirection

Binary Search and Exploiting Assumptions

- Search(L,e) is $O(\text{len}(L))$
- Yes, if we don't know about the relationship of the values of the elements or the order
- If all integers stored in ascending order

```
def search(L, e):  
    """Assumes L is a list, the elements of which are in  
        ascending order.  
        Returns True if e is in L and False otherwise"""  
    for i in range(len(L)):  
        if L[i] == e:  
            return True  
        if L[i] > e:  
            return False  
    return False
```

Binary Search and Exploiting Assumptions.

- We can get a considerable improvements in the worst-case complexity using – Binary Search

The idea is simple:

1. Pick an index, i , that divides the list L roughly in half.
2. Ask if $L[i] == e$.
3. If not, ask whether $L[i]$ is larger or smaller than e .
4. Depending upon the answer, search either the left or right half of L for e .

Binary Search and Exploiting Assumptions.

```
def search(L, e):
    """Assumes L is a list, the elements of which are in
       ascending order.
       Returns True if e is in L and False otherwise"""

def bSearch(L, e, low, high):
    #Decrements high - low
    if high == low:
        return L[low] == e
    mid = (low + high)//2
    if L[mid] == e:
        return True
    elif L[mid] > e:
        if low == mid: #nothing left to search
            return False
        else:
            return bSearch(L, e, low, mid - 1)
    else:
        return bSearch(L, e, mid + 1, high)

if len(L) == 0:
    return False
else:
    return bSearch(L, e, 0, len(L) - 1)
```

Binary Search and Exploiting Assumptions.

- Assumption – ascending order
- Checking assumption would take $O(\text{len}(L))$ time
- Functions such as search are often called wrapper functions.
- Why low – high??
- User don't have to do anything with them.
- The complexity $O(\log(\text{len}(L)))$

Sorting Algorithms

- Sorted \rightarrow searching
- Does it mean every time first sort then search?
- Sorting $\rightarrow O(\text{sortComplexity}(L))$
- Search $\rightarrow O(\text{len}(L))$
- Is $\text{sortComplexity}(L) + \log(\text{len}(L)) < \text{len}(L)$
- Answer is ???
- One can not sort a list without looking at each element in the list at least once.

Sorting Algorithms...

- If we expect to search the list k times, the relevant question becomes, is $(\text{sortComplexity}(L) + k * \log(\text{len}(L)))$ less than $k * \text{len}(L)$?
- As k becomes large, the time required to sort the list becomes increasingly irrelevant.
- How big k needs to be depends upon how long it takes to sort a list.
- If, for example, sorting were exponential in the size of the list, k would have to be quite large.

Sorting Algorithms...

- Fortunately, sorting can be done rather efficiently.
- For example, the standard implementation of sorting in most Python implementations runs in roughly $O(n \cdot \log(n))$ time, where n is the length of the list.
- use either Python's built-in sort method (`L.sort()` sorts the list `L`) or
- its built-in function `sorted` (`sorted(L)` returns a list with same elements as `L`, but does not mutate `L`)

Selection Sort

```
def selSort(L):  
    """Assumes that L is a list of elements that can be  
        compared using >.  
        Sorts L in ascending order"""  
    suffixStart = 0  
    while suffixStart != len(L):  
        #look at each element in suffix  
        for i in range(suffixStart, len(L)):  
            if L[i] < L[suffixStart]:  
                #swap position of elements  
                L[suffixStart], L[i] = L[i], L[suffixStart]  
        suffixStart += 1
```


Selection Sort...

- Partition of list into prefix($L[0:i]$) and a suffix ($L[i+1:\text{len}(L)]$)
- Base case: At the start of the first iteration, the prefix is empty, i.e., the suffix is the entire list. The invariant is (trivially) true.
- Induction step:
 - we move one element from the suffix to the prefix
 - appending a minimum element of the suffix to the end of the prefix
- When the loop is exited, the prefix includes the entire list, and the suffix is empty. Therefore, the entire list is now sorted in ascending order.

Selection Sort...

- The complexity of the inner loop is $O(\text{len}(L))$.
- The complexity of the outer loop is also $O(\text{len}(L))$.
- So, the complexity of the entire function is $O(\text{len}(L)^2)$. I.e., it is quadratic in the length of L .

Merge Sort

- Fortunately, we can do a lot better than quadratic time using a **divide-and conquer algorithm**.
- In general, a divide-and-conquer algorithm is characterized by
 - A threshold input size, below which the problem is not subdivided,
 - The size and number of sub-instances into which an instance is split, and
 - The algorithm used to combine sub-solutions.

Merge Sort...

- **Merge sort** is a prototypical divide-and-conquer algorithm.
- It was invented in 1945, by John von Neumann, and is still widely used.
 - If the list is of length 0 or 1, it is already sorted.
 - If the list has more than one element, split the list into two lists, and use merge sort to sort each of them.
 - Merge the results.

Merge Sort...

- The key observation made by von Neumann is that two sorted lists can be efficiently merged into a single sorted list.
- The idea is to look at the first element of each list, and move the smaller of the two to the end of the result list.
- When one of the lists is empty, all that remains is to copy the remaining items from the other list.

Merge Sort...

- Consider, for example, merging the two lists [1,5,12,18,19,20] and [2,3,4,17]:

<u>Left in list 1</u>	<u>Left in list 2</u>	<u>Result</u>
[1, 5, 12, 18, 19, 20]	[2, 3, 4, 17]	[]
[5, 12, 18, 19, 20]	[2, 3, 4, 17]	[1]
[5, 12, 18, 19, 20]	[3, 4, 17]	[1, 2]
[5, 12, 18, 19, 20]	[4, 17]	[1, 2, 3]
[5, 12, 18, 19, 20]	[17]	[1, 2, 3, 4]
[12, 18, 19, 20]	[17]	[1, 2, 3, 4, 5]
[18, 19, 20]	[17]	[1, 2, 3, 4, 5, 12]
[18, 19, 20]	[]	[1, 2, 3, 4, 5, 12, 17]
[]	[]	[1, 2, 3, 4, 5, 12, 17, 18, 19, 20]

Merge Sort...

- Complexity??
- Comparing the values and copying elements
- Comparison $\rightarrow O(\text{len}(L))$
- Copy $\rightarrow O(\text{len}(L1) + \text{len}(L2))$

```
def merge(left, right, compare):  
    """Assumes left and right are sorted lists and  
        compare defines an ordering on the elements.  
    Returns a new sorted (by compare) list containing the  
        same elements as (left + right) would contain."""
```

```
    result = []  
    i, j = 0, 0  
    while i < len(left) and j < len(right):  
        if compare(left[i], right[j]):  
            result.append(left[i])  
            i += 1  
        else:  
            result.append(right[j])  
            j += 1  
    while (i < len(left)):  
        result.append(left[i])  
        i += 1  
    while (j < len(right)):  
        result.append(right[j])  
        j += 1  
    return result
```

```
import operator
```

```
def mergeSort(L, compare = operator.lt):  
    """Assumes L is a list, compare defines an ordering  
        on elements of L  
    Returns a new sorted list containing the same elements as L"""  
    if len(L) < 2:  
        return L[:]  
    else:  
        middle = len(L)//2  
        left = mergeSort(L[:middle], compare)  
        right = mergeSort(L[middle:], compare)  
        return merge(left, right, compare)
```


Merge Sort...

- At each level of recursion the total number of elements to be merged is $\text{len}(L)$.
- Therefore, the time complexity of mergeSort is $O(\text{len}(L))$ multiplied by the number of levels of recursion.
- Since mergeSort divides the list in half each time, we know that the number of levels of recursion is $O(\log(\text{len}(L)))$.
- Therefore, the time complexity of mergeSort is $O(n \cdot \log(n))$, where n is $\text{len}(L)$.

Merge Sort...

- This is a lot better than selection sort's $O(\text{len}(L)^2)$.
- For example, if L has 10,000 elements, $\text{len}(L)^2$ is a hundred million but $\text{len}(L) * \log_2(\text{len}(L))$ is about 130,000.
- This improvement in time complexity comes with a price.
- Selection sort is an example of an **in-place** sorting algorithm. Because it works by swapping the place of elements within the list, it uses only a constant amount of extra storage (one element in our implementation).

Merge Sort...

- In contrast, the merge sort algorithm involves making copies of the list.
- This means that its space complexity is $O(\text{len}(L))$.
- This can be an issue for large lists.

Exploiting Functions as Parameters

```
def lastNameFirstName(name1, name2):  
    import string  
    name1 = string.split(name1, ' ')  
    name2 = string.split(name2, ' ')  
    if name1[1] != name2[1]:  
        return name1[1] < name2[1]  
    else: #last names the same, sort by first name  
        return name1[0] < name2[0]
```

```
def firstNameLastName(name1, name2):  
    import string  
    name1 = string.split(name1, ' ')  
    name2 = string.split(name2, ' ')  
    if name1[0] != name2[0]:  
        return name1[0] < name2[0]  
    else: #first names the same, sort by last name  
        return name1[1] < name2[1]
```

```
L = ['Chris Terman', 'Tom Brady', 'Eric Grimson', 'Gisele Bundchen']  
newL = mergeSort(L, lastNameFirstName)  
print 'Sorted by last name =', newL  
newL = mergeSort(L, firstNameLastName)  
print 'Sorted by first name =', newL
```

Exploiting Functions as Parameters...

- The list is list ['Chris Terman', 'Tom Brady', 'Eric Grimson', 'Gisele Bundchen'].
- When the code on previous slide run, it prints
- Sorted by last name – ['Tom Brady', 'Gisele Bundchen', 'Eric Grimson', 'Chris Terman']
- Sorted by first name – ['Chris Terman', 'Eric Grimson', 'Gisele Bundchen', 'Tom Brady']

Sorting in Python

- The sorting algorithm used in most Python implementations is called **timsort**.
- The key idea is to take advantage of the fact that in a lot of data sets the data is already partially sorted.
- Timsort's worst-case performance is the same as merge sort's, but on average it performs considerably better.

Sorting in Python...

```
L = [3,5,2]
D = {'a':12, 'c':5, 'b':'dog'}
print sorted(L)
print L
L.sort()
print L
print sorted(D)
D.sort()
```

Sorting in Python...

```
[2, 3, 5]
```

```
[3, 5, 2]
```

```
[2, 3, 5]
```

```
['a', 'b', 'c']
```

```
AttributeError: 'dict' object has no attribute 'sort'
```


Sorting in Python...

```
L = [[1,2,3], (3,2,1,0), 'abc']  
print sorted(L, key = len, reverse = True)
```

sorts the elements of L in reverse order of length and prints

```
[(3, 2, 1, 0), [1, 2, 3], 'abc']
```

Hash Tables

- We use merge sort to preprocess the list in $O(n \cdot \log(n))$ time, and then we use binary search to test whether elements are in the list in $O(\log(n))$ time.
- If we search the list k times, the overall time complexity is $O(n \cdot \log(n) + k \cdot \log(n))$.

Hash Tables...

- The basic idea behind a **hash table** is simple.
- We convert the key to an integer, and then use that integer to index into a list, which can be done in constant time.
- In principle, values of any immutable type can be easily converted to an integer.
- For example, the internal representation of 'abc' is the string of bits 011000010110001001100011, which can be viewed as a representation of the decimal integer 6,382,179.

Hash Tables...

- A **hash function** maps a large space of inputs (e.g., all natural numbers) to a smaller space of outputs (e.g., the natural numbers between 0 and 5000).
- Hash functions can be used to convert a large space of keys to a smaller space of integer indices.
- Since the space of possible outputs is smaller than the space of possible inputs, a hash function is a **many-to-one mapping**, i.e., multiple different inputs may be mapped to the same output.

Hash Tables...

- When two inputs are mapped to the same output, it is called a **collision**
- A good hash function produces a **uniform distribution**, i.e., every output in the range is equally probable, which minimizes the probability of collisions.

Hash Tables...

- Next code uses a simple hash function (recall that $i \% j$ returns the remainder when the integer i is divided by the integer j) to implement a dictionary with integers as keys.
- The basic idea is to represent an instance of class `intDict` by a list of **hash buckets**, where each bucket is a list of key/value pairs.
- By making each bucket a list, we handle collisions by storing all of the values that hash to the same bucket in the list.

Hash Tables...

- The instance variable `buckets` is initialized to a list of `numBuckets` empty lists.
- To store or look up an entry with key `dictKey`, we use the hash function `%` to convert `dictKey` into an integer, and use that integer to index into `buckets` to find the hash bucket associated with `dictKey`.
- We then search that bucket (which is a list) linearly to see if there is an entry with the key `dictKey`.
- If we are doing a lookup and there is an entry with the key, we simply return the value stored with that key.
- If there is no entry with that key, we return `None`.
- If a value is to be stored, then we either replace the value in the existing entry, if one was found, or append a new entry to the bucket if none was found.

```

class intDict(object):
    """A dictionary with integer keys"""

    def __init__(self, numBuckets):
        """Create an empty dictionary"""
        self.buckets = []
        self.numBuckets = numBuckets
        for i in range(numBuckets):
            self.buckets.append([])

    def addEntry(self, dictKey, dictVal):
        """Assumes dictKey an int. Adds an entry."""
        hashBucket = self.buckets[dictKey%self.numBuckets]
        for i in range(len(hashBucket)):
            if hashBucket[i][0] == dictKey:
                hashBucket[i] = (dictKey, dictVal)
                return
        hashBucket.append((dictKey, dictVal))

    def getValue(self, dictKey):
        """Assumes dictKey an int. Returns entry associated
           with the key dictKey"""
        hashBucket = self.buckets[dictKey%self.numBuckets]
        for e in hashBucket:
            if e[0] == dictKey:
                return e[1]
        return None

    def __str__(self):
        result = '{ '
        for b in self.buckets:
            for e in b:
                result = result + str(e[0]) + ':' + str(e[1]) + ', '
        return result[:-1] + '}' #result[:-1] omits the last comma

```


Hash Tables...

- The following code first constructs an `intDict` with twenty entries.
- The values of the entries are the integers 0 to 19.
- The keys are chosen at random from integers in the range 0 to $10^5 - 1$.
- The code then goes on to print the `intDict` using the `__str__` method defined in the class.
- Finally it prints the individual hash buckets by iterating over `D.buckets`.

Hash Tables...

```
import random #a standard library module

D = intDict(29)
for i in range(20):
    #choose a random int between 0 and 10**5
    key = random.randint(0, 10**5)
    D.addEntry(key, i)
print 'The value of the intDict is:'
print D
print '\n', 'The buckets are:'
for hashBucket in D.buckets: #violates abstraction barrier
    print ' ', hashBucket
```

The value of the intDict is:

```
{93467:5,78736:19,90718:4,529:16,12130:1,7173:7,68075:10,15851:0,  
47027:14,45288:8,5819:17,83076:6,55236:13,19481:9,11854:12,29604:11,  
45902:15,14408:18,24965:3,89377:2}
```

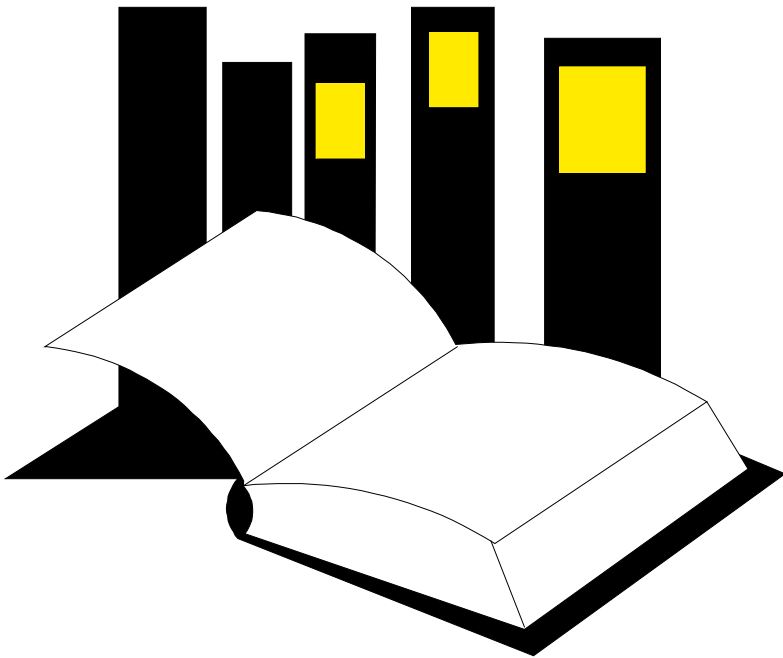
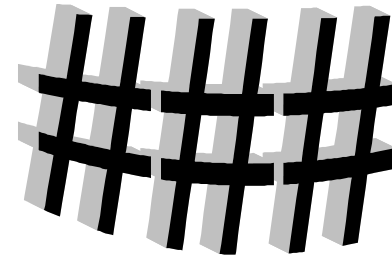
The buckets are:

```
[(93467, 5)]  
[(78736, 19)]  
[]  
[]  
[]  
[]  
[(90718, 4)]  
[(529, 16)]  
[(12130, 1)]  
[]  
[(7173, 7)]  
[]  
[(68075, 10)]  
[]  
[]  
[]  
[]  
[(15851, 0)]  
[(47027, 14)]  
[(45288, 8), (5819, 17)]  
[(83076, 6), (55236, 13)]  
[]  
[(19481, 9), (11854, 12)]  
[]  
[(29604, 11), (45902, 15), (14408, 18)]  
[(24965, 3)]  
[]  
[]  
[(89377, 2)]
```

Hash Tables...

- What is the complexity of `getValue`?
- If there were no collisions it would be $O(1)$, because each hash bucket would be of length 0 or 1.
- But, of course, there might be collisions.
- If everything hashed to the same bucket, it would be $O(n)$ where n is the number of entries in the dictionary, because the code would perform a linear search on that hash bucket.
- By making the hash table large enough, we can reduce the number of collisions sufficiently to allow us to treat the complexity as $O(1)$.

Hash Tables

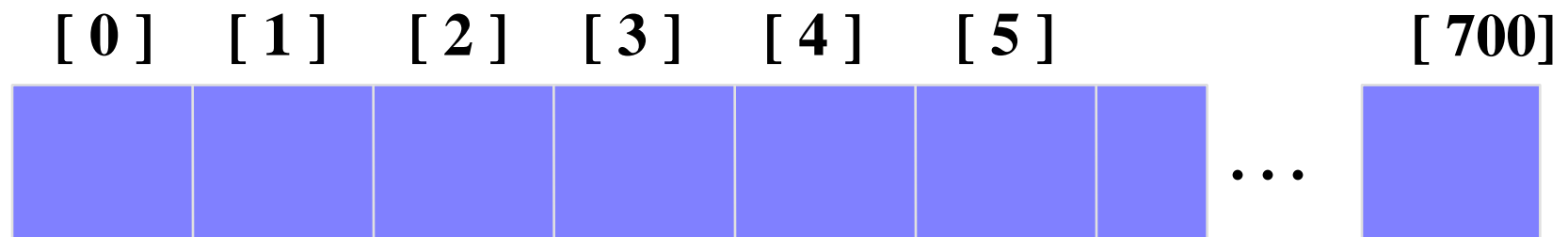


- Hash tables are a common approach to the storing/searching problem.
 - This presentation introduces hash tables.

**Data Structures
and Other Objects
Using C++**

What is a Hash Table ?

- The simplest kind of hash table is an array of records.
- This example has 701 records.

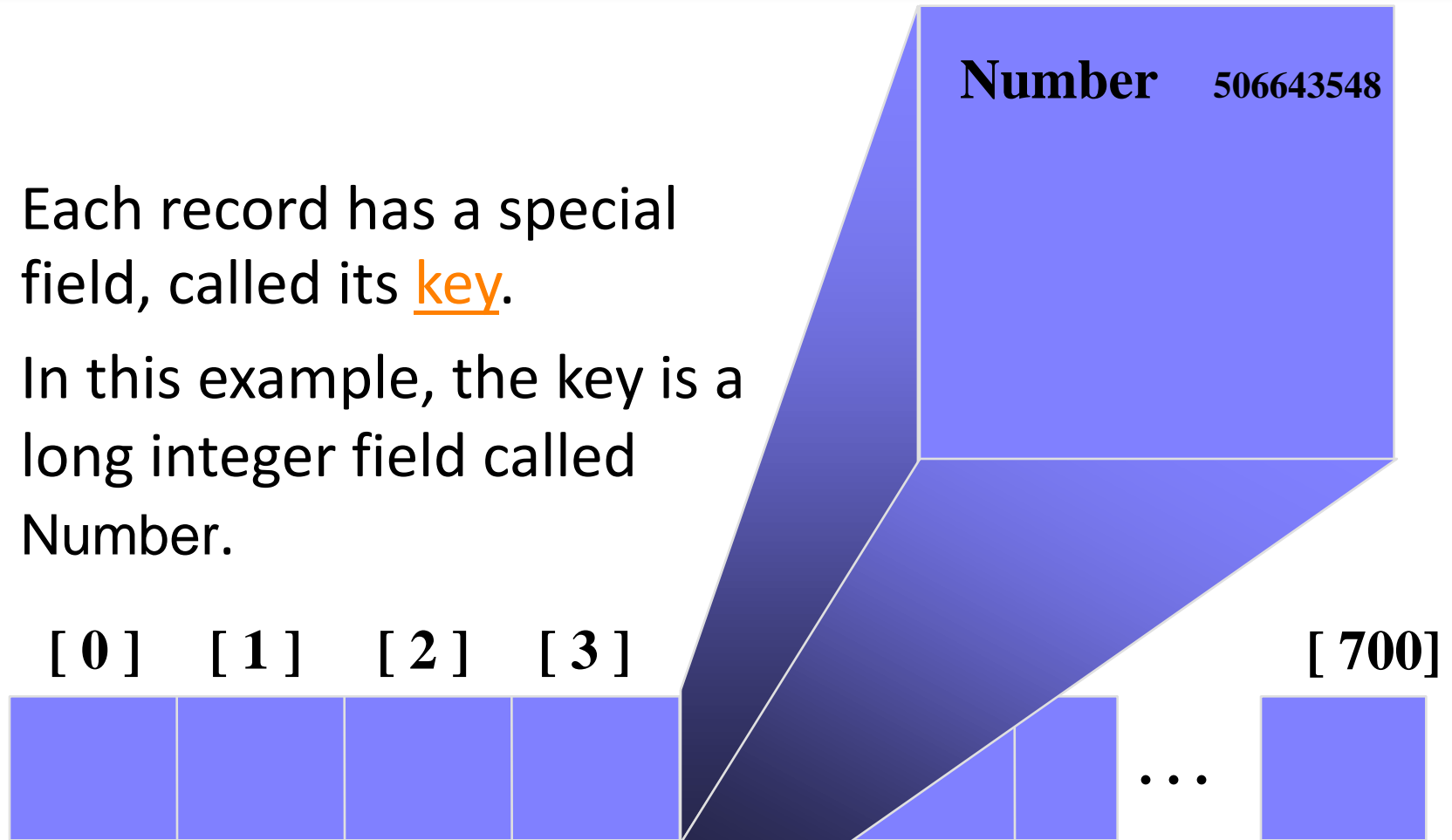


An array of records

What is a Hash Table ?

[4]

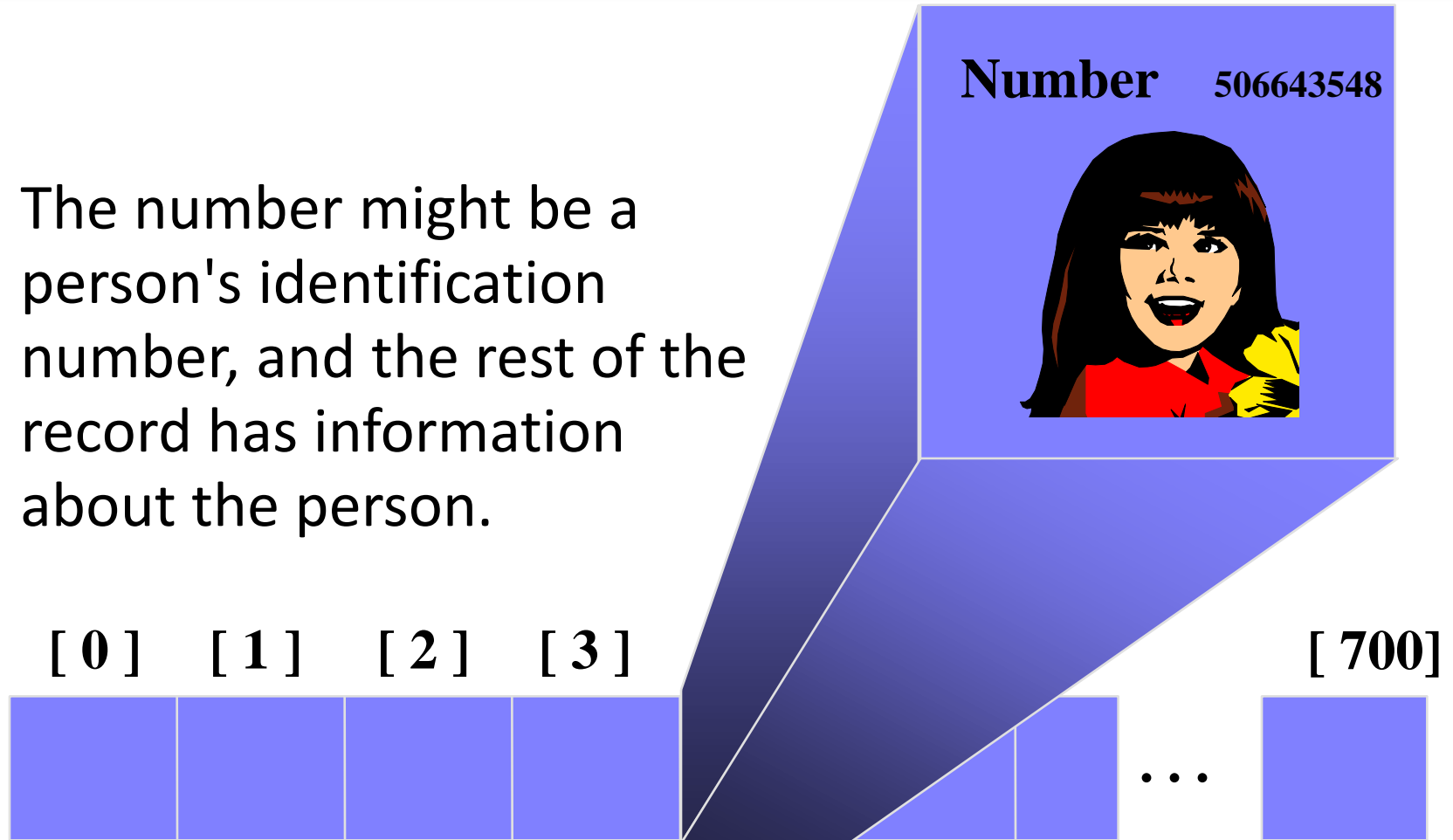
- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.



What is a Hash Table ?

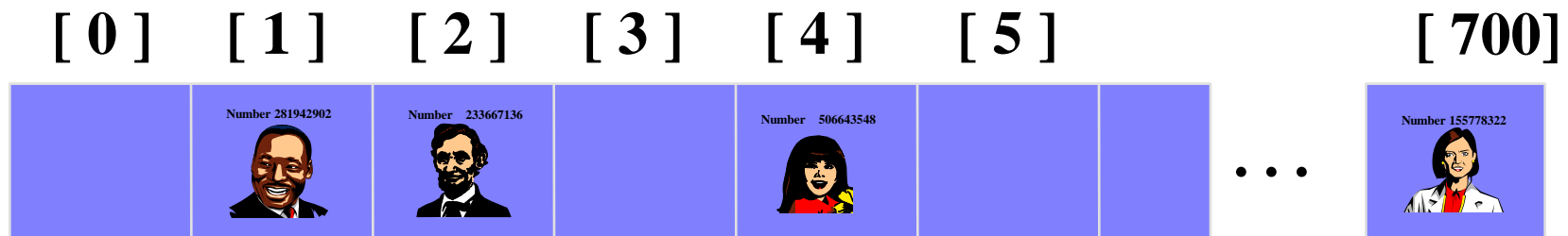
[4]

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



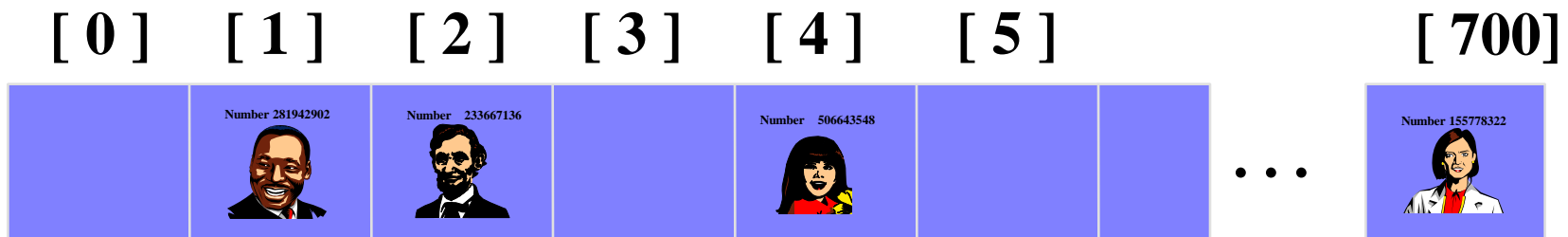
What is a Hash Table ?

- 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.

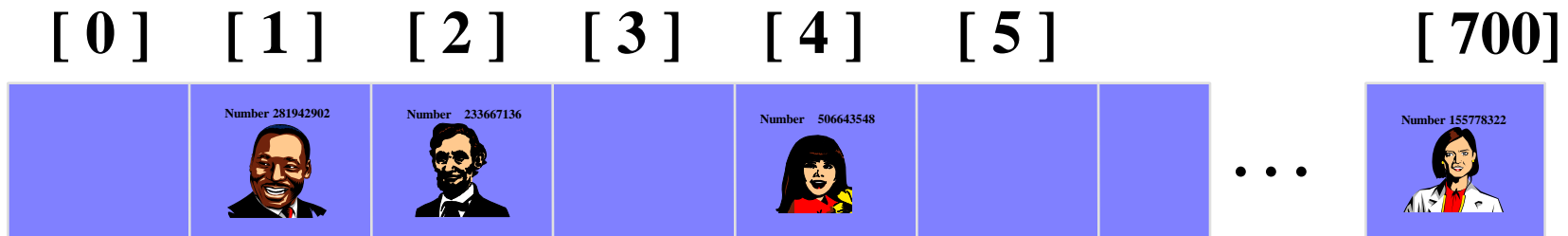


Inserting a New Record

- Typical way create a hash value:



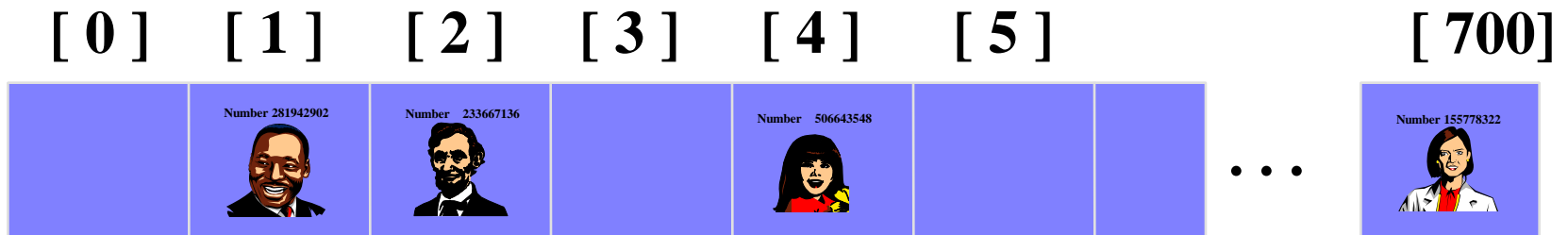
What is $(580625685 \bmod 701)$?



Inserting a New Record

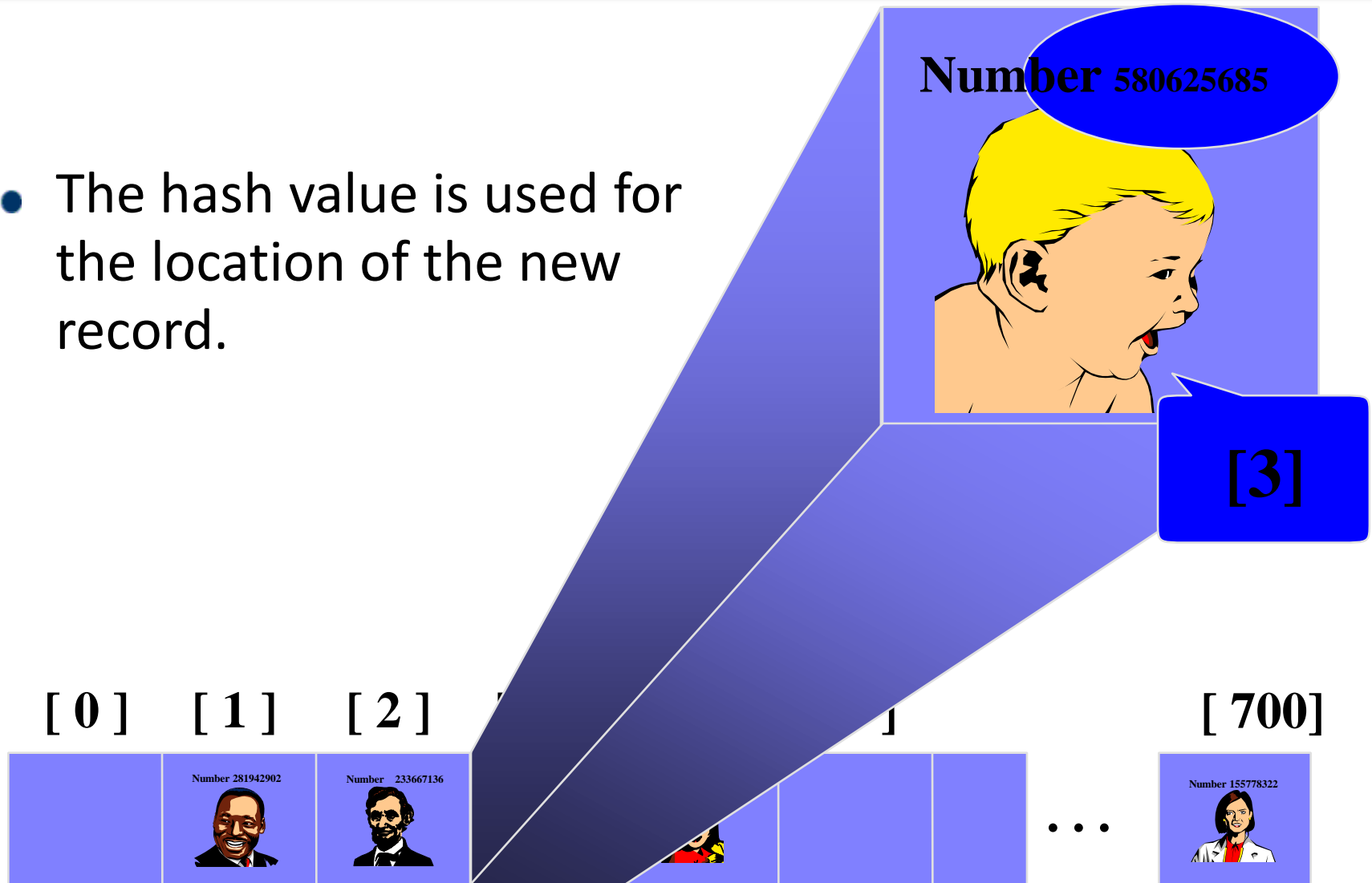
- Typical way to create a hash value:

What is $(580625685 \bmod 701)$?



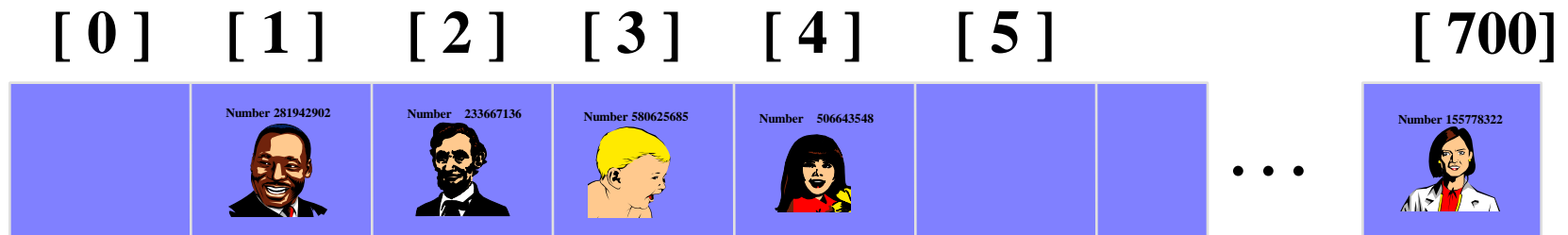
Inserting a New Record

- 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.

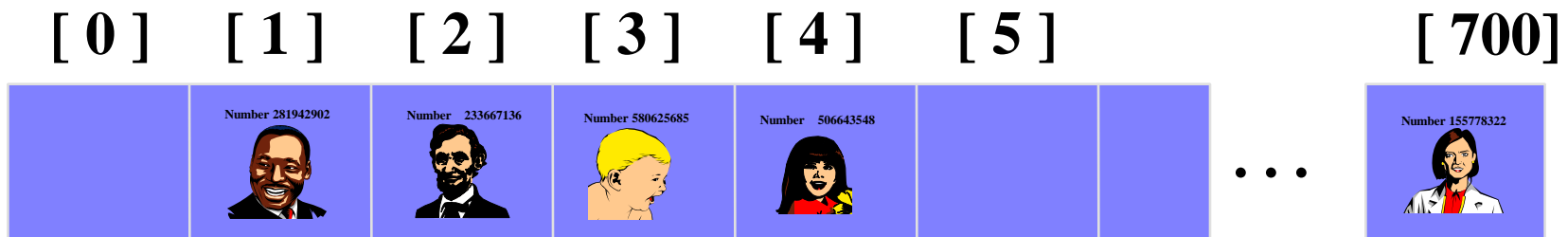


Collisions

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



My hash value is [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.

Number 701466868



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



Collisions

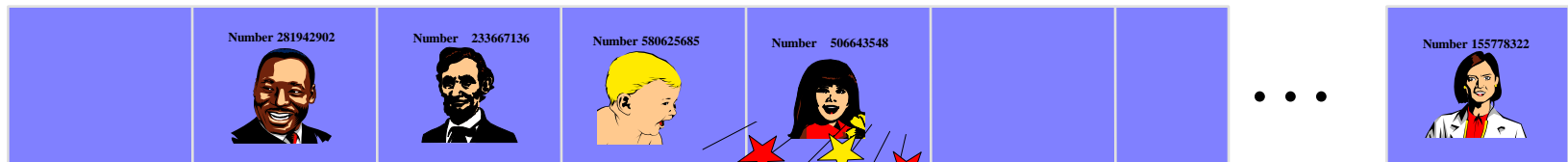
- This is called a **collision**, because there is already another valid record at [2].

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

Number 701466868



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



Collisions

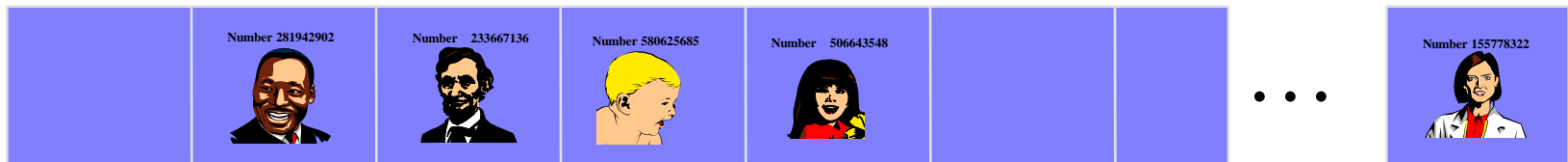
- This is called a **collision**, because there is already another valid record at [2].

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

Number 701466868



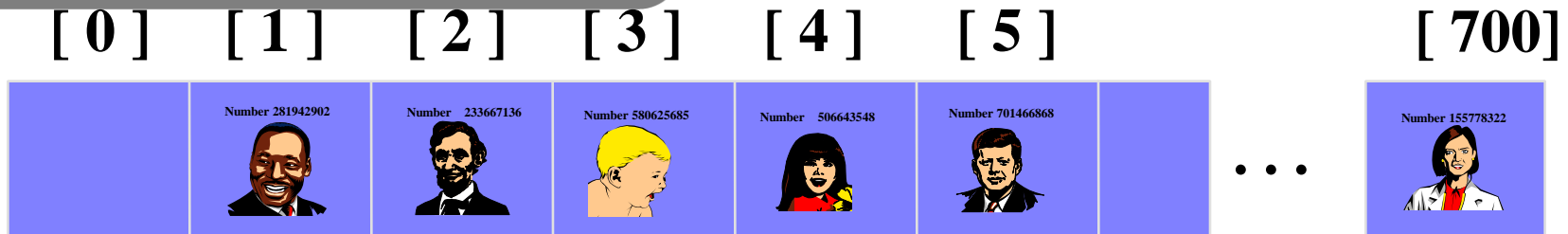
[0] [1] [2] [3] [4] [5] ... [700]



Collisions

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

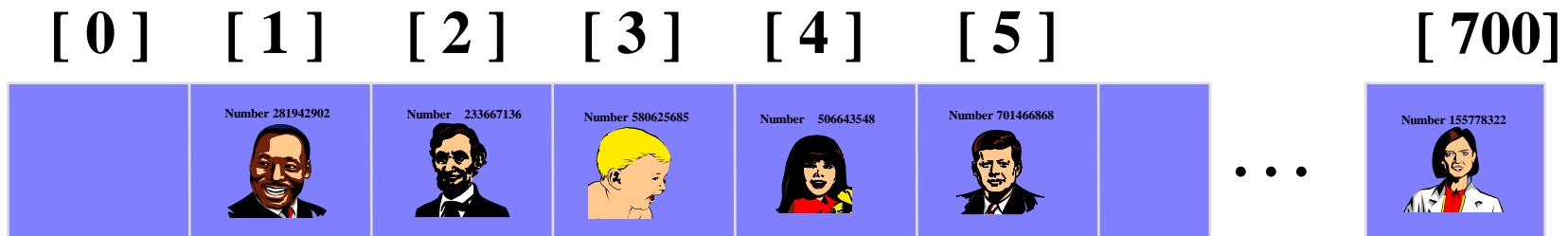
**The new record goes
in the empty spot.**



Searching for a Key

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

Number 701466868



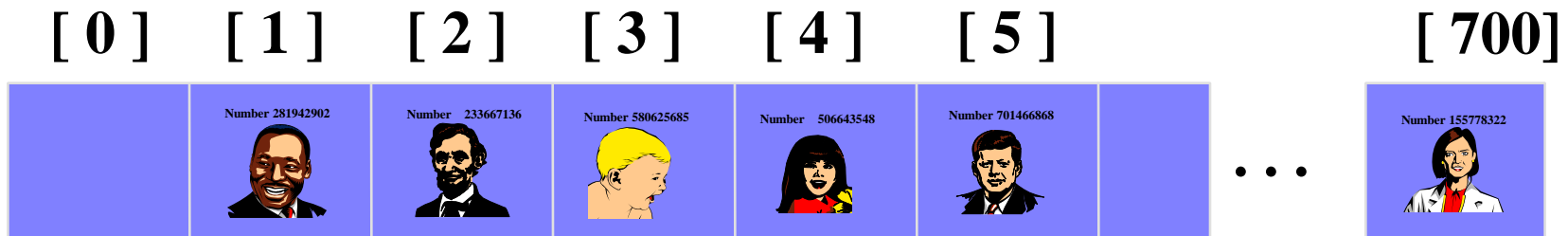
Searching for a Key

- Calculate the hash value.
- Check that location of the array for the key.

Number 701466868

My hash value is [2].

Not me.



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.

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.

Number 701466868

My hash value is [2].

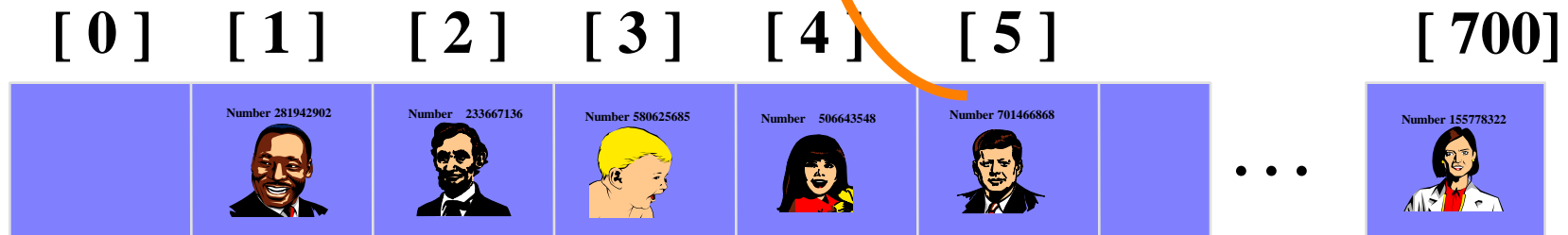
Yes!

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

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

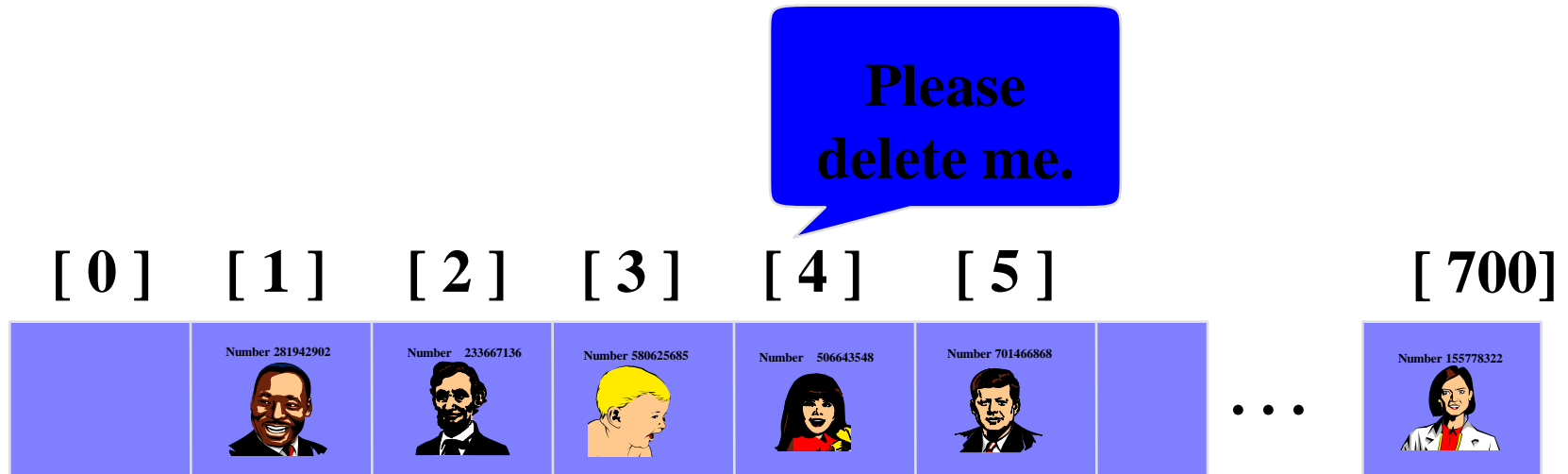
Searching for a Key

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



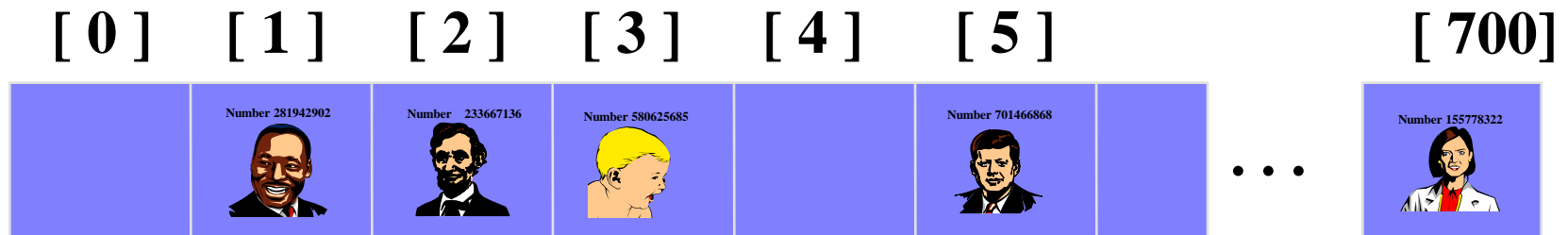
Deleting a Record

- Records may also be deleted from a hash table.



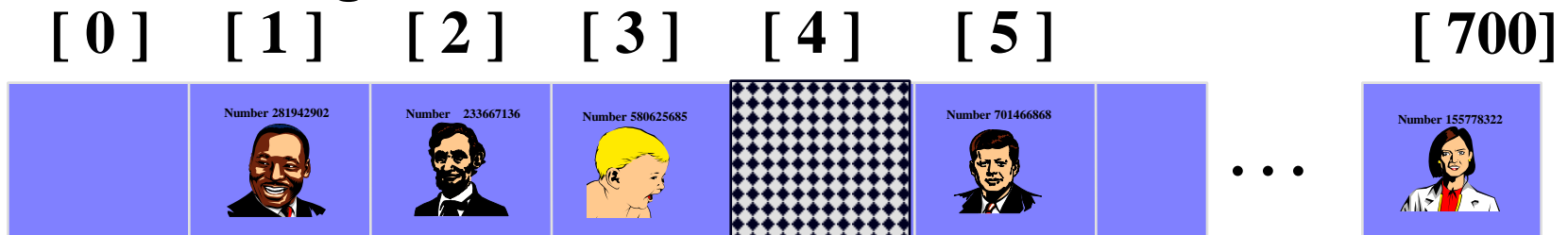
Deleting a Record

- 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.



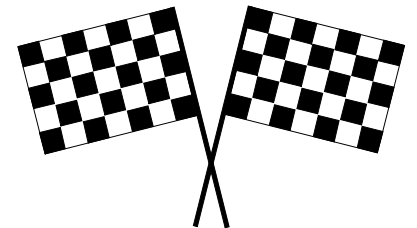
Summary

- ❑ 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.

Presentation copyright 2010 Addison Wesley Longman,
For use with *Data Structures and Other Objects Using C++*
by Michael Main and Walter Savitch.

Some artwork in the presentation is used with permission from Presentation Task Force (copyright New Vision Technologies Inc) and Corel Gallery Clipart Catalog (copyright Corel Corporation, 3G Graphics Inc, Archive Arts, Cartesia Software, Image Club Graphics Inc, One Mile Up Inc, TechPool Studios, Totem Graphics Inc).

Students and instructors who use *Data Structures and Other Objects Using C++* are welcome to use this presentation however they see fit, so long as this copyright notice remains intact.



Thank you!!!