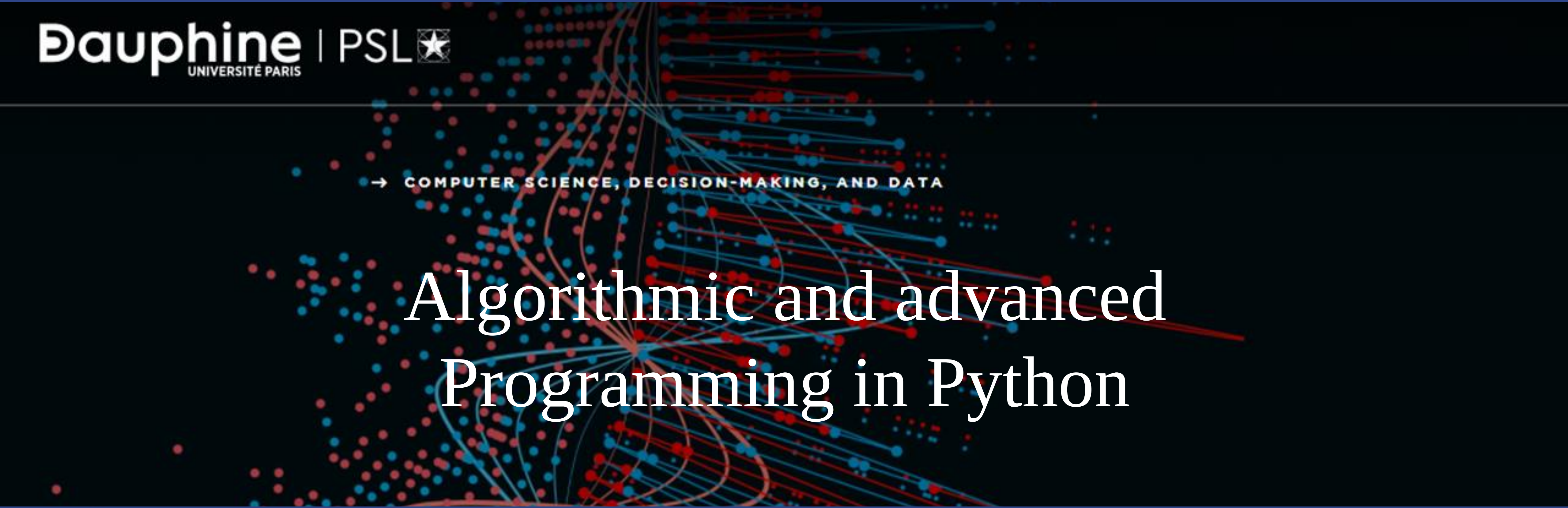


Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu

Masterclass 4

Outline

1. Hashing concept
2. Hashtable
3. Load factor and collisions

Reminder of the objective of this course

- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask

- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures

Reminder of previous session

- In Master class 3, we discuss about practical issue concerning Flask, data structure
- Question: can you summarize what Flask is and how it works?

What is hashing?

- Hashing is a technique used for storing and retrieving information as quickly as possible and is useful in implementing tables.
- Question: can you cite some mainstream application or usage of hashing?

- Why hashing?
- In the tree class, we saw that balanced binary search trees support operation like *insert*, *delete* and *search in O(logn)* time. Sometimes, if you need these operations in O(1), then hashing is a way. Remember that for hashing, worst case complexity is O(n) but in average O(1).

HashTable operations

- The common operations for hash table are
  - Create Hash Table
  - Hash Search
  - Hash Insert
  - Hash Delete
  - Delete Hash Table

Intuition of hashing - 1

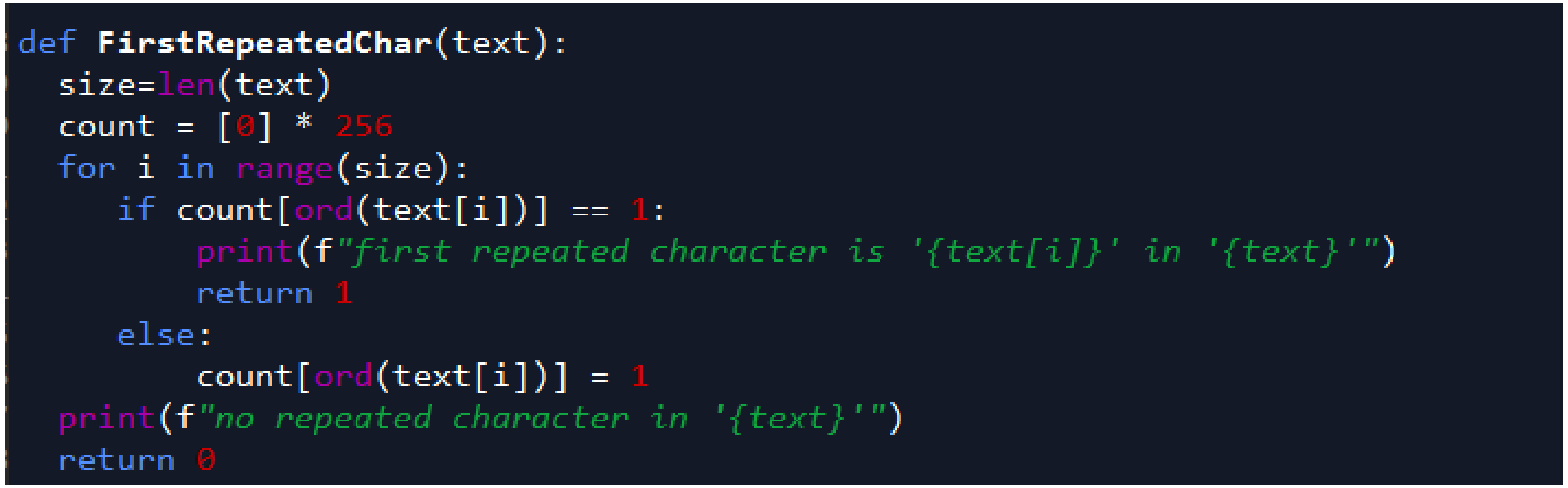
- In simple terms we can treat $array$ as a hash table. For understanding the use of hash tables, let us consider the following example: Give an algorithm for printing the first repeated character if there are duplicated elements in it. Let us think about the possible solutions.
- The simple and brute force way of solving is: given a string, for each character check whether that character is repeated or not. The time complexity of this approach is O(n^2)
- Question: can you think of a better solution?

Intuition of hashing - 2

- Now, let us find a better solution for this problem. Since our objective is to find the first repeated character, what if we remember the previous characters in some array?
- We know that the number of possible characters is 256 (for simplicity assume $ASCII$ characters only).
- Create an array of size 256 and initialize it with all zeros.
- For each of the input characters go to the corresponding position and increment its count.
- Since we are using arrays, it takes constant time for reaching any location.
- While scanning the input, if we get a character whose counter is already 1 then we can say that the character is the one which is repeating for the first time.

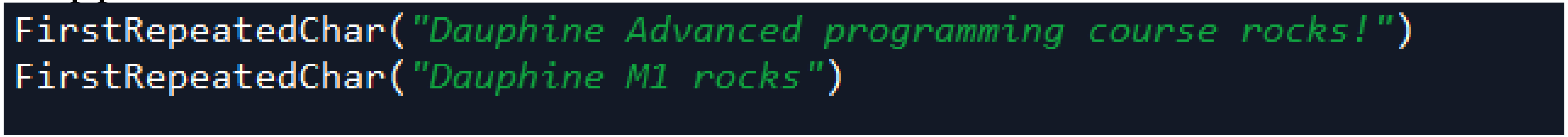
Intuition of hashing - 3

- In python terms:



```
def FirstRepeatedChar(text):
  size=len(text)
  count = [0] * 256
  for i in range(size):
     if count[ord(text[i])] == 1:
         print(f"first repeated character is '{text[i]}' in '{text}'")
         return 1
     else:
         count[ord(text[i])] = 1
  print(f"no repeated character in '{text}'")
  return 0
```

- Application:

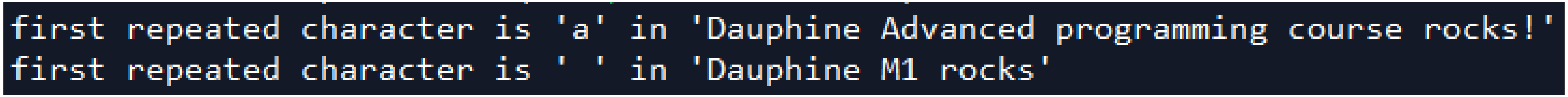


```
FirstRepeatedChar("Dauphine Advanced programming course rocks!")
FirstRepeatedChar("Dauphine M1 rocks")
```

Question: what would you get?

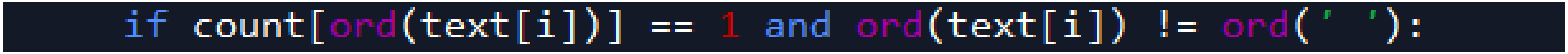
Result

- You would get:



```
first repeated character is 'a' in 'Dauphine Advanced programming course rocks!'
first repeated character is ' ' in 'Dauphine M1 rocks'
```

- Hence you should remove at least space as follows:



```
    if count[ord(text[i])] == 1 and ord(text[i]) != ord(' '):
```

Going further, array is not such a good idea?

- Question: any idea why?

Why not arrays?

- Arrays can be seen as a mapping, associating with every integer in a given interval some data item.
- It is finitary, because its domain, and therefore also its range, is finite. There are many situations when we want to index elements differently than just by integers.
- Common examples are strings (for dictionaries, phone books, menus, data base records), or structs (for dates, or names together with other identifying information).

Arrays work well for define and finite keys

- In many applications requiring associative arrays, we are storing complex data values and want to access them by a key which is derived from the data. A typical example of keys are strings, which are appropriate for many scenarios.

- For example, the key might be a student id and the data entry might be a collection of grades, perhaps another associative array where the key is the name of assignment or exam and the data is a score. We make the assumption that keys are unique in the sense that in an associative array there is at most one data item associated with a given key. In some applications we may need to complicate the structure of keys to achieve this uniqueness. This is consistent with ordinary arrays, which have a unique value for every valid index.

- But keys may not be that simple!

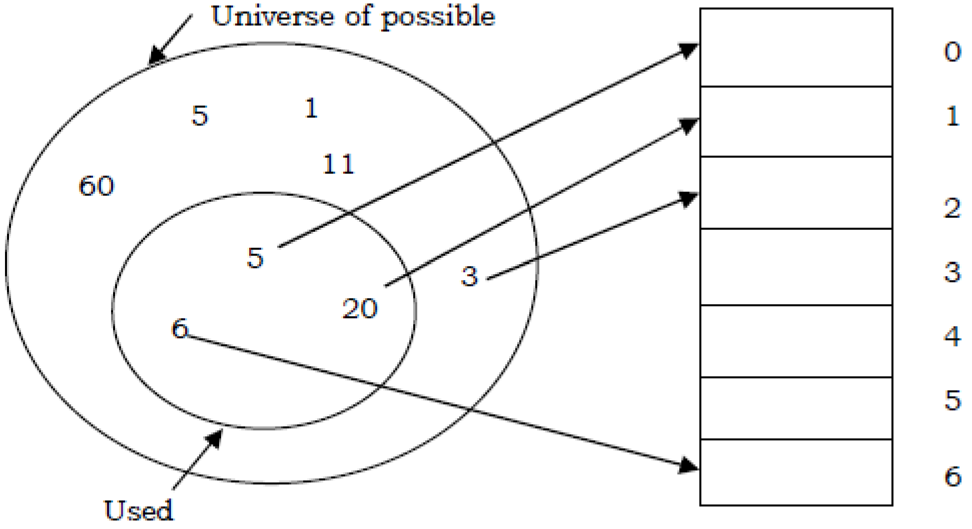
What is the universe is too large?

- In the previous problem, we have used an array of size 256 because we know the number of different possible characters [256] in advance.
- Now, let us consider a slight variant of the same problem. Suppose the given array has numbers instead of characters, then how do we solve the problem?
- In this case the set of possible values is infinity (or at least very big). Creating a huge array and storing the counters is not possible. That means there are a set of universal keys and limited locations in the main memory. To solve this problem we need to somehow map all these possible keys to the possible memory locations.

For numbers this is not feasilbe

- From the above discussion and diagram below it can be seen that we need a mapping of possible keys to one of the available locations. As a result, using simple arrays is not the correct choice for solving the problems where the possible keys are very big. The process of mapping the keys to available main memory locations is called

hashing.



Components of Hashing

- Hashing has four key components:
    - Hash Table
    - Hash Functions
    - Collisions
    - Collision Resolution Techniques

Hash Table

- Hash table is a generalization of array. With an array, we store the element whose key is k at a position k of the array. That means, given a key k, we find the element whose key is k by just looking in the k^{th} position of the array. This is called $direct\ addressing$.

- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key. But if we do not have enough space to allocate a location for each possible key, then we need a mechanism to handle this case. Another way of defining the scenario is: if we have less locations and more possible keys, then simple array implementation is not enough.

Hash function

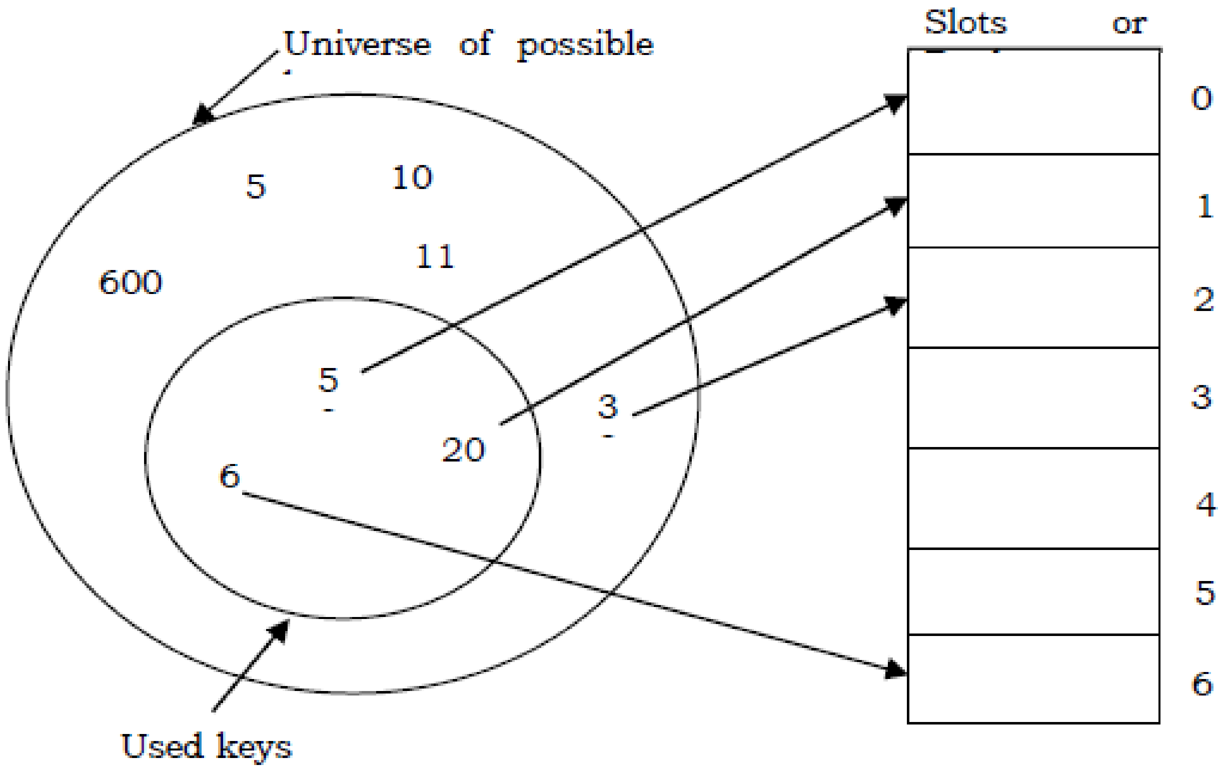
- In these cases one option is to use hash tables. Hash table or hash map is a data structure that stores the keys and their associated values, and hash table uses a hash function to map keys to their associated values. The general convention is that we use a hash table when the number of keys actually stored is small relative to the number of possible keys.

- A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a $slot$ (or a $bucket$), can hold an item and is named by an integer value starting at 0.

Slots

- For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special NULL.

Hashing


Universe of possible
5
10
11
600
5
20
3
6
Used keys
Slots or
0
1
2
3
4
5
6

Hash function

- The first idea behind hash tables is to exploit the efficiency of arrays. So: to map a key to an entry, we first map a key to an integer and then use the integer to index an array A. The first map is called a $hash\ function$. The hash function is used to transform the key into the slot index (or bucket index). Ideally, the hash function should map each possible key to a unique slot index, but it is difficult to achieve in practice.
- Given a collection of elements, a hash function that maps each item into a unique slot is referred to as a $perfect\ hash\ function$. If we know the elements and the collection will never change, then it is possible to construct a perfect hash function. Unfortunately, given an arbitrary collection of elements, there is no systematic way to construct a perfect hash function. Luckily, we do not need the hash function to be perfect to still gain performance efficiency.

Hash function

- One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the element range can be accommodated. This guarantees that each element will have a unique slot. Although this is practical for small numbers of elements, it is not feasible when the number of possible elements is large. For example, if the elements were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.
- Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the elements in the hash table. There are a number of common ways to extend the simple remainder method. We will consider a few of them here.

Folding method

- The $folding\ method$ for constructing hash functions begins by dividing the elements into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our element was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219 % 11=10.

How to Choose Hash Function?

- The basic problems associated with the creation of hash tables are:
  - An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
  - An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a location previously inserted in the hash table.
  - We must choose a hash function which can be calculated quickly, returns values within the range of locations in our table, and minimizes collisions.

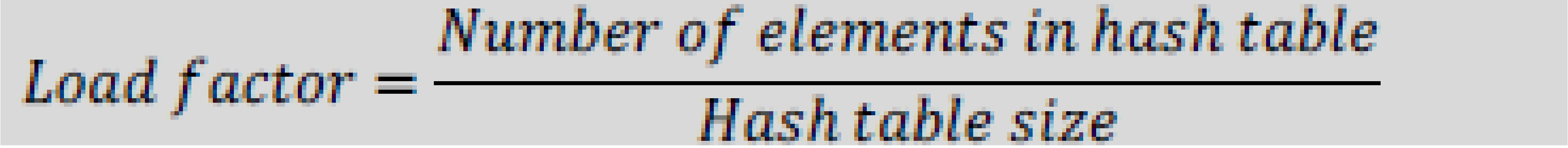
Question: according to you, what would be a good hash function?

Characteristics of Good Hash Functions

- A good hash function should have the following characteristics:
  - Minimize collisions
  - Be easy and quick to compute
  - Distribute key values evenly in the hash table
  - Use all the information provided in the key
  - Have a high load factor for a given set of keys

Load factor

- The load factor of a non-empty hash table is the number of items stored in the table divided by the size of the table. This is the decision parameter used when we want to rehash *or* expand the existing hash table entries. This also helps us in determining the efficiency of the hashing function. That means, it tells whether the hash function is distributing the keys uniformly or not.



$$Load\ factor = \frac{Number\ of\ elements\ in\ hash\ table}{Hash\ table\ size}$$

Collisions and resolutions

- Hash functions are used to map each key to a different address space, but practically it is not possible to create such a hash function and the problem is called *collision*. Collision is the condition where two keys are hashed to the same slot.

- Fortunately, there are effective techniques for resolving the conflict created by collisions. The process of finding an alternate location for a key in the case of a collision is called *collision resolution*. Even though hash tables have collision problems, they are more efficient in many cases compared to all other data structures, like search trees. There are a number of collision resolution techniques, and the most popular are direct chaining and open addressing.

Direct chaining and open adressing

- Direct Chaining (or Closed Addressing): An array of linked list application:
  - Separate chaining (linear chaining)

- Open Addressing: Array-based implementation:
  - Linear probing (linear search)
  - Quadratic probing (nonlinear search)
  - Double hashing (use multiple hash functions)

- Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function.

Separate chaining

- A first idea to explore is to implement the associative array as a linked list, called a chain or a linked list. Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists.
- Collision resolution by chaining combines linked representation with hash table. When two or more elements hash to the same location, these elements are constituted into a singly-linked list called a *chain*.
- In chaining, we put all the elements that hash to the same slot in a linked list. If we have a key k and look for it in the linked list, we just traverse it, compute the intrinsic key for each data entry, and compare it with k.
- If they are equal, we have found our entry, if not we continue the search.
- If we reach the end of the chain and do not find an entry with key k, then no entry with the given key exists.

Separate chaining example

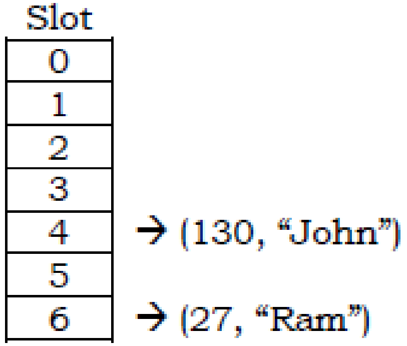
- In separate chaining, each slot of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

- As an example, consider the following simple hash function:

$$(key) = key \ \% \ table\ size$$

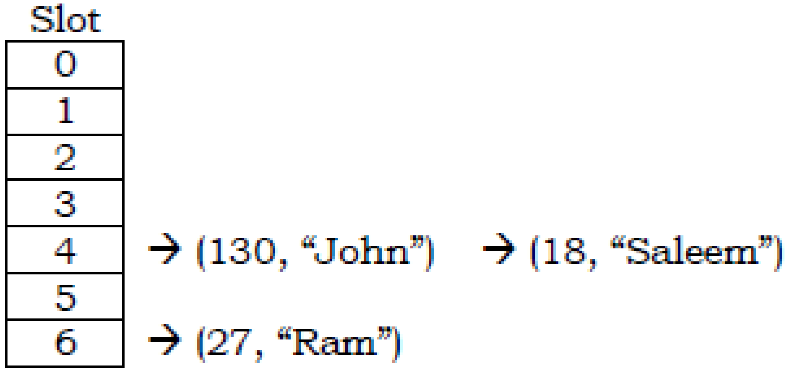
Algo in details

1. In a hash table with size 7, keys 27 and 130 would get 6 and 4 as hash indices respectively (as 27%7 = 6 and 130%7=4).



Slot
0
1
2
3
4 → (130, "John")
5
6 → (27, "Ram")

2. If we insert a new element (18, "Saleem"), that would also go to the fourth index as 18%7 is 4.



Slot
0
1
2
3
4 → (130, "John") → (18, "Saleem")
5
6 → (27, "Ram")

Hashing cost

- The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list.
- For this reason, chained hash tables remain effective even when the number of table entries (n) is much higher than the number of slots.

- For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list.
- The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (n) of entries in the table.

Worst case behavior

- The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus)(n) plus the time to compute the hash function--no better than if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance.

Open addressing

- In open addressing all keys are stored in the hash table itself. This approach is also known as $closed\ hashing$. This procedure is based on probing. A collision is resolved by probing.

Linear Probing

- The interval between probes is fixed at 1. In linear probing, we search the hash table sequentially. starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

$$rehash(key) = (\text{old\_hash} + 1)\%\ table\ size$$

Clustering

- One of the problems with linear probing is that table items tend to cluster together in the hash table. This means that the table contains groups of consecutively occupied locations that are called $clustering$.
- Clusters can get close to one another, and merge into a larger cluster. Thus, the one part of the table might be quite dense, even though another part has relatively few items. Clustering causes long probe searches and therefore decreases the overall efficiency.
- The next location to be probed is determined by the step-size, where other step-sizes (more than one) are possible. The step-size should be relatively prime to the table size, i.e. their greatest common divisor should be equal to 1. If we choose the table size to be a prime number, then any step-size is relatively prime to the table size. Clustering cannot be avoided by larger step-sizes.

Quadratic probing

- The interval between probes increases proportionally to the hash value (the interval thus increasing linearly, and the indices are described by a quadratic function). The problem of clustering can be eliminated if we use the quadratic probing method. Quadratic probing is also referred to as $mid - square$ method.

- In quadratic probing, we start from the original hash location i. If a location is occupied, we check the locations $i + 1^2$, $i + 2^2$, $i + 3^2$, $i +$ 4^2... We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

$$rehash(key) = (\text{old\_hash} + k^2)\ \%\ table\ size$$

Example

- *Example*: Let us assume that the table size is 11 (0. .10)

Hash Function: h(key) = key mod 11

$31 \; mod \; 11 = 9$

$19 \; mod \; 11 = 8$

$2 \; mod \; 11 = 2$

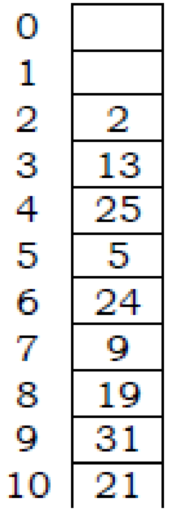
$13 \; mod \; 11 = 2 \rightarrow 2 + 1^2 = 3$

$25 \; mod \; 11 = 3 \rightarrow 3 + 1^2 = 4$

$24 \; mod \; 11 = 2 \rightarrow 2 + 1^2, 2 + 2^2 = 6$

$21 \; mod \; 11 = 10$

$9 \; mod \; 11 = 9 \rightarrow 9 + 1^2, 9 + 2^2 \; mod \; 11, 9 + 3^2 \; mod \; 11 = 7$



0	
1	
2	2
3	13
4	25
5	5
6	24
7	9
8	19
9	31
10	21

Question: could we do something better?

Inutition

- Even though clustering is avoided by quadratic probing, still there are chances of clustering. Clustering is caused by multiple search keys mapped to the same hash key. Thus, the probing sequence for such search keys is prolonged by repeated conflicts along the probing sequence. Both linear and quadratic probing use a probing sequence that is independent of the search key.

Double hashing

- The interval between probes is computed by another hash function. Double hashing reduces clustering in a better way. The increments for the probing sequence are computed by using a second hash function. The second hash function $h2$ should be:

$$h2(key) \neq 0 \text{ and } h2 \neq h1$$

- We first probe the location $h1(key)$. If the location is occupied, we probe the location $h1(key) + h2(key)$, $h1(key) + 2 * h2(key)$, ...

Example

- Table size is 11 (0. .10) Hash Function: assume $h1(key) = key\ mod\ 11$ and $h2(key) = 7 – (key\ mod\ 7)$

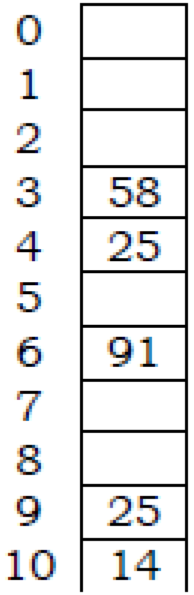
Insert keys:

$58\ mod\ 11 = 3$

$14\ mod\ 11 = 3 \rightarrow 3 + 7 = 10$

$91\ mod\ 11 = 3 \rightarrow 3 + 7, 3 + 2 * 7\ mod\ 11 = 6$

$25\ mod\ 11 = 3 \rightarrow 3 + 3, 3 + 2 * 3 = 9$



0	
1	
2	
3	58
4	25
5	
6	91
7	
8	
9	25
10	14

Comparison of Collision Resolution Techniques

- **Comparisons: Linear Probing vs. Double Hashing**

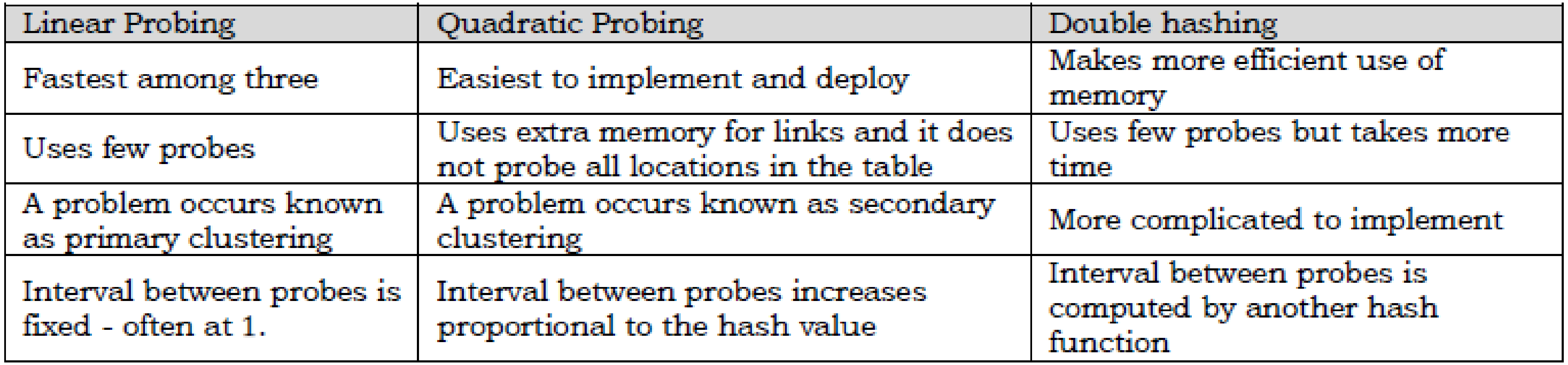
The choice between linear probing and double hashing depends on the cost of computing the hash function and on the load factor [number of elements per slot] of the table. Both use few probes but double hashing take more time because it hashes to compare two hash functions for long keys.

- **Comparisons: Open Addressing vs. Separate Chaining**

It is somewhat complicated because we have to account for the memory usage. Separate chaining uses extra memory for links. Open addressing needs extra memory implicitly within the table to terminate the probe sequence. Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is to use separate chained hash tables.

More generally

- Comparisons: Open Addressing methods



Linear Probing	Quadratic Probing	Double hashing
Fastest among three	Easiest to implement and deploy	Makes more efficient use of memory
Uses few probes	Uses extra memory for links and it does not probe all locations in the table	Uses few probes but takes more time
A problem occurs known as primary clustering	A problem occurs known as secondary clustering	More complicated to implement
Interval between probes is fixed - often at 1.	Interval between probes increases proportional to the hash value	Interval between probes is computed by another hash function

Question: Now that you have seen in details hashing, do you have an idea of complexity?

How Hashing Gets O(1) Complexity

- We stated earlier that in the best case hashing would provide a O(1), constant time search technique. However, due to collisions, the number of comparisons is typically not so simple. Even though a complete analysis of hashing is beyond the scope of this course, we can state some well-known results that approximate the number of comparisons necessary to search for an item. From the previous discussion, one doubts how hashing gets O(1) if multiple elements map to the same location.
- The answer to this problem is simple. By using the load factor we make sure that each block (for example, linked list in separate chaining approach) on the average stores the maximum number of elements less than the $load\ factor$. Also, in practice this load factor is a constant (generally, 10 or 20). As a result, searching in 20 elements or 10 elements becomes constant.

Complexity intuition

- If the average number of elements in a block is greater than the load factor, we rehash the elements with a bigger hash table size. One thing we should remember is that we consider average occupancy (total number of elements in the hash table divided by table size) when deciding the rehash.
- The access time of the table depends on the load factor which in turn depends on the hash function. This is because hash function distributes the elements to the hash table. For this reason, we say hash table gives O(1) complexity on average. Also, we generally use hash tables in cases where searches are more than insertion and deletion operations.

Hashing Techniques

- There are two types of hashing techniques:
  - static hashing
  - and dynamic hashing

- **Static Hashing**

If the data is fixed then static hashing is useful. In static hashing, the set of keys is kept fixed and given in advance, and the number of primary pages in the directory are kept fixed.

- **Dynamic Hashing**

If the data is not fixed, static hashing can give bad performance, in which case dynamic hashing is the alternative, in which case the set of keys can change dynamically.

Question: Can you imagine problems that are not suitable for hash tables?

Problems for which Hash Tables are not suitable

- Problems for which data ordering is required
- Problems having multidimensional data
- Prefix searching, especially if the keys are long and of variable-lengths
- Problems that have dynamic data
- Problems in which the data does not have unique keys.

Bloom filters

- A Bloom filter is a probabilistic data structure which was designed to check whether an element is present in a set with memory and time efficiency. It tells us that the element either definitely is not in the set or may be in the set. The base data structure of a Bloom filter is a Bit $Vector$. The algorithm was invented in 1970 by Burton Bloom and it relies on the use of a number of different hash functions.

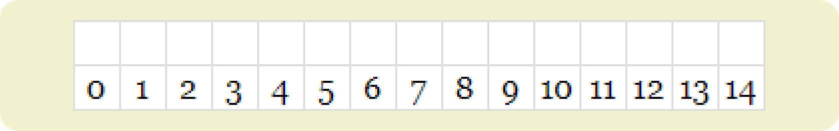
How it works?

- A Bloom filter starts off with a bit array initialized to zero. To store a data value, we simply apply *k* different hash functions and treat the resulting *k* values as indices in the array, and we set each of the *k* array elements to 1. We repeat this for every element that we encounter.
- Now suppose an element turns up and we want to know if we have seen it before. What we do is apply the *k* hash functions and look up the indicated array elements. If any of them are 0 we can be 100% sure that we have never encountered the element before - if we had, the bit would have been set to 1.
- However, even if all of them are one, we still can't conclude that we have seen the element before because all of the bits could have been set by the *k* hash functions applied to multiple other elements.
- All we can conclude is that it is likely that we have encountered the element before.

In summary

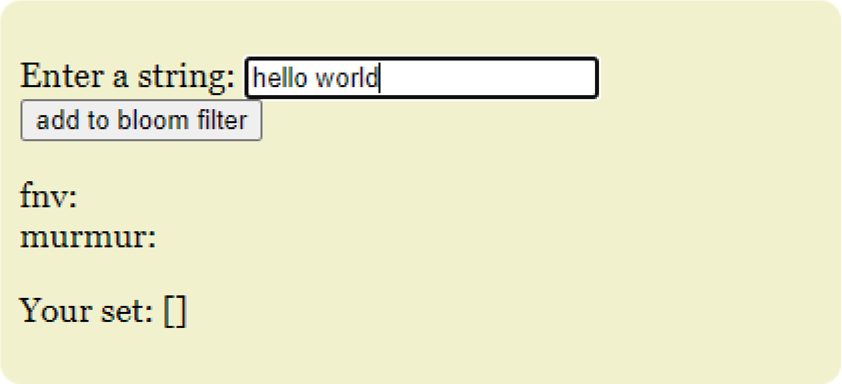
https://llimllib.github.io/bloomfilter-tutorial/

- Initial bloom filter



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

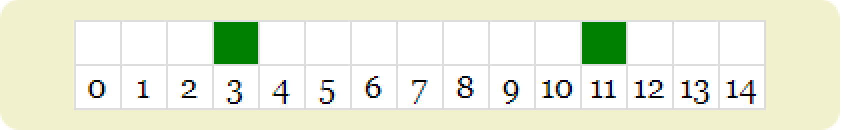
We enter hello world



Enter a string: hello world

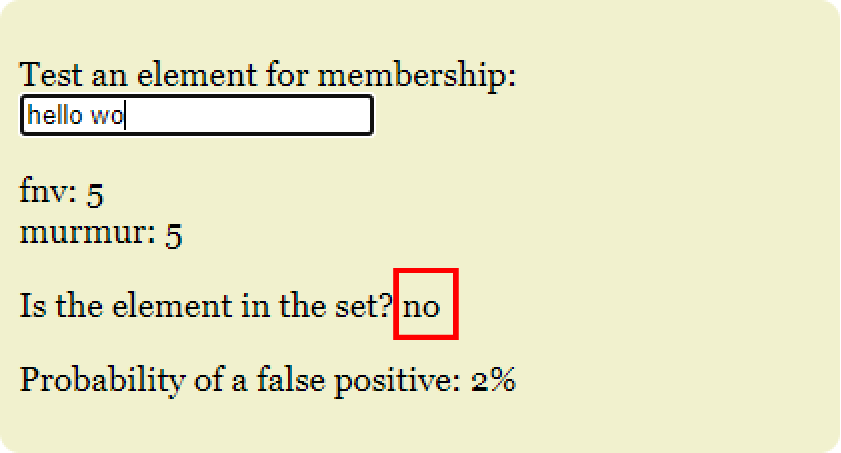
fnv:
murmur:

Your set: []



			🟩								🟩			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

We compare



Test an element for membership:

hello wo

fnv: 5
murmur: 5

Is the element in the set? no

Probability of a false positive: 2%

Bloom filter continued

- Note that it is not possible to remove an element from a Bloom filter. The reason is simply that we can't unset a bit that appears to belong to an element because it might also be set by another element.
- If the bit array is mostly empty, i.e., set to zero, and the *k* hash functions are independent of one another, then the probability of a false positive (i.e., concluding that we have seen a data item when we actually haven't) is low. For example, if there are only *k* bits set, we can conclude that the probability of a false positive is very close to zero as the only possibility of error is that we entered a data item that produced the same *k* hash values - which is unlikely as long as the 'has' functions are independent.
- As the bit array fills up, the probability of a false positive slowly increases. Of course when the bit array is full, every element queried is identified as having been seen before. So clearly we can trade space for accuracy as well as for time.

Removal

- One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains elements that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach, re-adding a previously removed item is not possible, as one would have to remove it from the $removed$ filter.

Selecting hash functions

- The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple $different$ hash functions by slicing its output into multiple bit fields.
- Alternatively, one can pass k different initial values (such as 0, 1, ..., k - 1) to a hash function that takes an initial value – or add (or append) these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in the false positive rate.

Selecting size of bit vector

- A Bloom filter with 1% error and an optimal value of k, in contrast, requires only about 9.6 bits per element — regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. The 1% false-positive rate can be reduced by a factor of ten by adding only about 4.8 bits per element.

Space advantages

- While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers.
- However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element.

Time advantages

- Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, O(k), completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its k lookups are independent and can be parallelized.

In Lab session

- You will play with the concepts and starts getting more and more familiar with hashing, hash table and bloom filter
- This can be useful for your project

- Lab is done by Remy Belmonte