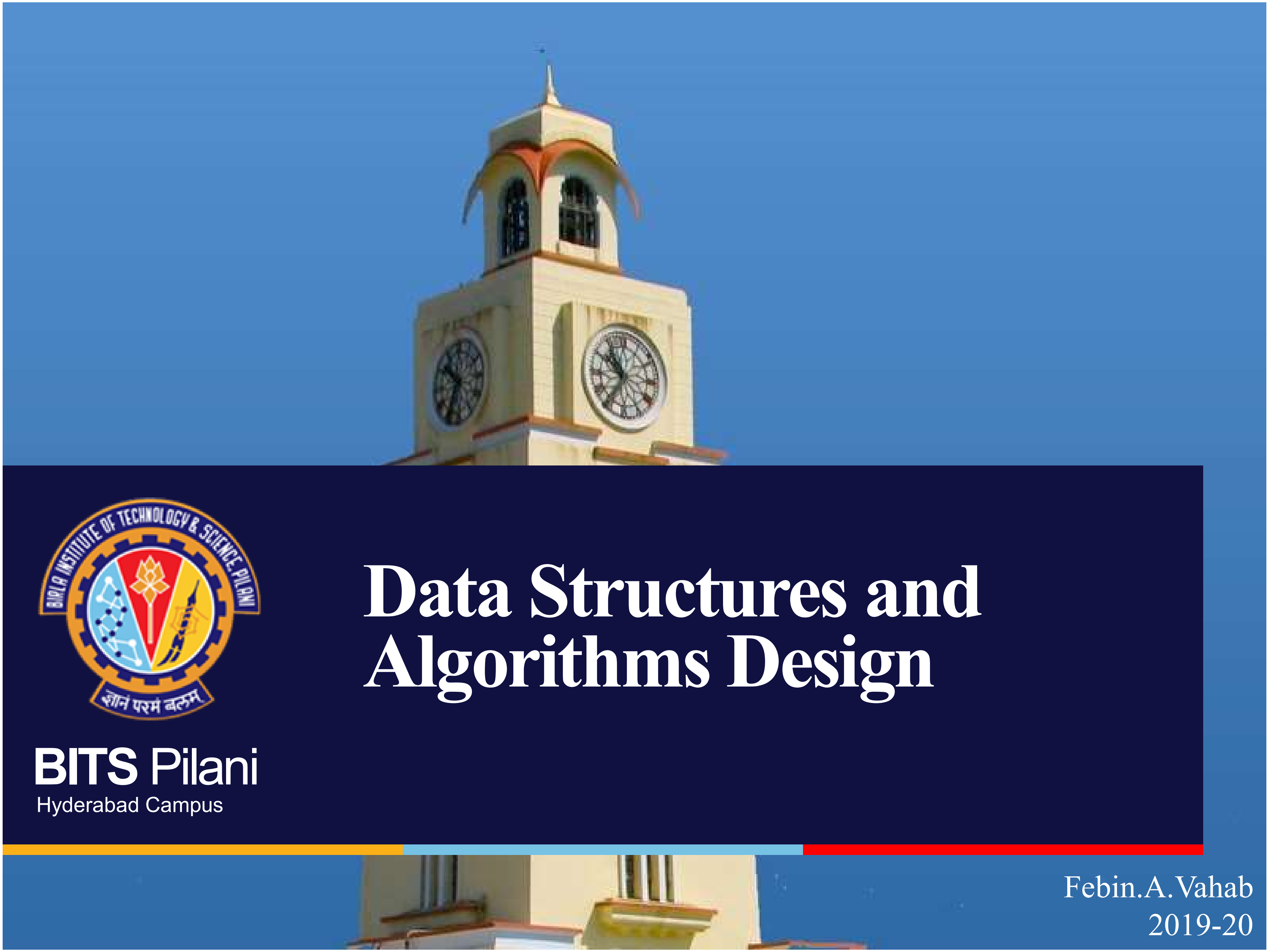


BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

Febin.A.Vahab
2019-20

SESSION 7 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
7	Unordered Dictionary :ADT, Applications Hash Tables: Notion of Hashing and Collision (with a simple vector based hash table)Hash Functions: Properties, Simple hash functions Methods for Collision Handling: Separate Chaining, Notion of Load Factor, Rehashing, Open Addressing [Linear; Quadratic Probing, Double Hash]	T1: 2.5

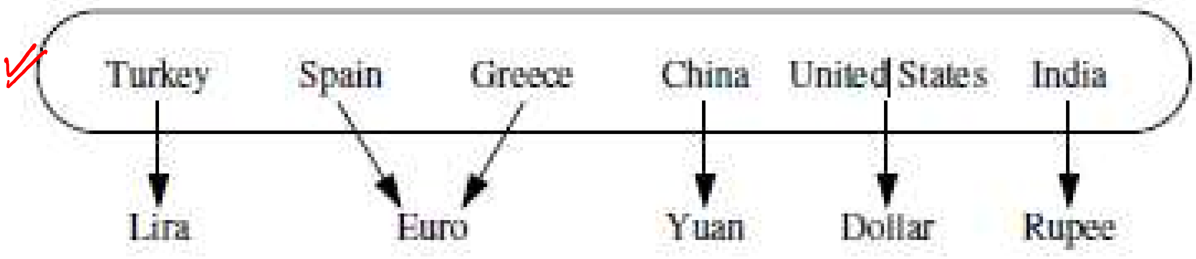
Dictionary ADT

- The dictionary ADT models a searchable collection of key-element items.
- A dictionary stores **key-element pairs (k , e),** which we call **items**, where k is the key and e is the element
- The main operations of a dictionary are searching, inserting, and deleting items
- A key is an identifier that is assigned by an application or user to an associated element.
- *Multiple items with the same key are allowed*
- In cases when keys are unique, the key associated with an object can be viewed as an "address" for that object in memory.

The Dictionary ADT

- For example, in a dictionary storing student records (such as the student's name, address, and course grades), the key might be the **student's ID number. (**we would probably want to disallow two students having the same ID).

The Unordered Dictionary ADT



From countries (the keys) to their units of currency (the values).

- Dictionaries use an array-like syntax for indexing such as **currency[Greece]** to access a value associated with a given key **currency[Greece] = New value** to remap it to a new value.
- Unlike a standard array, indices for a dictionary need not be consecutive nor even numeric.

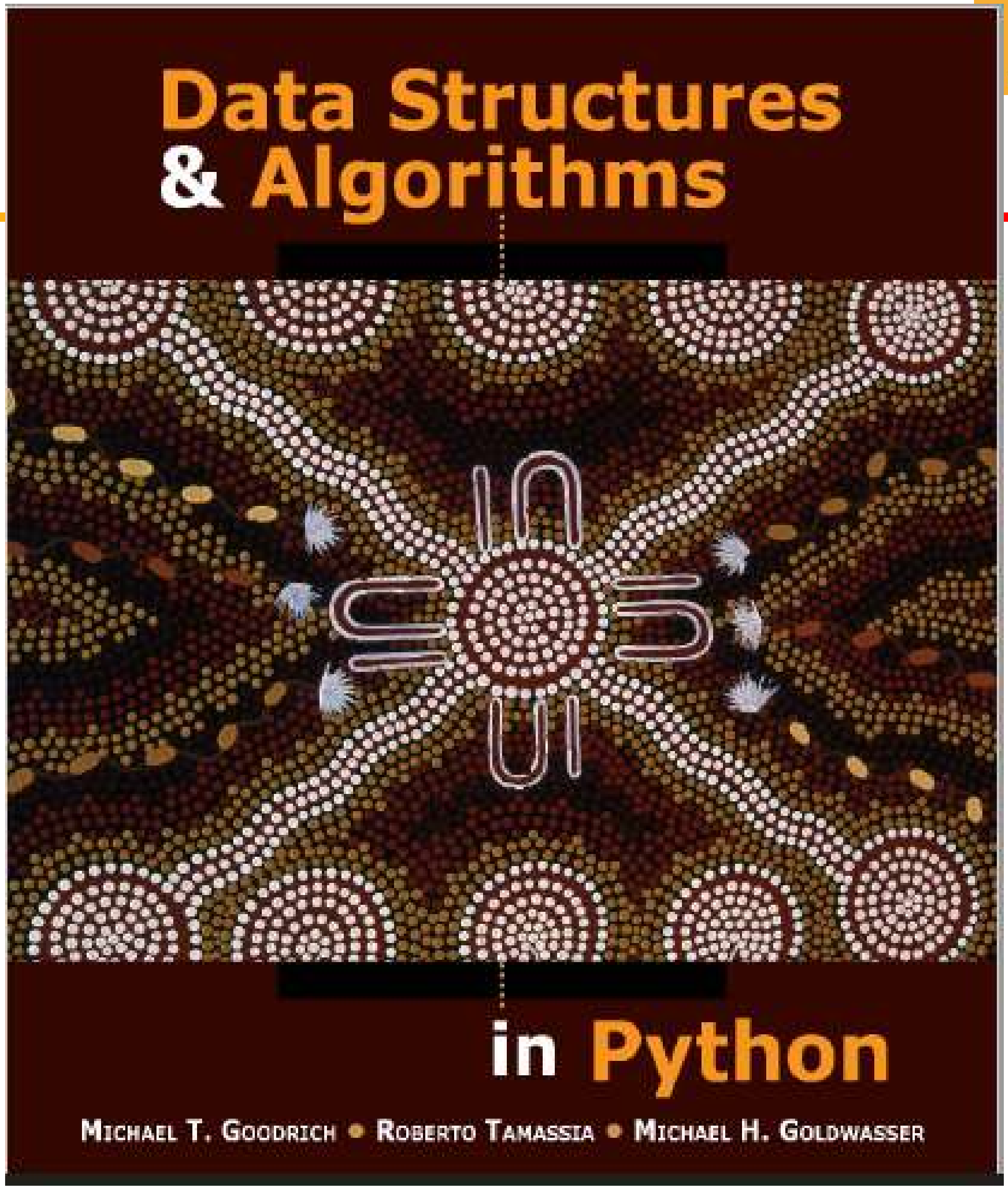
Applications

- The domain-name system (DNS)maps a host name, such as www.bits-pilani.ac.in,to an Internet-Protocol (IP) address.
- A social media site typically relies on a (nonnumeric) username as a key that can be efficiently mapped to a particular user's associated information.
- A computer graphics system may map a color name, such as turquoise , to the triple of numbers that describes the color's RGB (red-green-blue) representation,such as (64,224,208).
- Python uses a dictionary to represent each namespace, mapping an identifying string, such as pi , to an associated object, such as 3.14159.

Applications-HW

- Counting Word Frequencies
  - Consider the problem of counting the number of occurrences of words in a document.
  - A dictionary is an ideal data structure to use here, for we can use words as keys and word counts as values.

Try implementing this using Python Dictionary class.!!!


Data Structures
& Algorithms
in Python
Michael T. Goodrich • Roberto Tamassia • Michael H. Goldwasser

Dictionary ADT methods:

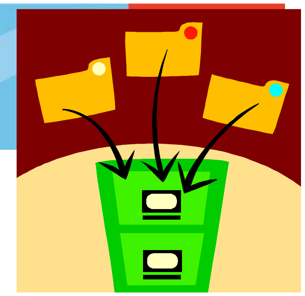
(k, e)

- Dictionary ADT methods:
  - **findElement(k)**: if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - **insertItem(k, e):** Insert an item with element e and key k into D
  - **removeElement(k)**: if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - **size(), isEmpty()**
  - **keys(), elements()-Iterators**

Dictionary ADT methods:

- Special element (NO_SUCH_KEY)is known as a sentinel.
- If we wish to store an item 'e' in a dictionary so that the item is itself its own key, then we would insert e with the method call insertltem(e, e) .
- *findAIIElements(k) - which returns an iterator of all elements with key equal to k*
- *removeAIIElements (k) , which removes from D all the items with key equal to k*

Log Files/Unordered Sequence Implementation



- A log file is a dictionary implemented by means of an **unsorted sequence**
- Often called **audit trail**
- We store the items of the dictionary in a sequence (based on an array or list to store the key-element pairs), in arbitrary order
- The space required for a log file is O(n), since the array data structure can maintain its memory usage to be proportional to its size.

Log Files

- Performance:
  - **Insertion?** insert Item (k, e)
  - **insertItem takes O(1) time** since we can insert the new item at the end of the sequence
  - **Search?? Removal??**[findElement(k), removeElement(k)]
  - **findElement and removeElement take O(n) time** since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

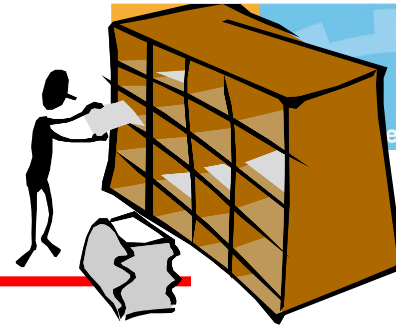
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed
- (e.g., historical record of logins to a workstation)

Dictionary Implementation using Hash Tables

- A hash table for a given key type consists of
  - Array (called table) of size N-(Bucket Array)
  - Hash function h

- **When implementing a dictionary with a hash table, the goal is to store item (k, e) at index i = h(k)** → hash value

Bucket Arrays

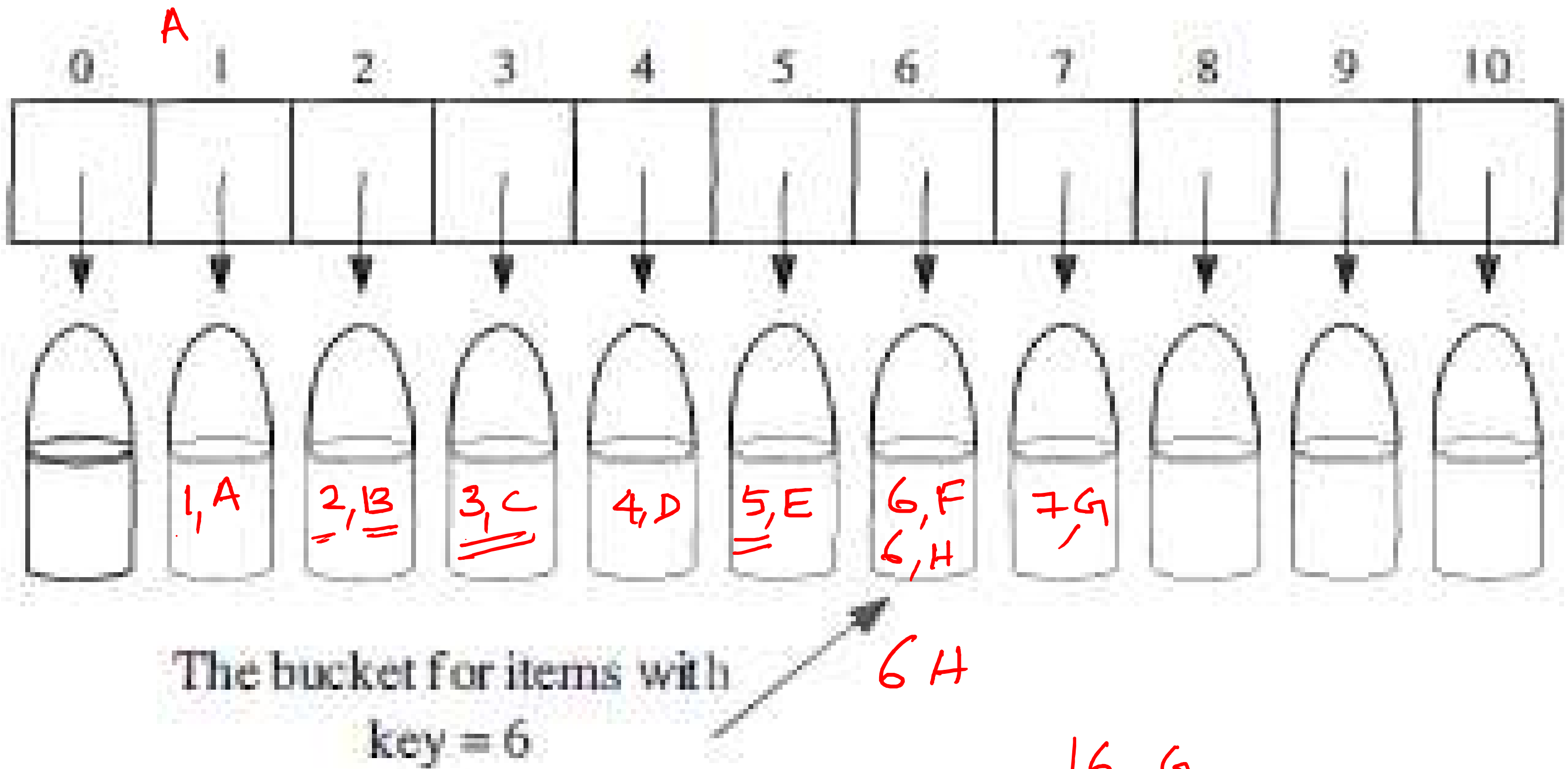


- A bucket array for a hash table is an array A of size N, where each cell of A is thought of as a "bucket" (that is, a container of key-element pairs)
- Integer N defines the capacity of the array.
- **An element e with key k is simply inserted into the bucket A [k]** .
- Any bucket cells associated with keys not present in the dictionary are assumed to hold the special NO_SUCH _KEY object.

Bucket Arrays

A[k]

N = 11



16, G

ABC, 23

Bucket Arrays

- **If keys are not unique**, then two different elements may be mapped to the same bucket in A .
- A ***collision*** has occurred.
- If each bucket of A can store only a single element, then we cannot associate more than one element with a single bucket
- problem in the case of collisions.
- There are ways of dealing with collisions
- The best strategy is to try to avoid them in the first place

Bucket Arrays-Analysis

- **If keys are unique**, then collisions are not a concern, and searches, insertions, and removals in the hash table take
- **worst-case time O(1) .**
- **Uses O(N) space**

Dictionary Implementation using Hash Tables-Motivation

- The bucket array requires keys be **unique integers** in the $\boxed{\textbf{range [0 ,N - 1]}}$, which is often not the case

- There are two challenges in extending this framework to the more general setting
- **What can we do if we have at most 100 entries with integer keys but the keys are in range 0 to 1,000,000,000 ?**
- **What can we do if keys are not integers ?**

Bucket Arrays-Analysis

- What we can do???
- Define the hash table data structure to consist of a bucket array together with a "good" mapping from our keys to **1. integers,**

  2.in the range [0, N - 1]

Hash Functions

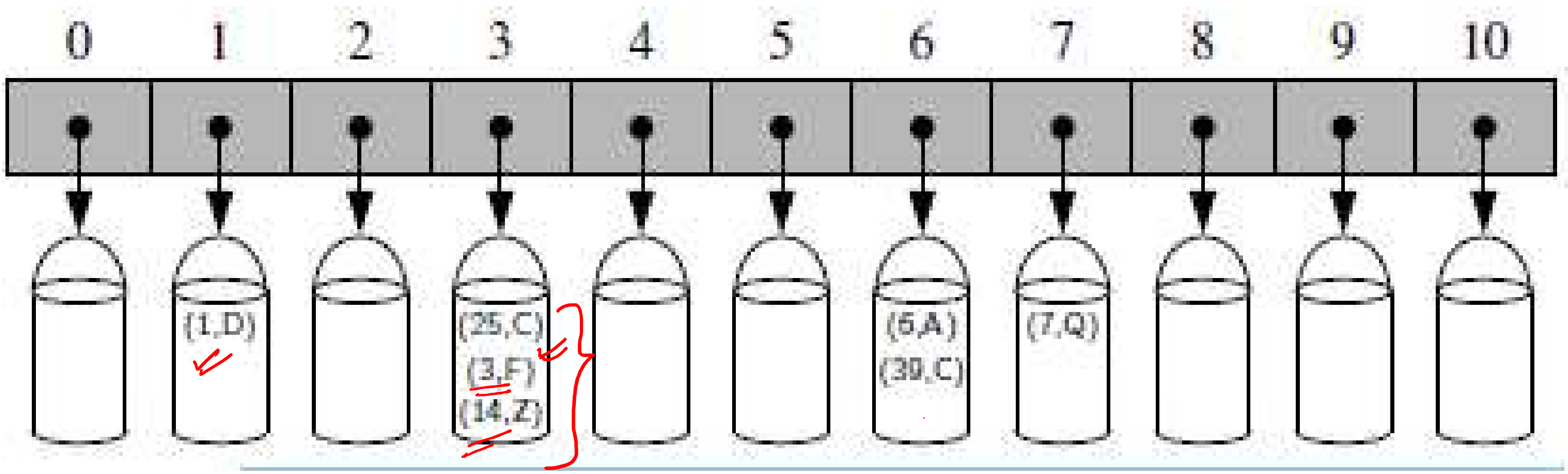
- A hash function h maps keys of a given type to integers in a fixed interval [0, N - 1]
- Now, bucket array method can be applied to arbitrary keys
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer h(x) is called **the hash value of key x.**

Bucket Arrays with a hash function

A[h(k)]



$N = 11$

$h(k) = Key \% N$

A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function

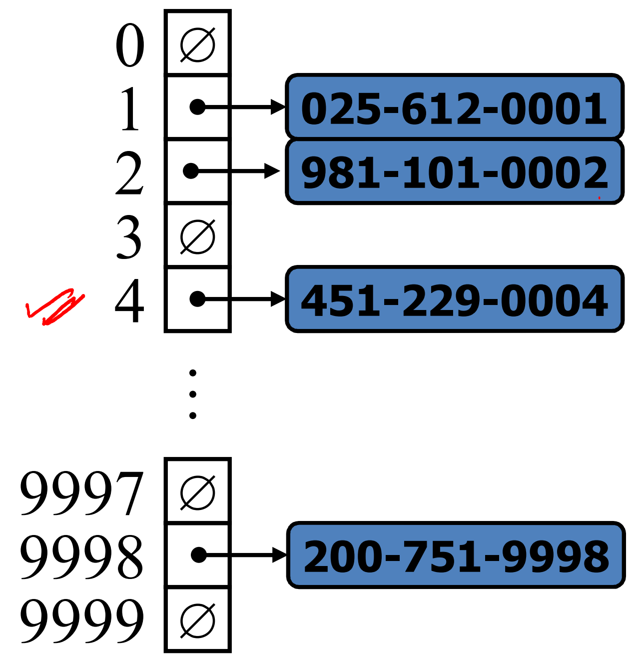
Hash Functions

- **Main Idea**
- Use the hash function value, h(k) , as an index to bucket array, A, instead of the key k (which is most likely inappropriate for use as a bucket array index).
- That is, store the item(k, e) in the bucket **A [h(k)]** .
- A hash function is "good" if it maps the keys in our dictionary so as to **minimize collisions** as much as possible.

Hash Tables-Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Hash table uses an array of size

$N = 10{,}000$ and the hash function
$h(x)$ = last four digits of x



Evaluation of a hash function, h(k)

- A hash function ,h(k) , is usually specified as the composition of two functions:

Hash code map:

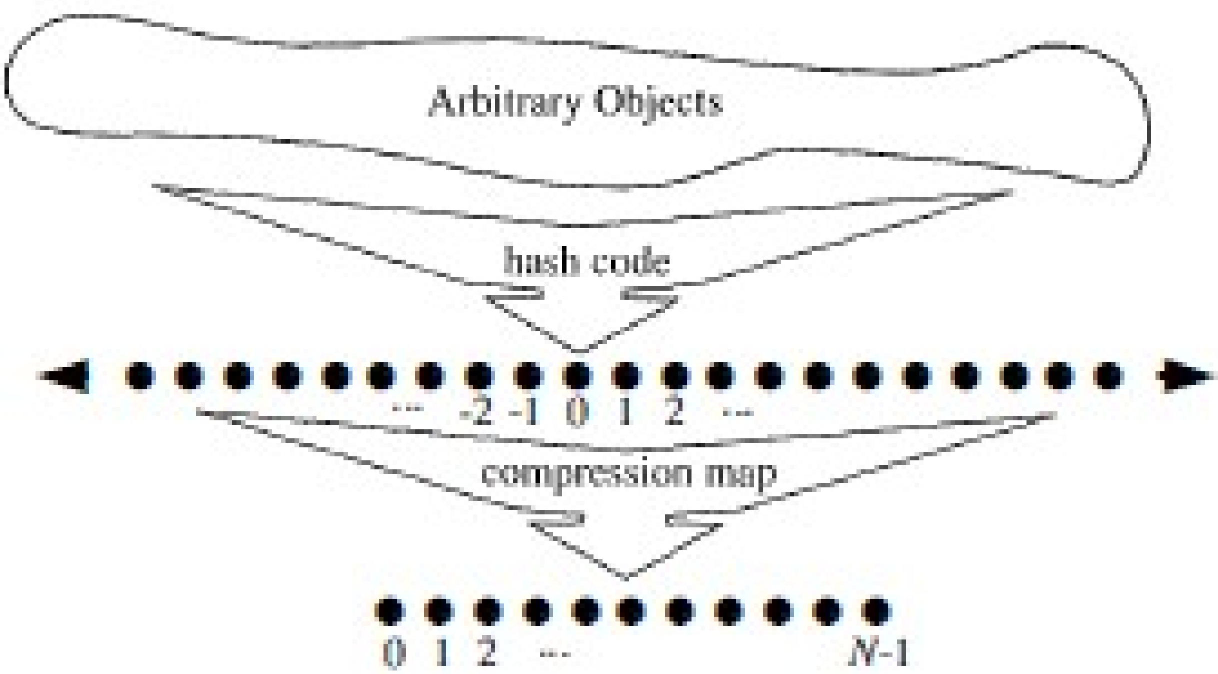
h1: keys → integers[mapping the key k to an integer]

Compression map:

h2: integers → [0, N - 1] size of Bucket Array

[mapping the hash code to an integer within the range of indices of a bucket array]

Evaluation of a hash function



Evaluation of a hash function, h(k)

- The hash code map is applied first, and the compression map is applied next on the result, i.e., $\boxed{\boldsymbol{h}(\boldsymbol{x})} = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$
- The goal of the hash function is to "disperse" the keys in an apparently random way.

Hash Code Maps

key → integers

- To be consistent with all of our keys, the hash code we use for a key k should be the same as the hash code for any key that is equal to k.
- **Memory address as Hash Codes**:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys

Hash Code Maps

key → integer

2 bits

2

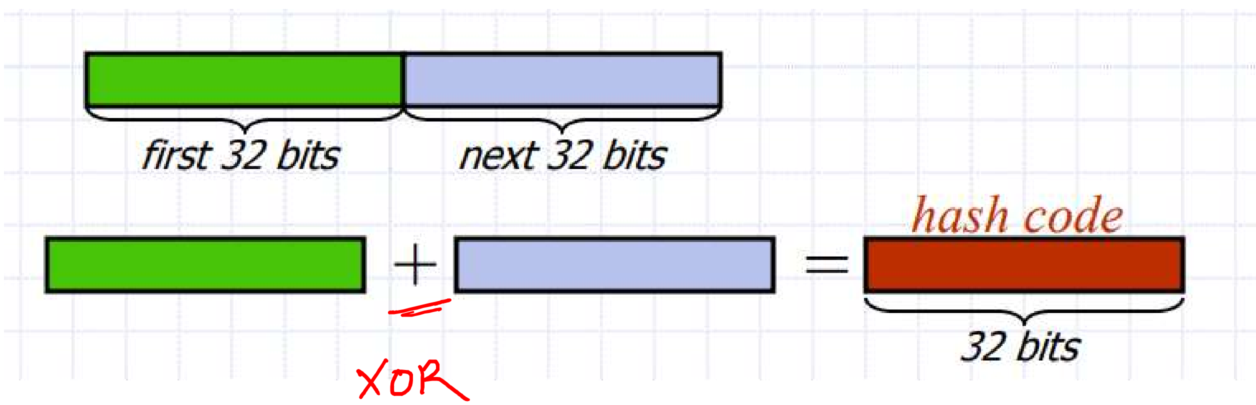
10 | 11

10 | 10

- **Integer cast**:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
  - Float.floatToIntBits(x)
  - For a type whose bit representation is longer than a desired hash code, the above scheme is not immediately applicable.
  - Python relies on 32-bit hash codes. If a floating-point number uses a 64-bit representation, its bits cannot be viewed directly as a hash code.

Hash Code Maps

- **Component sum**:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)



Hash Code Maps

- **Component sum**:
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Ex:

static int *hashCode(**long** i)*

*{**return** (**int**)((i >>> 32) + (**int**) i);}*

Hash Code Maps

- This approach of summing components can be further extended to any object x whose binary representation can be viewed as a k-tuple $(x_0, x_1, \ldots, x_{k-1})$ of integers, for we can then form a hash code by summing x_i.
- Ex. Given any floating point number, we can sum its mantissa and exponent as integers and then apply a hash code for long integers to the result.
- These computations of hash codes for the primitive types are actually used by the corresponding wrapper classes in their implementations of the method hashCode().

Hash Code Maps-Strings

- Suppose we apply the above idea to string
- Sum up ASCII (or Unicode) values for characters in string s
- Component Sum Hash code for string “abracadabra” is ASCII(“a”)+ ASCII(”b”)+…+ ASCII(”a”)
- *Problem:*
  - Lots of collisions possible
  - position of individual characters is important, but not taken into account by the component sum hash code

S P O T

83 + 80 + 79 + 84

POST

POTS

Hash Code Maps

- **Polynomial accumulation**:
  - We choose a nonzero constant, a != 1, and calculate
  $(x_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2} a + x_{k-1})$
  as the hash code, ignoring overflows.
  - This is simply a polynomial in a that takes the components $(x_0, x_1, \ldots, x_{k-1})$ of an object x as its coefficients
  - Especially suitable for strings (e.g., the choice a = 33 gives at most 6 collisions on a set of 50,000 English words)
  - Polynomial p(a) can be evaluated in $O(n)$ **time** using **Horner's rule**
  - ***Refer 30.1 in R2***

Hash Code Maps

S	P	O	T	S
83	80	79	84	83

$$h(SPOTS)=83\,(\underline{a^4})+80\,(a^3)+79\,(a^2)+84\,(a^1)+\underline{83}\,(a^0)$$

Hash Code Maps

- Many Java implementations choose the polynomial hash function.
- For the sake of speed, however, some Java implementations only apply the polynomial hash function to a fraction of the characters in long strings, say every 8 characters.
- This computation can cause an overflow, especially for long strings. Java ignores these overflows
- 31,33, 37, 39, and 41 are particularly good choices for ‘a’ when working with character strings that are English words
- Default Java String.hashCode() uses $a = 31$

Hash code in python

```
In [12]: print('Hash for 220 is:', hash(220))

         # hash for decimal
         print('Hash for 220.34 is:',hash(220.34))

         # hash for string
         print('Hash for Data is:', hash('Data'))

Hash for 220 is: 220
Hash for 220.34 is: 783986623132664028
Hash for Data is: 2539907043859605924

In [13]: id(220.34)

Out[13]: 3207284319792

In [14]: id('Data')

Out[14]: 3207240693944
```

Compression Maps

integer key → [0, N-1]

- **Division:**
  - **Let $y=h_1(x)$//integer hash code for a key object k**
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - **The reason has to do with number theory and is beyond the scope of this course!!!**

[200, 205, 210, 215, 300, 305, 310, 315, 400, 405, 410, 415]

N=100   N=101

[0, 5, 10, 15, 0, 5, 10, 15, 0, 5, 10, 15]

Compression Maps

- **Multiply, Add and Divide (MAD):**
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

  Hash index = ((a × hashCode + b) % p) % N with:

  p = prime number > N

  a = integer from range [1..(p-1)]

  b = integer from range [0..(p-1)]

  - From Mathematical analysis (group theory), this compression function will spread integer (more) evenly over the range [0..(N-1)] *if* we use a prime number for p

Evaluating Hashing Functions:TRY OUT!!!

The following three hashing functions can be considered:

- t1: using the length of the string as its hash value
- t2: adding the components of the string as its hash value
- t3: hashing the first three characters of the string with polynomial hashing

- The compression function - division method
- The input file can have nearly 4000 random names and 4000 unique words from the C code

input1.txt input2.txt

THANK YOU!

BITS Pilani
Hyderabad Campus