Extended Introduction to Computer Science CS1001.py

Lecture 17B: Hash Functions and Hash Tables (The Dictionary Problem)

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University
Spring Semester 2017
http://tau-cs1001-py.wikidot.com

Lecture 17B

- OOP (lec 15)

- Data Structures
  1. Linked Lists (lec 16)
  2. Binary Search Trees (lec 17)
  3. Hash tables (lec 17-18)
  4. Iterators and generators (lec 19)

Lecture 17B-18: Plan

We'll introduce an additional, very efficient data structure: hash table.

- hash functions
- hash tables: storing and searching elements efficiently
- Resolving collisions:
  - Chaining
  - Open addressing (with the example of cuckoo hashing)

Hash

- Definition (from the Merriam-Webster dictionary):

  hash - transitive verb

  1 a: to chop (as meat and potatoes) into small pieces

  b: confuse, muddle

  2 : to talk about : review -- often used with over or out

  ***Synonyms**: dice, chop, mince*

  ***Antonyms**: arrange, array, dispose, draw up, marshal (also marshall), order, organize, range, regulate, straighten (up), tidy*

- In computer science, hashing has multiple meanings, often unrelated. For example, universal hashing, perfect hashing, cryptographic hashing, and geometric hashing, have very different meanings. Common to all of them is a mapping from a large space into a smaller one.

- Today, we will study hashing in the context of the hash tables

A hash function example

- Hash function: a function that maps a large (possible infinite) set to a smaller set of a fixed size.
- example for a hash function from strings to integers:

```
def hash4strings(st):
    """ ord(c) is the ascii value of character c
          2**120+451 is a prime number """
    s = 0
    for i in range(len(st)):
        s = (128*s + ord(st[i])) % (2**120+451)
    return s**2 % (2**120+451)
```

- Executions in class

- Note that this function spreads the (infinite) set of strings over a finite range (albeit large).

- But what can such a function be possibly good for? soon…

Python's built-in hash Function

- Python comes with its own hash function, from everything immutable to integers (both negative and positive).

```
>>> hash("Benny")
5551611717038549197
>>> hash("Amir")
-6654385622067491745 # negative
>>> hash((3 ,4))
3713083796997400956
>>> hash([3 ,4])
Traceback ( most recent call last ):
    File "<pyshell #16 >", line 1, in <module >
          hash ([3 ,4])
TypeError : unhashable type : 'list '
```

Using Python's hash for storing elements

- We can use hash functions to store elements in a list and search them later.
  - Suppose we have a list of size m.
  - The range of Python's hash function is very large, and includes negative integers. How can we use it to map elements to the table entries?
  - To take care of this, we simply reduce its outcome modulo m, the size of the list.
    - It is recommended to use a prime modulus (for reasons beyond our scope).
  - suppose the elements we want to store have unique keys. Then store element with key in the i'th index in the list :

```
i = hash(key)%m
```

  - Note that Python's hash function is fully deterministic (= not random). Why is this crucial?

Hash Tables

- Let us now define this mechanism more formally.
- Suppose elements belong to a large set (possibly infinite), called the "**universe**", denoted ***U*** (for example, all possible strings, all possible ID numbers etc.)

- Suppose we need to store *n* elements from ***U***, and ***n*** << |***U***|.

  (for example, genes of an organism, ID's of students in class right now)

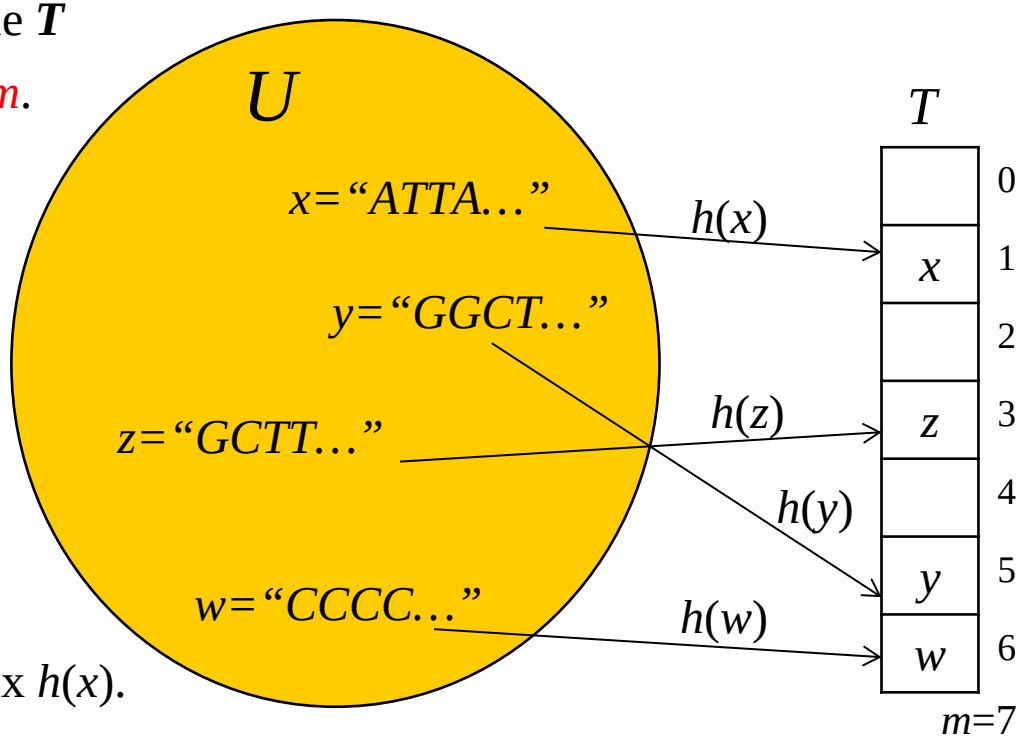
- We can keep the elements in a table ***T*** called **hash table**, whose size is *m*.

  $|T| = m << |U|$

- To map elements from *U* to *T*

  we will use a **hash function**

  $h: U \rightarrow \{0, 1, \ldots, m\text{-}1\}$



Element $x \in U$ will be stored at index $h(x)$.

Implementing Insert, Delete, Search

- The universe of all possible keys, $\mathcal{U}$, is much much larger than the set of actual keys, whose size is up to n.

- Mapping is by a (fixed) hash function, h: $\mathcal{U}$ →T that does not depend on which elements are actually stored in the table.

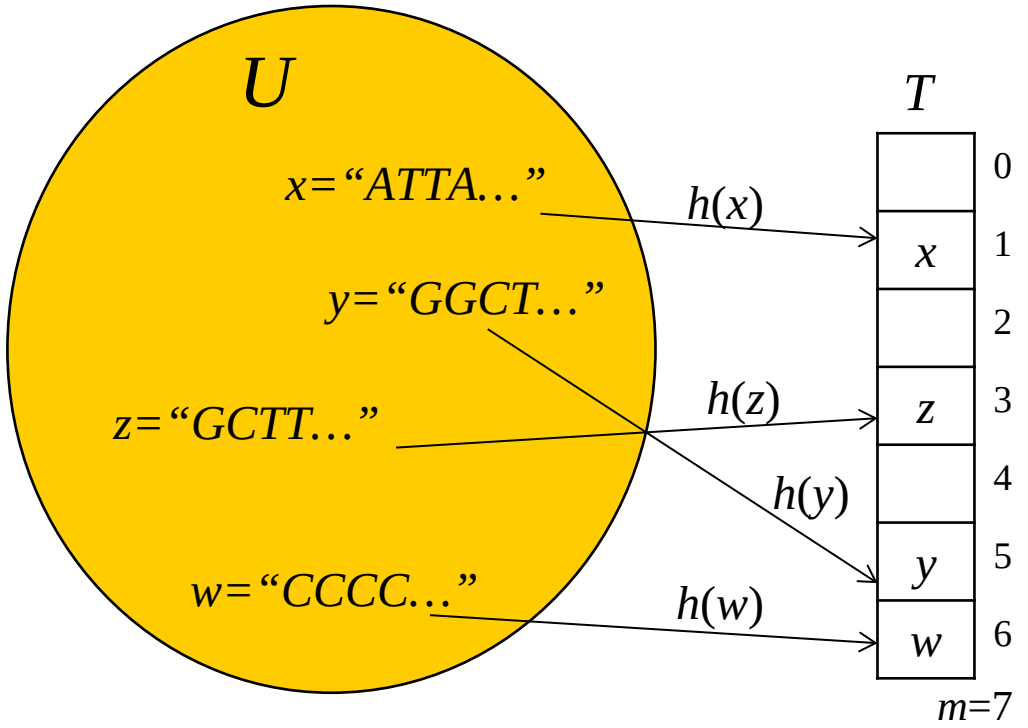
- Given an item with key k ∈ $\mathcal{U}$.

  - Search: compute h(k) and check if in T.

  - Insert: if not in T, insert item to cell h(k).
    If it is in T, can replace item in cell h(k)
    (can also decide to or leave as is)

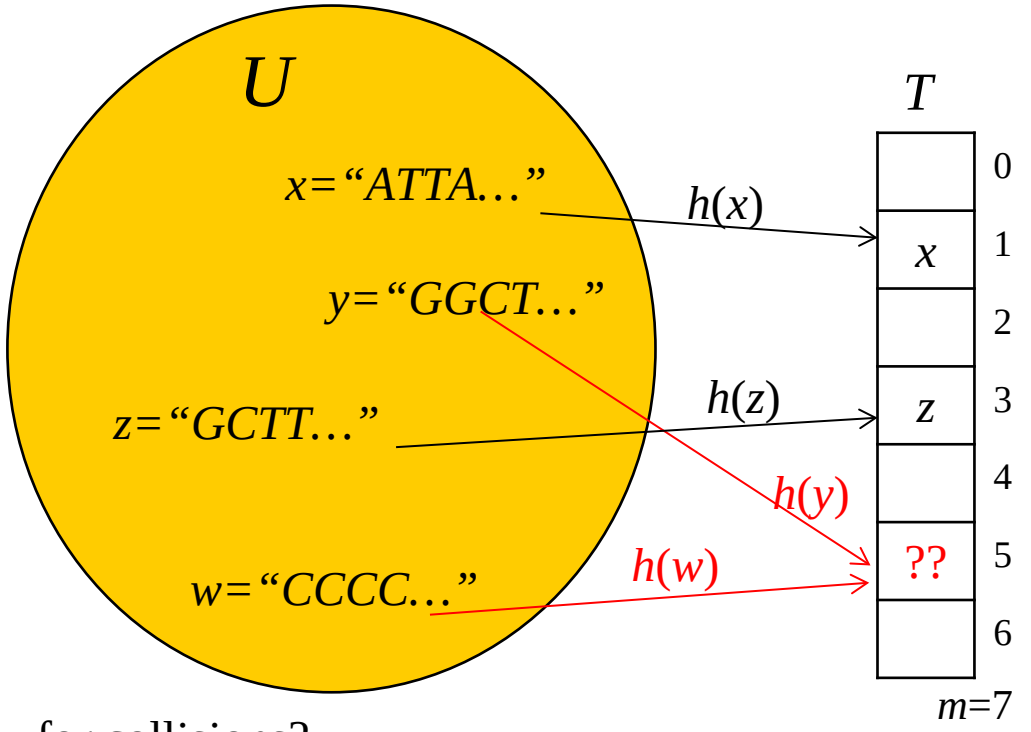
2 main Issues

1. Collisions
2. Is it efficient?



Collisions

- Collision: $h(k_1) = h(k_2)$ for $k_1 \neq k_2$.



- How can we decrease the probability for collisions?
  - recall we do not have any control over which elements will be stored

- Can we totally avoid collisions, if $m << |\boldsymbol{U}|$?

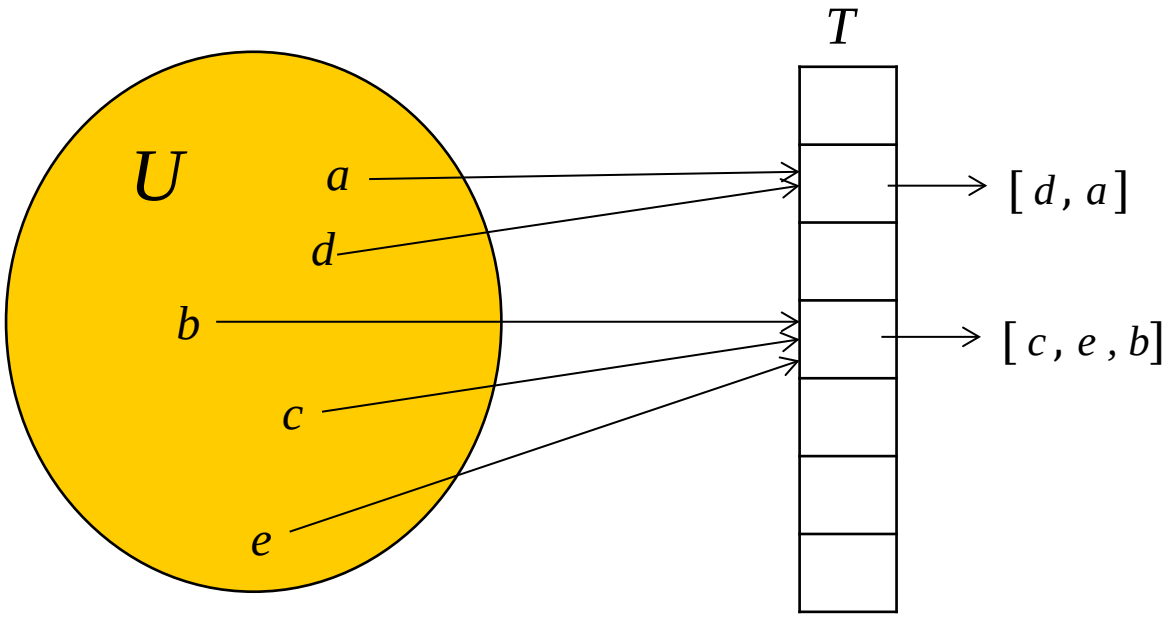
Pigeonhole principle:

if n+1 pigeons enter n holes,

at least 1 hole will contain at least 2 pigeons.

Approaches for Dealing with Collisions: The First Approach - Chaining

Chaining:

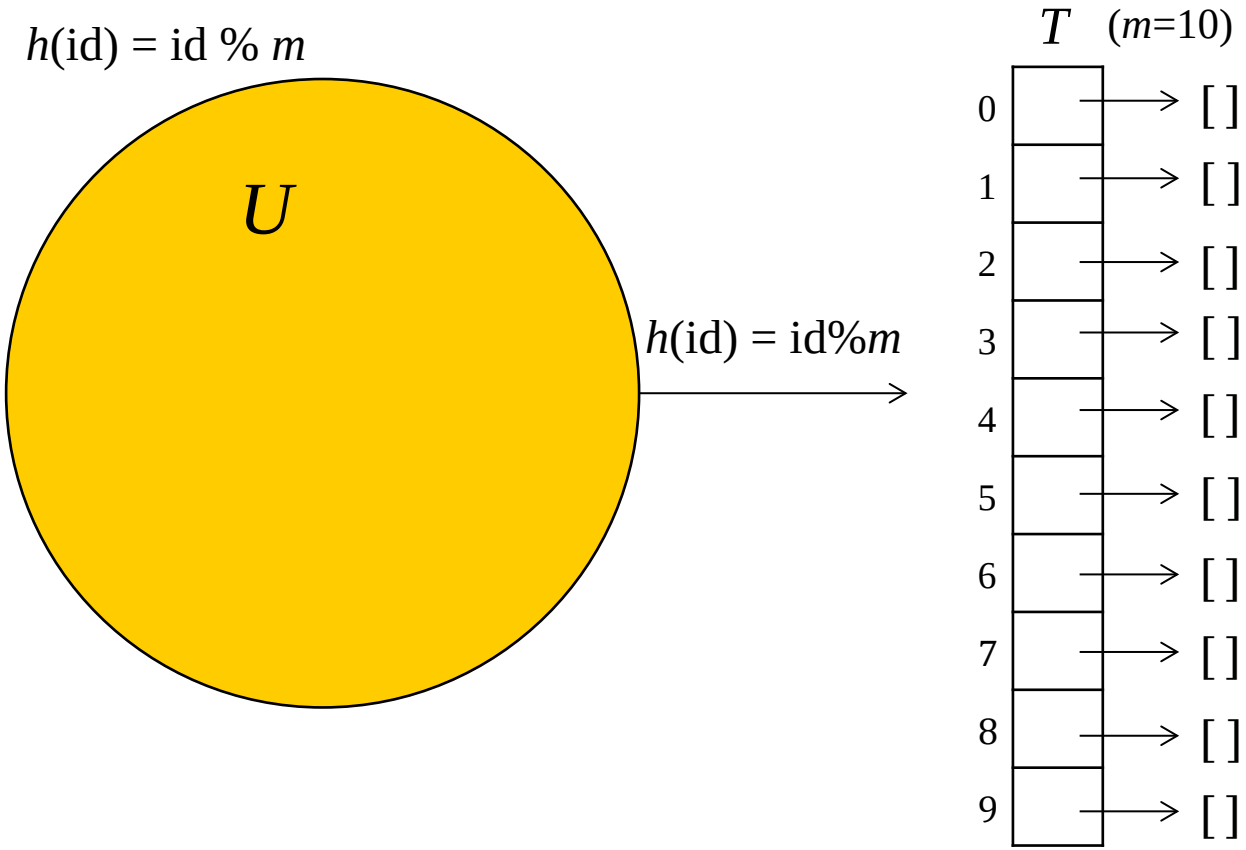


- Each cell in the table will contain a list (can be linked or not)
- How do we search an element in the table then?
- insert?
- delete?

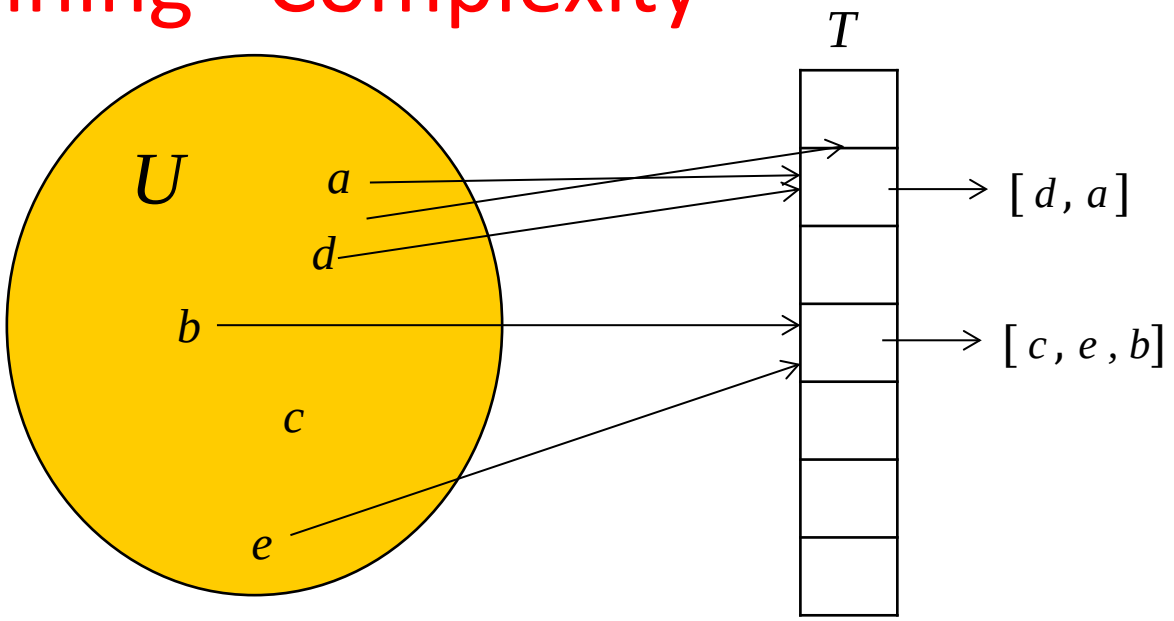
Simple (interactive) Example for ID's

We want to store all students who attended class today, by their ID.

- U = {all possible Israeli ID numbers }
  $|U|$ = ?
- n = ?
- $|T| = m = 10$
- $h(\text{id}) = \text{id} \% m$



Chaining - Complexity



- The average length of a chain is n/m. This is called the "load factor", denoted α.
- We don't want α to be too large or too small (why?)
- if we choose m such that n = O(m), then α=O(1).
- This requires some estimation of the number of element we expect to be in the table, or a mechanism to dynamically update the table size
- what is the average time complexity of search, insert, delete?
- worst case?