Hash Tables

Nicolas Renet

CS224, Fall 2020

Discussion: Associative tables

Python dictionaries

A very useful type in Python is the associative table, or dictionary. Creating a new Python dictionary (`dict`) object:

```
>>> key_value_table = dict()
```

Inserting pairs key/value pairs into the table:

```
>>> key_value_table['Charles']='Hit the road, Jack'
>>> key_value_table['Eno']='Golden hours'
>>> key_value_table['Eno']
'Golden hours'
```

Hashing object references

Even object references can be used as keys:

```
>>> list_of_things=('t-rex', 'LaTeX', 'croissant', 'leek soup') # a tuple
>>> key_value_table[list_of_things]=lambda x: x+2  # tuple ref is the key
>>> key_value_table[ list_of_things ]
<function <lambda> at 0x7f5d22316ea0>
>>> key_value_table[ list_of_things ]( 4 ) # call function from table
6
```

Listing all keys, values, and pairs:

```
>>> key_value_table.keys()
dict_keys(['Charles', 'Eno', ('t-rex', 'LaTeX', 'croissant', 'leek soup')])
>>> key_value_table.values()
dict_values(['Hit the road, Jack', 'Golden hours', <function <lambda> at 0x7f5d
>>> key_value_table.items()
dict_items([('Charles', 'Hit the road, Jack'), ('Eno', 'Golden hours'),
    \ (('t-rex', 'LaTeX', 'croissant', 'leek soup'),
    \ <function <lambda> at 0x7f5d22316ea0>)])
```

Associative tables

Associative tables are an essential structure, that you expect to find in the standard library of every language:

- Dictionary (Python)
- HashMap (Java)
- Hash (Ruby)
- Map<T,T> (C++)

If Javascript does not provide a specific associative type, it is because every object in JS is a dictionary! Object properties can be set dynamically as key/value pairs, using either the dot notation, or a subscript:

```
> country =  {}
> country.name='France'
> country['capital']='Paris'
> country['name']
'France'
> country
{ 'name': 'France', 'capital': 'Paris'}
```

Associative tables are space-efficient, and fast: on average, search operations take $\Theta(1)$.

This lecture is about implementing such tables, and establishing their properties.

Direct Addressing

What is an element?

When introducing other dictionary sets, such as queues, and linked lists, we avoided being overly specific about the kind of elements they would store. This lecture is based on similar assumptions:

- the **element** of a set is uniquely identified by its **key** (typically: a numerical value, or a string)
- some **satellite data** may be associated to the key: in this case, the set element is a structure, or object, that has the key as one of its fields

Example: in a healthcare management system, the SSN is used as key for each patient account; a PatientAccount object stores the key, as well as the name, age, address, etc.

For the sake of clarity, this lecture sometimes does not bother about the data, and just stores the key.

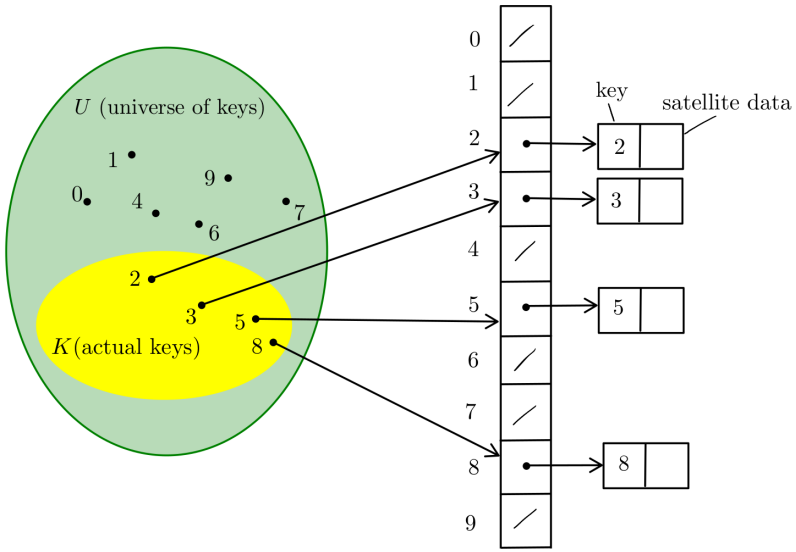
Principle

`The key is the address.`

We store m elements, where each element x is identified by its key k.

Use an m-**element table** T (an array) $T[0 \ldots m-1]$, where all **slots** are initialized to Nil, and equip it with the following functions:

```
Direct-Address-Search(T, k):
    return T[k]
Direct-Address-Insert(T, x):
    T[x.key]=x
Direct-Address-Delete(T, x):
    T[x.key]=Nil
```



Since they only involve a constant number of read/write operations in random access arrays, all three functions run in $\Theta(1)$.

Limitations

Direct-address tables are trivial to implement, and may save you some time in prototyping phase, when your standard library does not already provide hash tables (in C, for instance).

However, the drawback of direct-address tables is obvious: **space complexity** is the issue.

- in some cases, the table size may be smaller than the value of the largest key: f.i. if every key k verifies $1\,000\,000 \leq k \leq 1\,000\,100$, we can subtract $1\,000\,000$ from each k in order to compute the slot index $0 \leq i \leq 100$.
- However, the table must be at least **as large as U, the set of possible keys**, which means that the *range* of values must be narrow in order for the table to have a practical size.
- If the set of actual keys is small, most of this space is wasted.

Application: storing applications settings

Direct-address tables work well when the set of possible keys (or universe) $U = \{0, 1, \ldots, m-1\}$ is small. As an example, let us consider a typesetting web app where a number of settings are represented by 13 enumerated constants, such as:

```
from enum import Enum

class Config(Enum):
    DEBUGGING_MODE=0,
    USE_MATHJAX=1,
    MAIL_ADDRESS=2,
    ...
    USE_LOCAL_STORAGE=12
```

Each constant is a key. To store the data, use an array with 13 slots, where the key is a subscript. For instance:

```
setting=[ None ]*13
settings[Config.MAIL_ADDRESS]='nprenet@bsu.edu'
settings[Config.USE_LOCAL_STORAGE]=True
```

Application: storing Boolean constants

If only Boolean settings are used (no satellite data), we can use a 13-bit vector, or an integer, where the bit corresponding to a given setting can be set or read with bitwise operators:

```
settings = 0
settings |= (1<<USE_MATHJAX)        # set bit 2 to True
settings |= (1<<USE_LOCAL_STORAGE) # set bit 12 to True
settings & (1<<USE_MATHJAX)         # check that bit 1 is set
settings ^= (1<<USE_LOCAL_STORAGE) # unset bit 12
```

Why not go one step further and store the keys directly as powers of 2? For example, given the following enumerated constants

```
USE_MATHJAX=1              # bit 0
USE_TRUE_TYPE_FONTS=2      # bit 1
ALLOW_SCRIPTING=4          # bit 2
...                        ...
USE_LOCAL_STORAGE=2**12    # bit 12
```

We can store and check them as follows:

```
settings |= (USE_MATHJAX | ALLOW_SCRIPTING ) # set both bits 0 and 2
settings & ALLOW_SCRIPTING                    # check that bit 2 is set
```

Hash Tables

Benefits

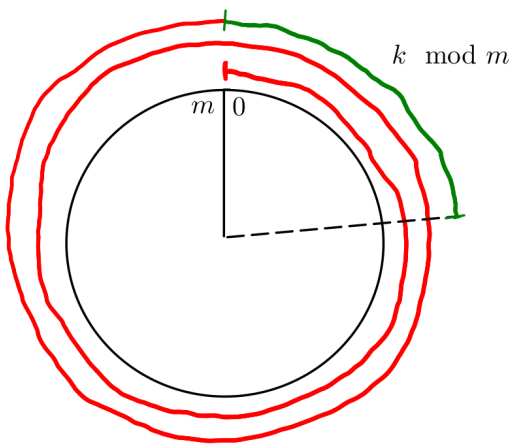
Hash tables address the limitations of direct-address tables:

- their storage requirements do not depends on the key values: the size of the table only needs to be proportionate with the size of the key set (K).
- they guarantee $\Theta(1)$ operations on average

Principle: hashing

Given a table of size m $T = [0 \ldots m-1]$, a **hash function** $h(k)$ is used to compute a slot in T:

- $h(k)$ reduces the range of possible values for a slot index, ensuring that the **hash value** is always within $[0 \ldots m-1]$, even if $m < |U|$, where U is the universe of possible values (typically: $\mathbb{N}$).

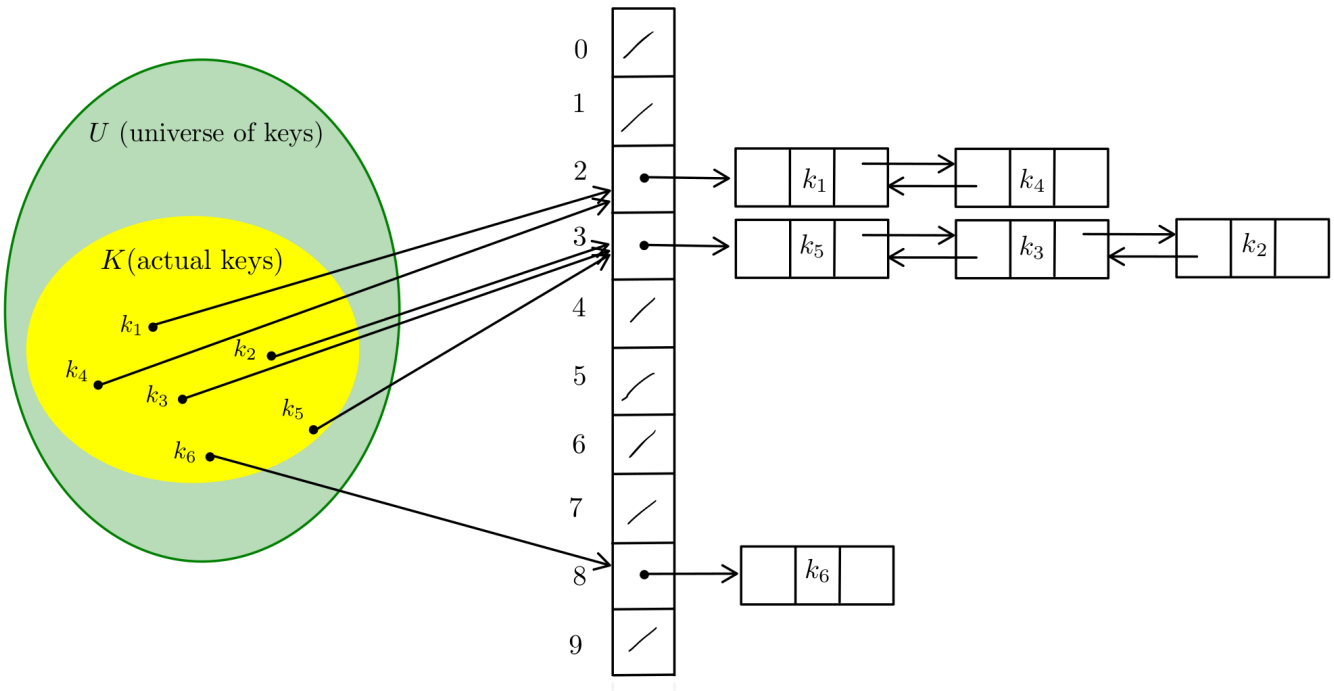


For example, no matter how large the key value, we can wrap it around the modular circle of perimeter m:

$$h(k) = k \mod m$$

Resolving collisions

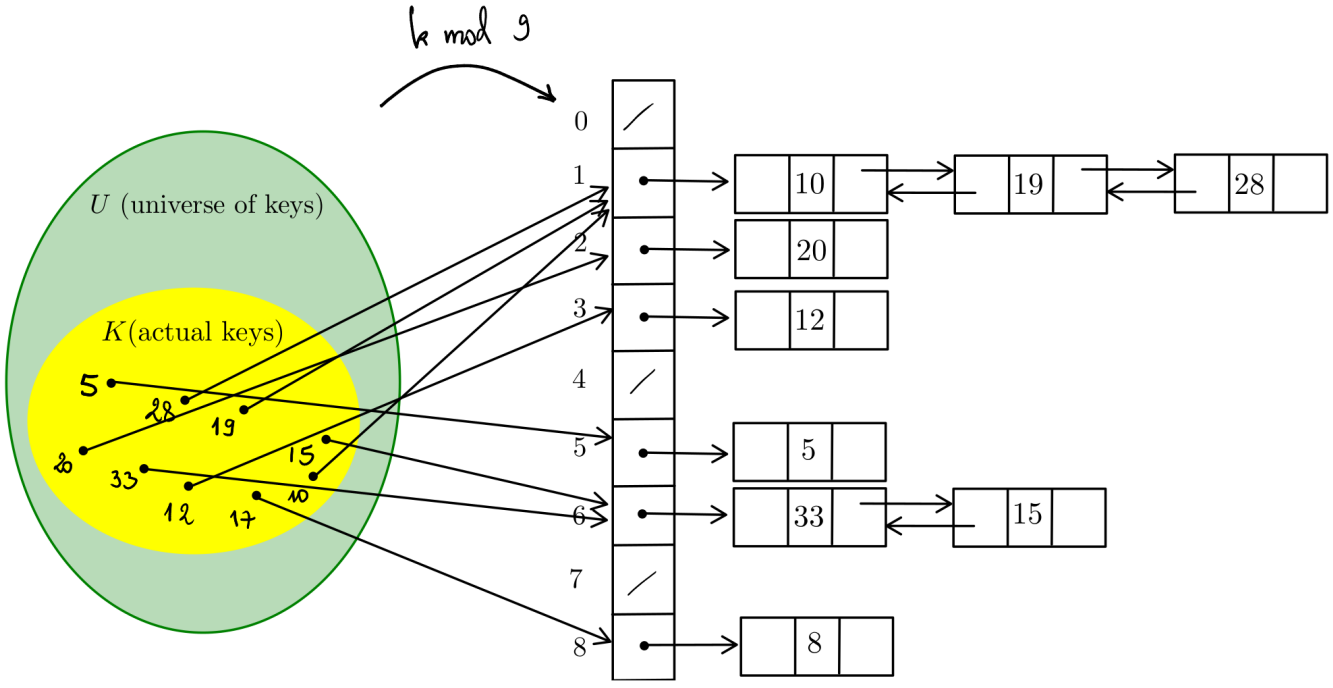
- because $m < |U|$, two keys may hash to the same slot (**collision**): a good hash function minimizes the probability of collisions (they appear to be random).
- when a collision happens, the colliding keys are **chained** in the same slot, in a doubly-linked list (where deletions execute in $\Theta(1)$)



An example

Example: We insert the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ into a hash table of size 9 with collisions resolved by chaining, and the following hash function: $k \mod 9$.

- The keys are inserted in the order in which they are given.
- Remember that insertions in linked-list happen at the head: the leftmost key has been inserted last



Hash table operations

```
Search(T, k):
    return T[ h(k) ]
```

$T_{Search}(n) = \Theta(1)$ on average (proof

```
Insert(T, x):
    T[ h(x.key) ]=x
```

$T_{Insert}(n) = \Theta(1)$

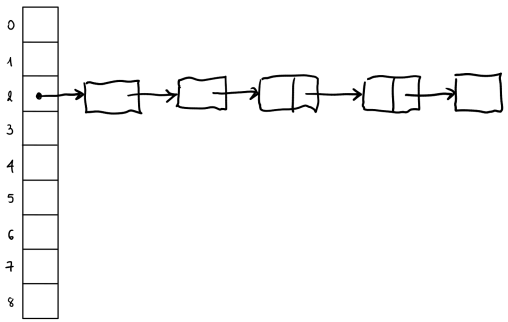
```
Delete(T, x):
    Delete x from list T[ h(x.key) ]
```

$T_{Delete}(n) = \Theta(1)$

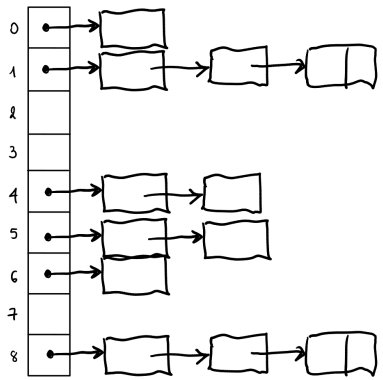
Note that in all 3 operations, the first step is to compute $h(x.key)$, a constant-time operation.

Hash functions: distributions patterns

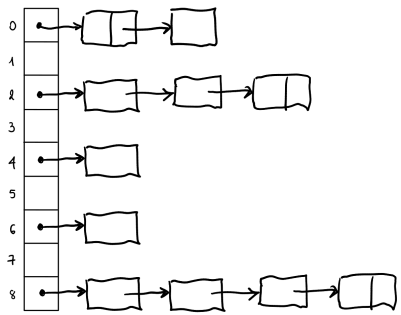
Catherine, Federica, Josep, and Javier have all been given a set of random keys, to be stored in a hash table T of size m. Because everyone uses a different hash function, their collision patterns are quite different from each other:



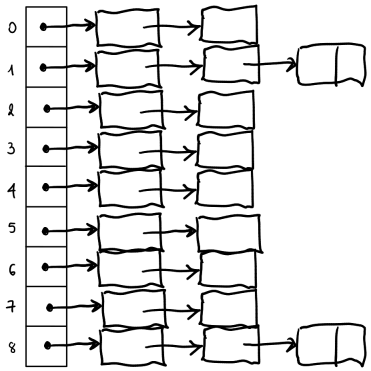
(Catherine)



(Federica)



(Josep)



(Javier)

Who do you think uses a suitable hash function?

What is a good hash function?

Federica's hash function seems to satisfy the **simple uniform hashing** condition:

Each key is equally likely to hash on any slot, independently from where any other key has already hashed.

In more practical terms:

- the resulting distribution of keys should not exhibit an obvious bias for a slot (think of a loaded die)
- a regular pattern in the distribution of keys (f.i. key hashing only to even-numbered slots) suggests that the function does not meet the condition
- the key distribution should not reflect a pattern in the data: by their nature, data most often follow patterns (apparent or not), but a hash table should not make them apparent!

What is a good hash function? Answer to the exercise.

Federica's table shows no obvious pattern of collisions, but a *random distribution of the keys* over the table.

- Catherine's is a worst case, where all keys hash to the same slot.
- Josep's has less obvious defect: only the even subscripts are actually used.
- As for Javier's distribution, it is too good to be true: its hash function is so well-behaved that every list has length $\alpha = \frac{n}{m}$! Not what would you expect from a random distribution of a handful of keys. There is something fishy here, like a pattern in the data, or a ad-hoc hash function that only works on a small set of keys.

Hashing methods: Division

Because hash functions sometimes need to be tailored to a specific domain or type of input, they are still the subject of active research. The methods shown in this lecture may nonetheless be sufficient for most homebrew implementations.

The **division method** is the one we used in the example above. It is simple to code, and works well enough for non-critical applications. We wrap the key around a modulus circle of perimeter m, in order to assign a slot in $T[0 \ldots m-1]$:

$$h(k) = k \mod m$$

(In most languages, the **mod** function is implemented through the % operator.)

Hashing methods: multiplication

Suppose that every key k is a real number such that verifies $0 \leq k < 1$: then it is easy to map every key to a subscript in $\{0, 1, \ldots, m-1\}$ by computing

$$h(k) = \lfloor km \rfloor$$

This is the basic intuition of the multiplication method. In practice, most keys are integers, therefore you need a couple more steps. The recipe:

1. Multiply k by a real constant A in the range $0 < A < 1$. Donald Knuth suggests that $A = (\sqrt{5} - 1)/2$ works quite well.
2. Extract the fractional part of kA (or: $kA \mod 1$)
3. Multiply this value by m and take the floor of the result

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

Appendix A: Implementing the Multiplication Method shows how to implement this operation efficiently with integer arithmetic and bitwise operators.

Non-numerical keys: strings

Typically, the universe of keys is $\mathbb{N} = \{0, 1, 2, \ldots, ...\}$, which covers:

- typical numerical IDs (SSN, student IDs, license number, etc.)
- object references (or any other memory address) -as seen in the discussion

Strings, however, must be hashed in two steps:

1. Encode a n-character long string $s[0 \ldots n-1]$ as an integer - by interpreting each character as a digit (character code), with a base r appropriately chosen:
   $radix(s) = s[0] \times r^{n-1} + s[1] \times r^{n-2} + \cdots + s[n-1] \times r^2 + s[n-2] \times r + s[n-1]$
   The integer so obtained can be very large, depending on your choice of radix r. Base 128 matches the ASCII character set, but keep in mind that most real-life strings nowadays are made of multibyte-characters (UTF-8). Even if Python allows for manipulating arbitrary large integers, you may want to choose a smaller radix, or encode only parts of the longer strings (every other character, f.i.): in both cases, two distinct strings may map to the same integer.
2. Compute $h(radix(s))$

From a string to a integer: example

Interpret the string ‘python’ as a natural number, using a radix-128 notation:

$$\mathit{radix}(\texttt{'python'}) = \texttt{'p'} \times 128^5 + \texttt{'y'} \times 128^4 + \texttt{'t'} \times 128^3 + \texttt{'h'} \times 128^2 + \texttt{'o'} \times 128 + \texttt{'n'} \tag{1}$$

$$= 112 \times 128^5 + 121 \times 128^4 + 116 \times 128^3 + 104 \times 128^2 + 111 \times 128 + 110 \tag{2}$$

$$= 3\,881\,016\,375\,278 \tag{3}$$

Exercise: write in Python a function that accomplishes the same purpose.

```
def encode( string, radix ):
    ...
```

Hashing with chaining: analyzing search operations

Worst case: all elements hash to the same slot

Performance is terrible, because the table behaves just as a single linked list:

$$T_{Search_{wc}}(n) = \Theta(n)$$

Average case

After computing $h(k)$ -this takes $\Theta(1)$-, and assuming **simple uniform hashing** (see above)

- in an **unsuccessful search**, the key is *not* in the table yet, and is therefore equally likely to hash on any of the m slots: the expected search time on average is $\alpha = \frac{n}{m}$ elements, so that

$$T_{Search_{avg}} = \Theta(1 + \alpha)$$

- a **successful search** is slightly different: since the key is already in the table, longer lists are more likely to be searched than the shorter ones. However, even in this case,

$$T_{Search_{avc}} = \Theta(1 + \alpha)$$

The (optional) proof is detailed in CLRS3, 11.2 (Theorem 11.2), p. 59-60. My document **The Average Case explained** (separate from this lecture, on Canvas) adds some probabilistic glue to it, in case your fundations on the subject are a bit shaky.

Take-away: constant-time search, on average

Now, if the table size m is chosen to be proportional to n, which a very reasonable requirement, the load factor, or average length of a list verifies:

$$\alpha = \frac{n}{m} = \frac{n}{cn} = \frac{1}{c} \text{ with } c \text{ a constant}$$

Therefore, the average time complexity of search operations is

$$T_{Search_{avg}}(n) = \Theta(1 + \alpha) = \Theta(1 + c) = \Theta(1)$$

Appendices

Appendix A: Implementing the Multiplication Method

The multiplication method computes:

$$h(k) = \lfloor m \times (kA \bmod 1) \rfloor \text{ with } A = (\sqrt{5} - 1)/2$$

We use low-level bitwise operations to implement this function. Given

- a choice of table size $m = 2^p$;
- a length of machine word w, that fits the largest key, for instance, $w = 64$;
- a choice of integer $s = A \times 2^w$: because, in floating point arithmetic, we use a w-bit integer to represent the fractional value $A = s[0] \cdot 2^{-1} + s[1] \cdot 2^{-2} + \cdots + s[w-1] \cdot 2^{w-1} + s[w] \cdot 2^w$; in Python, for a word size $w = 64$, you would obtain s as follows:

```
>>> import math
>>> int((math.sqrt(5)-1)/2 * 2**64)
11400714819323199488
```

Implementing the Multiplication Method (continued)

Execute the following steps:

1. Compute:
$$k \cdot s$$
2. Extract the fractional part of ks, i.e. the w lower bits, through bitwise AND, with the appropriate mask:
$$fractional = k \cdot s \ \& \ (2^w - 1)$$
3. Extract the p most significant bits, through a shift-right operation:
$$h = fractional \gg (w - p)$$

   This operation has two effects:
   - it multiplies the fractional number by $2^p = m$
   - it drops the fractional part (takes the floor) of the resulting product, by dropping the $w - p$ lower bits

Since Python 3.* uses integers of variable length, w is at the programmer's discretion.