WORKSHEET, WEEK 11: HASHTABLES

A hash function $\mathcal{U} \to \{0, 1, \ldots, m-1\}$ maps the universe $\mathcal{U}$ of keys into m non-negative integers (the size of the hashtable). It is used for map ADT (insert, delete and exact search).

Hash function should follow the following guidelines:

- Keys are discrete objects having a discrete representation (large integers, strings, lists and other data structures). Hash function could be defined on float variables as well, but it is **not** a real-valued function: it depends on the way the real values are encoded.
- The hash of an object must not change during its lifetime (either the hashable object is immutable or the hash function depends only on immutable attributes).
- `a == b` implies `hash(a) == hash(b)`. (The reverse might fail during a hash collision).
- A hash function should be fast. It is often assumed to be constant-time $O(1)$; sometimes it is linear-time $O(|k|)$ for longer input keys k.

Simple Uniform Hashing Assumption: Under this assumption a key maps to any slot $\{0, \ldots, m-1\}$ with the same probability, independent from the hashes of all the other keys.

This can be approximated by randomized seed to the hashing algorithm; hash function should map similar, but non-identical objects far from each other. (People sharing the same lastname or birth year should have considerably different hashes.)

Note: Hashing that we discuss here should not be confused with *cryptographic hashing*. In addition to the properties listed above it would satisfy a few more requirements – it serves as a unique *fingerprint* for the object being hashed. Examples include SHA-256 and other algorithms in SHA-2 family (also MD5, which is no longer considered safe as one can create lots of collisions).

In practice most hash functions are computed as a composition $h(obj) = h_2(h_1(obj))$:

- $N = h_1(obj)$ is a *prehash function* which takes objects of various types and returns large integers.
- $h_2(N)$ is usually a modular division to get a nonnegative integer – an index in a hash table.

For example, in Python the prehash function is named simply `hash(...)`; it returns signed 64-bit integers. It can take different argument types. For small integer arguments it returns the integer itself; larger integer objects are "hashed" to fit into 64-bits. For example, you can run the Python environment on Linux like this (tested on Ubuntu command-line):

```
export PYTHONHASHSEED=0
python
```

On Windows Powershell (such as Anaconda terminal) run Python environment like this:

```
$Env:PYTHONHASHSEED=0
```

On Windows regular terminal run Python environment like this:

```
set PYTHONHASHSEED=0
python
```

```
>>> hash(10**18)
1000000000000000000
>>> hash(10**19)
776627963145224196
>>> hash('\x61\x62\x63')
5573379127532958270
>>> hash('abc')  # 'abc' is same string as '\x61\x62\x63'
5573379127532958270
>>> hash(3.14)
322818021289917443
>>> hash((1,2))
-3550055125485641917
```

Hash Collisions: Finding hash collisions can sometimes be used to slow down Python-related data structures such as dictionaries. Here is advice how to find hash collisions efficiently: https://bit.ly/3nf4bdk. Instead of looping until two hash values will collide, the approach uses some reverse enginering and also "Meet in the Middle" which reduces the number of checks to be made. In an unsophisticated Python implementation of a Web application one could cause massive hash collisions to stage denial of service attacks. Since Python 3.2 the hash computations are randomized with a seed (`PYTHONHASHSEED` environment variable); such attacks are now much harder. See https://bit.ly/3CeAVYi on how these collisions affect other programming languages.

11.1 Hashtables with Chaining

The hash function used in this exercise is computed as Python's `hash()`, and then the remainder `hash(x)` mod 11 is found.

(A) Draw a hashtable with exactly 11 slots (enumerated as `T[0]`, `T[1]`, and so on, up to `T[10]`). Insert the following items into this hashtable (keys are country names, but values are float numbers showing their population in millions):

```
('Austria',8.9), ('Azerbaijan',10.1), ('Belgium',11.6), ('Bulgaria',6.9),
('Estonia',1.3), ('Italy',59.6), ('Latvia',1.9), ('Lithuania',2.8)
```

If there are any collisions between the hash values, add the additional hash values to a linked list using *chaining*. Each item in the linked list contains a key-value pair and also a link to the next item.

(B) What is the expected lookup time, if we randomly search any of the 8 countries to look up its population. Finding an item in a hash table takes 1 time unit; following a link in a linked list also takes 1 time unit.

(C) What is the expected lookup time, if the 11-slot hashtable is randomly filled with 8 keys (each key has equal probability to be in any of the slots). You can find this lookup time rounded up to one tenth (one decimal digit precision) – analytic methods as well as a computer simulation is fine.

11.1.1 Solution

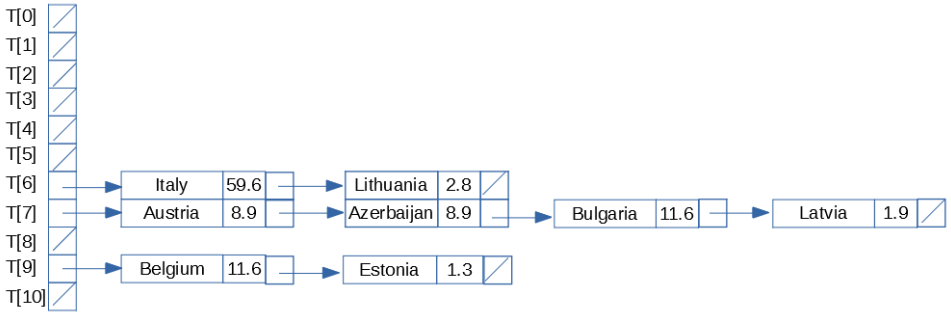
(A) This Python code snippet computes hash values for strings `'a'`, `'ab'`, `'abc'`. First set up the Python environment to compute (repeatable) hash function values:

```
$Env:PYTHONHASHSEED=0
python
```

In the interactive Python environment compute the values of the hash function:

```
list(map(lambda x: hash(x) % 11, ['Austria', 'Azerbaijan', 'Belgium',
'Bulgaria', 'Estonia', 'Italy', 'Latvia', 'Lithuania']))

[7, 7, 9, 7, 9, 6, 7, 6]
```



(B) The lookup time can take values $T = 1$, $T = 2$, $T = 3$, and $T = 4$, but with different probabilities. The expected lookup time for a random value is given by the expression:

$$E(T) = 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{8} = \frac{16}{8} = 2.$$

(C) The *load factor* in this case is $8/11$, so the average linked list has length $8/11$. The total space of this data structure is $O(m + n)$ (first store a table with m entries, then store n items belonging to our map). The total time for a lookup can be computed as $1 + 8/11$ (one plus the load factor of the hashtable).