Rolling Hash

Rolling hash is an abstract data type that maintains a list and supports the following operations:

- `hash()`: computes a hash of the list.
- `append(val)`: adds `val` to the end of the list.
- `skip(val)`: removes the front element from the list, assuming it is `val`.

In the case of strings, the list is a list of characters.

Data Structure - Algorithms for Each Operations

Characters can be interpreted as integers, with their exact values depending on what type of encoding is being used (e.g. ASCII, Unicode). For example the ASCII code for 'A' is 65 and 'B' is 66. This means we can treat strings as lists of integers.

Key idea: treat a list of items as a multidigit number u in base a ('concatenate' list items into a big number). For example, we can choose $a = 256$, the alphabet size for ASCII code.

- `hash()`: u mod p for prime p (division method).
- `append(val)`: $((u \cdot a) + val)$ mod p = $[(u \bmod p) \cdot a + val] \bmod p$
- `skip(val)`: $[u - val \cdot (a^{|u|-1} \bmod p)] \bmod p$ = $[(u \bmod p) - val \cdot (a^{|u|-1} \bmod p)] \bmod p$

Hashing intuition: choose $a = 100$ for easy illustration purpose, and $p = 23$. hash([61, 8, 19, 91, 37]) = (6108199137 mod 23) = 12. In general, hash($[d_3, d_2, d_1, d_0]$) $= d_3 \cdot a^3 + d_2 \cdot a^2 + d_1 \cdot a^1 + d_0 \cdot a^0) \bmod p$.

Sliding intuition:

- list: [3, 14, 15, 92, 65, 35, 89, 79, 31]
- from [3, 14, 15, 92, 65] to [14, 15, 92, 65, 35], we get hash values from 11 to 6.
- from [14, 15, 92, 65, 35] to [15, 92, 65, 35, 89], we get hash values from 6 to 5.

Fast rolling hash:

- Cache the result u mod p.
- Need to avoid exponentiation in skip: cache $a^{|u|-1}$ mod p (skip multiplier).
  - append: multiply it by base
  - skip: divide it by base (division is expensive, can use multicative inverse).

Amortized Analysis

Prove that in-order traversal in a BST using SUCCESSOR (a.k.a. NEXT-LARGEST) is $O(1)$ per node, amortized.

Mention that LIST in the problem set can also be implemented using SUCESSOR, with a similar analysis.

Pre-Hash Functions

Python

Pre-hashing is done by calling the `__hash__` instance method on an object. When overriding `__eq__` you should also override `__hash__`, so that objects can be used as keys in a dictionary.

Good Functions

There are four main characteristics of a good hash function:

1. The hash value is fully determined by the data being hashed.
2. The hash function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates very different hash values for similar strings.

Let's examine why each of these is important.

- Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.
- Rule 2: If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.
- Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.
- Rule 4: In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

Bad Functions

Credit card numbers

Credit card numbers have structure. The first 4 digits in the 16-digit number are the bank number, so using them in a hash function can yield collisions – for example, MIT students most likely have cards issued by the few banks in the area (MITFCU, Bank of America MA, etc.). Also, the last digit is a checksum digit, designed so that most typos can be corrected offline, without querying a bank server, so a hash function that uses the last digit wastes time without gaining any additional randomness.

XORhash

Let key be a list of integers.

```
1 def xorhash(key)
2   h = 0
3   for k in key:
4     h ^= k
5   return h
```

Why is this bad?