Lecture 22: Sets and Dictionaries

built on 2019/07/03 at 15:45:51

In Java and many other modern languages, the *map* (aka. dictionary) data type is one of the most significant and versatile built-in data types. It represents an abstraction that stores a mapping between unique keys and their associated values. As a basic example of this abstraction, we schematically depict mapping from country names and their currencies:

```
Thai    -->  Baht
Japan   -->  Yen
German  -->  Euro
Greece  -->  Euro
USA     -->  Dollar
```

Notice that the keys (country names) are unique whereas the values (currency names) need not be. We can think of it as an extension of a set where every member of the set is "tagged" with a value. Python's syntax in fact reflects this: To represent the information above in Python, we'll simply write

```
{ "Thai": "Baht", "Japan": "Yen", "German": "Euro", "Greece": "Euro", "USA": "Dollar" }
```

The nature of such a data type is that extra keys may be added, existing entries may be deleted, existing mappings may be updated, and queries are perpetual.

In general, it is typically captured as an ADT that provides the following interface at minimum:

- `new` — make a new empty dictionary;
- `get(k)` — retrieve the value corresponding to the key `k`;
- `containsKey(k)` — test if the key `k` is a member of the dictionary
- `put(k, v)` — remember in the dictionary that the key `k` is assigned the value `v`;
- `remove(k)` — remove the key `k` from the dictionary

Notice that the type of the keys and the type of the values are often not the same.

Common applications of the dictionary abstraction include:

- A regular dictionary is a mapping from words to their definitions.
- A book's index is a mapping from terms to the page numbers where the terms appear.
- A university's information system relies on some form of a student ID as a key that is mapped to that student's record.
- The domain name system (DNS) maps a host name (e.g., `www.mahidol.ac.th`) to an IP address (e.g., 202.28.162.3).

From a practical point of view, most modern languages offer some form of the dictionary data type or another. The key question is, *how does that work under the hood?*

1 Basic Implementation Strategies

As with several data types we have looked at recently, we're interested in supporting the operations of the data type as fast as possible using as little space as we can. There are two basic strategies that we have considered in the past for keeping a collection.

First, we could keep the collection as a list of key-value pairs. This strategy appears to be a low-maintenance option but looks to be expensive for operations that require looking up by key.

We could try to improve on the look-up performance by keeping the list sorted by key. While this strategy leads to a marked improvement in the lookup performance, it is a high-maintenance option: keeping the list sorted is expensive.

Notice that in both strategies, to implement `put` and `remove`, we'll first locate that key in the list. If the list is sorted, we could use binary search to locate the key; otherwise, it seems that linear search is about the best thing one can do. Following that, we either update the value or delete that entry altogether. This leads to the following cost table, where we use n to denote the number of keys kept in the table:

	Unsorted List	Sorted List
`get(k)`	$O(n)$	$O(\log n)$
`containsKey(k)`	$O(n)$	$O(\log n)$
`put(k, v)`	$O(n)$	$O(\log n)$ or $O(n)$ if k is new
`remove(k)`	$O(n)$	$O(n)$
Space Usage	$O(n)$	$O(n)$

2 Take III: Trees To Our Rescue

Take any balanced binary search tree data structure for example. It maintains a binary search tree with tree about $O(\log n)$ deep when the tree contains n keys. This means essentially that `split` and `join`—two main primitives in our BST abstraction—can be implemented on $O(\log n)$ time. Now as was discussed, looking up a key, inserting a new item, or deleting an existing one can be implemented in terms of at most one `split` and one `join`. Therefore, we have the following bounds:

	Unsorted List	Sorted List	Balanced BST
`get(k)`	$O(n)$	$O(\log n)$	$O(\log n)$
`containsKey(k)`	$O(n)$	$O(\log n)$	$O(\log n)$
`put(k, v)`	$O(n)$	$O(\log n)$ or $O(n)$ if k is new	$O(\log n)$
`remove(k)`	$O(n)$	$O(n)$	$O(\log n)$
Space Usage	$O(n)$	$O(n)$	$O(n)$

We now know how to support these basic operations reasonably efficiently, in $O(\log n)$ time each. How can we support the likes of set operations—such as union, intersection, set difference, etc.? There are natural analogs of these operations in the setting of dictionaries but for our discussion, we'll keep things simple and only talk about set operations on sets (which only have keys, no associated values).

We'll look at union in detail. The other operations are analogous.

2.1 How to Union

Let's remember that given two sets A and B, the union of A and B is the set $\{x : x \in A \text{ or } x \in B\}$—that is, an element belongs to the union if it is in A *or* it is in B (or both). Therefore, we can imagine implementing the union operation as follows:

```
# assume |B| <= |A|
def basic_union(A, B):
  C = set(A)
  for b in B:
    if (!C.contains(b)) C.put(b)
  return C
```

What's the running time of this approach? Let's assume that $|B| \leqslant |A|$—that is B is the smaller set of the two. Call $n = |A|$ and $m = |B|$. First, we create a new copy of A, taking $O(n)$. Then, we go over each element of B checking if it already belongs to C, and if not, we add it to C. This costs us $O(\log |C|)$ for each element of B and where $|C|$ denotes the size of C at that moment. But $|C|$ is no more than $|A| + |B| \leqslant 2n$. Hence, a total cost of $O(n + m \log n)$. Can we do better? (Ans: You'll see later on)

3 Take IV: A Hash Table

We want better performance. One thing we learned so far was that it takes only $O(1)$ to retrieve an item in an `ArrayList` (or an array in general). Is it possible to use this efficiency to speed up our dictionary?

3.1 Tackle An Easier Problem: Keys Are Small Int's

If keys are small integers, we could just use an array to keep the dictionary. The main idea is for the i-th index in the list to store the value for the key i. With this scheme, if the keys are numbers between 0 and $H - 1$, we will allocate a list of length H, filled with `null` initially so as to represent that none of the keys have yet been associated with a value. Following that, we can support $\texttt{get}(k)$ by looking at the k-th position and $\texttt{set}(k, v)$ by setting the k-th position to v.

The obvious challenge for the general case is: *what if $H \gg n$, the number of entries we intend to store, or we have non-integer keys?*

The first concern means using this scheme, our space usage will be about H which is significantly larger than the number of entries n, a situation that we wish to avoid. The second concern is equally important; there are many applications where the keys are, for example, strings.

At any rate, now we know that

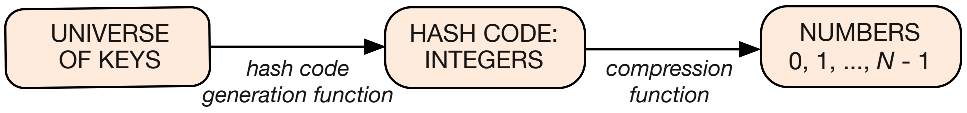
> **Proposition:** If keys are integers between 0 and $H - 1$, inclusive, it is possible to keep a dictionary in $O(H)$ space supporting all operations in $O(1)$ time, except for `new`, which takes $O(H)$ time.

3.2 Hash Functions: All Keys Are Small Int's Once We Hash Them

To overcome both challenges, we will come up with a function that takes an arbitrary key k and maps it to a small range, say $[0, N - 1]$, where N is not much larger than the total number of keys n. More specifically, a hash function h maps each key k to an integer in the range $[0, N - 1]$, where N, hopefully $N = O(n)$, is the capacity of the bucket array for a hash table.

Once we have such a hash function, we can resort to the method we just developed: use the hash value $h(k)$ as an index into our list A, instead of using the real key k directly, which may be unsuitable.

It helps to conceptually break down the hash function into two components: a component that generates hash codes and a component that compresses a hash code into the desired range. As the hash table's size may need to change over time, this conceptual view is desirable as it decouples hash code generation from the capacity of the hash table being used. They work together as illustrated in the diagram below:



More concretely, let's look at an example: given a key $k =$"déjà vu", the function that generates a hash code may produce a number denoting this key. For example, this could result in the hash code `0x10010abcdeadbeef`, a 64-bit number. This number is clearly too large to be a valid range of any reasonable-size list. Hence, we use a compression table to convert this hash code to a number in the desired range. If, say, we have a list of size $N = 1001$, then the compression function may give us 507.

Hash Codes: To generate the hash codes, we wish to design a function that takes an arbitrary key k and computes an integer—the so-called hash code—for k. Moreover, we wish to avoid collisions, i.e., minimizing the likelihood of two different keys getting assigned the same hash code.

Some good (and simple) ideas:

- **Mix High With Low:** We can treat our key as a large number; after all, it is a series of bytes. Once interpreted in this way, this simple idea mixes the higher-order bits with the lower-order ones. For example, suppose you have a 64-bit number and you want to generate a 32-bit hash code. One way to mix them is to use the XOR operator (the ^ in Java). So:

```
      +-----------+---------+
key = |   high    |    low  +  where lo = key & 0xffffffff, high = key >> 32
      +-----------+---------+
hash_code = high ^ low
```

You could repeat this process to reduce the number bits as many times as you wish. However, there's a glaring deficiency: the high bits and the low bits are symmetrical in the sense that if you were to swap them, you'll still get the same hash code, which is not ideal.

- **Polynomial Hash Code:** In particular, the above scheme is not a good option for character strings where the order of the elements is important. For example, the strings "evil" will have the same hash code as "live"—an undesirable consequence. A better hash code for this type of data should take into account the positions of the data element. We can fix this by using a polynomial: First, pick an $a \neq 0$ and to compute the hash code for a list x, where $n = \texttt{len}(x)$, we calculate:

$$x_0a^{n-1} + x_1a^{n-2} + \cdots + x_{n-2}a + x_{n-1} = (\ldots((x_0a + x_1)a + x_2)a + \ldots x_{n-1})$$

  That is, one can view this process as keeping a running sum, and as we encounter a new data item, we multiply the running run by a and add the new value in.
  According to Goodrich and Tammasia, experimental studies have shown that 33, 37, 39, and 41 are particularly good choices for a when working with English words. In a list of over 50,000 English words, they found using these numbers result in less than 7 hash-code collisions in each case.

- **Cyclic-Shift Code:** Another popular function for hashing strings is the cyclic-shift code. Instead of multiplying by a each time, it performs a cyclic shift of the running time. Here's an implementation of cyclic-shift code that shifts by 5 bits:

```
def hash_code(s):
  mask = 0xffffffff   # this is 32 1's
  h = 0
  for ch in s:
    h = (h << 5 & mask) | (h >> 27) # cyclic shift by 5
    h += ord(ch)     # add in the data
  return h
```

- **Java Built-in Hash:** For built-in data types, Java provides a hash function `obj.hashCode()`.

Compression: The hash code for a key k may not be readily useable as an index into the list we're keeping. Often, this is because the hash codes are intentionally on a large space (e.g., 64-bit or 128-bit numbers) to minimize collisions. As outlined before, we'll use a compression function to map such a hash code to an integer between 0 and $N - 1$ (inclusive).

Some good ideas:

- **Modulo N:** The simplest (yet useable) compression function is to map an integer h to $h \mod N$. A bit of number theory shows that if N is taken to be a prime number, this compression function will help nicely spread out the hash values. If N is not prime, we run a greater risk of generating collisions.
- **Linear Congruential Compression:** A more sophisticated function (with better results, of course) maps an integer h to

$$[(a \cdot h + b) \mod p] \mod N$$

  where p is a prime number larger than N, and $a \neq 0, b$ are numbers chosen arbitarily between 0 and $p - 1$ (inclusive).

3.3 Collision Resolution

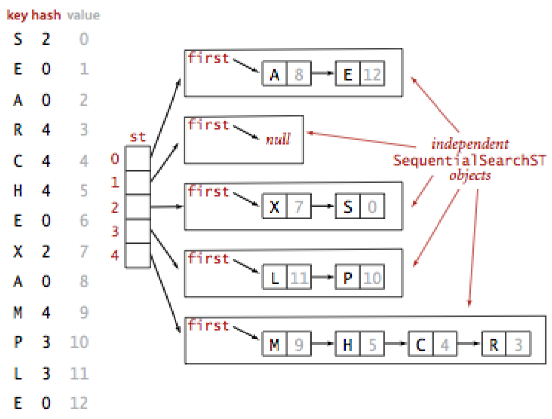
Our plan was to keep a list `table` of size about $O(n)$ and use the hash value of each key as the index into this `table` list. But this story hasn't yet found a happy ending.

What's the problem? Despite the efforts we put into crafting hash functions that will not send two keys to the same index, it is inevitable that collisions will happen. This is because we're mapping a (much) larger domain into a smaller range. The following scenario is typical:

```
Key   | Hash
-----+-----
ant   | 0
be    | 1
is    | 2
moo   | 20
am    | 1
```

The problem is both "am" and "be" hash to the same spot in our list. What're we to do?

Separate Chaining: If multiple things happen to hash to the same index, no worries! In separate chaining, we will build a list (a bucket) for that index and keep everyone who got sent here. The hope is that there aren't too many of them, so we always keep small buckets, which are easy to manage and search. (Below, figure from Sedgewick, Algorithms in Java):



Hashing with separate chaining for standard indexing client

To understand the performance implication, let's make an assumption (this is a standard assumption in the literature, which can be justified both theoretically and experimentally):

Assumption: (uniform hashing assumption). The hash function that we use uniformly distributes keys among the integer values between 0 and $N-1$. Mathematically, for any key k in the domain, $\mathbf{Pr}[h(k)=i]=1/N$.

This means on average (i.e., in expectation), the number of items falling into the same bucket is n/N. Hence, we should be good to go if we make $N=\Theta(n)$.

In conclusion:

	Unsorted List	Sorted List	Hash/Sep. Chaining
`get(k)`	$O(n)$	$O(\log n)$	$O(1)$, expected
`contains(k)`	$O(n)$	$O(\log n)$	$O(1)$, expected
`put(k, v)`	$O(n)$	$O(\log n)$ or $O(n)$ if k is new	$O(1)$, expected
`remove(k)`	$O(n)$	$O(n)$	$O(1)$, expected
Space Usage	$O(n)$	$O(n)$	$O(n)$

We are (practically) as good as an array (an `ArrayList`) but much more general.

...to be continued in Algorithms.