Hash tables

2019.10.8

In this post you'll learn what hash tables are, why you would use them, and how they're used to implement dictionaries in the most popular Python interpreter — CPython.

What are hash tables?

Hash tables are an implementation of the **dictionary** abstract data type, used for storing key-value pairs.

The main dictionary operations are:

- `set_item(key, val)`
- `get_item(key)`
- `delete_item(key)`

A dictionary is a useful data type that's implemented in most languages — as objects in JavaScript, hashes in Ruby, and dictionaries in Python, to name just a few. Often, dictionaries are implemented with hash tables.

A **hash table** stores items in an array. The index for an item is calculated from the key using a **hashing function**, which generates a fixed-size hash value from an input of arbitrary size.

Commonly, this is done in two steps:

```
hash = hash_function(key)
index = hash % array_size
```

A value can then be stored in the array:

```
arr[index] = val
```

It's possible that a key maps to the same index as another key. This is known as a **collision**. The process of handling a collision is known as **collision resolution**.

The two most common strategies for collision resolution are chaining and open addressing.

Chaining is where each item in the hash table array is a list. When an item is added to the array at an index, it's added to corresponding list.

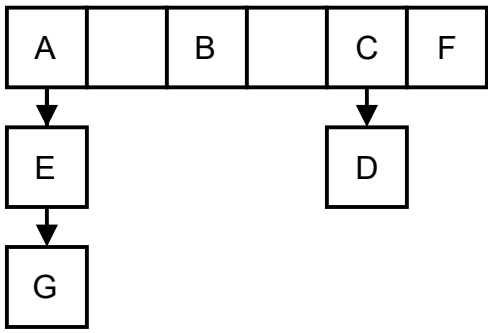


Figure 1: A hash table using chaining

Open addressing handles collisions by searching for an empty slot in the array by following a deterministic sequence. This checking is known as **probing**, and the sequence is known as a probing sequence. A simple probing sequence is to check

each slot one after the other.

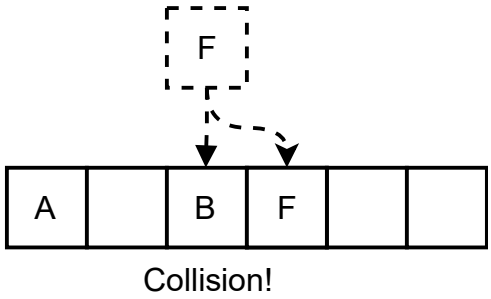


Figure 2: A hash table using open addressing

This post will look at hash tables in CPython which uses open addressing.

So what are the benefit of hash tables?

Why use hash tables?

Commonly, dictionaries are implemented with either search trees (which will be covered in future posts), or hash tables.

Hash tables are (slightly) simpler to implement than search trees and have better average-case performance.

CPython in particular uses hash tables because, compared to B-trees, hash tables give “better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler”.

Although hash tables have better average-case performance than search trees, they have much more extreme worst-case behavior.

The time complexity for hash table operations:

Operation	Average-case	Amortized Worst-case
Search	O(1)	O(n)
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)

The time complexity of red-black trees (a kind of search tree):

Operation	Average-case	Amortized Worst-case
Search	O(log n)	O(log n)
Insertion	O(log n)	O(log n)
Deletion	O(log n)	O(log n)

Because the worst-case time complexity for search tree operations is generally a consistent O(log n), search trees are often preferred in systems where large pauses for rebalancing/ reallocating introduces unacceptable latency (like the high-resolution timer code in Linux).

Although hash tables sound simple in theory, there are many nuances to implementing them. The next section looks at how hash tables are used in CPython to implement dictionaries.

Dictionaries in Python

Python is an interpreted programming language, defined in the Python reference.

The reference implementation (and most popular interpreter) is CPython.

Python provides dictionaries as a built-in data type.

You can define a dictionary in Python like so:

```
tel = {
    'alice' : 2025550143,
    'bob'   : 2025550196,
    'carol': 2025550191
}
```

You can access items from a dictionary using the subscript (`[]`) notation:

```
print(tel['alice']) # 2025550143
```

Internally, CPython implements dictionaries with hash tables.

Note: The CPython code in this post is from version 3.9.0 alpha 0.

Hash tables in CPython

The CPython dictionary hash tables store items in an array and use open addressing for conflict resolution.

Python optimizes hash tables into combined tables and split tables (which are optimized for dictionaries used to fill the `__dict__` slot of an object). For simplicity, this post will only look at combined tables.

In a combined table, the hash table has two important arrays. One is the entries array. The **entries array** stores entry objects that contain the key and value stored

in the hash table. The order of entries doesn't change as the table is resized.

The other array is the indices array that acts as the hash table. The **indices array** elements contain the index of their corresponding entry in the entries array.

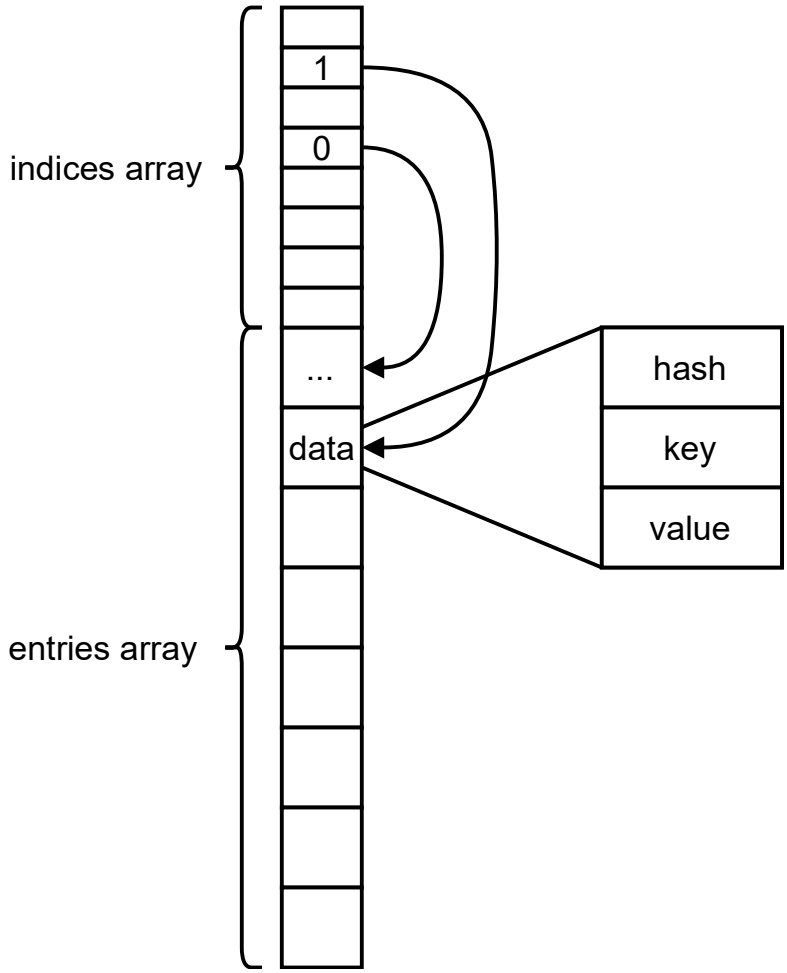


Figure 3: The indices array and entries array

CPython uses a few different structs to represent a dictionary and these arrays.

Representing dictionaries

There are three important structures used to represent dictionaries:

- The dictionary struct (`PyDictObject`), representing the entire dictionary object.

- A keys object (`PyDictKeysObject`), which contains the hash table indices array (`dk_indices`).
- An entries array, which appears directly after the corresponding `PyDictKeysObject` in memory and holds the entries referenced by `dk_indices` .

As well as other metadata, the `PyDictObject` structure contains the number of items used, and a keys object:

```
typedef struct {
    // ..
    Py_ssize_t ma_used; // Number of entries in dictionary
    // ..
    PyDictKeysObject *ma_keys; // Keys object
    // ..
} PyDictObject;
```

The keys object is important. It contains the array of indices (`dk_indices`) and metadata about the size of the hash table and the number of usable entries.

Although it isn't part of the struct, the entries array always follows a `PyDictKeysObject` . The entries array is an array of `PyDictKeyEntry` with size `dk_usable` . It's allocated at the same time a `PyDictKeysObject` is allocated.

You can see all the members of `PyDictKeysObject` :

```
struct _dictkeysobject {
    Py_ssize_t dk_refcnt;
    Py_ssize_t dk_size; // size of hash table (dk_indices)
    dict_lookup_func dk_lookup;
    Py_ssize_t dk_usable; // number of usable entries in dk_entr
    Py_ssize_t dk_nentries; // number of used entries in dk_entr
```

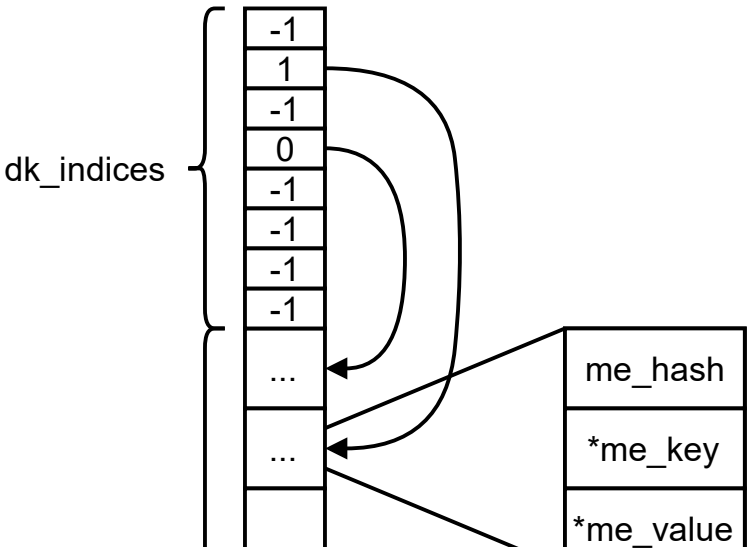
```
    char dk_indices[];  // char required to avoid strict aliasing
    // "PyDictKeyEntry dk_entries[dk_usable];" array follows:
};

// ..

typedef struct _dictkeysobject PyDictKeysObject;
```

The size of each `dk_indices` element depends on the capacity of the entries array. Since `dk_indices` holds an index value, the element size can be small if the number of entries in the entries array is low. For example, if there are 127 possible entries or less, every address value can fit in 1-byte.

The `dk_entries` array contains `PyDictKeyEntry` objects. These contain the hash, the key, and the value for an item:

```
typedef struct {
    Py_hash_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictKeyEntry;
```

Together, these structs represent a Python dictionary.



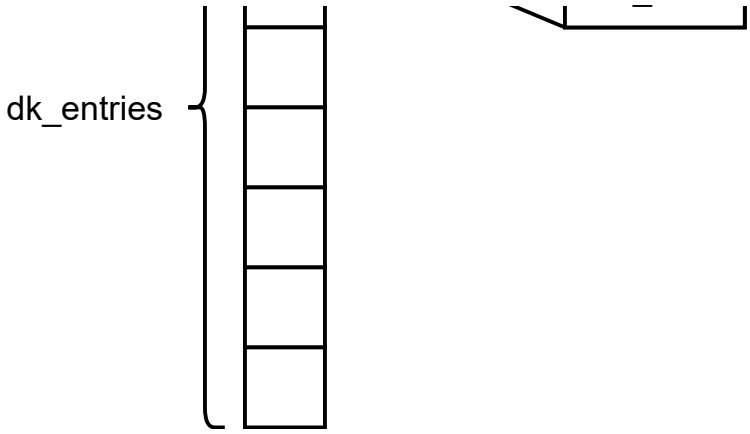


Figure 4: The `dk_indices` and `dk_entries` arrays

The next part of the hash table implementation to look it is how an index for a key is generated.

Generating an index

An index for the hash array is generated by calling a hash function with a given key.

A hash function is a one-way function that produces the same output for a given input. It's one-way in the sense that, given the hash function, it should be difficult to convert the output back to the input without trial and error.

Python supports different data types as keys for a dictionary. Each supported data type has its own associated hashing algorithm. For strings, Python uses the SipHash algorithm.

SipHash is a relatively fast hash function. On a 64-bit machine, SipHash returns a 64-bit hash. The hash is then converted into an index to be used in an array.

Traditionally, a hash is converted into an index using the modulo operator. For example, if the hash array has 40 slots, the index can be calculated with `hash %`

`40` , giving a value between 0-39. Unfortunately, the modulo operator performs division, and division is a slow operation on most CPUs.

The alternative is to use a bitmask. A **bitmask** is a pattern of bits that can be logically ANDed with another value to remove (or *mask*) unwanted bits. Any bit that is a 0 in the mask will become 0 in the result, and any bit that is 1 in the mask will remain the same after being ANDed. For example:

```
10110110
00000111 3-bit bitmask
&
00000110 Result
```

In order for the bitmask to produce a value in the full range of the array, the size of the array must be a power of 2, so the bitmask can be a full sequence of 1s.

Note that any number that is a power of 2 has a single 1 bit:

```
2  000010
4  000100
8  001000
16 010000
```

This can be converted into a bit mask of all 1s by subtracting 1 from the value:

```
(2 - 1)   1 00001
(4 - 1)   3 00011
(8 - 1)   7 00111
(16 - 1) 15 01111
```

A 2^n - 1 bit mask has the same effect as modulo 2^n. As an example, consider a

16-bit int. You have an array that contains 32 (2^5) items, with index 0-31. You can use a 5-bit mask to convert your 16-bit value into an index within the range:

```
1011 0011 1011 1001
0000 0000 0001 1111 mask

0000 0000 0001 1001
```

The following is how CPython converts a hash into an array index (where `DK_SIZE` macro gets the size of a dictionary):

```
size_t mask = (size_t)DK_SIZE(keys) - 1;

// ..
size_t i = hash & mask;
```

So that's how CPython generates the initial index `i`. If the slot at index `i` is empty, then the index of the entry can be added to the hash table. If it's not, CPython must resolve the conflict.

Conflict resolution

Conflict resolution occurs when an element already exists at an index generated from a new key. CPython uses open addressing to resolve conflicts.

The simplest implementation of open addressing is to linearly search through the array in case of a conflict. This is known as linear probing:

```
while(1) {
  if(table[++i] == EMPTY) {
    return i;
```

```
    }
}
```

Linear probing can be inefficient in CPython, because some of the CPython hash functions result in many keys mapping to the same index. If there are many collisions at the same index, linear probing results in clusters of active slots causing the linear probe to go through many iterations before finding a match.

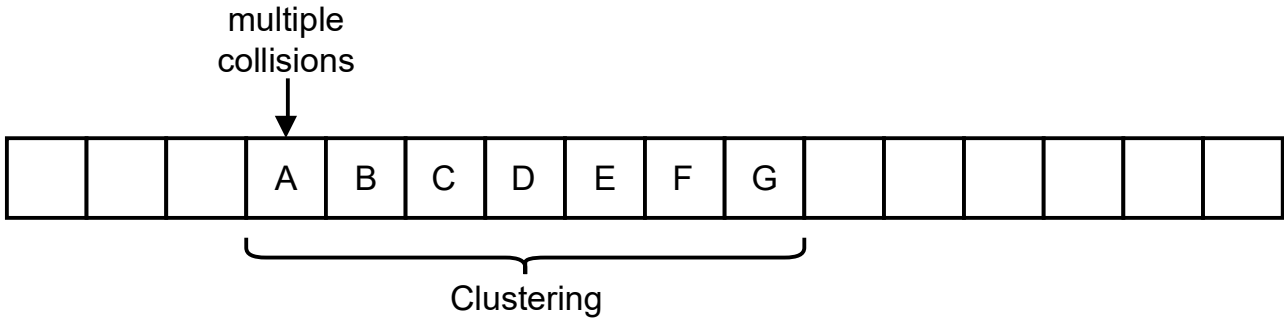


Figure 5: Multiple collisions with linear probing

One solution would be to use improved hash functions at the price of slower hashing. Instead, CPython makes the probing more random. It uses the rest of the hash to generate a new index. This is done by storing the hash in a variable named `perturb` and shifting `perturb` down 5 bits (`PERTURB_SHIFT`) each iteration. This is combined with the following calculation:

```
perturb >>= PERTURB_SHIFT;
i = mask & (i*5 + perturb + 1);
```

After a few shifts, `perturb` becomes 0, meaning just `i*5 + 1` is used. This is fine because `mask & (i*5 + 1)` produces every integer in range 0- `mask` exactly once.

The next part of the hash table to discuss is deletion.

Deleting from a dictionary

When items are deleted from a dictionary, the entry in the entries array is deleted (set to `NULL`) and the slot in the indices array is put in a dummy state by adding a negative value to the slot.

The reason items aren't removed from the indices array when they are deleted is because it would mess up probing that had previously run when the index was active.

When a dictionary is resized, any entries that were deleted since the last resizing aren't copied over to the new dictionary.

Now that you have an overview of the hash table implementation, it's time to see how dictionaries are implemented in detail.

Implementing a dictionary

This section will walk through the CPython code that implements the following snippet:

```
x = {}
x['key'] = 'a'
x['key2'] = 'b'
```

When you run CPython with an input file, CPython parses the source code and (after several stages of parsing) converts it into byte-code, which is then executed by the CPython virtual machine.

Byte-code is simply an object with associated values.

The following is the byte-code produced from the above Python snippet:

```
  1           0 BUILD_MAP                0
              2 STORE_NAME               0 (x)
              4 LOAD_CONST               0 ('a')
              6 LOAD_NAME                0 (x)
              8 LOAD_CONST               1 ('key')
             10 STORE_SUBSCR
             12 LOAD_CONST               2 ('b')
             14 LOAD_NAME                0 (x)
             16 LOAD_CONST               3 ('key2')
             18 STORE_SUBSCR
             20 LOAD_CONST               4 (None)
```

Note: You can view byte-code using the Python dis module.

The byte-code is then evaluated by CPython one opcode at a time in a large switch statement:

```
for (;;) {
    // ..
    switch (opcode) {
        case TARGET(BUILD_MAP): {
            // do something
        }
        // ..
    }
```

This post will focus on what happens when the `BUILD_MAP`, and `STORE_SUBSCR` opcodes are evaluated.

Creating a dictionary

The `BUILD_MAP` opcode initializes a map. The `BUILD_MAP` case statement

allocates a dictionary object and initializes its values.

The important function here is `new_dict()`. `new_dict()` allocates a `PyDictObject` object by calling `PyObject_GC_New`. It then sets the initial values for the `PyDictObject` object:

```
static PyObject *
new_dict(PyDictKeysObject *keys, PyObject **values)
{
    PyDictObject *mp;

    // ..

    mp = PyObject_GC_New(PyDictObject, &PyDict_Type);

    // ..

    mp->ma_keys = keys;
    mp->ma_values = values;
    mp->ma_used = 0;
    mp->ma_version_tag = DICT_NEXT_VERSION();
    ASSERT_CONSISTENT(mp);
    return (PyObject *)mp;
}
```

There's now a dict object in memory, which will be assigned to `x` when the next opcode runs.

Storing a value

The other opcode to look at is `STORE_SUBSCR`, short for store subscript (subscript notation is `[]`, e.g. `x['key'] = 'hello'`). `STORE_SUBSCR` adds a value to the map.

`STORE_SUBSCR` is a generic opcode. It can be legally called on many objects, like lists or strings.

A quick primer on Python objects. In CPython, each Python object has an associated **object type** that contains generic functions and values. For example, a length function used to calculate the object's length.

The object type function for dictionary objects that's used during `STORE_SUBSCR` execution is the dubiously-named `dict_ass_sub()` . `dict_ass_sub()` calls `PyDict_SetItem()` when a value is set using subscript notation.

`PyDict_SetItem()` is where the hash is created. CPython first checks to see if a hash has already been created for the string. If a hash hasn't been created, CPython calls `PyObject_Hash()` with the key:

```
int
PyDict_SetItem(PyObject *op, PyObject *key, PyObject *value)
{
    Py_hash_t hash;
    // ..
    if (!PyUnicode_CheckExact(key) ||
        (hash = ((PyASCIIObject *) key)->hash) == -1)
    {
        hash = PyObject_Hash(key);
        // ..
    }
    // ..
}
```

`PyObject_Hash` calls the relevant hash function for the object type to generate a hash (check the _Py_HashBytes() source code if interested). The hash is then stored on the object so it can be used in the future without running the hash function

again.

Once the hash has been generated, `PyDict_SetItem()` can continue. `PyDict_SetItem()` optimizes for an empty dictionary by calling `insert_to_emptydict()`. `insert_to_emptydict()` creates a new keys object of size `PyDict_MINSIZE` (currently 8).

```
static int
insert_to_emptydict(PyDictObject *mp, PyObject *key, Py_hash_t ha
                    PyObject *value)
{
    // ..
    PyDictKeysObject *newkeys = new_keys_object(PyDict_MINSIZE);
    // ..
}
```

When the keys object has been created, the entry can be added. This is done by calculating the index with bitmasking:

```
static int
insert_to_emptydict(PyDictObject *mp, PyObject *key, Py_hash_t ha
                    PyObject *value)
{
    // ..
    size_t hashpos = (size_t)hash & (PyDict_MINSIZE-1);
    // ..
}
```

The index is then stored in the hash table by calling `dictkeys_set_index()`:

```
static int
insert_to_emptydict(PyDictObject *mp, PyObject *key, Py_hash_t ha
                    PyObject *value)
```

```
{
    // ..
    dictkeys_set_index(mp->ma_keys, hashpos, 0);
    // ..
}
```

`dictkeys_set_index()` determines the size of the keys based on the number of entries that can be addressed by the hash table. It gets the number of items from the keys, then checks to see if the size is below a threshold.

For example, if the number of addressable items is less than 127 (0x7f), the indices only need to be 1 byte to store all possible addresses. There are other cases for 2 bytes and 4 bytes. You can see the other cases in the dictkeys_set_index() code.

```
/* write to indices. */
static inline void
dictkeys_set_index(PyDictKeysObject *keys, Py_ssize_t i, Py_ssize
{
    Py_ssize_t s = DK_SIZE(keys);

    // ..

    if (s <= 0xff) {
        int8_t *indices = (int8_t*)(keys->dk_indices);
        assert(ix <= 0x7f);
        indices[i] = (char)ix;
    }
    // ..
}
```

After `dictkeys_set_index()` has stored the index of the key entry in the `dk_indices` array, the key entry is accessed. Since this is first item in the list, the entry is known to be at index 0 in the entries array. The entry item values then get set as well as the dictionary values:

```
static int
insert_to_emptydict(PyDictObject *mp, PyObject *key, Py_hash_t ha
                    PyObject *value)
{
 // ..
    PyDictKeyEntry *ep = &DK_ENTRIES(mp->ma_keys)[0];
    dictkeys_set_index(mp->ma_keys, hashpos, 0);
    ep->me_key = key;
    ep->me_hash = hash;
    ep->me_value = value;
    mp->ma_used++;
    mp->ma_version_tag = DICT_NEXT_VERSION();
    mp->ma_keys->dk_usable--;
    mp->ma_keys->dk_nentries++;
    return 0;
}
```

Success! An object with the value of "a" has been added to the dictionary.

The more interesting case is when an item is added to a non-empty dictionary. During this operation the hash table must potentially resize and resolve collisions. This happens in `PyDict_SetItem()`. In the case of a non-empty dictionary, `insertdict()` is called with the dictionary structure, the key, the hash, and the value.

`insertdict()` does several things:

- Handles collisions.
- Inserts item.
- Checks if the table needs resizing (resizing if needed).

First `insertdict()` calls the dictionary's `dk_lookup()`, which is assigned depending on the key types used in the dictionary.

```
static int
insertdict(PyDictObject *mp, PyObject *key, Py_hash_t hash, PyOb
{
    PyObject *old_value;
    // ..

    Py_ssize_t ix = mp->ma_keys->dk_lookup(mp, key, hash, &old_v

    // ..
}
```

Initially `dk_lookup()` is set to an optimized lookup function that doesn't check for dummy values. Once an item is deleted, it's updated to `lookdict_unicode()`.

`lookdict_unicode()` uses the key hash and mask to generate an index for the item (`i`). `dictkeys_get_index()` is then called to access the element at index `i` (`ix`). If an item already exists at the index, it enters conflict resolution.

First, consider the happy path. In this case, `ix` is empty, and `lookdict_unicode()` can return:

```
static Py_ssize_t _Py_HOT_FUNCTION
lookdict_unicode(PyDictObject *mp, PyObject *key,
                 Py_hash_t hash, PyObject **value_addr)
{
   // ..

    PyDictKeyEntry *ep0 = DK_ENTRIES(mp->ma_keys);
    size_t mask = DK_MASK(mp->ma_keys);
    size_t perturb = (size_t)hash;
    size_t i = (size_t)hash & mask;

    for (;;) {
        Py_ssize_t ix = dictkeys_get_index(mp->ma_keys, i);
        if (ix == DKIX_EMPTY) {
```

```
            *value_addr = NULL;
            return DKIX_EMPTY;
        }
       //
    }
    // ..
```

Note: `dictkeys_get_index()` *performs logic to get the correct memory address depending on the size of the table, as seen earlier in this post.*

If `ix` isn't empty, there are two possibilities:

1. The key already exists in the dictionary.
2. Another key exists in the index (potentially a dummy).

The first option is that the key used to generate the index already exists in the array. In that case, `lookdict_unicode()` checks the entry using the index and compares the entry key with the key that's being added. If they are the same, the entry can be overwritten with the new value:

```
static Py_ssize_t _Py_HOT_FUNCTION
lookdict_unicode(PyDictObject *mp, PyObject *key,
                 Py_hash_t hash, PyObject **value_addr)
{
   // ..
    for (;;) {
        // ..
        if (ix >= 0) {
            PyDictKeyEntry *ep = &ep0[ix];
            assert(ep->me_key != NULL);
            assert(PyUnicode_CheckExact(ep->me_key));
            if (ep->me_key == key ||
                    (ep->me_hash == hash && unicode_eq(ep->me_key
                *value_addr = ep->me_value;
```

```
                return ix;
            }
        }
        perturb >>= PERTURB_SHIFT;
        i = mask & (i*5 + perturb + 1);
    }
    // ..
}
```

If not, then a new index must be generated. This is done in the way described earlier, by shifting the hash (`perturb`), multiplying by 5 and adding 1:

```
static Py_ssize_t _Py_HOT_FUNCTION
lookdict_unicode(PyDictObject *mp, PyObject *key,
                 Py_hash_t hash, PyObject **value_addr)
{
   // ..
    for (;;) {
        // ..
        if (ix >= 0) {
            // ..
        }
        perturb >>= PERTURB_SHIFT;
        i = mask & (i*5 + perturb + 1);
    }
    // ..
}
```

This probing sequence continues until an available index is found. This is guaranteed because there are always empty spaces available, and the recurrence `(i*5 + 1) mod 2**i` will generate each int in `range(0 - 2**i)` .

Resizing the dictionary

Python dictionaries are dynamic. You can continue to add new entries and

CPython will increase the dictionary in order to accommodate them.

To handle this, Python resizes dictionary hash tables as needed when new items are added. The hash table is resized when $^2/_3$ of the space of the indices array (`dk_indicies`) is reached (this ratio is known as the **load factor**).

The reason only $^2/_3$ of the hash table is ever used is to keep the array sparse, and therefore to reduce collisions.

Resizing involves:

- Calculating the new table size and the usable fraction of the size.
- Allocating a new keys object, which contains the hash table (`dk_indices`) and the entries array.
- Adding all entries to the new entries array. If any entries have been deleted since the last resize, they won't be copied over.
- Rebuilding the hash table indices array (by calculating new indices for each of the items).

The current growth rate is 3, so the new size of a hash table resized during insertion is the number of used entries multiplied by 3.

The size has to be a power of 2, so the minimum power of 2 is calculated during a resize. You can see this in `dictresize()` , which is called to resize a dictionary. Since the hash table must be a power of 2, `dictresize()` calculates the next lowest power of 2 that is greater than `minsize` by left shifting a single bit of a signed integer (initially with value 8) until the integer is larger than the `minsize` :

```
static int
dictresize(PyDictObject *mp, Py_ssize_t minsize)
```

```
{
    Py_ssize_t newsize, numentries;
    // ..

    /* Find the smallest table size > minused. */
    for (newsize = PyDict_MINSIZE;
         newsize < minsize && newsize > 0;
         newsize <<= 1)
        ;
    // ..
}
```

`dictresize()` gets a reference to the old keys object (`mp->ma_keys`). It allocates a new keys object table (`new_keys_object`). Then copies over the old entries into the new entries location. It also copies over other values like function pointers, and performs housekeeping, including freeing the old keys object.

If entries have been deleted since the last resize, `mp->ma_used` will be less than `oldkeys->dk_nentries` . In this case, the deleted keys entry value will be NULL and they won't be to the `newentries` :

```
static int
dictresize(PyDictObject *mp, Py_ssize_t minsize)
{
    // ..
    numentries = mp->ma_used;
    // ..
        if (oldkeys->dk_nentries == numentries) {
            memcpy(newentries, oldentries, numentries * sizeof(Py
        }
        else {
            PyDictKeyEntry *ep = oldentries;
            for (Py_ssize_t i = 0; i < numentries; i++) {
                while (ep->me_value == NULL)
                    ep++;
```

```
                    newentries[i] = *ep++;
                }
            }
        // ..
        return 0;
    }
```

The final part after resizing is to rebuild the indices array. This is done by looping over each entry and recalculating its position in the array using the updated bitmask and the probing sequence seen earlier.

That's an overview of the CPython dictionary implementation. If you're interested, I encourage you to read the source code for interesting edge cases and optimizations!

Conclusion

Hash tables are a popular way to implement the dictionary data type. Theoretically they are intuitive to understand. However, real-life implementations of hash tables can be complex.

Future posts in this series will focus on different search trees that can be used to implement the dictionary data type as an alternative to hash tables.

吴兆洋 • 2 years ago

Great post.
Pretty clear and easy to read!

• Reply • Share ›

Ingo Kognito • 2 years ago

Great summary about Python dictionaries, I really enjoyed reading.

A few (minor) remarks:

* In the snippet about linear probing, shouldn't it be `++i` in order to yield the correct index (i.e. `if(table[++i] == EMPTY]) return i`.
* In the text following the random probing snippet, where you describe `perturb` might reach zero at some point, the line `mask & i*5 + 1` misses the parentheses, i.e. it should be `mask & (i*5 + 1)`.
* In the text right before the "Resizing the dictionary" section, you write that "This probing sequence continues until an available index is found, which is guaranteed because there are always empty spaces." However another important requirement is that, even if `perturb` reaches zero, the update `i = mask & (i*5 + 1)` doesn't enter a cycle of length smaller than `mask+1` (for `mask = pow(2, n) - 1` and `i <= mask`). This isn't obvious and if the update rule used another multiplication factor, e.g. `4`, then the requirement would not be met.

• Reply • Share ›

Edd Mod ➚ Ingo Kognito • 2 years ago • edited

Thanks I'm glad you enjoyed it. I've updated the post to address your comments.

1 • Reply • Share ›