Basic structure — PHP Internals Book

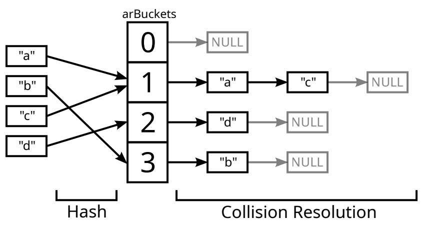
Basic concepts¶

Arrays in C are just regions of memory that can be accessed by offset. This implies that keys have to be integers and need to be continuous. For example, if you have the keys 0, 1 and 2, then the next key has to be 3 and can't be 214678462. PHP arrays are very different: They support both string keys and non-continuous integer keys and even allow mixing both.

To implement such a structure in C there are two approaches: The first is using a binary search tree, where both lookup and insert have complexity `O(log n)` (where `n` is the number of elements in the table). The second is a hashtable, which has an average lookup/insert complexity of `O(1)`, i.e. elements can be inserted and retrieved in constant time. As such hashtables are preferable in most cases and are also the technique that PHP uses.

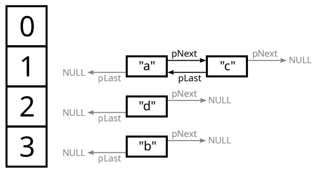
The idea behind a hashtable is very simple: A complex key value (like a string) is converted into an integer using a hash function. This integer can then be used as an offset into a normal C array. The issue is that the number of integers (`2^32` or `2^64`) is much smaller than the number of strings (of which there are infinitely many). As such the hash function will have collisions, i.e. cases where two strings have the same hash value.

As such some kind of collision resolution has to take place. There are basically two solutions to this problem, the first being *open addressing* (which is not covered here). The second one is *chaining* and is employed by PHP. This method simply stores all elements having the same hash in a linked list. When a key is looked up PHP will calculate the hash and then go through the linked list of "possible" values until it finds the matching entry. Here is an illustration of chaining collision resolution:

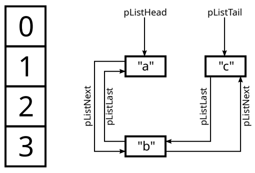


The elements of the linked list are called `Buckets` and the C array containing the heads of the linked lists is called `arBuckets`.

Consider how you would delete an element from such a structure: Say you have a pointer to the bucket of `"c"` and want to remove it. To do this you'd have to set the pointer coming from `"a"` to `NULL`. Thus you need to retrieve the bucket of `"a"` which you can do either by traversing the linked list for the hash value or by additionally storing pointers in the reverse direction. The latter is what PHP does: Every bucket contains both a pointer to the next bucket (`pNext`) and the previous bucket (`pLast`). This is illustrated in the following graphic:



Furthermore PHP hashtables are *ordered*: If you traverse an array you'll get the elements in same order in which you inserted them. To support this the buckets have to be part of another linked list which specifies the order. This is once again a doubly linked list, for the same reasons as outlined above (and to support traversation in reverse order). The forward pointers are stored in `pListNext`, the backward pointers in `pListLast`. Additionally the hashtable structure has a pointer to the start of the list (`pListHead`) and the end of the list (`pListLast`). Here is an example of how this linked list could look like for the elements `"a"`, `"b"`, `"c"` (in that order):



The HashTable and Bucket structures¶

To implement hashtables PHP uses two structures, which can be found in the `zend_hash.h` file. We'll first have a look at the `Bucket` struct:

```
typedef struct bucket {
    ulong h;
    uint nKeyLength;
    void *pData;
    void *pDataPtr;
    struct bucket *pListNext;
    struct bucket *pListLast;
    struct bucket *pNext;
    struct bucket *pLast;
    char *arKey;
} Bucket;
```

You already know what the `pNext`, `pLast`, `pListNext` and `pListLast` pointers are for. Let's quickly go through the remaining members:

`h` is the hash of the key. If the key is an integer, then `h` will be that integer (for integers the hash function doesn't do anything) and `nKeyLength` will be 0. For string keys `h` will be the result of `zend_hash_func()`, `arKey` will hold the string and `nKeyLength` its length.

`pData` is a pointer to the stored value. The stored value will not be the same as the one passed to the insertion function, rather it will be a copy of it (which is allocated separately from the bucket). As this would be very inefficient when the stored value is a pointer PHP employs a small trick: Instead of storing the pointer in a separate allocation it is put into the `pDataPtr` member. `pData` then points to that member (`pData = &pDataPtr`).

Let's have a look at the main `HashTable` struct now:

```
typedef struct _hashtable {
    uint nTableSize;
    uint nTableMask;
    uint nNumOfElements;
    ulong nNextFreeElement;
    Bucket *pInternalPointer;
    Bucket *pListHead;
    Bucket *pListTail;
    Bucket **arBuckets;
    dtor_func_t pDestructor;
    zend_bool persistent;
    unsigned char nApplyCount;
    zend_bool bApplyProtection;
#if ZEND_DEBUG
    int inconsistent;
#endif
} HashTable;
```

You already know that `arBuckets` is the C array that contains the linked bucket lists and is indexed by the hash of the key. As PHP arrays don't have a fixed size `arBuckets` has to be dynamically resized when the number of elements in the table (`nNumOfElements`) surpasses the current size of the `arBuckets` allocation (`nTableSize`). We could of course store more than `nTableSize` elements in the hashtable, but this would increase the number of collisions and thus degrade performance.

`nTableSize` is always a power of two, so if you have 12 elements in a hashtable the actual table size will be 16. Note though that while the `arBuckets` array automatically grows, it will *not* shrink when you remove elements. If you first insert 1000000 elements into the hashtable and then remove all of them again the `nTableSize` will still be 1048576.

The result of the hash function is a `ulong`, but the `nTableSize` will usually be a lot smaller than that. Thus the hash can not be directly used to index into the `arBuckets` array. Instead `nIndex = h % nTableSize` is used. As the table size is always a power of two this expression is equivalent to `nIndex = h & (nTableSize - 1)`. To see why let's see how `nTableSize - 1` changes the value:

```
nTableSize     = 128 = 0b00000000.00000000.00000000.10000000
nTableSize - 1 = 127 = 0b00000000.00000000.00000000.01111111
```

`nTableSize - 1` has all bits below the table size set. Thus doing `h & (nTableSize - 1)` will only keep the bits of the hash that are lower than `nTableSize`, which is the same thing `h % nTableSize` does.

The value `nTableSize - 1` is called the table mask and stored in the `nTableMask` member. Using a masking operation instead of a modulus is just a performance optimization.

The `nNextFreeElement` member specifies the next integer key that will be used when you insert an element using `$array[] = $value`. It will be one larger than the largest integer key that was ever used in this hashtable.

You already know the role of the `pListHead` and `pListTail` pointers (they are the head/tail of the doubly linked list specifying the order). The `pInternalPointer` is used for iteration and points to the "current" bucket.

When an item is deleted from the hashtable a destruction function can be called for it, which is stored in the `pDestructor` member. For example, if you are storing `zval *` items in the hashtable, you will probably want `zval_ptr_dtor` to be called when an element is removed.

The `persistent` flag specified whether the buckets (and their values) should use persistent allocation. For most use cases this will be `0` as the hashtable is not supposed to live longer than one request. The `bApplyProtection` flag specifies whether the hashtable should use recursion protection (defaults to 1). Recursion protection will throw an error if the recursion depth (stored in `nApplyCount`) reaches a certain level. The protection is used for hashtable comparisons and for the `zend_hash_apply` functions.

The last member `inconsistent` is only used in debug builds and stores information on the current state of the hashtable. This is used to throw errors for some incorrect usages of the hashtable, e.g. if you access a hashtable that is in the process of being destroyed.