Maps and Dictionaries

Storing and searching (key, value) pairs. This approach won't have a good guarantee on complexity, but on average will be very fast.

Dictionaries and Maps

A dictionary or map is a storage and look-up structure maintaining (key, value) pairs.

Formally, each value has a unique key. Specifying the key allows us to retrieve the value from the structure.

Python provides the `dict` data type. We will use the word *map* for the general concept, to avoid confusion with Python's implementation.

Map ADT

- `getitem(key)` – return the value with given key, or None
- `setitem(key, value)` – assign value to element with key; create new element if needed
- `contains(key)` – return True is the map has an element with key
- `delitem(key)` – remove element with key and return its value or None
- `length()` – return the number of elements in the map
- `is_empty()` – return True if there are no items in the map

Unsorted List Implementation

Append new elements to the end of the list (with Python resizing).

- `getitem(k)` – search the unordered list to find element with key k
- `setitem(k, v)` – search the list to find the key, change the value, or append new element if key not found
- `delitem(k)` – search the unordered list to find the key and pop the element

These implementations are all O(n).

Implementation Comparison

Implementation	`getitem`	`contains()`	`setitem()`	`delitem()`	build full map
unsorted list	O(n)	O(n)	O(n)	O(n)	O(n)
sorted list	O(log(n))	O(log(n))	O(n)	O(n)	
AVL tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	

Map Search Time

The operation we will want to do most often is to find an item (which is needed to read its value or to update it).

AVL trees offer O(log(n)) for searching. Can we do better?

In a standard Python list, we can access an item in O(1) if we know the item is in the list and we know the index of the item. But in a standard map, we don't have this info.

Example: Book of Modules

In the book of modules example, we could strip the letters off the front of the key, and use the numbers as the index in a list. Position 1110 would contain the title "Systems Organisation 1".

This gives us quick access but means there's a lot of space wasted, as a lot of 4-digit numbers aren't used for module codes.

So this doesn't work in general, but can we adapt the idea?

- Use the structure of the key to identify the location.
- Use a more compact list, and use the size of the list to determine the location.

Hash Tables

Maintain a list of known size, with explicit None items.

To add a new item:

1. Compute an integer corresponding to the key.
2. Compute an index in the list based on that integer and the list size.
3. Store the item in the list at that index.

Line 1 requires a function from the set of keys to a subset of the integers. Once we can do lines 1 and 2 efficiently, the whole thing can be O(1).

Python's Hash Function

Python has a built-in function `hash()` which will convert any hashable object into an integer.

Hash values must remain the same during an object's lifetime, and two objects which evaluate as equal must have identical hash values. This means you have to use objects that don't change value (immutable objects).

All built-in immutable objects (strings, integers, floats, booleans, tuples) are hashable. If you create a string called s, the underlying string object can't be changed, you can only change what is referenced by the variable name s.

User-defined classes can include a `__hash__` method, which should be defined on some immutable attribute(s). By default, Python will use the id or memory address of the object.

Basic Compression

Once we have computed a hash value, we need to decide on a location within our list.

Suppose our list has N cells – our location must be in range [0, N-1].

If our key has hash value `hash(k) = h`, we set `location(k) = h mod(N)`.

Determining this location is O(1), for a given list size N.

Example

List size of 10:

```
item = Element('CS2515', 'Algorithms and DataStructures 1')

hash('CS2515') evaluates to 1860694193
hash('CS2515') % 10 evaluates to 3
hash('CS1112') % 10 evaluates to 1
```

So those elements go into positions 3 and 1.

Note: Python's `hash()` function uses a random seed, and so will give different values for each session (for security – there's a way of overloading people's hashing functions if you can predict them). There's a way to obtain consistent hashing if you want [note to self: look

this up].

But we only have 10 cells in our list. Some keys will be directed to the same location, and so if we try to insert them, they will collide with other entries.

Chaining

Each cell in the list will maintain its own list of values (this is known as a bucket array).

Search Complexity in Bucket Arrays

To find an item in a bucket array, we have to compute the hash, then the compressed location, and then search the bucket.

If we're unlucky, all of our items could end up being sent to the same bucket, and so search is O(n).

If we have n items to insert into a bucket array of size N, then we expect n/N items in each bucket (on average), and the expected complexity is O(n/N).

But we were aiming for O(1).

Dynamic Bucket Arrays

So we aim to keep n/N below a certain value, by increasing N (the size of the list), e.g. by doubling it, whenever we reach that threshold value.

Then we can guarantee O(1).

But, when we double the list, we have to rehash and recompress the list. As with list resizing, any particular add may not be O(1), but on average it will be O(1).

Implementations

Version 1 (fixed size list)

[get code from lecture slides]

Version 2 (dynamic size list)

- If I've just added something and the size is greater than a certain function of the length of the list, then we resize by our chosen factor.

[get code from lecture slides]

Hash Function Properties

We want to avoid collisions:

- Distinct items should get distinct hashes, so we should use as much of the content as possible.
- Some collisions are always inevitable.

Simple Hash Function

For integers, use an exclusive-or or the first half with the second half.

For strings, use a polynomial sum of characters with a fixed constant `c`.

Don't allow floats as keys (as you have to account for small errors, make sure that they're registered as the same number).

Compression

The main aim is to distribute the hast values as evenly as possible across the range.

We used `h % N`, where N is the size of the list. This works, but if a sequence of values has a pattern related to N, many will end up colliding.

If we have a sequence of numbers going up in 4, mod 12 will give us a lot of grouping, as 4 is a factor of 12. Using mod 13 instead will give us much better spread.

In general, it's good to try to choose prime numbers for N.

However, we don't want to spend time calculating prime numbers.

Multiply, Add, and Divide

```
MAD_hash(key) = ((a * key + b) % p) % N
```

where p is a prime number, [check]

It can be shown using group theory that this method gives better compression.

SHA-1

SHA-1 is considered insecure against well-funded opponents, because it's possible to cause collisions with specific input (if you have enough computing power).

Collision Handling

The chaining method and the bucket array requires the use of additional data structures, which take up extra space.

Instead, we can store everything within the top list.

If we get a collision, let's go right until we find an empty cell, and put it there.

When searching for items, we check the item at the position given by the hash, and move right if it isn't what we're looking for. If we hit an empty cell, we know that the element isn't here.

Open Addressing: Linear Probing

To set an element, we search right from the hash value until:

- we find the key, and we update the value
- we find an empty or available cell, and we add the new element

To delete an element, we search right from the hash value until:

- we reach an empty cell, and do nothing (element not in map)
- we find the key, and we delete the element

However, deleting elements can lead to us not finding elements because we encounter spaces.

To overcome this, whenever we take something out, we put a special 'available' marker in. This won't be registered as a stop when searching, but will take new elements.

Notes

All values must get the same opportunity to be added. Search must wrap around the end of the list, which we accomplish with modular arithmetic.

- `hash(k)`, then `(hash(k) + 1) % N`, ...

The load factor (`n/N`) must never get above 1, because then there are no empty spaces left. Standard practice is to grow the list whenever `n/N = 0.5`. Rather than doubling the size, we grown to `2N - 1` or `2N + 1` because that gives us more chance of the new size being a prime number.

Other Probing Schemes

Linear probing tends to produce long chains of occupied cells, and so searching tends to O(n).

- **Quadrating probing** tries the cell `(hash(k) + i**2) % N` for each search step `i`.
- **Double hashing** tries the cell `(hash(k) + i*h'(k)) % N` for each search step `i`, where `h'()` is another hash function.
- **Pseudo-random probing** tries the cell `(hash(k) + f(i)) % N` for each search step `i`, where `f(i)` is determined by a pseudo-random number generator.

The last option is what Python uses for its dictionary implementation.

Implementation Comparison

Implementation	**`getitem`**	**`contains()`**	**`setitem()`**	**`delitem()`**	**build full map**
unsorted list	O(n)	O(n)	O(n)	O(n)	O(n) | O(n squared)
sorted list	O(log(n))	O(log(n))	O(n)	O(n)	O(n squared)
AVL tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(nlogn)
Hash Table	O(n)	O(n)	O(n)	O(n)	O(n squared)
Expected:	O(1)				