Hashing and Dictionaries

15-110 – Monday 10/14

Learning Goals

Use **data structures** to represent data in different formats for different purposes

- Map **different types of data** to different data structures
- Use **dictionaries** to store and access data that maps keys to values

Analyze and improve the **efficiency** of different programs

- Discuss how **searching** can be implemented efficiently in a variety of data structures
- Understand how **hash functions** make it possible to search for items in constant time

Improving Search

We've now discussed linear search (which runs in O(n)), and binary search (which runs in O(logn)).

We use search all the time, so we want to search as quickly as possible. **Can we search for an item in O(1) time?**

We can't *always* search for things in constant time, but there are certain circumstances where we can...

Search in Real Life – Post boxes

Consider how you receive mail. You know that all of your mail is sent to the post boxes at the lower level of the UC.

Do you have to check every box to find your mail? No- you just check the one assigned to you!

This is possible because your mail has an **address** on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes.

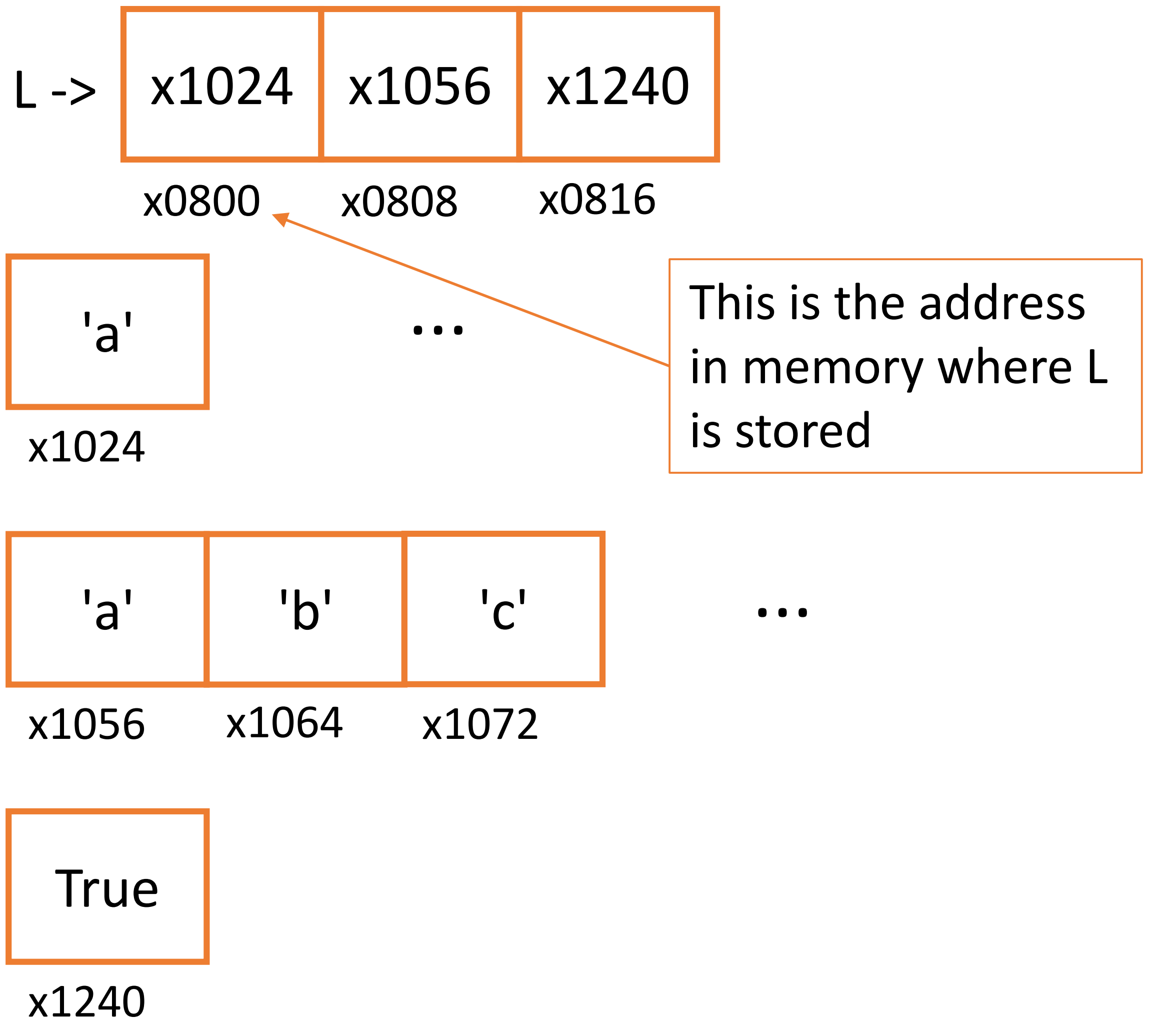
Picking up your mail is a O(1) operation!

Search in Programming – List Indexes

We can't search a list for an item in constant time, but we **can** look up an item based on an index in constant time. How does this work?

Python stores the list in memory as a series of **adjacent bytes**. Each byte holds a reference to a memory address; the value at that memory address is the item at that position in the list.

So if L = ["a", "abc", True]...



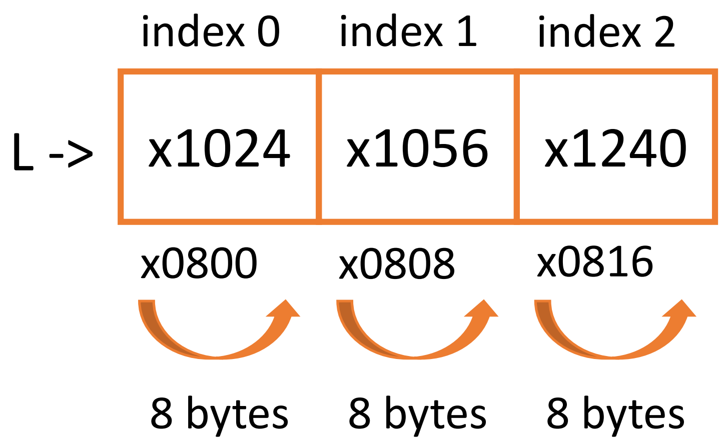
Search in Programming – List Indexes

Because each memory address takes the same amount of space (one byte, or 8 bits), we can calculate the exact starting location of an index's byte based on the first address where L is stored. We do this with the formula:

```
start + 8 * index
```

For example, in the list to the right, to access L[2], we compute `x0800 + 8 * 2 = x0816`. When we evaluate the byte at `x0816`, that gives us the address `x1240`, so we can retrieve the value there.

Given a memory address, we can get the value from that address in constant time. So looking up an index in a list is O(1)!



Combine the Ideas

To implement constant-time search, we want to combine the ideas of post boxes and list index lookup. Specifically, we want to be able to determine **which index a value is stored in based on the value itself**.

If we can calculate the index based on the value, we can retrieve the value in constant time.

Hashing

Hash Functions Map Values to Integers

In order to determine which list index should be used based on the value itself, we'll need to **map values to indexes**, i.e, non-negative integers.

We call a function that maps values to integers this way a **hash function**. This function must follow a few rules:

- Given a specific value x, hash() must **always** return the same output i
- Given two different values x and y, hash() should **usually** return two different outputs, i and j

Activity: Design a Hash Function

How would you design a hash function for strings? Talk to a partner to come up with your algorithm.

Remember to follow the rules:

- Given a specific value x, hash() must **always** return the same output i
- Given two different values x and y, hash() should **usually** return two different outputs, i and j

Built-in Hash Function

We don't need to write our own hash functions most of the time- Python already has one!

```
x = "abc"
print(hash(x))
```

hash() works on integers, floats, Booleans, strings, and some other types as well.

Hashtables Organize Values

Now that we have a hash function, we can use it to organize values in a specialty data structure.

A **hashtable** is a list with a set number of indexes. When we place a value in the list, we put it into an index **based on its hash value**, instead of placing it at the end of the list.

For example, we can set up a new hashtable with 20 indexes:

```
h = [None] * 20
```

Let's add strings to the hashtable, using a basic hash function which uses the first letter of the string and ord().

```
def hash(s):
    return ord(s[0]) - ord('a')
```

Adding Values to a Hashtable

First, let's add "book" to h. `hash("book")` is 1, so we'll set the value in that index.

```
i = hash("book") # 1
h[i] = "book"
```

Next, let's add "zebra". hash("zebra") is 25, which is outside the range of our table. How do we assign it?

```
j = hash("zebra") % 20 # 5
h[j] = "zebra"
```

Use x % tableSize to map integers larger than the size of the table to an index.

Dealing with Collisions

When you add lots of values to a hashtable, they might start **colliding** if two values are assigned to the same index.

This is okay! Just put the collided values in a list, and assign that list to the index.

If your table size is reasonably big and the indexes are reasonably spread out, there will only be a **constant** number of values in the list.

```
i = hash("cmu")%20 # 2
j = hash("college")%20 # 2

h[2] = ["cmu", "college"]
```

Searching a Hashtable is O(1)!

Finally, we get to the real task- searching a hashtable to see if it holds a value.

To search for a value, you just run the hash function on it, then mod the result by the table size. **The index produced is the only index you need to check!**

If the value is in the table, it will be at that index. If it isn't, it won't be there, and it won't be anywhere else either.

Because we only need to check one index, this is O(1)!

```
# Search for "book"
i = hash("book")%20 # 2
print(h[i] == "book") # True

# Search for "stella"
j = hash("stella")%20 # 18
print(h[j] == "stella") # False
```

Activity: compute the hashed index

Say we have a really simple hash function that maps floats to indexes by hashing them to the value in the 1s place of the digit. For example, 42.5 would hash to 2.

We want to place the number 17.46 in a four-bucket hash table.

Which bucket should it go into- 0, 1, 2, or 3? Enter your answer on Piazza!

Caveat: Don't hash mutable values!

Hash functions feel a bit like magic, and they are very handy to use! However, they aren't good for everything.

For example, what happens if you try to put a list L in a hashtable? This might seem fine at first, but it will become a problem if you change L before searching, as is shown on the right.

If the values in L change, the hashed value may change as well. Then it won't be stored in the correct index anymore!

Because of this, **we don't put mutable values into hashtables**. In fact, if you try to run the built-in hash() on a list, it will crash.

```
h = [None] * 10

L = [ 1, 2, 3, 4 ]
i = hash(L) % 10
h[i] = L

L.append(5)
j = hash(L) % 10 # uh oh, j != i
print(h[j] == L) # probably False!
```

Dictionaries

More Uses of Hash Functions

Now that we've demonstrated how hash functions work, we can use them to store data in new ways.

For example- our current hashtable is not a direct replication of the post box system we discussed earlier. Could we implement a post-box-like system, where we map addresses to letters?

Dictionaries Map Keys to Values

To implement a post box system, we'd want to use a **dictionary**, or **hashmap**. Dictionaries map keys (which are hashed items) to values (which can be anything!).

We use dictionary-like data in the real world all the time! Post boxes mapping addresses to mail are one example; phonebooks (which map names to phone numbers) are another.

Dictionaries in Python

Dictionaries are already implemented for us by Python.

```
d = { } # makes a new dictionary
d[key] = value # adds a new key-value pair
del d[key] # removes a key-value pair
```

The syntax for dictionaries is similar to lists; the main difference is that it uses keys to index, instead of indexes. And keys can be any immutable data type!

Searching a dictionary

We can directly search a dictionary for a key by using the in operator.

Because the keys are **hashed**, this search takes O(1) time!

We can't do constant-time lookups of values, though; you'd need to check every key to see if it has the sought value.

```
d = { }
d["s"] = 3
d["k"] = 7

print("s" in d) # True
print("m" in d) # False
```

Looping over Dictionaries

How do we see how many key-value pairs are in a dictionary? Just use `len(d)`!

We can get a list of just the keys with `d.keys()`, and a list of just the values with `d.values()`.

To iterate over the keys of the dictionary, use a for-each loop. You can't use a for-range loop, because you can't index by number.

```
d = { }
d["s"] = 3
d["k"] = 7
d["a"] = 0

for key in d:
    print("key: ", key)
    print("val: ", d[key])
```

Example – Storing Information

One common use of dictionaries is to **store information** about a list of values.

For example- given a list of numbers, how many times does each digit appear as the 1s digit of a number?

```
def countOnesDigit(L):
    d = { }
    for item in L:
        digit = item % 10
        if digit not in d:
            d[digit] = 0
        d[digit] = d[digit] + 1
    return d
```

Example – Finding Most Common Values

We also use dictionaries to find the most common value of an iterable, by mapping values to counts.

For example, given the dictionary returned by the previous function, which digit occurs the most often?

```
def mostCommonDigit(d):
    bestVal = None
    bestCount = 0
    for digit in d:
        if d[digit] > bestCount:
            bestVal = digit
            bestCount = d[digit]
    return bestVal
```

Learning Goals

Use **data structures** to represent data in different formats for different purposes

- Map **different types of data** to different data structures
- Use **dictionaries** to store and access data that maps keys to values

Analyze and improve the **efficiency** of different programs

- Discuss how **searching** can be implemented efficiently in a variety of data structures
- Understand how **hash functions** make it possible to search for items in constant time