Quiz2 & Activity: Designing Super-Fast Search

15-110 – Wednesday 10/06

Quiz2

Announcements

- Check3 grades released; lots of confusion around binary search.
  - When selecting the middle element, if there are an even number of items, pick the **larger item of the pair** (index `len(lst)//2`)
  - When do you stop recursing? When the **middle element is the target**, or when the **list is empty**.

Learning Objectives

- Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**

Activity: Designing Fast Search

Last week we talked about two search algorithms: linear search and binary search. Binary search is faster than linear search, but can we do even better?

We can often increase the efficiency of an algorithm by **thinking about the problem in a different way**. Try using a different data structure or an entirely different algorithmic approach to solve the problem.

Our question today is: can we design the **fastest possible** search algorithm?

Optimizing Search: Constraints

Search in Real Life – Post Boxes

Consider how you receive mail. 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- just check the one assigned to you.

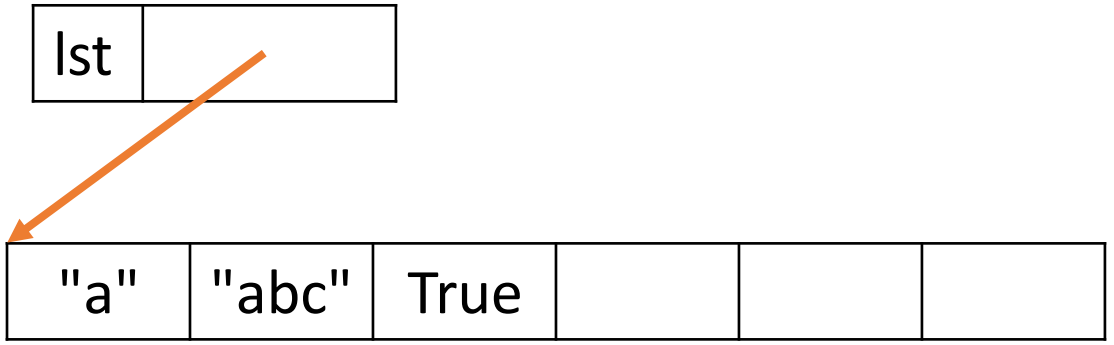
This is possible because your mail has an **address** on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes.

Picking up your mail is a **fast** operation!

Search in Programming – List Indexes

We can't search a list for an item that quickly, but we **can** look up an item based on an index with just a few steps.

Reminder: Python stores lists in memory as a series of **adjacent parts**. Each part holds a single value in the list, and all these parts use the **same amount of space**.



Example:

```
lst = ["a", "abc", True]
```

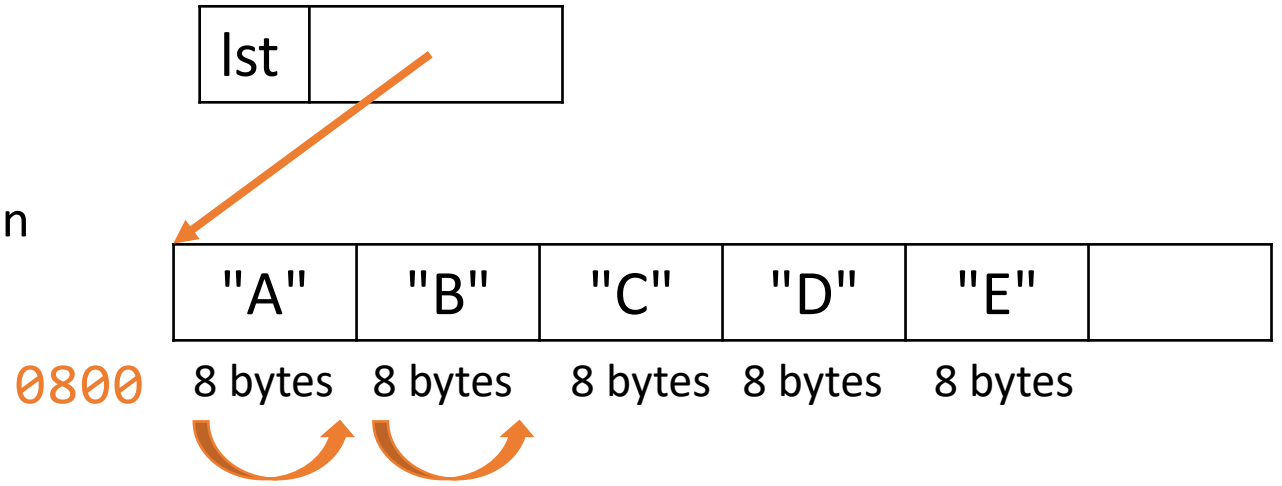
Search in Programming – List Indexes

We can calculate the exact starting location of a list index's memory address based on the first address where `lst` is stored. If the size of a part is N, we can find an index's address with the formula:

```
start + N * index
```

Example: in the list to the right, each part is 8 bytes in size and the memory values start at `0800`. To access `lst[2]`, compute:

`0800 + 8 * 2 = 0816`



Given a memory address, we can get the value from that address with one calculation. Looking up an index in a list is fast!

Combine the Concepts

To implement constant-time search, we want to combine the ideas of post boxes and list index lookup. Specifically, we want 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 really quickly, without needing to check other indexes.

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** (integers).

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

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

Built-in Hash Function

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

```
x = "abc"
hash(x) # some giant number
```

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

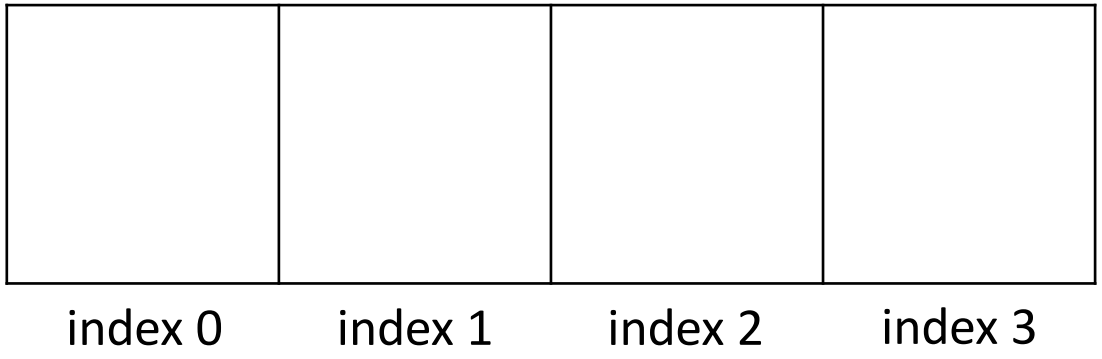
Optimizing Search: Hashtables

Hashtables Organize Values

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

A **hashtable** is a list with a fixed 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.

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets. **Important:** actual hashtables have far more buckets than this.



Adding Values to a Hashtable

For simplicity, let's say this hashtable uses a hash function that maps strings to indexes using the first letter of the string, as shown to the right. (This is not a good hash function, but it will serve as an example).

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

First, add `"book"` to the table. `hash("book")` is `1`, so we'll put the value in bucket 1.

Next, add `"yay"`. `hash("yay")` is `24`, which is outside the range of our table. How do we assign it?

Use `value % tableSize` to map integers larger than the size of the table to an index. `24 % 4 = 0`, so we put `"yay"` in bucket 0.

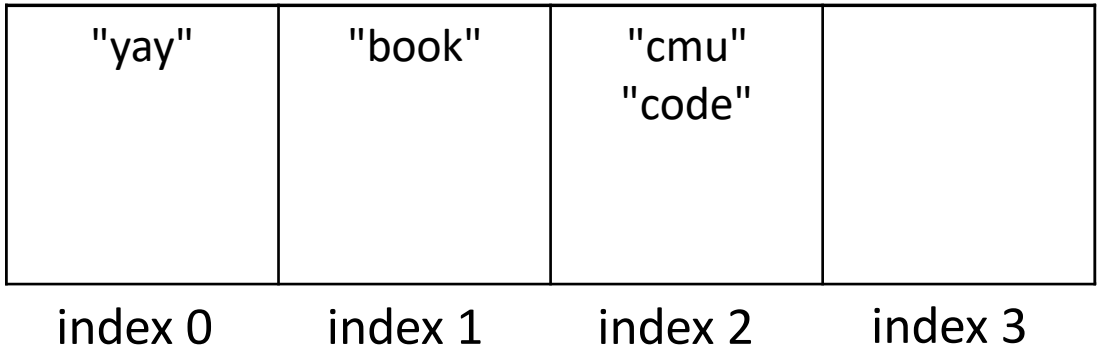
Dealing with Collisions

When you add lots of values to a hashtable, two elements may **collide**. This happens if they are assigned to the same index. For example, if we try to add both `"cmu"` and `"code"` to our table, they will collide.

Hashtables are designed to handle collisions. One algorithm for handling collisions is to put the collided values in a list and put that list in the bucket. *If* your table size is reasonably big and the indexes returned by the hash function are reasonably spread out, each bucket will normally hold a small number of values.

Our example hash function is not good because it only looks at the first letter. A function that uses all the letters would be better.

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

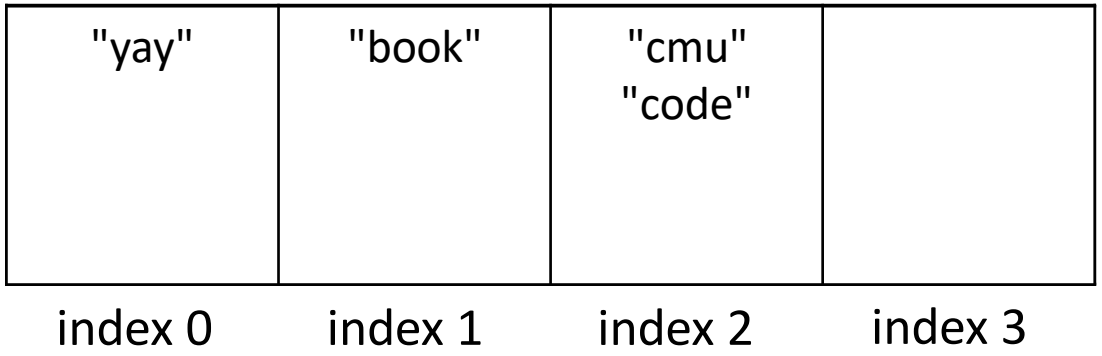


You Do: Search a Hashtable

Let's say that we want to algorithmically check whether the string `"friday"` is in our hashtable.

You do: Which buckets does the algorithm need to check?

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



Searching a Hashtable is Fast!

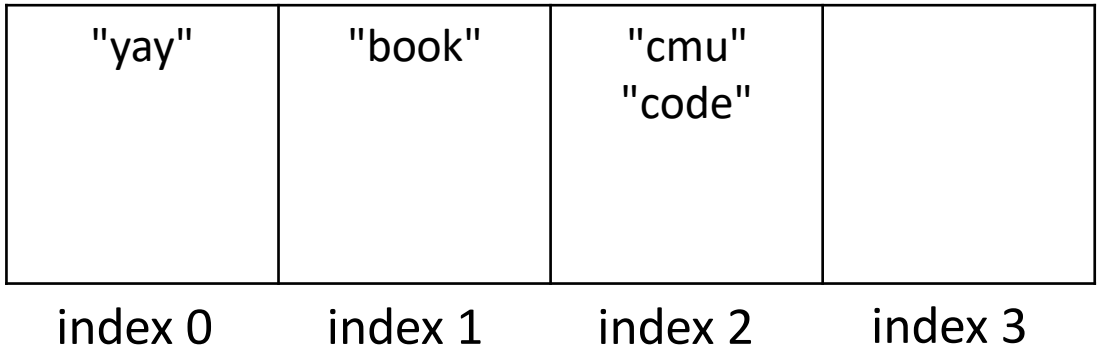
To search for a value, call the hash function on it and mod the result by the table size. **The index produced is the only index you need to check!**

For example, we can check if `"book"` is in the table just by checking bucket 1.

If the value is in the table, it will be at that index. If it isn't, it won't be anywhere else either. To check for `"stella"` just look in in bucket 2.

Because we only need to check one index and each index holds a constant number of items, finding a value only takes a few steps, **even if the hashtable is huge**.

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



Caveat: Don't Hash Mutable Values!

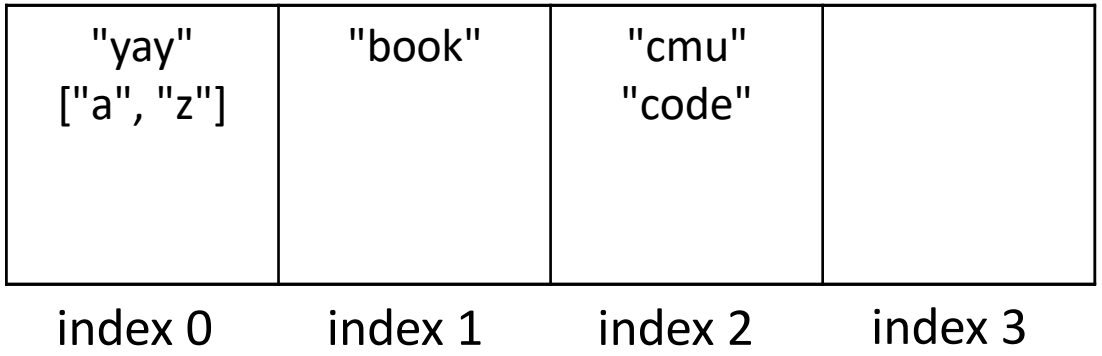
What happens if you try to put a list in a hashtable? Let's set `lst = ["a", "z"]` and use the given hash to add `lst`.

This might seem fine at first, but it will become a problem if you change the list before searching. Let's say we set `lst[0] = "d"`.

When we hash the list again, the hashed value is `3`, not `0`. But the list isn't stored in bucket 3! We can't find it reliably.

For this reason, **we don't put mutable values into hashtables**. If you try to run the built-in `hash` on a list, it will crash.

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



Dictionaries Use Hashed Search

Because hashed search requires immutable search values and a hashtable, it isn't used in lists or strings. However, it **is** used to implement dictionary search.

Recall that the keys of a dictionary must be **immutable**. This is because those keys are all stored in a hashtable. Each key points to its own value; that's how values can still be accessed.

This means that searching for a key in a dictionary is really fast! Dictionaries are **super efficient** for basic lookup tasks.

The Power of Hashing

Hashed search is absurdly fast! It doesn't matter how large your dataset is; you can always look up a value in the same amount of time.

This ridiculous speed of hashed search has made search a common tool across all computational devices.

Discuss: how would your interactions on your computer, smartphone, or other digital devices be different if search was slower? How would this affect your day-to-day life?

Learning Objectives

- Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**

Feedback: https://bit.ly/110-f21-feedback