# Search Algorithms II

15-110 – Monday 10/20

# Announcements

- Welcome back from fall break!
- Check3/Hw3 Revision deadline: **tomorrow at noon**
- Final exam scheduled: **Friday 12/12 1-4pm**
  - Do not schedule travel before this time!
- Remember to fill out the mid-semester surveys
  - See announcement on Piazza

# Learning Objectives

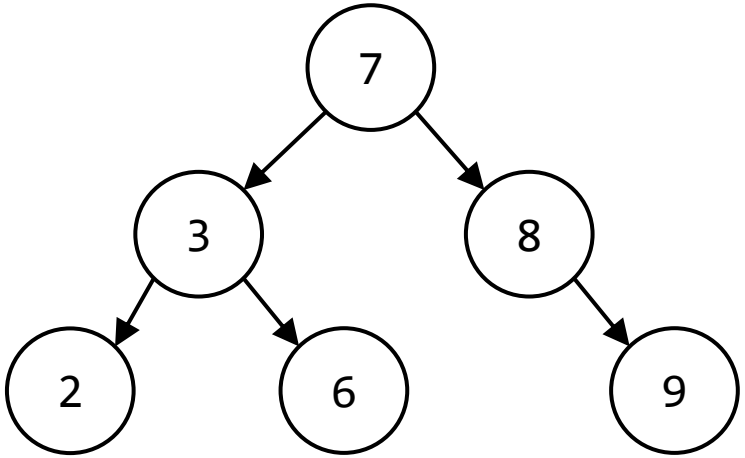
- Identify whether a tree is a **tree**, a **binary tree**, or a **binary search tree (BST)**
- Search for values in trees using **linear search** and in BSTs using **binary search**
- Analyze the **efficiency** of binary search on a **balanced** vs. **unbalanced** BSTs
- Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**

# Binary Search Trees

# Revisiting Search Algorithms

Recall the first lecture on Search Algorithms, when we discussed linear and binary search.

We've applied these algorithms to lists; can we apply them to other data structures too? Let's investigate how to search a **tree**.



# Linear Search on a Tree

In linear search, we step through each element in a list until we either find the target item or run out of items to look at.

To visit all nodes in a tree, run the function recursively on both the left and right children. If we find the target in **either** subtree, we should return `True`.

We also have two base cases: one for when we reach an empty tree, and one for when we find the target. In both cases, we know what to return right away.

```
def search(t, target):
  if t == None:
    return False
  elif t["contents"] == target:
    return True
  else:
    leftSide = search(t["left"], target)
    rightSide = search(t["right"], target)
    return leftSide or rightSide
```

# Binary Search on a Tree

If we want to search trees more efficiently, we'll need to apply **constraints**. For example, how could we apply Binary Search to a tree?

First, recall that for binary search to work the input list must be **sorted**. We'll also need to find a way to split the tree similarly to how we split the list in binary search (where we broke the list into two sides and only looked at one side).

**Discuss:** how could we "sort" and "split" a tree?

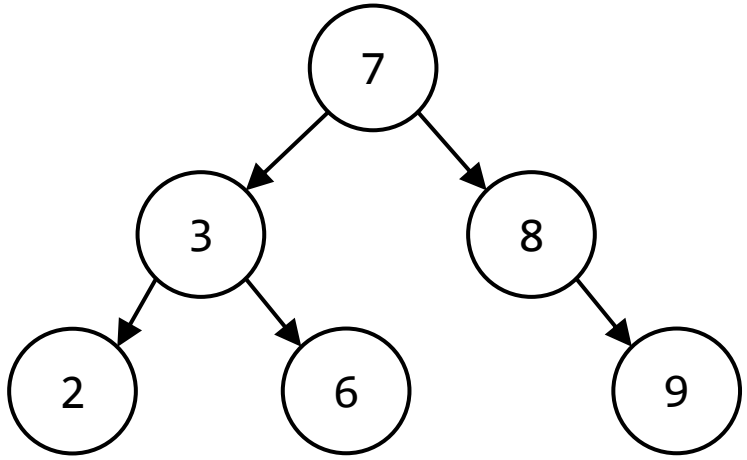
# Binary Search Trees (BSTs) are "sorted"

We'll define a new kind of tree, a **Binary Search Tree,** as a binary tree that follows these constraints:

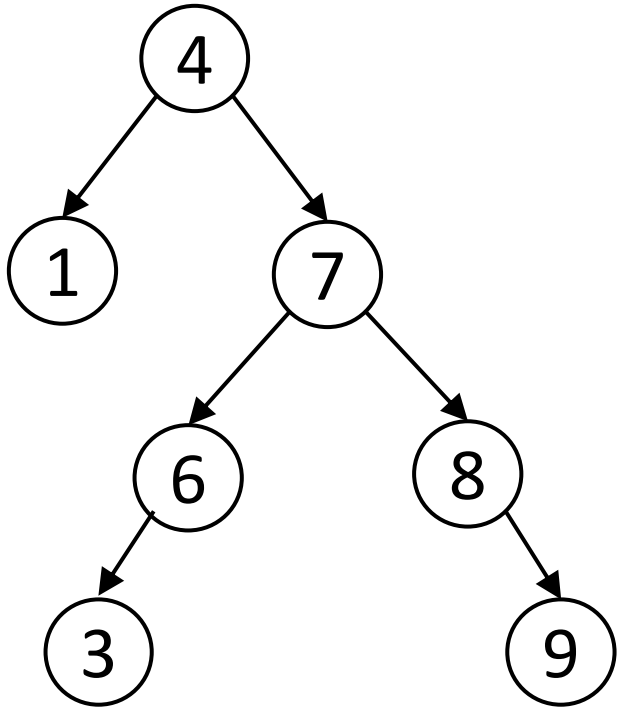
For **every** node n with value v:

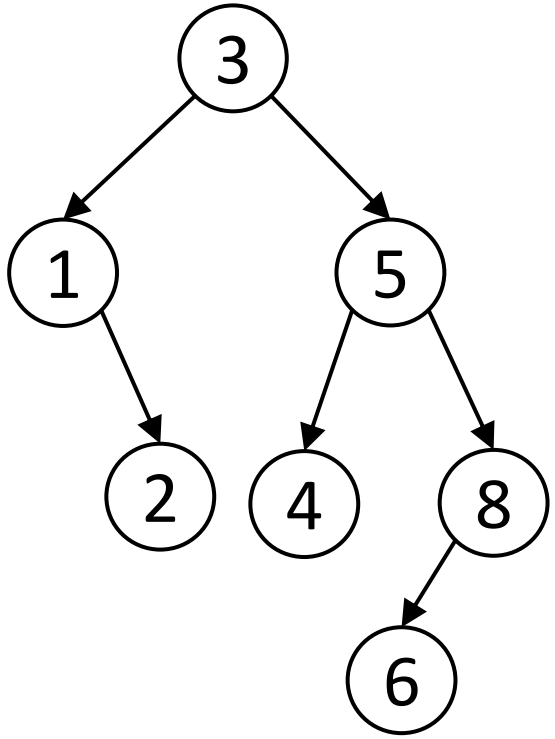
- Its **left** child (and all its children, etc.) must have a value strictly **less** than v
- Its **right** child (and all its children, etc.) must have a value strictly **greater** than v

The left and right subtrees must also be BSTs. BST constraints are **recursive**.



# Example: Is this a BST?



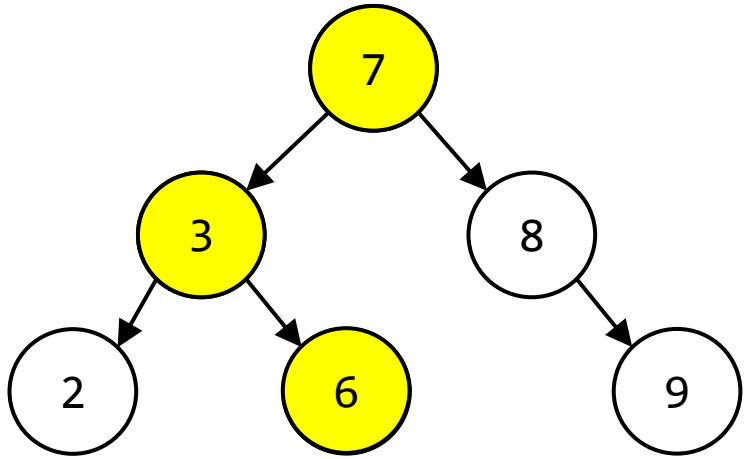
# Binary Search Trees Can Use Binary Search

When we want to search for the value 5 in the tree to the left, we start at the root node, 7.

Because all nodes less than 7 must be in the left child tree and 5 is less than 7, **we only need to search the left child tree.**

Then, when we compare 5 to 3, we know that all values greater than 3 (but less than 7) must be in the right child tree of 3. **We only need to search the right child tree**.

We 'split' the tree by only looking at one of the node's two children each time.



# BST Search in Python

We would write binary search for a BST as follows:

```
def search(t, target):
    if t == None:
        return False
    elif t["contents"] == target:
        return True
    elif target < t["contents"]:
        return search(t["left"], target)
    else:
        return search(t["right"], target)
```

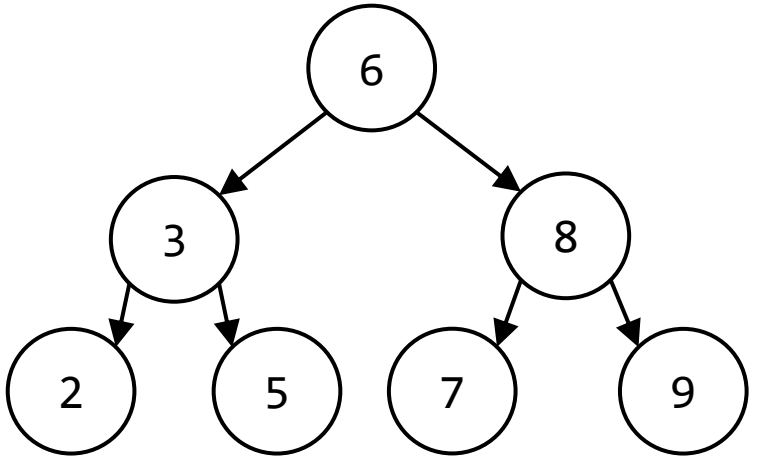
Note that we do just **one** recursive call, either on the left subtree or on the right subtree.

# BST Search Runtime – Balanced Trees

Do we get the same O(log n) runtime for BST binary search that we did for list binary search? **It depends on the tree.**

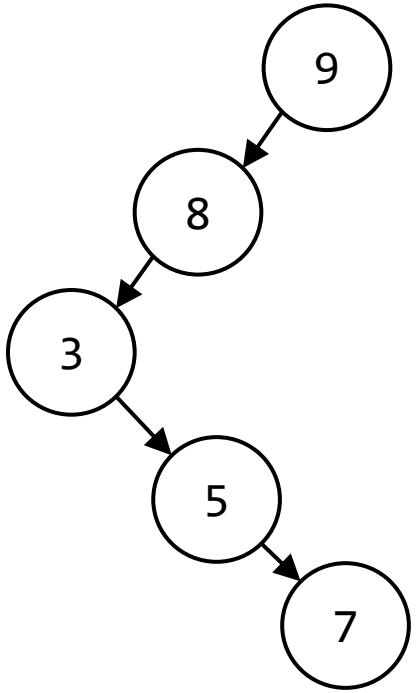
A tree is **balanced** if for every node in the tree, the node's left and right subtrees are approximately the same size. This results in a tree that **minimizes** the number of recursive levels.

Every time you take a search step in a balanced tree, you cut the number of nodes to be searched in half. This means that the algorithm will indeed take O(log n) time.



# BST Search Runtime – Unbalanced Trees

A tree is considered **unbalanced** if at least one node has significantly different sizes in its left and right children. For example, consider the tree on the right.




This is a valid BST, but it is still difficult to search! You must visit every single node to determine a number like 6 isn't in the tree. In the worst case, this can still take **O(n)** time.

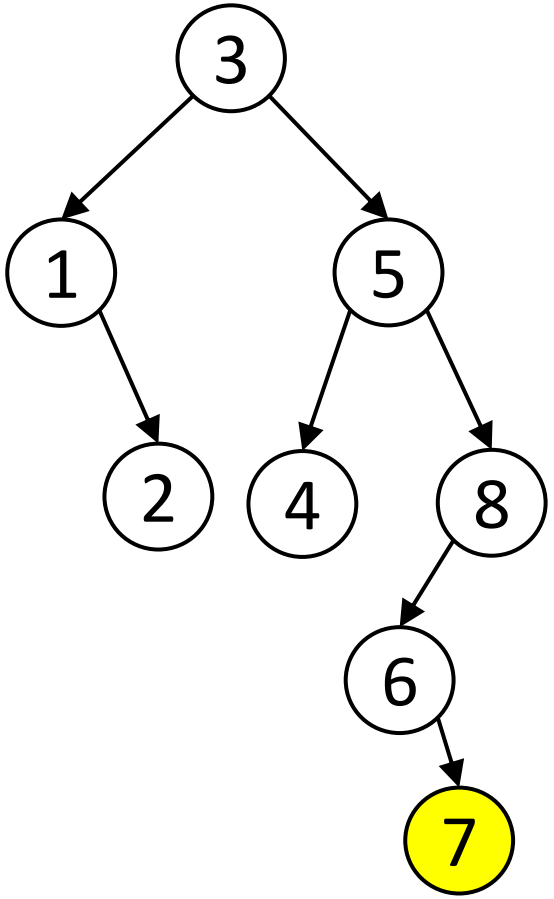
When we put data into BSTs, we usually strive to make them balanced to avoid these edge cases. For efficiency purposes, assume that well-designed BSTs are balanced and the worst case is O(log n).

# Benefits of BSTs

At first glance, BSTs may seem less useful than sorted lists. However, they have a few added perks!

BSTs make it much easier to **add new data** to a dataset. In a sorted list, you would need to slide a bunch of values over to make room for a new value; in a BST, you can just run a search for this new value. When you reach a leaf, add a node with the new value.

This is very helpful for systems like hospital priority queues, where patients frequently need to be moved around the queue based on changing circumstances.



# Can We Do Even Better?

We've now shown that we can apply linear search and binary search in several circumstances. 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. Or try putting new **constraints** on the problem to speed the process up.

New goal: can we add more constraints to 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 - you just check the box **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!

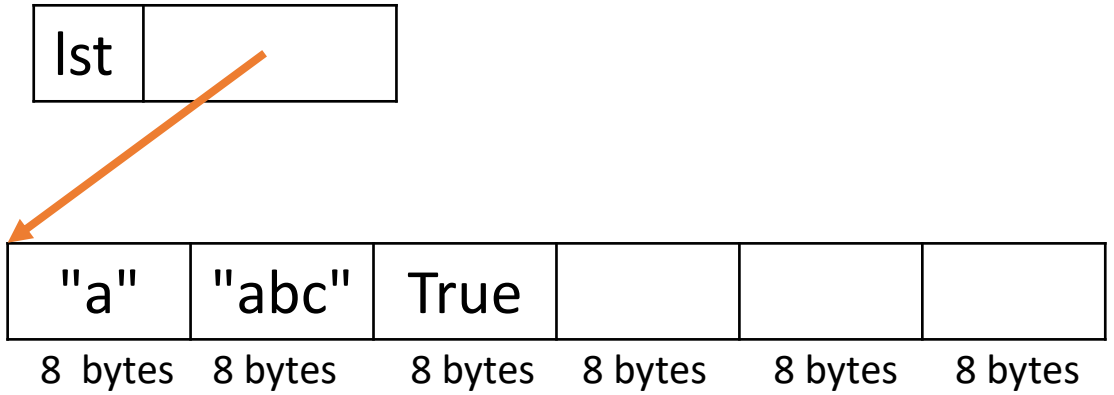
Compare this to picking up a package. Everyone picks up packages at the same window, so you must wait in line. If there are n students, picking up a package is a **O(n)** operation.

# Search in Programming – List Indexes

We can't search a list for an item that quickly, because we don't know where the item will be. But we **can** look up an item based on its index very quickly!

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**.

We can calculate exactly where an index is located in memory with a single equation; no repeated search is required.



# Combine the Concepts

To implement super-fast search, we want to combine the ideas of post boxes and list index lookup. We want to determine which index a value should be stored in **based on the value itself**.

If we can calculate the index based on the value, we can look for 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:

- The function should be **deterministic**. Given a specific value `x`, `hash(x)` must **always** return the same output `i`.
- The function should produce **different outputs**. 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

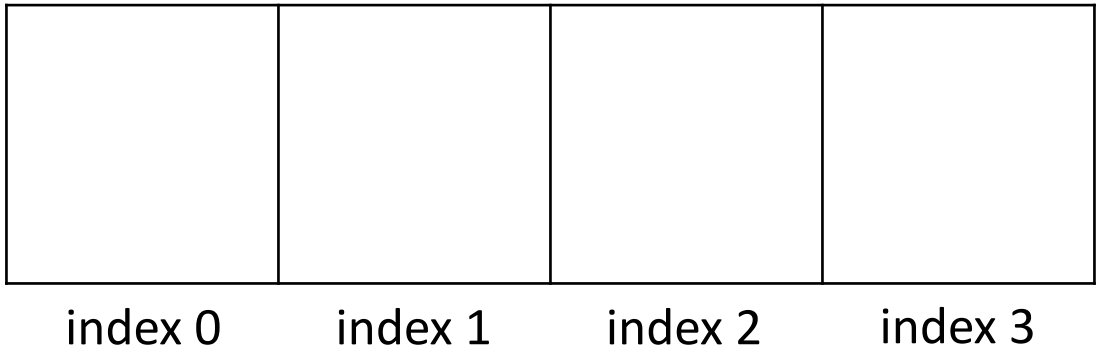
# Hashtables Organize Values

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

A **hashtable** is a structure with a fixed number of indexes. When we place a value in the structure, we put it into an index **based on its hash value** instead of placing it in the first open position of the structure.

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets.

**Important:** actual hashtables are **huge** and 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.

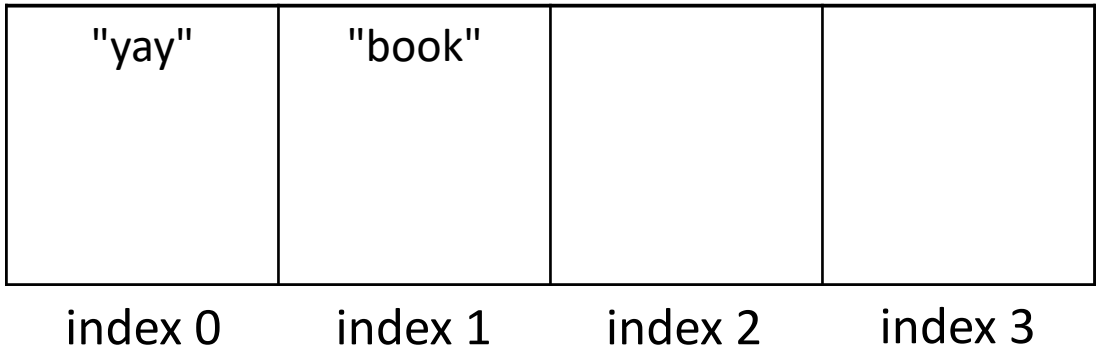
First, add `"book"` to the table. `hash("book")` is `1`, so we'll put the value in bucket 1.

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.

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. And our example hashtable is far too small!

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```

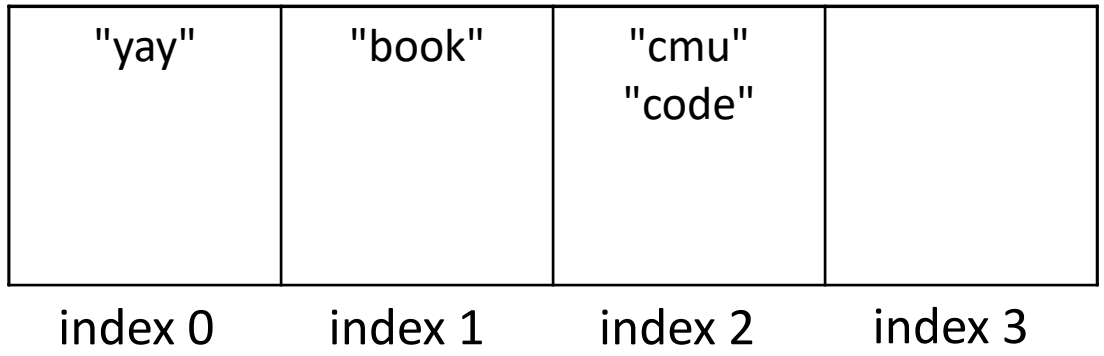


# Dealing with Collisions

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.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```

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 usually hold a small number of values.

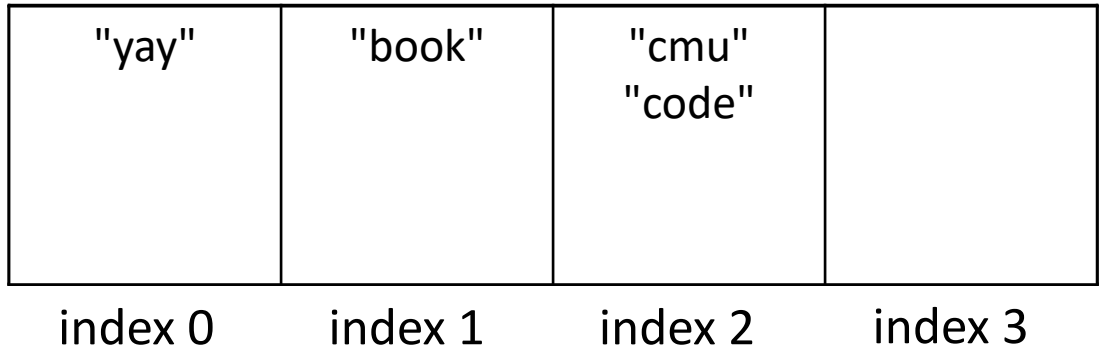


# You Do: Search a Hashtable

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

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```

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



# Searching a Hashtable is Fast!

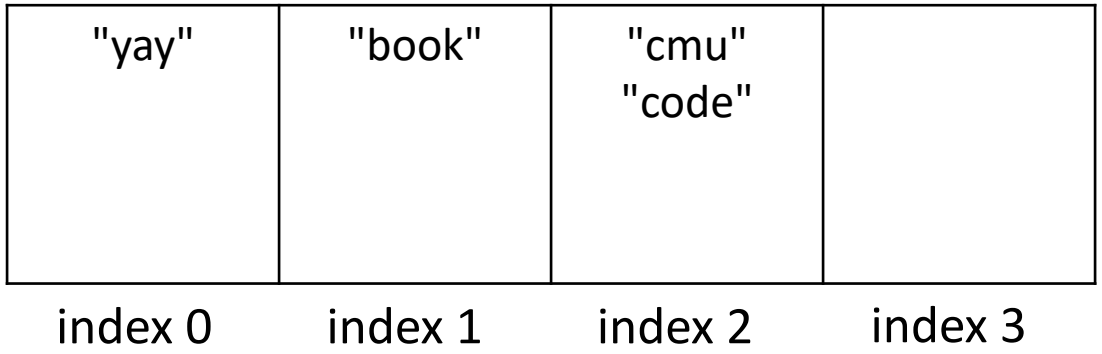
To search for a value, use the same algorithm you would use to insert that value. **The index produced is the only index you need to check!**

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 **O(1) time**, **even if the hashtable is huge**. Wow!

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```



# Caveat: Don't Hash Mutable Values!

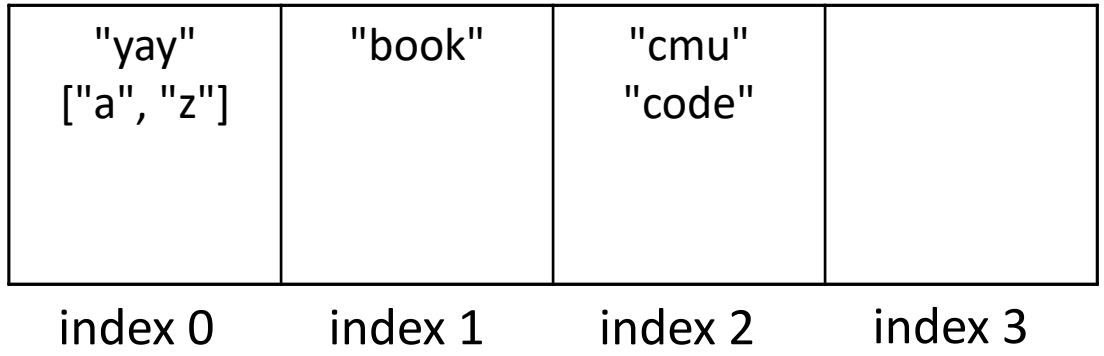
What happens if you try to put a list in a hashtable? Let's set `lst = ["a", "z"]` and use the given hash to add `lst`.

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.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```



# 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 O(1)! Dictionaries are **super efficient** for basic lookup tasks.

# Searching Dictionaries vs. Lists

This has a practical effect on the efficiency of the programs you write. Recall the built-in operator `in`, which checks for membership in a data structure.

`item in lst` runs in **linear time** if `lst` is a list, because Python can't guarantee that the list is sorted. It uses **linear search**.

`item in dict` runs in **constant time** if `dict` is a dictionary, because Python uses hashing.

If you know that you'll need to do a lot of searching for specific values, it's better to store your data in a dictionary than a list, even if it’s a sorted list!

# 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.

# Learning Objectives

- Identify whether a tree is a **tree**, a **binary tree**, or a **binary search tree (BST)**
- Search for values in trees using **linear search** and in BSTs using **binary search**
- Analyze the **efficiency** of binary search on a **balanced** vs. **unbalanced** BSTs
- Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**