# Dictionaries

# Searching...

Searching an array takes longer if there's more "stuff"

It would be great to have a data structure that works like an array...

- Meaning you can jump right to a piece of data

...but can use different data for its index

- Remembering that "Shiro" is player 3 isn't great
- `players["Shiro"]` would be much better!

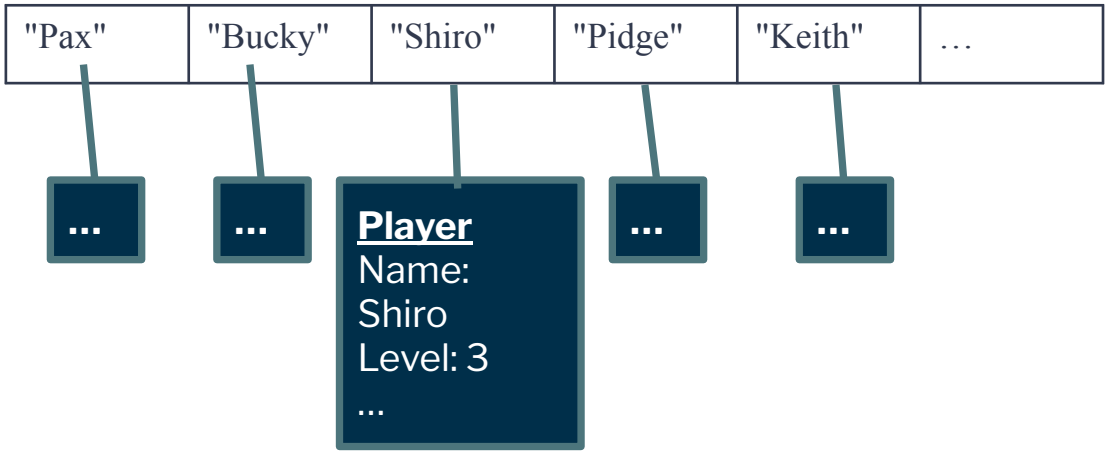
In other words, We want to:

- Store data in an array/list like structure
- But search it in constant time (i.e. O(n) = 1)

# Dictionaries to the Rescue!

Lets us store data (a **value**) that is indexed with a **key**

- We can use `players["Shiro"]`
- And it won't require a linear search!!!



# Dictionaries in C#

Dictionaries require TWO type specifiers

- It's generic, similar to the List<> class
- We can use any type of data we want!

`Dictionary <KeyType, ValueType>`

- You can use any data type as the key
- You can store any data type as the value

```
Dictionary<String, Player> playersDictionary =
   new Dictionary<String, Player>();
```

**NOTE: KeyType can be ANY type!** We use strings a lot, but keys can be int, strings, bools, Players, floats, ….. *anything!*

# Adding Data

Data can be added with the Add() method
Or by direct index (similar to an array).

```
// Create Player objects
Player bobVariable = new Player("Bob Smith", 200);
Player jimVariable = new Player("Jim Miller", 9872);

// Add via Add method
playersDictionary.Add("Bob", bobVariable);

// Add via direct index
playersDictionary["Jim"] = jimVariable;
```

# Count

Like Lists, Dictionaries have a Count property.

```
// Number of items inside dictionary
Console.WriteLine(roster.Count);
```

# Retrieving Data

```
Player bob = playersDictionary ["Bob"];
```

What happens if bob doesn't exist?

- An exception!  Your program crashes

Must check for the key first!

```
Player bob = null;
if( playersDictionary.ContainsKey("Bob") )
   bob = playersDictionary ["Bob"];
```

# Common Dictionary Methods

`Add(key, value)` – Adds a key/value pair

`Remove(key)` – Removes a key/value pair

`Clear()` – Removes all data

`ContainsKey(key) - O(1)`

- Determines if the specified key exists
- This is a fast operation!  Not a linear search.

`ContainsValue(value)` - O(n)

- Determines if the specified value exists
- This is a slow (linear search) operation!

# Demo Code

Try running this in a console app.

Use the debugger to see what the underlying data looks like!

```
// Create a new dictionary that maps
// strings to other strings
Dictionary<String, String> dict = new
Dictionary<String, String>();

// Add some data
dict.Add("Bob", "123-4567");
dict.Add("Bobx", "cell number");
dict.Add("Sally", "555-5555");
dict["Jenny"] = "867-5309";

Console.WriteLine("Bob's number: " + dict["Bob"]);

// Check to see if the key "Larry" is in
// the dictionary before attempting to
// retrieve it
if (dict.ContainsKey("Larry"))
{
     Console.WriteLine(dict["Larry"]);
}
```

# How do dictionaries work?

# Dictionary

Stores pairs of data

- Keys & Values
- Both are required

Keys are used to look up values

- Need the key to find the value

Allows for very fast lookups

- O(1) complexity - "Constant time"
- Searching a List is O(n) – "Linear time"

# Dictionary Uses

Great for lookups based on a unique key

- Finding a player by name
- Finding an image based on its filename
- Usually dictionary keys are non-numeric

Not great for general storage

- A list of grades
- All the enemies in a level
- A player's inventory

Dictionaries shouldn't replace arrays or lists in all cases!

# Why are Dictionary lookups "faster"?

The *key* plays a vital role

Dictionaries store data in arrays internally

- As do most complex data structures
- Can store multiple pieces of data easily

Arrays are indexed by number

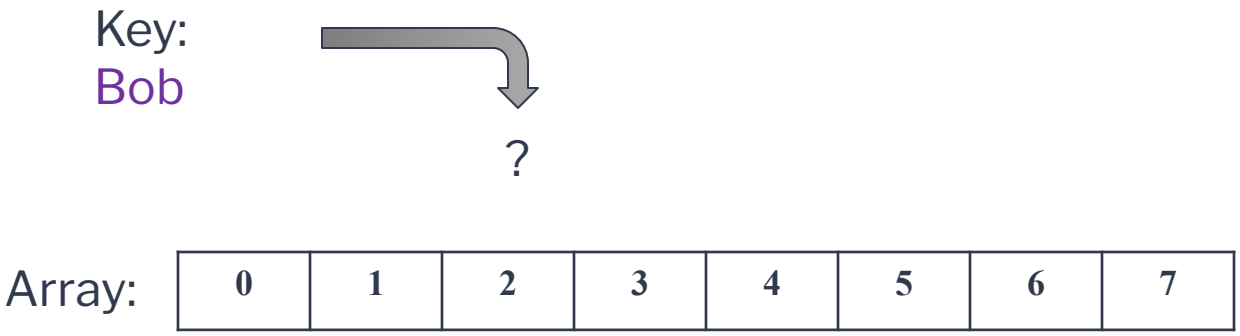
- But our keys aren’t necessarily numbers
- What can we do about that?

# Turning Keys into Numbers

Assume our keys are strings for this example

I have a string, I need a number

What could I do?



# String □ Number

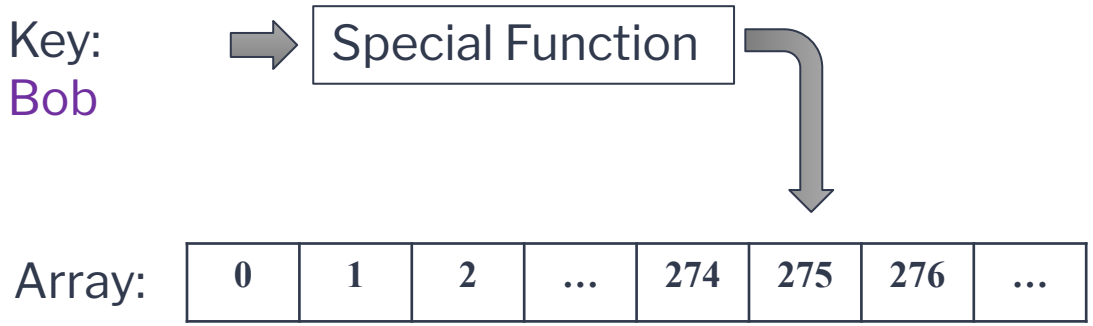
Strings are made up of characters

Each character has a particular numeric value

- An ASCII code, for example – Just a number

What if I add up all the character values?

- B = 66
- o = 111
- b = 98
- Total = 275



# Why is this faster?

If we have the key, we can jump directly to a spot in the array

We don’t actually care how much data exists

We bypass it and get exactly the data we need

- In a perfect world…
- It’s not exactly that easy

# Hashing & Hash Tables

The process of converting a string into a key that represents the original string

"Hash Table"

- General term for data structures like C#’s Dictionary
- Exist in many languages

Hash function

- The particular set of steps we use to get a hash from an arbitrary key

# More about hash functions...

# For example: simple hash of a string....

Add up all the ASCII values in a string

Will always give us a number

- B = 66
- o = 111
- b = 98
- Total = 275

What are some potential issues with this simple hash function?

# Issue – Collisions

A collision is when two different keys hash to the same number

- Results in a conflict
- Both keys point to the same place in the array

Could happen because of the mod operation

Could happen if two keys are similar

- coat
- taco
- Same ASCII codes!

# Okay, get a better hash function!

Multiply the ASCII code by its position in the string!

The following will give different results:

- (c * 1) + (o * 2) + (a * 3) + (t * 4)
- (t * 1) + (a * 2) + (c * 3) + (o * 4)

Better, although still has room for error

# Issue – Really Big Number

Example: My dictionary has an internal array that can store up to 100 values

My hash function produces numbers >100

Is there something I can do to the number to keep it in the range?

# Solution – Really Big Number

**Mod** to the rescue

I can mod the hash by the size of the array

- Hash value = 275
- Array size = 100
- 275 **%** 100 = 75

The result is within the valid range (0 – 99)

Any other potential issues with this?

# What about other types?!

ALL types in C# can produce their own hash values!

Including instances of classes that you define!

Remember that ALL classes inherit from Object by default!

*This is how they all have a ToString() method even if you don't override it!*

Object also supplies a default GetHashCode()!

- If you don't override it, it computes a hash code based on an object's reference *(i.e. where the object is in memory)*
- You could override to define 2 objects as being the same based on only some fields *(but we don't do that often)*.

# Still, there is NO perfect hash function!

A hash function, *by definition*, projects a value from:

**A set with many (or even infinite) members**

To

**A value from a set with a fixed number of members**

With enough keys, collisions are inevitable!

# Collision Handling

Two different keys could result in the same hash code and thus end up placed in the same “bucket”.

To be useful, dictionary implementations have to handle collisions somehow!

*(Also called collision “resolution”)*

# Collision Handling

There are several ways of handling collisions

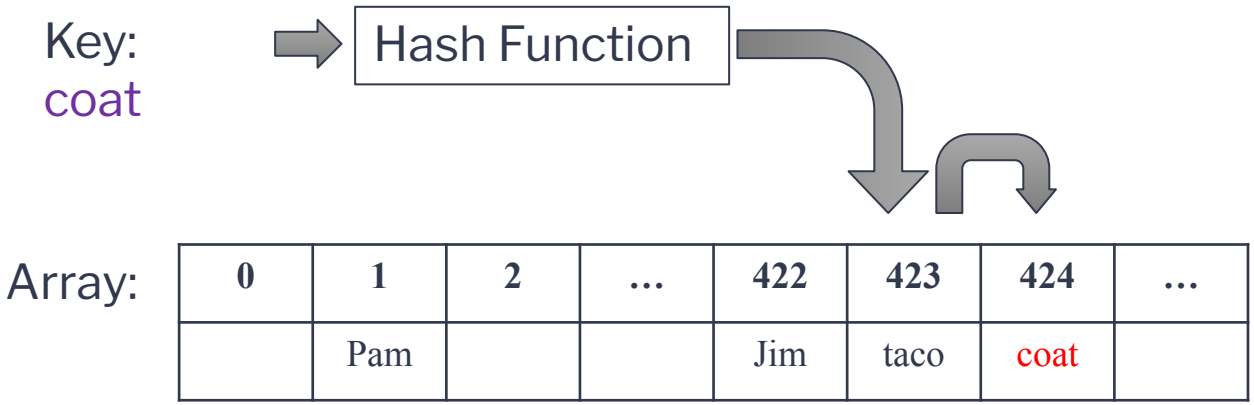
- "Open Addressing"
- "Chaining"
- And several others

All of them will slow the process down a little

- But still faster than searching an entire List

# Open Addressing

If the key hashes to an array spot that is already full…




Choose the next open spot!

- Using the next spot is “open addressing with linear probing”.
- Open addressing with other strategies is also an option

# Open Addressing Issues?

What issues arise from just picking another open spot?

How do we find this data again?!

- We have to search for it
  *(“probe” starting from where the hash sent us)*

How do we search if we’re not storing keys?

- We have to also store the keys

# Key/Value Pairs

Most hash tables store the **key** *and* the **value**

Once we find a potential value

- Ensure the keys match
- Still fast since it’s just one extra comparison

One extra step, but allows for simple collision handling

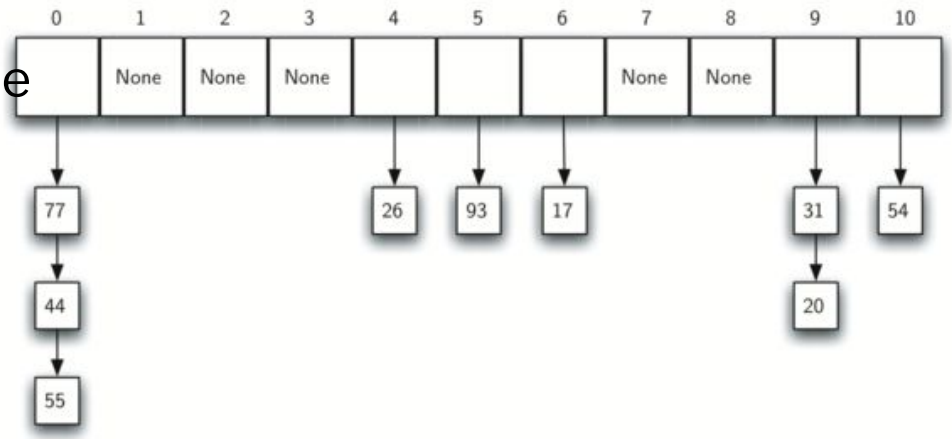
# Chaining

Each spot in the array is actually a List of Key/Value pairs

- An array of lists
- `List<KeyValuePair>[ ] buckets;`

Each time you look up a key…

- Must also search a (usually small) list of KV pairs for the one you want
- Still faster than searching the entire dictionary




This is what C#'s Dictionary class uses