Intro to Hashing

CMSC 420

Terminology

- Looking at texts, you will see stuff such as
  - **Maps** (implemented in C++ STL)
  - **Multi-Maps** (also implemented in C++ STL)
  - **Associative Arrays** (PHP and others)
  - **Dictionaries** (Python's dict())
  - **HashMaps / HashTrees** (Java)
  - **Lookup tables** (academic papers by people who haven't programmed in a decade)

Terminology

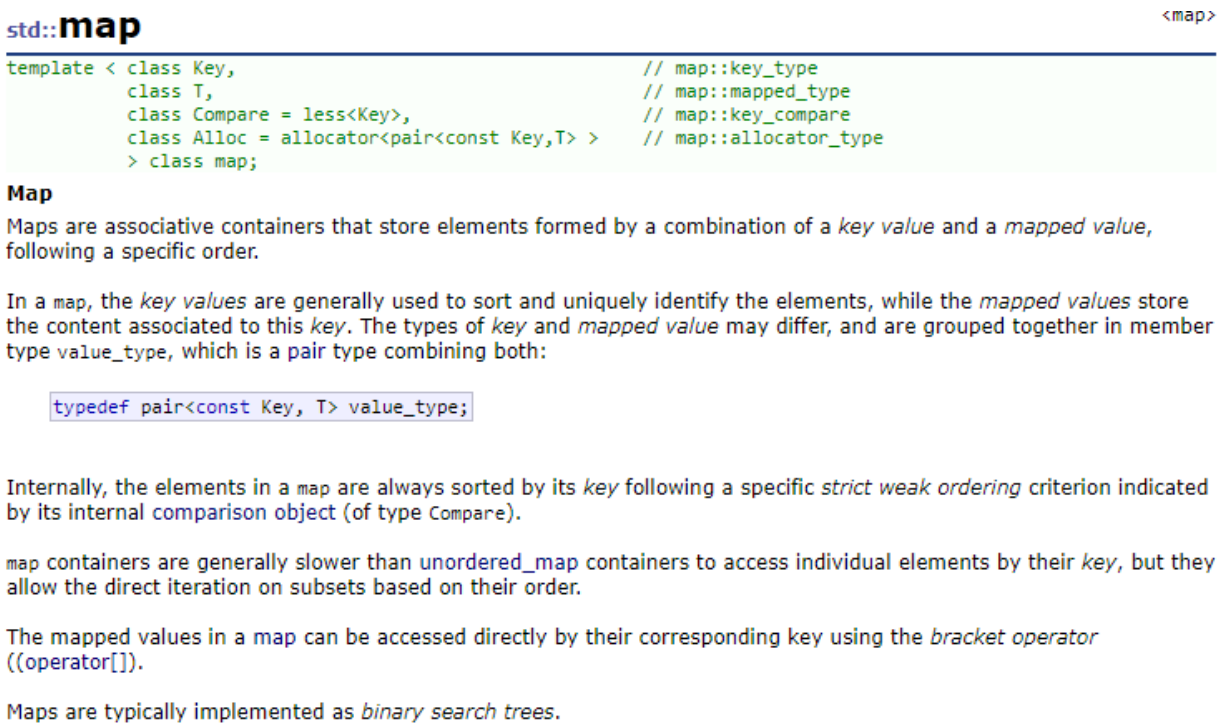
- Looking around, you will see stuff such as
  - **Maps** (implemented in C++ STL)
  - **Multi-Maps** (also implemented in C++ STL)
  - **Associative Arrays** (PHP and others)
  - **Dictionaries** (Python's dict())
  - **HashMaps / HashTrees** (Java)
  - **Lookup tables** (academic papers by people who haven't programmed in a decade)
- Very few people can intelligently converse about the differences between those….
- Today, we will try to move towards that direction, and then some!

Key-Value pairs

- There is only one mistake you can do here, and it is to equate Key-Value pairs <*K, V*> with hash tables.
- The vast majority of the course concerns organization of **`Comparable`** keys such that we can search for keys efficiently.
  - Values assumed to be reachable in $O(1)$ time from the key.
- Classic examples: Maps, Treemaps (C++, Java)
- If you work on a modern database (Mongo, Kassandra), you will witness the shift from the relational model of databases (RDBMS) to the much more flexible *Key-Value Store* model.

Speaking of Maps…

- **Map**s are modern programming language's ways of saying "Key-Value store" (a store of pairs <K, V>, where K has to be some Comparable type)
- C++: std::map



std::**map** <map>

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

Speaking of Maps…

- **Map**s are modern programming language's ways of saying "Key-Value store" (a store of pairs <K, V>, where K has to be some Comparable type)
- C++: std::map

Ctrl + F "hash" on that page returns at most 0 results!

std::**map** <map>

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

Speaking of Maps…

- Java also has Maps!
  - Interface Map<K, V>

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterState
TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

---

public interface **Map<K,V>**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Speaking of Maps…

- Java also has Maps!
  - Interface Map<K, V>

R-B Tree based (we've already seen this…)

Almost entirely the same (HashTable is thread-safe)

Interesting reads for you

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterState, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

---

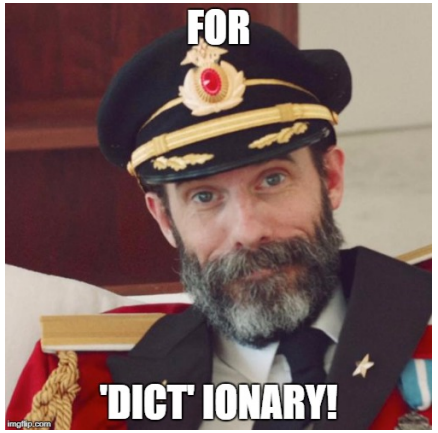
public interface **Map<K,V>**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Take-home message

- A Map (or Key-Value store) is **not** a hash table!
- A hash table is simply an **implementation** of a Map. Whether or not you should use a hash **depends on the kinds of queries you want to speedup**.
  - **Range query**, groupby, aggregations? **R-B Trees / B-Trees.**
  - Insertion, deletion, search of a **single record**? **Hash Tables.**
- If you're able to understand this distinction:
  - **You will become better Computer Scientists**
  - **Jason will be happy**

Python's `dict()`s



- In Python, you can do this:

```
d = dict([(1, “Jason”), (2, “Shandra”), (4, 5), (5,
    date.today()), (11, {“pi”:3.14, “e”: 0.58})]}
```

- And then this:

```
print(d[2]) // Will print Shandra
d[2]=“Monica” //  Updates the value pointed by key 2
print(d[2]) //  Will print Monica
```

Python's `dict()`s

- In Python, you can do this:

```
d = dict([(1, “Jason”), (2, “Shandra”), (4, 5), (5,
   date.today()), (11, {“pi”:3.14, “e”: 0.58})]}
```

- But you **can't do this**!

```
print(d[2:6])
```

```
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print(d[2:6])
TypeError: unhashable type: 'slice'
```

Python's `dict()`s

- In Python, you can do this:

```
d = dict([(1, “Jason”), (2, “Shandra”), (4, 5), (5,
   date.today()), (11, {“pi”:3.14, “e”: 0.58})])
```

- But you **can't do this**!

```
print(d[2:6])
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print(d[2:6])
TypeError: unhashable type: 'slice'
```

2:6 is a “slice” in Python

- If you want to do **range search**, you'll need **to write code on top** (sort, loop,…)
- ***Implication***: Python `dict()`s are ***hash tables!***
  - *(With flexible value types!)*
- They will find / update / delete `d[key]` very fast, but **they're not useful for other queries!**

Associative arrays

- In PHP, you can define an array ages like this

```
$ages = array(“Jason”->28, “Sravanthi”->29, “Varun”->27,
              “Keith”->30);
```

- And you can do the same things you could do with a Python `dict()`:

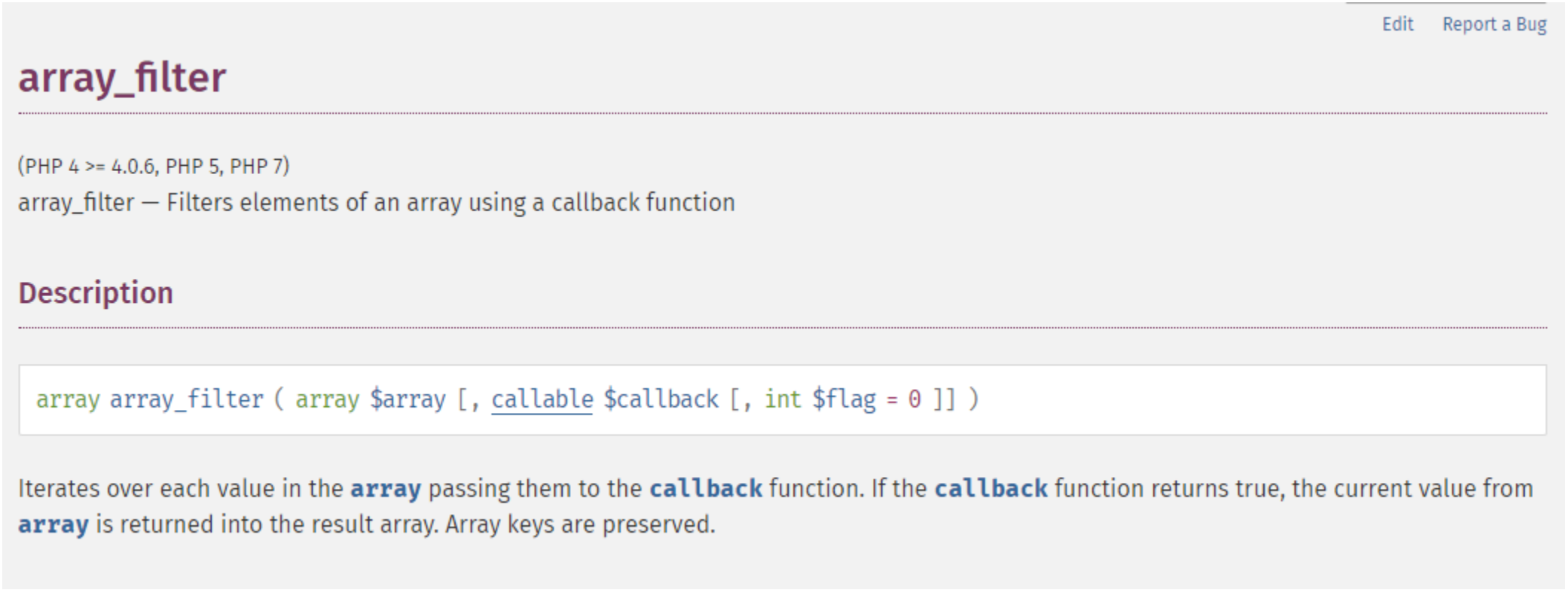
```
echo $ages[“Sravanthi”]; // prints 29
$ages[“Sravanthi”]=26; // Updates the value pointed to by “Sravanthi”
echo $ages[“Sravanthi”]; // prints 26
```

Associative arrays

- But you **still** don’t have native support for a range search ☹

Associative arrays

- But you **still** don't have native support for a range search ☹
- Even in methods where it looks like a range can be efficiently generated:



Edit Report a Bug

array_filter

(PHP 4 >= 4.0.6, PHP 5, PHP 7)

array_filter — Filters elements of an array using a callback function

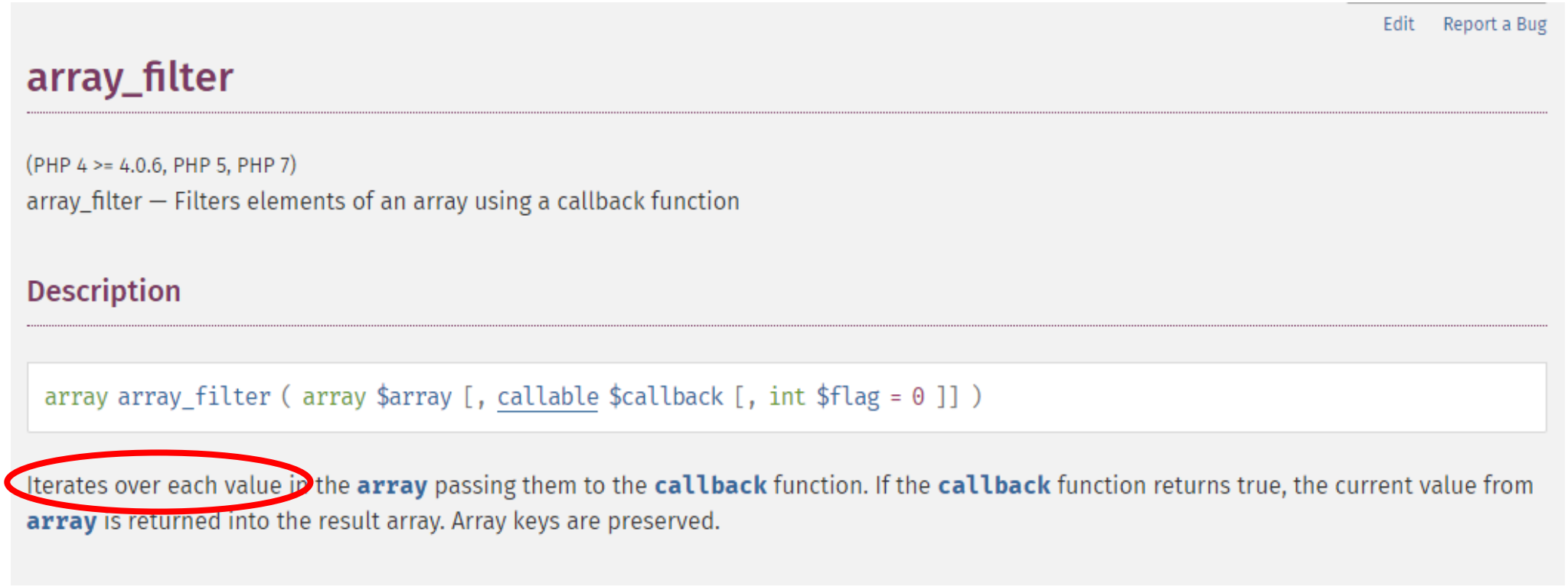
Description

```
array array_filter ( array $array [, callable $callback [, int $flag = 0 ]] )
```

Iterates over each value in the **array** passing them to the **callback** function. If the **callback** function returns true, the current value from **array** is returned into the result array. Array keys are preserved.

Associative arrays

- But you **still** don't have native support for a range search ☹
- Even in methods where it looks like a range can be efficiently generated:



Edit Report a Bug

array_filter

(PHP 4 >= 4.0.6, PHP 5, PHP 7)

array_filter — Filters elements of an array using a callback function

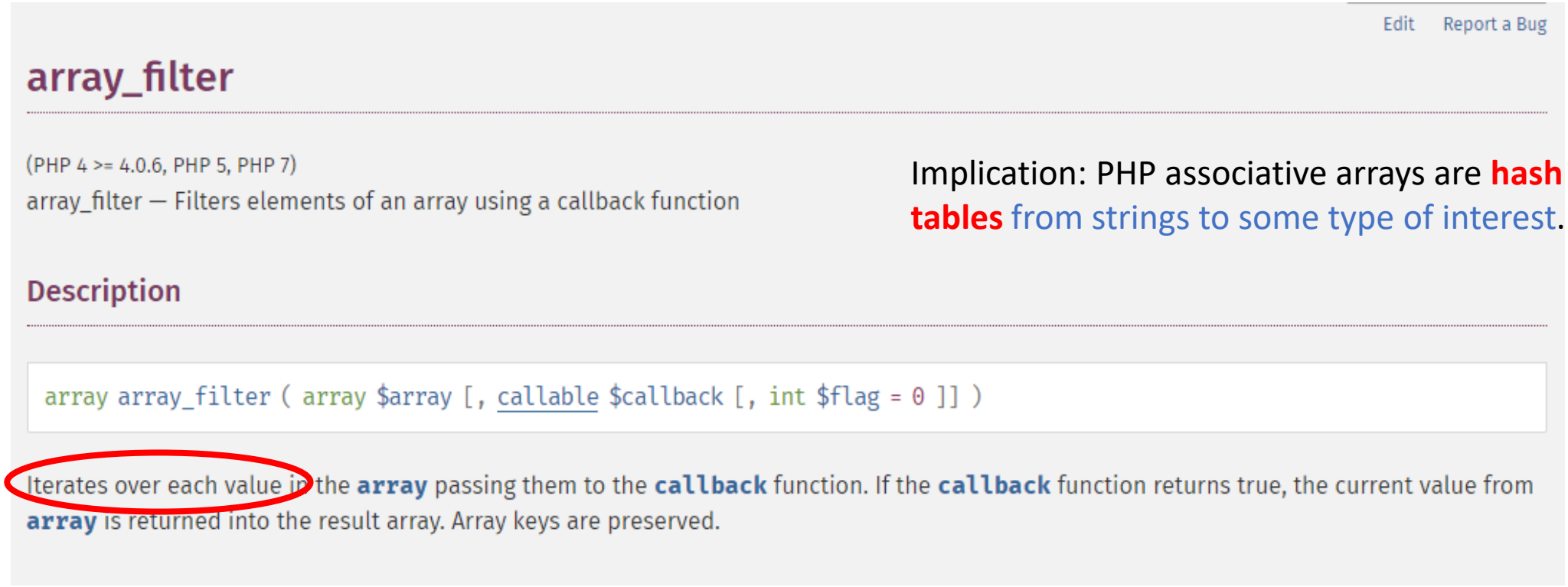
Description

```
array array_filter ( array $array [, callable $callback [, int $flag = 0 ]] )
```

Iterates over each value in the **array** passing them to the **callback** function. If the **callback** function returns true, the current value from **array** is returned into the result array. Array keys are preserved.

Associative arrays

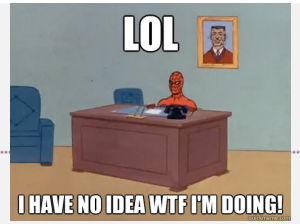
- But you **still** don't have native support for a range search ☹
- Even in methods where it looks like a range can be efficiently generated:



Implication: PHP associative arrays are **hash tables** from strings to some type of interest.

Associative arrays

- But you **still** don't have native support for a range search ☹
- Even in methods where it looks like a range can be efficiently generated:



Edit Report a Bug

array_filter

(PHP 4 >= 4.0.6, PHP 5, PHP 7)

array_filter — Filters elements of an array using a callback function

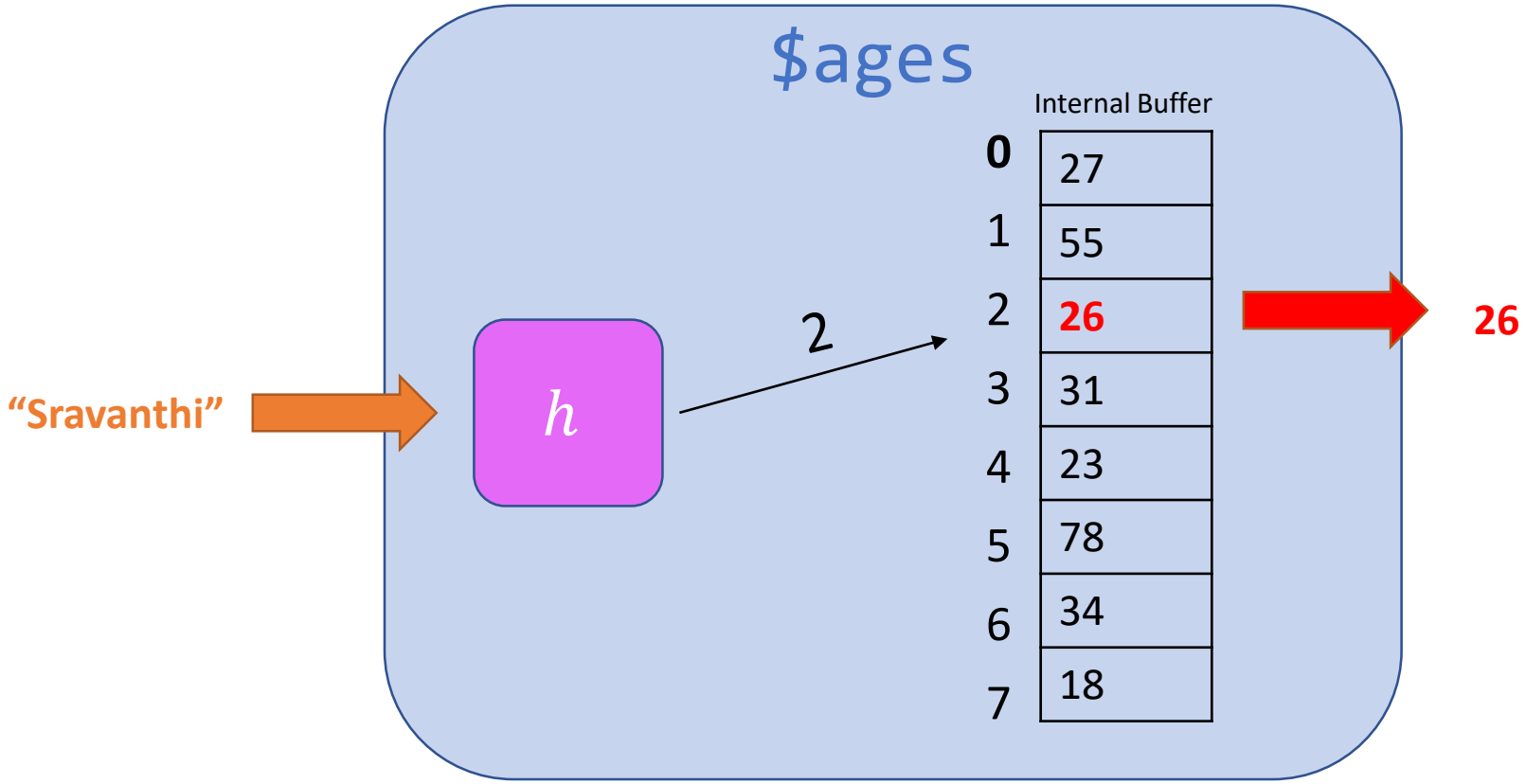
Implication: PHP associative arrays are **hash tables** ***from strings*** to some type of interest**???**

Description

```
array array_filter ( array $array [, callable $callback [, int $flag = 0 ]] )
```

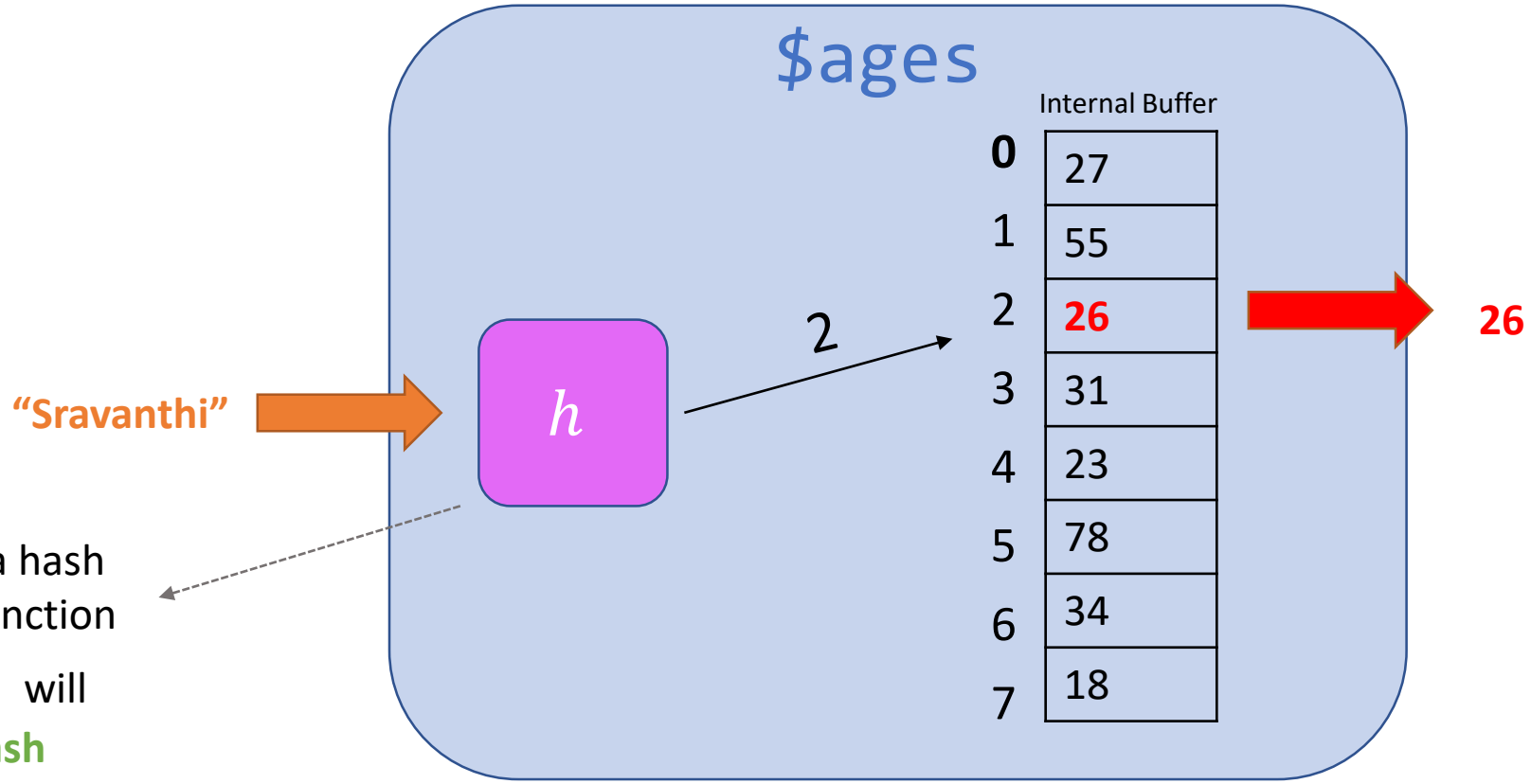
Iterates over each value in the **array** passing them to the **callback** function. If the **callback** function returns true, the current value from **array** is returned into the result array. Array keys are preserved.

Strings are hashable




```
echo $ages[“Sravanthi”];
```

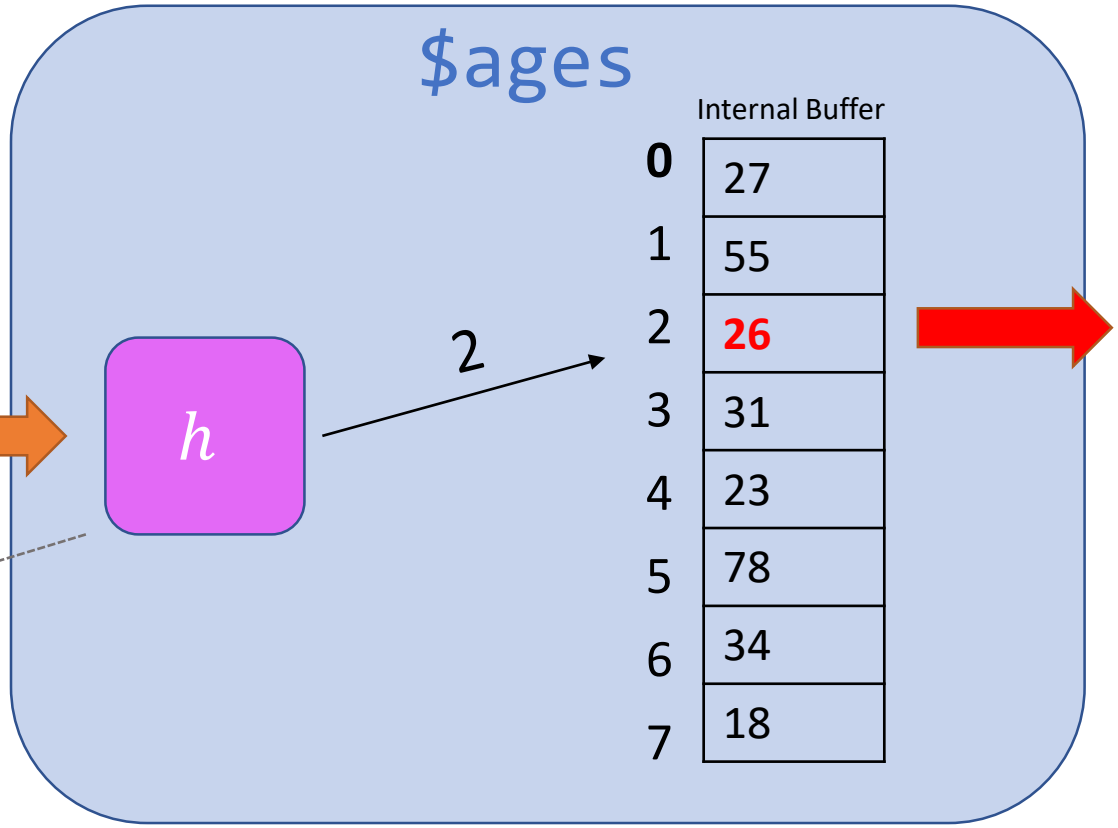
Strings are hashable



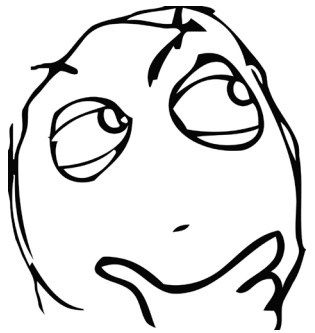
Hand-wavy definition of a hash function: A function $h: T \mapsto \mathbb{N}$ will be called a **hash function for type T**

```
echo $ages["Sravanthi"];
```

Strings are hashable



Hand-wavy definition of a hash function: A function $h: T \mapsto \mathbb{N}$ will be called a **hash function for type T**



It looks like a good idea to find an h that achieves **good randomization** of input keys over **available buffer indices…**

```
echo $ages[“Sravanthi”];
```

Question

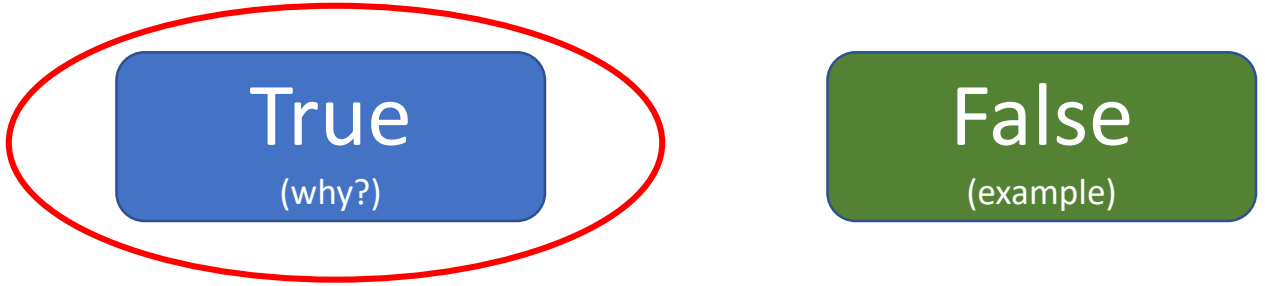
- True or False: Only immutable data types can be hashable

True
(why?)

False
(example)

Question

- True or False: Only immutable data types can be hashable



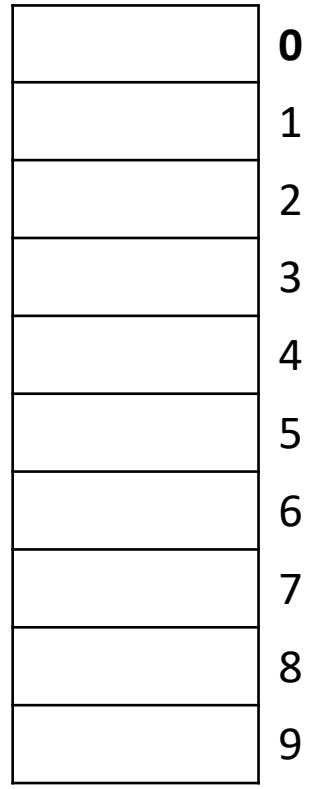
- Allowing mutation of a key after you use it breaks search.
- No guarantee of correctness = guarantee of non-usefulness

If memory were infinite…

- There would be no need for anything but huge arrays.
- Constant access everywhere
- But memory, as we all know, is not infinite
  - CPU Registers->Cache->RAM->Disk
  - SSDs have helped a lot to make that last part faster for seeking, reading and writing.

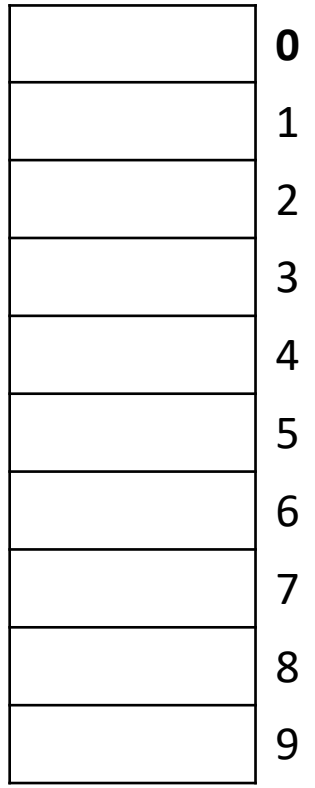
Storage smaller than the available data ☹

Caroline, Jordan, Damien, Phong,
Muhammad, Chris, Dorothy,
D'Angelo, Mark, Julie, Sabrina, Jacqueline,
Connie, Melanie,
Trisha, Fred, Fabio, Hans, Harry, …



Storage smaller than the available data ☹

Caroline, Jordan, Damien, Phong,
Muhammad, Chris, Dorothy,
D'Angelo, Mark, Julie, Sabrina, Jacqueline,
Connie, Melanie,
Trisha, Fred, Fabio, Hans, Harry, …



Only good news: you won't need to store the entire data in memory.
Problem: By pigeonhole principle, sooner or later I am guaranteed a collision ***: An assignment of two different keys at the same index.***

The two elements of hashing

1. Find a good hash function for your data type
   - "Good" means:
     a) It minimizes collisions (so it uniformly distributes keys over the array)
     b) It is easy to compute (since we'll need at least one computation for every operation)
     c) It is consistent: `k1.equals(k2)` implies `k1.hashCode() == k2.hashCode()`
        - `Object.hashCode` declares this requirement for all overridings!

The two elements of hashing

1. Find a good hash function for your data type
   - "Good" means:
     a) It minimizes collisions (so it uniformly distributes keys over the array)
     b) It is easy to compute (since we'll need at least one computation for every operation)
     c) It is consistent: `k1.equals(k2)` implies `k1.hashCode() == k2.hashCode()`
        - `Object.hashCode` declares this requirement for all overridings!

Example: Hashing **Social Security Numbers**

The two elements of hashing

1. Find a good hash function for your data type
   - "Good" means:
     a) It minimizes collisions (so it uniformly distributes keys over the array)
     b) It is easy to compute (since we'll need at least one computation for every operation)
     c) It is consistent: `k1.equals(k2)` implies `k1.hashCode() == k2.hashCode()`
        - `Object.hashCode()` declares this requirement for all overridings!
2. Resolve collisions in your hash table
   - Even with the best hash function, collisions will happen
   - Separate chaining
   - Open Addressing
     - Linear Probing
     - Quadratic / Exponential Probing
     - Double hashing
   - Hybrid methods

The true story

- For most common data types, Java will give you a good `hashCode()` implementation.
- As long as you define a `dict()` with a hashable type, Python will give you an **excellent** hashcode completely transparently.
- In PHP, once you call $age[“Sravanthi”], again completely transparently, a hash function for strings will be used.

The true story

- For most common **immutable** data types, Java will give you a good `hashCode()` implementation.
- As long as you define a `dict()`with a hashable type, Python will give you an **excellent** hashcode completely transparently.
- In PHP, once you call $age[“Sravanthi”], again completely transparently, a hash function for strings will be used.
- Take-home message: some of the things we discuss about are solved in a modern language.
- But! Defining your own types will require re-thinking the default `hashCode()`!
- If you can be given any sort of control over which collision resolution strategy you should employ, data statistics (data type size, which bits typically change from one entry to another, etc) can be leveraged to help you choose!