

Hash Tables

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 16, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2005–2020 Glenn G. Chappell

Some material contributed by Chris Hartman

Review

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
 - Access items [single item: retrieve/find, all items: traverse].
 - Add new item [insert].
 - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Unit Overview

Tables & Priority Queues

Major Topics

- ✓ ■ Introduction to Tables ← Lots of lousy implementations
 - ✓ ■ Priority Queues
 - ✓ ■ Binary Heap Algorithms
 - ✓ ■ Heaps & Priority Queues in the C++ STL
 - ✓ ■ 2-3 Trees
 - ✓ ■ Other self-balancing search trees
 - Hash Tables
 - Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
 - Tables in the C++ STL & Elsewhere
- Idea #1: Restricted Table
- Idea #2: Keep a tree balanced
- Idea #3: Magic functions

Review

Introduction to Tables

A **Table** allows for arbitrary key-based look-up.

Three single-item operations: retrieve, insert, delete by key.

A Table implementation typically holds **key-value pairs**.

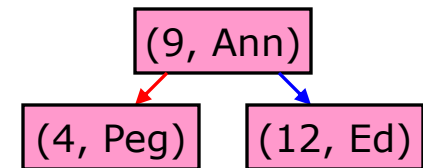
Table

Key	Value
12	Ed
4	Peg
9	Ann

Inefficient Implementations

(12, Ed) (4, Peg) (9, Ann)

(4, Peg) (9, Ann) (12, Ed)



Three ideas for efficient implementations:

1. Restricted Table → Priority Queues
2. Keep a tree balanced → Self-balancing search trees
3. Magic functions → Hash Tables

Overview of Advanced Table Implementations

We cover the following advanced Table implementations.

- Self-balancing search trees
 - To make things easier, allow more children (?):
 - ✓ ■ **2-3 Tree**
 - Up to 3 children
 - ✓ ■ **2-3-4 Tree**
 - Up to 4 children
 - ✓ ■ **Red-Black Tree**
 - Binary Tree representation of a 2-3-4 Tree
 - Or back up and try for a strongly balanced Binary Search Tree again:
 - ✓ ■ **AVL Tree**
- Alternatively, forget about trees entirely:
 - **Hash Table**
- Finally, “the Radix Sort of Table implementations”:
 - **Prefix Tree**

Idea #2:
Keep a tree balanced

Later, we will cover other
self-balancing search
trees: B-Trees, B+ Trees.

Idea #3:
Magic functions

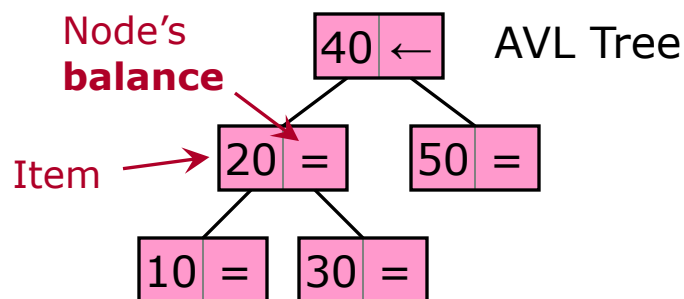
We looked at four kinds of **self-balancing search trees**:

2-3 Trees, 2-3-4 Trees, Red-Black Trees, and AVL Trees.

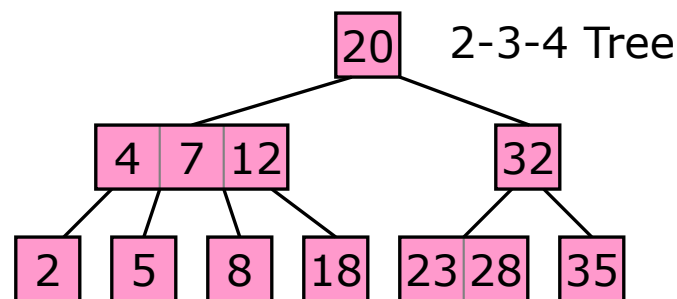
All of these have:

- $\Theta(\log n)$ retrieve, insert, & delete.
- $\Theta(n)$ traverse (sorted).

AVL Tree: strongly balanced Binary Search Tree in which each node holds its **balance**: left-high, right-high, or even.



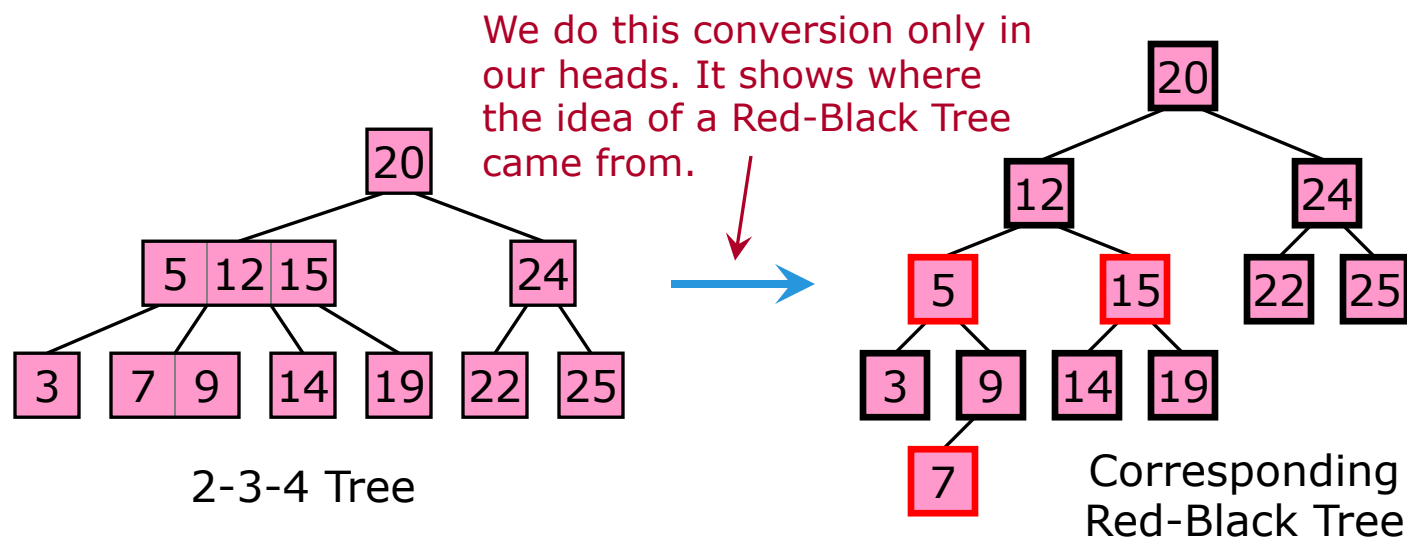
2-3-4 Tree: exactly like a 2-3 Tree, except that 4-nodes are allowed.



Red-Black Tree: Binary Search Tree representation of 2-3-4 Tree.

- Each node in a Red-Black Tree is either **red** or **black**.
 - Each **black node** corresponds to a 2-3-4 Tree node.
 - Red nodes** are extras needed since each node holds only one item.

A Red-Black Tree may not be strongly balanced. However, each path from the root to a leaf hits the same number of **black nodes**, and no more than one **red node** for each **black node**. As a result, Red-Black Trees have logarithmic height.



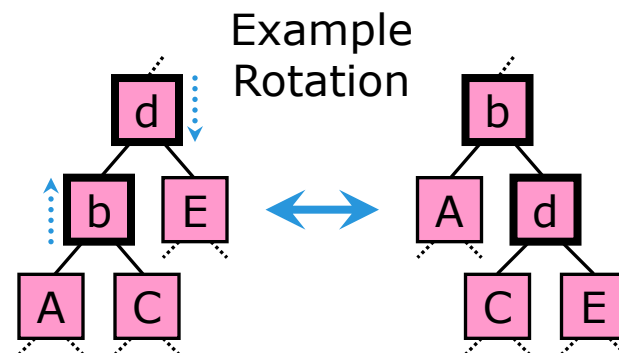
Summary of Ideas Behind the Algorithms

Retrieve & (Sorted) Traverse

- For Red-Black Trees & AVL Trees, use the Binary Search Tree algorithms (traverse = inorder traverse).
- For 2-3 Trees & 2-3-4 Trees, use straightforward generalizations of Binary Search Tree algorithms.

Insert & Delete

- We covered the 2-3 Tree insert & delete algorithms in detail.
- The 2-3-4 Tree algorithms are similar, generally requiring fewer operations.
- Algorithms for Red-Black Trees and AVL Trees are based on **rotations**.



Generally, the Red-Black Tree is agreed to have the best *overall* performance, for in-memory datasets with many insert & delete operations, when worst-case performance is important.

A Red-Black Tree, or some variant, is the usual implementation for `std::map`.

AVL Trees are used less often, but they do have their niche.

2-3 Trees and 2-3-4 Trees are fine data structures. However, they are basically never used, since, in all use cases, some other data structure is a better choice.

We covered 2-3 Trees and 2-3-4 Trees in order to give an idea of where Red-Black Trees* came from and what they are like, without having to cover all the details.

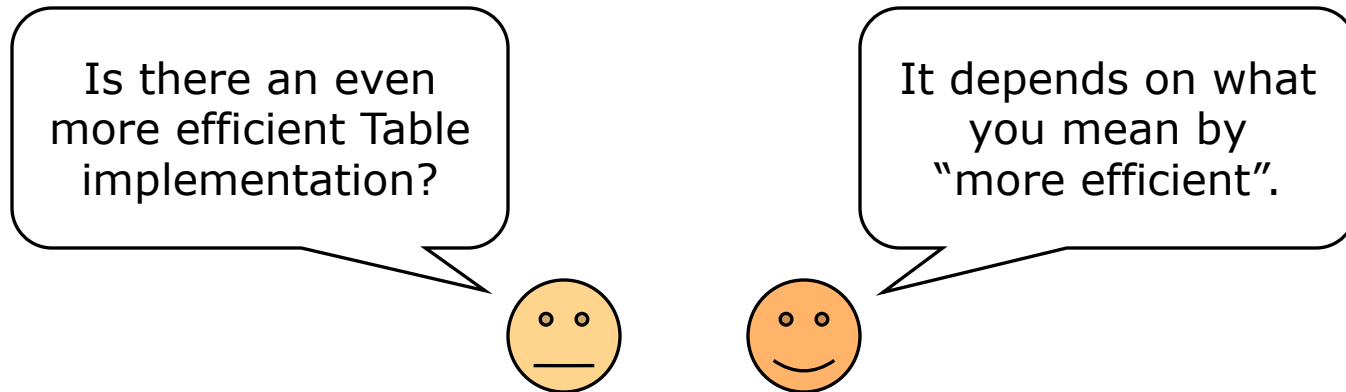
*Also B-Trees & B+ Trees, when we get to them.

Hash Tables

Hash Tables

Introduction [1/4]

Self-balancing search trees allow the three primary Table operations—retrieve, insert, and delete by key—to be $\Theta(\log n)$.



Recall idea #3 for designing an efficient structure to hold an associative dataset: *magic functions*.

Let's see what it leads to ...

Hash Tables

Introduction [2/4]

Consider a Table stored as an unsorted array of key-value pairs.

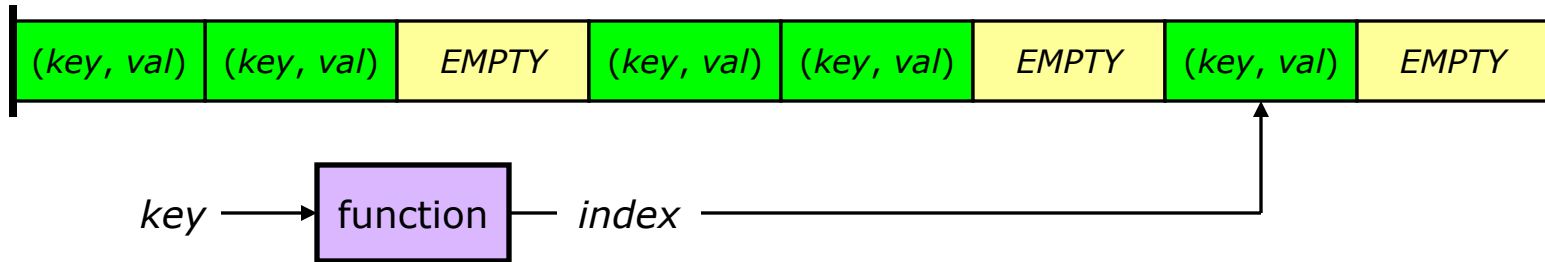
(key, val)	(key, val)	EMPTY	(key, val)	(key, val)	EMPTY	(key, val)	EMPTY
------------	------------	-------	------------	------------	-------	------------	-------

- *Delete by index* can be very fast, if we allow gaps in the data.
 - To delete an item at a given index, just mark it *EMPTY*. Constant time.
- But to do *Table delete*—by key—we must first search by key.
- *Insert by key* would appear to be fast, if we allow duplicate keys.
 - Constant time if no reallocation; generally, amortized constant time.
- But if our dataset does not allow for duplicate keys, then insert requires search by key, to find any existing duplicate key.
- And of course retrieve (by key) requires doing a search by key.
- Unfortunately, search by key is **slow**. We would use Sequential Search: $\Theta(n)$ —**or worse**, due to the *EMPTY* spots.

So speeding up search by key might make *everything* fast.

Hash Tables

Introduction [3/4]



What if we had a magic function that, given a key, returned the index at which the key is stored?

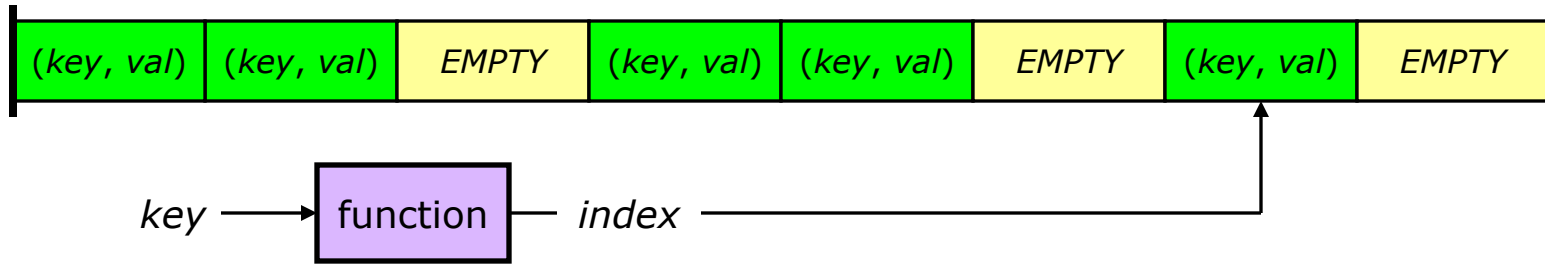
Then all three primary Table operations—retrieve, insert, and delete by key—would be (amortized?) constant time!

Alas, such a function is generally impractical. In many typical use-cases, it is actually *impossible*. (What if there are more possible keys than array locations?)

But we can often find a function that is *almost* what we want.

Hash Tables

Introduction [4/4]



How do we find a magic *almost*-perfect-key-finder function?

[Insert high-quality dramatic production here]

Conclusion: we do not need magic, only *consistency*. For a particular key, the function must always output the same index. The reason the function “knows” where to find an item, is that we asked it where to put the item, when it was inserted.

Data structures based on this idea are called *Hash Tables*.

Hash Tables

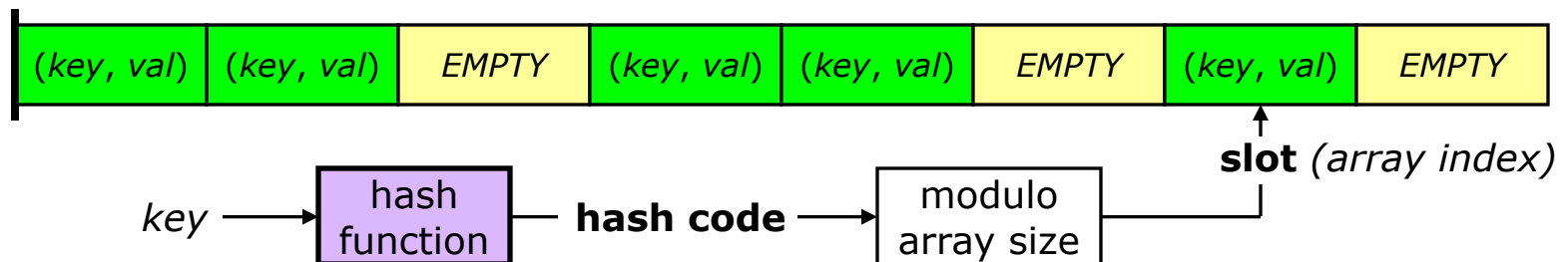
Definitions [1/2]

A **hash function** is a function that “wants to be” our perfect key finder. It takes a key and returns a number: the **hash code**.

- When we pass a key to a hash function, we are **hashing** the key.
- Typically, the function *messes up* its input, so that the output bears little resemblance to the given key. Thus: *hash*.

A **Hash Table** is a data structure in which each item is stored in a location based on a hash code.

- The structure is *something like* an array of key-value pairs (or just keys). Each array index is called a **slot**.
- The hash code may be larger than the size of the array. The slot is typically computed as $\text{hash_code} \% \text{array_size}$.



Hash Tables

Definitions [2/2]

The big problem with Hash Tables: the number of possible keys is usually larger than the number of slots. So there will be distinct keys that get the same slot.

- Key sets can be very large. Consider 20-character strings with printable ASCII characters. There are about 3.6×10^{39} such strings.

A **collision** happens when two items with unequal keys get the same slot. Dealing with this is called **collision resolution**.

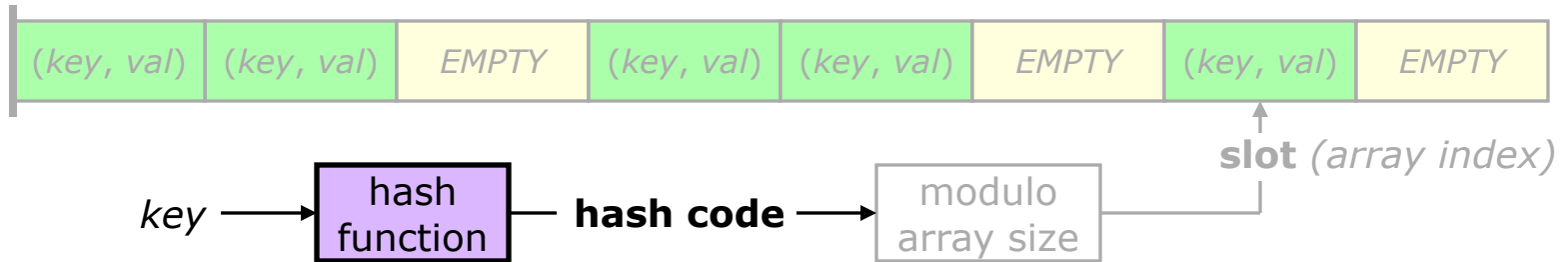
Four issues when putting together a Hash Table:

- What function is used as the hash function?
- How is collision resolution done?
- What if the dataset outgrows the array?
- How efficient is the resulting data structure?

We look at each of these in turn.

Hash Tables

Hash Functions — Properties [1/2]



A hash function *must*:

- Take a key and return a nonnegative integer (**hash code**).
- Be **deterministic**: the output depends only on the input. Passing the same key multiple times results in the same hash code every time.
- Return the same hash code for keys that compare equal ($==$).

Consistency
requirement
mentioned
previously

A *good* hash function:

- Is fast.
- Spreads out its results evenly over the possible output values.
- Turns patterns in its input into unpatterned output.

Hash Tables

Hash Functions — Properties [2/2]

The hash function is typically separate from the Hash Table implementation.

- It may be specified by the client code, when the key type is specified.
- It is possible to design a good hash function without knowing a great deal about the Hash Table implementation.

However, the Hash Table implementation will place *some* requirements on the hash function:

- How it is called.
- What it returns.
 - These days, a hash code is typically a 32-bit or 64-bit unsigned integer.
- How exactly the client code specifies a custom hash function.
 - Template/ctor parameters, specially named member functions, etc.
 - Usually, a **hash function** and an **equality comparison** are both given.

A Hash Table implementation typically comes with hash functions for common key types.

- The C++ Standard Library has hash functions for `int`, `bool`, `char`, `double`, `std::string`, `std::shared_ptr`, and other types.

But not *all* possible key types!

There is no way to generate hash functions automatically for all possible key types.

So if you provide your own type for the keys, then you provide a hash function and an equality comparison.

Puzzle

1. Come up with an idea for generating hash functions automatically.
2. Figure out why this idea will not work for some key types.

This issue does not apply to client-provided types used as *associated values*. Only the key needs to be hashed.

Hash functions for standard types are typically accessible by applications. So, for example, if you need to hash a type that is a wrapper around an `int`, then you can easily write your own hash function that calls the standard hash function for `int`.

In all cases, while you might write a hash function yourself, you probably should not design the technique it uses. Designing a good general-purpose hash function requires some expertise and a lot of time. Designing a hash function that is better than those already known is a *huge* challenge.

There are a number of top-notch hash functions to be found on the net. If you need a custom hash function, then do a little research, grab an implementation, and use it!

Hash Tables

Hash Functions — In Practice [3/4]

For example, here is a hash function using a variant of *MurmurHash* [Austin Appleby 2008]. Keys & hashcodes are 32-bit unsigned integers.

```
uint32_t hash(uint32_t key)                // Based on MurmurHash3
{
    const uint32_t salt = 0xdeadbeefU;    // Value chosen by GGC
    uint32_t h = key * 0xcc9e2d51U;
    h = (h << 15) | (h >> 17);
    h *= 0x1b873593U;
    h ^= salt;
    h = (h << 13) | (h >> 19);
    h = (h * 5) + 0xe6546b64U;
    h ^= 4 ^ (h >> 16);
    h *= 0x85ebca6bU;
    h ^= h >> 13;
    h *= 0xc2b2ae35U;
    return h ^ (h >> 16);
}
```

However, the true measure of a good hash function is the resulting application performance *in actual practice*.

For example, you *may* find that the performance of your application is significantly improved if you switch from the hash function on the previous slide to the following.

```
uint32_t hash(uint32_t key)
{
    return key;
}
```

And if that really is the case—as verified by thorough testing—then you may use this hash function without embarrassment.

Hash Tables

Collision Resolution — Overview [1/2]

Collision: when two items with unequal keys get the same slot.

Generally:

- For a well designed hash function, collisions are relatively rare for average data.
- But we typically cannot *guarantee* that there will be few collisions.
- In fact, it is possible that *every* item we insert will be assigned the same index in the array. (Ick!)

Regardless, in a Hash Table that allows arbitrary insertions, we must have a way to deal with collisions.

Hash Tables

Collision Resolution — Overview [2/2]

Collision-resolution methods come in two categories.

Closed Hashing

- In each slot there is at most *one data item*.
- If we get a collision, then we look for another slot.

Open Hashing

- In each slot is a data structure that can store *multiple data items*.
- Each such data structure is called a **bucket**.

How collisions are resolved is the primary design decision involved in a Hash Table implementation.

Hash Tables

Collision Resolution — Closed Hashing [1/4]

In **closed hashing**:

- In each slot there is at most *one data item*.
- So the Hash Table is basically an array of data items. However ...
- Each slot can be marked as *EMPTY*.

(key, val)	(key, val)	EMPTY	(key, val)	(key, val)	EMPTY	(key, val)	EMPTY
------------	------------	-------	------------	------------	-------	------------	-------

If there is a collision:

- We are inserting a new item, but the slot we want to store it in is already used by an item with a different key.
- So we find a different slot to store the new item in. We keep looking until an empty slot is found.

Hash Tables

Collision Resolution — Closed Hashing [2/4]

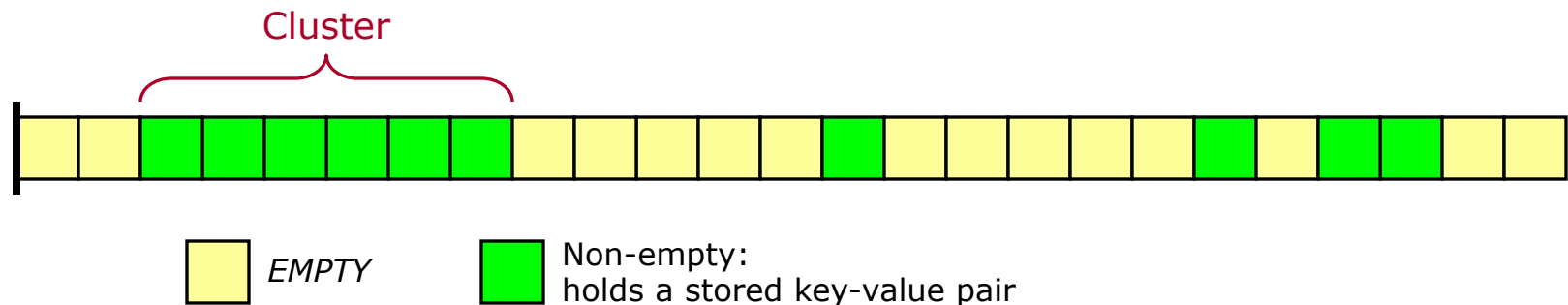
Looking for a given key, or a slot to store a new key, is a **search**.

When doing a search, we check a sequence of slots.

- The first is the slot computed from the hash code.
- Each time we check a slot, we are doing a **probe**.
- The sequence of slots to check is called the **probe sequence**.

The simplest probe sequence is the one in which we look at slot t , then $t+1$, then $t+2$, etc. This is **linear probing**.

- Linear probing tends to form **clusters**, which slow down searches.



To avoid clusters, we can use **quadratic probing**, which has the probe sequence $t, t+1^2, t+2^2, t+3^2$, etc.

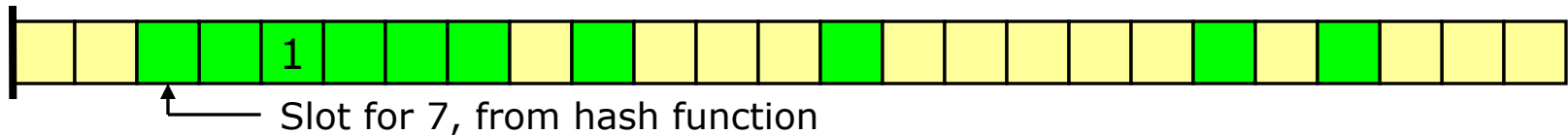
Some techniques involve using a secondary hash function when a collision occurs.

- These generally go under the name of **double hashing**.
- For example, do a variation of linear probing with a step size other than 1. The step size is given by the second hash function.
- Or, just use the second hash function to give a second slot, after which one of the simpler probe sequences is used.

Hash Tables

Collision Resolution — Closed Hashing [4/4]

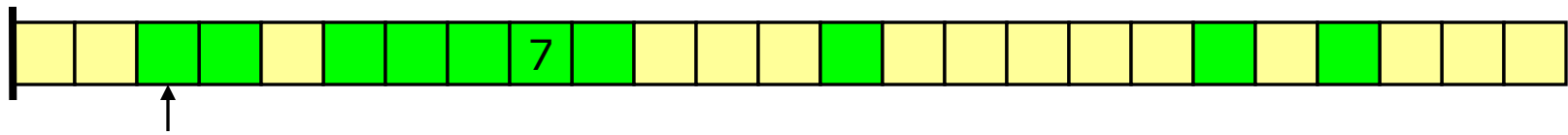
Trickiness in closed hashing: how to be sure a key is *not* present?



In above Hash Table, retrieve 7 (not present). Use linear probing.

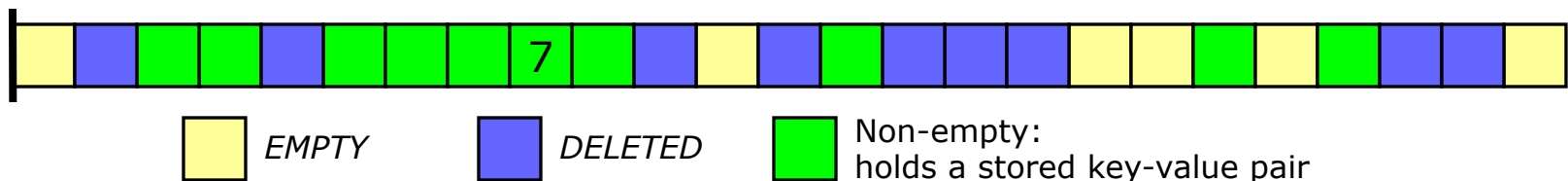
- Begin at arrow, look until we find an *EMPTY*. Return NOT FOUND.

Now insert 7 and then delete 1.



Again, retrieve 7. The above strategy gives NOT FOUND. ☹

Solution. Allow *DELETED* marks. Stop only on key found or *EMPTY*.



Hash Tables

Collision Resolution — Open Hashing

In **open hashing**:

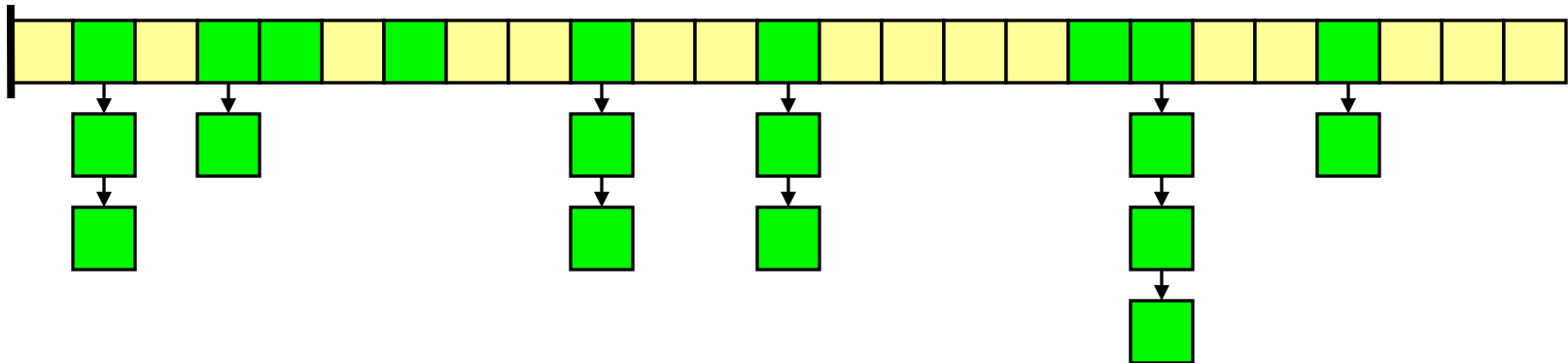
- In each slot is a data structure that can store *multiple data items*.
- Each such data structure is called a **bucket**.
- A search always stays within a single bucket.

If there is a collision:

- Insert a new item in the proper bucket.

Buckets are virtually always Singly Linked Lists.

To search for a key: do a Sequential Search in its bucket.



Hash Tables

TO BE CONTINUED ...

Hash Tables will be continued next time.