Hash Tables continued
Prefix Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, November 20, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL
- ✓ 2-3 Trees
- ✓ Other self-balancing search trees
- (part) Hash Tables
- Prefix Trees
- Tables in the C++ STL & Elsewhere

Review

Where Are We? — The Big Challenge

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.

Note the three primary single-item operations: **retrieve**, **insert**, **delete**. We will see these over & over again.

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

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

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

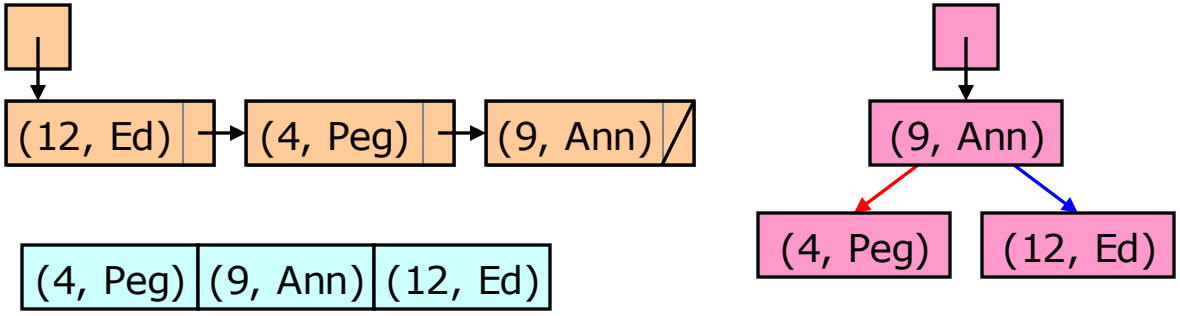
Three single-item operations: retrieve, insert, delete by key.

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

Table

Key	Value
12	Ed
4	Peg
9	Ann

Inefficient Implementations



Three ideas for improving efficiency:

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

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables ← Several lousy implementations
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL

  Idea #1: Restricted Table
- ✓ 2-3 Trees
- ✓ Other self-balancing search trees

  Idea #2: Keep a tree balanced
- (part) Hash Tables

  Idea #3: Magic functions
- Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
- Tables in the C++ STL & Elsewhere

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**

Idea #2:
Keep a tree balanced

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

- Alternatively, forget about trees entirely:

(part)
- **Hash Table**

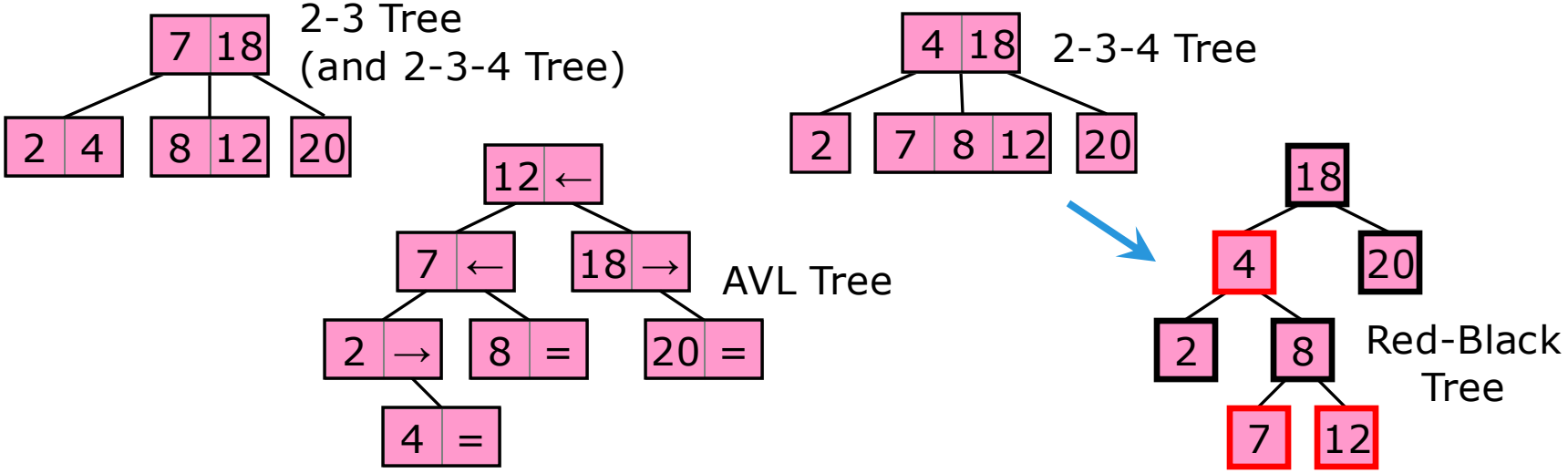
Idea #3:
Magic functions

- Finally, "the Radix Sort of Table implementations":
  - **Prefix Tree**

Review
2-3 Trees, Other Self-Balancing Search Trees

The **self-balancing search trees** include 2-3 Trees, 2-3-4 Trees, Red-Black Trees, AVL Trees, and other kinds.

- All are generalizations of Binary Search Trees.
- The ones we covered all allow for Table implementations with logarithmic-time retrieve/insert/delete by key.



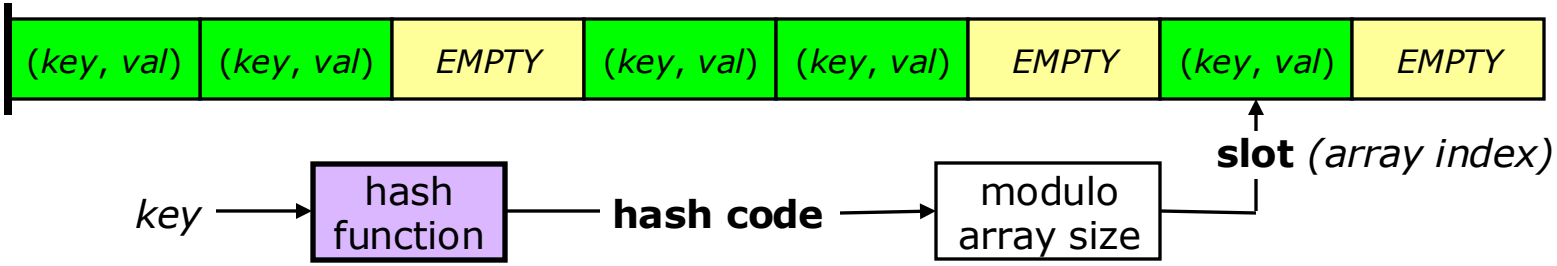
Generally, the Red-Black Tree is agreed to have the best *overall* performance, for in-memory datasets with many insert & delete operations, when worst-case performance is important.

Review
Hash Tables [1/5]

A **Hash Table** is a Table implementation that stores key-value pairs in an unsorted array. Array indices are **slots**.

- A key's slot is computed using a **hash function**.
- An array location can be *EMPTY*.

Or just keys, if there are no associated values.

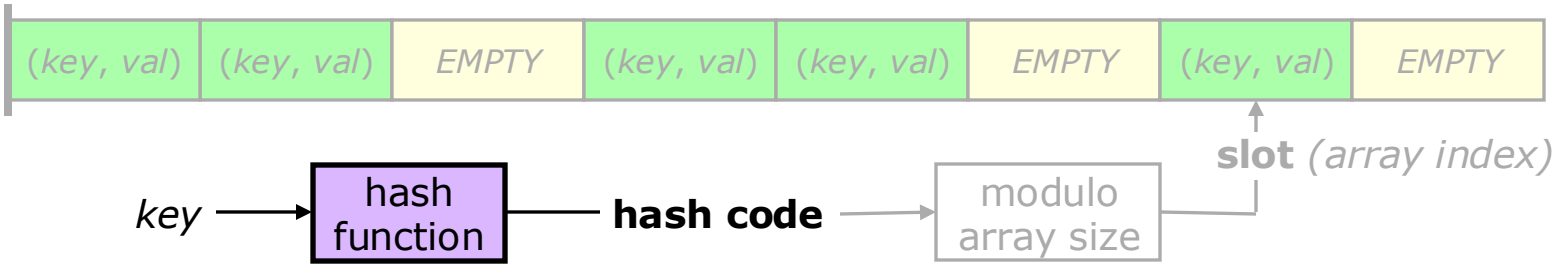


Collision: when an item gets a slot that already holds an item. The *possibility* of collisions is typically an unavoidable problem; there are often far more possible keys than slots.

Needed

- Hash function (typically separate from Hash Table implementation).
- **Collision-resolution** method.

Review
Hash Tables [2/5]



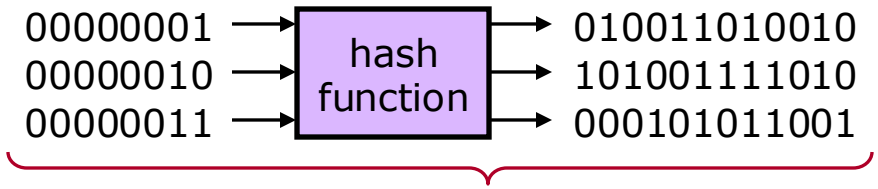
A hash function *must*:

- Take a key and return a nonnegative integer (**hash code**).
- Be **deterministic**: output depends only on input.
  A particular key always gives the same hash code.
- Return the same hash code for equal (==) keys.

Consistency requirement

A *good* hash function:

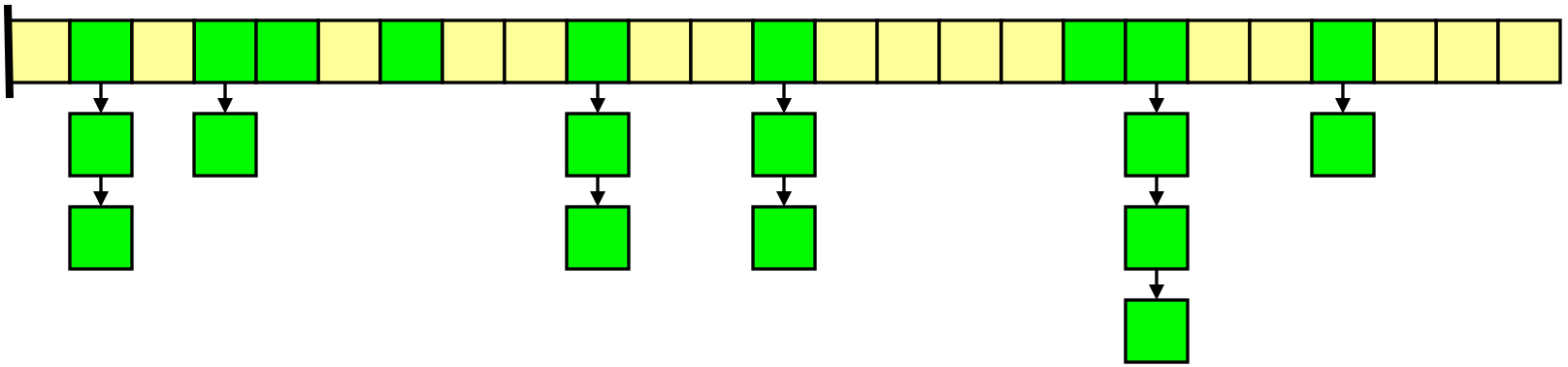
- Is fast.
- Spreads its results evenly over the possible output values.
- Turns patterns in its input into unpatterned output.



Review
Hash Tables [3/5]

Collision resolution methods, category #1: **Open Hashing**

- Each array item holds a data structure (a **bucket**) that can store multiple key-value pairs.
- Buckets are virtually always Singly Linked Lists.
- To find a key, determine which bucket to look in based on the hash code. Do a Sequential Search in that bucket.



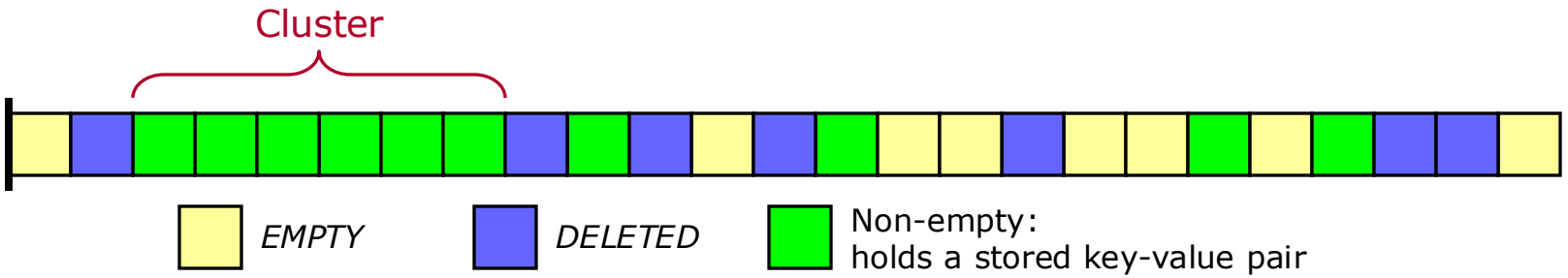
Review
Hash Tables [4/5]

Collision resolution methods, category #2: **Closed Hashing**

- Each array item holds one key-value pair, or a mark indicating *EMPTY* or *DELETED*.
- To find a key, begin at the slot given by the hash function, and **probe** in a sequence of slots: the **probe sequence**. End when desired key or *EMPTY* slot is found.

Some probe sequences (*t* is initial slot given by hash function):

- **Linear probing**: *t*, *t*+1, *t*+2, *t*+3, etc., wrapping at array end.
  - Tends to form **clusters**, which slow things down. ☹
- **Quadratic probing**: *t*, $t+1^2$, $t+2^2$, $t+3^2$, etc.
- **Double hashing**: Base probe sequence on a 2nd hash function.



Review
Hash Tables [5/5]

All Hash Table implementations that allow insertion will suffer poor performance when the data structure gets too full.

When the number of items gets too high, we remake the Hash Table, doing a reallocate-and-copy to a larger array—as we did with resizable smart arrays—and calling the hash function for each key present. This is called **rehashing**.

Two Questions

1. How do we decide when to do rehashing?
2. How does periodic rehashing affect Hash Table performance?

We look at these after discussing basic Hash Table efficiency.

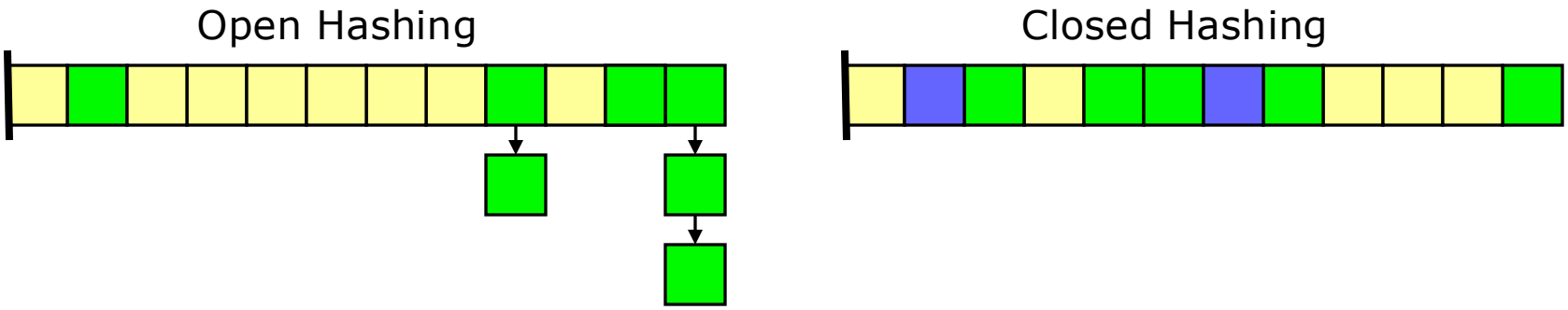
Hash Tables

continued

How efficient are Hash Tables?

Let's look at Hash Tables where duplicate keys are not allowed:

- A Hash Table using open hashing, in which buckets are Linked Lists.
- A Hash Table using closed hashing.



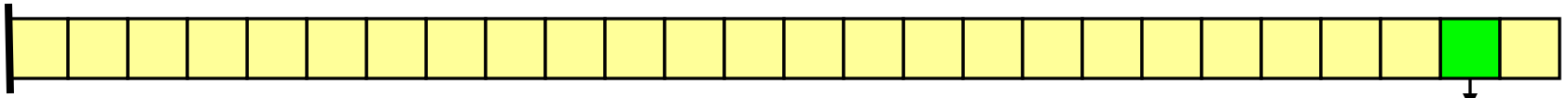
For now, assume there is no rehashing—an unrealistic assumption if we can do arbitrary insertions, but only a temporary one.

The time taken for *retrieve*, *insert*—assuming no rehashing—or *delete* is essentially the time required to **search** by key (right?).

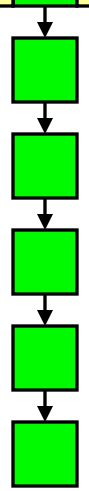
So: *how fast can we do a search by key?*

Hash Tables
Efficiency — Worst Case

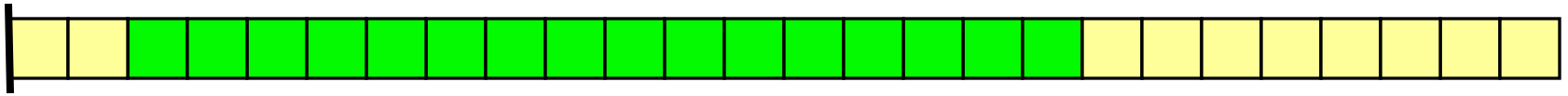
In the worst case, every item inserted gets the same slot.



With open hashing, this essentially turns our Table into a Linked List. So a search is linear-time.



With closed hashing, a search may require probing every stored item. Again, linear-time.



Conclusion. In a Hash Table, search by key is linear-time.

Thus, *retrieve*, *insert*, and *delete* are linear-time (worst case!).

What about the average case?

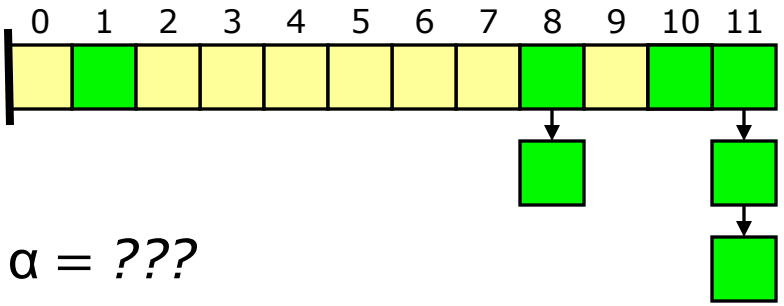
Hash Tables
Efficiency — Load Factor [1/2]

Average-case performance of a Hash Table can be analyzed based on its **load factor**: α = (# of keys present) / (# of slots).

Lower-case Greek letter **alpha**

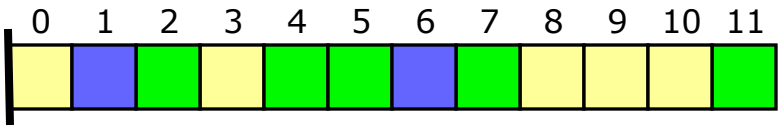
Find the load factors.

Open Hashing



α = *???*

Closed Hashing



α = *???*

Answers on next slide.

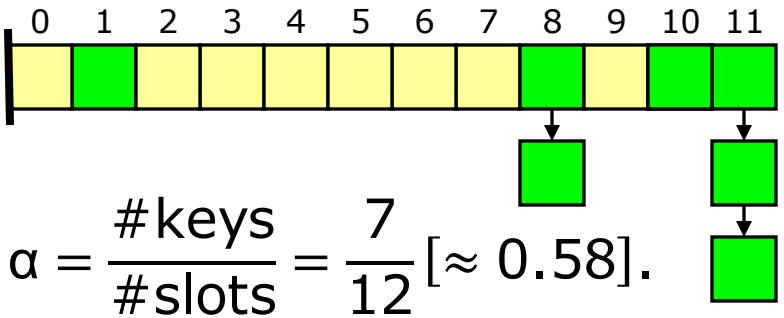
Efficiency — Load Factor [2/2]

Average-case performance of a Hash Table can be analyzed based on its **load factor**: α = (# of keys present) / (# of slots).

Lower-case Greek letter **alpha**

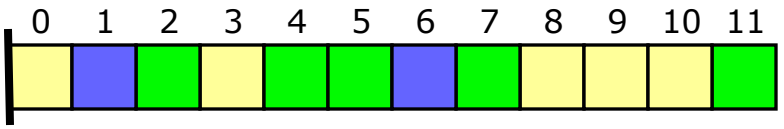
Answers

Open Hashing



$$\alpha = \frac{\#\text{keys}}{\#\text{slots}} = \frac{7}{12} [\approx 0.58].$$

Closed Hashing



$$\alpha = \frac{\#\text{keys}}{\#\text{slots}} = \frac{5}{12} [\approx 0.42].$$

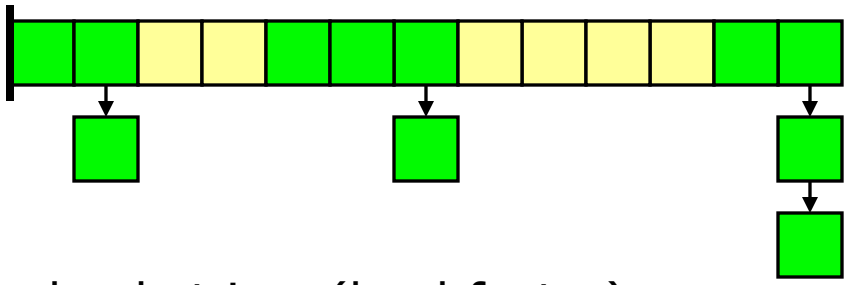
Hash Tables keep load factors *small* (usually well below 1).

In the following slides, we assume α is significantly less than 1. For example, we might require that α < 2/3.

Efficiency — Average Case: Open Hashing

Consider open hashing (buckets are Linked Lists, no duplicate keys, α significantly less than 1).

Search worst case: linear time.



Search average case:

- The average number of items in a bucket is α (load factor).
- Thus, the average number of comparisons required for a search resulting in NOT FOUND is α.
- The average number of comparisons required for a search resulting in FOUND is less than 1 + α/2.
  - Average search looks at item found + half of other items in bucket:

$$1+\frac{1}{2}\cdot\frac{n-1}{b} = 1+\frac{n/b}{2}-\frac{1}{2b} < 1+\frac{\alpha}{2}$$

α

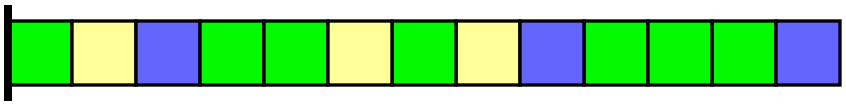
Small negative number

If α < 1, then these are both less than 1.5. That's *fast*!

Hash Tables
Efficiency — Average Case: Closed Hashing

Consider closed hashing (no duplicate keys, α significantly less than 1).

Search worst case: linear time.



Search average case:

- Comparisons for linear probing:
  - NOT FOUND: $(1/2)[1+1/(1-\alpha)]^2$.
  - FOUND: $(1/2)[1+1/(1-\alpha)]$.
- Comparisons for quadratic probing:
  - NOT FOUND: $1/(1-\alpha)$.
  - FOUND: $-\ln(1-\alpha)/\alpha$.

The analysis is complicated.
It actually took several years
to be properly figured out.
We will not give details.

Important to know:

- Larger α means a slower average case for search.
- Requiring $\alpha < r$, for r a fixed number between 0 and 1 (so $r \neq 1$), gives a fast constant-time average case for search.

Hash Tables
Efficiency — With Rehashing

Summary, so far. For both open & closed hashing:

- Worst-case performance for *retrieve*, *insert*, *delete*: linear time.
- Average-case performance for *retrieve*, *delete*: constant time. Also for *insert*—not counting the time required for rehashing.

But of course we will need to do rehashing occasionally.

The *Two Questions* from a few slides back

1. How do we decide when to do rehashing?
   - The usual trigger for rehashing: the load factor becomes too large.

2. How does periodic rehashing affect Hash Table performance?
   - The effect of rehashing on Hash-Table efficiency is similar to that of reallocate-and-copy on a resizable array. Again, when we increase the array size, we need to increase it by a *constant factor*.
   - When we count rehashing, *insert* in a well written Hash Table will have an average case of amortized constant time.
   - Note the *double average*! Average data & amortized time.

Efficiency Comparison (duplicate keys not allowed)

Idea #1 Idea #2 Idea #3

	Priority Queue using Heap	Self-Balancing Search Tree	Hash Table: worst case	Hash Table: average case
Retrieve	Constant*	Logarithmic	Linear	Constant
Insert	Amortized** logarithmic	Logarithmic	Linear	Amortized constant***
Delete	Logarithmic*	Logarithmic	Linear	Constant

*Priority Queue *retrieve* & *delete* are not Table operations in full generality. Only the item with the highest priority (key) can be retrieved/deleted.

**Logarithmic if enough memory is preallocated. Otherwise, occasional reallocate-and-copy—linear time—may be required. Time per *insert*, averaged over many consecutive *inserts*, will be logarithmic. Thus, *amortized logarithmic time* (which is not a term I expect you to know).

***Hash Table *insert* is constant-time only in a *double average* sense: averaged both over all possible inputs and over a large number of consecutive *inserts*.

Hash Tables
Efficiency — Issues

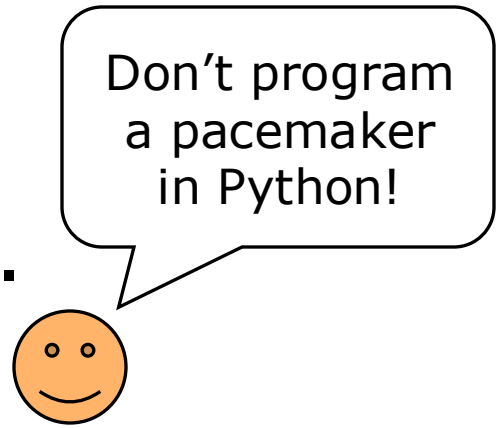
Hash Tables generally have excellent average-case performance, but poor worst-case performance.

Can a **malicious user** force poor Hash Table performance?

- Sometimes. For many applications this matters little, but for others it can matter a great deal. This issue is something to keep in mind.
- A **cryptographic hash function** is one that is extremely difficult to reverse. This makes it hard to force poor performance. However, cryptographic hash functions are time-consuming to compute, which makes them unsuitable for most applications of Hash Tables.

Hash Tables are appropriate for many use cases of Tables. But their worst-case performance can be problematic in some contexts.

When using Hash Tables, do so intelligently.



Prefix Trees

Prefix Trees
Background

In a list of strings, there are often strings that start with the same sequence of characters.

For example, in the list to the right, "dot", "dote", and "doting" all begin with "dot…".

Such strings are said to have a common **prefix**.

dig
dog
dot
dote
doting
eggs

We will use *string* in a general sense, just as when we discussed Radix Sort. A **string** is a sequence; its entries will be called **characters**.

- The words in the above list are strings of letters.
- A nonnegative integer can be regarded as a string of digits.
- Many kinds of data can be regarded as strings of **bits** (0s & 1s).

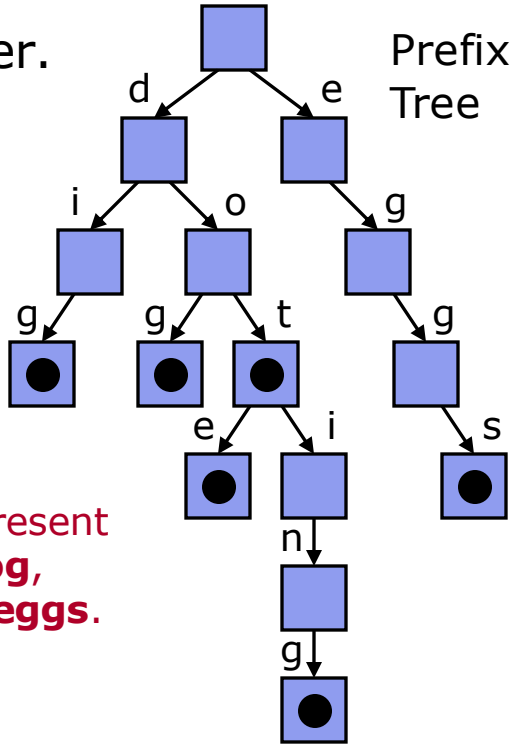
We look at a data structure that can be used to store a Table in which the keys are such strings: a *Prefix Tree*.

Prefix Trees
Definition

A **Prefix Tree** (a.k.a. **Trie***) [René de la Briandais 1959, Edward Fredkin 1960] is a tree used to implement a Table whose keys are strings—in the general sense.

- Each child is associated with a character.
  A node has at most one child for each character.
- A node has:
  - A Boolean—whether it represents a stored key.
  - Child pointers—one for each possible character.
  - The value associated with a key, if needed.
- We can make a Prefix Tree that can store duplicate keys. Do you see how?

"**Trie**" for "re**TRIE**val".
Some say "tree".
Some say "try".
I say "Prefix Tree".



Nodes with dots represent stored keys: **dig**, **dog**, **dot**, **dote**, **doting**, **eggs**.

*Fredkin gets the ~~credit~~ *blame* for the term "Trie".

Prefix Trees
Efficiency

For a Prefix Tree, *retrieve*, *insert*, and *delete* by key all take a number of steps proportional to the **length of a key**.

If key length is considered fixed, then all are constant time!

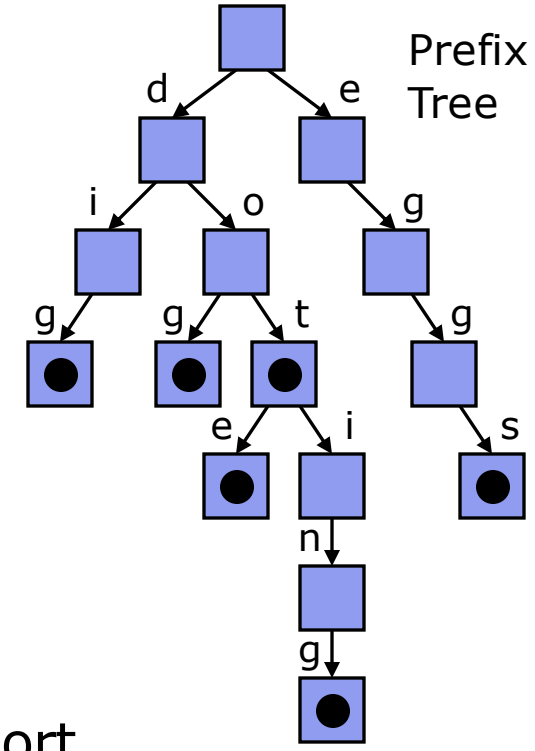
But larger datasets tend to have longer keys.

If C is the size of the character set, and L is the length of a key, then the number of possible keys is $K = C^L$.

Solve for L, and we get $L = \log_C K$.

As we noted when we covered Radix Sort, the **length of a key** is logarithmic in the number of possible keys.

So there is a hidden logarithm, as with Radix Sort.



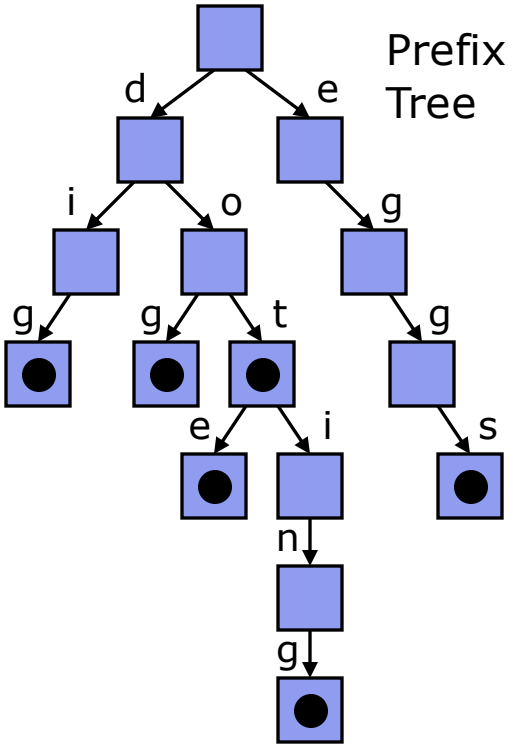
Prefix Trees
Thoughts

A Prefix Tree is a good basis for a Table implementation, when keys are short-ish sequences of characters from a small-ish set.

- Words in a dictionary, ZIP codes, etc. (just like Radix Sort).

Prefix Trees are **easy to implement well**.

- If you feel like writing a Red-Black Tree for production code, you might want to sit down until the feeling goes away.
- But if you feel like writing a Prefix Tree, then go for it (just like Radix Sort).



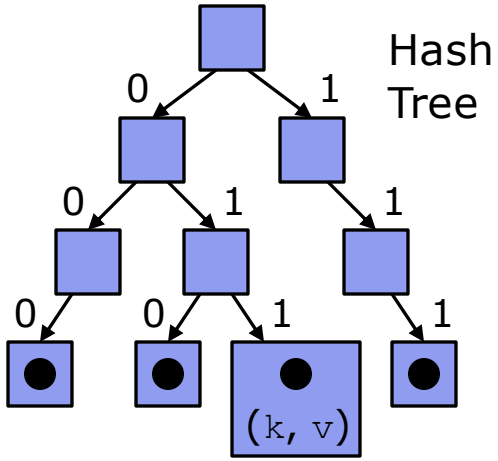
The idea behind Prefix Trees is also used in other data structures. *See the next slide.*

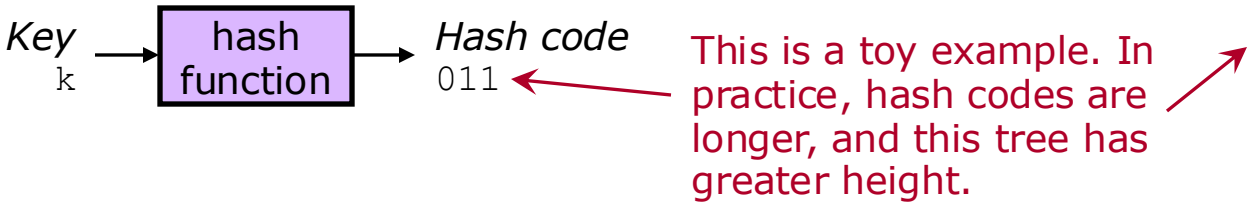
Prefix Trees work only for keys that can be treated as *strings*. ☹

Idea. Using a hash function, hash each key. The hash code, treated as a string of bits (a **bit** is a 0 or 1), is the key for a Prefix Tree. Associated data = original key + the usual data.

Hash Trees are based on this idea.

Here we show how key-value pair (`k`, `v`) would be stored in a Hash Tree. Different keys can have the same hash code, so nodes may hold multiple key-value pairs.





Hash Tree variants have names like HAMT (Hash Array Mapped Trie) and CHAMP (Compressed Hash Array Mapped Prefix tree).