Hash Tables continued
Prefix Trees

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 30, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

Review
Where Are We? — The Big Problem

Our problem for much of the rest of the semester:

- Store: a collection of data items, all of the same type.
- Operations:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

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

Generic containers: those in which client code can specify the type of data stored.

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

} Idea #1: Restricted Table

- ✓ 2-3 Trees
- ✓ Other balanced search trees

} Idea #2: Keep a Tree Balanced

- (part) Hash Tables

} Idea #3: “Magic Functions”

- Prefix Trees
- Tables in various languages

Review
Introduction to Tables

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant???	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #1: Restricted Table

- Perhaps we can do better if we do not implement a Table in its full generality.

Idea #2: Keep a Tree Balanced

- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

Idea #3: "Magic Functions"

- Use an unsorted array of key-data pairs. Allow array items to be marked as "empty".
- Have a "magic function" that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:

- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables

Overview of Advanced Table Implementations

We will cover the following advanced Table implementations.

- Balanced Search Trees
  - Binary Search Trees are hard to keep balanced, so to make things easier we allow more than 2 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 a balanced Binary Tree again:
    - ✓ **AVL Tree**
- Alternatively, forget about trees entirely:
  - (part) **Hash Tables**
- Finally, “the Radix Sort of Table implementations”:
  - **Prefix Tree**

Review
2-3 Trees [1/4]

A Binary-Search-Tree style node is a **2-node**.

- This is a node with 2 subtrees and 1 data item.
- The item’s value lies between the values in the two subtrees.

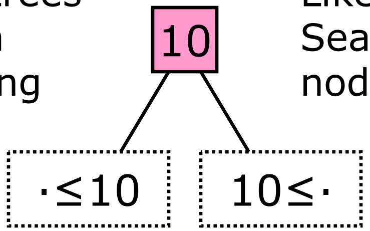
In a “2-3 Tree” we also allow a node to be a **3-node**.

- This is a node with 3 subtrees and 2 data items.
- Each of the 2 data items has a value that lies between the values in the corresponding pair of consecutive subtrees.

Later, we will look at “2-3-4 trees”, which can also have **4-nodes**.

2 subtrees
1 item
ordering

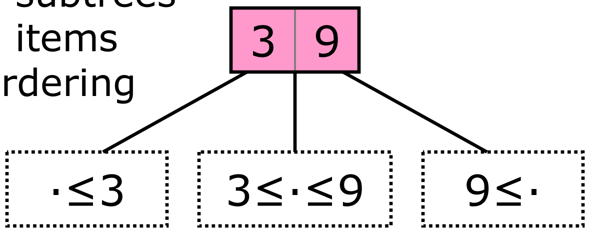
2-node



Like a Binary-Search-Tree node

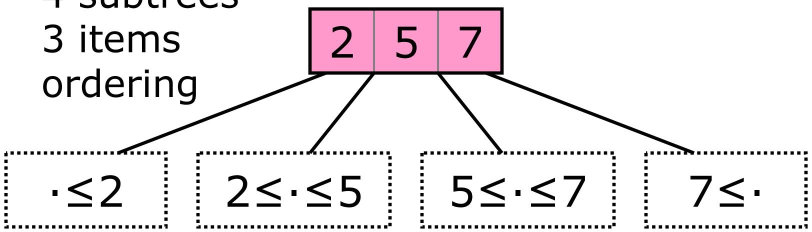
3 subtrees
2 items
ordering

3-node



4 subtrees
3 items
ordering

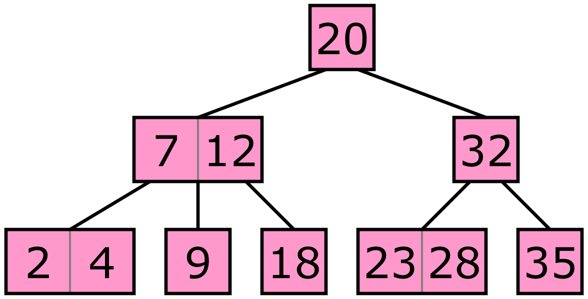
4-node



Review
2-3 Trees [2/4]

A **2-3 Search Tree** (generally we just say **2-3 Tree**) is a tree with the following properties.

- All nodes contain either 1 or 2 data items.
    - If 2 data items, then the first is ≤ the second.
- All leaves are at the same level.
- All non-leaves are either *2-nodes* or *3-nodes*.
    - They must have the associated order properties.



To **retrieve** in a 2-3 Tree:

- Begin at the root, and go down, using the order properties, until the item is found, or clearly not in the tree.

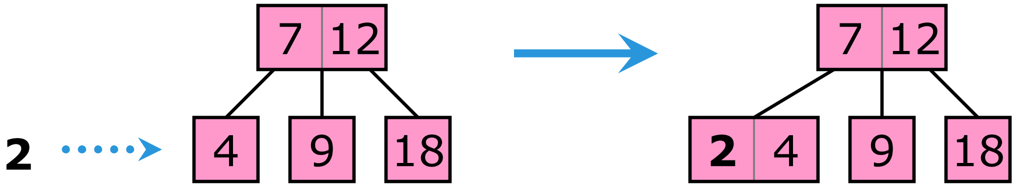
To **traverse** a 2-3 Tree:

- Use the appropriate generalization of inorder traversal.
- Items are visited in sorted order.

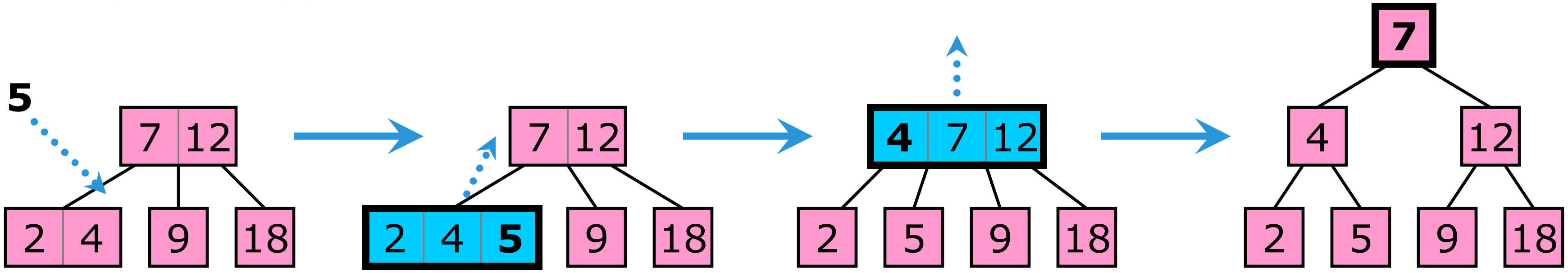
Review
2-3 Trees [3/4]

To **insert** in a 2-3 Tree:

- Find the leaf that the new item should go in.
- If it fits, then simply put it in.



- Otherwise, there is an overfull node. Split it, and move the middle item up, either recursively inserting it in the parent, or else creating a new root.



Review
2-3 Trees [4/4]

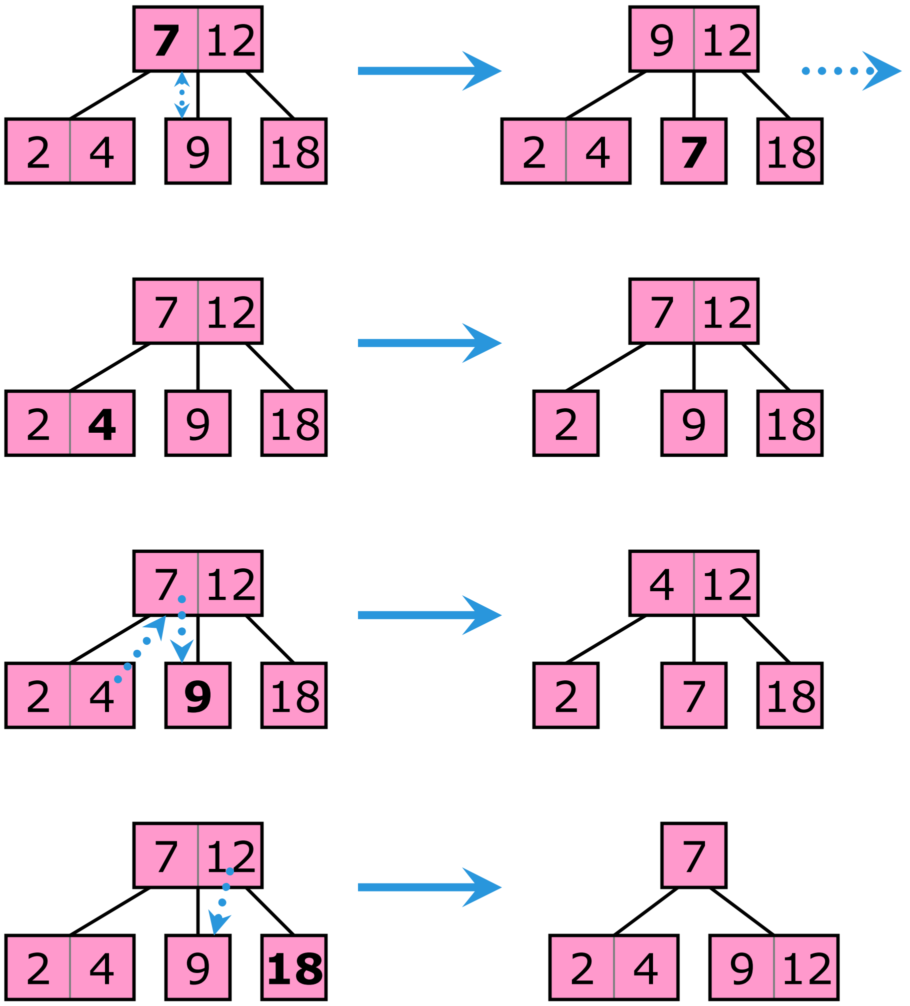
To **delete** in a 2-3 Tree:

- Find the item. If it is not in a leaf, swap with its successor.
- Do the recursive delete-a-leaf procedure.

To delete-a-leaf:

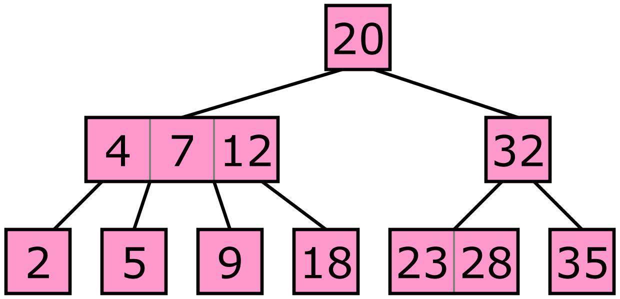
- **Easy Case**: If the item is in a node with another item, simply remove it.
- **Semi-Easy Case**: Otherwise, if the node has a consecutive sibling with two items, do a rotation with the parent.
- **Hard Case**: Otherwise, bring the parent down, combining it with a consecutive sibling.
  - Use recursive delete-a-leaf on the parent.

When doing a recursive "delete-a-leaf" on a non-leaf node, drag along subtrees.



Review
Other Balanced Search Trees [1/4]

In a **2-3-4 Tree**, we also allow 4-nodes.



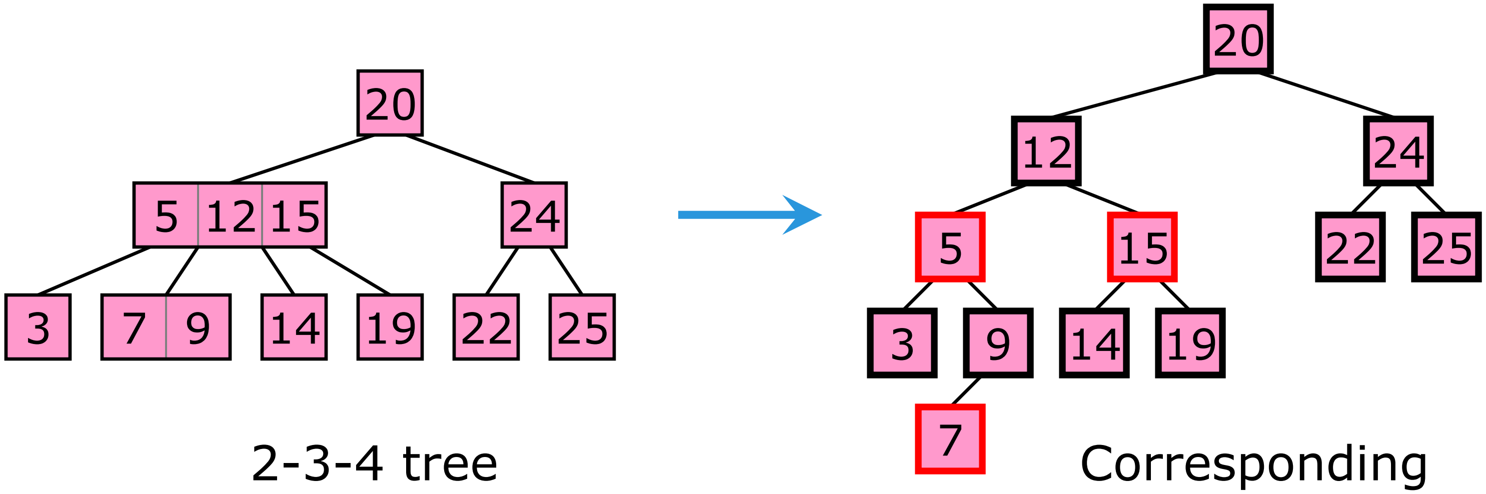
The insert and delete algorithms are not terribly different from those of a 2-3 Tree.

- They are a little more complex.
- And they tend to be a little faster.

Review
Other Balanced Search Trees [2/3]

A very efficient kind of balanced search tree is a **Red-Black Tree**.

- This is a Binary-Search Tree representation of a 2-3-4 tree.
- Each node in a Red-Black Tree is either **red** or **black**.
- Each node in the 2-3-4 Tree corresponds to a **black node**.
- The **red nodes** are the extra ones we need to add.
- Red-Black Trees may not be balanced (in the strict sense). However, each path from the root to a leaf must pass through the same number of **black nodes**.



2-3-4 tree

Corresponding Red-Black Tree

Review
Other Balanced Search Trees [3/3]

All balanced search trees (2-3 Trees, 2-3-4 Trees, Red-Black Trees, AVL Trees, etc.) have:

- *O*(log *n*) retrieve, insert, delete.
- *O*(*n*) traverse (sorted).

Best **overall** performance for **in-memory** data, when we mix up retrieves, inserts, and deletes.

Retrieve & Sorted Traverse

- For Red-Black Trees and AVL Trees, use the B.S.T. algorithms (traverse = inorder traverse).
- For 2-3 Trees and 2-3-4 Trees, use the obvious generalization of the B.S.T. algorithms.

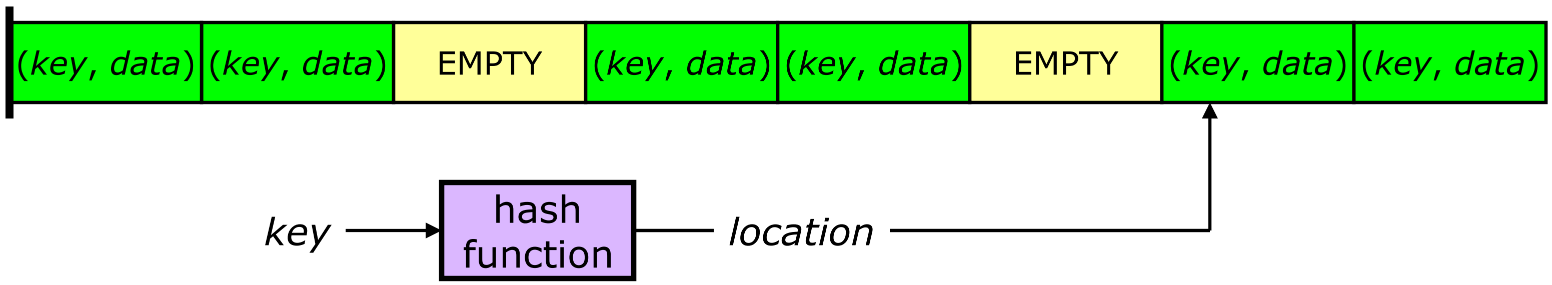
Insert & Delete

- These are more complicated.
- For 2-3 Trees, we looked at the algorithms in some detail.
- The 2-3-4 Trees and Red-Black Trees, the algorithms use the same ideas.

Review
Hash Tables — Introduction

A **Hash Table** is a Table implementation that uses a **hash function** for key-based look-up.

- A Hash Table is generally implemented as an array. The index used is the output of the hash function.



(*key*, *data*)	(*key*, *data*)	EMPTY	(*key*, *data*)	(*key*, *data*)	EMPTY	(*key*, *data*)	(*key*, *data*)

key → hash function — *location*

Needed:

- **Hash function**.
- **Collision resolution** method.
  - **Collision**: hash function gives same output for different keys.

Review
Hash Tables — Good Hash Functions

A hash function **must**:

- Take a valid key and return an integer.
- Be **deterministic**.
  - Its value depends only on its input (the key). Using the same input multiple times results in the same output each time.

A **good** hash function:

- Can be computed quickly.
- Spreads out its results evenly over the possible output values.
  - To help spread out the results, some implementations give the Hash Table a **prime** number of locations.
- Turns patterns in its input into random-looking output.

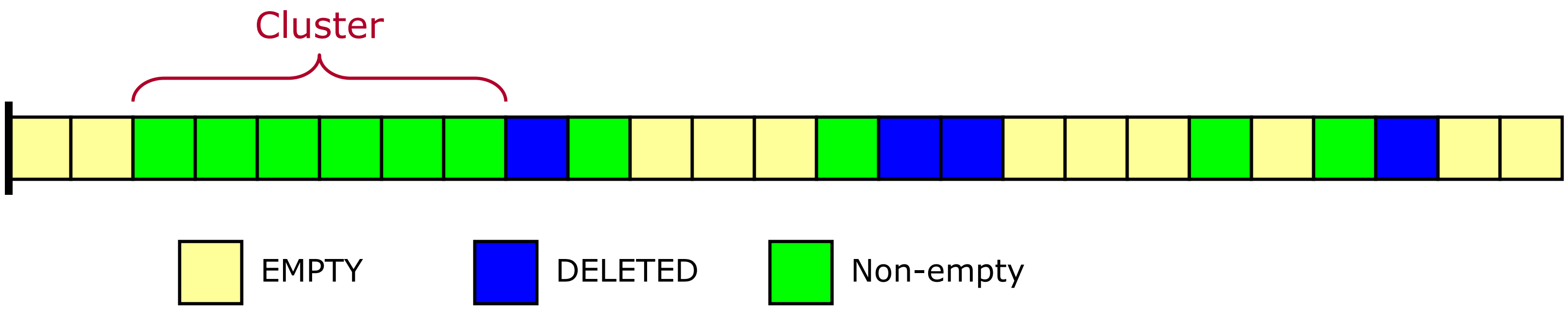
Each key type has its own hash function.

- For client-defined key types, a hash function must be provided by the client.
- Can put different key types, each with its own hash function, in the same Hash Table.
- Hash Table sends the output of the provided hash function through a secondary function (“%”?) to make the output a valid index.

Review
Hash Tables — Collision Resolution [1/2]

Collision Resolution Methods — Type 1: **Open Addressing**

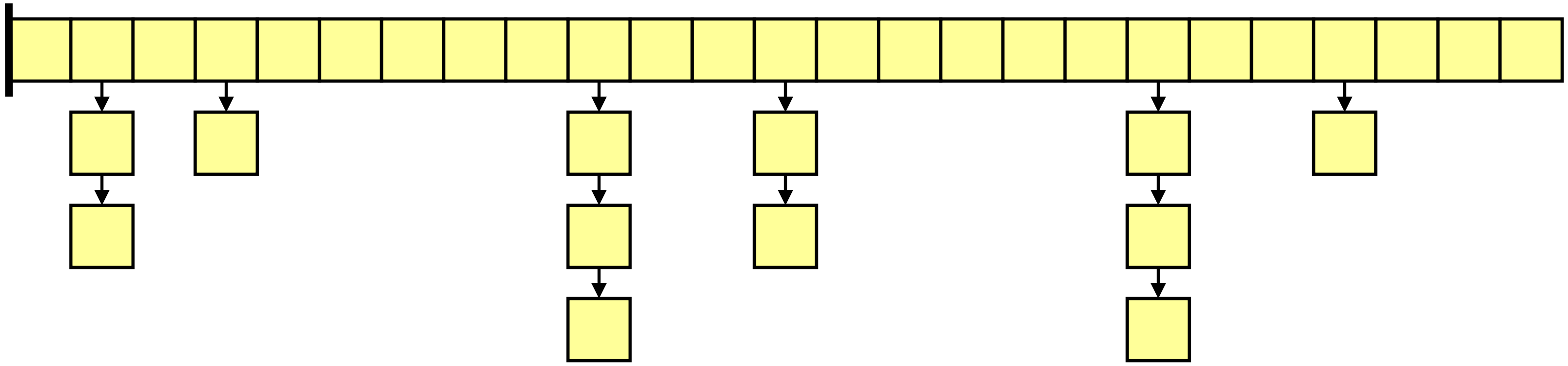
- Hash Table is an array. Each location holds one key-data pair, “empty”, or “deleted”.
- Search in a sequence of locations (the **probe sequence**), beginning at the location given by the hashed key.
- **Linear probing**: t, t+1, t+2, etc.
  - Tends to form **clusters**.
- **Quadratic probing**: t, $t+1^2$, $t+2^2$, etc.
- **Double hashing**: Use another hash function to help determine the probe sequence.



Review
Hash Tables — Collision Resolution [2/2]

Collision Resolution Methods — Type 2: **“Buckets”**

- Hash Table is an array of data structures, each of which can hold multiple key-data pairs.
- Array locations are **buckets**.
- **Separate chaining**: Each bucket is a Linked List.
  - This is very common.



Review
Hash Tables — Table-Remake

Sometimes it is necessary to remake the Hash Table.

- All implementations have performance degradation as the number of data items rises.

In these cases, we need to do a reallocate-and-copy, as we did with smart arrays.

This is one of the downsides of Hash Tables.

Hash Tables continued
Efficiency — Introduction

A **perfect hash function** (one without collisions) results in insert, delete, and retrieve operations that are $O(1)$.

- In practice, we **cannot** guarantee this, *if* we allow insert & delete operations.
- But this might be a good idea, for a fixed data set (no insert/delete).

In the **worst case**, all items get the same hashed value, and so collisions happen nearly all the time.

- Thus, retrieve is linear time (worst case), for most implementations.
- *But what if our buckets are Red-Black Trees?*

However, we generally use a Hash Table when we are interested in **average-case** performance.

The average-case performance of a Hash Table can be analyzed based on the **load factor**.

- The *load factor*, denoted by α, is:
  (# of items present) / (# of locations in table)
- We generally want α to be small. In the following slides, we will assume α is significantly less than 1 (less than 2/3, maybe?).
- We will also assume, *for now*, that no Table-remake is required.

Hash Tables
Efficiency — Separate Chaining

For example, consider separate chaining.

- Worst Case
    - Insert is constant time, assuming we do not search.
        - We can avoid a search, if we allow duplicate keys.
    - Retrieve and delete require a search: linear time.
    - Similarly, if we do not allow duplicate keys, then insert requires a search, and so is linear time.
- Average Case
    - The average number of items in a bucket is α (the 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 *approximately* $1 + \alpha/2$.
    - This applies to operations requiring a search: retrieve and delete certainly, insert maybe. Insert without search is constant time.

Hash Tables
Efficiency — Open Addressing

With open addressing, retrieve, insert, and delete all require a search, even if duplicate keys are allowed.

Worst Case

- Linear time.

Average Case

- For linear probing:
  - NOT FOUND: $(1/2)[1+1/(1-\alpha)]^2$.
  - FOUND: $(1/2)[1+1/(1-\alpha)]$.
- For quadratic probing:
  - NOT FOUND: $1/(1-\alpha)$.
  - FOUND: $-\ln(1-\alpha)/\alpha$.
- Again:
  - We assume α is significantly less than 1, and that the Table-remake operation is not done.
  - The efficiency of insert, delete, and retrieve is essentially the same in all cases.

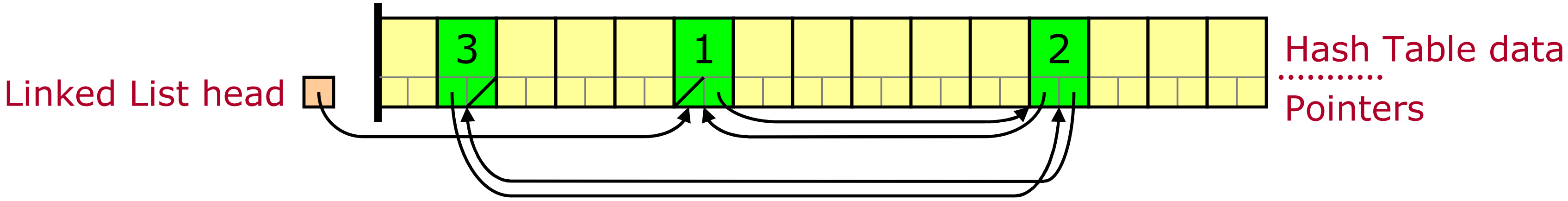
Hash Tables
Efficiency — Traverse

Hash Table **traverse** can be slow, because of the empty locations.

- Assume:
  - Either open addressing is used, or else buckets are implemented using structures that can be traversed in linear time.
  - We do not want a sorted traverse.
- Then traverse is $O(n + b)$, where n is the number of **items** in the Hash Table, and b is the number of **locations** (buckets?).

A speed-up: Use an auxiliary Doubly Linked List containing all stored key-data pairs.

- Each key-data pair gets two pointers (previous node, next node).
- Table insert & delete modify the Linked List.
- Table traverse uses the Linked List. Result: traverse is $O(n)$.



Hash Tables
Efficiency — Issues

The Table-remake operation has a similar effect on Hash-Table efficiency to that of reallocate-and-copy on a smart array.

- Constant time becomes amortized constant time.

All reasonable implementations of a Hash Table have **average-case** performance of constant time for retrieve and delete, and also for insert, if no Table-remake is required.

- For the insert operation, this becomes an average case of amortized constant time, if Table-remake operations are done intelligently.

In common Hash Table implementations, **worst-case** performance is linear time for retrieve and delete, and also for insert, if duplicate keys are not allowed.

An important issue is whether a **malicious user** can force worst-case performance.

- A well-chosen hash function makes this difficult.
- The design of such a function is beyond the scope of this class, but information and implementations are not hard to find.

Hash Tables
Efficiency — Comparison

	Idea #1	Idea #2	Idea #3	
	Priority Queue using Heap	Red-Black Tree	Hash Table: average case	Hash Table: worst case
Retrieve	Constant*	Logarithmic	Constant	Linear
Insert	(Amortized)** logarithmic	Logarithmic	Amortized constant***	Linear****
Delete	Logarithmic*	Logarithmic	Constant	Linear

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

**This is logarithmic if (1) the PQ does not manage its own memory, or (2) enough memory is preallocated. Otherwise, occasional linear-time reallocate-and-copy may be required. Time per-operation, averaged over many consecutive operations, will be logarithmic. Thus, "amortized logarithmic".

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

****This is amortized constant time if *both* of the following are true: (1) separate chaining is used, and (2) duplicate keys are allowed.

Hash Tables
Efficiency — Conclusion

We have another example of average-case vs. worst-case efficiency trade-off.

- One that we saw was Quicksort vs. $O(n \log n)$ sorts. But we do not need to worry about that any more.
- However, Hash Tables vs. balanced search trees is still an issue.

Hash Tables have very good performance for “typical” situations.

- Its occasional drawbacks can be serious.

When using a Hash Table, do so intelligently.

Prefix Trees
Background

Consider a list of words.

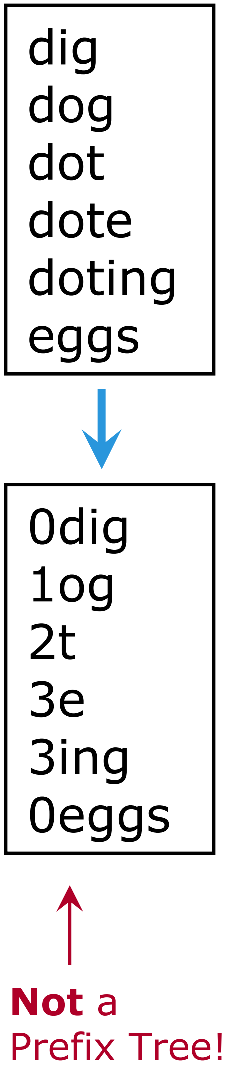
- In practice, our list might be *much* longer.
- Alphabetically order the words. Each is likely to have many letters in common with its predecessor.
- That is, consecutive words tend to have a **prefix** in common.

One easy way to take advantage of this is to store each word as a number followed by letters.

- This method is very suitable for use in a text file that is loaded all at once.
- But it does not support fast look-up by key (word).

A method more suited for in-memory use is a **Prefix Tree**.

- Also (and, sadly, more commonly) called a "Trie".
  - For "reTRIEval".
  - You're supposed to say "TREE". ☹
  - I've heard "TRY". 😐
  - Ick.



dig
dog
dot
dote
doting
eggs

0dig
1og
2t
3e
3ing
0eggs

Not a
Prefix Tree!

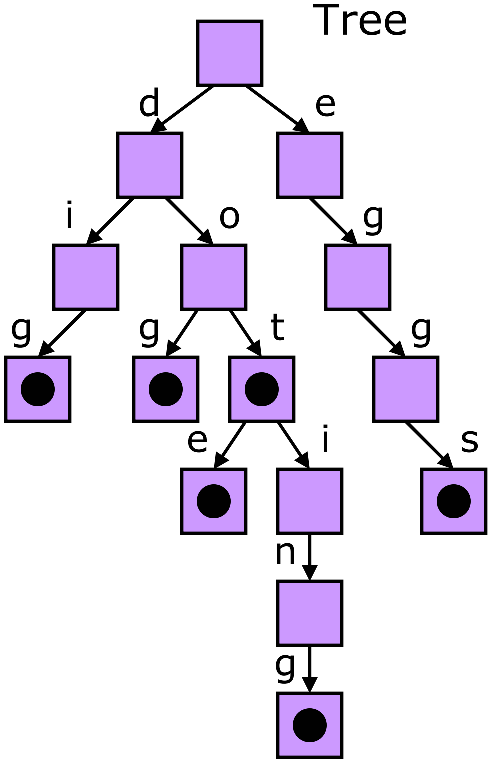
Prefix Trees
Definition [1/2]

A **Prefix Tree** (or **Trie**) is a Table implementation in which the keys are strings.

- We use “string” in a general sense, as in our discussion of Radix Sort.
  - A nonnegative integer is a string of digits.
- The quintessential key type is **words**, as in the previous slide.
- A Prefix Tree is space-efficient when keys tend to share prefixes.

A Prefix Tree is a kind of tree.

- Each node can have one child for each possible character.
- Each node also contains a Boolean value, indicating whether it represents a stored key.
  - Duplicate keys are not allowed.
- Lastly, each node can hold the data associated with a key.



Prefix Trees
Definition [2/2]

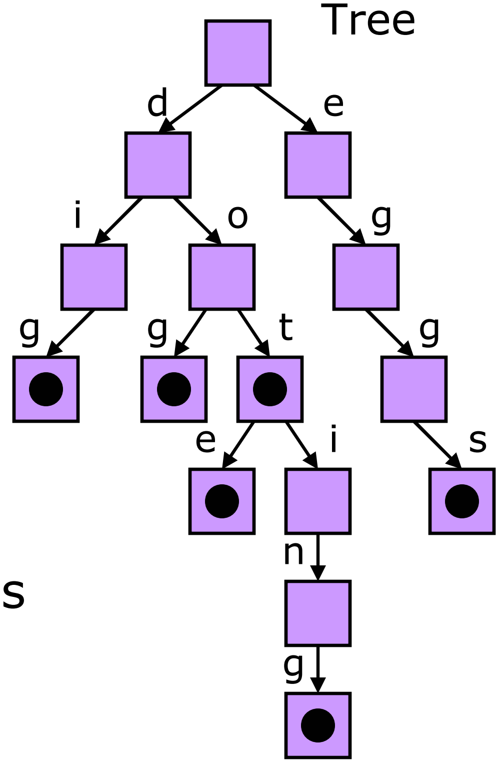
In a Prefix Tree for storing words composed only of lower-case English letters, each node has:

- 26 child pointers (one for each letter).
- A Boolean value
- A spot for the associated data.

The keys in the Prefix Tree to the right are those from our word list: **dig**, **dog**, **dot**, **dote**, **doting**, **eggs**.

- Rather than draw 26 pointers for each node, I have labeled each pointer with the appropriate letter.
- A node with a black circle is one that represents a word in the list.

Prefix Tree



Prefix Trees
Implementation

How would we implement a Prefix Tree node?

- Example:

```
struct PTNode {
    (PTNode *) ptrs_[26];  // a .. z ptrs; NULL if none
    bool isWord_;          // true if a word ends here
    DataType data_;
};
```

An RAII class would be good to have here. *See Boost's* **`shared_ptr`***.*

- Another possibility:

```
struct PTNode {
    std::map<char, PTNode *> ptrs_;
    bool isWord_;
    DataType data_;
};
```

An STL Table implementation (think "Red-Black Tree")

Prefix Trees
Any Good?

Efficiency

- For a Prefix Tree, Table retrieve, insert, and delete all take a number of steps proportional to the length of the key.
- If word length is considered fixed, then all are constant time.
- However, word length is logarithmic in the number of *possible* words.
  - A hidden logarithm, just like Radix Sort.

A Prefix Tree is a good basis for a Table implementation, when keys are short-ish sequences from a not-too-huge alphabet.

- Words in a dictionary, ZIP codes, etc.
- Just like Radix Sort.

A Prefix Tree is **easy to implement**.

The idea behind Prefix Trees is also used in other data structures.