Introduction to Tables

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, April 21, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview
The Basics of Trees

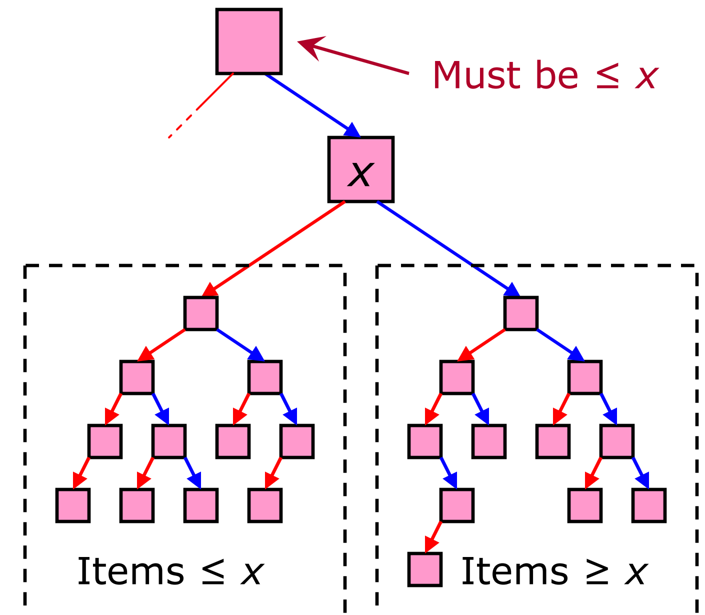
Major Topics

- ✓ Trees in General
- ✓ Binary Trees
- ✓ Binary Search Trees
- ✓ Treesort

Review
Binary Search Trees — Introduction

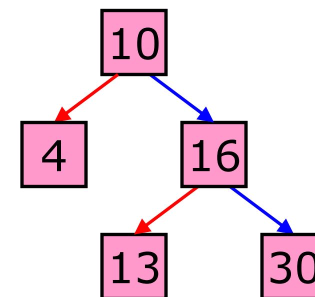
A **Binary Search Tree** is a Binary Tree in which, for each node:

- Descendants holding data less than the node's data are in its left subtree.
- Descendants holding data greater than the node's data are in its right subtree.



In other words, an inorder traversal gives items in sorted order.

This is another value-oriented ADT.



Review
Binary Search Trees — Operations: Retrieve

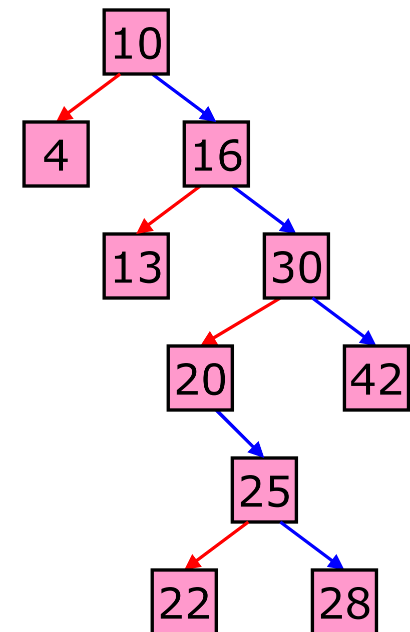
To **retrieve** a value from a Binary Search Tree, we begin at the root and repeatedly follow left or right child pointers, depending on how the search key compares to the key in each node.

For example, search for the key 20 in the tree shown:

- $20 > 10 \rightarrow$ right.
- $20 > 16 \rightarrow$ right.
- $20 < 30 \rightarrow$ left.
- $20 = 20 \rightarrow$ FOUND.

When searching for a key that is not in the tree, we stop when we find where the key "should" be. Search for the key 18 in the same tree:

- $18 > 10 \rightarrow$ right.
- $18 > 16 \rightarrow$ right.
- $18 < 30 \rightarrow$ left.
- $18 < 20 \rightarrow$ left.
- No left child $\rightarrow$ NOT FOUND.



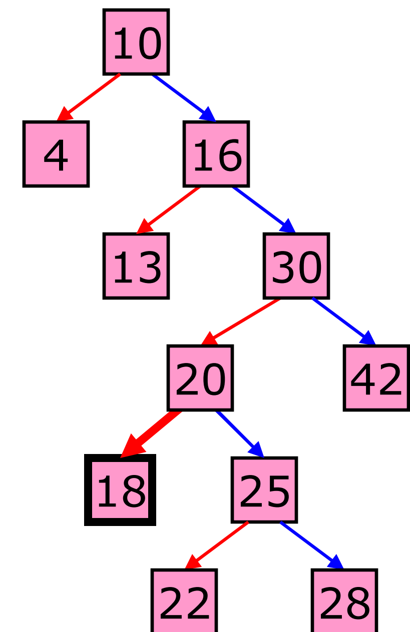
Review
Binary Search Trees — Operations: Insert

To **insert** a value with a key that is **not in the tree**, we find where it "should" go, and then put it there.

- For example, we just found where 18 should go.

When we insert a value with a key that is **already in the tree**, our action depends on the specification of the Binary Search Tree.

- We may **do nothing**.
- Or we may **replace** the value with the given key.
- Or we may **add a new value** having the same key.
  - Result: multiple equivalent keys.
- Or we may signal an **error**.



Review
Binary Search Trees — Operations: Delete [1/3]

Deletion is the most complex of the three operations.

We assume the key to be deleted is in the tree.

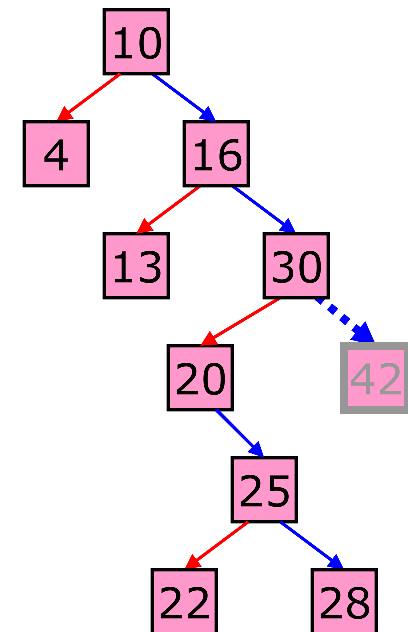
- Otherwise, the spec's will tell us what to do.
  - Flag an error?
  - Do nothing?

There are three cases:

- The node to be deleted has no children (it is a leaf).
- The node has 1 child.
- The node has 2 children.

The **no-children** (leaf) case is easy:

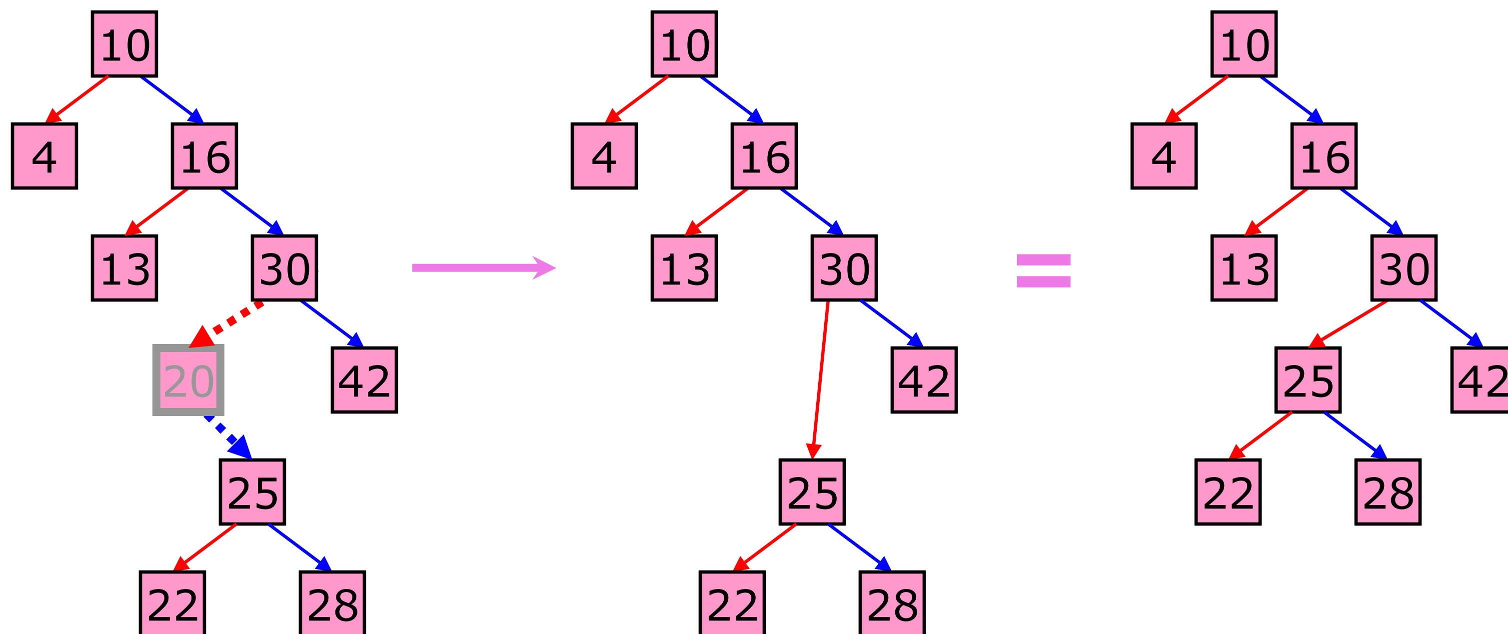
- Just delete the node, using the appropriate Binary Tree operation.
- Example: Delete key 42.



Review
Binary Search Trees — Operations: Delete [2/3]

If the node to delete has exactly **one child**, then we replace the subtree rooted at it with the subtree rooted at its child.

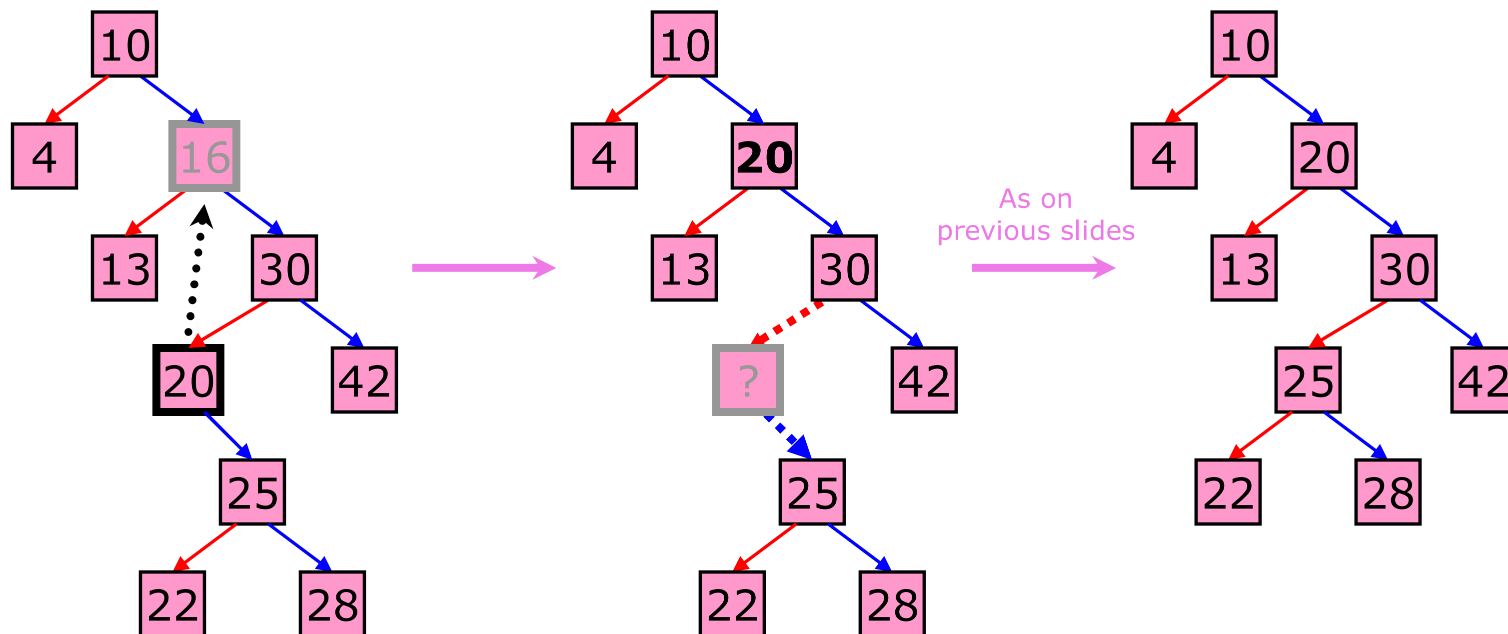
- This is a constant-time operation, once the node is found.
- Example: Delete key 20.



Review
Binary Search Trees — Operations: Delete [3/3]

The tricky case is when the node to delete has **two children**.

- Replace the node's data with the data in its **inorder successor**.
  - By copying or swapping.
- The inorder successor cannot have two children. Delete it as before.
- Example: Delete key 16.



Review
Binary Search Trees — Operations: Summary

Algorithms for the three primary B.S.T. operations:

- Retrieve
  - Start at root. Go down, left or right as appropriate, until key or empty spot found.
- Insert
  - Retrieve, then …
  - Put the value is the spot where it should be.
- Delete
  - Retrieve, then …
  - Check number of children node has:
    - 0. Delete node.
    - 1. Replace subtree rooted at node with subtree rooted at child.
    - 2. Copy from (or swap with) inorder successor. Proceed as above.

All three operations, in the worst case, require a number of steps proportional to the **height of the tree**.

The height of a tree is small (and all three operations are fast) if the tree is **balanced**.

Review
Binary Search Trees — Efficiency [1/3]

The **minimum size** of a balanced Binary Tree with **height** h is $F_{h+2} - 1$, where F_k is Fibonacci number k.

Thus, the **maximum height** of a balanced Binary Tree with **size** n is $O(\log n)$.

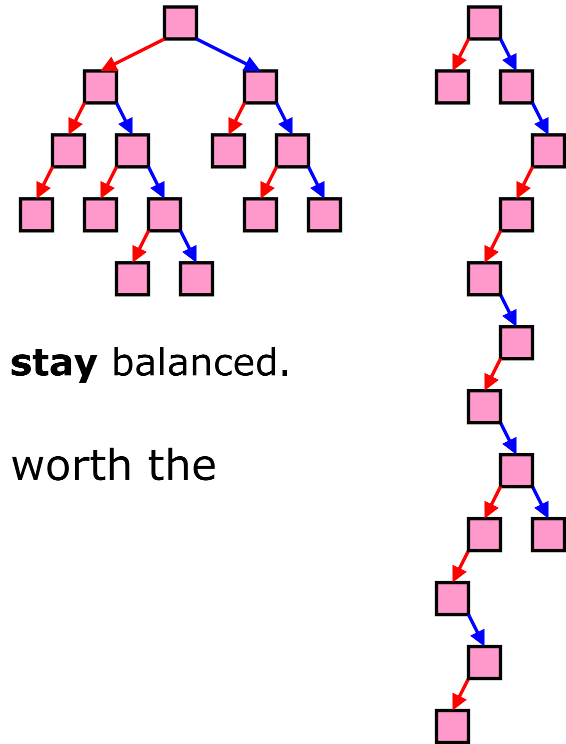
It is also true that, with random data, the height of a Binary Tree is, with high probability, $O(\log n)$.

However, in the worst case, the height of a Binary Tree is $O(n)$.

This is important, since the number of steps required by many Binary Search Tree algorithms is proportional to the height of the tree.

Review
Binary Search Trees — Efficiency [2/3]

How efficient are the B.S.T. operations (using the standard implementation)?



- isEmpty
  - Constant time.
- Retrieve
  - Linear time.
  - Worst case is roughly the height.
    - If balanced: logarithmic time. But it might not **stay** balanced.
    - Logarithmic time on average for random data.
  - Retrieve does not modify the tree. It may be worth the time to create a balanced tree beforehand.
- Insert
  - Linear time.
  - See second point under "Retrieve".
- Delete
  - Linear time.
  - See second point under "Retrieve".
- Pre-, in-, postorder traversal
  - Linear time.

Review
Binary Search Trees — Efficiency [3/3]

A B.S.T. has a nice interface for sets and key-based look-up.

Further, a B.S.T. has good average performance; Insert, Delete, and Retrieve are logarithmic time for typical data.

But in the worst case, a B.S.T. is worse than a sorted array.

- It is also worse in memory usage.

	B.S.T. (balanced & average case)	Sorted Array	B.S.T. (worst case)
Insert	Logarithmic	Linear	Linear
Delete	Logarithmic	Linear	Linear
Retrieve	Logarithmic	Logarithmic	Linear

Can we efficiently keep a Binary Search Tree balanced?

- We will look at this question again later.

Review
Treesort — Introduction

For most sorted containers, there is an associated sorting algorithm.

- Insert all items into the container, and then iterate through it.
- For a sorted array, this algorithm is pretty nearly Insertion Sort.
  - It would be a non-in-place version of Insertion Sort.
- For a Binary Search Tree, the algorithm is called **Treesort**.
  - Note: We must allow equivalent items in our B.S.T.

Treesort is not a very good algorithm, but it is worth looking at.

What is the order of Treesort?

- $O(n^2)$.
  - There are n insert operations, each of which is $O(n)$, plus a single $O(n)$ traversal.
- However, it is **usually** pretty fast: $O(n \log n)$ on average.
  - Because B.S.T. Insert is $O(\log n)$ for average data.

Have we seen Treesort before?

- Kind of. It is basically Quicksort in disguise.
- The main practical difference is that Treesort is not in-place.

Review
Treesort — Analysis

Efficiency ☹

- Treesort is $O(n^2)$.
- Treesort has a good average-case time: $O(n \log n)$. ☺

Requirements on Data ☺*

- Treesort does not require random-access data.
  - *This is not much of an advantage for an algorithm that is inefficient in both space and time. (We could start by copying to an array.)

Space Usage ☹

- Treesort is not in-place.
- It requires $O(n)$ additional space for the tree.

Stability ☺

- Treesort is stable.
  - Even though Quicksort is not. Do you see why?

Performance on Nearly Sorted Data ☹

- Treesort is **slow** on nearly sorted data: $O(n^2)$.
  - Just like unoptimized Quicksort.

Unit Overview
Tables & Priority Queues

Our next unit covers ADTs Table and Priority Queue and their implementations (in particular, Heaps and the associated algorithms, balanced search trees, and Hash Tables).

- Topics will include:
  - Introduction to Tables
  - Priority Queues
  - Heap algorithms
  - Heaps and Priority Queues in practice
  - 2-3 Trees
  - Other balanced search trees
  - Hash Tables
  - Prefix Trees
  - Tables in practice

This will be the last *big* unit in the class. After this, if time permits, we will look briefly at:

- External methods.
- Graph algorithms.

Introduction to Tables
Types of ADTs

Position-Oriented ADTs

- Get an item based on where it is stored.
- Organize data according to where the client wants it.

Examples

- Sequence
- Stack
- Queue
- Binary Tree

Value-Oriented ADTs

- Get an item based on its value.
  - Or part of the value: key-based look-up.
- Organize data for greatest efficiency.

Examples

- SortedSequence
- Binary Search Tree

Since the client doesn't care about the data organization, but only wants efficiency, maybe we can do better here?

Introduction to Tables
Databases

A value-oriented ADT can be thought of an as interface to a general database.

Consider the four data-manipulation commands in the Structured Query Language (SQL):

- **Select**
  - Retrieve a record. Key-based look-up.
- **Update**
  - Change a record.
  - A redundant operation, since we can always delete and then insert. Alternatively, have Select return a reference (or iterator or whatever).
- **Insert**
  - Given a record, insert it.
- **Delete**
  - Given a key, delete the associated record.

These are essentially the operations in a value-oriented ADT.

- We want an implementation that makes them efficient.

Introduction to Tables
Operations — Possibilities

A general value-oriented ADT is called "Table".

What operations should Table have?

- The Usual
  - **create**, **destroy**, **copy**, **setEqual**.
  - **isEmpty**.
  - **size** (maybe).
- Data Manipulation
  - **retrieve** (like SQL Select).
  - *Maybe* **set** (like SQL Update).
    - We generally handle this either by having retrieve return a value in modifiable form, or by simply using delete, then insert.
  - **insert**.
  - **delete**.
- Access All Data
  - **traverse**.

Introduction to Tables
Operations — Issues

Allow multiple items with the **same key**?

- It depends on the requirements of the client.

Require **traverse** to list items in sorted order?

- There are sorted & unsorted implementations. Requiring a sorted traverse would make the latter inefficient.

Allow modification of data in the Table?

- If we have key-data pairs, then modifying the **data** part is no problem.
- Modifying the **key** is tricky, since the item is generally stored according to its key. Changing the key means we have to move the item.

Have a separate interface in which the key is the entire value?

- Maybe. Call it "Set".
- Certainly a separate **implementation** would be worthwhile.

Conclusions

- There is no single, best interface to a Table. But they are all very similar.
- Therefore, we will be a little vague about *exactly* what a Table is.

Introduction to Tables
Applications

What do we use a Table for?

- To hold **sparse** array data.
  - `arr[6] = 1; arr[1000000000] = 2;`
- To hold “arrays” whose indices are not nonnegative integers.
  - `arr2["hello"] = 3;`
- To hold data accessed by key fields. For example:
  - Customers accessed by phone number.
  - Students accessed by student ID number.
  - Any other kind of data with an ID code.

Introduction to Tables
Possible Implementations [1/3]

How can we implement a Table?

- Using a Sequence.
  - Sorted or unsorted.
  - Array-based or Linked-List-based.
- Using a Binary Search Tree.
  - Implemented using a pointer-based Binary Tree.

For each of these, consider:

- How efficient are each of the operations?
- When (if ever) might we want to implement a Table this way?

Introduction to Tables
Possible Implementations [2/3]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced*** Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Amort. const. (or constant)*	Linear**	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear**	Linear**	Linear	Logarithmic

*Amortized constant time, if we might have to reallocate.

**Inserting and deleting in a Linked List are constant-time operations if we already know where to insert/delete. But the Table delete operation includes finding the item, and in a sorted list, the Table insert operation needs to find the correct spot. These "find" operations require linear time.

***Of course, we do not (yet!) know any way to guarantee the tree will *stay* balanced, unless we can restrict ourselves to read-only operations (no insert, delete).

Introduction to Tables
Possible Implementations [3/3]

Tables can be implemented in many ways.

- Different implementations are appropriate in different circumstances.

In special situations, the (amortized) constant-time insertion for an unsorted array and the logarithmic-time retrieve for a sorted array can be combined!

- Insert all data into an unsorted array, sort the array, then use Binary Search to retrieve data.
- This is a good way to handle Table data with **separate filling & searching phases** (and few or no deletes).

Introduction to Tables
Better Ideas? [1/3]

	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.
- Maybe do not allow retrieve & delete on *all* keys.

Application: Priority Queue

Introduction to Tables
Better Ideas? [2/3]

	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 #2: Keep a Tree Balanced

- Figure out how to keep a Binary Search Tree balanced, without compromising efficiency.
- Maybe loosen the “binary” requirement.
- Maybe loosen the “balanced” requirement, too.

Application: Balanced search trees

- 2-3 Tree & 2-3-4 Tree
- **Red-Black Tree**
- AVL Tree
- **B-Tree & B+-Tree**

Introduction to Tables
Better Ideas? [3/3]

	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 #3: "Magic Functions"

- Consider an unsorted Linked List.
  - Only **finding** takes time.
  - A "magic function" that gives us an key's index might make things very fast …
  - … if we could get to the item quickly (Linked List look-up by index is linear time).
- Suppose we used an unsorted **array**.
  - Then we could get to the item quickly, but we could not delete it quickly.
  - If we allowed **empty** items, we might be able to insert, delete, and retrieve in **constant time**.
- But we would still need that "magic function".

Application: Hash Tables