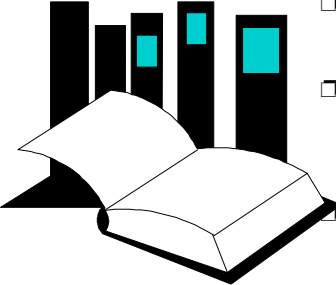



Binary Search Trees



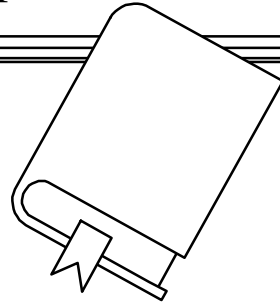
- ❑ One of the tree applications in Chapter 10 is **binary search trees**.
- ❑ In Chapter 10, binary search trees are used to implement bags and sets.
- ❑ This presentation illustrates how another data type called a **dictionary** is implemented with binary search trees.

This lecture shows a common application of binary trees: Binary Search Trees used to implement a Dictionary data type.

Before this lecture, students should have a good understanding of binary trees, and should have seen some basic container data types similar to a dictionary (for example, a bag or a set).

The Dictionary Data Type

- ❑ A dictionary is a collection of items, similar to a bag.
- ❑ But unlike a bag, each item has a string attached to it, called the item's key.

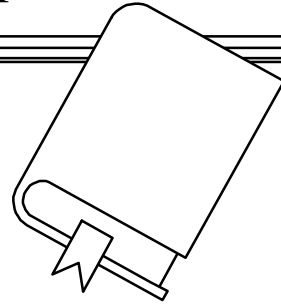


This lecture shows how to use a binary tree to implement an abstract data structure called a Dictionary. We'll start by explaining what a Dictionary is, without any reference to trees. Then we will show how the trees can be used to actually implement a Dictionary. It's important to realize that trees are but one possible way to implement a Dictionary, and the actual explanation of "What is a Dictionary?" will not refer to trees at all. In other words, a Dictionary is an abstract data type, and the trees are one of the mechanisms that can be used to implement the Dictionary.

So, what is a Dictionary? In many ways it is like other ADTs that you have seen, such as a bag which contains a collection of items. The difference is that each item in a Dictionary is attached to a string called the item's key.

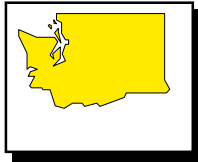
The Dictionary Data Type

- ❑ A dictionary is a collection of **items**, similar to a bag.
- ❑ But unlike a bag, each item has a string attached to it, called the item's **key**.



Example:

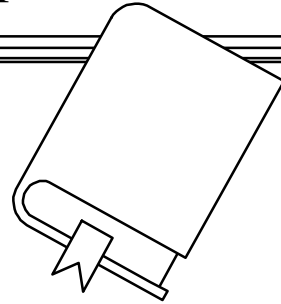
The **items** I am storing are records containing data about a state.



For this example, each item that I'm putting in the Dictionary is a record which contain a bunch of geographical information about a state.

The Dictionary Data Type

- ❑ A dictionary is a collection of **items**, similar to a bag.
- ❑ But unlike a bag, each item has a string attached to it, called the item's **key**.



Example:

The **key** for each record is the name of the state.



The key for each record is the name of the state. In general, the keys could be some other sort of value such as social security numbers. The keys must have the property that they form a total order under some comparison operation such as “less than”.

The Dictionary Data Type

```
void Dictionary::insert(The key for the new item, The new item);
```

- The insertion procedure for a dictionary has two parameters.



When an item is placed into the Dictionary, we must specify both the record of information and the key that is attached to that information. For example, if the Dictionary is implemented as an object type, then there will be an Insert method. The Insert method will have two parameters: a string (which is the key) and a record (which is the item being inserted).

The Dictionary Data Type

- When you want to retrieve an item, you specify the **key**...

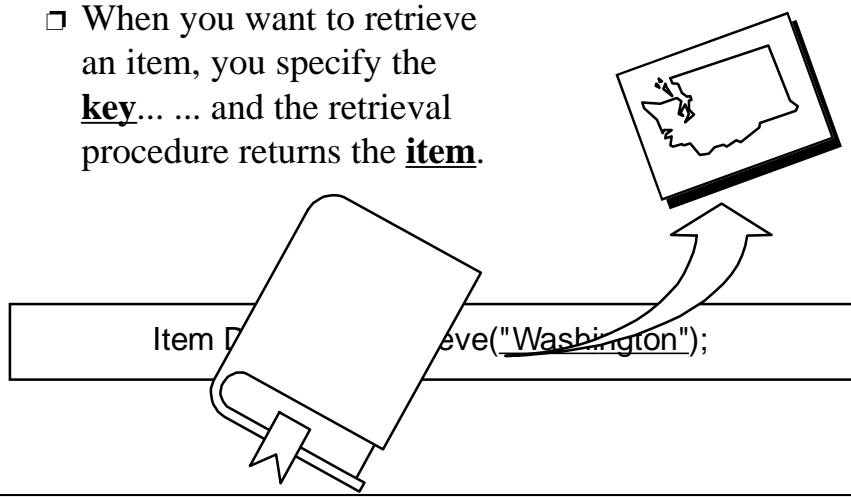


```
Item Dictionary::retrieve("Washington");
```

When you want to retrieve information from the Dictionary, you call a retrieval method, and specify the key of the item that you are looking for. This key is the parameter of the retrieval procedure.

The Dictionary Data Type

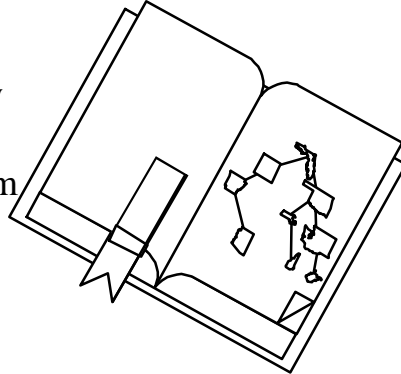
- When you want to retrieve an item, you specify the **key**... .. and the retrieval procedure returns the **item**.



The Dictionary finds the information, and returns it.

The Dictionary Data Type

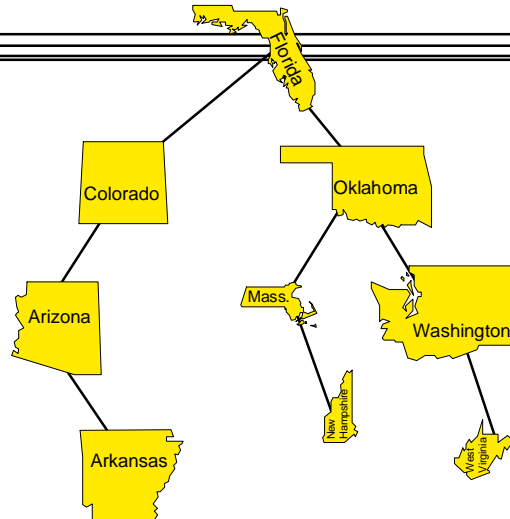
- We'll look at how a binary tree can be used as the internal storage mechanism for the dictionary.



That's enough about the abstract workings of a Dictionary. Now we are going to look at how a binary tree can be used to store the information of a Dictionary in a way that makes it fairly easy to add new items, to retrieve existing items, and to remove items. (You never know when a state might want to secede from the union.)

A Binary Search Tree of States

The data in the dictionary will be stored in a binary tree, with each node containing an **item** and a **key**.

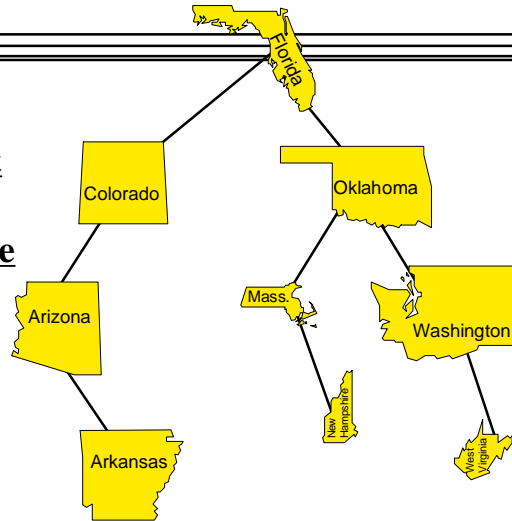


As you might have guessed, the data in the dictionary will be stored in a binary tree, with each node containing both a record of information and the key that's attached to that information. In this example, the Dictionary currently has only 9 of the 50 states, but that's enough to illustrate the idea.

A Binary Search Tree of States

Storage rules:

- 1 Every key to the **left** of a node is alphabetically **before** the key of the node.



The nodes cannot appear in just any order. The nodes must follow the special storage rules of a binary search tree. There are two such rules:

1. Every key to the left of a node is alphabetically before the key of the node.

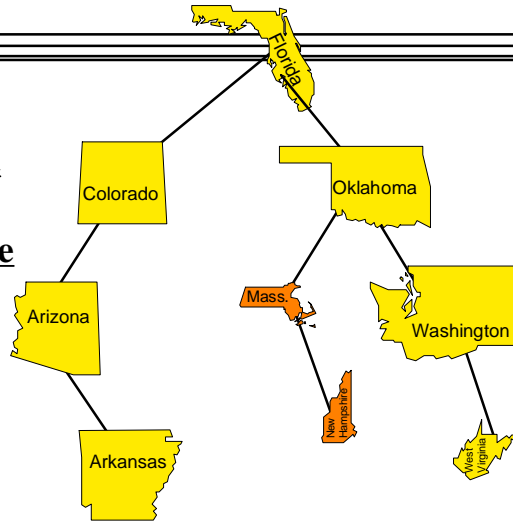
A Binary Search Tree of States

Storage rules:

- ❶ Every key to the **left** of a node is alphabetically **before** the key of the node.

Example:

'Massachusetts' and
'New Hampshire'
are alphabetically
before 'Oklahoma'



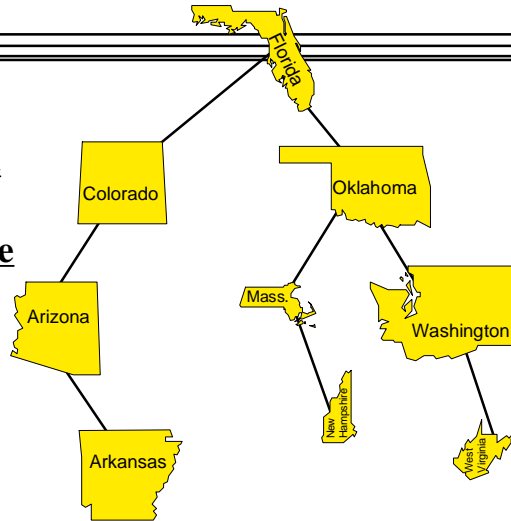
Notice that this rule applies to every which can be reached by starting down the left branch of a node. For example, if I start at Oklahoma, and head down the left branch, I can reach Massachusetts and New Hampshire.

So, both Massachusetts and New Hampshire must be alphabetically before Oklahoma.

A Binary Search Tree of States

Storage rules:

- ① Every key to the **left** of a node is alphabetically **before** the key of the node.
- ② Every key to the **right** of a node is alphabetically **after** the key of the node.



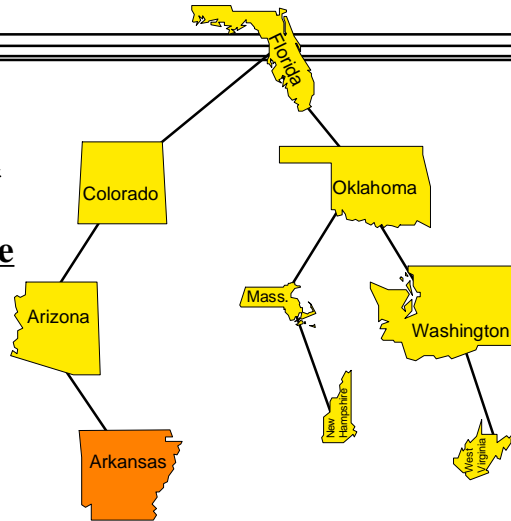
The second rule is the mirror image of the first rule:

2. Every key to the right of a node is alphabetically after the key of the node.

A Binary Search Tree of States

Storage rules:

- ① Every key to the **left** of a node is alphabetically **before** the key of the node.
- ② Every key to the **right** of a node is alphabetically **after** the key of the node.

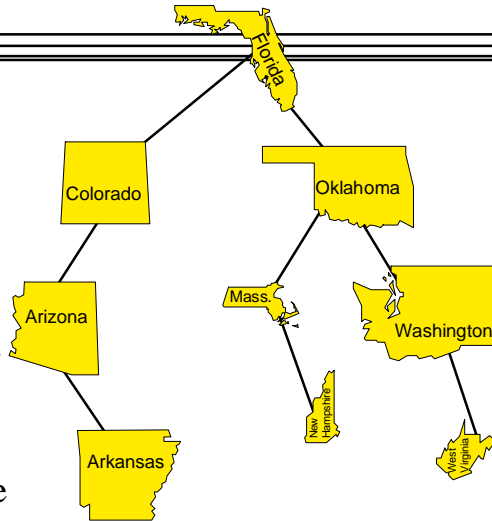


Here's an example: Arkansas is alphabetically after Arizona.

Retrieving Data

Start at the root.

- ❶ If the current node has the key, then stop and retrieve the data.
- ❷ If the current node's key is too **large**, move **left** and repeat 1-3.
- ❸ If the current node's key is too **small**, move **right** and repeat 1-3.



Once a tree is organized according to the storage rule of a binary search tree, it is easy to find any particular key, following this algorithm.

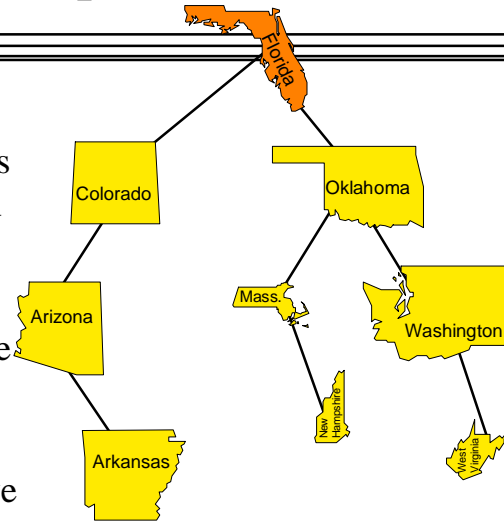
The algorithm starts at the root and repeatedly executes these steps.

1. First check to see if we found the key we were looking for. If so, then we can stop and return the associated information.
2. On the other hand, if the key at the current node is larger than the key that we're searching for, then we'll continue our search by moving leftward.
3. And if the key at the current node is smaller than the key that we're searching for, then we'll continue our search by moving rightward.

Retrieve 'New Hampshire'

Start at the root.

- ❶ If the current node has the key, then stop and retrieve the data.
- ❷ If the current node's key is too **large**, move **left** and repeat 1-3.
- ❸ If the current node's key is too **small**, move **right** and repeat 1-3.

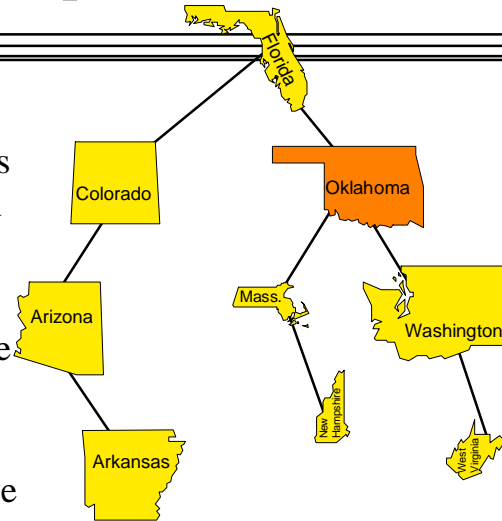


As an example, suppose we are searching for New Hampshire. We start at Florida, and since this is not the node we are after we must move down. Do we move left or right?

Retrieve 'New Hampshire'

Start at the root.

- ❶ If the current node has the key, then stop and retrieve the data.
- ❷ If the current node's key is too **large**, move **left** and repeat 1-3.
- ❸ If the current node's key is too **small**, move **right** and repeat 1-3.



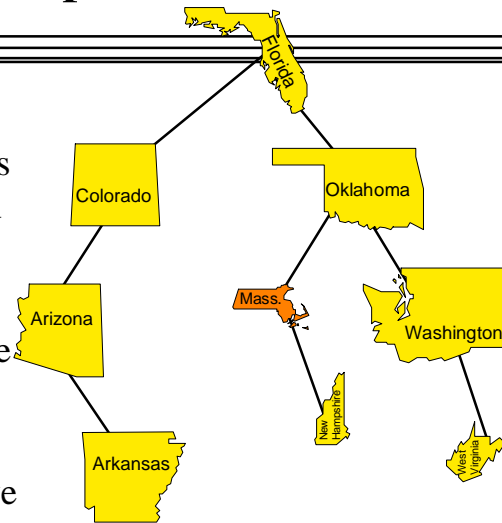
We have moved right from Florida, because Florida is smaller than New Hampshire. Or to be more specific: The name "Florida" is alphabetically before the name "New Hampshire". That's what we mean by "smaller".

Now we have arrived at Oklahoma. Which way next?

Retrieve 'New Hampshire'

Start at the root.

- ❶ If the current node has the key, then stop and retrieve the data.
- ❷ If the current node's key is too **large**, move **left** and repeat 1-3.
- ❸ If the current node's key is too **small**, move **right** and repeat 1-3.

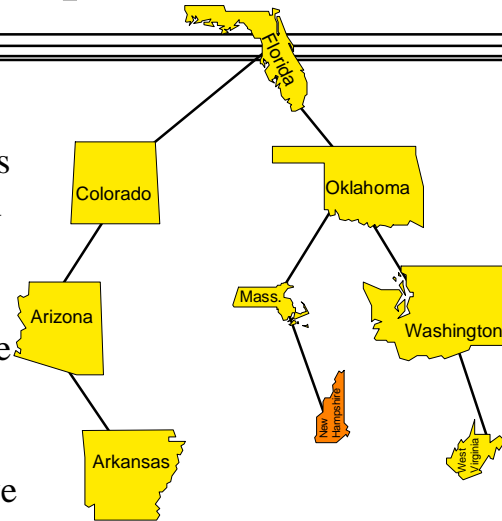


We move left from Oklahoma to Massachusetts. The leftward step was because Oklahoma is larger than the key we are looking for.

Retrieve 'New Hampshire'

Start at the root.

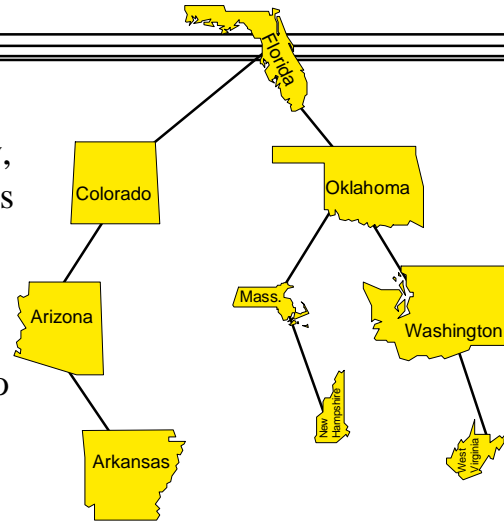
- ❶ If the current node has the key, then stop and retrieve the data.
- ❷ If the current node's key is too **large**, move **left** and repeat 1-3.
- ❸ If the current node's key is too **small**, move **right** and repeat 1-3.



From Massachusetts we continue searching, and take a right step. Now we have found New Hampshire, so we can return the data from this node.

Adding a New Item with a Given Key

- ❶ Pretend that you are trying to find the key, but stop when there is no node to move to.
- ❷ Add the new node at the spot where you would have moved to if there had been a node.

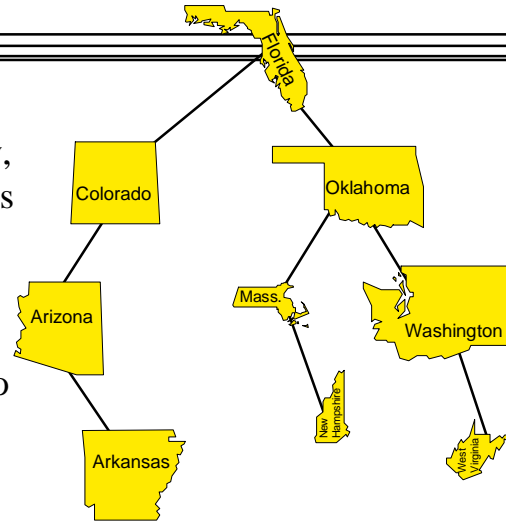


Adding a new node requires two steps, the first of which is similar to searching. In this first step we pretend that we are trying to find the key. Of course, we won't find the key, because it is not yet in the tree. So eventually we will reach a spot where there is no "next node" to step onto. At this point we stop and add the new node.

Adding



- ❶ Pretend that you are trying to find the key, but stop when there is no node to move to.
- ❷ Add the new node at the spot where you would have moved to if there had been a node.

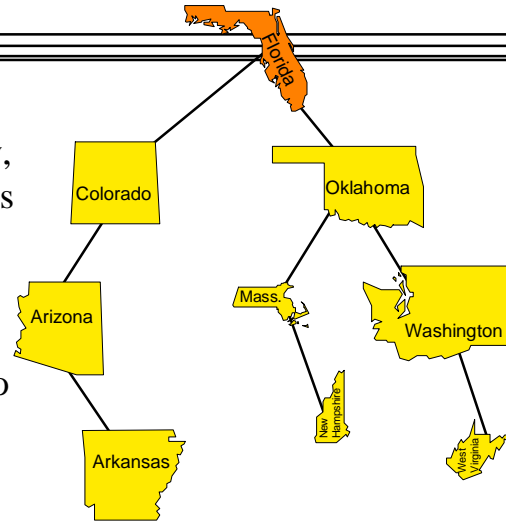


For example, suppose that Iowa wants to join our dictionary. We start by pretending to search for Iowa, beginning at the root...

Adding



- 1 Pretend that you are trying to find the key, but stop when there is no node to move to.
- 2 Add the new node at the spot where you would have moved to if there had been a node.

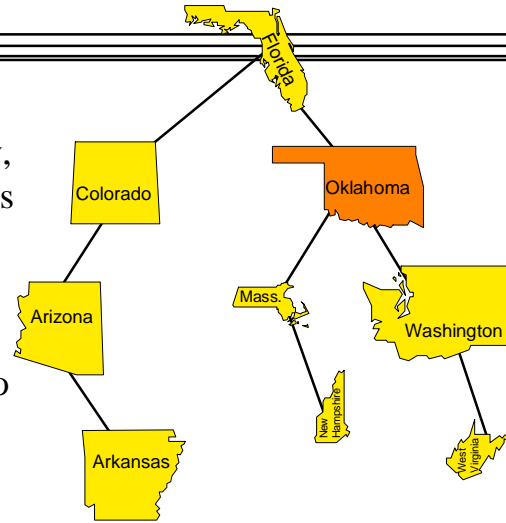


Which way will we move from the root if we are searching for Iowa?

Adding



- 1 Pretend that you are trying to find the key, but stop when there is no node to move to.
- 2 Add the new node at the spot where you would have moved to if there had been a node.

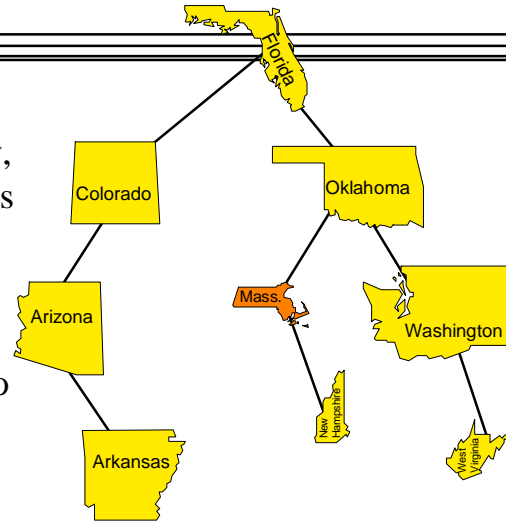


From the root we have moved right because Florida was smaller than Iowa.

Adding



- 1 Pretend that you are trying to find the key, but stop when there is no node to move to.
- 2 Add the new node at the spot where you would have moved to if there had been a node.

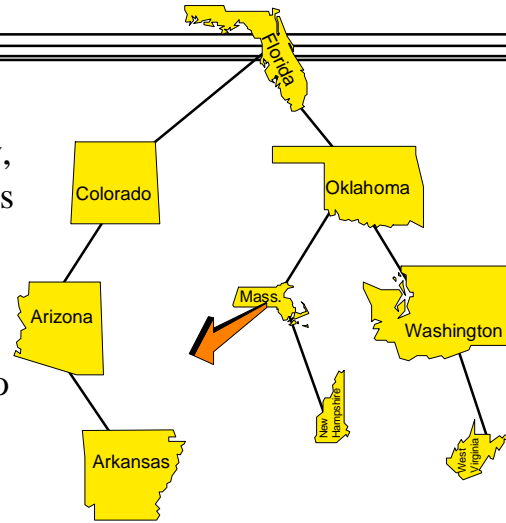


From Oklahoma we move left, onto Massachusetts,...

Adding



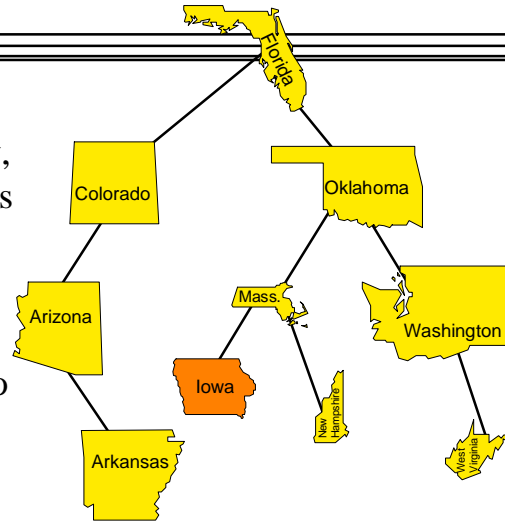
- ❶ Pretend that you are trying to find the key, but stop when there is no node to move to.
- ❷ Add the new node at the spot where you would have moved to if there had been a node.



...and from Massachusetts we would move left again, if we could. But there is no node here. So we stop...

Adding

- ❶ Pretend that you are trying to find the key, but stop when there is no node to move to.
- ❷ Add the new node at the spot where you would have moved to if there had been a node.



...and this is the location for Iowa.

Later, when we are searching for Iowa, we will follow the same path down to Massachusetts. We will step left from Massachusetts, and there we find our goal of Iowa.

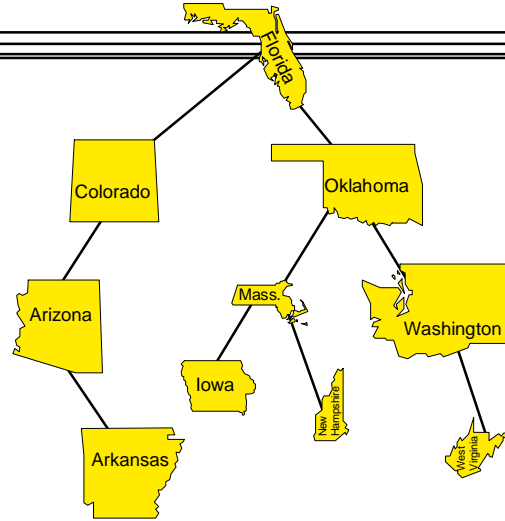
Important note: New nodes are always added at the leaves.

Adding



26

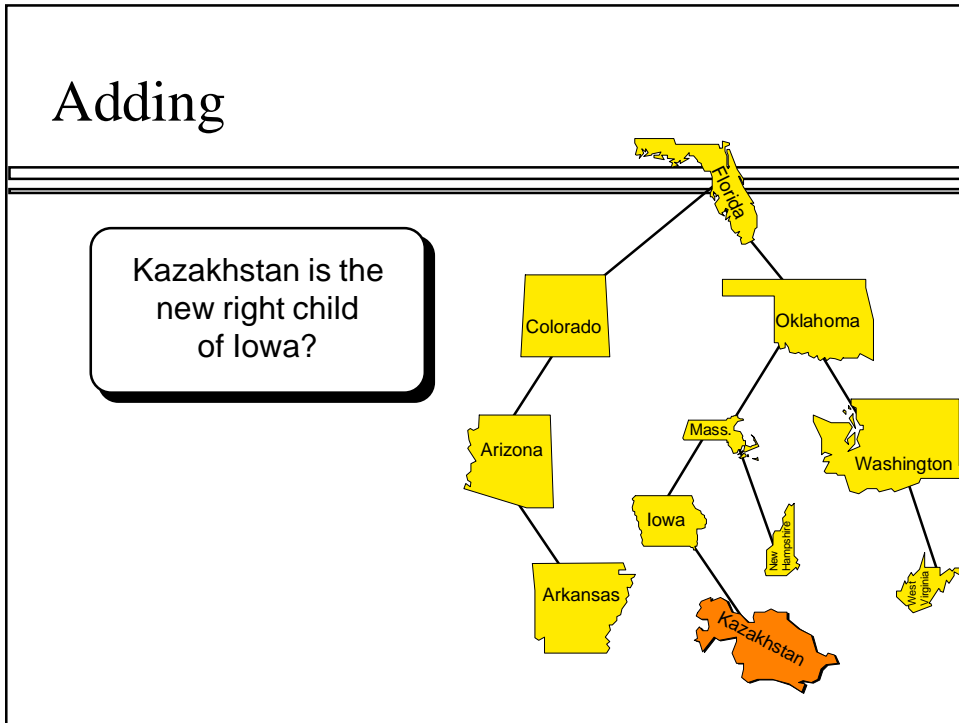
*Where would you
add this state?*



One more example: Where would you add the new state of Kazakhstan?...

Adding

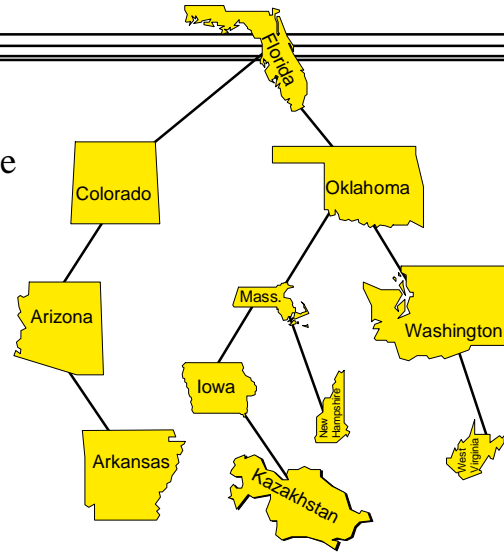
27



If you followed the algorithm, then you saw that Kazakhstan is added as the right child of Iowa.

Removing an Item with a Given Key

- ❶ Find the item.
- ❷ If necessary, swap the item with one that is easier to remove.
- ❸ Remove the item.



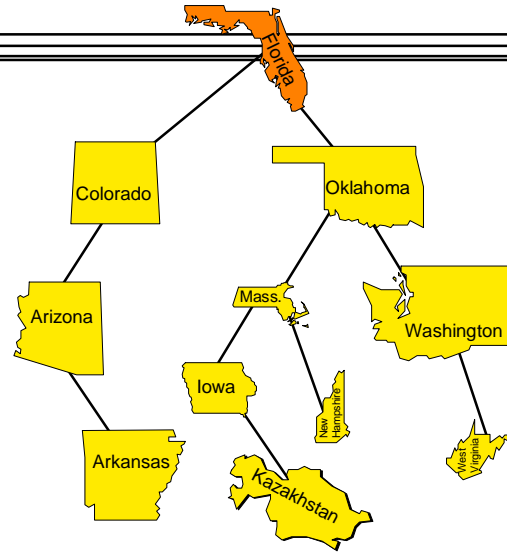
Removing an item requires three steps. We can look at the steps in detail in a moment, but first let's go through this outline so that you know roughly what to expect.

1. Find the item that you want to remove.
2. Some items are harder to remove than others. In particular, an item with two children is particularly hard to remove. So, in this second step we sometimes take an item that is hard to remove, and exchange it with an item that is easier to remove.
3. Finally, once the item is in a spot that is easy to remove, we remove it.

Let's look at the three steps in detail.

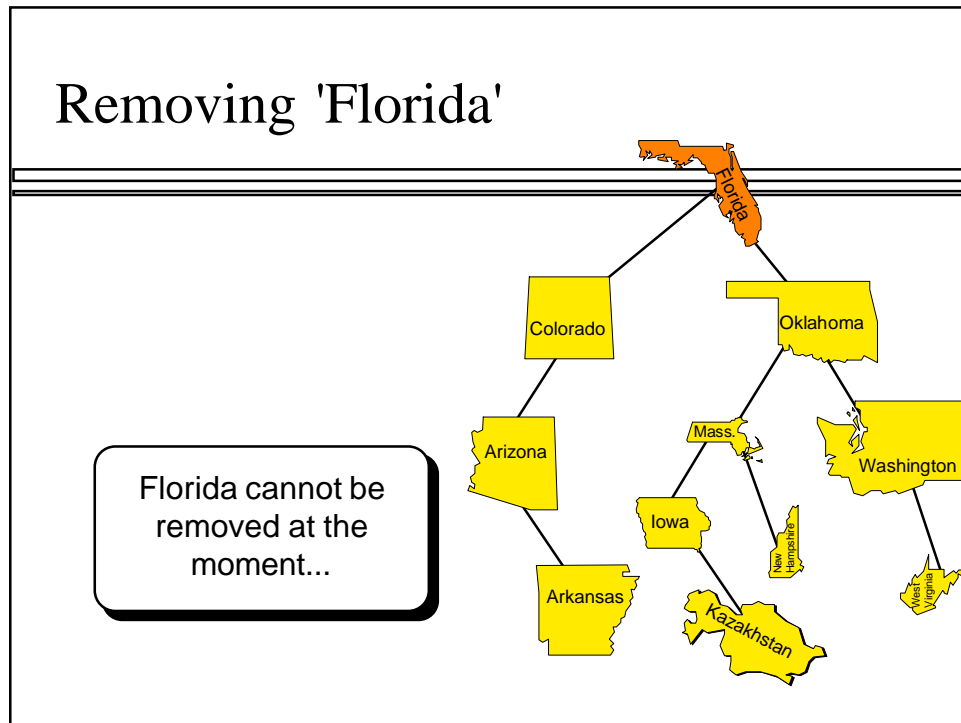
Removing 'Florida'

❶ **Find** the item.



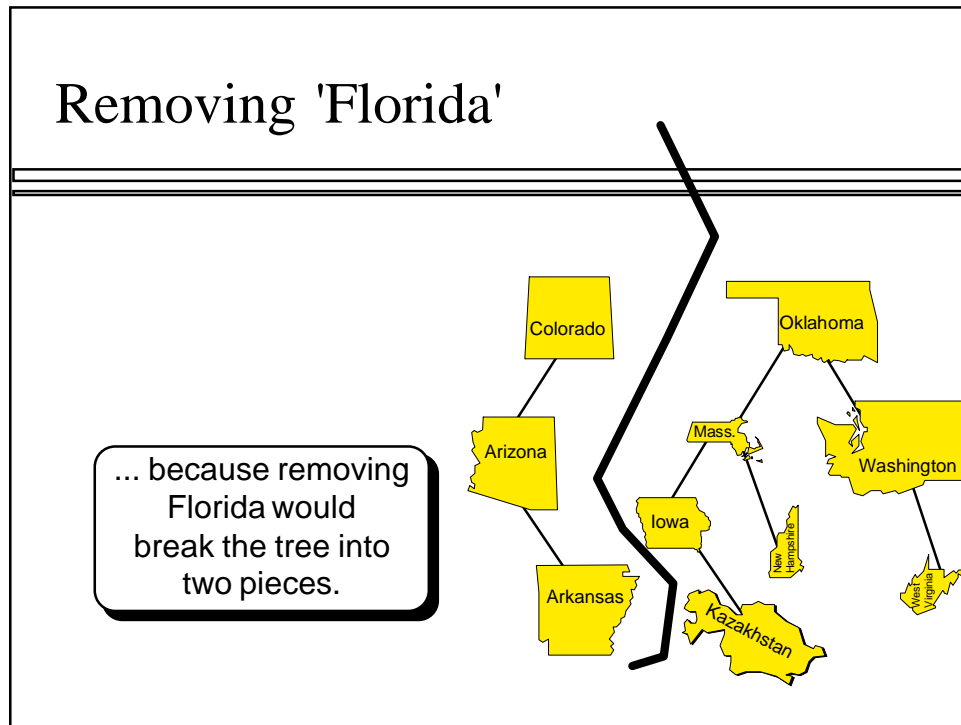
As an example, let's remove Florida. (It's way too hot to be part of the union anyway). First we find it, which is easy enough by using the usual method to search for a key.

Removing 'Florida'



In the second step we need to swap Florida with another item that is easier to remove. The reason that we need to do this swap is because of a problem that occurs if we just try to remove Florida...

Removing 'Florida'

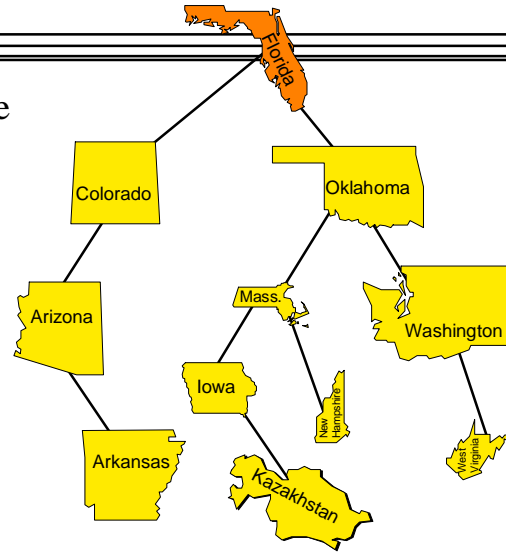


We end up breaking the tree into two separate parts.

Removing 'Florida'

- ② If necessary, do some rearranging.

The problem of breaking the tree happens because Florida has 2 children.

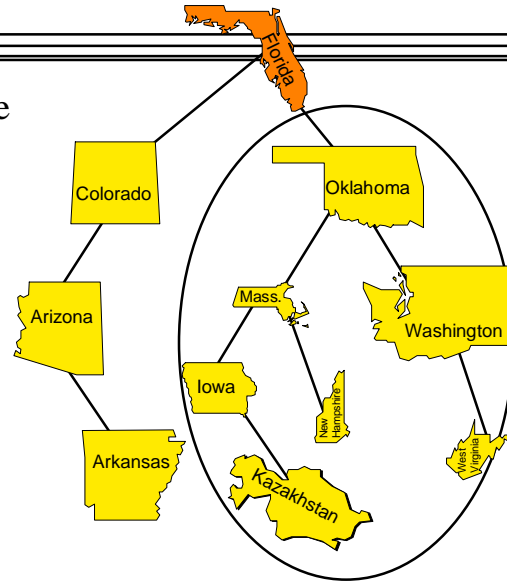


In general it is hard to recombine these two parts into a single tree. So, our goal is to find another item that is easier to remove, and copy that other item spot that we are trying to remove.

Removing 'Florida'

- ② If necessary, do some rearranging.

For the rearranging, take the **smallest** item in the right subtree...



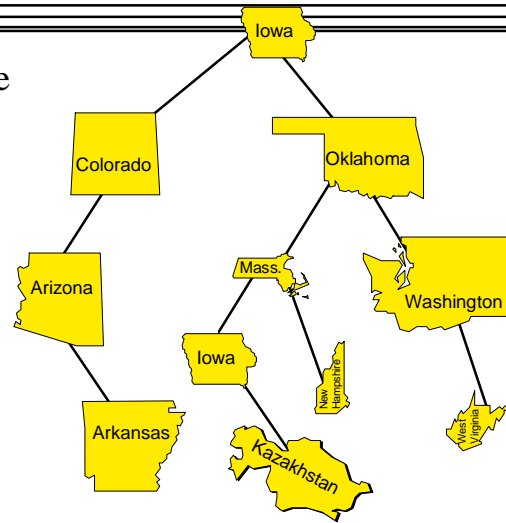
In general, there are two items that we could copy on top of Florida. In the book you'll see that one of these items is in the left subtree. In this lecture I'll use the other one, from the right subtree. In an actual program you can follow either approach.

Anyway, the approach we'll take here is to copy the smallest item in the right subtree onto Florida. To find that smallest item, step onto the right subtree (Oklahoma), and then race as far left as you can -- onto Iowa.

Removing 'Florida'

- ② If necessary, do some rearranging.

...**copy** that smallest item onto the item that we're removing...

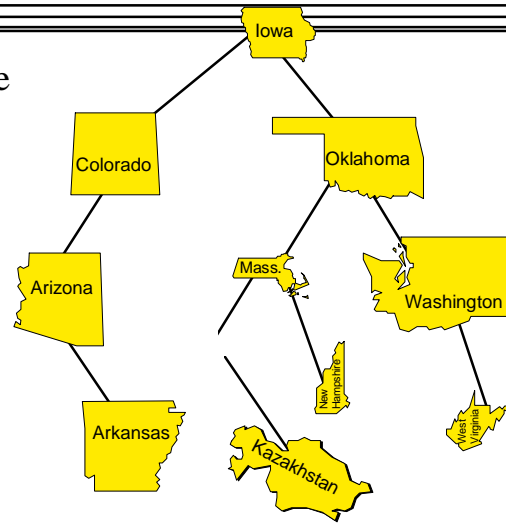


Copy Iowa on top of the item that we are removing.

Removing 'Florida'

- ② If necessary, do some rearranging.

... and then remove the extra copy of the item we copied...

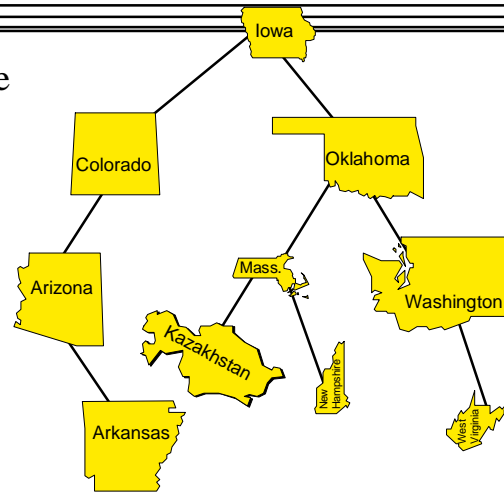


We have eliminated Florida, but now we have two copies of Iowa. If you've ever been to Iowa, you know that one Iowa is more than enough, so next we must remove the extra Iowa...

Removing 'Florida'

- ② If necessary, do some rearranging.

... and reconnect the tree

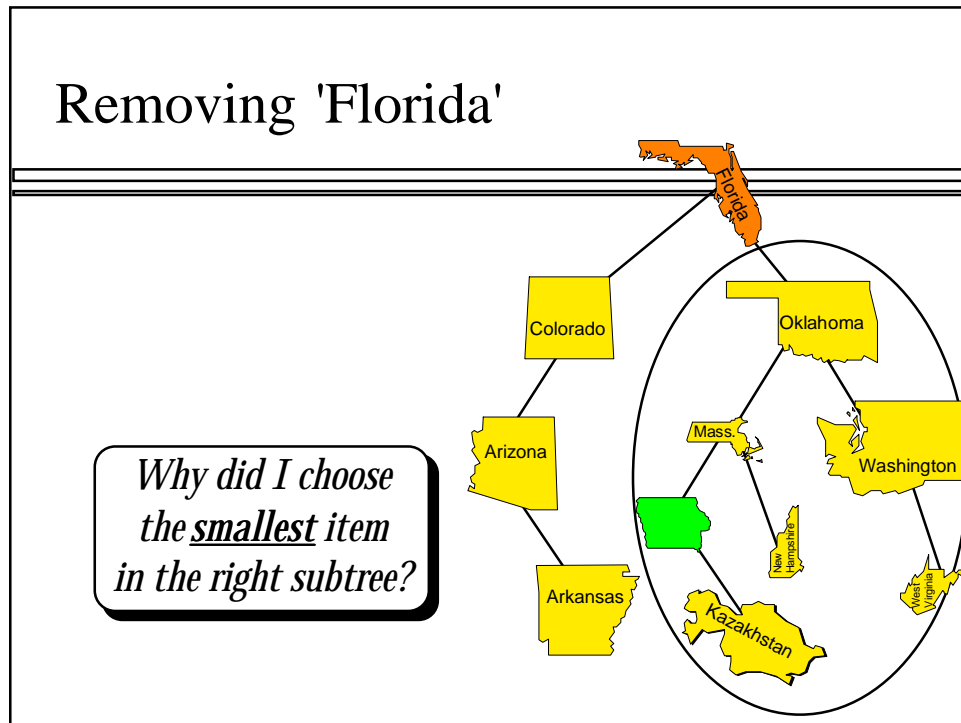


Note that the extra Iowa had one child (Kazakhstan), so that child is reconnected to the parent.

Here's a good question for you: Remember that it's hard to remove nodes with two children. How do you know that the smallest item in the right subtree does not have two children?

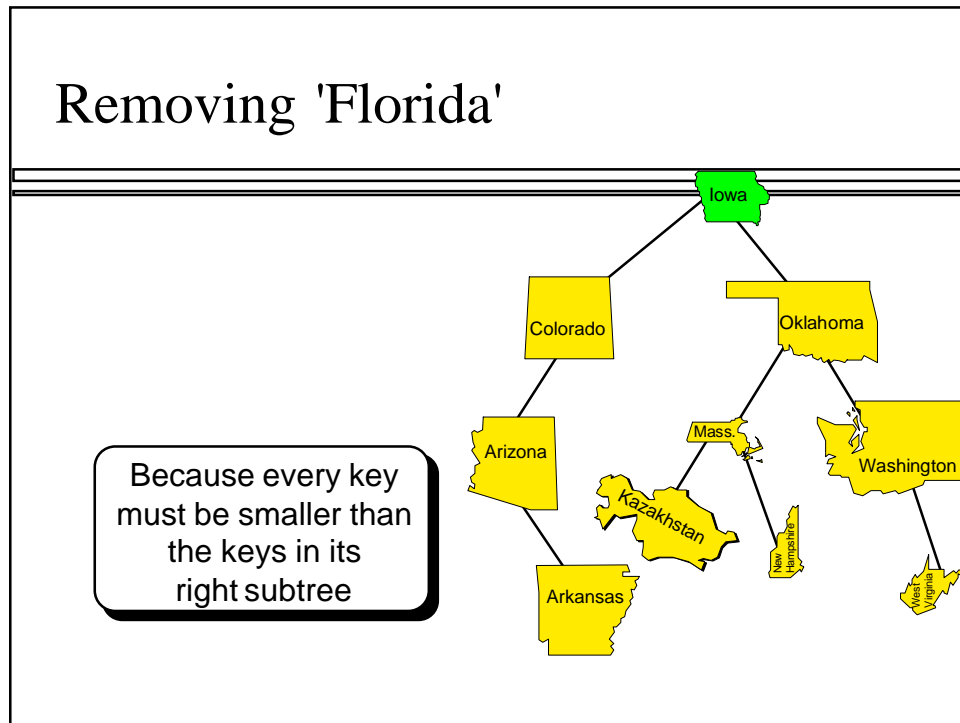
Answer: Since it is the smallest item, it can't have a left child because that left child would be even smaller.

Removing 'Florida'



In fact, the guarantee that the smallest item has at most one child is one of the reasons why I selected the smallest item. There's a second reason, can you think of it?

Removing 'Florida'



The second reason is that I am take this smallest item and place it in Florida's location. In order to maintain the binary search tree storage rules, this item must be smaller than anything that remains in the right subtree--therefore I must choose the smallest item in the right subtree.

Removing an Item with a Given Key

39

- ❶ Find the item.
- ❷ If the item has a right child, rearrange the tree:
 - ☐ Find smallest item in the right subtree
 - ☐ Copy that smallest item onto the one that you want to remove
 - ☐ Remove the extra copy of the smallest item (making sure that you keep the tree connected)else just remove the item.

Here's a summary of the removal steps. Note that if the item that we want to remove does not have a right child, then we can just remove it (and reconnect its left child if there is one).

In the text book, the process is done in a symmetrical way--using the largest item in the left subtree. Either approach works fine.

Also, the textbook implements a Bag rather than a Dictionary. One of the resultant differences is that the Bag may have several copies of the same item, so that items in the left subtree can be less than or equal to the item in a node.



Summary

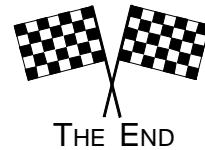
- ❑ Binary search trees are a good implementation of data types such as sets, bags, and dictionaries.
- ❑ Searching for an item is generally quick since you move from the root to the item, without looking at many other items.
- ❑ Adding and deleting items is also quick.
- ❑ But as you'll see later, it is possible for the quickness to fail in some cases -- can you see why?

A quick summary . . .

Presentation copyright 1997 Addison Wesley Longman,
For use with *Data Structures and Other Objects Using C++*
by Michael Main and Walter Savitch.

Some artwork in the presentation is used with permission from Presentation Task Force
(copyright New Vision Technologies Inc) and Corel Gallery Clipart Catalog (copyright
Corel Corporation, 3G Graphics Inc, Archive Arts, Cartesia Software, Image Club
Graphics Inc, One Mile Up Inc, TechPool Studios, Totem Graphics Inc).

Students and instructors who use *Data Structures and Other Objects Using C++* are welcome
to use this presentation however they see fit, so long as this copyright notice remains
intact.



Feel free to send your ideas to:

Michael Main

main@colorado.edu