Binary Search Trees continued
Introduction to Tables

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, November 4, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
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.
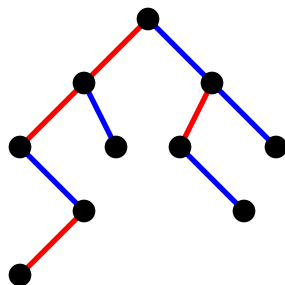
Unit Overview
The Basics of Trees

Major Topics

    ✓ ▪ Introduction to Trees

    ✓ ▪ Binary Trees

(part) ▪ Binary Search Trees

A **Binary Tree** consists of a set *T* of nodes so that either:

- *T* is **empty** (no nodes), or
- *T* consists of a node *r*, the **root**, and two subtrees of *r*, each of which is a Binary Tree:
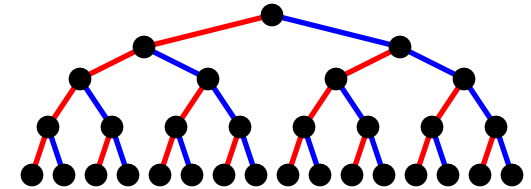 - the **left** subtree, and
  - the **right** subtree.

We make a strong distinction between left and right subtrees. Sometimes we use them for very different things.

a

a

b

b

Different Binary Trees

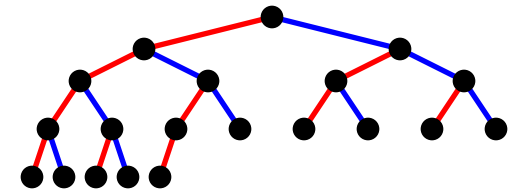The left and/or right subtree of a vertex may be empty.

Full Binary Tree

- Leaves all lie in the same level.
- All other nodes have two children each.

Complete Binary Tree

- All levels above the bottom are full.
- Bottom level is filled left-to-right.
- Importance. Nodes are added in a fixed order. Has a useful array representation.

Strongly Balanced Binary Tree

- For each node, the left and right subtrees have heights that differ by at most 1.
- Importance. Height of entire tree is small. This can allow for fast operations.
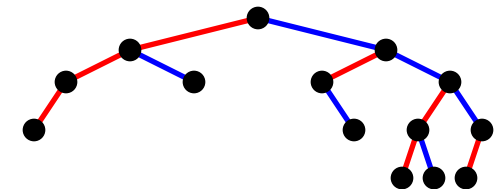
Every full Binary Tree is complete.

Every complete Binary Tree is strongly balanced.

To **traverse** a Binary Tree means to visit each node in some order.

Standard Binary Tree traversals: *preorder*, *inorder*, *postorder*. The name tells where the root goes: before, between, after.
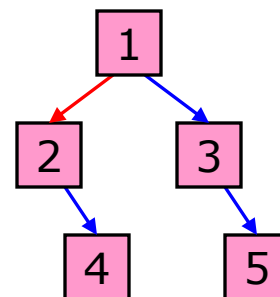
Preorder traversal:

- Root.
- Preorder traversal of left subtree.
- Preorder traversal of right subtree.

Inorder traversal:

- Inorder traversal of left subtree.
- Root.
- Inorder traversal of right subtree.

Postorder traversal.

- Postorder traversal of left subtree.
- Postorder traversal of right subtree.
- Root.

```
Preorder traversal:    1 2 4 3 5
Inorder traversal:     2 4 1 3 5
Postorder traversal:   4 2 5 3 1
```

A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.
- There *might* also be a pointer to the parent—if that would be helpful.

head

………………………………………………………

A *complete* Binary Tree can be implemented by simply putting the items in an array and keeping track of the size of the tree.
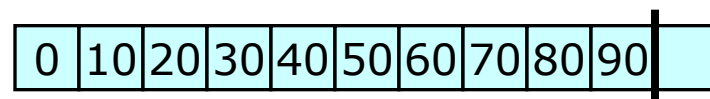
This implementation is very efficient (time & space), but it is only useful when the tree will *stay complete*.
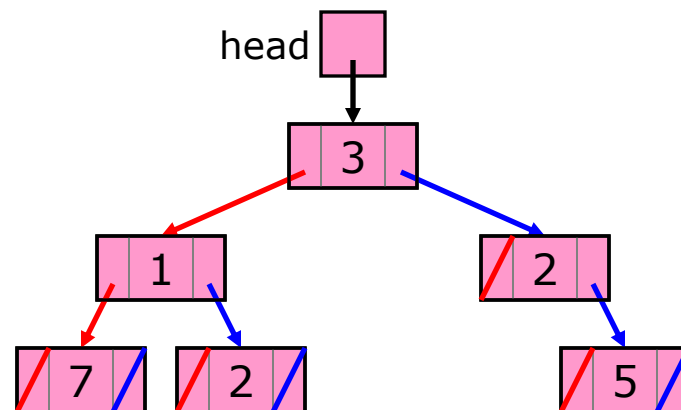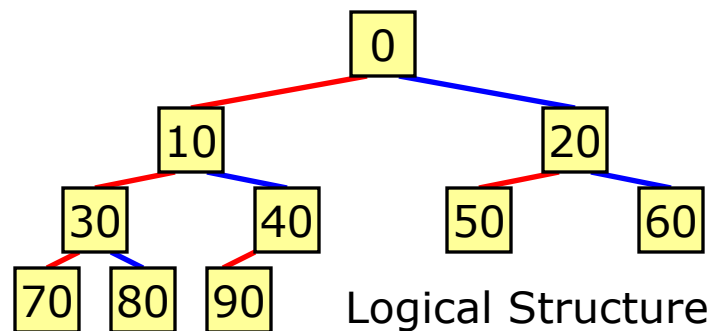
Logical Structure

| 0	10	20	30	40	50	60	70	80	90	

Physical Structure

A **Binary Search Tree** is a Binary Tree in which each node contains a single data item, which includes a **key**, and:

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

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

Must be ≤ x

x

Items ≤ x Items ≥ x
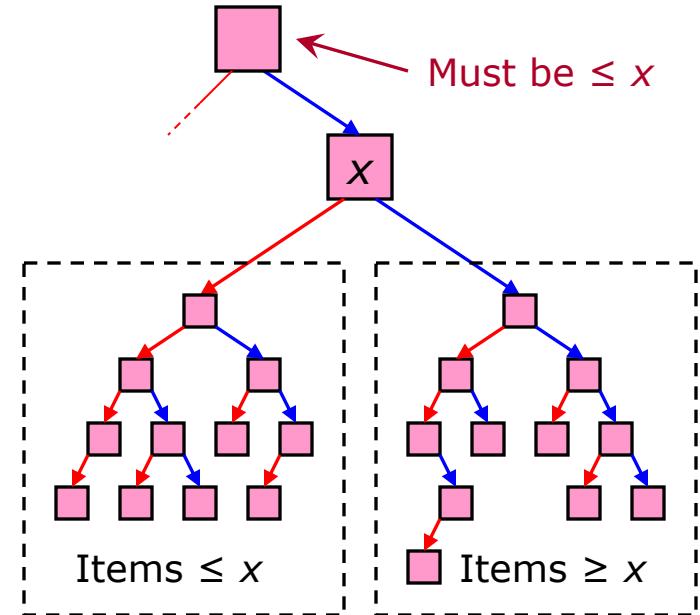
This is another value-oriented way to deal with data (while Binary Trees are position-oriented).

Binary Search Trees and SortedSequences are examples of **sorted containers**.

10

4   16

13   30

Inorder traversal:
4 10 13 16 30
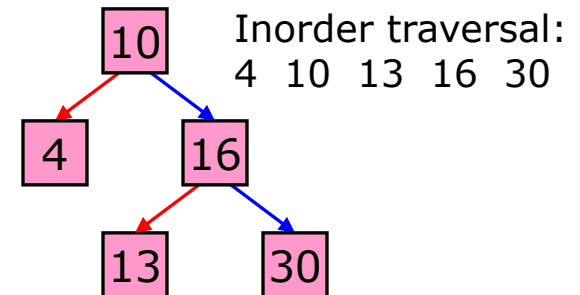
Three primary single-item Binary Search Tree operations:

- Retrieve
- Insert   *Here they are again.*
- Delete

To **retrieve** an item in a Binary Search Tree, given its key:

- **Search**—begin at the root and repeatedly follow left or right child pointers, depending on how the search key compares with the key in each node.
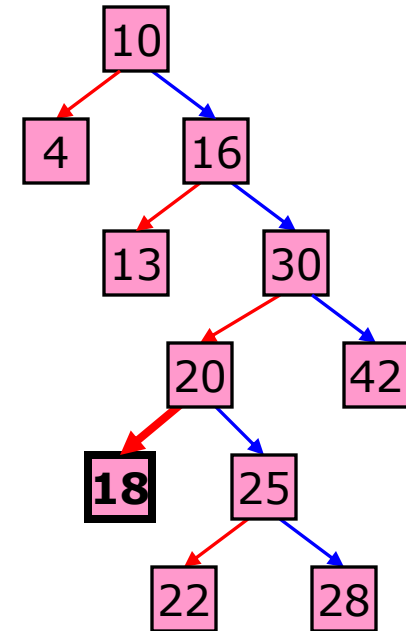
To **insert** a value with a given key:

- Find where the key *should* go (**search**).
- Put the data there.

Example. Insert key 18.

Binary Search Trees

continued

Delete is the most complex of the three single-item operations.

We will assume the key to be deleted is present in the tree.
Otherwise, again, the specification should tell us what to do.

Begin by finding the node holding the key to be deleted (**search**).

Then proceed to one of three cases, depending
on how many children this node has:

- No children (leaf).
- One child.
- Two children.

The **no-children** (leaf) case is easy:
Just remove the node.

Example. Delete key 28.
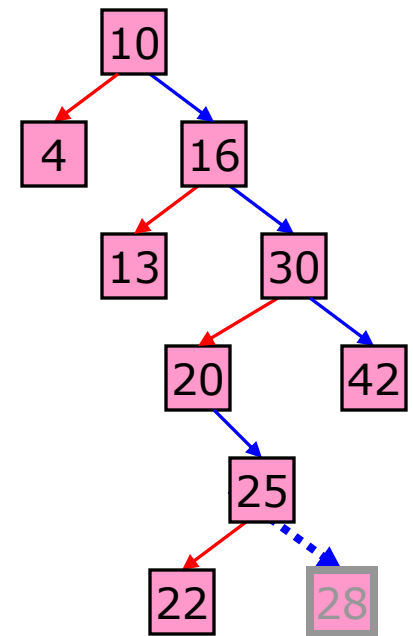
If the node to remove has exactly **one child**, replace the subtree rooted at the node with the subtree rooted at its child. (This is generally a constant-time operation, once the node is found.)

Example. Delete key 20.

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

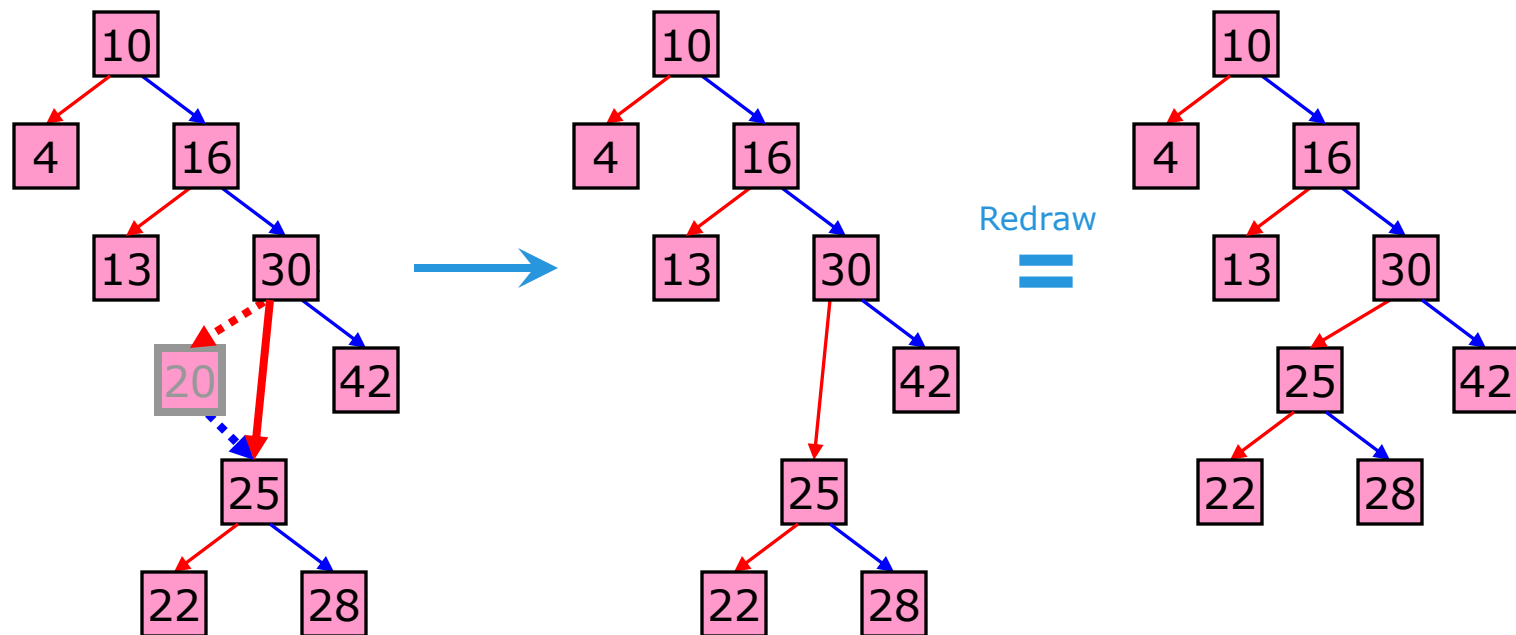- Replace its data with data is its *inorder successor* (copy or swap).
- Delete the inorder successor, which must have at most one child.

The **inorder successor** is the node that comes next in an inorder traversal, that is, the leftmost node in the node's right subtree.

To find the inorder successor:

- Go to right child.
- Then left child, left child, left child, … until you hit an empty spot.

The inorder successor of **this** node is **this** node.
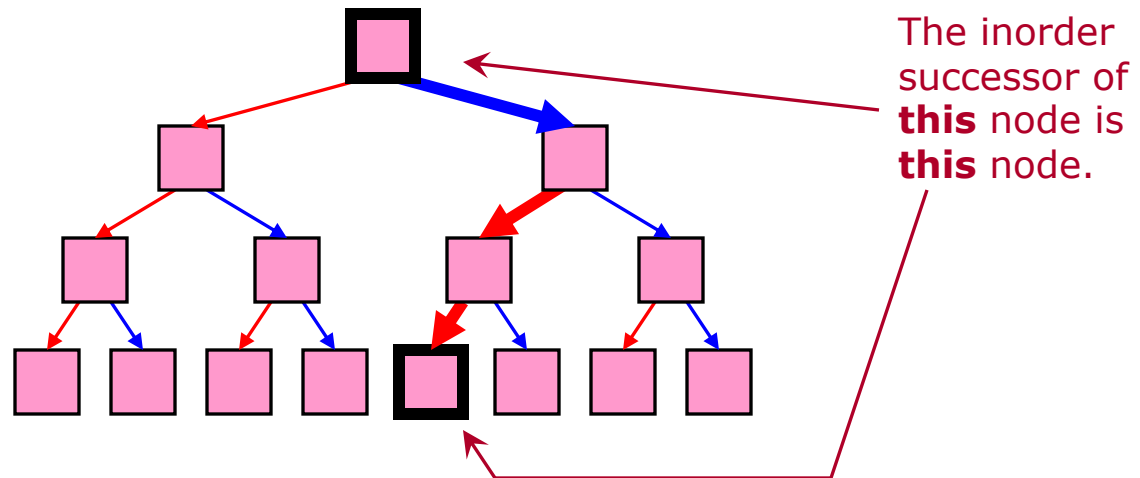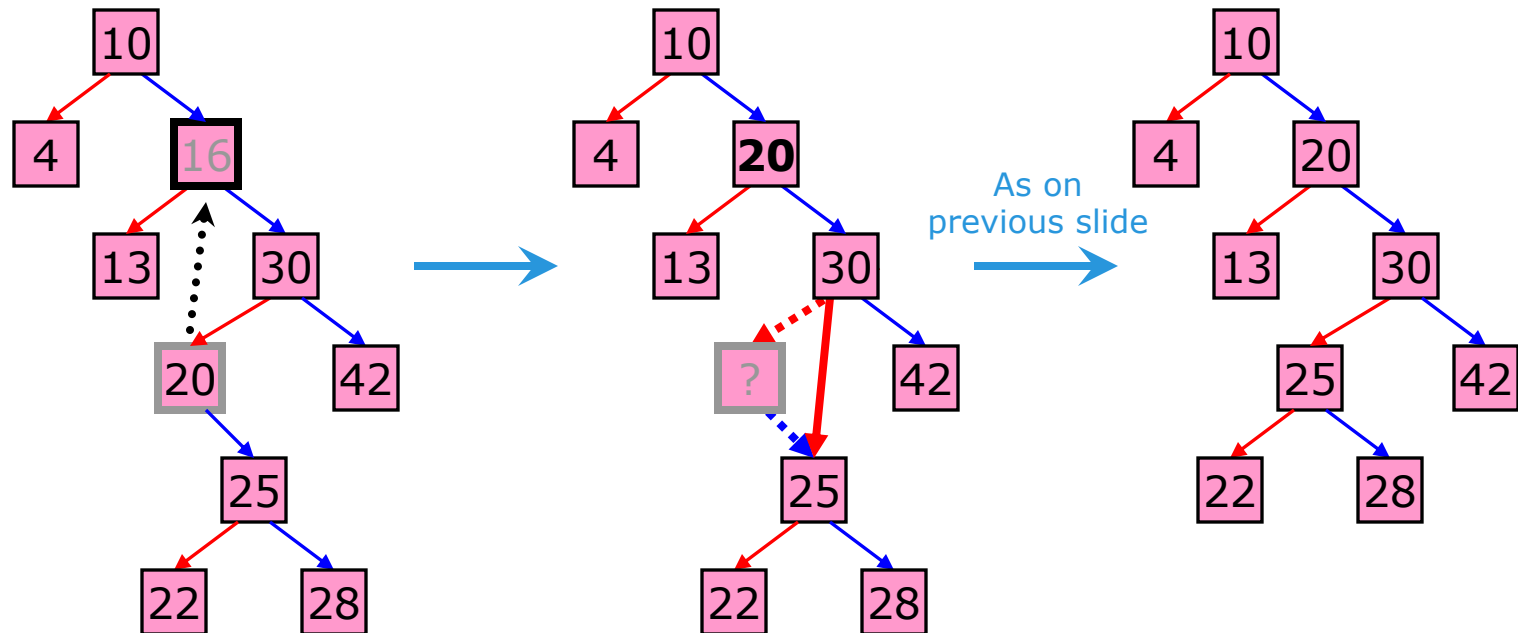
When the node to delete has **two children**:

- Replace its data with data is its inorder successor (copy or swap).
- Delete the inorder successor, which must have at most one child.

Example. Delete key 16.

Algorithms for the three primary single-item BST operations:

- Retrieve
  - Search. Start at root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - Search, then …
  - Put the value in the spot where it should go.
- Delete
  - Search, then …
  - Check the number of children the node has:
    - 0. Delete node.
    - 1. Replace subtree rooted at node with subtree rooted at child.
    - 2. Copy data from (or swap data 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**.

It turns out that the height of the tree is small (so all three operations are fast) if the tree is *strongly balanced*.

Do delete key 58 on the Binary Search Tree shown. Draw the resulting tree.
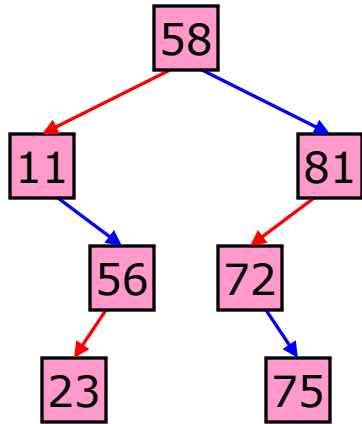
Answer on next slide.

Do delete key 58 on the Binary Search Tree shown. Draw the resulting tree.
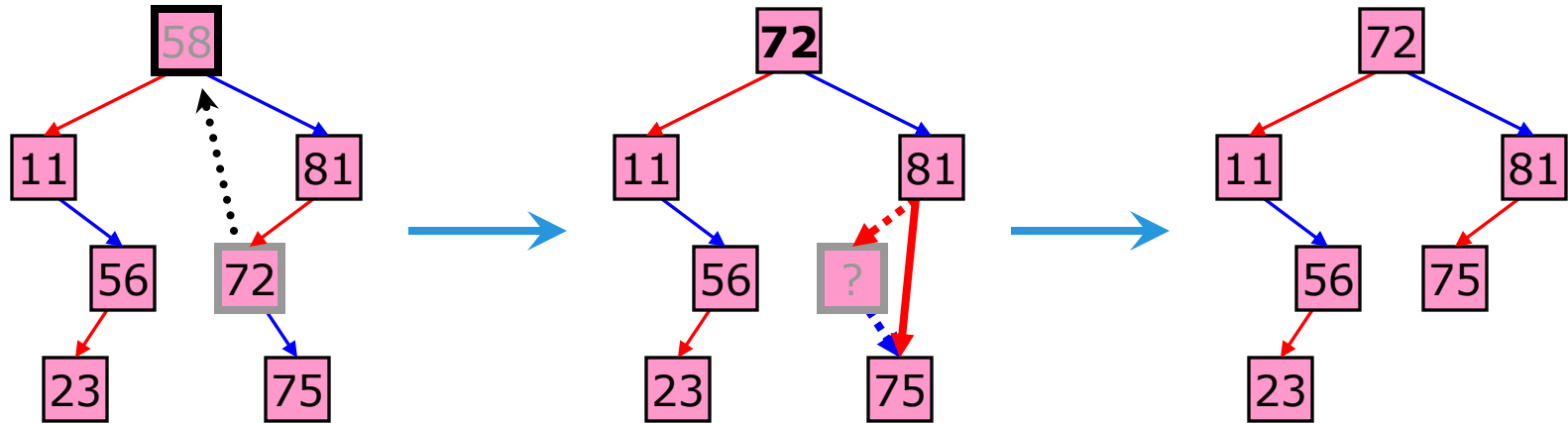
Procedure

- The 58 node is 2-children case.
- Find the inorder successor: the 72 node.
- Copy/swap 72 to the root node.
- Delete the old 72 node. This is 1-child case.
- Replace the subtree rooted at the old 72 node by the subtree rooted at its child (75).

BST retrieve, insert, and delete follow links down from the root.

- The number of steps is something like the height of the tree.
- In the worst case, the height of a tree is its size (number of nodes).
- But what about when the tree is strongly balanced?

Given the *size* of a strongly balanced Binary Tree, how *large* can its *height* be?

In order to answer this, first look at the reverse question: Given the *height* of a strongly balanced Binary Tree, how *small* can its *size* be? That is, what is the minimum size of a strongly balanced Binary Tree with height h?

Answer. Apparently, $F_{h+2} - 1$, for $h = 0, 1, 2, …$

- F_k is Fibonacci number k. $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_2 = 2$, etc.

It is not too hard to prove this using mathematical induction.

Back to the original question: Given the size of a strongly balanced Binary Tree, how large can its height be?

- We know that, if we have a strongly balanced Binary Tree with height h and size n, then $n \geq F_{h+2} - 1$.

- Fact: Let $\varphi = \dfrac{1+\sqrt{5}}{2}$. Then $F_k \approx \dfrac{\varphi^k}{\sqrt{5}}$. (Remember `fibo_formula.cpp`?)

- Thus, roughly: $n \geq \dfrac{\varphi^{h+2}}{\sqrt{5}}$.

- Solving for h, we obtain, roughly: $h \leq \log_\varphi\left(\sqrt{5}n\right) - 2$.

- We conclude that, for a strongly balanced Binary Tree, h is $O(\log n)$.

Even better, the height of a Binary Search Tree is, with high probability, $O(\log n)$ for random data. (We will not verify this statement.)

Order of the BST operations, using the algorithms discussed:

Retrieve

- Linear time.

- Worst case is roughly the height.
  - If strongly balanced: logarithmic time.
    But if we insert & delete, then it might
    not *stay* strongly balanced.
  - Logarithmic time on average for random data.

- Retrieve does not modify the tree. If that is all we do, we
  might want to create a strongly balanced tree beforehand.
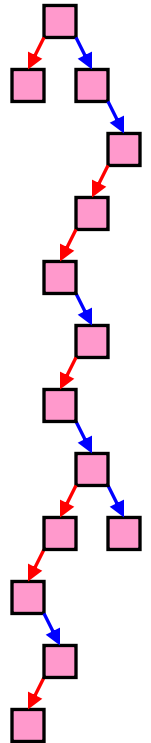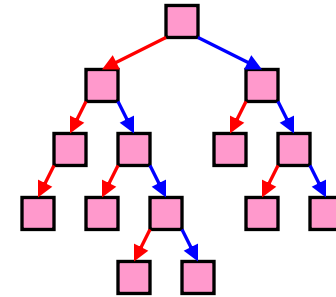
Insert

- Linear time (but see the second point under *Retrieve*).

Delete

- Linear time (but see the second point under *Retrieve*).

Traverse: inorder traversal

- Linear time.

	BST: strongly balanced OR average case	Sorted Array	BST: worst case
Retrieve	Logarithmic	Logarithmic	Linear
Insert	Logarithmic	Linear	Linear
Delete	Logarithmic	Linear	Linear

A BST has good average performance; *retrieve*, *insert*, and *delete* are logarithmic time for average data.

But in the worst case, a BST is worse than a sorted array.

Can we efficiently *keep* a Binary Search Tree strongly balanced, while allowing for insert & delete operations?

Keep this question in mind. We will eventually answer it.

Unit Overview
Tables & Priority Queues

Next we begin a unit on ADTs Table and Priority Queue and their implementations.

Major Topics

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

This will be the last *big* unit in the class. After this, we look briefly at other topics: external data, graph algorithms.

Introduction to Tables

Position-Oriented ADT

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

Examples

- Sequence
- Stack
- Queue
- Binary Tree

Value-Oriented ADT

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

Examples

- SortedSequence
- Binary Search Tree

Client code often only needs efficiency, so does not need to know how items are organized internally.

Can we do better here?

Table is a general value-oriented ADT, not tied to any particular implementation.

Table

Key	Value
12	Ed
4	Peg
9	Ann

Data

- A collection of items, each with a **key**.

Operations

- Single-Item Operations
  - **retrieve** by key.
  - **insert** item.
  - **delete** by key.

  Here they are yet again.

- Access All Data
  - **traverse**.
- The Usual
  - **create**, **destroy**, **copy**.
  - **isEmpty**.
  - **size**.

Table

Key	Value
12	Ed
4	Peg
9	Ann

Allow multiple items with the same key?

- It depends on the requirements of the client.

Require *traverse* to visit items in sorted order?

- Maybe. This is inefficient in some implementations.

Allow modification of data while it is in the Table?

- Modifying a **key** is problematic, since an item is generally located according to its key. Changing the key requires moving the item.
- Modifying the associate **value** is typically not a problem.

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

- Sure. Call it **Set**.

Conclusion

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

We have looked at restricted versions of a position-oriented ADT:

- Sequence allows retrieval/deletion at any position.
- Stack & Queue only allow retrieval/deletion at a single position—the highest (or lowest) position.

What about doing the same with value-oriented ADTs?

- Table allows retrieval/deletion using any key.
- Is there a useful ADT that only allows retrieval/deletion of the item with the highest key?

Yes! We call it *Priority Queue*. (More on this later.)

Introduction to Tables
Applications

What do we use a Table for?

- Data accessed by a **key** field. For example:
    - Customers accessed by phone number.
    - Students accessed by student ID number.
    - Any other kind of data with an ID code.
- **Set** data.
    - Each item has only a key, with no associated value.
    - Fundamentally, the only questions we can answer concern which keys lie in the dataset.
- Array-like datasets whose indices are not nonnegative integers.
    - `arr2["hello"] = 3;`
- Array-like datasets that are **sparse**.
    - `arr[6] = 1; arr[1000000000] = 2;`

What are possible Table implementations?

- A Sequence holding key-value pairs.
  - Array-based or Linked-List-based. ← Or deque—basically just a complicated array.
  - Sorted or unsorted.
- A Binary Search Tree holding key-value pairs.
  - Implemented using a pointer-based Binary Tree.

Table

Key	Value
12	Ed
4	Peg
9	Ann

Array Implementations

Linked List Implementations
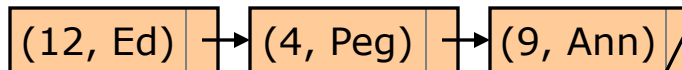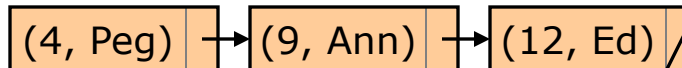
Binary Search Tree Implementation

Unsorted

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

Unsorted

(12, Ed) → (4, Peg) → (9, Ann)

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

Sorted

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

Sorted

(4, Peg) → (9, Ann) → (12, Ed)
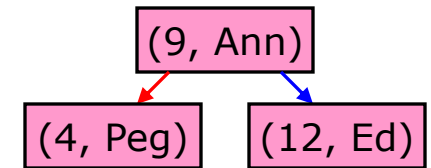
How efficient are these implementations?

Find the order of Table operations, for each implementation.

- Allow multiple equivalent keys, where it matters.
- If multiple equivalent keys are not allowed, then **insert** must first do a **search** (much the same as retrieve). The order of **insert** would thus be the order of **retrieve**+**insert** shown below.

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	*Strongly Balanced* BST?*
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	*Logarithmic*
Insert	Linear	Linear/ amortized constant**	Linear	Constant	Linear	*Logarithmic*
Delete	Linear	Linear	Linear	Linear	Linear	*Logarithmic*

Ick!

*We do not (yet?) know how to ensure that the tree will *stay* strongly balanced, unless we restrict ourselves to read-only operations (no insert, delete).

**Constant time if we have pre-allocated enough storage.

Tables can be implemented in many ways. Different implementations are appropriate in different circumstances.

In some 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 little or no deletion.

We will cover some sophisticated Table implementations. But remember that sometimes a simple technique is best.

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	*Strongly Balanced (how?) BST*
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	*Logarithmic*
Insert	Linear	Constant-ish	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 full generality.
- Do not allow retrieve & delete on *all* keys. Instead allow these operations only on the *greatest* key.

In practice: Priority Queue

Introduction to Tables
Implementation — Better Ideas? [2/3]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	*Strongly Balanced (how?) BST*
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	*Logarithmic*
Insert	Linear	Constant-ish	Linear	Constant	Linear	*Logarithmic*
Delete	Linear	Linear	Linear	Linear	Linear	*Logarithmic*

Idea #2. Keep a tree balanced

- Can we keep a Binary Search Tree strongly balanced—efficiently?
- Perhaps we could loosen the *strongly balanced* requirement? What we really need is *small height*.
- Loosen the *binary* requirement, too?

In practice: Self-balancing search trees

- 2-3 Tree & 2-3-4 Tree, Red-Black Tree
- AVL Tree
- B-Tree & variations (B+ Tree, etc.)

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	*Strongly Balanced (how?) BST*
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	*Logarithmic*
Insert	Linear	Constant-ish	Linear	Constant	Linear	*Logarithmic*
Delete	Linear	Linear	Linear	Linear	Linear	*Logarithmic*

Idea #3. Magic functions

- Consider a simple structure: unsorted array.
- Arrays have fast look-up by index. A *magic function* that tells a key's index might make *retrieve* very fast.
- What about *delete*? Idea: Allow *empty* items, so that *delete* does not need to move items down.
- *Retrieve/insert/delete* in (amortized?) constant time—maybe?

In practice: Hash Tables

Unit Overview
Tables & Priority Queues

Major Topics

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