Binary Search Trees continued
Introduction to Tables

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, November 8, 2023

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

Some material contributed by Chris Hartman

Unit Overview
The Basics of Trees

Topics

- ✓ Introduction to Trees
- ✓ Binary Trees
- (part) Binary Search Trees

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.

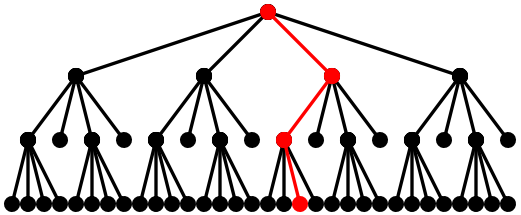
Note the three primary single-item operations: **retrieve**, **insert**, **delete**. We will see these over & over again.

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

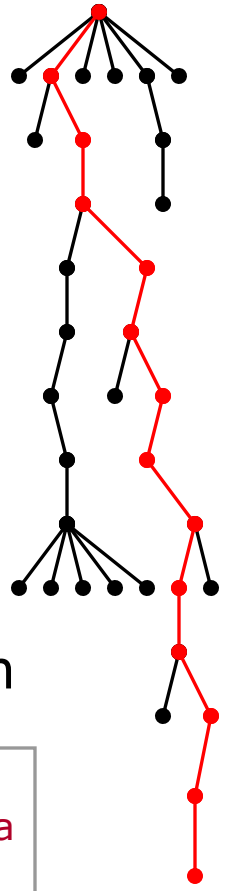
In a **generic container**, client code can specify the value type.

Informally, we might categorize trees as *bushy* or *stringy*.

Bushy Tree



Stringy Tree



When we use trees as the basis for data structures, we like bushy trees, because, while they may have a large number of vertices, paths between vertices are relatively short.

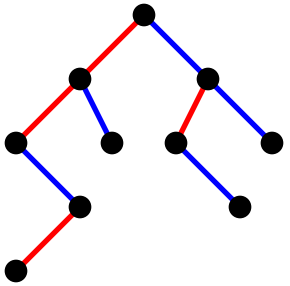
Suppose we can organize a tree so that some operation only works with a single path between vertices. If the tree is bushy, then the operation is fast.

This is how trees can be useful in data structure design.

Binary Trees — Definitions [1/2]

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**.



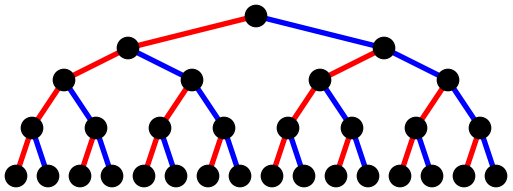
We make a strong distinction between left and right subtrees. Sometimes we use them for very different things.

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

Binary Trees — Definitions [2/2]

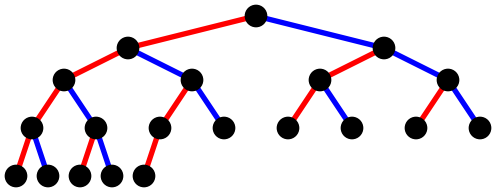
Full Binary Tree

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



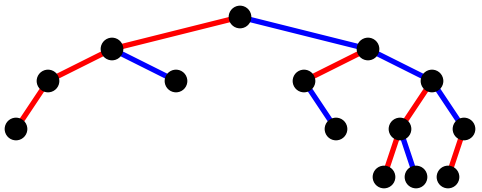
Complete Binary Tree

- 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. (An empty Binary Tree has height -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.

All three of these concepts can be useful notions of "bushy".

Binary Trees — Traversals

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:

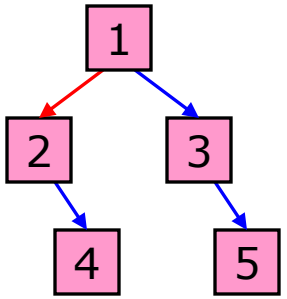
- Root.
- Preorder traversal of left subtree.
- Preorder traversal of right subtree.

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.

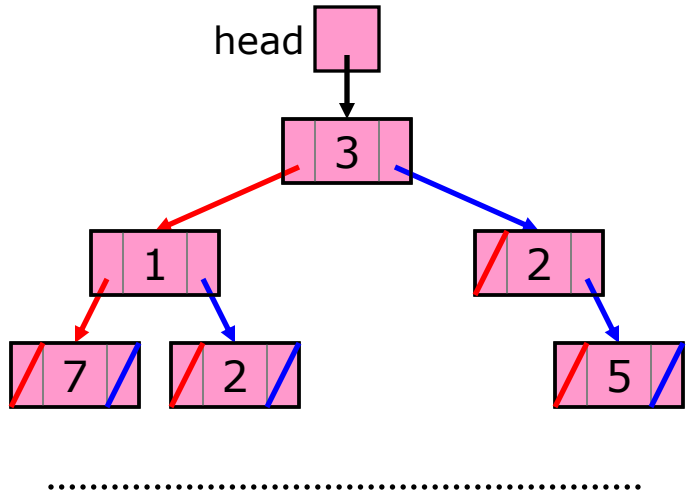


Preorder traversal: *???*
Inorder traversal: *???*
Postorder traversal: *???*

Review
Binary Trees — Implementation

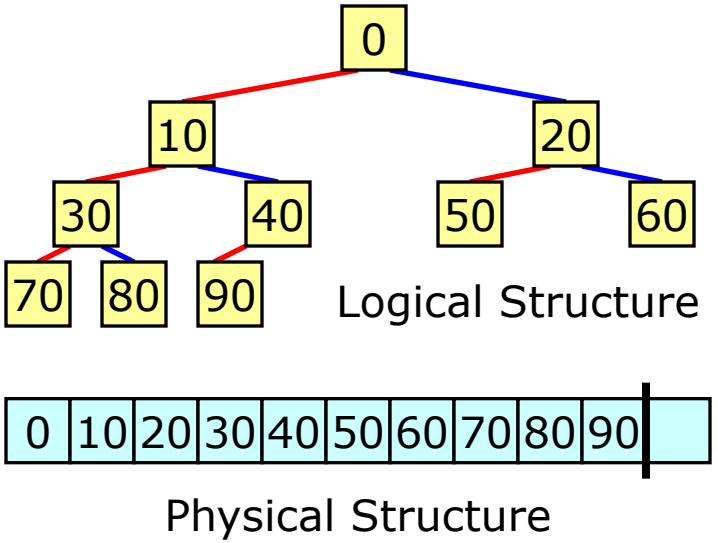
A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

- 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.



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*.

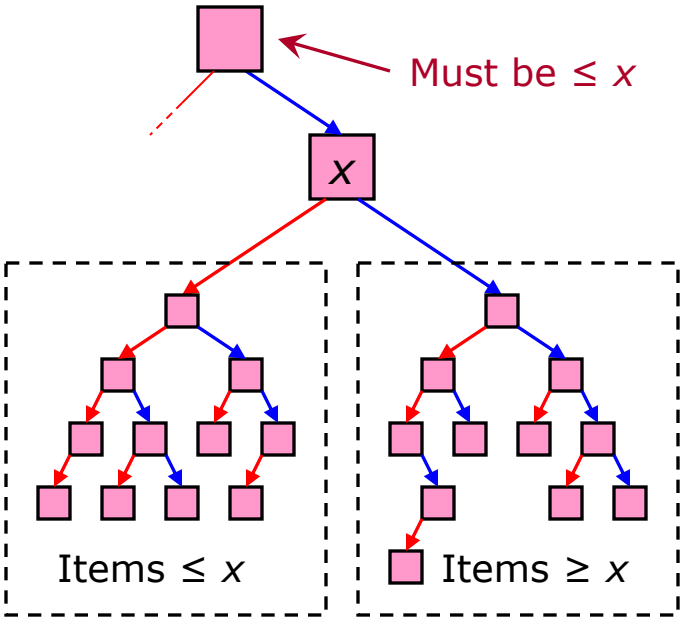


Binary Search Trees — What a Binary Search Tree Is

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

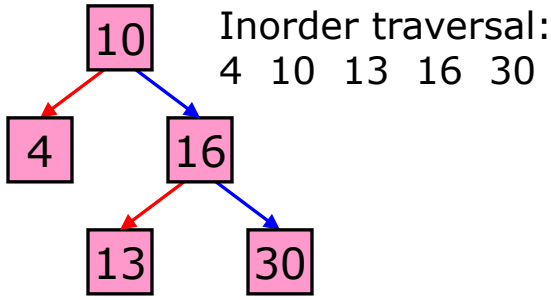
- Descendants holding keys less than the node's are in its left subtree.
- Descendants holding keys greater than the node's are in its right subtree.

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



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**.

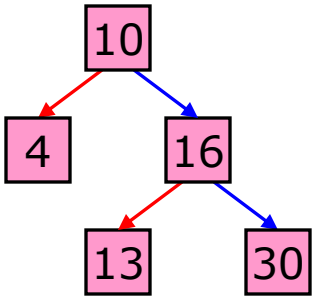


Binary Trees — Operations

Algorithms for the BST operations:

- Traverse
  - Recursively traverse left subtree of root.
  - Visit the root.
  - Recursively traverse right subtree of root.

  } Inorder Traversal
- Retrieve
  - **Search**. Start at the 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
  - *To be covered*



Binary Search Trees

continued

Operations — Delete [1/4]

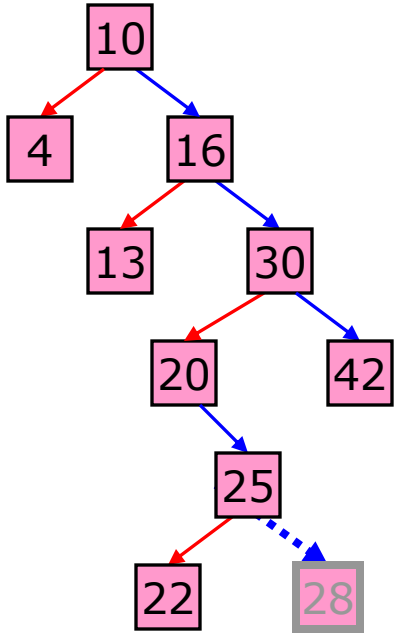
Delete is the most complex of the three single-item operations.

Here, we will assume the key to be deleted is present in the tree.
Otherwise, as before, 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.



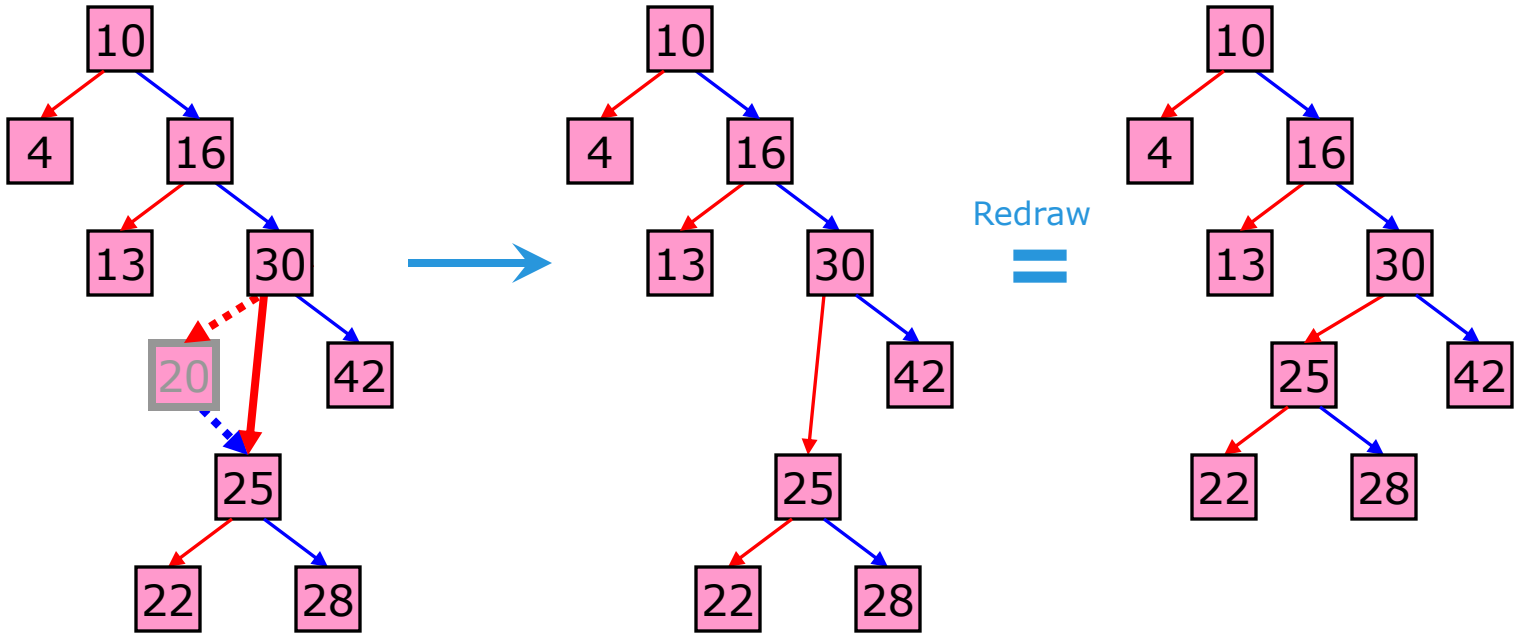
The **no-children** (leaf) case is easy:
Just remove the node.

Example. Delete key 28.

Binary Search Trees
Operations — Delete [2/4]

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.



Binary Search Trees
Operations — Delete [3/4]

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

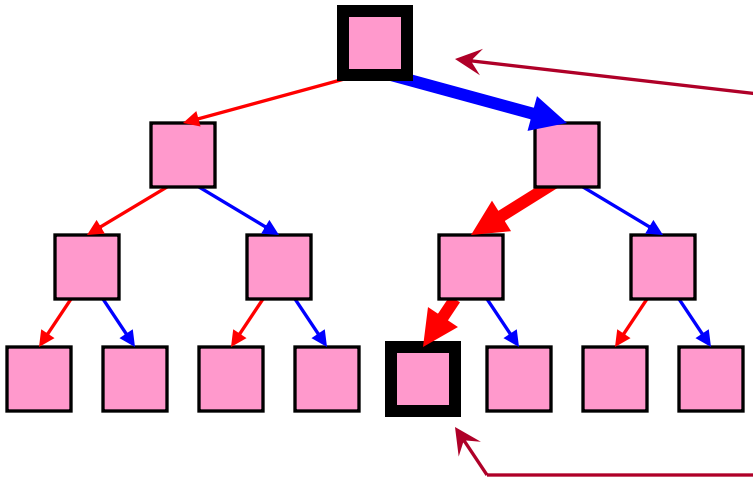
- Replace its data with data is its *inorder successor* (copy or swap).
- Delete the inorder successor, which must have at most one child.

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 of a node with two children:

- Go to the right child.
- Then left child, left child, left child, … until we reach a node that has no left child.

This is why the inorder successor must have at most one child.



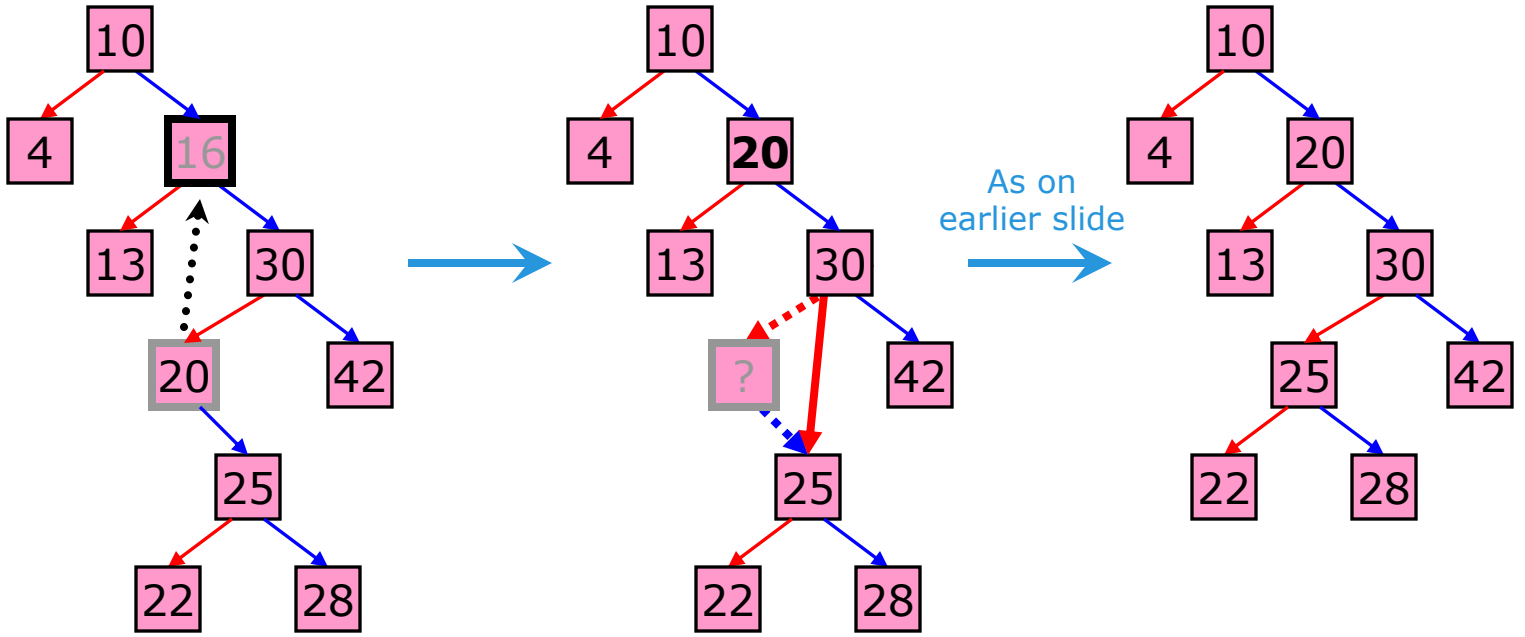
The inorder successor of **this** node is **this** node.

Binary Search Trees
Operations — Delete [4/4]

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.



Binary Search Trees
Operations — Summary & Thoughts

Algorithms for the three primary single-item BST operations:

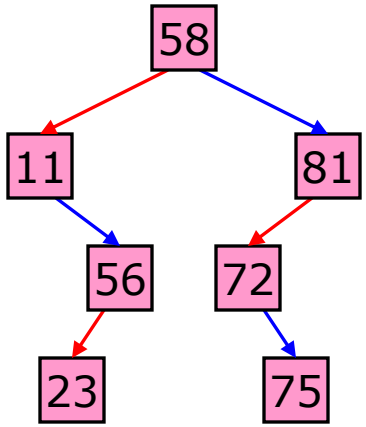
- Retrieve
  - **Search**. Start at the root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - *Search*, then …
  - Put the value in the spot where it should go.
- Delete
  - *Search*, then …
  - Check the number of children the node has:
    - 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 that is something like 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*.

Binary Search Trees
Operations — Try It! [1/2]

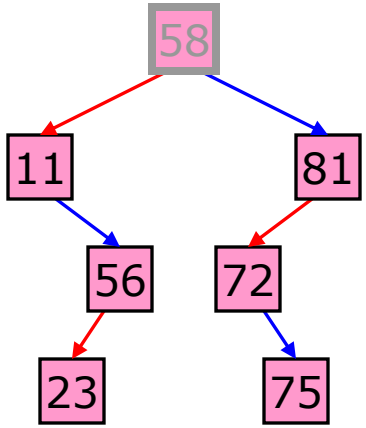
Do delete key 58 on the Binary Search Tree shown. Draw the resulting tree.



Answer on next slide.

Binary Search Trees
Operations — Try It! [2/2]

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



Procedure

…

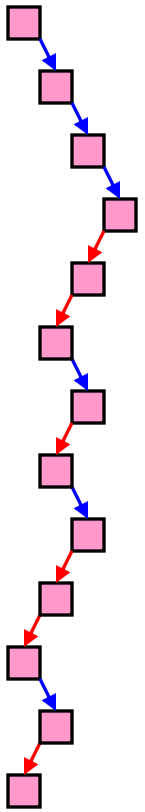
Binary Search Trees
Efficiency — Introduction

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

So these operations look at only a single path between vertices.

But can we make/keep a BST "bushy"?

Our notion of "bushy", for now, is *strongly balanced*.



The number of steps required for a BST single-item operation is something like the height of the tree.

Worst case: *height* = *# of nodes* – 1. ☹

But what about when the tree is strongly balanced?

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

Efficiency — Height of a Strongly Balanced Binary Tree [1/2]

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, *what???*

Binary Search Trees
Efficiency — Height of a Strongly Balanced Binary Tree [2/2]

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 $\varphi = \frac{1+\sqrt{5}}{2}$. Then $F_k \approx \frac{\varphi^k}{\sqrt{5}}$. (Remember `fibo_formula.cpp`?)
- Thus, roughly: $n \geq \frac{\varphi^{h+2}}{\sqrt{5}}$.
- 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. But we will not verify this statement.

Binary Search Trees
Efficiency — Order of Operations

Order of the BST operations, using the algorithms discussed:

Retrieve

???

Insert

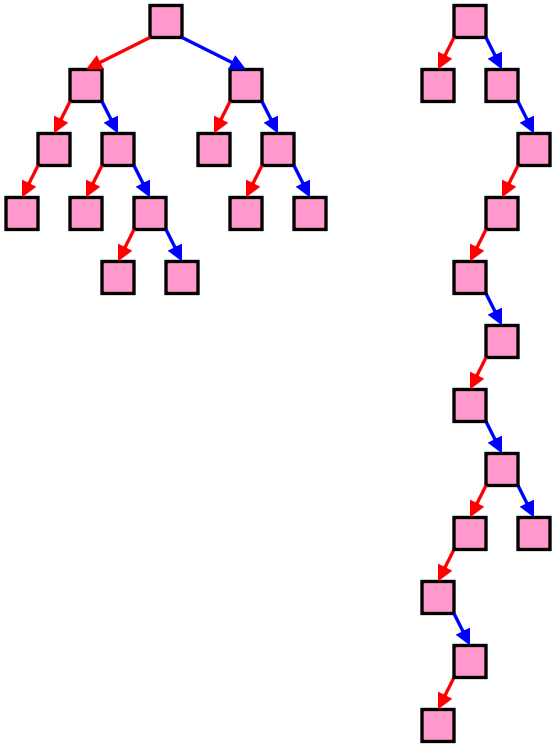
???

Delete

???

Traverse: inorder traversal

???



Binary Search Trees
Efficiency — A Problem

	BST: strongly balanced OR average case	Sorted Array	BST: worst case
Retrieve	*???*	*???*	*???*
Insert	*???*	*???*	*???*
Delete	*???*	*???*	*???*

Unit Overview
Tables & Priority Queues

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

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

Value-Oriented ADTs — Idea

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?

Value-Oriented ADTs — Table

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

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

Value-Oriented ADTs — Issues [1/2]

Table

Key	Value
12	Ed
4	Peg
9	Ann

Allow multiple items with the same key?

…

Require *traverse* to visit items in sorted order?

…

Allow modification of data while it is in the Table?

…

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

…

Value-Oriented ADTs — Issues [2/2]

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;`

*Or not sparse. Sequences can be stored as Tables. Indices become keys.

Introduction to Tables
Implementation — Possibilities [1/3]

What are possible Table implementations?

Table

Key	Value
12	Ed
4	Peg
9	Ann

Introduction to Tables
Implementation — Possibilities [2/3]

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	*???*	*???*	*???*	*???*	*???*	*???*
Insert	*???*	*???*	*???*	*???*	*???*	*???*
Delete	*???*	*???*	*???*	*???*	*???*	*???*

*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).

Implementation — Possibilities [3/3]

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!

Introduction to Tables
Implementation — Better Ideas? [1/3]

	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 #1. Restricted Table

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 (& bushy)

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

	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 #3. Magic functions

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
- Hash Tables

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