Binary Search Trees
Treesort

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, November 13, 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
The Basics of Trees

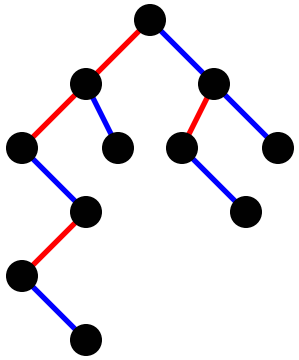
Major Topics

- ✓ Introduction to Trees
- ✓ Binary Trees
- Binary Search Trees
- Treesort

Binary Trees — What a Binary Tree Is

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

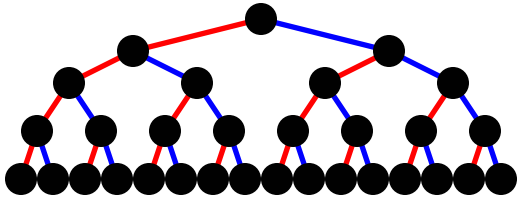
- *T* is empty (no nodes), or
- *T* consists of a node *r*, the root, and two subtrees of *r*, each of which is a Binary Tree:
  - the **left subtree**, and
  - the **right subtree**.



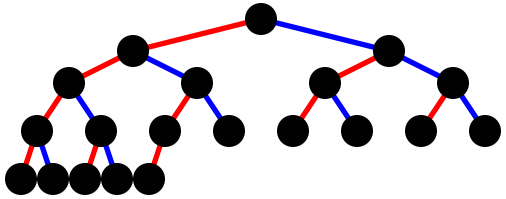
We make a strong distinction between left and right subtrees. Sometimes, we use them for very different things.

Review
Binary Trees — Three Special Kinds

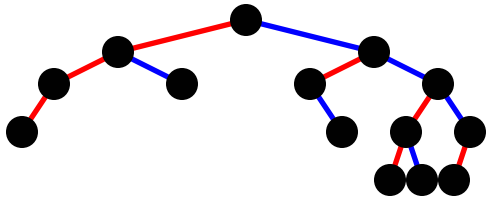
A typical **full** Binary Tree:



A typical **complete** Binary Tree:



A typical **balanced** Binary Tree:



A full Binary Tree is complete; a complete Binary Tree is balanced.

Full → Complete → Balanced

Review
Binary Trees — Traversals [1/3]

One thing we do with Binary Trees is to "traverse" them.

- **Traversing** a tree means visiting each node.

There are three standard traversals of Binary Trees: preorder, inorder, and postorder.

- The name tells us where the root goes: before, in between, after.

Preorder traversal:

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

Inorder traversal:

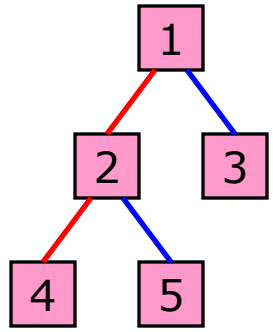
- Inorder traversal of left subtree.
- Root.
- Inorder traversal of right subtree.

Postorder traversal.

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

Binary Trees — Traversals [2/3]

Write preorder, inorder, and postorder traversals of the Binary Tree to the right.



Preorder: **1** **2** 4 5 3

root — left subtree — right subtree

Inorder: 4 **2** 5 **1** 3

left subtree — root — right subtree

Postorder: 4 5 **2** 3 **1**

left subtree — right subtree — root

Binary Trees — Traversals [3/3]

Given a drawing of a Binary Tree, draw a path around it, hitting the left, bottom, and right sides of each node, as shown.

The order in which the path hits the **left** side of each node gives the **preorder** traversal.

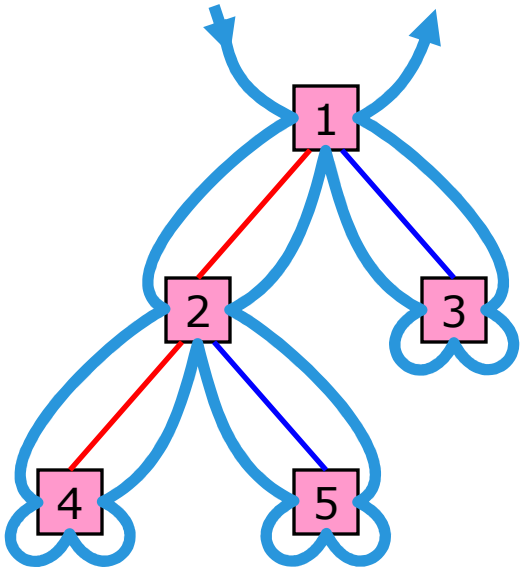
- 1 2 4 5 3

The order in which the path hits the **bottom** side of each node gives the **inorder** traversal.

- 4 2 5 1 3

The order in which the path hits the **right** side of each node gives the **postorder** traversal.

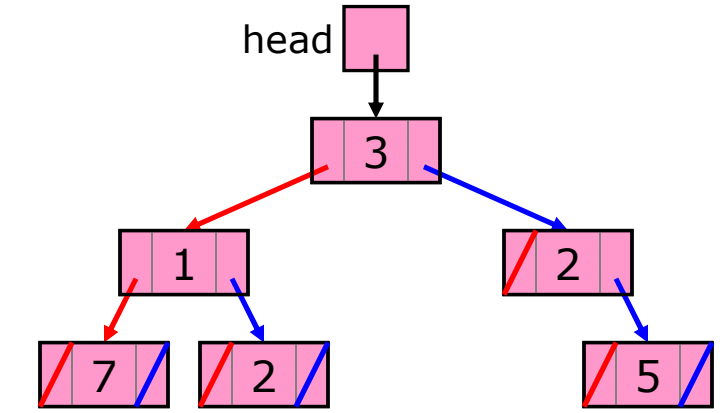
- 4 5 2 3 1



Binary Trees — Implementation

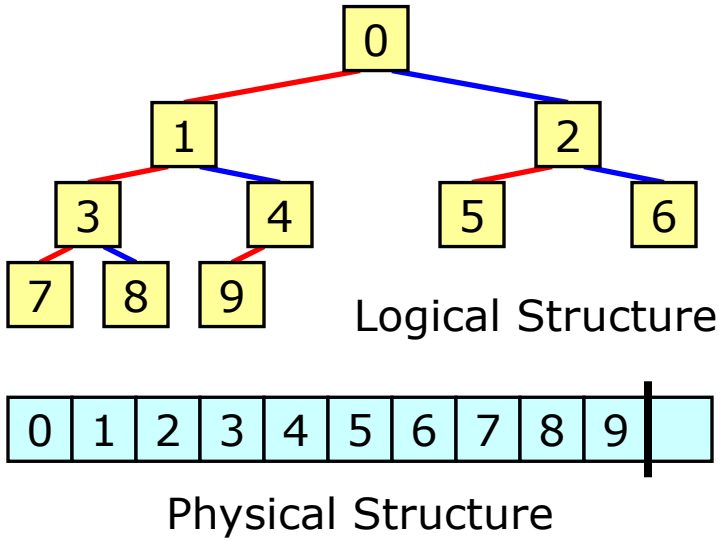
A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

- This is very similar to the way we implemented a Linked List.
- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.



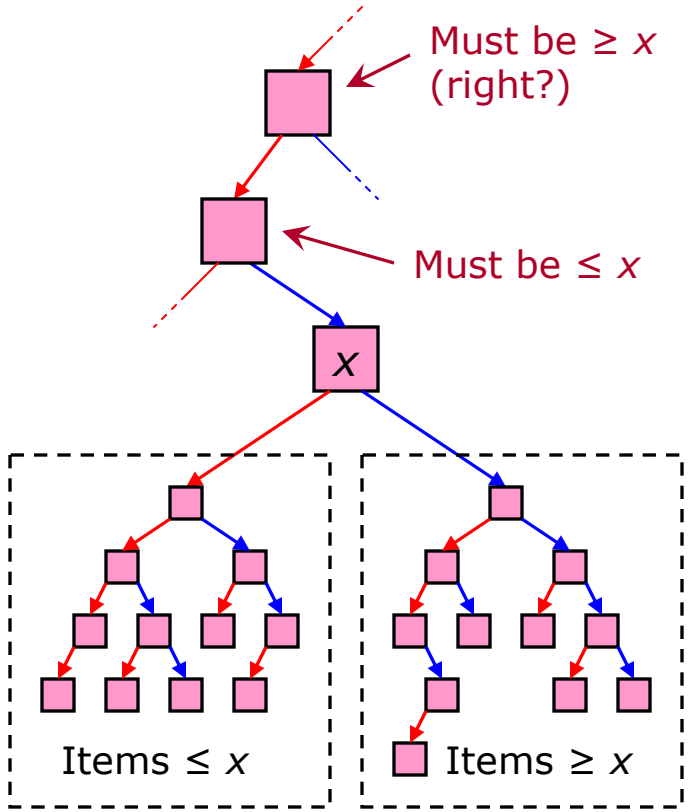
A **complete** Binary Tree can be implemented simply by putting the items in an array, and keeping track of the size of the tree.

- This is a nice implementation, but it is only useful in situations in which the tree *stays complete*.



Our final "application" of Binary Trees is our next ADT:
Binary Search Tree.

- Think: A Binary Tree in which each node's subtrees have an order relationship with the node.
  - All data items in a node's left subtree are ≤ the node's item.
  - All data items in a node's right subtree are ≥ the node's item.
  - Note: Sometimes we replace "≤" and "≥" by "<" and ">", so that items having equivalent values are not allowed.
- Thus, an inorder traversal lists items in sorted order.

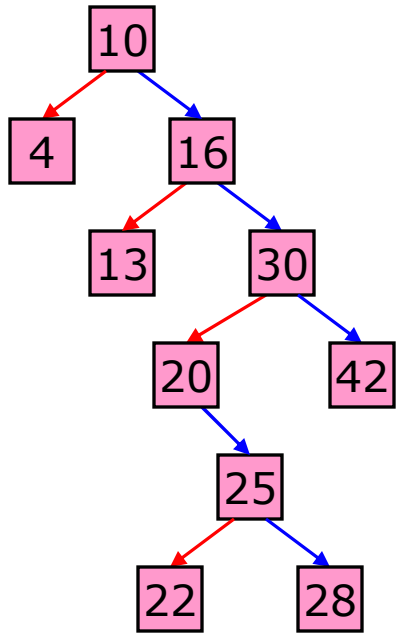


Binary Search Trees
ADT [1/3]

ADT Binary Search Tree

- Data
  - A collection of nodes, each with a data item.
- Operations
  - **create** empty B.S.T.
  - **destroy**.
  - **isEmpty**.
  - **insert** (given item [which includes a key]).
  - **delete** (given key).
  - **retrieve** (given key, returns item).
  - The three traversals: **preorderTraverse**, **inorderTraverse**, **postorderTraverse**.

Here they are again



Binary Search Trees
ADT [2/3]

This ADT is significantly simpler than Binary Tree.

What can you observe about this ADT if you remove the preorder & postorder traversals from the list of operations (leaving inorder)?

- The ADT no longer has anything to do with Trees.
- A Binary Tree **might** be a reasonable implementation, however.

Conclusion: A Binary Search Tree is essentially a big bag that things can be thrown in and retrieved from.

- The main questions we answer are "Is this key in the bag?", and if so, "What is the associated value?"

Binary Search Trees
ADT [3/3]

Recall the comments on the ADT SortedSequence:

- In practice, the ordering of a SortedSequence is often not of primary importance. Rather, we are interested in items being **easy to find**.

Binary Search Trees are similar.

- Sequence & Binary Tree are **position-oriented** ADTs.
- SortedSequence & Binary Search Tree are **value-oriented** ADTs.
- Binary Search Trees are another step toward a good value-oriented ADT, and implementation thereof.
  - But we can do both of these better, as we will see.

In value-oriented ADTs, data items have a "**key**".

- A *key* is the part of the data that is search for & compared.
  - This *might* be the entire value.
- Thus, two items whose *keys* do not compare as "different" are, for the purposes of inclusion in (say) a Binary Search Tree, identical.

Binary Search Trees
Operations — Introduction

Now we look at the details of B.S.T. operations.

Most of the B.S.T. operations are just wrappers around the essentially identical Binary Tree operations:

- create
- destroy
- isEmpty
- the three traversals
- copy (?)

Three of them, however, are specific to Binary Search Trees:

- retrieve
- insert
- delete

These three take advantage of the B.S.T. **invariants** (that is, the order property). They must also *maintain* these invariants.

We now look at how to implement these operations.

Binary Search Trees
Operations — Retrieve

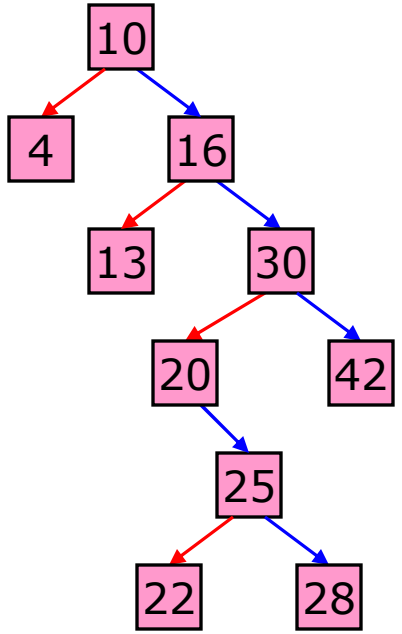
To **retrieve** a value from a Binary Search Tree, we begin at the root and repeatedly follow left or right child pointers, depending on how the search key compares to the key in each node.

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

- 20 > 10 → right.
- 20 > 16 → right.
- 20 < 30 → left.
- 20 = 20 → 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 → right.
- 18 > 16 → right.
- 18 < 30 → left.
- 18 < 20 → left.
- No left child → NOT FOUND.



Binary Search Trees
Operations — Insert

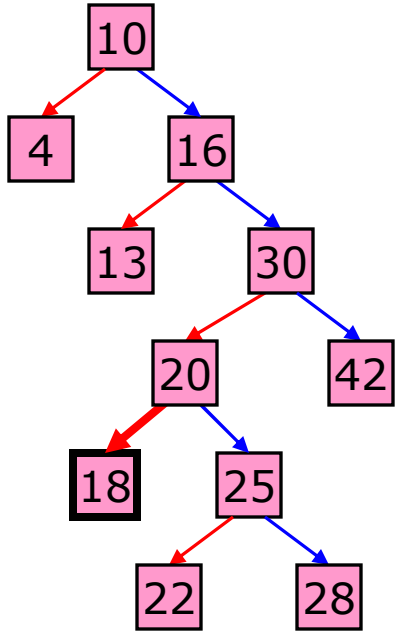
To **insert** a value with a given key, we find where the key "should" go (**retrieve** operation), and then we put the value there.

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

If our key turns out to be **in the tree already**, then our action depends on the specification of the Binary Search Tree.

- We may **add a new value** having the same key.
  - Result: multiple equivalent keys.
- Or we may **replace** the value with the given key.
- Or we may leave the tree **unchanged**.
  - If the last option is taken, we may wish to signal an **error condition**.

Note: Not just for Binary Search Trees! These are always the options, when we insert a duplicate key into a data set.



Operations — Delete [1/3]

Delete is the most complex of the three operations.

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

- Otherwise, once again, the spec's will tell us what to do.

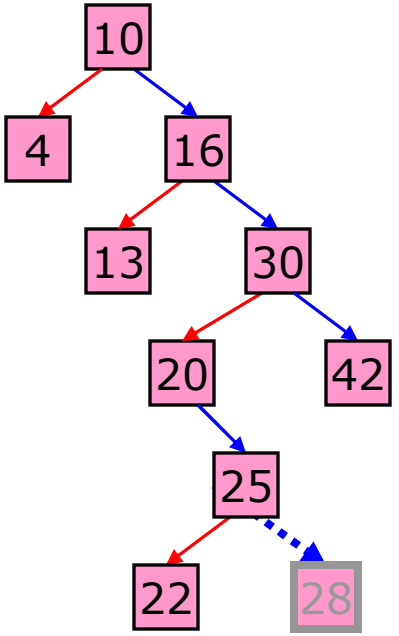
We begin by finding the appropriate node (**retrieve** operation).

Once we find it, 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:

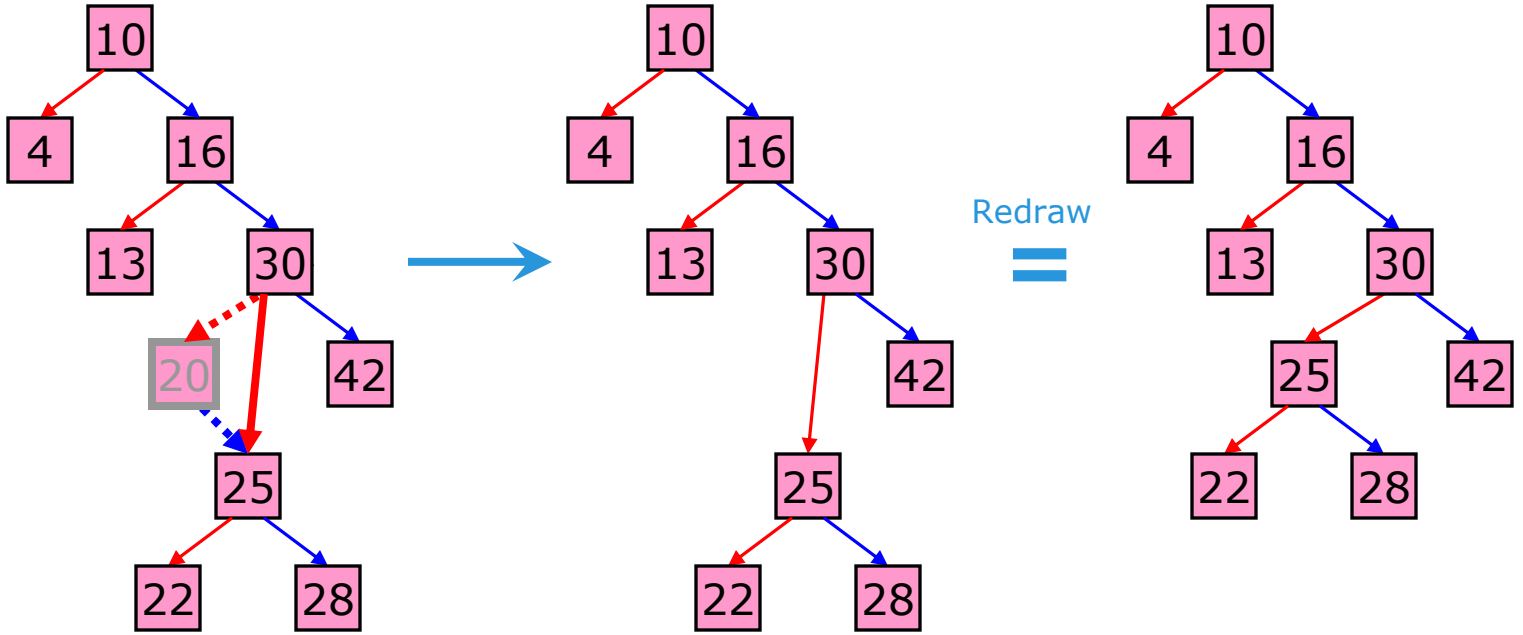
- Just delete the node, using the appropriate **Binary Tree** operation.
- Example: Delete key 28.



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.

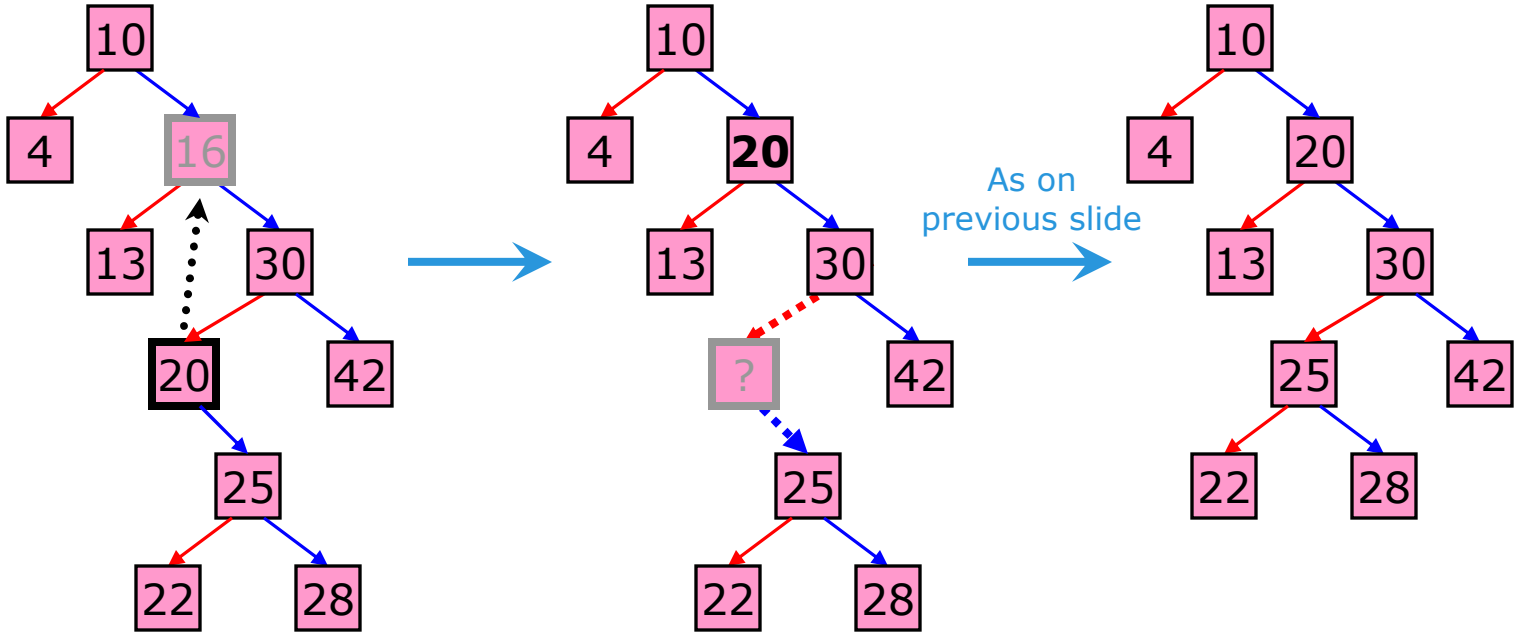
- This is a constant-time operation, once the node is found.
- Example: Delete key 20.



Binary Search Trees
Operations — Delete [3/3]

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

- Replace its data with data in its **inorder successor** (copy or swap).
- Delete the inorder successor, which has at most one child (right?).
- Example: Delete key 16.



Binary Search Trees
Operations — Summary & Thoughts

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

- Retrieve
  - Start at root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - Retrieve, then …
  - Put the value in 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 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**.

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

Binary Search Trees
Efficiency — Height of a Balanced Binary Tree [1/3]

B.S.T. insert, delete, and retrieve follow links down from the root.

- The number of steps required is usually something like the height of the tree.
- In the worst case, the height of a tree is its size. But what about when a tree is balanced?
- So: Given the **size** of a balanced Binary Tree, how **large** can its **height** be?

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

Answer (apparently): $F_{h+2} - 1$, for $h = 0, 1, 2, \ldots$

- 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.

Efficiency — Height of a Balanced Binary Tree [3/3]

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

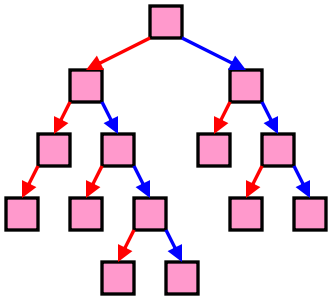
- We know that, if we have a balanced Binary Tree with height *h* and size *n*, then $n \geq F_{h+2} - 1$.
- Fact: Let $\varphi = \frac{1+\sqrt{5}}{2}$. Then $F_k \approx \frac{\varphi^k}{\sqrt{5}}$. (Remember **`fibo5.cpp`**?)
- Thus, roughly: $n \geq \frac{\varphi^{h+2}}{\sqrt{5}}$.
- Solving for *h*, we obtain, roughly: $h \leq \log_{\varphi}\left(\sqrt{5}n\right) - 2$.
- We conclude that, for a balanced Binary Tree, ***h* is *O*(log *n*)**.

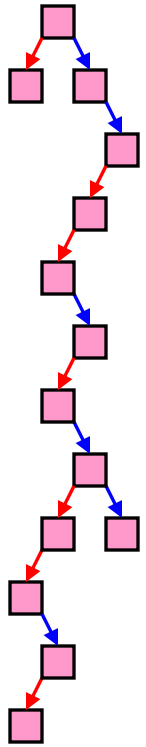
Even better, the height of a Binary Search Tree is, with high probability, *O*(log *n*) for **random data**.

- We will not verify this statement.

Binary Search Trees
Efficiency — Order of Operations

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.

Efficiency — The Problem

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

Further, a B.S.T. has good average performance; **retrieve**, **insert**, and **delete** 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)
Retrieve	Logarithmic	Logarithmic	Linear
Insert	Logarithmic	Linear	Linear
Delete	Logarithmic	Linear	Linear

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

- We will look at this question again later.

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 requires a large auxiliary data structure.

Treesort Analysis

Efficiency ☹

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

Requirements on Data ☺?*

- Treesort does not require random-access data.
- It works with Linked Lists.

Space Usage ☹

- Treesort requires $O(n)$ additional space for the tree.
  - And this space holds data items.

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.

*This is not much of an advantage for an algorithm that is inefficient in both time and space. (Suppose it did require random-access data. To fix this, we could simply start by copying to an array.)

Unit Overview
Tables & Priority Queues

Next we begin a unit on ADTs Table and Priority Queue and their implementations (in particular, Binary Heaps and the associated algorithms, balanced search trees, and Hash Tables).

Major Topics

- Introduction to Tables
- Priority Queues
- Binary Heap algorithms
- Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other balanced search trees
- Hash Tables
- Prefix Trees
- Tables in various languages

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

- External methods
- Graph algorithms