

Binary Trees

Binary Search Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, November 2, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2020 Glenn G. Chappell
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.

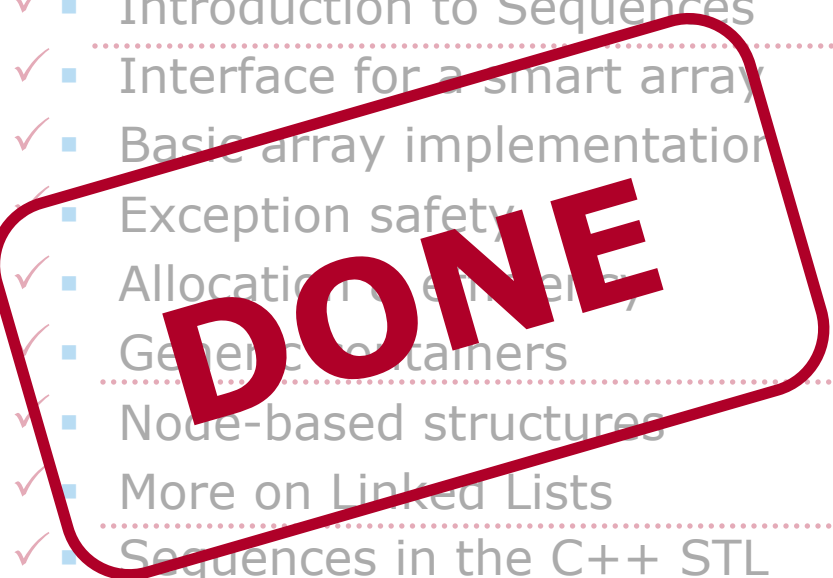
A solution to this problem is a **container**.

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

Unit Overview

Data Handling & Sequences

Major Topics

- ✓ ■ Data abstraction
 - ✓ ■ Introduction to Sequences
 - ✓ ■ Interface for a smart array
 - ✓ ■ Basic array implementation
 - ✓ ■ Exception safety
 - ✓ ■ Allocation & memory
 - ✓ ■ Generic containers
 - ✓ ■ Node-based structures
 - ✓ ■ More on Linked Lists
 - ✓ ■ Sequences in the C++ STL
 - ✓ ■ Stacks
 - ✓ ■ Queues
- Smart Arrays
- Linked Lists
- 

Review Stacks

Stack: another container ADT.
Restricted version of Sequence:
Last-In-First-Out (LIFO).

Three primary operations:

- **getTop**
- **push**
- **pop**

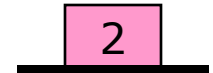
A Stack can be implemented simply
as a wrapper around some
existing Sequence type.

The STL has `std::stack`—a
container adapter. You can
choose the container. Default:
`std::deque`.

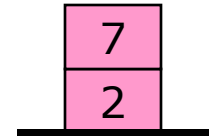
1. Start:
empty Stack.



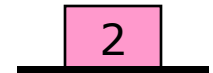
2. Push 2.



3. Push 7.



4. Pop.



5. Pop.
Stack is empty again.



6. Push 5.



Review

Queues [1/2]

Queue: another container ADT.
Restricted version of Sequence:
First-In-First-Out (FIFO).

Three primary operations:

- **getFront**
- **enqueue**
- **dequeue**

A Queue can be implemented:

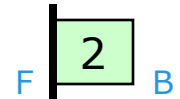
- As a wrapper around an *appropriate* Sequence type.
- Using a **circular buffer**.

The STL has `std::queue`—a **container adapter**. You can choose the container. Default: `std::deque`.

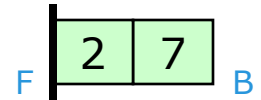
1. Start:
Empty Queue.



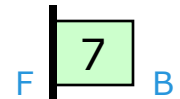
2. Enqueue 2.



3. Enqueue 7.



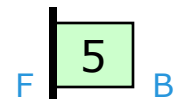
4. Dequeue.



5. Dequeue.
Empty again.



6. Enqueue 5.



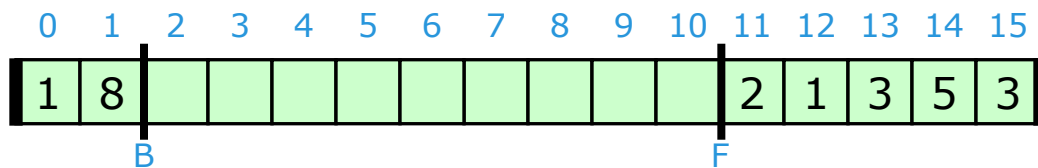
Review

Queues [2/2]

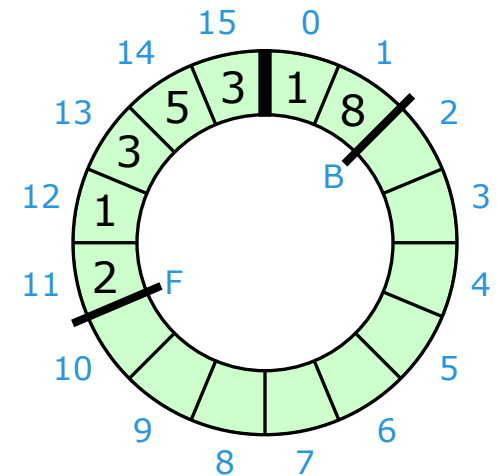
A **circular buffer** is an ordinary Sequence (probably an array) where we think of the ends as being joined.

We can implement a Queue using a circular buffer with markers indicating the front and back of the Queue.

If possible, when the Queue expands or contracts, we avoid resizing the underlying Sequence, and merely move the markers.



Physical Structure



Logical Structure

Unit Overview

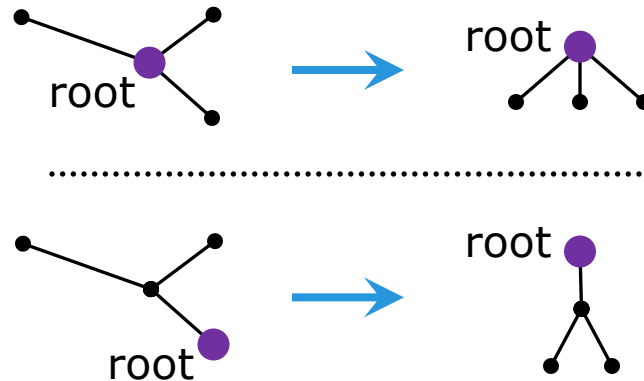
The Basics of Trees

Major Topics

- ✓ ■ Introduction to Trees
 - Binary Trees
 - Binary Search Trees

A **rooted tree** is a tree with one vertex designated as the **root**.

- When we *draw* a rooted tree, we usually place the root at the top. Each non-root vertex hangs from some other vertex.



For most of the rest of the semester, we will use **tree** to mean **rooted tree**. We will usually draw them as on the right, above.

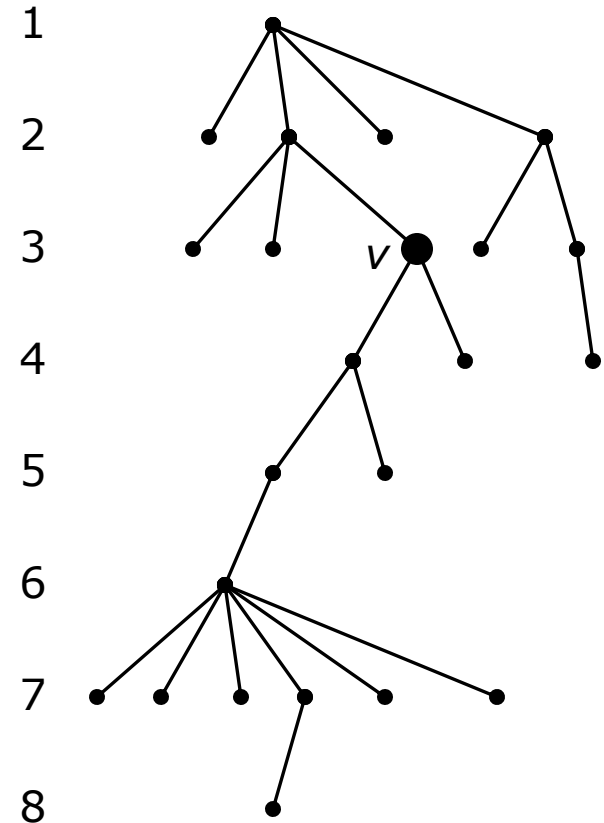
Review

Introduction to Trees [2/2]

Terminology for rooted trees:

- Root
- Leaf
- Parent
- Child
- Sibling
- Ancestor
- Descendant
- Level
- Height
- Subtree
 - The subtree *rooted at* vertex v
 - A subtree *of* vertex v

Know these!



Binary Trees

Binary Trees

Overview

Next we look at a special kind of (rooted) tree: a *Binary Tree*.

We cover:

- Definitions
- Traversals
- Implementation
- Applications

Q. What is missing above?

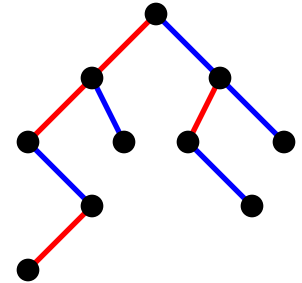
A. “Binary Trees in the C++ STL”, because there are none.

- That is, Binary Trees *in full generality* are not found in the STL *interface*.
- Binary Trees will certainly be used in any STL *implementation*.
- The STL interface does include a special kind of Binary Tree called a *Binary Heap*. We will cover Binary Heaps in our next unit.

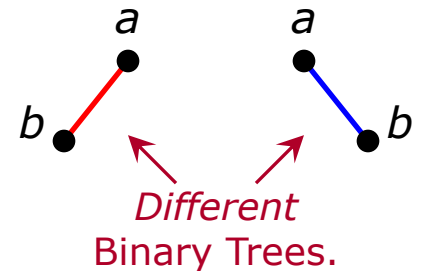
Binary Trees Definitions [1/2]

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.



A Binary Tree is **empty** if it has no nodes.
Note that, in a Binary Tree, the left and/or right subtree of a vertex may be empty.

Empty Binary Tree

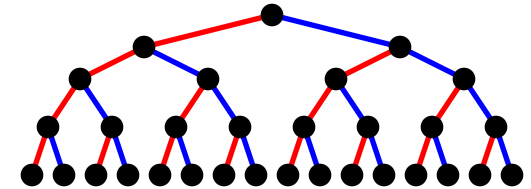


Binary Trees

Definitions [2/2]

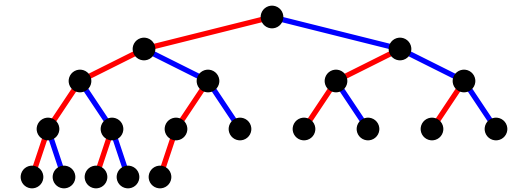
Full Binary Tree

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



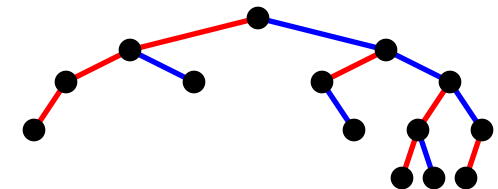
Complete Binary Tree

- 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.
- 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.

Binary Trees

Traversals — Idea

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:

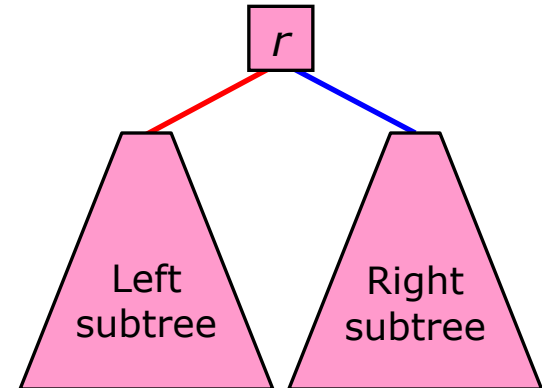
- Root.
- Preorder traversal of left subtree.
- Preorder traversal of right subtree.

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.



Binary Trees

Traversals — Example

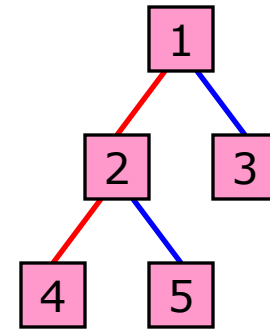
What are the preorder, inorder, and postorder traversals of the Binary Tree shown below?

Preorder: 1 2 4 5 3
 ↑ ↑ ↑
 Root Preorder Preorder
 of left of right
 subtree subtree

Inorder: 4 2 5 1 3
 ↑ ↑ ↑
 Inorder Root Inorder
 of left of right
 subtree subtree

Postorder: 4 5 2 3 1
 ↑ ↑ ↑
 Postorder Postorder Root
 of left of right
 subtree subtree

List the data items
in the order in which
they are visited.



Binary Trees

Traversals — A Trick

Given a drawing of a Binary Tree, trace a counter-clockwise path around it from upper left to upper right. Hit the left, bottom, and right sides of each node when the path passes them.

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

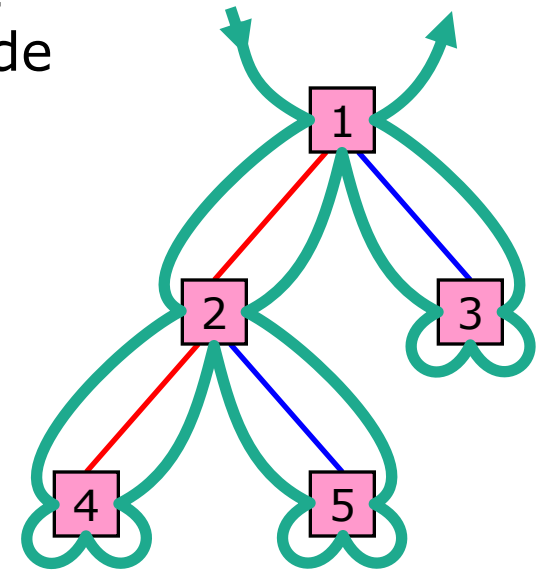
- 1 2 4 5 3

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.

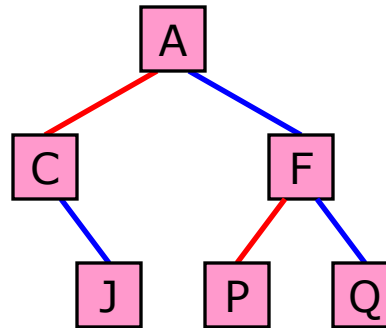
- 4 5 2 3 1



Binary Trees

Traversals — Try It! [1/2]

Write preorder, inorder, and postorder traversals of the Binary Tree shown below.

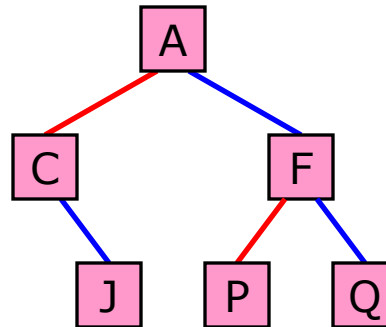


Answers on next slide.

Binary Trees

Traversals — Try It! [2/2]

Write preorder, inorder, and postorder traversals of the Binary Tree shown below.



Answers

Preorder: A C J F P Q

Inorder: C J A P F Q

Postorder: J C P Q F A

Binary Trees

Traversals — Expressions

Consider the Binary Tree at right.

- This is an *abstract syntax tree* of an expression (see CS 331).

Postorder traversal: 3 6 + 2 7 - *

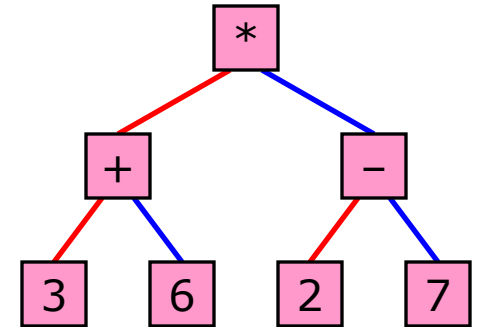
- This is the RPN form of the expression.

Inorder traversal: 3 + 6 * 2 - 7

- This looks like normal infix notation. However, *as an expression*, it is not what we mean; there are problems with precedence.
- Redo. Insert "(" before each subtree, and ")" after.
Result: (((3) + (6)) * ((2) - (7)))

Preorder traversal: * + 3 6 - 2 7

- This is (non-reversed) Polish notation.
- Now add parentheses & commas: $*(+(3, 6), -(2, 7))$
- Thinking of "*", "+", and "-" as names of functions, we can see that this is standard functional notation.
- A more familiar looking form: `times(plus(3, 6), minus(2, 7))`



Binary Trees

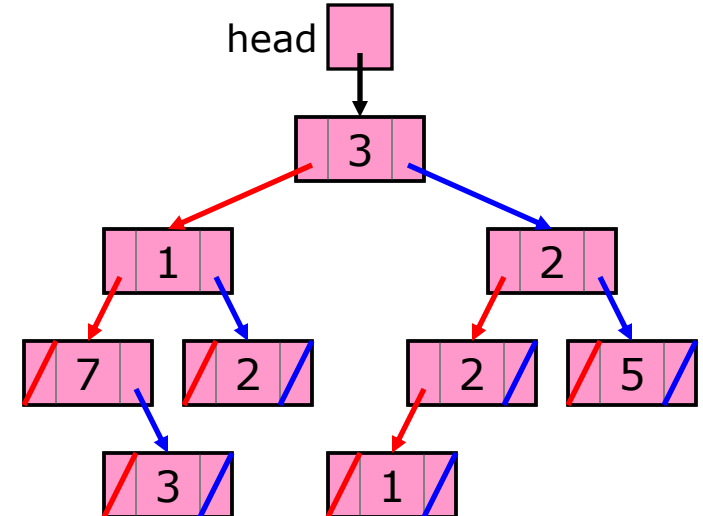
Implementation — Pointer-Based

A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers—similar to our implementation of a Linked List.

- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.

In some cases, each non-root node might keep a pointer to its parent.

- This would allow some operations to be much quicker.
- When do we do this? When it helps. That is, when the benefit gained by speeding up certain operations exceeds the cost of maintaining the additional pointer. This would depend on the purpose of the tree.

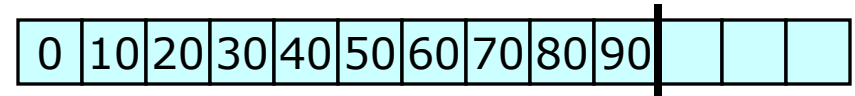
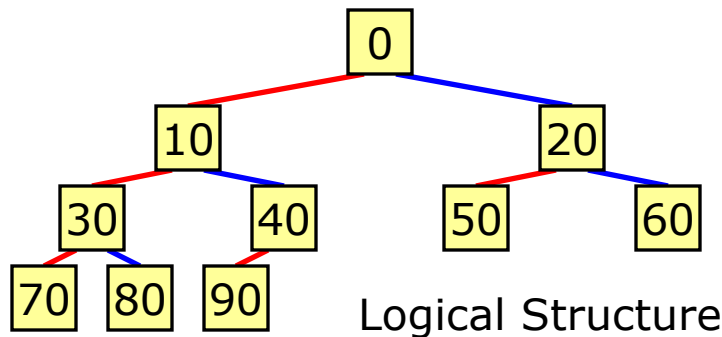


Binary Trees

Implementation — Array-Based Complete [1/2]

A *complete* Binary Tree can be stored efficiently in an array.

- Put the root, if any, at index 0. Other items follow in left-to-right, then top-to-bottom order.



We only store:

- Array** holding data items.
- Tree **size** (number of items).

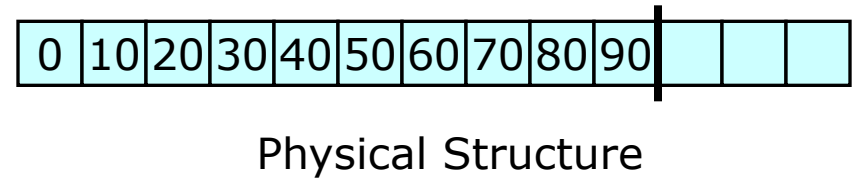
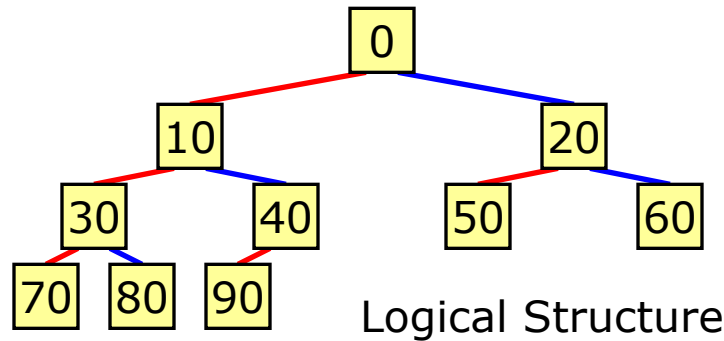
No pointers required!

This greatly limits the operations available, since we must preserve the property of being complete.

A *Binary Heap*—covered later—is implemented using this idea.

Binary Trees

Implementation — Array-Based Complete [2/2]



Without pointers, how do we move around in the tree?

- The **root**, if any, is at index 0.
 - The root exists if $size > 0$, that is, if the tree is nonempty.
- The **left child** of node k is at index $2k + 1$.
 - The left child exists if $2k + 1 < size$.
- The **right child** of node k is at index $2k + 2$.
 - The right child exists if $2k + 2 < size$.
- The **parent** of node k is at index $(k - 1)/2$ [integer division].
 - The parent exists if $k > 0$.

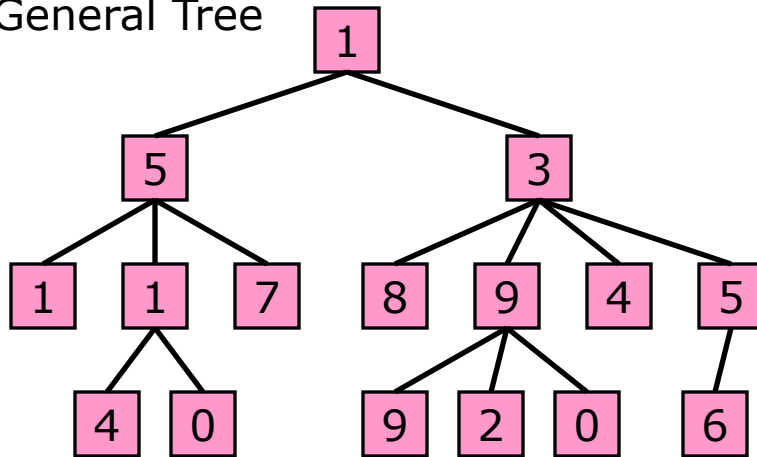
Binary Trees

Applications — Representing General Trees

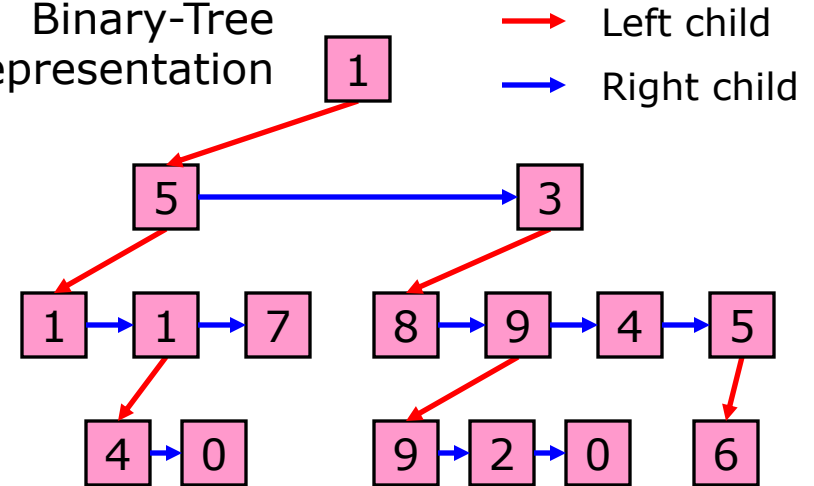
We can represent *any* tree using a Binary Tree.

- The left child of a node is its first child in the general tree.
- The right child of a node is its next sibling in the general tree.

General Tree




Binary-Tree Representation



This makes the children of each node into a Linked List, with all the associated advantages & disadvantages.

Binary Trees

Applications — Recursive Lists

Binary Trees with data only in the leaves can represent **recursive lists**: lists whose items may be lists. 

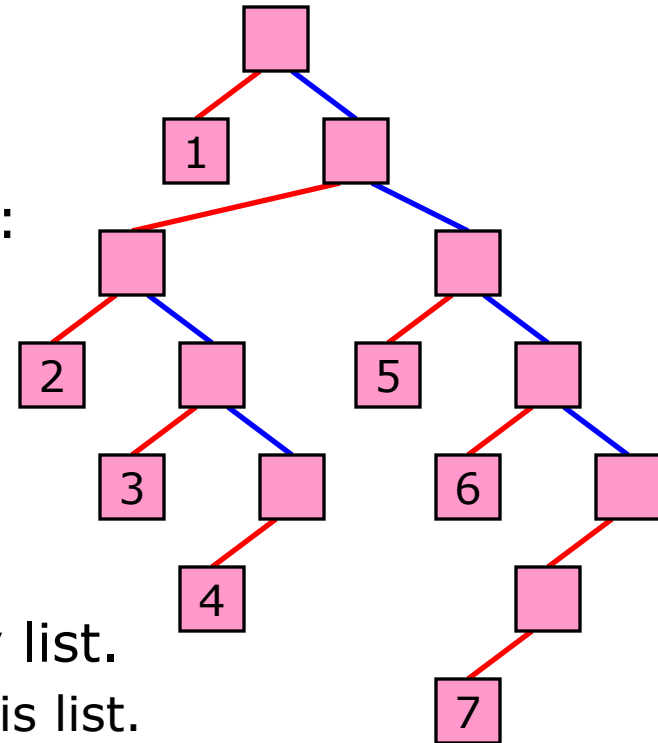
Example. The list (1 (2 3 4) 5 6 (7)) is represented by the tree shown. Its items:

- 1
- the list (2 3 4)
- 5
- 6
- the list (7)

A node with no data represents a nonempty list.

- Its left child represents the first item in this list.
- If there are more items, then the right child represents a list of these.

Binary Trees used in this way form the primary data structure in the Lisp family of programming languages.



Binary Search Trees

Binary Search Trees

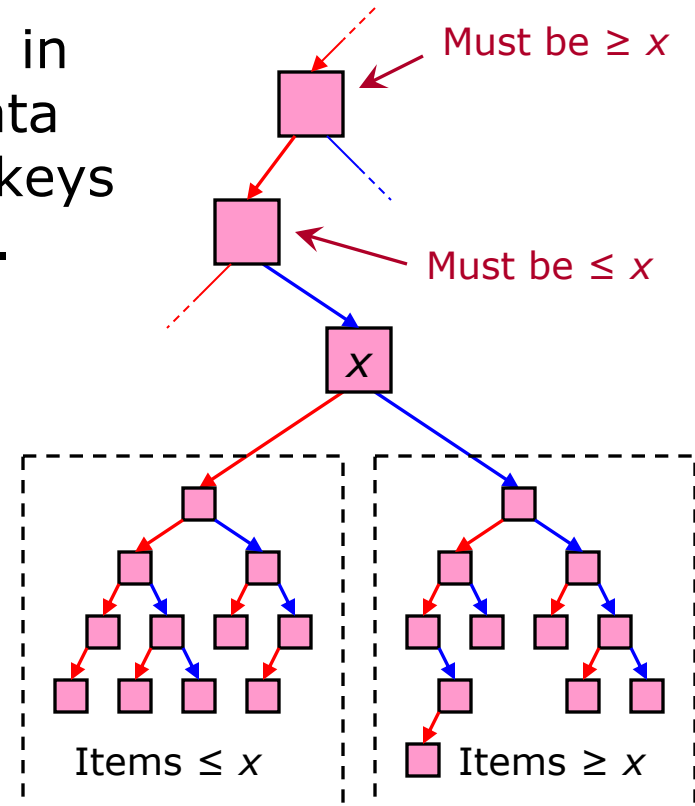
What a Binary Search Tree Is — Definition

Another application of Binary Trees is our next topic:
Binary Search Trees.

A **Binary Search Tree** is a Binary Tree in which each node contains a single data item, which includes a **key**, and the keys have the following order relationship.

- All keys in a node's left subtree are \leq the node's key.
- All keys in a node's right subtree are \geq the node's key.
- In other words, an inorder traversal gives the keys in sorted order.

A Binary Search Tree is a kind of **sorted container**. Other kinds include arrays & Linked Lists that are kept sorted.

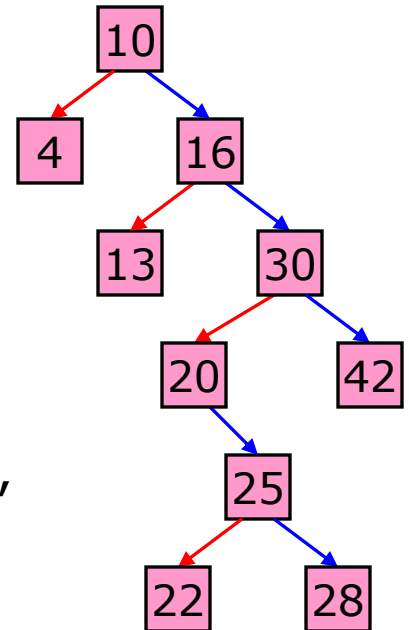


Binary Search Trees

What a Binary Search Tree Is — ADT?

We might consider Binary Search Tree as an ADT.

- When we insert a key, we do not get to choose where it goes.
- So we look up items by value, not by position.
- Binary Search Tree, like SortedSequence, would be a value-oriented ADT. The point is to make keys *easy to find*.
- Binary Tree, like Sequence (and Stack and Queue), would be a position-oriented ADT.



Position-Oriented ADT	Corresponding Value-Oriented ADT
Sequence	SortedSequence
Binary Tree	Binary Search Tree

Like SortedSequence, Binary Search Tree is inadequate as a *value-oriented* ADT. Both constrain the form of data in ways that are irrelevant to its use.

Binary Search Trees

Operations — Introduction

What operations do we perform on a Binary Search Tree?

Primarily those involving key-based access with arbitrary keys:
traverse, retrieve, insert, delete.

We already know how to **traverse** a BST: do an inorder traversal exactly as for a Binary Tree.

We have not covered the three primary single-item operations:

- **Retrieve**
- **Insert**
- **Delete**

These three can take advantage of the BST order property. When altering a BST, they must also *maintain* this property.

We now look at algorithms to implement these operations.

Binary Search Trees

Operations — Retrieve

To **retrieve** a value in a Binary Search Tree, 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.

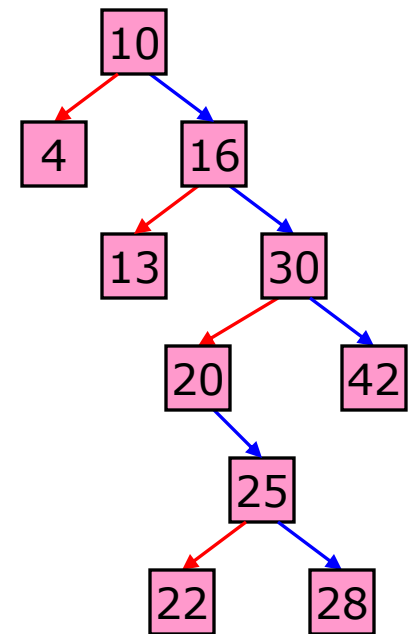
For example, retrieve key 20 in the tree shown:

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

A given key may not lie in the tree. Stop if we find an empty spot where the key *should* go.

Retrieve 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.



Binary Search Trees

Operations — Insert

To **insert** a value with a given key:

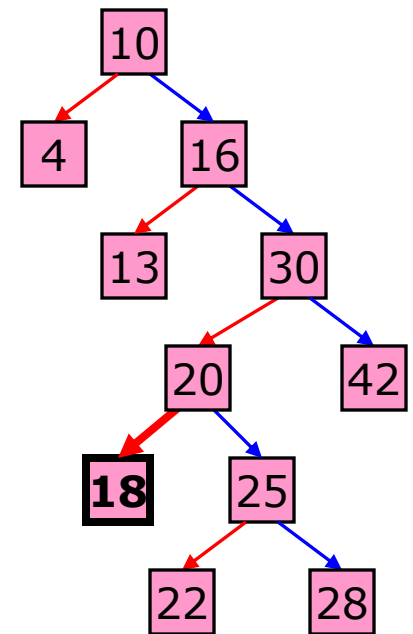
- Find where the key *should* go (do a **search**—much like retrieve).
- Put the data there.

For example, we just found where 18 should go. Insert it.

The above works—unless the search finds an equivalent key. In that case, our action depends on the specification of the BST.

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

Not just for Binary Search Trees! These are always the options when inserting a duplicate key into an associative dataset.



Binary Search Trees

TO BE CONTINUED ...

Binary Search Trees will be continued next time.