Introduction to Trees
Binary Trees

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, November 11, 2009

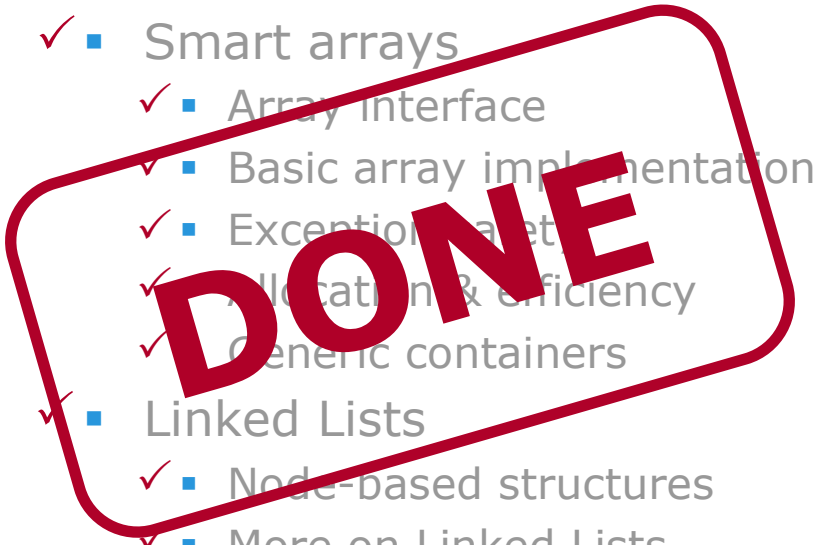
Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell

Unit Overview
Handling Data & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Smart arrays
  - ✓ Array interface
  - ✓ Basic array implementation
  - ✓ Exception safety
  - ✓ Allocation & efficiency
  - ✓ Generic containers
- ✓ Linked Lists
  - ✓ Node-based structures
  - ✓ More on Linked Lists
- ✓ Sequences in the C++ STL
- ✓ Stacks
- ✓ Queues



Review
Stacks

A Stack is:

- A kind of container.
- A Last-In-First-Out (LIFO) structure.
- A restricted version of a Sequence.

Conceptually, a Stack carries out the idea of **top-down design**.

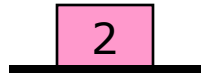
Three primary operations:

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

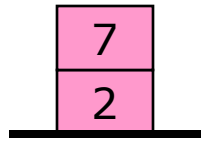
A Stack can be implemented simply as a wrapper around some existing Sequence type.

1. Start:
   Empty Stack.

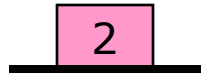
2. Push 2. →



3. Push 7. →



4. Pop. →



5. Pop.
   Empty again. →

6. Push 5. →

Review
Queues

A Queue is:

- A kind of container.
- A restricted version of a Sequence: First-In-First-Out (FIFO).

Conceptually, a Queue carries out the idea of **waiting in line**.

Three primary operations:

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

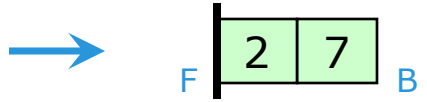
A Queue can be implemented:

- As a wrapper around some existing Sequence type.
- Using a **circular buffer**.

1. Start:
   Empty Queue.

2. Enqueue 2.

3. Enqueue 7.



4. Dequeue.

5. Dequeue.
   Empty again.

6. Enqueue 5.

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.

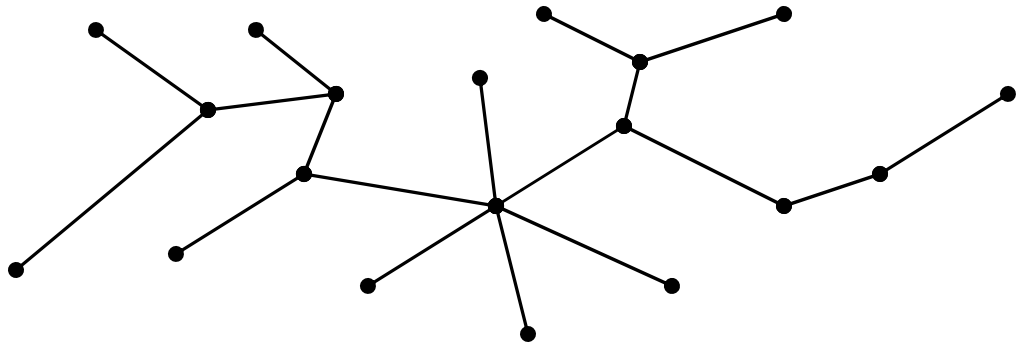
Unit Overview
The Basics of Trees

Major Topics

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

Introduction to Trees
What is a Tree?

By a **tree**, mathematicians mean a structure like this:



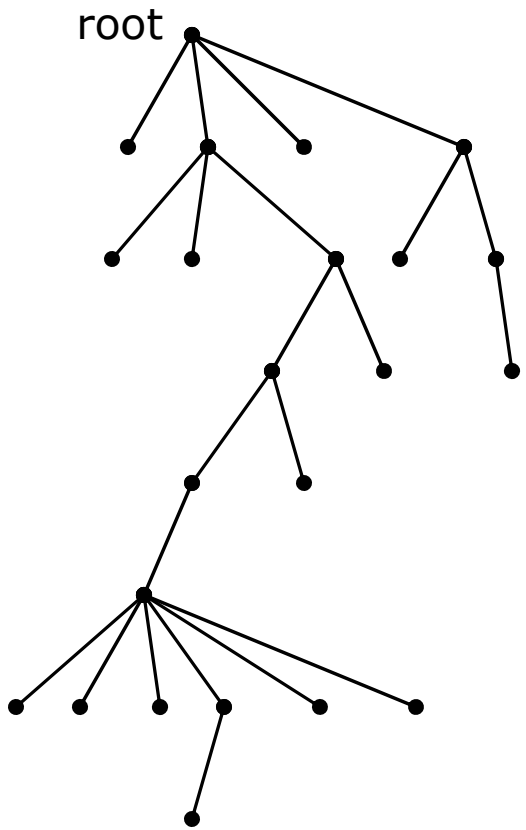
- Each dot is called a **vertex** (note the Latin plural "**vertices**") or a **node**.
  - I will use *vertex* for the element of the tree as a conceptual object, and *node* for the small data substructure representing it.
- Each line is called an **edge**.
- Each edge joins two vertices.
- A tree is connected (all one piece) and there are no cycles.

Rooted Trees — Introduction

Often we use trees to represent hierarchical structures.

We place one vertex at the top, and we call it the **root**. Each other vertex of the tree hangs from some vertex. The result is a **rooted tree**.

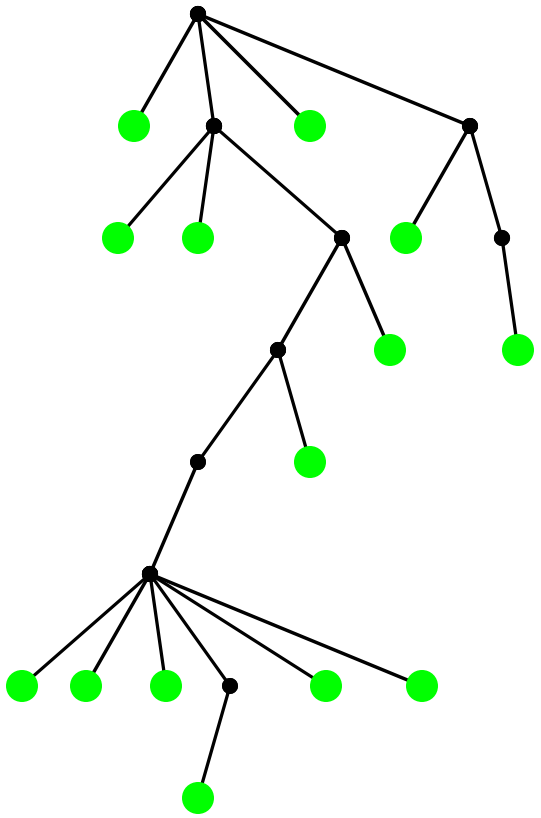
From now on in this class, "tree" means "rooted tree".



Introduction to Trees
Rooted Trees — Terminology [1/5]

Some of the terminology for rooted trees comes from plants.

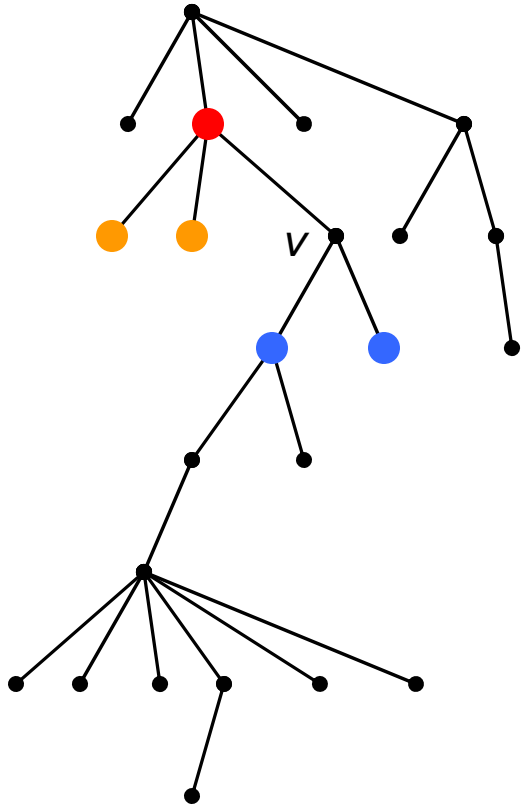
- "Root" is an obvious example.
- Another: A vertex with nothing hanging off of it is called a **leaf**.
  - Leaves are shown in green.
  - What if a tree has just one vertex?



Rooted Trees — Terminology [2/5]

Other terminology comes from family trees.

- To illustrate this, we label a vertex "*v*" in the tree at right.
- The vertex that *v* hangs from (shown in red) is *v*'s **parent**.
  - Every vertex except the root has exactly one parent.
- The vertices that hang from *v* (shown in blue) are *v*'s **children**.
  - A leaf has no children.
- The other children of *v*'s parent (shown in orange) are *v*'s **siblings**.



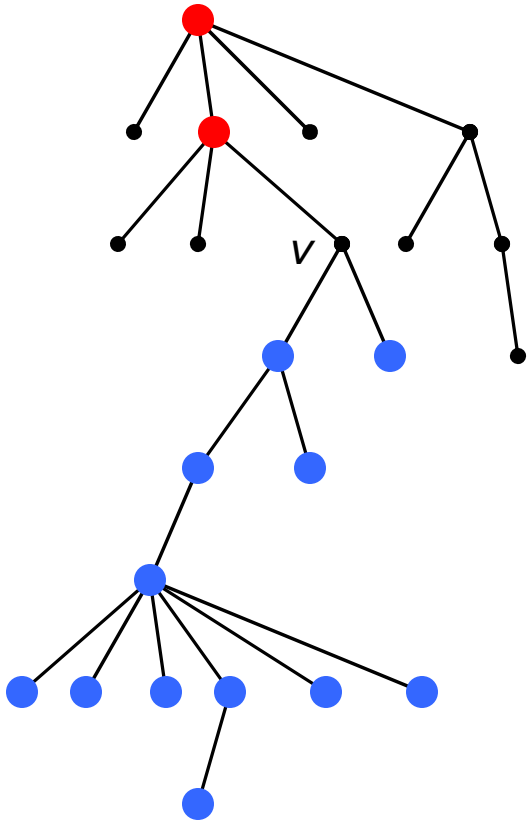
Rooted Trees — Terminology [3/5]

The parent of *v*, and its parent, and its parent, etc., are *v*'s **ancestors**.

- These are shown in red.

v's children, and their children, and their children, etc., are *v*'s **descendants**.

- These are shown in blue.



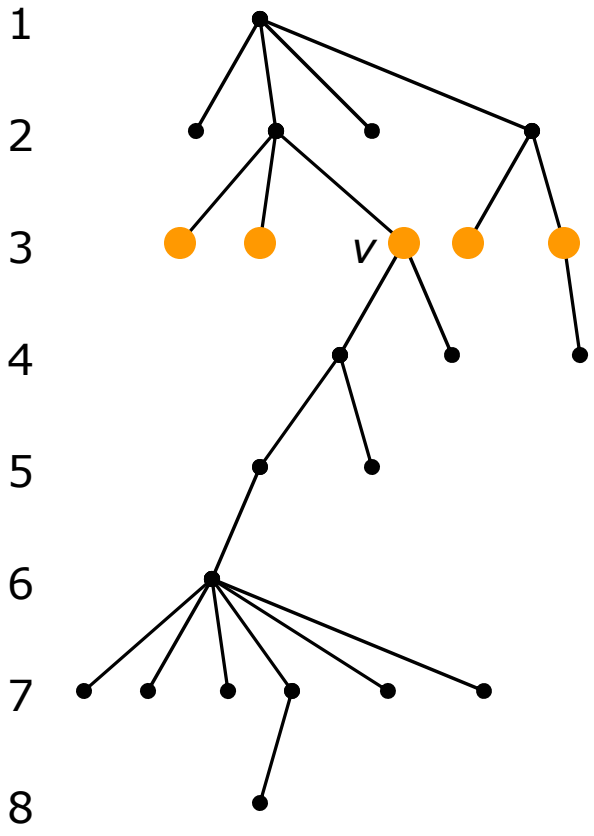
Rooted Trees — Terminology [4/5]

The vertices of a rooted tree come in **levels**.

- The root is at level 1.
- Each other vertex has a level 1 greater than its parent.
- Level 3, which includes *v*, is shown in orange.
- We often draw vertices at the same level in a horizontal row.

The **height** of a tree is the number of levels.

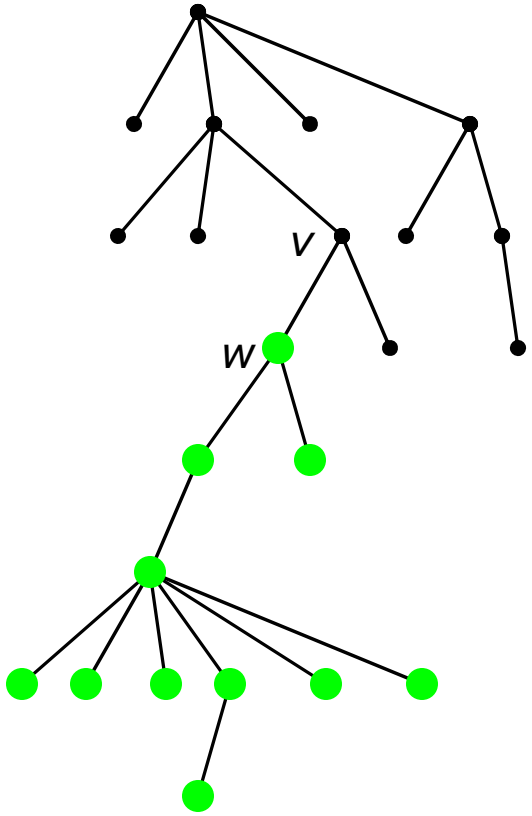
- This tree has height 8.
- *Note: Some people define "height" slightly differently.*



Rooted Trees — Terminology [5/5]

A **subtree** consists of a node and all its descendants.

- Given a node *n*, the **subtree rooted** *at n* consists of *n* and all its descendants.
- Given a node *n*, a **subtree of** *n* is a subtree rooted at some child of *n*.
- Shown in green is a subtree of *v*. It is the subtree rooted at *v*'s child *w*.



Rooted Trees — General Trees

A "general tree" is a somewhat more precise version of what we have been talking about.

- A **general tree** consists of a node (called the *root*) and zero or more subtrees of the root, each of which is a general tree.
- Note that a general tree must have at least one node.

The above is a **recursive definition**.

Binary Trees
Overview

Our next ADT is **Binary Tree**.

We will cover:

- What a Binary Tree is.
- Three special kinds.
- Traversals.
- Implementation.
- Applications.

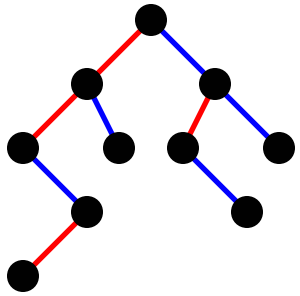
What is missing above?

- "Binary Trees in the C++ STL", because there aren't any.
  - Not in the *interface*, anyway. They are used internally.

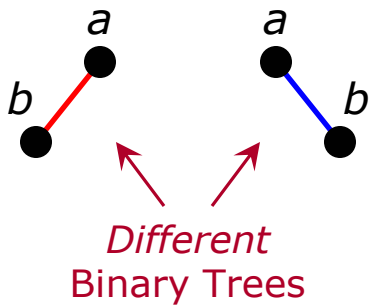
Binary Trees
What a Binary Tree Is — Idea

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.



Different Binary Trees

An **empty** Binary Tree is a Binary Tree with no nodes.

Empty Binary Tree

Binary Trees
What a Binary Tree Is — ADT

Data

- A set of nodes.

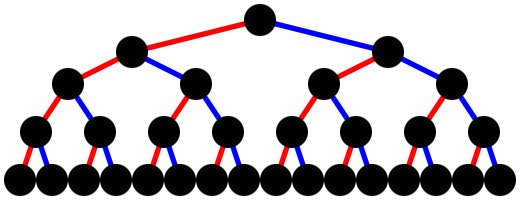
Operations

- **Create** (empty).
- **Create**, given a root and two subtrees.
- **Destroy**.
- **isEmpty**.
- **getRootData** & **setRootData**.
  - Access to data in root node.
- **attachLeft** & **attachRight**.
  - Attach a child to the root.
- **attachLeftSubtree** & **attachRightSubtree**.
  - Attach a subtree to the root.
- **detachLeftSubtree** & **detachRightSubtree**.
  - Detach a subtree from the root.
- **leftSubtree** & **rightSubtree**.
  - Returns a subtree.
- **preorderTraverse**, **inorderTraverse**, & **postorderTraverse**.
  - Visit all nodes in the appropriate order.

Binary Trees
Three Special Kinds

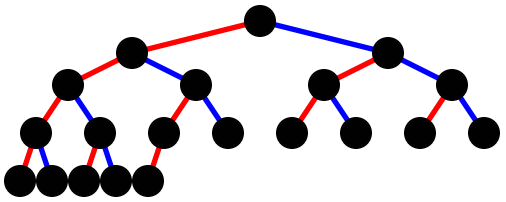
Full Binary Tree

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



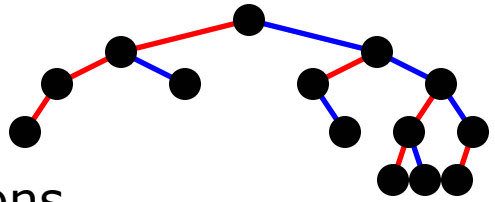
Complete Binary Tree

- All levels above bottom are completely full.
- Bottom level is filled left-to-right.
- Importance: As such trees grow, nodes must be added in a particular order. This gives them a useful array representation, which we look at later.



Balanced Binary Tree

- Left and right subtrees of each node have heights that differ by at most 1.
- Importance: Height is small, even if there are many nodes. This can allow for fast operations.



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

Full → Complete → Balanced

Binary Trees
Traversals — Idea

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:

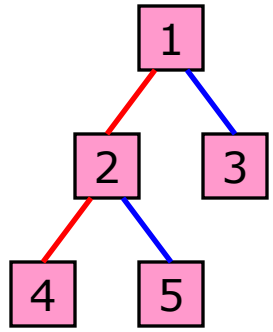
- Inorder traversal of left subtree.
- Root.
- Inorder traversal of right subtree.

Postorder traversal.

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

Binary Trees
Traversals — Example

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 — A Trick

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.

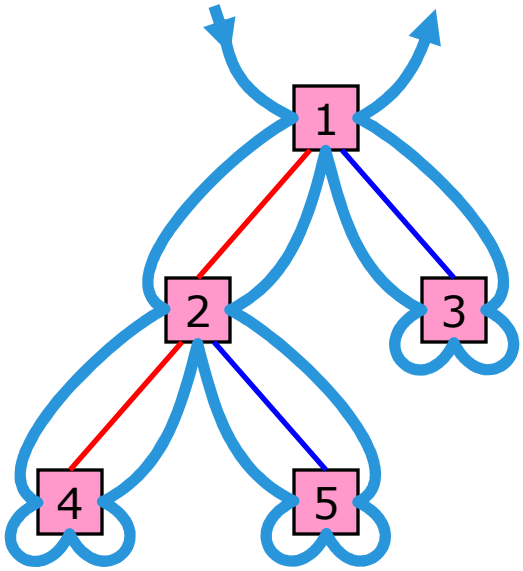
- 1 2 4 5 3

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.

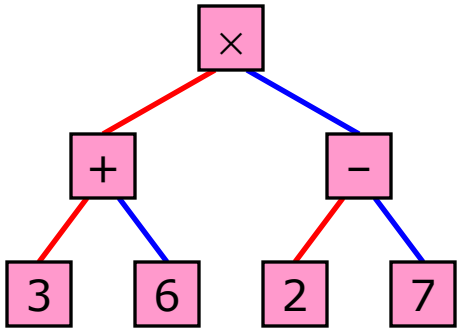
- 4 5 2 3 1



Binary Trees
Traversals — Expressions

Consider the Binary Tree at right.

- This is the **parse tree** of an expression.



Postorder traversal: 3 6 + 2 7 − ×

- This is Reverse Polish Notation for 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: before starting a (sub)tree, insert "(" if there is more than one node in the subtree. Similarly, insert ")" when done.
- Result: ((3 + 6) × (2 − 7)).

Preorder traversal: × + 3 6 − 2 7

- Add parentheses and commas: ×(+(3, 6), −(2, 7)).
- Thinking of "×", "+", and "−" as names of functions, we see that this is standard functional notation.
- This may be clearer: times(plus(3, 6), minus(2, 7)).

Binary Trees
Traversals — Algorithms

There are many reasons why we might traverse a Binary Tree: finding the sum of the data items, printing all data items, etc.

Can we write a single function that can be used to do all these things?

What should our traversal function do? Possibilities:

- It might provide an iterator that goes through the items in the proper order.
- It might return a list holding the data items in the proper order.
- It might be given a function, which it would call for each data item, in order.
  - "**Visitor pattern**".

How would we implement the last option above?

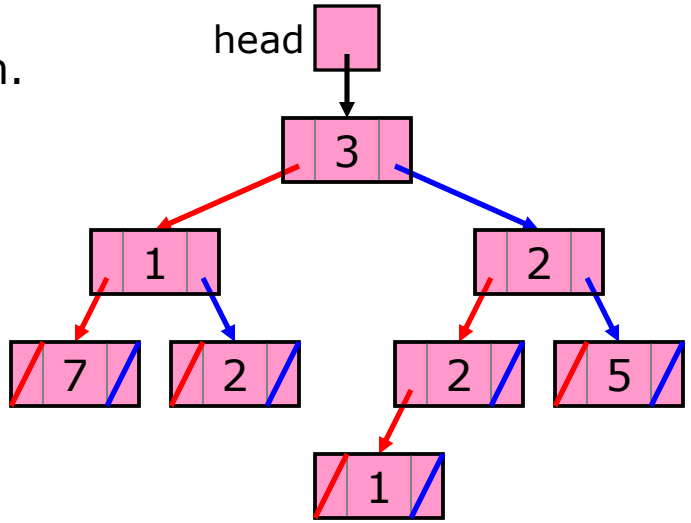
- We could write a recursive function. It would be given a "handle" (pointer?) to a node and a function to call for each item.
- Algorithm for a preorder traversal:
  - If the handle is null, return.
  - Call the function for the data in the given node.
  - Make a recursive call: left child, given function.
  - Make a recursive call: right child, given function.

For inorder, postorder, move this operation

Binary Trees
Implementation — #1: Pointer-Based [1/2]

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

- Very similar to our implementation of a Linked List.
- Each node has a data item and two child pointers: left & right.
- Each node owns its subtrees.
  - It is thus responsible for destroying them.
- A pointer is null if there is no child.



Each node *might* also have a pointer to its parent.

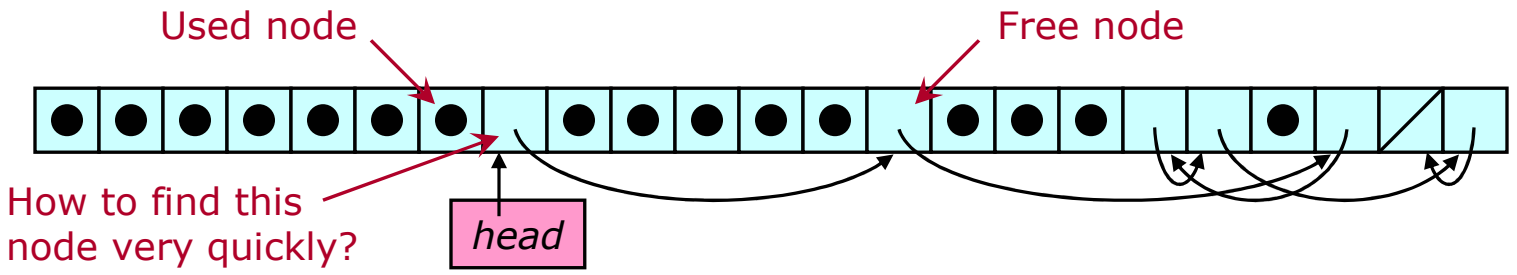
- This would allow some operations to be much quicker.
  - Such as finding the parent of a node.
  - Note that such operations are not included the text's ADT Binary Tree.
- Whether we do this, would depend on the purpose of the tree.

Implementation — #1: Pointer-Based [2/2]

Once again, we *could* put our nodes in an array.

- As before, the primary differences involve memory management: who does it and when it is done.

Q: If we do this, then how can we find a free node quickly?



A: To be able to get new free nodes quickly, we can make free nodes into a Linked List.

Constant time

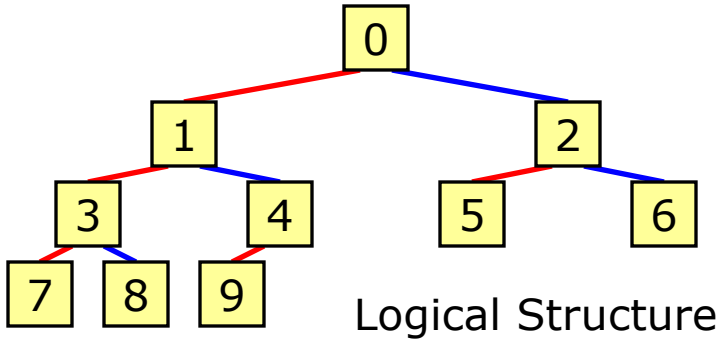
Notes

- This is *easy*. Nodes already have pointers in them (right?). And all we need to do is insert/remove at the beginning of the Linked List.
- This is a common technique, used on all kinds of node-based structures, including Linked Lists.

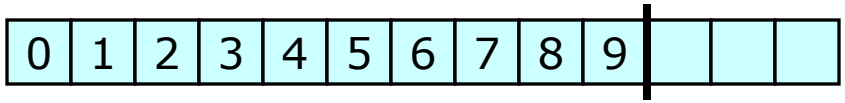
Binary Trees
Implementation — #2: Array-Based Complete [1/2]

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 need to store *only* an **array** of data items and a record of the number of nodes (***size***).
  - No pointers/indices are required!
- This greatly limits the operations available to us, since we must preserve the property of being complete.



Logical Structure

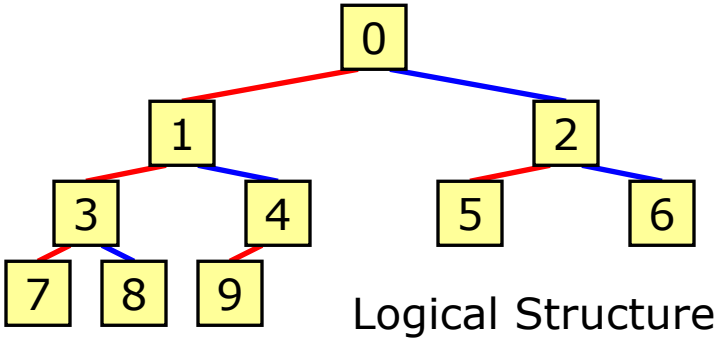


Physical Structure

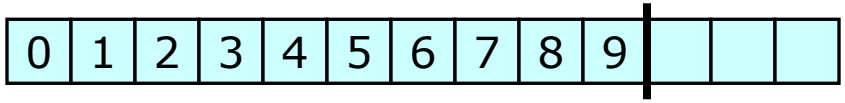
This array-based complete Binary Tree is commonly used to implement a data structure called a "Binary Heap".

- We will discuss this later in the semester.

Binary Trees
Implementation — #2: Array-Based Complete [2/2]



Logical Structure



Physical Structure

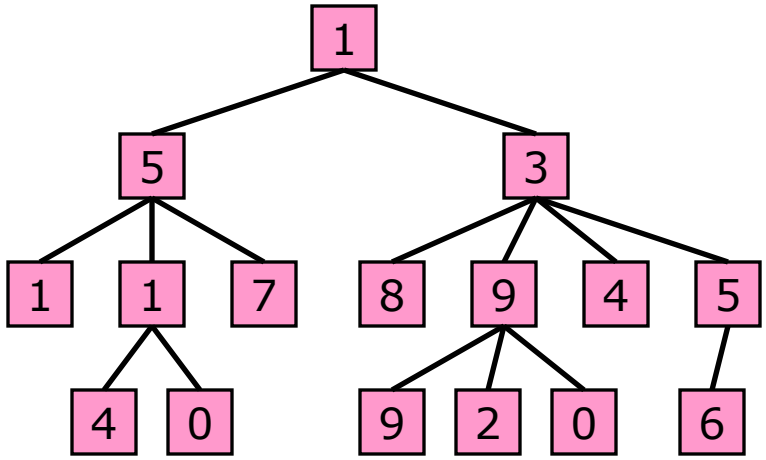
Without pointers, how do we move from one node to another?

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

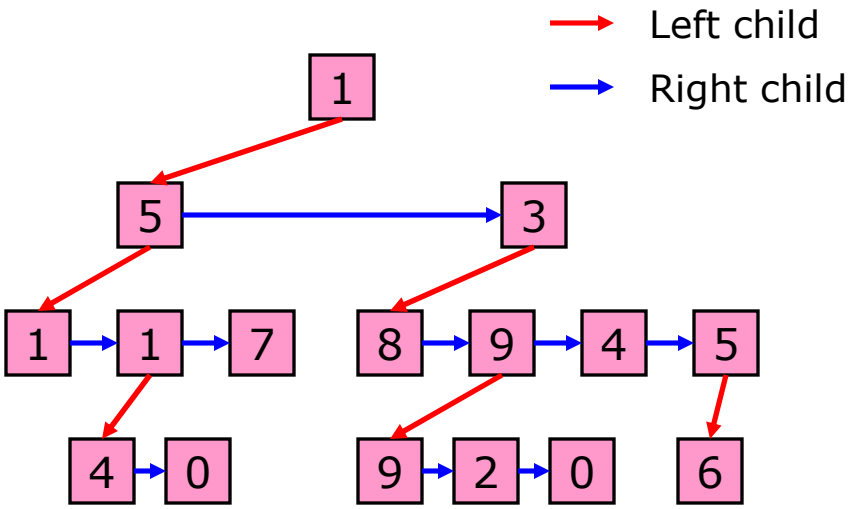
Binary Trees
Applications — Representing General Trees [1/2]

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

- A node's left child is its first child in the general tree.
- A node's right child is its next sibling in the general tree.



General Tree



Binary-Tree Representation

Applications — Representing General Trees [2/2]

What is **good** about this representation method?

- Nodes are small and simple. We do not need to give each node a way of holding an arbitrary number of pointers.
- It saves work & space in implementing general trees.

What is **bad** about this representation method?

- Finding a given child is linear time — that is, $O(n)$, where n is the *number of children*. However, situations in which we want random access to an arbitrarily large number of children are rare (in my experience).
- It may make operations that change the tree more difficult.

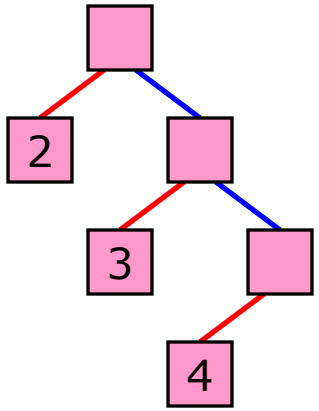
We would probably *not* use this for a tree in which nodes are restricted to having at most a **small fixed number** (more than 2) of children: 2-3 Trees, 2-3-4 Trees, Quadtrees, Octrees, etc.

- Some of these will be covered later in the semester.

Binary Trees
Applications — Lists of Lists [1/3]

Consider a Binary Tree with data items only in its leaves. We can use such a tree to implement a Linked List.

- A node represents either a list or a data item.
- The left child of a list node is a data-item node holding the first item in the list.
- If there are items remaining in the list, then the right child of a list node is a list node representing the remainder of the list. Otherwise, there is no right child.
- A data-item node is a leaf.



The tree at the right represents the list (2 3 4).

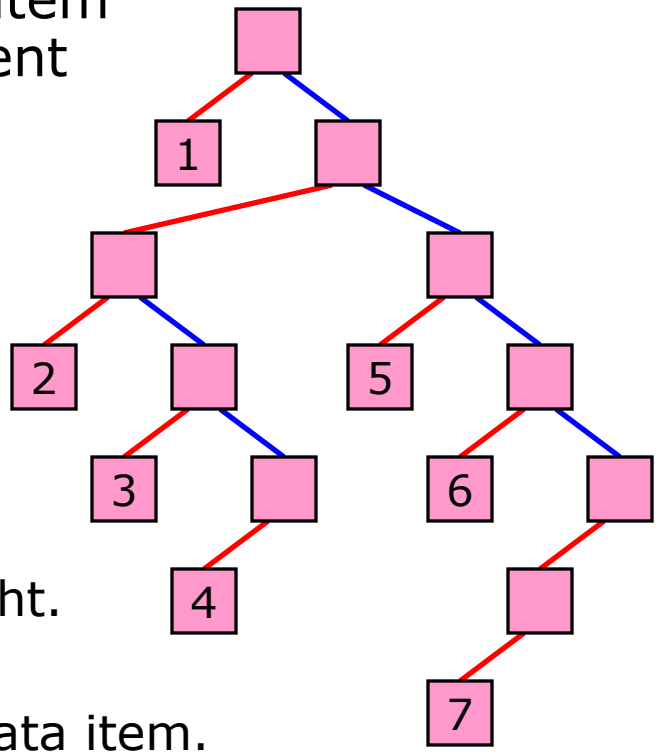
Since a node can represent either an **item** or a **list**, we can do more with this representation …

Binary Trees
Applications — Lists of Lists [2/3]

Since, again, a node represents either an item or a list, we can use this idea to represent lists whose members may be lists.

For example: (1 (2 3 4) 5 6 (7)).

- This is a list with 5 items:
  - 1
  - the list (2 3 4)
  - 5
  - 6
  - the list (7)
- This list is represented by the tree at right.



In general:

- Each node represents either a list or a data item.
- The left child of a list node is the first item in the list.
- The right child of a list node is the list of all the other items, if any.

"Recursive lists" are the data structures used in the language LISP. They are also the idea behind XML documents.

Applications — Lists of Lists [3/3]

Where do we commonly see lists of lists?

- File Systems
  - A directory is a list of files, some of which may be directories.
  - A file systems could store its directory structure using a Binary Tree.
- Programs
  - A block ("**{** … **}**") is a list of statements, some of which may be blocks.
  - Thus, binary trees maybe used during compilation.
- Outlines of Documents
  - The outline as a whole is a list of points to be made. Each point may have a list of sub-points.

Binary Trees
Applications — Recursively Partitioned Data

Sometimes we have data that is partitioned into two pieces, each of which is further partitioned, etc.

- Such data is naturally represented using a Binary Tree.

Example

- In computer graphics, we sometimes use a "Binary Space Partition" tree (BSP tree).
- Each node represents a region of space.
- We slice a region with a plane. Child nodes represent the regions on either side of this plane.
- BSP trees allow efficient computation of a front-to-back ordering of the objects in a static scene with a moving camera. This is useful for hidden surface removal, translucency, and other effects.