

# Binary Search Trees

## Treesort

---

CS 311 Data Structures and Algorithms  
Lecture Slides  
Wednesday, April 10, 2013

Chris Hartman  
Department of Computer Science  
University of Alaska Fairbanks  
cmhartman@alaska.edu  
Based on material by Glenn G. Chappell  
© 2005–2009 Glenn G. Chappell

## 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

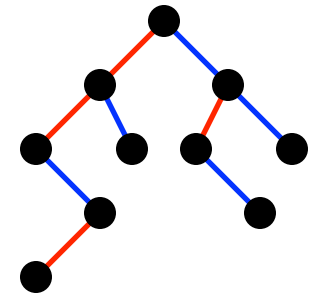
- ✓ ■ Introduction to Trees
- ✓ ■ Binary Trees
  - Binary Search Trees
  - Treesort

# Review: Binary Trees

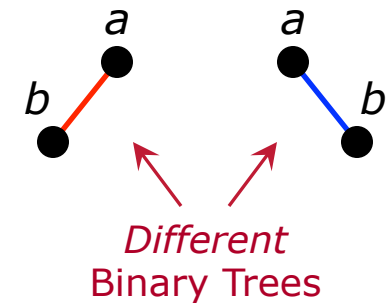
## What a Binary Tree Is

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.



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



# Review: 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.

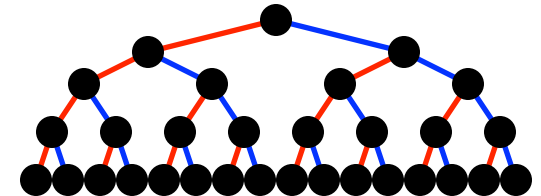
## Review: Binary Trees

### Three Special Kinds

---

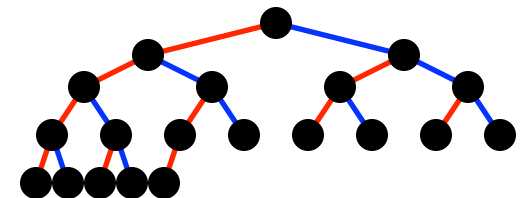
#### **Full** Binary Tree

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



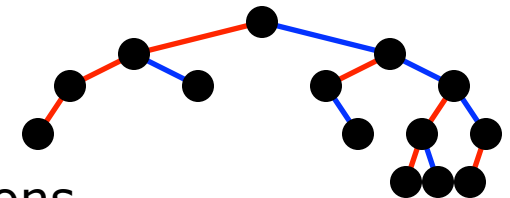
#### **Complete** Binary Tree

- 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

## Review: 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.

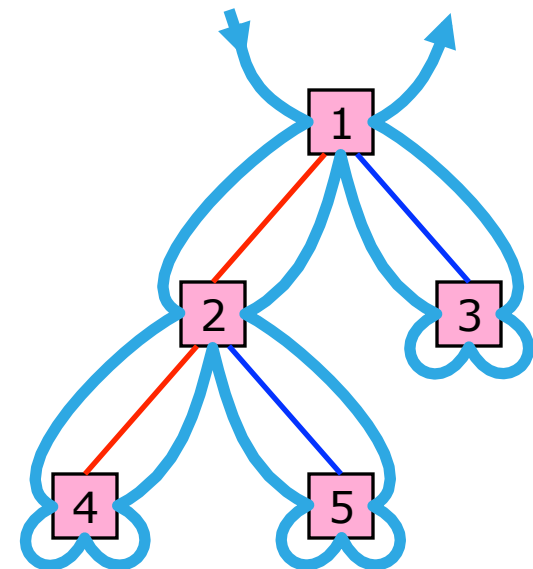
- 1 2 4 5 3

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



## Review: Binary Trees

### Traversals — Expressions

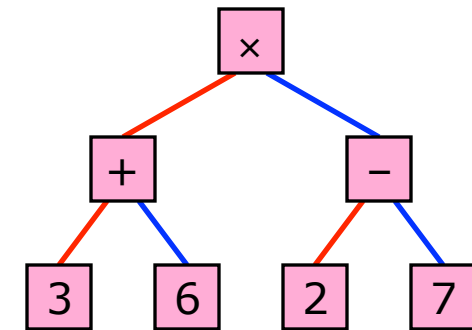
---

Consider the Binary Tree at right.

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

Postorder traversal:  $3\ 6\ +\ 2\ 7\ -\ \times$

- This is Reverse Polish Notation for the expression.



Inorder traversal:  $3\ +\ 6\ \times\ 2\ -\ 7$

- 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:  $((3 + 6) \times (2 - 7))$ .

Preorder traversal:  $\times\ +\ 3\ 6\ -\ 2\ 7$

- Add parentheses and commas:  $\times(+ (3, 6), -(2, 7))$ .
- Thinking of “ $\times$ ”, “ $+$ ”, and “ $-$ ” as names of functions, we see that this is standard functional notation.
- This may be clearer:  $\text{times}(\text{plus}(3, 6), \text{minus}(2, 7))$ .



## 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**”.
  - A restricted version of this might just put each item into an output iterator, like “write” for linked list.

**How** would we implement the last option above?

- We could write a recursive function. It would be given a “handle” (pointer? iterator?) 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]

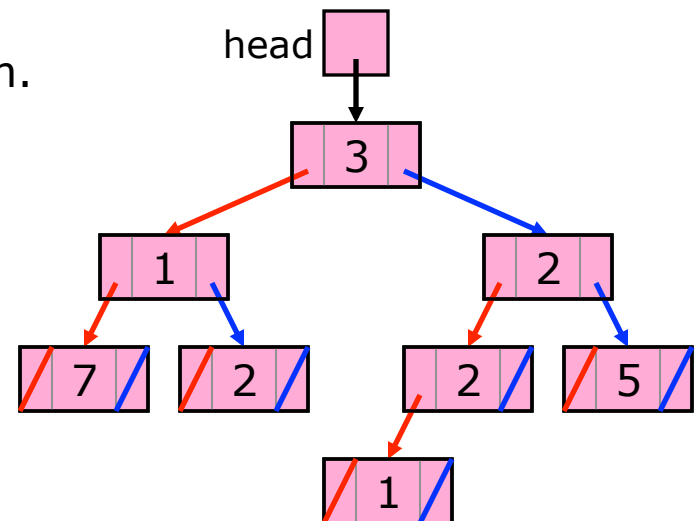
---

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.
- Whether we do this, would depend on the purpose of the tree.



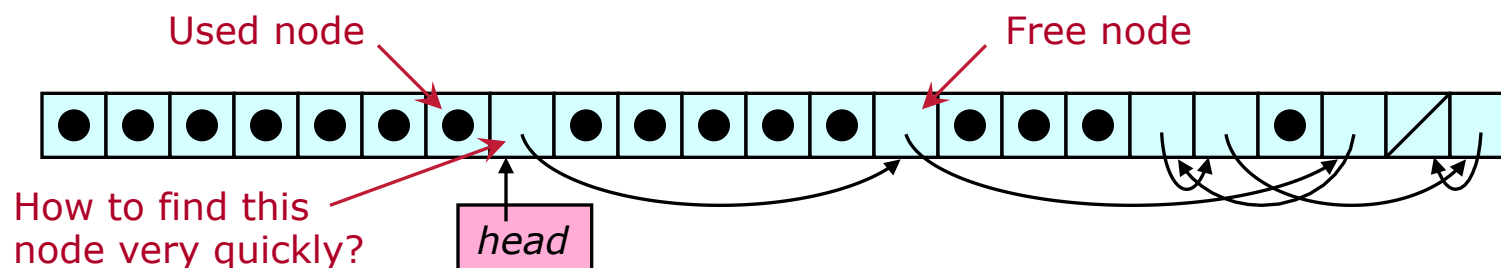
## Binary Trees

### 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.

Notes

Constant time

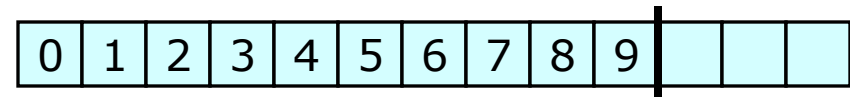
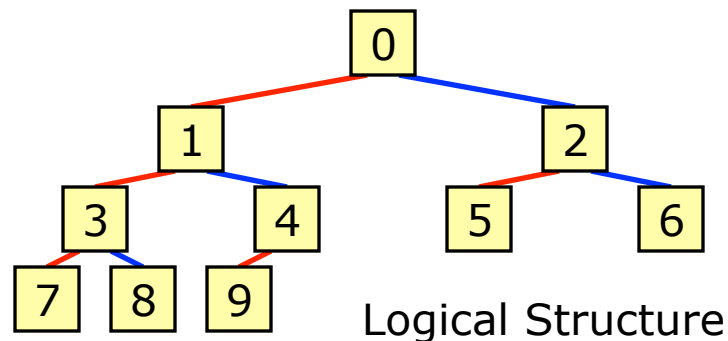
- This is *easy*. Nodes already have pointers in them (right?). And all we need to do is insert/remove at the beginning of the Linked List.
- This is a common technique, used on all kinds of node-based structures, including Linked Lists.

## 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.



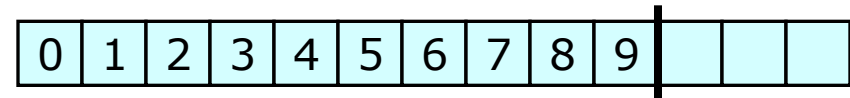
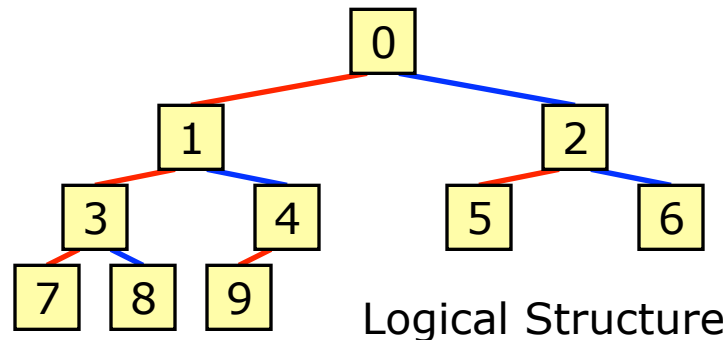
Physical Structure

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]



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

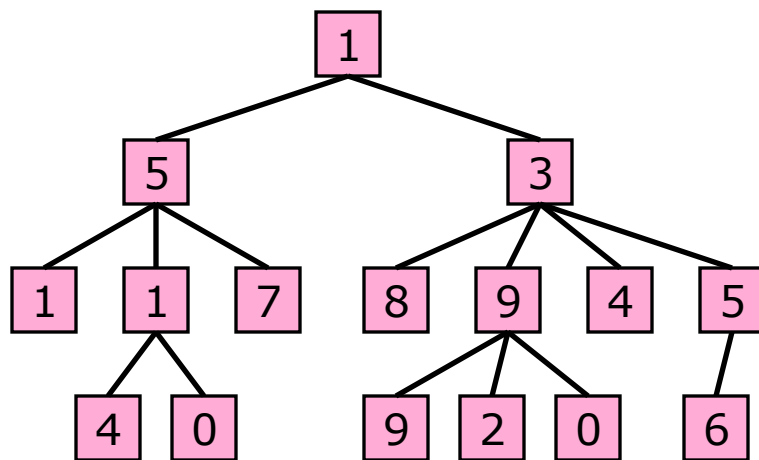
- The **root**, if any, is at index 0.
  - The root exists if  $0 < \text{size}$ , that is, if the tree is nonempty.
- The **left child** of node  $k$  is at index  $2k + 1$ .
  - The child exists if  $2k + 1 < \text{size}$ .
- The **right child** of node  $k$  is at index  $2k + 2$ .
  - The child exists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

# Binary Trees

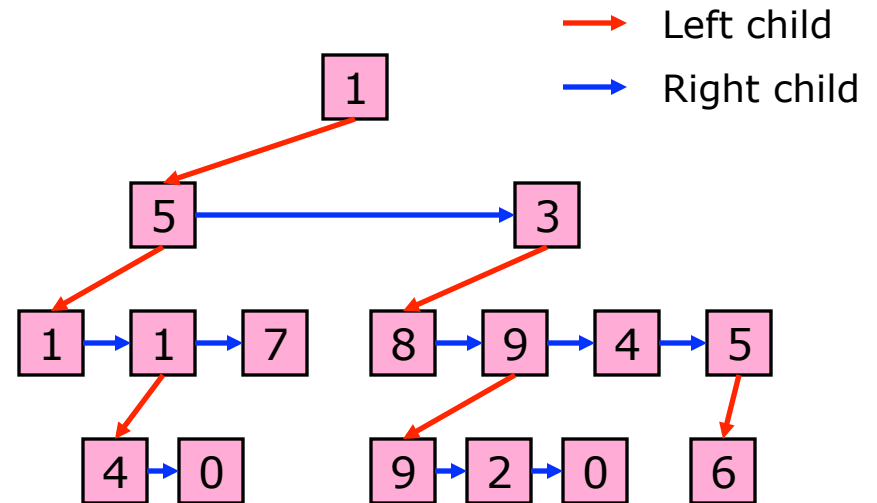
## Applications — Representing General Trees [1/2]

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

## Binary Trees

### 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

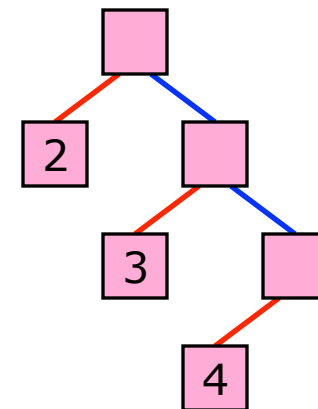
### Applications — Lists of Lists [1/3]

---

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

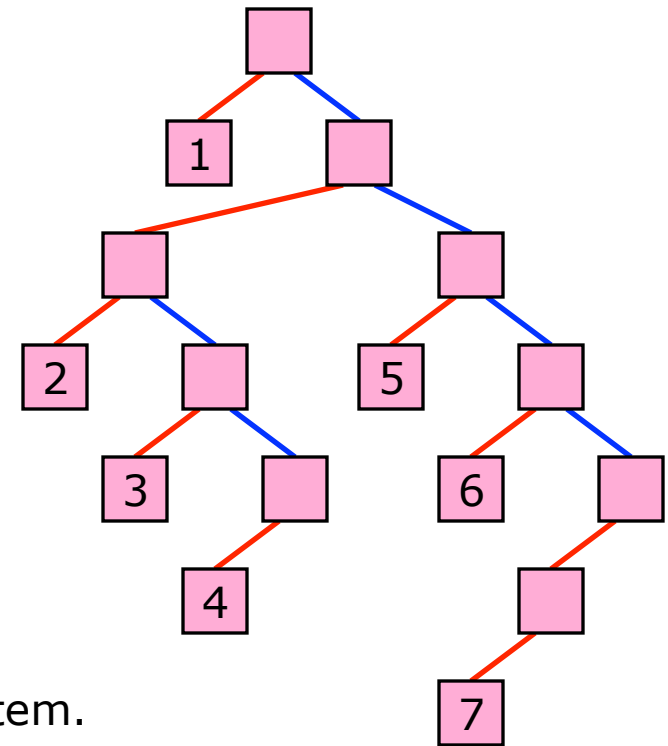
### Applications — Lists of Lists [2/3]

---

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 languages like LISP.  
They are also the idea behind XML documents.

## Binary Trees

### 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.
- This is related to the “Composite” design pattern.

## 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.

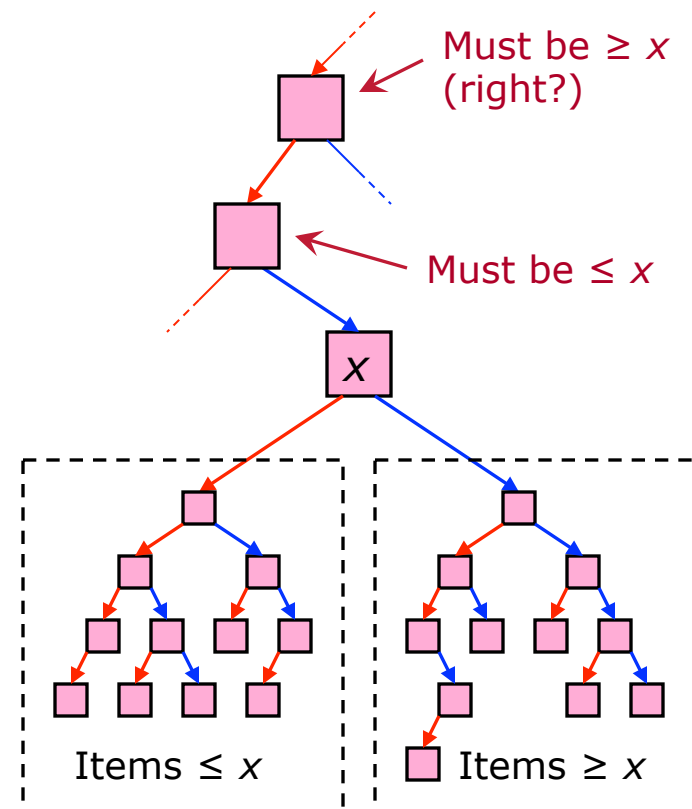
# Binary Search Trees

## What a Binary Search Tree Is

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  $\leq$  the node's item.
  - All data items in a node's right subtree are  $\geq$  the node's item.
  - Note: Sometimes we replace “ $\leq$ ” and “ $\geq$ ” by “ $<$ ” and “ $>$ ”, so that items having equivalent values are not allowed.
- Thus, an inorder traversal lists items in sorted order.



# Binary Search Trees

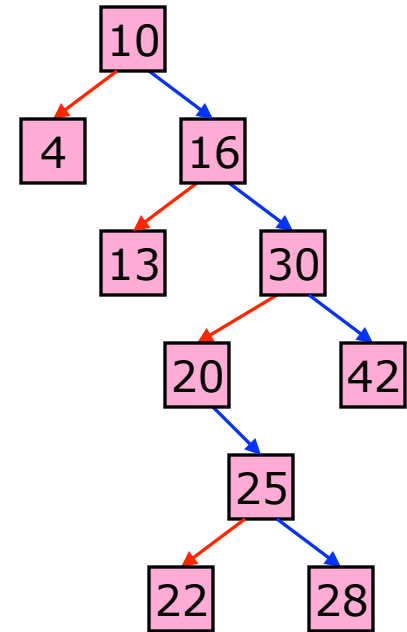
## ADT [1/3]

---

### 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

---

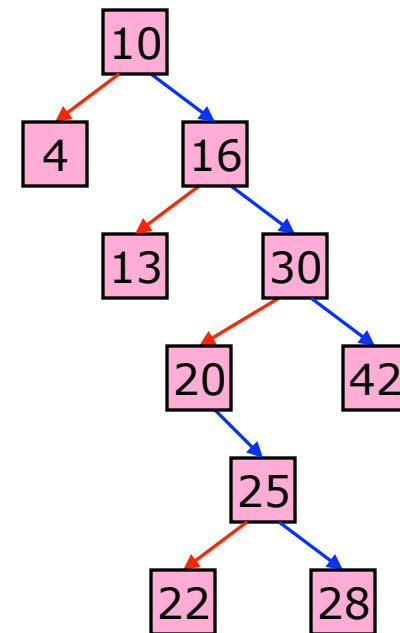
To **retrieve** a value from a Binary Search Tree, we begin at the root and repeatedly follow left or right child pointers, depending on how the search key compares to the key in each node.

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

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

- $18 > 10 \rightarrow$  right.
- $18 > 16 \rightarrow$  right.
- $18 < 30 \rightarrow$  left.
- $18 < 20 \rightarrow$  left.
- No left child  $\rightarrow$  NOT FOUND.



## Binary Search Trees

### Operations — Insert

---

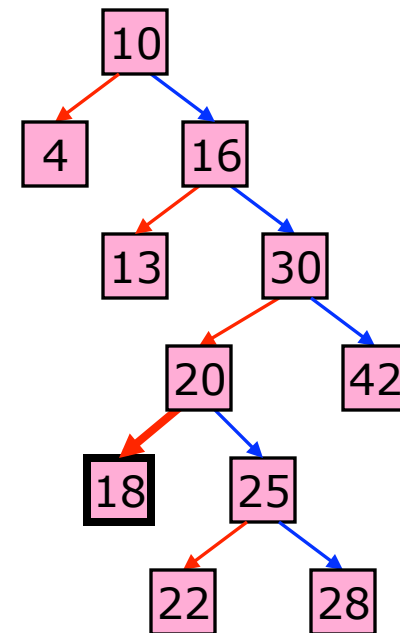
To **insert** a value with a given key, we find where the key “should” go (**retrieve** operation), and then we put the value there.

- 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.



## Binary Search Trees

### Operations — Delete [1/3]

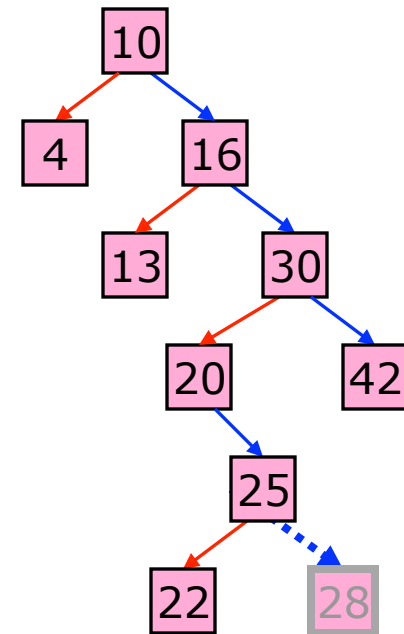
---

To **delete** an item from a Binary Search Tree, given its key:

- Find the node (**retrieve** operation).
- Then, three cases:
  - The node to be deleted has no children (it is a leaf).
  - The node has 1 child.
  - The node has 2 children.

**No-child** (leaf) case:

- Delete the node, using the appropriate **Binary Tree** operation.



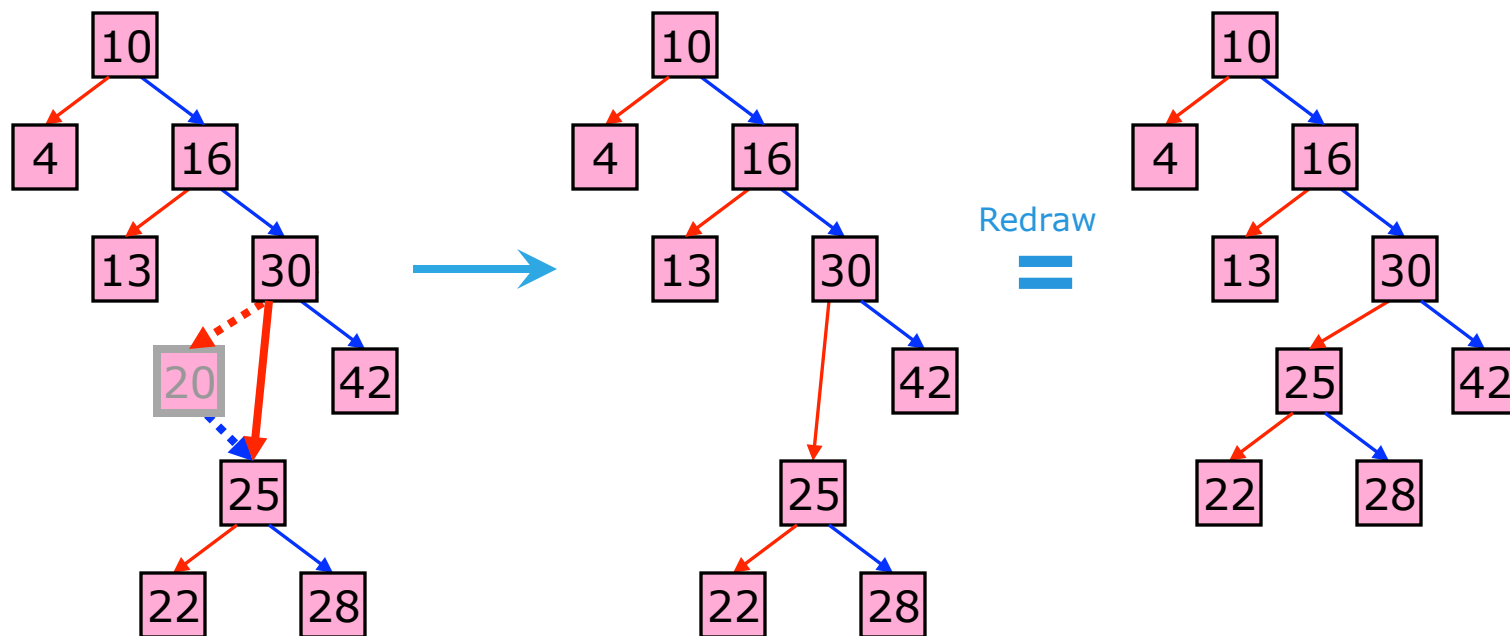
# Binary Search Trees

## Operations — Delete [2/3]

---

### One-child case:

- Replace the subtree rooted at the node with the subtree rooted at its child.
  - This is a constant-time operation, once the node is found.



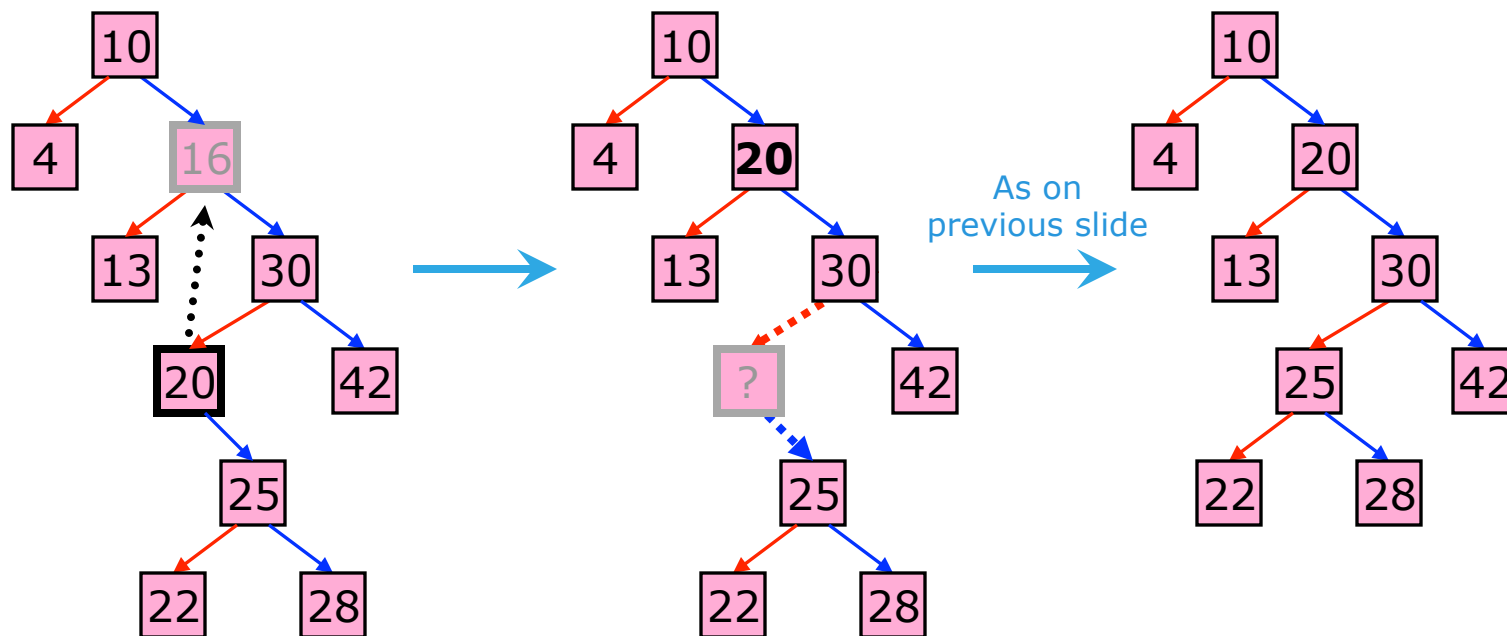
# Binary Search Trees

## Operations — Delete [3/3]

---

### Two-child case:

- Replace the data in the node with the data in its **inorder successor** (copy or swap).
- Delete the inorder successor.
  - This has at most one child.



## 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

---

	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

Binary Search Trees have poor worst-case performance.  
But they have very good performance:

- On average.
- If balanced.
  - But we do not know an efficient way to make them *stay* balanced.

Can we efficiently keep a Binary Search Tree balanced?

- We will look at this question again later.