Overview

- Definitions, Terminology, and Properties
- Binary Trees
- Search Trees: Improving Search Speed
- Traversing Binary Search Trees

Data Structures/ADTs as Tools in Your Tool Chest!

- The Swiss army knife is not a good model for a universal tool, handy as it is
- Bill Buxton, Principal HCI researcher at Microsoft Research: “Everything is best for something and worst for something else”
- Choosing the right tool for the job: analysis of the problem and its requirements may lead you to choose the right one, based on tradeoffs, e.g., relative frequencies of search vs. insert/delete for Arrays vs. ArrayLists vs. LinkedLists
  - but these are not the only operations we’re concerned with in managing collections – new needs require new forms

Searching in a Linked List (1/2)

- Searching for element in LinkedList involves pointer chasing and checking consecutive Nodes to find it (or not)
  - it is sequential access
  - O(N) - can stop sooner for element not found if list is sorted
- Getting Nth element in an Array or ArrayList is random access (which means O(1)), but searching for particular element, even with index, remains sequential O(N)
- Even though NodeLists support indexing (dictated by Java’s List interface), getting the Ith element is also done (under the hood) by pointer chasing and hence is O(N)
Searching in a Linked List (2/2)

- For N elements, search time is O(N)
  - unsorted: sequentially check every node in list until element ("search key") being searched for is found, or end of list is reached
    - if in list, for a uniform distribution of keys, average search time for a random element is N/2
    - if not in list, it is N
  - sorted: average search time is N/2 if found, N for not found (the win!)
  - we ignore issue of duplicates
- No efficient way to access Nth node in list
- Insert and remove similarly have average search time of N/2 to find the right place

*Actually more complicated than this – depends on distribution of keys

---

**Searching, Inserting, Removing**

<table>
<thead>
<tr>
<th></th>
<th>Search if unsorted</th>
<th>Search if sorted</th>
<th>Insert/remove after search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Array</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

---

**Binary Search (1/4)**

- Searching sorted linked list is sequential access
- We can do better with a *sorted array* that allows random access at any index to obviate sequential search
- Remember merge sort with search O(log N) where we did "bisection" on the array at each pass
- If we had a sorted array, we could do the same thing (like "20 questions")
  - start in the middle
  - keep bisecting array, deciding which portion of the sub-array the search key lies in, until we find that key or can't subdivide further (not in array)
  - For N elements, search time is O(log N) (since we reduce number of elements to search by half each time), very efficient!
- Website: [http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html](http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html)
Binary Search (2/4)

- $\log_2 N$ grows much more slowly than $N$, especially for large $N$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>27</td>
</tr>
</tbody>
</table>

*relatively small $n$ in this graph, but imagine how large the difference is as $n$ increases*

Binary Search (3/4)

- A sorted array can be searched quickly using bisection because arrays are indexed

- Java `ArrayList`s are indexed too, so they share this advantage! But inserting and removing from `ArrayList`s is slow!

- Inserting into or deleting from middle of `ArrayList` causes all successor elements to be shifted over to make room. For arrays, you manage this yourself. Both have same worst-case run time $O(N)$

- Advantage of linked lists is insert/remove by manipulating pointer chain is faster $O(1)$ than shifting elements $O(N)$, but search can’t be done with bisection 😞, a real downside if search is done frequently

Binary Search (4/4)

- Is there a data structure or Abstract Data Type that provides both search speed of sorted arrays and `ArrayList`s and insertion/deletion efficiency of linked lists?

- Yes, indeed! `Trees`! They provide faster searching than linked lists and faster insertions than arrays!
Trees vs Linked Lists (1/2)

- Singly linked list – collection of nodes where each node references only one neighbor, the node’s successor:

Trees vs Linked Lists (2/2)

- Tree – also collection of nodes, but each node may reference multiple successors/children
- Trees can be used to model a hierarchical organization of data

Technical Definition of a Tree

- Finite set, $T$, of one or more nodes such that:
  - $T$ has one designated root node
  - remaining nodes partitioned into disjoint sets: $T_1, T_2, \ldots, T_n$
  - each $T_i$ is also a self-contained tree, called subtree of $T$
- Look at the image on the right - where have we seen such hierarchies like this before?
### Inheritance Hierarchies as Trees

- Higher in inheritance hierarchy, more generic
  - Animal is most generic
- Lower in inheritance hierarchy, more specific
  - Dog is more specific than Mammal, which in turn is more specific than Animal

### Graphical Containment Hierarchies as Trees

- Levels of containment of GUI components
- Higher levels contain more components
- Lower levels contained by all above them
  - Panes contained by root pane of PaneOrganizer, which is contained by Scene

### Tree Structure

- Note that the tree structure has meaning
  - any subtree of $T_i$ is also a tree with specific values
- Can be useful to only examine specific subtrees of $T$
**Tree Terminology**

- **A** is the root node
- **B** is the parent of **D** and **E**
- **D** and **E** are children of **B**
- **(C ---- F)** is an edge
- **D**, **E**, **F**, **G**, and **I** are external nodes or leaves
  (i.e., nodes with no children)
- **A**, **B**, **C**, and **H** are internal nodes
- depth (level) of **E** is 2 (number of edges to root)
- height of the tree is 3 (max number of edges in path from root)
- degree of node **B** is 2 (number of children)

**Binary Trees**

- Each internal node has a maximum of 2 successors, called children
  - i.e., each internal node has degree 2 at most
- Recursive definition of binary tree: A binary tree is either an:
  - external node (leaf), or
  - internal node (root) with one or two binary trees
    as children (left subtree, right subtree)
  - empty tree (represented by a null pointer)

  Note: These nodes are similar to the linked list nodes, with one data and two child pointers – we show the data element inside the circle

**Top Hat Question**

Which of the following is true about internal nodes and external nodes for binary trees?

A. Internal nodes are nodes with no children
B. External nodes cannot be leaves of a tree
C. The root node is an external node
D. Internal nodes must have one or two children, while external nodes have none
Properties of Binary Trees (1/2)

- A binary tree is full when each node has exactly zero or two children.
- Binary tree is perfect when, for every level $i$, there are $2^i$ nodes (i.e., each level contains a complete set of nodes).
  - thus, adding anything to the tree would increase its height.

Properties of Binary Trees (2/2)

- In a full Binary Tree: ($\#$ leaf nodes) = ($\#$ internal nodes) + 1
- In a perfect Binary Tree: ($\#$ nodes at level $i$) = $2^i$
- In a perfect Binary Tree: ($\#$ leaf nodes) $\leq 2^{(\text{height})}$
- In a perfect Binary Tree: (height) $\geq \log_2(\# \text{ nodes}) - 1$

Binary Search Tree a.k.a BST (1/2)

- Binary search tree stores keys in its nodes such that, for every node, keys in left subtree are smaller, and keys in right subtree are larger.
  - Note: the keys here are sorted alphabetically!
BST (2/2)

- Below is also BST but much less balanced. Gee, it looks like a linked list!
- The shape of the trees is determined by the order in which elements are inserted.

```
        1
       / \
      2   3
     /   /
    4    5
   /   / \
  6   7   8
```

```
        1
       / \
      2   3
     /   /\n    4   5   6
   / /   / \
  7 8   9   10
```

BST Class (1/3)

- What do BSTs know how to do?
  - much the same as sorted linked lists: insert, remove, size
  - BSTs also have a special search method, since searching is more complicated than simply iterating through its nodes

- Let's see what an implementation of a BST class would look like...
  - In addition to data, left, and right child pointers, we'll add a parent pointer for ease of implementation (the remove method)
  - you'll learn more about implementing data structures in CS16!

Nodes, data, and keys

- _data is a composite that can contain many properties, one of which is a key that Nodes are sorted by (here, ISBN #)
BST Class (2/3)

```java
public class BinarySearchTree<Type extends Comparable<Type>> {
    Node<Type> _root;
    public BinarySearchTree(Type data) {
        _root = new Node(data, null);
    }
    public void insert(Type newData) {
        // ... } // end of class
```

BST Class (3/3)

- Our implementations of Linked Lists, Stacks, and Queues are "smart" data structures that chain "dumb" nodes together
  - the lists did all the work by maintaining previous and current pointers and did the operations to search for, insert, and remove information – thus, nodes were essentially data containers
- Now we will use a "dumb" tree with "smart" nodes that will delegate using recursion
  - tree will delegate action (such as searching, inserting, etc.) to its root, which will then delegate to its appropriate child, and so on
  - creates specialized Node class that stores its data, parent, and children, and can do operations such as insert and remove

BST: Node Class (1/3)

- "Smart" Node includes the following methods:
  /// pass in entire data item, containing key, so compareTo() will work
  public Node<Type> search(Type dataToFind);
  public Node<Type> insert(Type newData);
  /// remove deletes Node pointing to dataToRemove, which contains key; removing Node also will remove the matched data instance unless there's another reference to it
  public Node<Type> remove(Type dataToRemove);
  public Node<Type> swapData(); // helper to swap the data in two different nodes
- Plus setters and getters of instance variables, defined in the next slides ...
**BST: Node Class (2/3)**

- **Nodes** have a maximum of two non-null children that hold data implementing `Comparable<Type>`
  - Four instance variables: `_data`, `_parent`, `_left`, and `_right`, with each having a get and set method.
  - `_data` represents the data that Node stores. It also contains the key attribute that Nodes are sorted by—we’ll use a Tree that stores Books.
  - `_parent` represents the direct parent (another Node) of Node—only used in `remove` method.
  - `_left` represents Node’s left child and contains a subtree, all of whose data is less than Node’s data.
  - `_right` represents Node’s right child and contains a subtree, all of whose data is greater than Node’s data.
  - Arbitrarily select which child should contain data equal to Node’s data.

```java
public class Node<Type extends Comparable<Type>> {
    private Type _data;
    private Type _parent;
    private Node<Type> _left;
    private Node<Type> _right;

    public Node(Type data, Node<Type> parent) {
        // Construct a leaf node
        _data = data;
        _parent = parent;
        _left = null;
        _right = null;
    }
}
```

**BST: Node Class (3/3)**

```java
public class Node<Type extends Comparable<Type>> {
    private Type _data;
    private Type _parent;
    private Node<Type> _left;
    private Node<Type> _right;

    public Node(Type data, Node<Type> parent) {
        // Construct a leaf node
        _data = data;
        _parent = parent;
        _left = null;
        _right = null;
    }
}
```

**Smart Node Approach**

- **BinarySearchTree** is “dumb,” so it delegates to root, which in turn will delegate recursively to its left or right child, as appropriate.

  ```java
  // Search method for entire BinarySearchTree:
  public Node<T> search(T data) {
      return _root.search(data);
  }
  ```

- Smart node approach makes our code clean, simple and elegant.
  - Non-recursive method is much messier, involving explicit bookkeeping of which node in the tree we are currently processing.
  - We used the non-recursive method for sorted linked lists, but trees are more complicated, and recursion is easier—a tree is composed of subtrees!
Searching Simulation (animated)

● What if we want to know if 224 is in Tree?
● Tree says:

123 says:
"Let's see, I'm not 224. But if 224 is in tree, since it's larger, it would be to my right. I'll ask my right child and return its answer."

123
16
224

Searching Simulation (animated)

● What if we want to know if 224 is in Tree?
● 252 says:

"I'm not 224. I better ask my left child and return its answer."

123
16
224
252

Searching Simulation (animated)

● What if we want to know if 224 is in Tree?
● 224 says:

"224? That's me! Hey, caller (252) here's your answer."

123
16
252
224

Answer: 224 is in the Tree!
Searching Simulation (animated)

- What if we want to know if 224 is in Tree?

Answer: 224 is in the Tree!

123

Hey, caller (123)! Here's your answer.

Answer: 224 is in the Tree!

Searching Simulation (animated)

- What if we want to know if 224 is in Tree?

Answer: 224 is in the Tree!

123

Hey, Tree! Here's your answer

Answer: 224 is in the Tree!

Searching Simulation - Recap

- What if we want to know if 224 is in Tree?
- Tree says "Hey Root! Ya got 224?"
- 123 says: "Let's see. I'm not 224. But if 224 is in tree, it would be to my right. I'll ask my right child and return its answer."
- 252 says: "I'm not 224, it's smaller than me. I better ask my left child and return its answer."
- 224 says: "224? That's me! Hey, caller (252) here's your answer." (returning node indicates that query is in tree)
- 252 says: "Hey, caller [123]! Here's your answer."
- 123 says: "Hey, Tree! Here's your answer."
Searching a BST Recursively Is $O(\log_2 N)$

- Search path: start with root $M$ and choose path to $I$ (for a reasonably balanced tree, $M$ will be more or less “in the middle,” and left and right subtrees will be roughly the same size)
  - Structurally, the height of a reasonably balanced tree with $n$ nodes is $\log_2 n$
  - At most, we visit each level of the tree once
  - So, runtime performance of searching is $O(\log_2 N)$ as long as tree is reasonably balanced which will be true if entry order is reasonably random (slide 22)

Top Hat Question

What’s the runtime of (recursive) search in a BST and why?

A. $O(n)$ – because you only iterate once
B. $O(2n)$ – because you go visit both the left and right subtrees
C. $O(n/2)$ – because you incorporate the idea of “bisection” to mean half the nodes
D. $O(\log_2 n)$ – because you incorporate the idea of “bisection” to eliminate half the number of nodes to search at each recursion
E. $O(n^2)$ – because recursion makes your runtime quadratic

Searching a Binary Tree Recursively

```java
public Node<Type> search(Type dataToFind) {
    // if _data is the thing we're searching for
    if(_data.compareTo(dataToFind) == 0) {
        return this;
    // if _data > dataToFind, can only be in left tree
    } else if(_data.compareTo(dataToFind) > 0) {
        if(_left != null) {
            return _left.search(dataToFind);
        } else {
            if(_right != null) {
                return _right.search(dataToFind);
            } else {
                return null;
            }
        }
    // Only get here if dataToFind isn't in tree, otherwise would've returned sooner
    return null;
}
```
Binary Search: Nearest Value

- What if we wanted to find the closest possible value, rather than the exact value?
  - search the tree rooted at M for a value or its nearest neighbor:
    - search(H, M) would return H
    - search(D, M) would return B
- Can use recursive binary search (with the tree doing the recursion rather than the node)
  - with one modification:
    - instead of immediately returning the element if you find it, have a variable that keeps track of the closest neighbor
  - search is still efficient, \(O(\log N)\)

Nearest Value Pseudocode

```java
search(type dataToFind, Node currNode):
  //method in BinaryTree class, not node's search
  if currNode is null: //base case
    return _closestNode //initialized elsewhere in BinaryTree class
  //see how close current data is to target data
  //getDistanceFrom returns an integer distance between nodes
  distToTarget = currNode.getDistanceFrom(dataToFind)
  //if this is closer than our current assumption of closest node, update
  if distToTarget < _closestDist:
    _closestDist = distToTarget
    _closestNode = currNode
  //if the data is equal to our target data, return immediately
  if currNode.compareTo(currNode.data, dataToFind) == 0:
    return _closestNode
  else if currNode.compareTo(currNode.data, dataToFind) == -1:
    //otherwise, recurse
    return this.search(dataToFind, currNode.right)
  else:
    return this.search(dataToFind, currNode.left)
```

Note: distance for integers: subtract them. Distance for single alphanumeric characters: adjacent ones are distance 1, hence distA/B = 1. For character strings, get the distance between the first 2 characters that differ.

Insertion into a BST(1/2)

- Search BST starting at root until we find where the data to insert belongs:
  - insert data when we reach a Node whose appropriate L or R child is null
- That Node makes a new Node, sets the new Node's data to the data to insert, and sets child reference to this new Node
- Runtime is \(O(\log N)\), yay!
  - \(O(\log N)\) to search the nearly balanced tree to find the place to insert
  - constant time operations to make new Node and link it in
Example: Insert 115

Before:

After:

//Tree's insert delegates to _root
public Node<Type> insert(Type newData) {
  //if tree is empty, make first node. No traversal necessary!
  if (_root == null) {
    _root = new Node(newData, null); //root's _parent is null
    return _root;
  } else {
    return _root.insert(newData); //delegate to Node's insert() method
  }
}

Insertion Code in Node

public Node<Type> insert(Type newData) {
  //Insert method continued!
  if (_data.compareTo(newData) > 0) {
    //newData should be in left subtree
    if (_left == null) { //left child is null - we've found the place to insert!
      _left = new Node(newData, this);
      return _left;
    } else { //keep traversing down tree
      return _left.insert(newData);
    }
  } else { //newData should be in right subtree
    if (_right == null) { //right child is null - we've found the place to insert!
      _right = new Node(newData, this);
      return _right;
    } else { //keep traversing down tree
      return _right.insert(newData);
    }
  }
  return null;
}
Insertion Simulation (1/4)
- Insert: 224
- First call insert in BST:
  ```java
  _root = _root.insert(newData);
  ```

Insertion Simulation (2/4)
- 123 says: "I am less than 224. I'll let my right child deal with it."
  ```java
  if (_data.compareTo(newData) > 0) {
    return _right.insert(newData);
  } else {
    _right = new Node(newData, this);
    return _right;
  }
  ```

Insertion Simulation (3/4)
- 252 says: "I am greater than 224. I'll pass it on to my left child -- but my left child is null!"
  ```java
  if (_data.compareTo(newData) > 0) {
    _left = new Node(newData, this);
    return _left;
  } else {
    // code for continuing traversal elided
  }
  ```
Insertion Simulation (4/4)

- 252 says: “You belong as my left child. 224. Let me make a node for you, make this new node your home, and set that node as my left child. Lastly, I will return a pointer to the new left node”. (And each node, as its recursive invocation ends, passes the pointer to the new 224 node up to its parent, eventually up to whatever method called on the tree’s search). _left = new Node(newData, this);
  return _left;

Before

\[16\] \[224\] \[123\]

After

\[16\] \[224\] \[123\]

Notes on Trees (1/2)

- Different insertion order of nodes results in different trees
  - if you insert a node referencing data value of 18 into empty tree, that node will become root
  - if you then insert a node referencing data value of 12, it will become left child of root
  - however, if you insert node referencing 12 into an empty tree, it will become root
  - then, if you insert one referencing 18, that node will become right child of root
  - even with same nodes, different insertion order makes different trees!
  - on average, for reasonably random (unsorted) arrival order, trees will look similar in depth so order doesn’t really matter.

Notes on Trees (2/2)

- When searching for a value, reaching another value that is greater than the one being searched for does not mean that the value being searched for is not present in tree (whereas it does in linked lists!)
  - it may well still be contained in left subtree of node of greater value that has just been encountered
  - thus, where you might have given up in linked lists, you can’t give up here until you reach a leaf (but depth is roughly log₂N for a nearly balanced tree, which is much smaller than N/2!)
Top Hat Question
Which tree does the code at the top right represent below?

A.  
B.  
C.  
D.

Remove: No Child Case
- **Node** to remove has no children (is a leaf)
  - just set the parent’s reference to this **Node** to **null** – no more references means the **Node** is garbage collected!
- Example: Remove **P**
  - set **O**’s right child to **null**, and **P** is gone!

Remove: One-Child Case
- Harder case: **Node** to delete has one child
  - replace **Node** child
- Example: Remove **O**
  - **O** has one child
    - **Q** replaces **O** by replacing its left child, previously **O**, with **P**
    - we know that all of the children of **O** are less than **Q** and greater than **M**. So, making **O**’s child a child of **Q** results in a valid BST!
Remove: Two-Children Case (1/3)

- Hard case: node to remove has two internal children
  - brute force: just flag node for removal, and rewrite tree at a later time — bad idea, because now every operation requires checking that flag.
  - Instead, do the work right away
  - this is tricky, because not immediately obvious which child should replace its parent
  - slow solution: re-insert each member of one of the sub-trees. Might be bad, O(n), even for a balanced tree.
  - non-obvious solution: first swap the data in node to be removed with data in a node that doesn’t have two children, then remove node using one of simpler remove cases

Remove: Two-Children Case (2/3)

- Use an auxiliary method, swapData
  - swaps data in node to be removed with the data in the right-most node in its left subtree
  - this child has a key value less than all nodes in the to-be-removed node’s right subtree, and greater than all other nodes in its left subtree
  - since it is a right-most node, it has at most one child because if it is the right most child, it won’t have any right children
  - this swap is temporary—we then remove the node in the right-most position using simpler remove

Remove: Two-Children Case (3/3)

- How do we remove R?
  - R has two children
  - swap R with the right-most node in the left subtree, the largest node less than R, which will be Q.
  - observe the following relationship, which must be maintained after the swap:
    - children in R’s left subtree are smaller than Q
    - children in R’s right subtree are larger than Q
    - R is in the wrong place but…
  - remove R (in its new position) using the one-child case
Remove: BST Code

- Starts as usual with delegating to root
- Nodes are "smart," so they can remove themselves
- Need to first find the Node to remove; if not null, it removes itself
- O(log(N)) because of searching in a nearly balanced tree

```
// in BinarySearchTree class:
//BinarySearchTree's remove takes a data element
public void remove(Type dataToRemove) {
    Node<Type> toRemove = _root.search(dataToRemove);
    if (toRemove != null) {
        //smart node's remove takes no params
        toRemove.remove();
    }
}
```

Remove: Node Code (1/3)

- In the Node class, remove method allows Node to remove itself

```
public Node<Type> remove() {
    //Case 1 – Node to remove is a leaf node
    //Set its parent's reference that originally refers to this Node to null
    if (_left == null && _right == null) {
        //If it's a leaf, set appropriate parent pointer to null
        if (_parent.getLeft() == this) {
            _parent.setLeft(null);
        } else {
            _parent.setRight(null);
        }
    } //Case 2 on next slide...
```

Remove: Node Code (2/3)

```
//In the one child case, we replace the _parent's reference to node with the Node's child
    } else if (_left != null && _right == null) { //case 2.1 - Node only has left child
        if (_parent.getLeft() == this) {
            _parent.setLeft(_left);
        } else {
            _parent.setRight(_left);
        }
    } else if (_left == null && _right != null) { //case 2.2 - Node only has right child
        if (_parent.getLeft() == this) {
            _parent.setLeft(_right);
        } else {
            _parent.setRight(_right);
        }
    } //Case 3 on next slide...
```
Remove: **Node** Code (3/3)

- Successor is guaranteed to have at most one child, so we remove with simpler remove case

```java
public Node<Type> remove() {
    // code for case 1 (no children) elided
    else { // case 2 - one child
        Node<Type> toSwap = this.swapData(); // swap data with successor
        toSwap.remove(); // remove toSwap, which holds original Node's data
        return toSwap; // return toSwap, since toSwap was data we removed
    }
    // swapData() defined on next slide
}
```

Remove: **swapData** Code

- We find the right-most Node in left subtree, but we can also find the left-most Node in right subtree

```java
public Node<Type> swapData() {
    Node<Type> curr = _left; // first get left child
    while(curr.getRight() != null) { // go right as far as possible
        curr = curr.getRight();
    }
    // swap data of this Node and successor
    Type tempData = _data;
    _data = curr.getData();
    curr.setData(tempData);
    return curr; // swapData() defined on next slide
}
```

N-ary Tree Example

- Use the first character of last name as the start of a binary tree for all names with that initial character
  - 26-way division right away
- **Disadvantages**
  - some trees will be very small, e.g. "Z" name will be much larger than average, e.g. "E". "X"
  - dividing by 26 doesn't really get you that much (log26 and log2 aren't that different)
Tree Runtime
- Binary Search Tree has a search of $O(\log n)$ runtime → can we make it faster?
- Could make a ternary tree (each node has at least 3 children)
  - $O(\log n)$ runtime
- Or a 10-way tree with $O(\log n)$ runtime
- Let’s try the runtime for a search with 1,000,000 nodes
  - $\log_{10}1,000,000 = 6$
  - $\log_21,000,000 < 20$, so shallower but broader tree
- Analysis: the logs are not sufficiently different and the comparison (basically an n-way nested if-else-if) is far more time consuming, hence not worth it
- Furthermore, binary tree makes it easy to produce an ordered list (see slide 67)

Traversing a Binary Tree
- We often want to access every Node in tree
  - so far, we have only searched for a single element
  - we can use a traversal algorithm to perform some arbitrary operation on every Node in tree
- Many ways to traverse Nodes in tree
  - order children are visited is important
  - three traversal types: inorder, preorder, postorder
- Exploit recursion!
  - subtree has same structure as tree

Inorder Traversal of BST
- Considered “in order” because Nodes are visited in sorted order
- Traverse left subtree first, then visit self, then traverse right subtree
- Use recursion!
- If we print our current Node’s data, this will print an alphabetical list!

```java
public void inOrder() {
    // Check for null children elided
        _left.inOrder();
    System.out.println(curr.data);
        _right.inOrder();
}
```

A B H I J L M O P S Y
Preorder Traversal of BST

- **Preorder traversal**
  - "preorder" because self is visited before ("pre") visiting children
  - again, use recursion!
  - note that we can recover the tree structure using the preorder result

```java
public void preOrder() {
    // Check for null children elided
    System.out.println(curr.data);
    _left.preOrder();
    _right.preOrder();
}
```

To learn more about the exciting world of trees, take CS16 (CSCI0160): Introduction to Algorithms and Data Structures.

---

Postorder Traversal of BST

- **Postorder traversal**
  - "post-order" because self is visited after ("post") visiting children
  - again, use recursion!

```java
public void postOrder() {
    // Check for null children elided
    _left.postOrder();
    _right.postOrder();
    System.out.println(curr.data);
}
```

---

Prefix, Infix, Postfix Notation for Arithmetic Expressions

- **Infix, Prefix, and Postfix** refer to where the operator goes relative to its operands
  - **Infix** (fully parenthesized): 
    \((1\times2)+(3\times4)-(5\times1/2)\)
  - **Prefix**:
    \(+\times\times\times-\times\times\times/\)
  - **Postfix**:
    \(1\times3\times4+5+6-8++\)

- **Graphical representation for equation:**
Using Prefix, Infix, Postfix Notation

- When you type an equation into a spreadsheet, you use Infix; when you type an equation into many Hewlett-Packard calculators, you use Postfix, also known as “Reverse Polish Notation,” or “RPN,” after its inventor Polish Logician Jan Lukasiewicz (1924).
- Easier to evaluate Postfix because it has no parentheses and evaluates in a single left-to-right pass.
- Use Dijkstra’s 2-stack shunting yard algorithm to convert from user-entered Infix to easy-to-handle Postfix – compile or interpret it on the fly.

Dijkstra’s Infix-to-Postfix Algorithm (1/2)

2 stack algorithm for single-pass Infix to Postfix conversion, using operator precedence

\[(a + (b \times (c ^ d))) \Rightarrow a \ b \ c \ d \ ^ \ * \ +\]

Use rule matrix to implement strategy

A) Push operands onto operand stack; push operators in precedence order onto the operator stack.

B) When precedence order would be disturbed, pop operator stack until order is restored; evaluating each pair of operands popped from the operand stack and pushing the result back onto the operand stack.

Note that equal precedence displaces. At the end of the statement (marked by ; or CR) all operators are popped.

C) “(“ starts a new substack; “)” pops until it’s matching “(“.

Dijkstra’s Infix-to-Postfix Algorithm (2/2)

\[(a + (b \times (c ^ d))) \Rightarrow a \ b \ c \ d \ ^ \ * \ +\]
Challenge Questions Solutions

- Q: How would you print the elements of a Binary Search Tree in increasing order?
- A: You would traverse the BST in-order.
- Q: How would you find the ‘successor’ (i.e., next-greatest number) of a node in a Binary Search Tree?
- A: The pseudo-code for the solution is to find left-most node of right subtree since it is the smallest of the ones that are greater:

```java
if (node.hasRight()) {
    node = node.right();
    while (node.hasLeft()) {
        node = node.left();
    }
    return node;
}
```

Announcements

- Due to departmental machine issues we have extended the Tetris deadlines by one day.
- Tetris deadlines:
  - Early: Friday, 11/15
  - On-time: Sunday, 11/17
  - Late: Tuesday, 11/19