Binary Trees

Basic terminology
- Finite set of nodes (may be empty -- 0 nodes), which contain data
- First node in tree is called the root

Basic terminology
- Each node may be linked to 0, 1 or 2 child nodes (or children)
- A node with children is a parent, a node with no children is a leaf

Nodes d and e are children of b
Node c is parent of f and g

More botany & sociology
- The root node has no parent; it is the ancestor of all other nodes
- All nodes in the tree are descendants of the root node
- Nodes with the same parent are siblings
- Any node with descendants (any parent node) is the root node of a subtree

But wait, there’s more!
- Tree depth is the maximum number of steps through ancestor nodes from the deepest leaf back to root
  - tree with single node (root) has depth of 0
  - empty tree has depth of -1

Depth of entire tree is 2; depth of b’s subtree is 1; depth of f’s subtree is 0

Tree shape terminology
- Full binary tree
  - every parent has 2 children,
  - every leaf has equal depth
Tree shape terminology

- **Complete** binary tree
  - every level is full except possibly the deepest level
  - if the deepest level isn’t full, leaf nodes are as far to the left as possible

Array representation of a binary tree

- Complete binary tree is easily represented as an array (either static or dynamic)
- Root data is stored at index 0
- Root’s children are stored at index 1 and 2

Array representation of a binary tree

- In general:
  - root is tree[0]
  - for any node at tree[n], the parent node will be found at index (n-1)/2
  - for any node at tree[n], the children of n (if any) will be found at tree[2n + 1] and tree[2n + 2]
Array representation of a binary tree

- What if tree isn’t complete?
  - Can still use array, but problem is more complicated -- have to keep track of which children actually exist
  - One possibility would be to place a known dummy value in empty slots
  - Another is to add another private member to the class -- an array of bools paired to the data array -- index value is true if data exists, false if not

Linked representation of binary tree

- Again, as with linked list, entire tree can be represented with a single pointer -- in this case, a pointer to the root node
- Nodes are instances of class BTNode, described in the next several slides
- The class contains methods that operate on single nodes
- Since each node is potentially the root node of a tree (or subtree), we will also include operations on entire trees

Linked representation of binary tree

public class BTNode<E> {
    private E data;
    private BTNode<E> left, right;
}

Private member data holds the information in the node; members left and right are pointers to the node’s left and right children

Methods of BTNode class

- Constructor: creates node with specified data and left and right children
- Observer methods
  - getData() returns value of data member
  - getLeft() returns pointer to left child
  - getRight() returns pointer to right child
  - getLeftmostData() and getRightmostData return the data members from the leaves
  - isLeaf() returns true if node is a leaf

Methods of BTNode class

- Mutators
  - setData(): changes data portion of node
  - setLeft(): assigns new left child to node
  - setRight(): assigns new right child to node
  - removeLeftmost() and removeRightmost(): recursive methods that prune off leaves, returning the parent node

Static methods that operate at the tree level

- Observers (entire tree)
  - treeSize() returns number of nodes in tree
  - treeCopy() returns a new tree copied from a source tree argument
Output methods

- Traversal methods (entire tree):
  - preorderPrint(): prints data from nodes in preorder traversal pattern
  - postorderPrint(): prints data from nodes in postorder traversal pattern
  - print(): prints data from nodes in inorder traversal pattern

Constructor

```java
public BTreeNode<>(E initialData, BTreeNode<> initialLeft,
                   BTreeNode<> initialRight)
{
    data = initialData;
    left = initialLeft;
    right = initialRight;
}
```

Simple accessor methods

```java
public E getData() {
    return data;
}
public BTreeNode<> getLeft() {
    return left;
}
public BTreeNode<> getRight() {
    return right;
}
```

Leaf accessors

```java
public E getLeftmostData() {
    if (left == null)
        return data;
    else
        return left.getLeftmostData();
}
public E getRightmostData() {
    if (left == null)
        return data;
    else
        return left.getRightmostData();
}
```

Mutators

```java
public void setData(E newData) {
    data = newData;
}
public void setLeft(BTreeNode<> newLeft) {
    left = newLeft;
}
public void setRight(BTreeNode<> newRight) {
    right = newRight;
}
```

Mutators

```java
public BTreeNode<> removeLeftmost() {
    if (left == null)
        return right;
    else {
        left = left.removeLeftmost();
        return this;
    }
}
public BTreeNode<> removeRightmost() {
    if (right == null)
        return left;
    else {
        right = right.removeRightmost();
        return this;
    }
}
```
Static methods

```java
public static <E> BTNode<E> treeCopy(BTNode<E> source) {
    BTNode<E> leftCopy, rightCopy;
    if (source == null)
        return null;
    else {
        leftCopy = treeCopy(source.left);
        rightCopy = treeCopy(source.right);
        return new BTNode<E>(source.data, leftCopy, rightCopy);
    }
}
```

Tree traversal

- Any operation that performs some process on all the nodes in a tree must perform a tree traversal
- Traversal refers to visiting each node in turn
- Traversal is a recursive process: we visit each node, and we visit each node in the subtree of which the node is the root

Now, more ways to climb!

- There are three basic tree traversal patterns, referring to the order in which nodes are visited and processed
  - Pre-order: visit root, then left subtree, then right
  - In-order: visit left subtree, then root, then right
  - Post-order: visit left subtree, then right, then root

Example: printing all nodes

```java
// pre-order traversal
public void preorderPrint() {
    System.out.println(data);
    if (left != null)
        left.preorderPrint();
    if (right != null)
        right.preorderPrint();
}
```

Pre-order traversal in action

![Pre-order traversal in action](image_url)
Example: printing all nodes

```java
// post-order traversal
public void postorderPrint( )
{
    if (left != null)
        left.postorderPrint( );
    if (right != null)
        right.postorderPrint( );
    System.out.println(data);
}
```

Example: printing all nodes

```java
// in-order traversal, with indents
public void print(int depth) {
    int i;
    // Print the left subtree (or a dash
    // if there is a right child and
    // no left child)
    if (left != null)
        left.print(depth+1);
    else if (right != null) {
        for (i = 1; i <= depth+1; i++)
            System.out.print(" ");
        System.out.println("--");
    }
}
```

Example: printing all nodes

```java
// Print the right subtree (or a dash if
// there is a left child and no right child)
if (right != null)
    right.print(depth+1);
else if (left != null){
    for (i = 1; i <= depth+1; i++)
        System.out.print(" ");
    System.out.println("--");
}
```

Post-order traversal in action

Original tree: Results:
```
K
W
I
O
D
O
R
K
```

Example: printing all nodes

```java
// Print the indentation and the data
// from the current node:
for (i = 1; i <= depth; i++)
    System.out.print(" ");
System.out.println(data);
```

Example: printing all nodes

```java
K
I
W
K
R
O
D
```

In-order traversal in action

Original tree: Results: