Binary Trees

Chapter 12
Fundamentals

• A binary tree is a **nonlinear** data structure.

• A **binary tree** is either empty or it contains a root node and left- and right- subtrees that are also binary trees.

• Applications: encryption, databases, expert systems
Tree Terminology

- Root
- Parent
- Left-child
- Right-child
- Siblings
- Leaf
- Right-subtree
- Left-subtree
More Terminology

• Consider two nodes in a tree, X and Y.
• X is an ancestor of Y if
  X is the parent of Y, or
  X is the ancestor of the parent of Y.
  *It’s RECURSIVE!*

• Y is a descendant of X if
  Y is a child of X, or
  Y is the descendant of a child of X.
More Terminology

• Consider a node $Y$.

• The depth of a node $Y$ is
  0, if the $Y$ is the root, or
  $1 + \text{the depth of the parent of } Y$

• The depth of a tree is the maximum depth of all its leaves.
More Terminology

- A **full binary tree** is a binary tree such that
  - all leaves have the same depth, and
  - every non-leaf node has 2 children.
- A **complete binary tree** is a binary tree such that
  - every level of the tree has the maximum number of nodes possible except possibly the deepest level.
  - at the deepest level, the nodes are as far left as possible.
A full binary tree is always a complete binary tree.

What is the number of nodes in a full binary tree?
Tree Traversals

• preorder traversal
  1. Visit the root.
  2. Perform a preorder traversal of the left subtree.
  3. Perform a preorder traversal of the right subtree.

• inorder traversal
  1. Perform an inorder traversal of the left subtree.
  2. Visit the root.
  3. Perform an inorder traversal of the right subtree.

• postorder traversal
  1. Perform a postorder traversal of the left subtree.
  2. Perform a postorder traversal of the right subtree.
  3. Visit the root.
Traversal Example

preorder
A B D E C F G

inorder
D B E A F C G

postorder
D E B F G C A
Traversals Example

- Preorder: ABDFGCEHI
- Inorder: BFDGAEIHC
- Postorder: FGDBIHECA
Traversal Example
(expression tree)

preorder: *+AB/CD
inorder: A+B*C/D
postorder: AB+CD/*
More Examples

Preorder: ABC

Diagram showing a binary tree with nodes labeled A, B, and C, demonstrating preorder traversal.
Implementing a binary tree

• Use an array to store the nodes.
  - mainly useful for complete binary trees (next week)

• Use a variant of a **singly**-linked list where each data element is stored in a node with links to the left and right children of that node

• Instead of a head reference, we will use a **root** reference to the root node of the tree.
public class BTNode {
    private int data;
    private BTNode left;
    private BTNode right;

    // BTNode methods
}

Binary Tree Node (BTNode)
public BTNode(int initData) {
    data = initData;
    left = null;
    right = null;
}

public int getData()
public void setData(int newData)
public BTNode getLeft()
public void setLeft(BTNode newLeft)
public BTNode getRight()
public void setRight(BTNode newRight)
public void inorder()
{
    if (left != null)
        left.inorder();
    System.out.println(data);
    if (right != null)
        right.inorder();
}
BTNode methods (cont’d)

```java
public void preorder()
{
    System.out.println(data);
    if (left != null)
        left.preorder();
    if (right != null)
        right.preorder();
}
```
BTNode methods (cont’d)

```java
public void traverse()
{
    //preorder
    if (left != null)
        left.traverse();

    //inorder
    if (right != null)
        right.traverse();

    //postorder
}
```
Which traversal is more appropriate?

- Evaluating an expression tree.
- Add all numbers in a tree.
- Find the maximum element of a tree.
- Print a tree rotated 90 degrees (in a counter clockwise fashion).
- Find the depth of a tree.
- Set the data field of each node to its depth.
A Binary Tree Class

• We will implement a specific type of binary tree: a **binary search tree**.

• A binary search tree (BST) is a binary tree such that
  - all the nodes in the left subtree are less than the root
  - all the nodes in the right subtree are greater than the root
  - the subtrees are BSTs as well
public class BinarySearchTree { 
    private BTNode root;
    public BinarySearchTree() {
        root = null;
    }
    public boolean isEmpty() {
        return (root == null);
    }
}
public void inorder() {
    if (root != null) {
        root.inorder();
    }
}

This is a call to inorder() from the BTN Node class!
This is not a recursive call!

// other BinarySearchTree methods
Inserting into a BST

45 21 39 83 62 96 11
Inserting into a BST
public void insert(int item) {
    BTNode newNode;
    BTNode cursor;
    boolean done = false;
    if (root == null) {
        newNode = new BTNode(item);
        root = newNode;
    }
}
else {
    cursor = root;
    while (!done) {
        if (item < cursor.getData()) {
            if (cursor.getLeft() == null) {
                newNode = new BTNode(item);
                cursor.setLeft(newNode);
                done = true;
            }
        } else cursor = cursor.getLeft();
    }
}

else if (item > cursor.getData()) {
    if (cursor.getRight() == null) {
        newNode = new BTNode(item);
        cursor.setRight(newNode);
        done = true;
    }
}
else cursor = cursor.getRight();

else done = true;  // Why?
} // end while
Traversing the BST

An inorder traversal on a BST sorts the numbers!
Efficiency of “insert”

• For a full binary tree with \( N \) nodes, what is its depth \( d \)?

\[
N = 2^0 + 2^1 + 2^2 + \ldots + 2^d
= 2^{d+1} - 1
\]

Thus, \( d = \log_2(N+1) - 1 \).

• An insert will take \( O(\log N) \) time on a full BST since we have to examine one node at each level before we find the insert point, and there are \( d \) levels.
Efficiency of “insert” (cont’d)

• Are all BSTs full? **NO!**

Insert the following numbers in order into a BST: 11 21 39 45 62 83 96
What do you get?

• An insert will take O(N) time on an arbitrary BST since there may be up to N levels in such a tree.
Removing from a BST

General idea:

• Start at the root and search for the item to remove by progressing one level at a time until either:
  - the item is found
  - we reach a leaf and the item is not found

• If the item is found, remove the node and repair the tree so it is still a BST.
Removing from a BST

```java
public boolean remove(int item) {
    BTNode cursor = root;
    BTNode parentOfCursor = null;
    while (cursor != null && cursor.getData() != item) {
        parentOfCursor = cursor;
        if (item < cursor.getData())
            cursor = cursor.getLeft();
        else cursor = cursor.getRight();
    }
}
```
Removing from a BST (cont’d)

Case 1: item not in tree (e.g. item = 74)

if (cursor == null)
return false;

```java
parentOfCursor = null

45

21

11

39

62

83

96

cursor

cursor

cursor

cursor

cursor
```

```java
parentOfCursor = null
```
Removing from a BST (cont’d)

Case 2:
item is root and root has no left child (e.g. item=45)

```java
else {
    if (cursor == root &&
        cursor.getLeft() == null) {
        root = root.getRight();
    }
}
```
Case 3(a): item is in a non-root node without a left child

```java
else if (cursor != root && cursor.getLeft() == null) {
    if (cursor == parentOfCursor.getLeft()) {
        parentOfCursor.setLeft(cursor.getRight());
    }
}
```
Case 3 (b): item is in a non-root node without a left child

```java
else
    parentOfCursor.setRight(cursor.getRight());
}
```
Case 4: item is in a non-root node but node has a left child

Find the largest value in the left subtree of cursor and move it to the cursor position. This value is the “rightmost” node in the left subtree.
Removing from a BST (cont’d)

Case 4 (cont’d):
item is in a non-root node but node has a left child

Copy the rightmost data into the cursor node.
Then remove the rightmost node (be careful: it might have a left subtree!)
Removing from a BST (cont’d)

```java
else {
    cursor.setData(
        cursor.getLeft()
        .getRightmostData());
    cursor.setLeft(
        cursor.getLeft()
        .removeRightmost());
}

return true;
```
Additional **BTNode** methods

```java
public int getRightmostData() {
    if (right == null) return data;
    else return right.getRightmostData();
}
public BTNode removeRightmost() {
    if (right == null) return left;
    else {
        right = right.removeRightmost();
        return this;
    }
}
```

Order of complexity for BST removal?