Binary Tree Traversals

Preorder Traversal

(1) Process the **root**
(2) Process the nodes in the **left** subtree with a recursive call
(3) Process the nodes in the **right** subtree with recursive call

```
void BINARYTREE::preorderPrint(const BNODE* nodePtr) const {
  if (nodePtr != NULL) {
     cout << nodePtr->getData( ) << endl;
     preorderPrint(nodePtr->getLeft( ));
     preorderPrint(nodePtr->getRight( ));
  }
}
```

Inorder Traversal

(1) Process the nodes in the **left** subtree with a recursive call
(2) Process the **root**
(3) Process the nodes in the **right** subtree with recursive call

Postorder Traversal

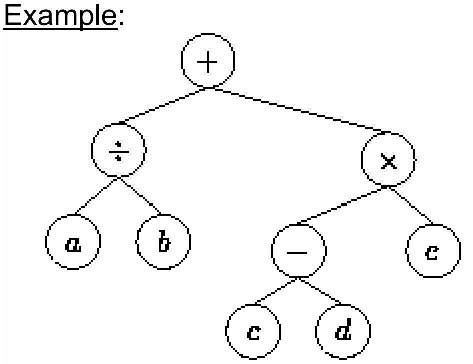
(1) Process the nodes in the **left** subtree with a recursive call
(2) Process the nodes in the **right** subtree with recursive call
(3) Process the **root**

Expression Trees

- Binary tree where **non-leaf (i.e., internal) nodes are operators** and **leaf nodes are operands**

- Inorder/preorder/postorder traversal gives infix/prefix/postfix expression

Example:



infix expression: (a / b) + ((c – d) * e)

prefix expression: + (/ a b) (* (- c d) e)

postfix expression: (a b /) ((c d -) e *) +

Binary Search Trees

Binary tree where:

(1) value in a node is **≥** value of every node in its **left subtree**, and
(2) value in a node is **<** value of every node in its **right subtree**

Insertion

Starting from root of tree, follow child links (according to ordering rules given above) to determine where new value should be. Then add it at that position. Note: It will always become a new leaf node.

Runtime analysis: In worst case, will have to traverse the height of the tree, so O(h).
Worst possible height in unbalanced tree is n.

***Note: A balanced binary tree containing n nodes will have height O(log n)**

Deletion

Input: root of tree (or subtree), and value to delete is x

(1) If **tree empty** (i.e., root == NULL), then x is not in the tree.

(2) If tree non-empty and **x < root's value**, then x must be in root's left subtree. So call delete passing it root->left and x.

(3) If tree non-empty and **x > root's value**, then x must be in root's right subtree. So call delete passing it root->right and x.

(4) If tree non-empty and **x == root's value**, then need to delete this node. But this node could have children!

   a. If **root has no left child**, then delete root node and make its right child be the new root node. (Note: this case also works if root is a leaf node.)

   b. If **root has a left child**, then replace root with largest value node in left subtree. (Note: see discussion of bst_remove_max on pp. 523-524 in text).

Runtime analysis: In worst case, will have to traverse the height of the tree, so O(h).
Worst possible height in unbalanced tree is n.