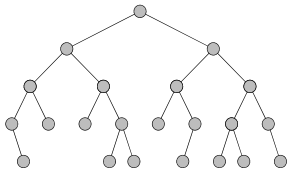


CSIS 3103

Ch 6: Binary trees,
Tree traversals,
Binary search trees

Tree Traversals

Often we want to process some or all of the nodes of a tree

Tree traversal: Walking through the tree in a prescribed order and visiting the nodes as they are encountered

Three kinds of tree traversal

- Inorder
- Preorder
- Postorder

Binary Tree Traversals

Preorder: Visit root node, traverse T_L, traverse T_R

Inorder: Traverse T_L, visit root node, traverse T_R

Postorder: Traverse T_L, Traverse T_R, visit root node

Algorithm for Preorder Traversal

1. if the tree is empty
2. Return.

else

3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

Algorithm for Inorder Traversal

1. if the tree is empty
2. Return.

else

3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

Algorithm for Postorder Traversal

1. if the tree is empty
2. Return.

else

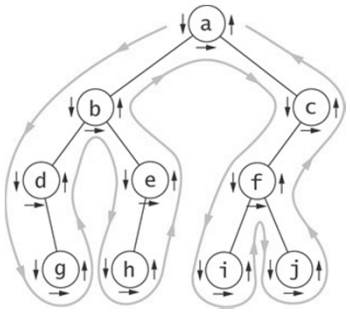
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Visualizing Tree Traversals

Visualize a tree traversal by imagining walking along the branches of the tree

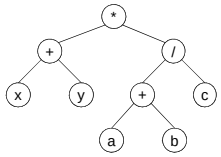
– If you always keep the tree to the left, you will trace a route known as the Euler tour
  - Preorder traversal processes each node is when it is first seen
  - Inorder processes each node when returning from traversing its left subtree
  - Postorder processes each node when it is last seen

Visualizing Tree Traversals



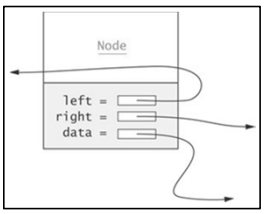
Traversals of Expression Trees

- An inorder traversal of an expression tree inserts parenthesis where they belong for infix form
- A postorder traversal of an expression tree results in postfix form

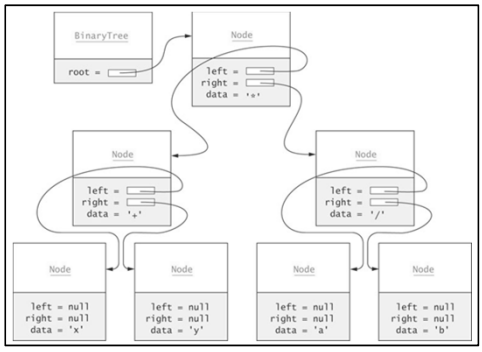


The Node<E> Class (Binary Tree)

A node consists of a data part and links to successor nodes (its left and right subtrees)



Binary Tree for an Expression



The BinaryTree<E> Class

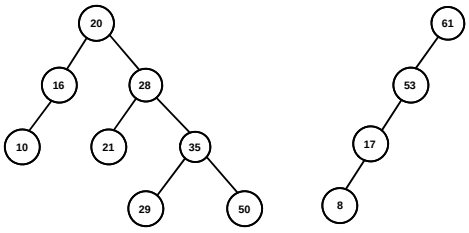
Data Field	Attribute
`protected Node<E> root`	Reference to the root of the tree.
Constructor	**Behavior**
`public BinaryTree()`	Constructs an empty binary tree.
`protected BinaryTree(Node<E> root)`	Constructs a binary tree with the given node as the root.
`public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree)`	Constructs a binary tree with the given data at the root and the two given subtrees.
Method	**Behavior**
`public BinaryTree<E> getLeftSubtree()`	Returns the left subtree.
`public BinaryTree<E> getRightSubtree()`	Returns the right subtree.
`public E getData()`	Returns the data in the root.
`public boolean isLeaf()`	Returns **`true`** if this tree is a leaf, **`false`** otherwise.
`public String toString()`	Returns a `String` representation of the tree.
`private void preOrderTraverse(Node<E> node, int depth, StringBuilder sb)`	Performs a preorder traversal of the subtree whose root is `node`. Appends the representation to the `StringBuilder`. Increments the value of `depth` (the current tree level).
`public static BinaryTree<E> readBinaryTree(Scanner scan)`	Constructs a binary tree by reading its data using `Scanner scan`.

Binary Search Trees

A set of nodes T is a *binary search tree* if either of the following is true

- T is empty
- Its root has two subtrees such that each is a binary search tree and the value in the root is greater than all values of the left subtree but less than all values in the right subtree

Binary Search Trees



Searching a Binary Search Tree