Trees

Professor Frank Kreimendahl

School of Computing and Data Science
Wentworth Institute of Technology

October 19, 2022

Introduction

Introduction

- All previous data structures have been linear – from a given element we can look forwards or backwards and find one element
- Trees are nonlinear
- Trees have a hierarchical structure that can signify data relationships:
  - Class hierarchies
  - Disk directories/file systems
  - Family tree

Binary Trees

- Trees can be defined recursively
- Recursive methods for querying/modifying trees are simple
- We will focus on *binary* trees: each element has two ‘next’ values and one ‘previous’ value
- These trees can be represented with an array or collection of nodes
- Some trees allow for more efficient operations than their linear data structure counterparts – fewer steps to accomplish the same task

Binary Trees

Introduction

Binary Trees

Terminology

Definition

Expression Tree

Huffman Tree

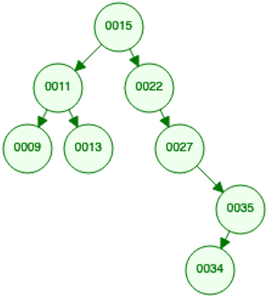
Binary Search Tree

Traversal

BinaryTree Class

Tree Terminology

- **Node**: An element holder regardless of implementation
- **Root**: The top node in a tree
- **Branch**: One of two 'next' nodes
- **Children**: Tree term for 'next' nodes, looking down a tree
- **Parent**: The reciprocal relation of a child
  - Every node has one parent except the *root*
- **Leaf node**: a node with no children
- **Subtree**: Any node from the tree combined with its descendants



More Tree Terminology

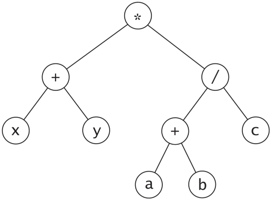
- **Level**: Distance to root node + 1
  - If node n is root, its level is 1
  - Otherwise, node n's level is $1+$ its parent's level
- **Height**: The number of nodes in the longest path from a leaf to the root
- **Binary Tree**: A tree in which every node has up to two children
- **Full Tree**: A binary tree where each node has 0 or 2 children
- **Perfect Tree**: A binary tree which has every level filled completely
- **Complete Tree**: A binary tree which only has gaps on the lowest level, and those gaps only appear to the right

Binary Tree Definition

- In a binary tree, each node has two subtrees
- A set of nodes T is a binary tree if either:
  - T is empty
  - T's root is a node with left and right subtrees which are binary trees

Expression Tree

- Each node contains an operator or operand
- Operands only appear in leaf nodes
- Parentheses are not stored because they are implicit in the tree structure
- Operators at levels closer to root are evaluated after deeper levels

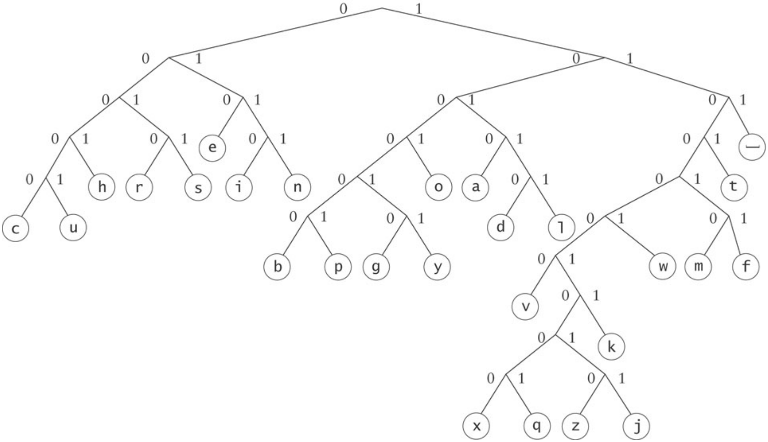


$$(x+y)*((a+b)/c)$$

Huffman Tree

- Represents Huffman codes for characters that appear in a text document
- Rather than ASCII, uses variable bit sizes to represent different characters
- More common characters are represented with fewer bits
- Allows memory compression based on character frequency
- Allows for fast compression and decompression with a Huffman binary tree

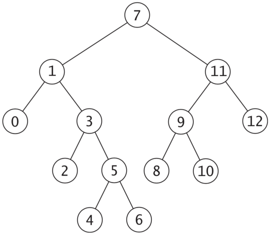
Huffman Tree



Huffman Tree

Binary Search Tree

- A binary tree where, for each node n, all values in n's left subtree are less than n, and all values in n's right subtree are greater than n
- New elements can only be inserted in specific positions
- Elements can only be found in certain positions
- Operations are efficient because not every element needs to be examined/shifted to maintain BST



Binary Search Algorithm

Starting at a tree's root, this algorithm recursively searches in a tree for a target:

SEARCH(tree, target)

```
if tree is empty then
    return null // target not found
else if target matches root of tree then
    return root node
else if target < root node then
    return SEARCH(root's left child, target)
else
    return SEARCH(root's right child, target)
```

Traversal

Tree Traversal

- The order of previous data structures is straightforward – move from front to back
- We can move through a tree to visit each node
- This process is called *tree traversal*
- There are three common ways to traverse a tree:
    - preorder traversal
    - inorder traversal
    - postorder traversal

Preorder Traversal

Starting at a tree's root, this algorithm recursively visits both subtrees:

PREORDER(root)

```
1: if tree is empty then
2:     return
3: else
4:     visit root
5:     preorder(root.left)
6:     preorder(root.right)
```

Inorder Traversal

INORDER(root)

```
if tree is empty then
    return
else
    inorder(root.left)
    visit root
    inorder(root.right)
```

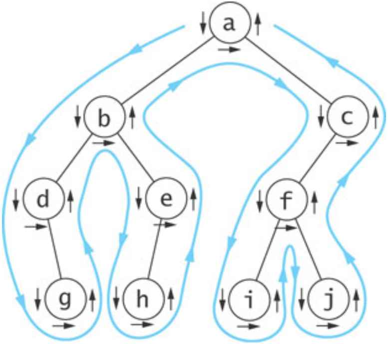
Postorder Traversal

POSTORDER(root)

```
if tree is empty then
    return
else
    postorder(root.left)
    postorder(root.right)
    visit root
```

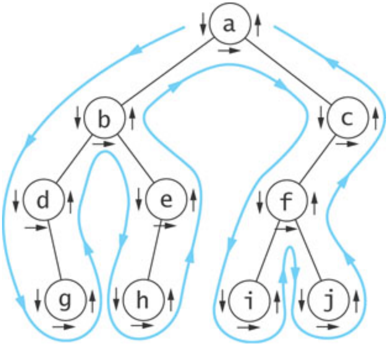
Traversal Visualization

- Imagine the tree painted on the ground
- Always walk with your left foot next to the tree as you walk
- This traversal is called an *Euler tour*



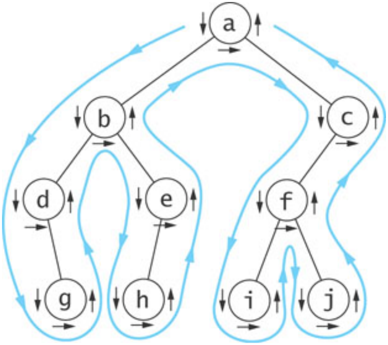
Preorder Traversal

- Blue path follows preorder traversal
- Visit a node before visiting subtrees
- Visitation occurs for downward pointing arrow – when node is first encountered
- The sequence in this example is a b d g e h c f i j



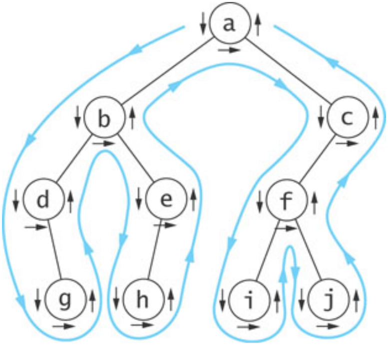
Inorder Traversal

- Visit a node between left and right subtree
- Visitation occurs for horizontal arrow – after left subtree but before right subtree
- The sequence in this example is d g b h e a i f j c



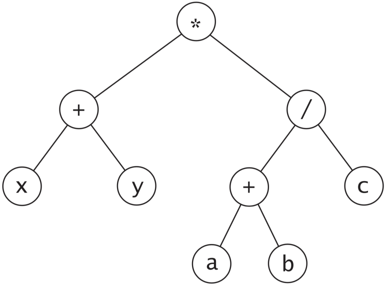
Postorder Traversal

- Visit a node just before leaving it for the last time
- Visitation occurs for upward arrow – after both subtrees have been fully explored
- The sequence in this example is g d h e b i j f c a



Traversal of Expression Tree

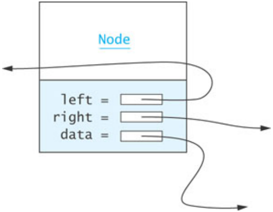
- A postorder traversal of an expression tree results in the sequence `x y + a b + c / *`
- This is postfix notation that we saw with stacks!
- We can generate prefix and infix notation similarly (though infix requires some parentheses)



BinaryTree Class

Node Class

- Similar to a linked list, a node holds data and references to other nodes
- The data is a reference to generic type E
- A node has a reference to the root of both subtrees



Node Class Implementation

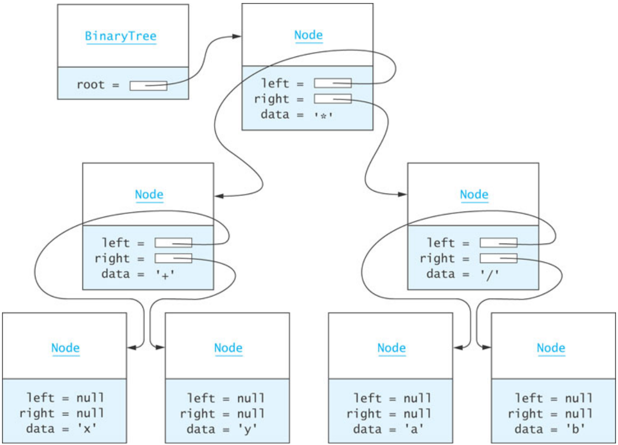
```
protected static class Node<E>
                              implements Serializable {
  protected E data;
  protected Node<E> left;
  protected Node<E> right;

  public Node(E data) {
    this.data = data;
    left = null;
    right = null;
  }

  public String toString() {
  return data.toString();
  }
}
```

Example Tree

The `BinaryTree` class only holds a reference to the `root`



Implementation

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`.

Class Definition

```
import java.io.*;

public class BinaryTree<E> implements Serializable {
  // Insert inner class Node<E> here

  protected Node<E> root;

  // Insert constructors and methods here
}
```

Constructors

```
public BinaryTree() {
  root = null;
}

protected BinaryTree(Node<E> root) {
  this.root = root;
}
```

Constructor

Given a data value and two subtrees to join under a new root

```
public BinaryTree(E data, BinaryTree<E> leftTree,
                  BinaryTree<E> rightTree) {
  root = new Node<E>(data);
  if (leftTree != null)
    root.left = leftTree.root;
  else
    root.left = null;

  if (rightTree != null)
    root.right = rightTree.root;
  else
    root.right = null;
}
```

Getting Subtrees

```
public BinaryTree<E> getLeftSubtree() {
  if (root != null && root.left != null)
    return new BinaryTree<E>(root.left);
  else
    return null;
}
```

`getRightSubtree` is symmetrical.

isLeaf

```
public boolean isLeaf() {
  return (root.left == null && root.right == null);
}
```

toString

Generate a string of data encountered in a preorder traversal with indentation to show each value's depth in the tree

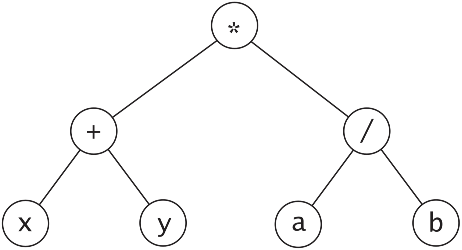
```
public String toString() {
  StringBuilder sb = new StringBuilder();
  preOrderTraverse(root, 1, sb);
  return sb.toString();
}
```

preOrderTraverse

```
private void preOrderTraverse(Node<E> node, int depth,
                              StringBuilder sb) {

  for (int i = 1; i < depth; i++) {
    sb.append("  "); // indentation
  }
  if (node == null) {
    sb.append("null\n");
  } else {
    sb.append(node.toString());
    sb.append("\n");
    preOrderTraverse(node.left, depth + 1, sb);
    preOrderTraverse(node.right, depth + 1, sb);
  }
}
```

toString Example



toString results

```
*
  +
    x
      null
      null
    y
      null
      null
  /
    a
      null
      null
    b
      null
      null
```