Trees

- Trees are hierarchical data structures
  - One-to-many relationship between elements

- Tree node / element
  - Contains data
  - Referred to by only 1 (parent) node
  - Contains links to any number of (children) nodes
Trees

Terminology

- **Root** ⇒ node with no parent
- **Leaf** ⇒ all nodes with no children
- **Interior** ⇒ all nodes with children
**Trees**

**Terminology**

- **Sibling** ⇒ node with same parent
- **Descendent** ⇒ children nodes & their descendents
- **Subtree** ⇒ portion of tree that is a tree by itself
  ⇒ a node and its descendents
Trees

Terminology

- **Level** ⇒ is a measure of a node’s distance from root
  - **Definition of level**
    - If node is the root of the tree, its level is 1
    - Else, the node’s level is 1 + its parent’s level

- **Height (depth)** ⇒ max level of any node in tree

![Diagram of a tree with height = 3](image)
Binary Trees

- **Binary tree**
  - Tree with 0–2 children per node
  - Left & right child / subtree

---

Binary Tree

- Parent
  - Left Child
  - Right Child
Tree Traversal

Often we want to

1. Find all nodes in tree
2. Determine their relationship

Can do this by

1. Walking through the tree in a prescribed order
2. Visiting the nodes as they are encountered

Process is called tree traversal
Tree Traversal

Goal
- Visit every node in binary tree

Approaches
- Depth first
  - Preorder $\Rightarrow$ parent before children
  - Inorder $\Rightarrow$ left child, parent, right child
  - Postorder $\Rightarrow$ children before parent
- Breadth first $\Rightarrow$ closer nodes first
Tree Traversal Methods

### Pre-order
1. Visit node // first
2. Recursively visit left subtree
3. Recursively visit right subtree

### In-order
1. Recursively visit left subtree
2. Visit node // second
3. Recursively right subtree

### Post-order
1. Recursively visit left subtree
2. Recursively visit right subtree
3. Visit node // last
Tree Traversal Methods

- **Breadth-first**

**BFS(Node n) {**

```
Queue Q = new Queue();
Q.enqueue(n); // insert node into Q
while ( !Q.empty()) {
    n = Q.dequeue(); // remove next node
    if ( !n.isEmpty()) {
        visit(n); // visit node
        Q.enqueue(n.Left()); // insert left subtree in Q
        Q.enqueue(n.Right()); // insert right subtree in Q
    }
}
```
Tree Traversal Examples

- **Pre-order (prefix)**
  - $+ \times 2 3 / 8 4$

- **In-order (infix)**
  - $2 \times 3 + 8 / 4$

- **Post-order (postfix)**
  - $2 3 \times 8 4 / +$

- **Breadth-first**
  - $+ \times / 2 3 8 4$

Expression tree
Binary Tree Implementation

Using a class to represent a Node

Class Node {
  KeyType key;
  Node left, right;  // null if empty
}

Node root = null; // Empty Tree

Using a Polymorphic Binary Tree

We will talk about this implementation later on
Types of Binary Trees

- **Degenerate**
  - Mostly 1 child / node
  - Height = $O(n)$
  - Similar to linear list

- **Balanced**
  - Mostly 2 child / node
  - Height = $O(\log(n))$
  - $2^{\text{Height}} - 1 = n$ (# of nodes)
  - Useful for searches

Degenerate binary tree

Balanced binary tree
Binary Search Trees

Key property

Value at node
- Smaller values in left subtree
- Larger values in right subtree

Example
- $X > Y$
- $X < Z$
Binary Search Trees

Examples

Binary search trees

Non-binary search tree
Tree Traversal Examples

- **Pre-order**
  - 44, 17, 32, 78, 50, 48, 62, 88

- **In-order**
  - 17, 32, 44, 48, 50, 62, 78, 88

- **Post-order**
  - 32, 17, 48, 62, 50, 88, 78, 44

- **Breadth-first**
  - 44, 17, 78, 32, 50, 88, 48, 62

**Binary search tree**

Sorted order!
Example Binary Searches

Find (2)

- 2 < 10, left
- 2 < 5, left
- 2 = 2, found

- 2 < 5, left
- 2 = 2, found
Example Binary Searches

Find (25)

25 > 10, right
25 < 30, left
25 = 25, found

25 > 5, right
25 < 45, left
25 < 30, left
25 > 10, right
25 = 25, found
Binary Search Properties

- **Time of search**
  - Proportional to height of tree
  - Balanced binary tree
    - \( \mathcal{O}( \log(n) ) \) time
  - Degenerate tree
    - \( \mathcal{O}( n ) \) time
    - Like searching linked list / unsorted array

- **Requires**
  - Ability to **compare** key values
Binary Search Tree Construction

How to build & maintain binary trees?
- Insertion
- Deletion

Maintain key property (invariant)
- Smaller values in left subtree
- Larger values in right subtree
Binary Search Tree – Insertion

Algorithm

1. Perform search for value X
2. Search will end at node Y (if X not in tree)
3. If X < Y, insert new leaf X as new left subtree for Y
4. If X > Y, insert new leaf X as new right subtree for Y

Observations

- O( log(n) ) operation for balanced tree
- Insertions may unbalance tree
Example Insertion

- Insert (20)

- 10
  - 5
    - 2
    - 25
      - 20
    - 30
  - 45

20 > 10, right
20 < 30, left
20 < 25, left
Insert 20 on left
Binary Search Tree – Deletion

Algorithm

1. Perform search for value X
2. If X is a leaf, delete X
3. Else // must delete internal node
   a) Replace with largest value Y on left subtree
      OR smallest value Z on right subtree
   b) Delete replacement value (Y or Z) from subtree

Observation

- $O(\log(n))$ operation for balanced tree
- Deletions may unbalance tree
Example Deletion (Leaf)

Delete (25)

25 > 10, right
25 < 30, left
25 = 25, delete
Example Deletion (Internal Node)

Delete (10)

Replacing 10 with largest value in left subtree
Replacing 5 with largest value in left subtree
Deleting leaf
Example Deletion (Internal Node)

Delete (10)

Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree
Building Maps w/ Search Trees

- Binary Search Trees often used to implement maps
  - Each non-empty node contains
    - Key
    - Value
    - Left and right child

- Need to be able to compare keys
  - Generic type <K extends Comparable<K>>
    - Denotes any type K that can be compared to K’s
BST (Binary Search Tree) Implementation

- Implementing Tree using traditional approach
- Based on the BST definition below let’s see how to implement typical BST Operations (constructor, add, print, find, isEmpty, isFull, size, height, etc.)

```java
public class BinarySearchTree <K extends Comparable<K>, V> {
    private class Node {
        private K key;
        private V data;
        private Node left, right;
        public Node(K key, V data) {
            this.key = key;
            this.data = data;
        }
    }
    private Node root;
}
```

See code distribution BinaryTreeCode.zip
BST Testing

How can we test the correctness of BST Methods?

What is the best approach?
Polymorphic Binary Search Trees

- Second approach to implement BST
- What do we mean by polymorphic?
- Implement two subtypes of Tree
  1. EmptyTree
  2. NonEmptyTree
- Use EmptyTree to represent the empty tree
  - Rather than null
- Invoke methods on tree nodes
  - Without checking for null (IMPORTANT!)
Polymorphic Binary Tree Implementation

Interface Tree {
    Tree insert ( Value data1 ) { ... }
}

Class EmptyTree implements Tree {
    Tree insert ( Value data1 ) { ... }
}

Class NonEmptyTree implements Tree {
    Value data;
    Tree left, right; // Either Empty or NonEmpty
    Tree insert ( Value data1 ) { ... }
}
Class Node {
    Node left, right;
}

Class EmptyTree {}{
    Tree left, right;
}

Class NonEmptyTree {
    Tree left, right;
}

Node X {
    left = Y;
    right = Z;
}

Node Y {
    left = null;
    right = null;
}

Node Z {
    left = null;
    right = W;
}

Node W {
    left = ET;
    right = null;
}

NonEmptyTree X {
    left = Y;
    right = Z;
}

NonEmptyTree Y {
    left = ET;
    right = ET;
}

NonEmptyTree Z {
    left = ET;
    right = W;
}

NonEmptyTree W {
    left = ET;
    right = ET;
}

EmptyTree { }
Singleton Design Pattern

Definition

One instance of a class or value accessible globally

Where to use & benefits

Ensure unique instance by defining class final
Access to the instance only via methods provided

EmptyTree class will be a singleton class
public final class MySingleton {
    // declare the unique instance of the class
    private static MySingleton uniq = new MySingleton();
    // private constructor only accessed from this class
    private MySingleton() { ... }
    // return reference to unique instance of class
    public static MySingleton getInstance() {
        return uniq;
    }
}
Using Singleton EmptyTree

Class Node {
    Node left, right;
}

Class EmptyTree {}
Class NonEmptyTree {
    Tree left, right;
}

Node X {
    left = Y;
    right = Z;
}

Node Y {
    left = null;
    right = null;
}

Node Z {
    left = null;
    right = W;
}

Node W {
    left = null;
    right = null;
}

NonEmptyTree X {
    left = Y;
    right = Z;
}

NonEmptyTree Y {
    left = ET;
    right = ET;
}

NonEmptyTree Z {
    left = ET;
    right = ET;
}

NonEmptyTree W {
    left = ET;
    right = ET;
}

EmptyTree ET { }
Let’s see a polymorphic list implementation
See code distribution PolymorphicListCode.zip