CMSC 132:
OBJECT-ORIENTED PROGRAMMING II

Trees & Binary Search Trees

Department of Computer Science
University of Maryland, College Park
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 descendants
  • Subtree ⇒ portion of tree that is a tree by itself
    ⇒ a node and its descendants

siblings

subtree
Trees

- **Depth** → Distance from the node to the root of the tree
  - Depth of the root is 0
  - Depth of a node is 1 + depth of its parent

- **Level**
  - The level of a node is its depth (e.g., level of root node is 0)
  - All the nodes of a tree with the same depth

- **Height** → Number of edges on the **longest** downward path from the root to a leaf node
  - A tree with one node has a height of 0

```
Height = 3
```
Binary Trees

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

- Often we want to
  - Find all nodes in tree
  - Determine their relationship
- Can do this by
  - Walking through the tree in a prescribed order
  - Visiting the nodes as they are encountered
- Process is called tree traversal
Tree Traversal

- **Goal**
  - Visit every node in binary tree
- **Approaches**
  - **Breadth first** ⇒ closer nodes first
  - **Depth first**
    - Preorder ⇒ *parent*, left child, right child
    - Inorder ⇒ left child, *parent*, right child
    - Postorder ⇒ left child, right child, *parent*

NOTE: left visited before right
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

Big O – O(n)
Tree Traversal Examples

- Breadth-first
  - $+ \times / 2 \ 3 \ 8 \ 4$
- Pre-order (prefix)
  - $+ \times 2 \ 3 / 8 \ 4$
- In-order (infix)
  - $2 \times 3 + 8 / 4$
- Post-order (postfix)
  - $2 \ 3 \times 8 \ 4 / +$
Binary Tree Implementation

- **Choice #1:** Using a class to represent a Node
  
  ```java
  Class Node {
    KeyType key;
    Node left, right;  // null represents empty tree
  }
  
  Node root = null; // Empty Tree
  ``

- **Choice #2:** Using a Polymorphic Binary Tree
  
  - An empty tree is represented using an object
Types of Binary Trees

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

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

- Key property
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
- Example
  - \( Y > X \)
  - \( Y < Z \)
Binary Search Trees

• Examples

Binary search trees

Non-binary search tree
Tree Traversal Examples

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

Sorted order!

Binary search tree
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
    - $O(\log(n))$ time
  - Degenerate tree
    - $O(n)$ time
    - Like searching linked list/unsorted array

- **Traversal**
  - $O(n)$

- **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**
  
  If tree is empty, just add the entry (which becomes root)

  else

  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 the tree
  - Value will be added a new leaf
  - Order of insertion of values determines the tree shape
Example Insertion

• Insert (20)

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: LectureBinarySearchTreeCode.zip
BST (Duplicate Keys)

- You can handle duplicate keys by arbitrarily placing duplicates of an entry in the entry’s right subtree.
- Updated BST definition
  - Data in a node is greater than the data in the node’s left subtree.
  - Data in a node is less than or equal to the data in the node’s right subtree.
BST Testing

• How can we test the correctness of BST Methods?
• What is the best approach?
Binary Tree Visualizer