Trees

Week 6

Required Activities

- Check Announcements regularly (every 2-3 days)
- Read DSA book: Chapter 8
- Review supplemental materials – BST pseudo code
- Start working on **Assignment 3 due 12/21** (last day before break)

Tree

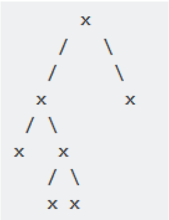
- Nonlinear data structure that represents hierarchical nature of the data (e.g. family tree)

- **Tree Terminology**
  - **root of tree –** node with no parents. There is only one root per tree
  - **edge** – link from parent node to child node
  - **leaf** – node with no children
  - **siblings** – node that have the same parents
  - **depth of a node** – length of the path from the root to the node (count edges)
  - **level** – all nodes at the same depth of the tree at a level. Root node is at level 0, its children at level 1, and so forth
  - **height of node** – length of path from node to the deepest node
  - **height of tree** == depth of tree – length of path from root to deepest node in tree. Tree with root only has height 0

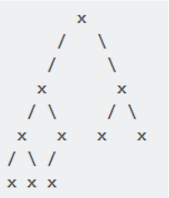
Binary Tree

Empty tree (null) or where each node can have **up to 2** children (left and right)

- **Strict Binary Tree** – each node has exactly two children or no children



- **Complete Binary Tree –** completely filled, with the possible exception of the bottom level. The bottom level is filled from left to right.



- **Full Binary Tree** - each node has exactly two children and all leaf nodes are at same level

Binary Tree operations

- **Insertion** – node inserted into tree
- **Deletion** – node is removed from tree
- **Traversal** – visit each node in a tree recursively; time and space complexity O(n)
  - **pre-order** – visit root, traverse left child sub-tree, traverse right child sub-tree
  - **in-order** - traverse left child sub-tree, visit root, traverse right child sub-tree
  - **post-order** – traverse left child sub-tree, traverse right child sub-tree, visit root
  - **depth-first order** – visit node farthest from root which is a child of node we already visited (e.g. pre-order)
  - **breath-first order** – visit node closest to root that has not yet visited (level order)

Binary Search Tree (BST)

- Empty tree (null) or where each node has a key which:
  - Left subtree of node only contains keys less than node’s key
  - Right subtree of node only contains keys greater than node’s key
  - Both left and right subtree are binary search trees

- Regular binary tree search time worst case is complexity O(n) as need to check each node; BST worst case time is improved to **O(log n)**

- **BST Operations (using recursion)**: O(n) for worst case time and space complexity
  - **Find element** – root then either right or left based on key
  - **Find minimum element** – left-most node that does not have a child
  - **Find maximum element** - right-most node that does not have a child
  - **Insert element** – use find where the element should be (same as find logic) and insert there
  - **Delete element –** find element to delete then If element is leaf it can be deleted; if has single child then copy child to node and delete child; if has two children, get smallest in right subtree (inorder successor) and set as data, delete inorder successor, and make right child null

- In-order traversal produces sorted list

BST Implementation

Using linked list data structure:

Node

    int data
    Node left
    Node right
    // operations to access and set values Node

BST

    Node root
    // operations to insert, delete, find, etc. data in the tree

To use BST (actual syntax will vary based on language):

- create an instance of this data structure (call constructor) ➔ **BST tree** or **tree = BST()**
- Call methods for that instance: ➔ **tree.insert(5)**

AVL (Adelson-Velskii and Landis) Tree

self-balancing binary search tree where heights of the two sub-trees of any node differ by at most one

- basic operations (lookup, insertion, deletion) take O(log n) time for average and worst case
- insertion and deletion may require tree to be rebalanced
- space complexity is O(n) for average and worst case

Heap

- **Heap** – complete binary tree where the value of node is greater than (or less than) than its children for all nodes

- **Min heap –** value of node is less than or equal to values of its children

- **Max heap** - value of node is greater than or equal to values of its children

- **Binary heap** – each node may have up to two children

Heap Sort (Ch 4)

- In-place comparison-based sort that uses heap structure instead of linear-time search:
  - Adjust array elements to construct a max heap (parent greater than its children)
  - Repeatedly
    - swap first value (root of heap) with last value
    - decrease range of list (sorted and unsorted partition)
    - restore heap structure
    - Stops when range of list is one element
- Best, average, and worst case of time complexity is O(n log n) and space complexity is O(1)
- Typically slower than well-implemented quicksort
- Typically uses array and structured as complete binary tree
- Not efficient for small n due to overhead of initial heap creation

```
      9
    /   \
   8     7
  / \   /
 1   5 3
```

8 5 7 1 9 3 make into heap   => 9 8 7 1 5 3
9 8 7 1 5 3 swap first and last unsorted =>   9 8 7 1 5 3
3 8 7 1 5 9 make heap again for unsorted => 8 7 5 1 3 9
8 7 5 1 3 9 swap first and last unsorted =>   8 7 5 1 3 9
3 7 5 1 8 9 make heap again for unsorted => 7 5 3 1 8 9
7 5 3 1 8 9 swap first and last unsorted =>   7 5 3 1 8 9
…
repeat until unsorted portion is one element

Recursion

Factorial => n! e.g. n=5 => 5x4x3x2x1

```
public long factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

BST pre-order traversal

```
public void printPreOrder()  {
      printPreOrder(root);
}

private void printPreOrder(Node node) {

    if (node == null)
        return;

    System.out.print(node.key + " ");        /* visit root: Print data of node */
    printPreorder(node.left);                /* traverse left subtree */
    printPreorder(node.right);               /* traverse right subtree */

}
```

Questions ?

- Post in the discussions
- Send email to RMcFadden@HarrisburgU.edu
- Respond usually within 48hours