LABORATORY 9 : Trees, Binary Trees, Binary Search Trees

OBJECTIVES

- to understand Trees, Binary Trees and Binary Search Trees
- to understand basic binary search tree operations
- to explore options in binary search tree implementation
- to see sample binary search tree applications

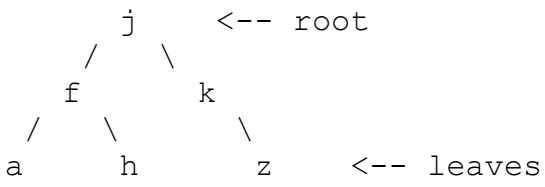
BACKGROUND

(ref: http://www.cs.bu.edu/teaching/c/tree/bst/)

1. Abstract idea of a tree:

   Here, we'll consider elements that each have a *key* (that identifies the element) and a *value* (that is the data for an element), however, we'll ignore the *value* part for now.

   Here is an example of a tree whose keys are letters:



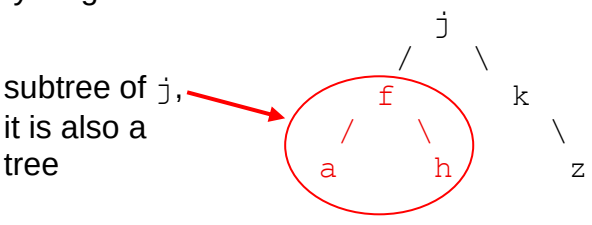
2. Tree Vocabulary

   The element at the top of the tree is called the *root*. The elements that are directly under an element are called its *children*. The element directly above something is called its *parent*. For example, `a` is a child of **`f`** and **`f`** is the parent of **`a`**. Finally, elements with no children are called *leaves*.

3. Recursive Data Structure

   A tree can be viewed as a *recursive data structure*. Why? Remember that *recursive* means that something is defined in terms of *itself*. Here, this means that trees are made up of *subtrees*.

   For example, let's look at our tree of letters and examine the part starting at `f` and everything under it.



4. Binary Trees

   We can talk about trees where the number of children that any element has is *limited*. In the tree above, no element has more than two children. A tree whose elements have *at most* two children is called a *binary* tree.

Since each element in a binary tree can have only 2 children, we typically name them the *left* and *right* child.

5. Ordering of Tree?

   The structure of a tree is *hierarchical*, meaning that things are ordered *above* or *below* other things. For example, the army is hierarchical, with generals above colonels, and colonels above lieutenants, etc.

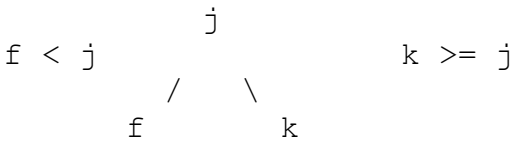
   Despite the hierarchical order of the structure of the tree, the order enforced on elements in the tree will depend on how we use the tree.

6. Binary Search Tree

   The tree that we presented before actually has an enforced order.

   First, remember that the letters in the tree are *keys* for the elements held in the tree.

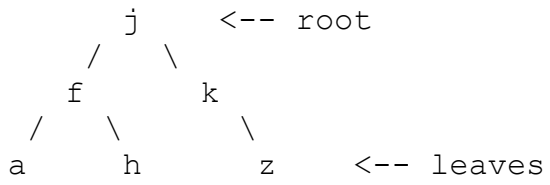
   For any element, keys to its left are *less than* its own key. Also, keys to its right are *greater than or equal to* its own key; e.g.



   Note that this must be true for *every* element – every subtree must also be a binary search tree.

   A tree with this *ordering* property AND that is binary is called a *binary search tree*. Why? Because in order to search for an element (with a specific key) in such a tree, you only need to make a series of binary (i.e., go *left* or *right*) decisions.

   For example, to find `h` starting from the tree's root.



   1. Key **`h`** is *less than* **`j`**, so go left.
   2. Key **`h`** is *greater than* **`f`**, so go right.
   3. Found **`h`** ☺

7. Binary Search Tree ADT

   Data elements:
   - A collection of nodes where each node contain a data item and links to left and right children. Each node can have at most two children.

   Basic operations:

   - Add new element into binary search tree

     ```
     void insert(Object element)
     ```

- Remove an element from binary search tree

```
Object remove(Object element)
```

- Check if binary tree is empty

```
boolean is_empty()
```

- Find and object in binary search tree

```
Object find(Object element)
```

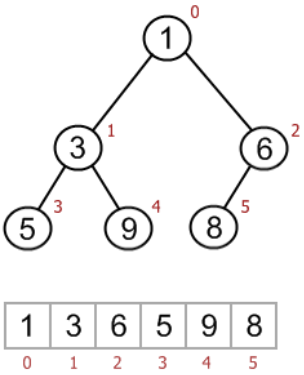
- Traverse binary tree – visit all nodes in inorder, preorder, postorder fashions

```
String inorder()
String preorder()
String postorder()
```

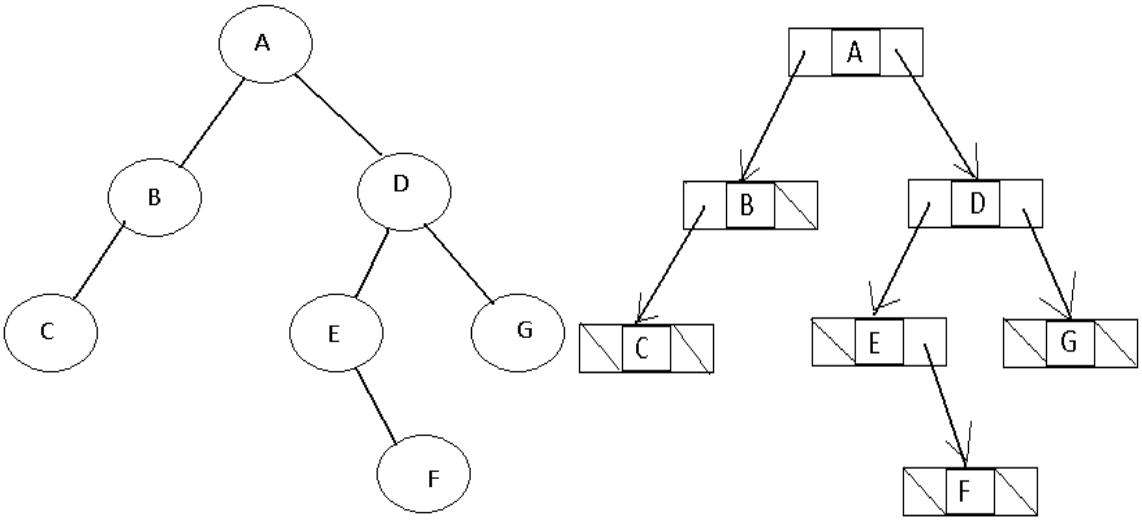
- Print tree

```
void print()
```

8. Binary Tree and Binary Search Tree implementation
   - Array based : each cell contains element and array index corresponds to node position in a binary tree.



   - Linked list based : each node contains element and two references to left and right children.



LABORATORY 9: In-lab

1. Implement a linked list based `BST` class (`BST.py`) according to the BST ADT in the *Background* section. Fill in all `# your code here`'s in the following code.

```
class Node:
    def __init__(self, username_in, password_in):
        # your code here
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None

    def insert(self, new_node):
        if (self.root is None):
            self.root = new_node
        else:
            self.__insert_node(self.root, new_node)

    def __insert_node(self, current_node, new_node):
        if (new_node.username <= current_node.username):
            if (current_node.left is not None):
                self.__insert_node(current_node.left, new_node)
            else:
                current_node.left = new_node
        elif (new_node.username > current_node.username):
            if (current_node.right is not None):
                self.__insert_node(current_node.right, new_node)
            else:
                current_node.right = new_node

    def find(self, username):
        return self.__find_node(self.root, username)

    def __find_node(self, current_node, username):
        # your code here

    def remove(self, username):
        # your code here

    def is_empty(self):
        # your code here

    def preorder(self):
        # your code here

    def inorder(self):
        # your code here

    def postorder(self):
        # your code here

    def print(self):
        # your code here
```

- Note that when removing a node that has two children, the removed node is replaced with its inorder successor. You may want to write a method `find_min()` to (recursively) find the inorder successor of a given node.

- You may use the algorithm below as a guideline for the method `print()` to print the tree sideway.

```
Algorithm print_subtree(tree, level)
    if(tree is not empty)
        print_subtree(right_subtree, level + 1)
        print space 3*level times
        print item at root and endOfLine
        print_subtree(left_subtree, level + 1)
```

2. Test all the methods. Make sure that they work correctly.

LABORATORY 9: Post-lab

1. Validating computer logins

   In the login process, the system checks username and password to verify that a user is a legitimate user. Because this user validation must be done many times each day, it is necessary to structure this information in such a way that it can be searched rapidly. Moreover, this must be a dynamic structure because new users are regularly added to the system.

   - Is BST a good candidate for this computer login validation program? Why?

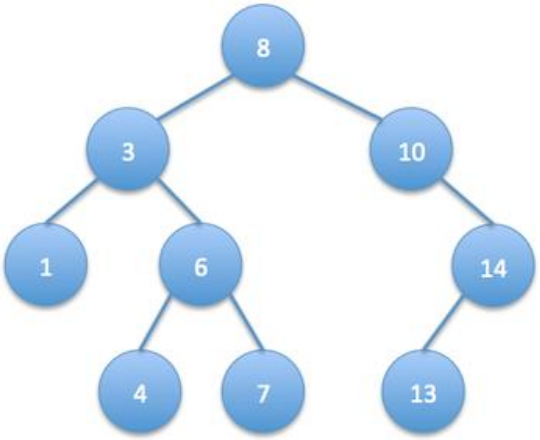
   Suppose we are to implement it with BST.

   - Write a program to validate computer logins using your BST class.
     - Valid username and password pairs are stored in the file (`users7.txt` and `users1000.txt`)
     - Three unsuccessful attempts are permitted. User will be revoked (removed from the system) after the 3rd unsuccessful attempt.
     - To read input from text file, consider the following options.

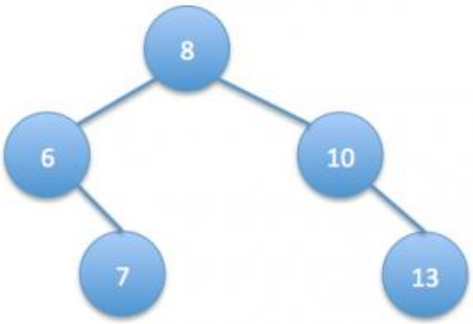
       1.
       ```
       with open("accounts.txt") as f:
        content = f.readlines()
       ```
       2.
       ```
       file = open("accounts.txt", "r")
       for line in file:
           content.append(line)
       ```

2. Write a recursive method to count number of nodes in a BST.
3. Write a (recursive?) method that trims the BST such that all the numbers in the result BST are between the given two values, min and max, inclusively.

   Example: a BST of integers :



`trim(5, 13)` will result in



Submission:

In-lab : Wednesday Oct 28 by 12:00pm

Post-lab : Tuesday Nov 3 at 2:30pm

You are to demonstrate your algorithm, test plan and program. Prepare to answer some questions individually.

Sample BST expected output

```
BST of Integers
insert 8, 3, 1, 6, 4, 7, 10, 14, 13 into an empty BST gives
      14
         13
   10
8
         7
      6
         4
   3
      1

BST after deleting a node : 8
   14
      13
10
         7
      6
         4
   3
      1

BST after trimming : min 5, max 13
   13
10
      7
   6

BST of Strings
insert “i”, “love”, “you”, “pca”, “is”, “easy”, “happy”, “day”, “kmitl” into an
empty BST gives
      you
   love
         kmitl
      is
i
         happy
      easy
   pca
      day

BST after deleting a node : pca
      you
   love
         kmitl
      is
i
      happy
   easy
      day
BST after trimming : min eee, max love
   love
         kmitl
      is
i
   happy

BST of Users (input from users7.txt)
      tisha:tisha111#
            sandy:sandy123#
         prim:prim123#
      panya:panya123#
hello:hello111#
      gun:gun123#
   dang:dang111#
```