George Mason University
INFS 519 – Data Structures
Fall 2018
Programming Assignment 3 – Binary Search Trees

Part 1) A Tree Based Address Book
DUE DATE: Nov 9th, midnight.

In this project you will replace the linked list from Project 1 with a binary search tree as the basis for your address book. You will also save your address book in a text file. The program stores pairs of values: a *key* and a *value*. The key must be unique and is used to identify the pair. There are three operations:

- `value lookup(key)` searches the address book for a pair with the given key and returns the corresponding value.
- `boolean insert(key, value)` stores the key/value pair in the address book.
- `boolean delete(key)` removes the key/value pair with given key from the address book.

Your program will appear the same as Project 1 to the user *except* that it will start by reading data from an input file before displaying the main menu to the user. Each pair of lines, from the input file, will be made an entry in the address book. The name of the input file must be provided as an argument to the main method of the program. The main menu in this program must look like the following:

```
Add a contact (a)
Look up a contact (l)
Update address (u)
Delete a contact (d)
Display all contacts (a)
Save and exit (q)

->
```

Note that some of the menu options of Project 1 are not needed for this project. The focus in this project in only on the address book functions: add entries to the address book, update an entry, delete an entry and display all entries of the address book. The "display all" option must display a **sorted** list of entries (an ascending list sorted by names), and must show height information for each tree node. Note that in this project, you will not be implementing the sending of messages function. Also note that the program menu now includes a new option "Save and exit", which will ask the user for the name of a file in which to write the address book, open the file and write the entire address book to the file. This operation will be very similar to the "display all" operation. You will use traversals to implement "display all" and "save".

Internals
Addressbook will now be implemented as a binary search tree. Each node will have two String fields (for the name and address) along with an *int height* field and an *int size* field. Public methods *insert*, *lookUp*, *delete*, and *update* will use binary search tree operations (which you will write). You may choose to store the tree internally using arrays or using a linked structure. You may NOT use any built in Java Collections Framework classes in your program (e.g. no **`DynamicArray`**, **`LinkedList`**, **`HashSet`**, etc.). Your code may be implemented either recursively, iteratively, or as a combination of the two.

Again, you will write (among other classes) a main class **Table** which will store entries comprised of (key/value) pairs of Strings. This class will have the following public methods (as in Project 1):

- `public boolean insert(String key, String value):` Inserts a new entry to the table. If an entry already exists with the given key value make no insertion but return `false`.
- `public String lookUp(String key):` Looks up the entry with the given key and returns the associated value. If no entry is found `null` is returned.
- `public boolean deleteContact(String key):` Deletes the entry with the given key. If no entry is found returns `false`.
- `public boolean update(String key, String newValue):` Replaces the old value associated with with the given key with the newValue string.
- `public int displayAll()`: Displays Name/Address for each table entry, the list of entries is sorted by the keys. This method must also additionally display the *height* of each entry node the address book tree structure. Returns total entry count (size at root).
- `public void save()`: reads the name of a text output file, and will write a list of the table entries to an the output file. Note that this function must implement **a pre-order** traversal of the tree structure and must write the address book entries to the output file in that order.

To turn in: As before, you will turn in an executable jar file along with well documented source listings via Blackboard.

Rubric for this Project:

- BST structure (can be array based or linked list array): 20 points.
- Read from input file and properly build a BST structure: 20 points.
- Tree walks and functions: 30.
- Proper menu functions: 20.
- Professional looking code/comments: 10 points

Part Two) Extra Credit

DUE DATE: Thursday Nov 15th, midnight.

In this second part, you will convert the BST from part 1 into an AVL tree as the basis for your address book. This part of the project is only optional and is provided to help you gain some bonus points to your overall projects grades (5%). Note that you must complete and **submit** part 1 in this project document before you start working on part 2. , a separate submission link will be provided for this extra credit task.

For this part, your program will appear the same as Part 1 to the user *except* that you will now be implementing AVL methods to maintain the BST structure balanced. Again, you will use an input file for reading in address book entries (key/value), however, you will be building an AVL tree (instead of a BST). Your program must also implement AVL methods for *add*, *update*, and *delete* function in the main menu. The "display all" operation will additionally display the *balance factor* of each node within the tree.

Internals (AVL implementation)
Again, you will write (among other classes) a class Table that will store entries comprised of (key/value) pairs of Strings. This class will have the same public methods as in part 1, and will include (at a minimum) the following additional private methods:

- `private void rebalance(Node n)`
- `private Node rotateLeft(Node n)`
- `private Node rotateRight(Node n)`
- `private Node rotateLeftThenRight(Node n)`
- `private Node rotateRightThenLeft(Node n)`

Table will now be implemented as an AVL tree. Your code may be implemented either recursively, iteratively, or as a combination of the two.