BST Social Security Numbers

Taha Quraishi

Programming Assignment #3
Data Structures and Algorithms
CS 146 Spring 2019
Professor Mike Wu

Design and Implementation

To provide a brief overview of the program, a random number generator generates 20 Social Security numbers for 20 retiring people. After linking a SSN to a person for all 20 people, a Binary Search Tree was built. The operations implemented are insert and delete; one inserts a name to the BST based on his/her SSN, and the other removes a person. Moreover, the program also allows the user to print the BST using in order traversal.

Since a Binary Search Tree revolves around nodes, creating the NodeBST class was the first step. Every node has a parent, left, and right node, thus 3 fields were implemented to portray this property. For this assignment, a node also contains a SSN (int) and a name (String) attached to it. As expected, setter methods to set the parent, left, or right node were also programmed, as well as getter methods to retrieve these particular nodes, the SSN, and the person's name.

Before programming the SocialSecurityNumber class, it was important to understand the schema; a Social Security number (SSN) has 3 parts to it. The first set of 3 digits are known as the area number, the second set of 2 digits is the group number, and the final set of four digits is the serial number. The area represents what location a person lived in when he or she applied for the Social Security number; these 3 digits can be from 1 to 999. The group number determines when a block of Social Security numbers were published; these 3 digits can be from 1 to 99. Series, which are the final 4 digits, help ensure that the Social Security numbers were unique; it can be anywhere from 1 to 9999, and these numbers were based off the group's numerical sequence. In all, this shows that a Social Security number has 9 total digits and is formatted as: AAA-GG-SSSS. For this homework assignment, we assumed that all Social Security numbers in existence were issued randomly, even though this did not happen until 2011. The

SocialSecurityNumber class has 5 private variables; 3 of the String variables are *area*, *group*, and *serial*. A private variable for storing the formatted SSN was also declared, including a Random object which would be used in generating all 3 parts in a SSN. In the constructor, the Random object is initialized and the formatted SSN is stored by calling *generateAreaNumber()*, *generateGroupNumber()*, and *generateSerialNumber()*, with dashes after the area number and group number. In g*enerateAreaNumber()*, a random int is created between 1 and 999 using the Random object. Since there was a chance that the generated area number may be less than 3 digits, it was important to add leading 0's. Using the formula "(number / value place) % 10", it was possible to get the digit at a certain value place. For example, if the number was 2 and the digit at the 10's place was wanted, dividing 2 by 10 and modding it by 10 gave the result 0, due to Integer division in Java. I applied this method by retrieving the digit at all 3 value places (1's place, 10's place, and 100's place) in the generated area number, concatenating them together, and returning this String; a String was returned since parsing the number to an int would have removed all leading 0's. This similar method was applied to *generateGroupNumber()* and *generateSerialNumber()*, but for 2 and 4 digits respectively. 4 getter methods were also present in this class for retrieving the area, group, serial, and formatted SSN as String objects. The final and most crucial method was returning the SSN as an int. Using *getArea()*, *getGroup()*, and *getSerial()*, I concatenated them together, parsed it into an int, and returned it. To provide clarification in this method, let us assume that area is 042, group is 15, and serial is 0973. The concatenated String becomes 042150973, but converting to an int causes the number to become 42150973.

Creating the BinarySearchTree was the penultimate step before creating the user interface. The only field in this class is the root node, which is initialized as null in the constructor. Since one of the functions in the instructions was to implement an insert method, *treeInsert(NodeBST z)* was first implemented. The method starts off by setting node y to null, and the x node to the root. In the while loop that runs only if x isn't null, it checks if z's key is less than x's key. If it is, it sets the x node to its left node. Otherwise, it sets it to the right node. After the loop, z's parent is set to y. In another check, if y is null, the root becomes the z node. Moreover, if z's key is less than y's key, y's left node is set to z, otherwise y's right node is set to z. To create the delete method, the *transplant(NodeBST u, NodeBST y)* method needed to be created. It starts off by checking if u's parent is null; if it is, the root becomes the v node. In another check, if u is equal to its parent's left node, u's parent's left node is set to v. Otherwise, its parent's right node is set to v. Finally, if v is not null, v's parent is set to whatever node u's parent is. The *treeMinimum(NodeBST x)* was also created to make the delete method easier. In this method, a loop runs only if x's left node is not null. While this condition is true, x becomes its left node. After the loop ends, the x node is returned. The *treeDelete(NodeBST z)* was implemented using the methods mentioned above; it starts off by checking if z's left node is null. If it is, the transplant method is called on z and z's right node. If, z's right node is null, the transplant method is called on z and z's left node. Otherwise, node y is set to the node returned using the minimum method called on z's right node. If y's parent is not z, the transplant method is called on y and y's right node. Then, y's right node is set to z's right node. Additionally, y's left node's parent is set to y. As expected, the *getRoot()* method returns the root of the BinarySearchTree. The i*nOrderTreeWalk(NodeBST x)* was also implemented to print the BST in

order. In this method, if x is not null, a recursive call is made to x's left method. The SSN and person's name it is linked to is then printed, which is followed by a recursive call on x's right node.

The Simulation class is meant to provide the user an interface. The 20 retiring people are stored in a file named "Names.txt." Using a method called *readFromFile(String fileName)*, it reads every line, stores the name into an ArrayList, and returns it. Additionally, 20 SSNs are generated in *generateSocialSecurityNumbers()*, where an ArrayList of unique SSNs are returned. In the *startUp()* method, the names from the file are stored in the *names* ArrayList containing Strings, and the SSNs are stored in the *SSNs* ArrayList containing ints. Using a loop that goes through every name in the ArrayList, a new node is created with the name and SSN at the current position in the for loop, which is added into the nodes ArrayList. Then, each newly created node is inserted into the BST. The UI provides 4 options: display BST, insert person, remove person, and exit. If the user enters the first option, the BST is printed using in order traversal. If the second option is entered, the *insertPerson()* method is called. This method uses the Scanner object to store the user's name. which is added into the *names* ArrayList. A unique SSN is created for this person, which is also added to an ArrayList, but to *SSNs* this time. A new node is created with the person's name and SSN, which is inserted into the BST. If the user enters the third option, the *removePerson()* method is called. This method starts off by storing the inputted name into a String. Using a loop, the given name is searched in the BST. If it's found, the node containing the given name is deleted from the BST and the *nodes* ArrayList.The fourth option ends the while loop, which exits the program. As expected, the *main(String[] args)* method calls the *startUp()* method, and then the *displayUI()* method.

Classes/Subroutines/Function Calls

SocialSecurityNumber:

Summary: Represents a Social Security Number (SSN), which has 3 parts: area (between 1 and 999), group (between 1 and 99), and serial (between 1 and 9999). A SSN has 9 digits and is in the format, AAA-GG-SSSS. For this simulation, it's assumed that every SSN in existence was generated randomly.

generateAreaNumber(): Generates and stores an area number between 1 and 999. The returned value includes leading 0's if generated area number is less than 3 digits.

generateGroupNumber(): Generates and stores a group number between 1 and 99. The returned value includes leading 0's if generated group number is less than 2 digits.

generateSerialNumber(): Generates and stores a serial number between 1 and 9999. The returned value includes leading 0's if generated serial number is less than 4 digits.

getArea(): Gets the area section in a SSN.

getGroup(): Gets the group section in a SSN.

getSerial(): Gets the serial section in a SSN.

getSSN(): Gets the formatted SSN.

getSSNWithoutFormatting(): Gets the unformatted SSN. Does not include any dashes and leading 0's in area, but includes leading 0's in group and serial.

NodeBST:

Summary: Represents a node in a BST. Every node has a parent, right, and left node, as well as a SSN and a name.

setParent(NodeBST parent): Sets the parent node.

setRight(NodeBST right): Sets the right node.

setLeft(NodeBST left): Sets the left node.

getParent(): Gets the parent node.

getRight(): Gets the right node.

getLeft(): Gets the left node.

getKey(): Gets the SSN of the node.

getName(): Gets the name of the node.

BinarySearchTree:

Summary: A data structure represented as a tree where the left subtree of a node has a key less than its parent node, and the right subtree of a node has key greater than or equal to its parent.

treeInsert(NodeBST z): Inserts the given node into the BST.

inOrderTreeWalk(NodeBST z): Prints the BST using in order traversal.

transplant(NodeBST u, NodeBST v): Replaces a subtree with another.

NodeBST treeMinimum(NodeBST x): Gets the minimum of the BST at node x.

treeDelete(NodeBST z): Deletes the given node from the BST.

getRoot(): Gets the root of the BST.

Simulation:

Summary: Provides a user interface that allows the user to print the BST, insert a person, and remove a person.

main(String[] args): The main method of the program. Calls startUp() and displayUI().

startUp(): Reads the names from "Names.txt", stores them into an ArrayList, generates 20 SSNs, stores them into an ArrayList, creates 20 nodes linking a SSN to a name, and inserts each node into the BST.

displayBST(): Prints out the BST using in order traversal.

insertPerson(): Inserts a person into the BST.

removePerson(): Removes a person from the BST.

displayUI(): Displays a UI providing the the user 4 options: display BST, insert person, remove person, and exit program.

generateSocialSecurityNumbers(): Generates 20 unique SSNs.

readFromFile(String fileName): Reads every line in a file, adds them into an ArrayList, and returns it.

Testing

Displays the UI:

```
20 users with generated SSNs have been inserted into the BST. Enter the first option to view the tree.

 ======================================
|                                      |
|      Binary Search Tree Simulation   |
|                                      |
 ======================================
| Options:                             |
| 1 >> Display BST                     |
| 2 >> Insert Person                   |
| 3 >> Remove Person                   |
| 4 >> Exit                            |
 ======================================

Please enter an option number above:
```

Prints the BST using in order traversal if the first option is called:

```
Please enter an option number above: 1
Ken         047-03-1742
Jacob       062-10-7854
Henry       117-76-4906
Victoria    183-59-9136
Ethan       198-98-9256
Marry       254-60-1738
Daniel      276-41-2671
Lucas       328-68-1608
Matthew     343-21-2210
Mike        503-44-1762
Jane        516-68-0491
Elijah      550-10-7745
Mason       665-02-4521
James       695-62-2235
Oliver      758-35-5838
Michael     808-45-4453
Ella        854-65-6364
Sofia       877-91-8948
Ben         953-48-3127
Jackson     976-03-3412
```

Removes a user from the BST if the third option is called:

```
Please enter an option number above: 3
Enter name: Mason
Mason has been removed from the BST.
```

BST using in order traversal after removal of a person:

```
Please enter an option number above: 1
Ken         047-03-1742
Jacob       062-10-7854
Henry       117-76-4906
Victoria    183-59-9136
Ethan       198-98-9256
Marry       254-60-1738
Daniel      276-41-2671
Lucas       328-68-1608
Matthew     343-21-2210
Mike        503-44-1762
Jane        516-68-0491
Elijah      550-10-7745
James       695-62-2235
Oliver      758-35-5838
Michael     808-45-4453
Ella        854-65-6364
Sofia       877-91-8948
Ben         953-48-3127
Jackson     976-03-3412
```

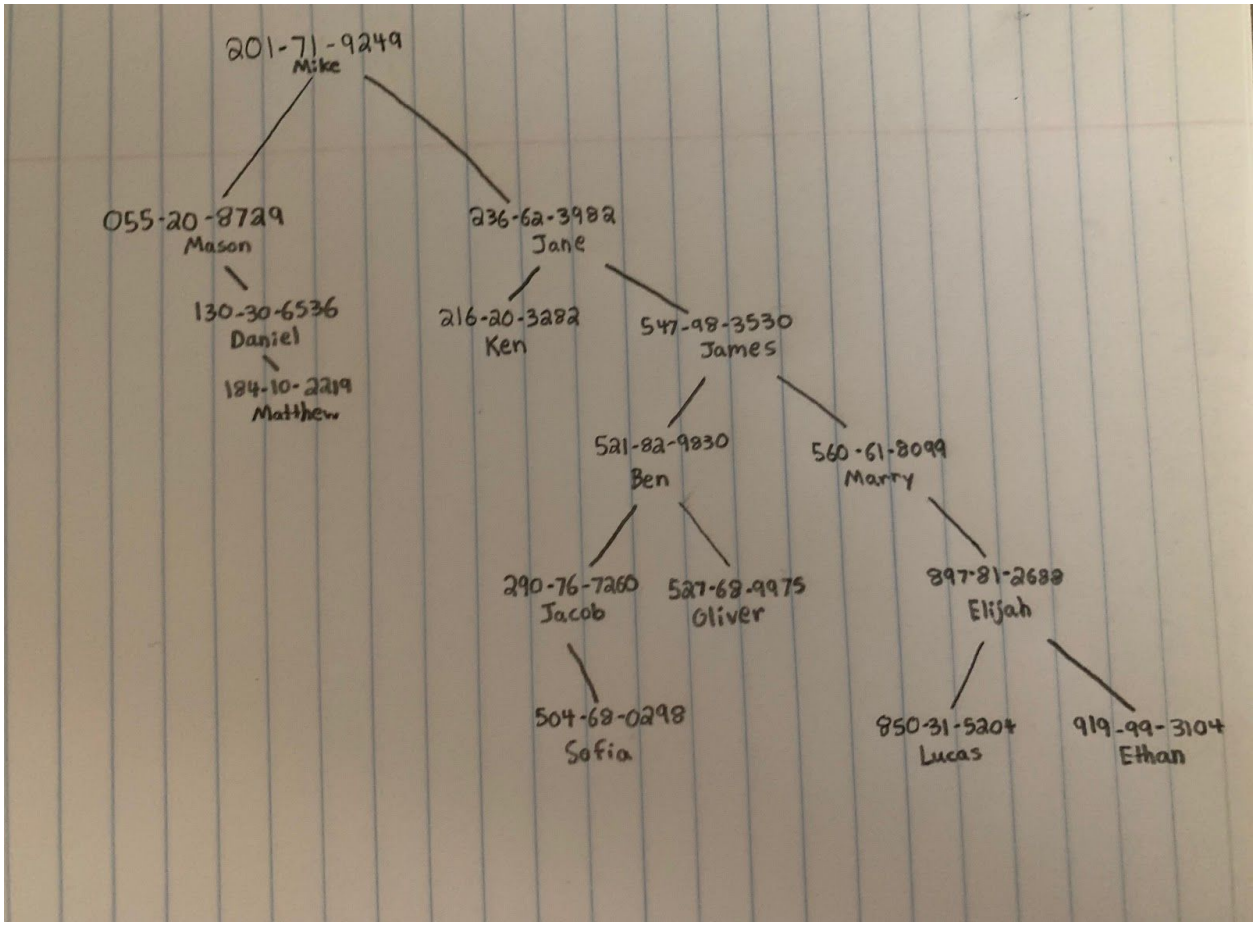
Inserts a person into the BST if the third option is called, a new unique SSN is created:

```
Please enter an option number above: 2
Enter name: Mason
Mason's generated SSN is 539-14-2478. This person has been added into the BST.
```

BST using in order traversal after insertion of a person:

```
Please enter an option number above: 1
Ken         047-03-1742
Jacob       062-10-7854
Henry       117-76-4906
Victoria    183-59-9136
Ethan       198-98-9256
Marry       254-60-1738
Daniel      276-41-2671
Lucas       328-68-1608
Matthew     343-21-2210
Mike        503-44-1762
Jane        516-68-0491
Mason       539-14-2478
Elijah      550-10-7745
James       695-62-2235
Oliver      758-35-5838
Michael     808-45-4453
Ella        854-65-6364
Sofia       877-91-8948
Ben         953-48-3127
Jackson     976-03-3412
```

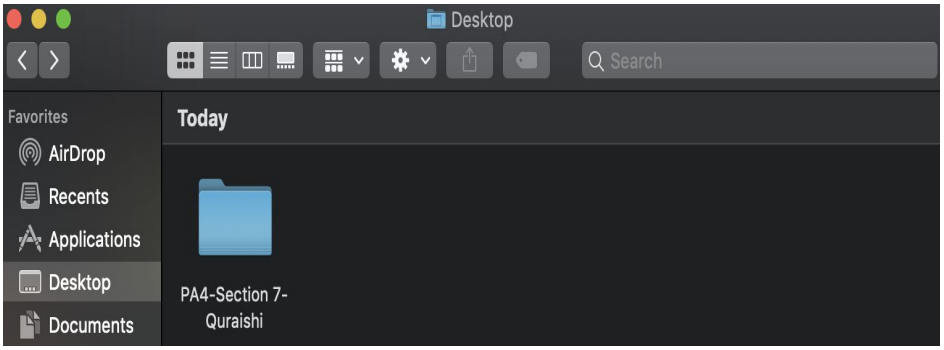
Drawing of a BST (using values different from the screenshots above):



Step by Step

1. In front of you will be a compressed folder named **PA4-Section 7-Quraishi.zip**. Double click this ZIP file to extract the "PA4-Section 7-Quraishi" folder.
2. This folder will contain 4 different files/folders:
    a. CS 146 BST Social Security Numbers by Taha Quraishi.pdf (pdf file)
    b. Java Files (folder)
        i. BinarySearchTree.java
        ii. NodeBST.java
        iii. Simulation.java
        iv. SocialSecurityNumber.java
    c. Text Files (folder)
        i. Names.txt
    d. BSTSocialSecurityNumbers.jar (jar file)
3. The Formal Report talks about: design and implementation, a list of classes/subroutines/function calls and what they do, self-testing screenshots, the step by step procedure, problems encountered, and the lessons learned. It is the document you are reading right now.
4. To view the .java files, double click the folder called "Java Files". All 4 .java files (classes) used in creating the program will be displayed.
5. To view the .txt file, double click the folder called "Text Files".
6. To run the program BSTSocialSecurityNumbers.jar (macOS):
    a. Important: Place "PA4-Section 7-Quraishi" folder in the "Desktop" directory

i. If the folder is not placed in this directory, using the options explained below will not work



b. Open Terminal

c. Type in “java -jar” and press space

d. Drag the .jar file in the file explorer to the terminal and press enter/return

e. The program will begin; follow the user interface

Problems Encountered

Since the pseudocode was provided in the book, errors were minimal when creating the first half of the homework assignment. In the *treeDelete(NodeBST z)* method, my initial implementation produced incorrect output. Upon careful review of the book's pseudocode, I realized that I was missing a line that would set y's left node to z's left node. After adding this line, the delete operation worked as expected. Likewise, my first implementation of the *treeInsert(NodeBST z)* method forgot to set y's right node to z if z's key was not less than y's key.

However, trying to code the Red Black Tree was extremely difficult. The textbook provided all the pseudocode needed to create a BST, but only half of the fixup method, and only one of the the rotate methods were provided in the section explaining RBT. Although I was able to successfully create another node class, but this time with a color, I was not able to implement a working Red Black Tree in time. Up until case 2, I kept on getting NullPointerExceptions, which was caused by right rotate method, which we had to code ourselves. Although the correct output would be displayed for case 1, case 2 and case 3 did not work at all. After giving hours into this, I was unable to debug the fixup and right rotate method.

The instructions required 20 people to be entered into the BST, meaning there would be 20 nodes. My first implementation was manually creating 20 separate nodes, each with a name and SSN. However, I realized that it would be a lot cleaner with a for loop. This for loop goes through every name, creates a node with an SSN and name at the current position in the loop, and inserts it into the Binary Search Tree.

Lessons Learned

From this homework assignment, I learned how to create a Binary Search Tree to use operations such as insertion, deletion, and in order traversal on nodes. Moreover, I implemented a user interface that allows these options to be selected.

Implementing the insertion and deletion method were not straightforward. I learned that several checks are required to have method that guarantees the correct output, such as checking if z's key is less than x's key, or if y does not exist. Understanding the deletion method was far from straight forward. A transplant method was used in order to simply the deletion method by replacing a subtree with another. Likewise, several checks were required in the delete method such as checking if z's left node is null, or its right node is null. Additionally, by implementing a method that prints the BST using in order, I strengthened my skills of one of the 3 ways to traverse through the data structure.

My motivation behind giving my best effort into this programming assignment was due to my love of coding. Everyday, I want to learn new skills, data structures, and algorithms, that I can implement into my personal programs and in the field. I want to be successful in coding interviews; even though this assignment was for extra credit, I give the same amount of effort as if it was not optional. As you may know, my sister graduated from Berkeley this week, which caused me to have an extremely packed schedule, where I had to study for all my other classes, as well as travel to Berkeley for the ceremony. Unfortunately, this caused me to only finish half of the program, leaving out Red Black Trees. However, although the assignment due date will pass, I am going to finish the functional requirement that I skipped. Red Black Trees are in important concept, so I must know how to implement it.