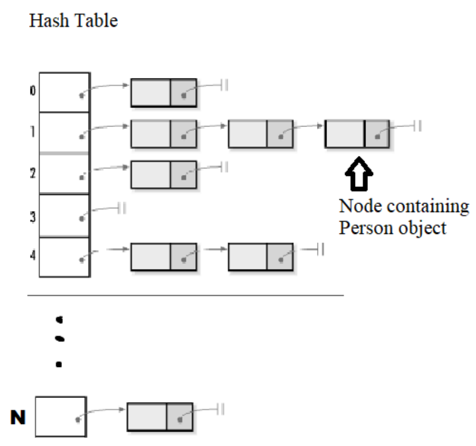
PA3:
Social Media Application

Jia Siang Fung

1. Design and Implementation:

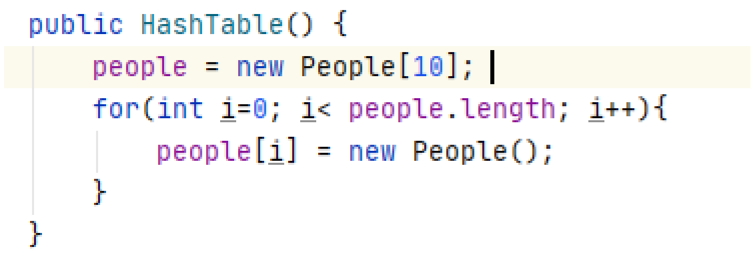
1.1 Hash Table

Design:



The hash table represents a database of Person objects, each having their own string name and a linked list of friends pointing to other Person objects in the hash table. Every Node containing a Person object in our hash table represents a registered user in our social media application. I designed my hash table to be an array of my linked list class 'People' in order to deal with collisions because the hash table has a size of 10, but there are 50 users at the start and even more if we choose to insert more Person objects. Each Person object gets inserted into the hash table based on our hashing function that converts the string name of the Person object into a natural number that hashes to a certain index using the division hashing method.

Implementation:

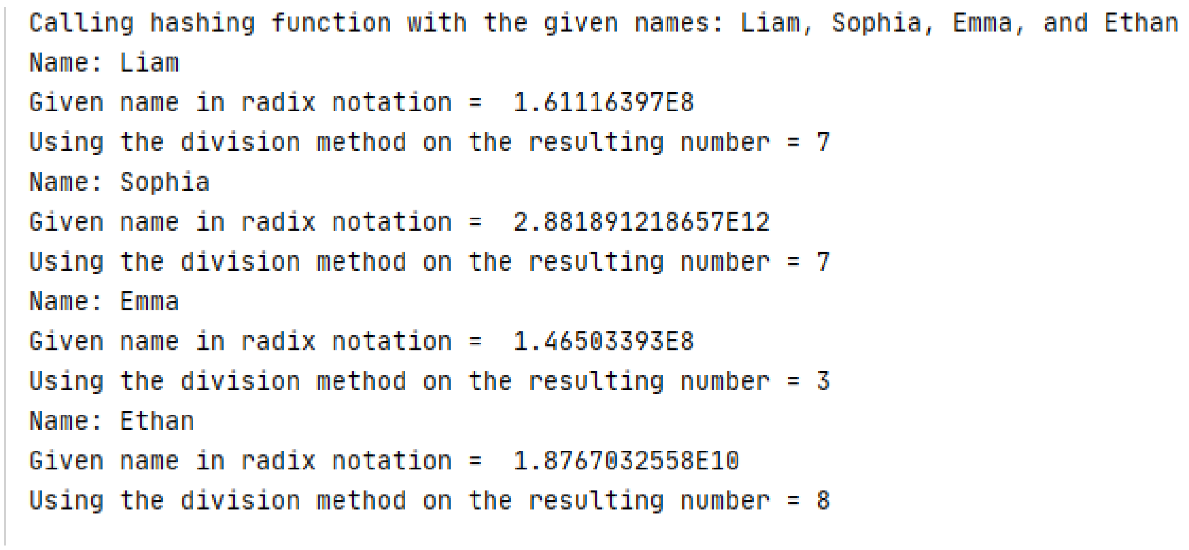


```
public HashTable() {
    people = new People[10];
    for(int i=0; i< people.length; i++){
        people[i] = new People();
    }
}
```

A linked list is created at each index of the hash table, to deal with collisions whenever we insert new Person objects into the hash table.

1.2 Hashing Function

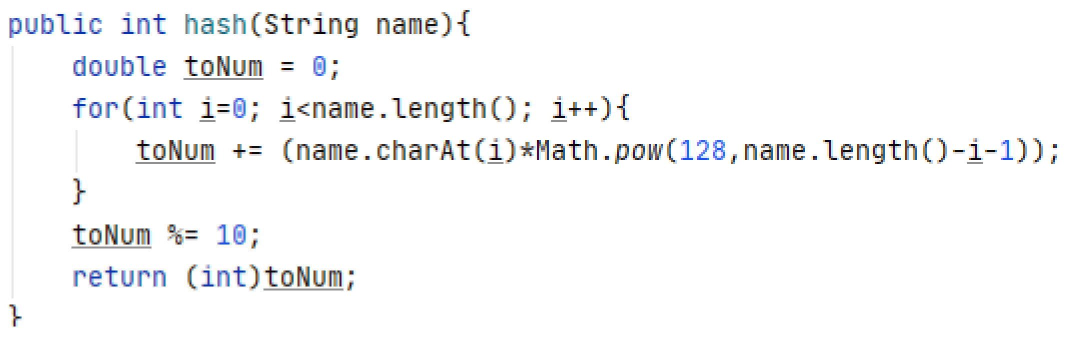
Design:



```
Calling hashing function with the given names: Liam, Sophia, Emma, and Ethan
Name: Liam
Given name in radix notation =  1.61116397E8
Using the division method on the resulting number = 7
Name: Sophia
Given name in radix notation =  2.881891218657E12
Using the division method on the resulting number = 7
Name: Emma
Given name in radix notation =  1.46503393E8
Using the division method on the resulting number = 3
Name: Ethan
Given name in radix notation =  1.8767032558E10
Using the division method on the resulting number = 8
```

The hash function hashes using the radix notation for ASCII code since we're dealing with names in the hash table. The input string name is converted to radix notation using its base 128 ASCII notation, and then we take the mod of it by the size of our hash table, which is 10. The hash function returns a number between 0 to 9 that decides where we insert the new Person object based on the formula in the implementation shown below.

Implementation:



```java
public int hash(String name){
    double toNum = 0;
    for(int i=0; i<name.length(); i++){
        toNum += (name.charAt(i)*Math.pow(128,name.length()-i-1));
    }
    toNum %= 10;
    return (int)toNum;
}
```

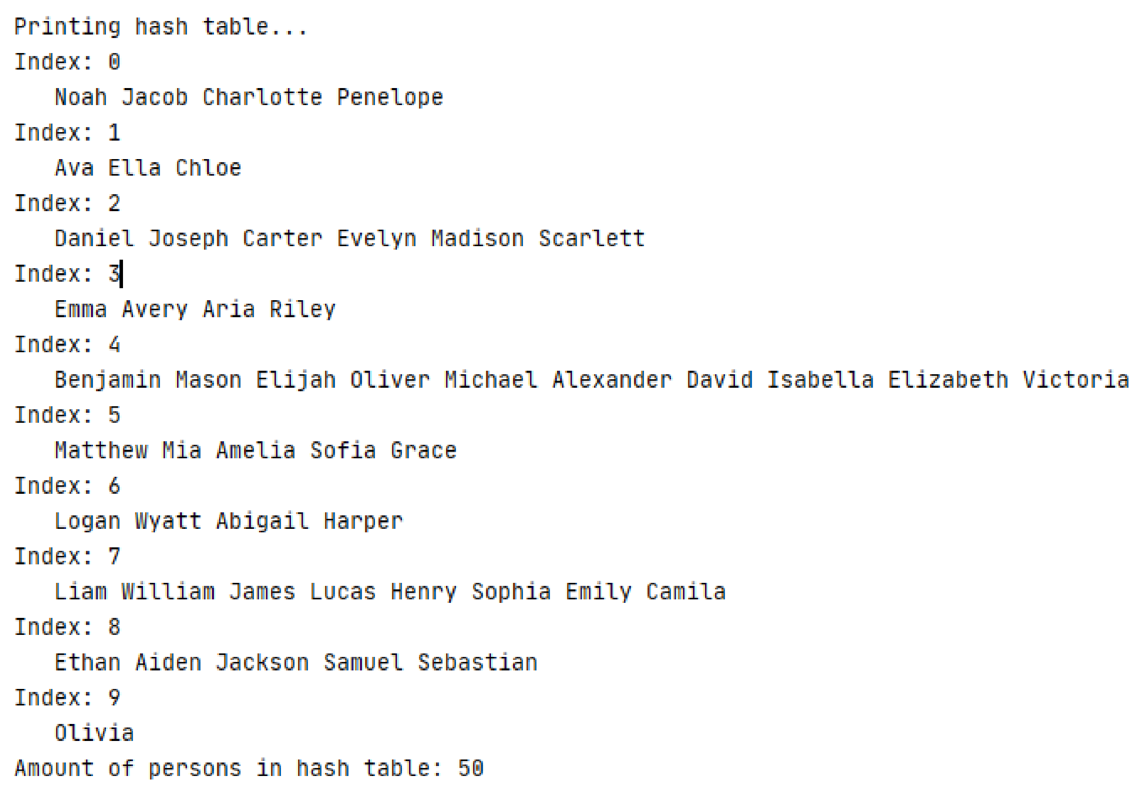
A given string name is turn into a natural number based on radix notation. Our formula for the radix notation is as follows:

hash(name) = ((first digit as ASCII number) *128^(num of digits in name -1) + (second digit as ASCII number)*128^(num of digits in name -2) + … + (last digit as ASCII number)*128^(0)) mod 10,

which gives us a double that is casted as an integer and then returned.

1.3 Chained-Hash-Insert

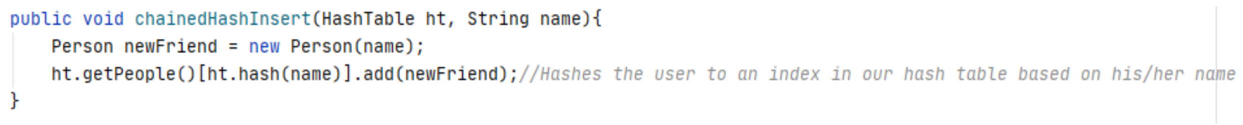
Design:



```
Printing hash table...
Index: 0
   Noah Jacob Charlotte Penelope
Index: 1
   Ava Ella Chloe
Index: 2
   Daniel Joseph Carter Evelyn Madison Scarlett
Index: 3
   Emma Avery Aria Riley
Index: 4
   Benjamin Mason Elijah Oliver Michael Alexander David Isabella Elizabeth Victoria
Index: 5
   Matthew Mia Amelia Sofia Grace
Index: 6
   Logan Wyatt Abigail Harper
Index: 7
   Liam William James Lucas Henry Sophia Emily Camila
Index: 8
   Ethan Aiden Jackson Samuel Sebastian
Index: 9
   Olivia
Amount of persons in hash table: 50
```

The function helps "create a new account" in our hash table by letting the user type in a name that will be created into a Person to be inserted into the hash table. The hash function shown 1.2 helps us choose an index to insert the "new account" into base on the person's name. Any manipulations to a registered user's friends list can be made using their Person class's functions.

Implementation:

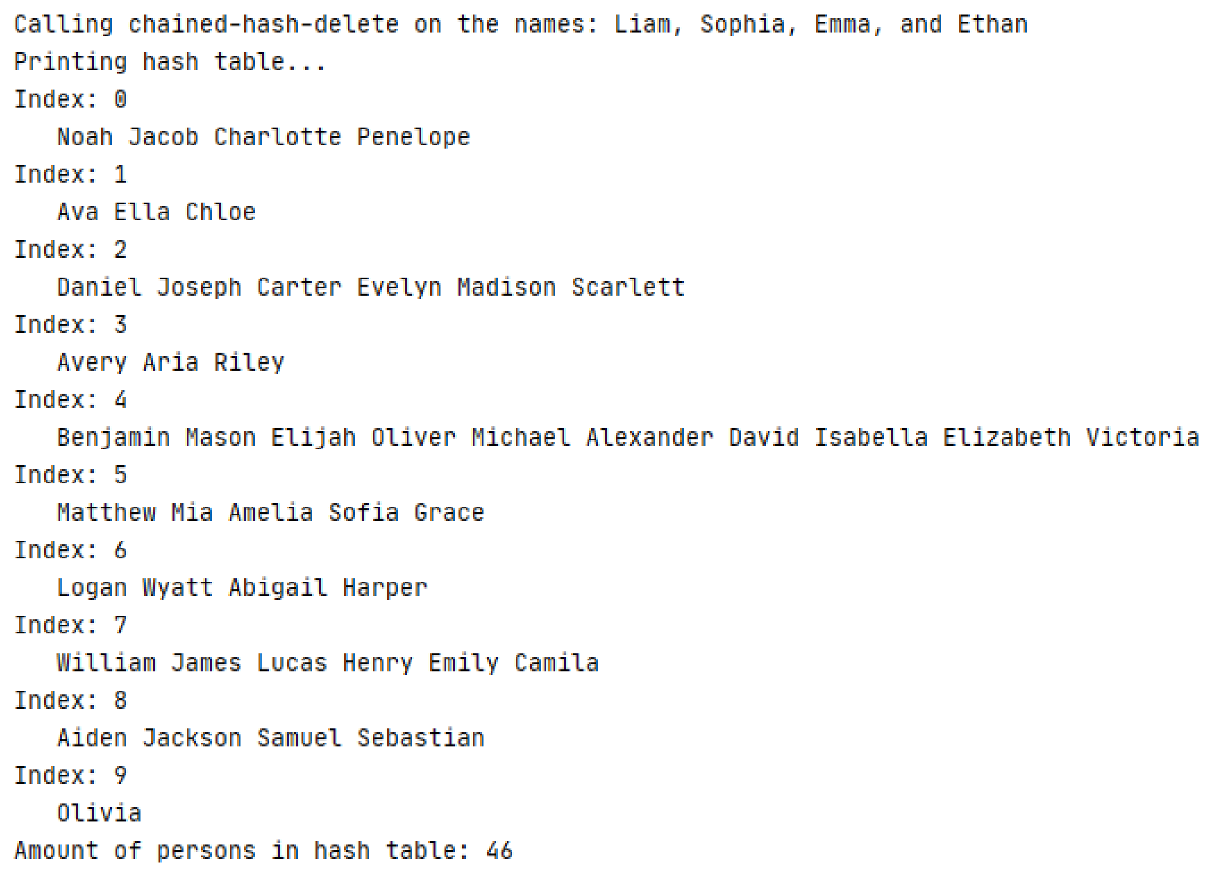


```
public void chainedHashInsert(HashTable ht, String name){
    Person newFriend = new Person(name);
    ht.getPeople()[ht.hash(name)].add(newFriend);//Hashes the user to an index in our hash table based on his/her name
}
```

The function takes in the global hash table containing registered users and a given name that isn't already in the hash table. A new Person object is made using the given name and is then inserted into the hash table based on our hashing function with their name.

1.4 Chained-Hash-Delete

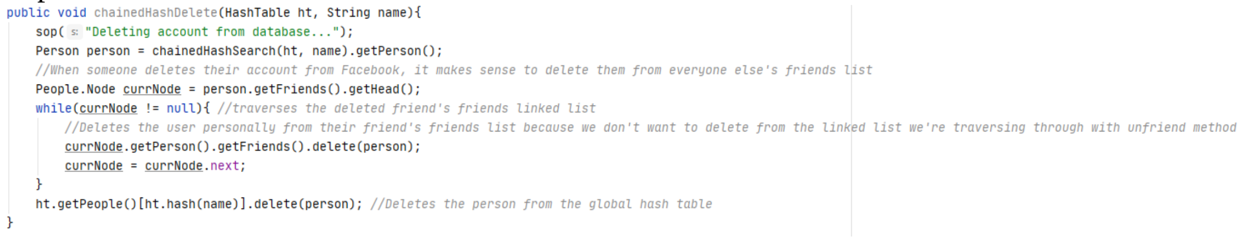
Design:



```
Calling chained-hash-delete on the names: Liam, Sophia, Emma, and Ethan
Printing hash table...
Index: 0
   Noah Jacob Charlotte Penelope
Index: 1
   Ava Ella Chloe
Index: 2
   Daniel Joseph Carter Evelyn Madison Scarlett
Index: 3
   Avery Aria Riley
Index: 4
   Benjamin Mason Elijah Oliver Michael Alexander David Isabella Elizabeth Victoria
Index: 5
   Matthew Mia Amelia Sofia Grace
Index: 6
   Logan Wyatt Abigail Harper
Index: 7
   William James Lucas Henry Emily Camila
Index: 8
   Aiden Jackson Samuel Sebastian
Index: 9
   Olivia
Amount of persons in hash table: 46
```

The function allows the user to "delete an account" by typing in a string name that is searched for in the hash table, and then procedurally deleted. The figure above calls delete on the hash table from 1.3, and as you can see, the corresponding names that the function calls on get deleted from the hash table. It also makes sense for us to delete the person from the friends list of whoever the person was friends with.

Implementation:

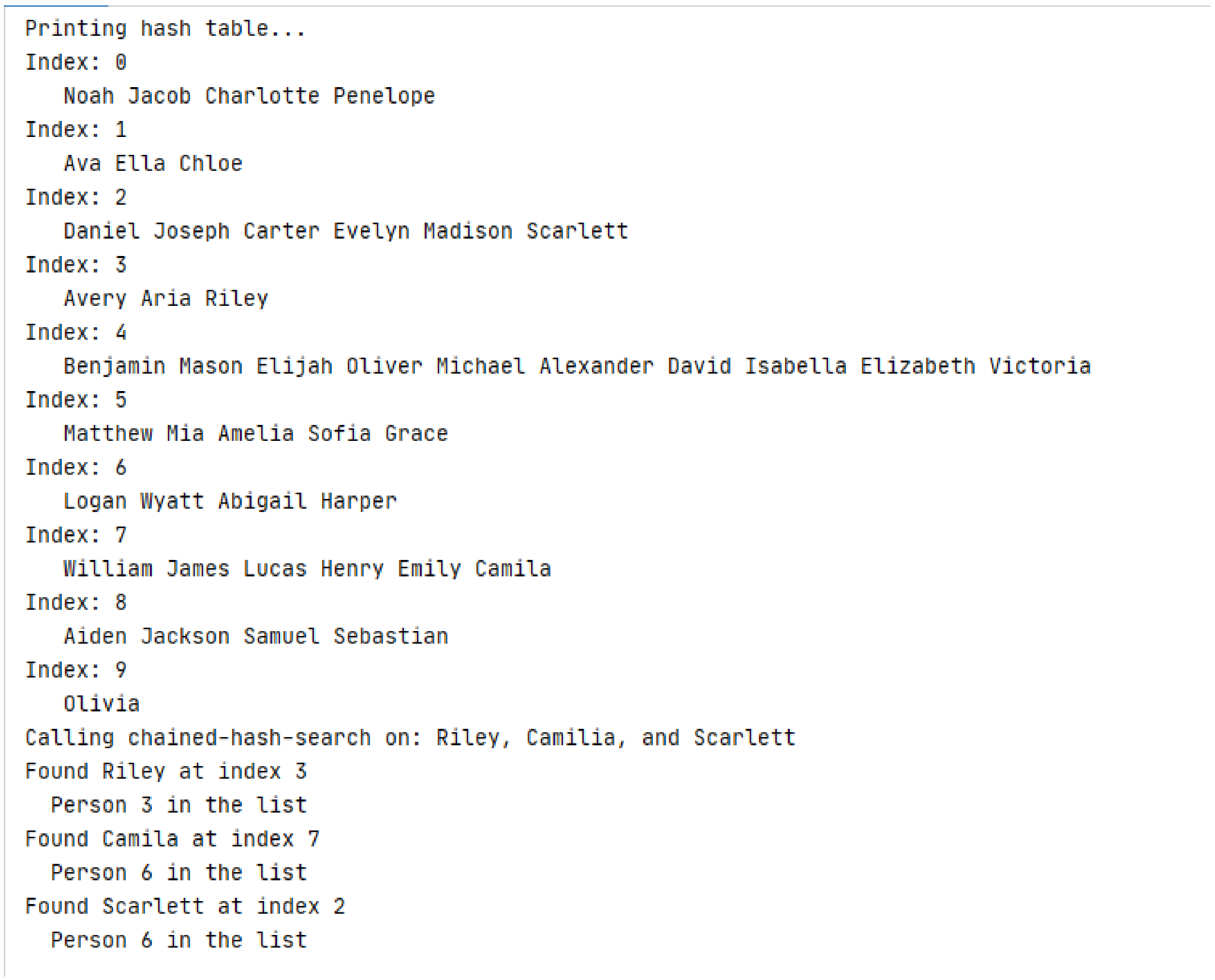


```java
public void chainedHashDelete(HashTable ht, String name){
    sop( s: "Deleting account from database...");
    Person person = chainedHashSearch(ht, name).getPerson();
    //When someone deletes their account from Facebook, it makes sense to delete them from everyone else's friends list
    People.Node currNode = person.getFriends().getHead();
    while(currNode != null){ //traverses the deleted friend's friends linked list
        //Deletes the user personally from their friend's friends list because we don't want to delete from the linked list we're traversing through with unfriend method
        currNode.getPerson().getFriends().delete(person);
        currNode = currNode.next;
    }
    ht.getPeople()[ht.hash(name)].delete(person); //Deletes the person from the global hash table
}
```

The function starts by using chainedHashSearch() to find the person we want to delete using the given string name within our hash table. A while loop is used with a dummy node to traverse through the deleted user's friends list. We don't call unfriend and directly delete the deleted friend from the traversed node's friends list to avoid null pointers since we don't want to delete nodes from the linked list we're traversing. After the deleted user is deleted from all of his/her friends' friends list, we finally delete the person from the hash table.

1.5 Chained-Hash-Search

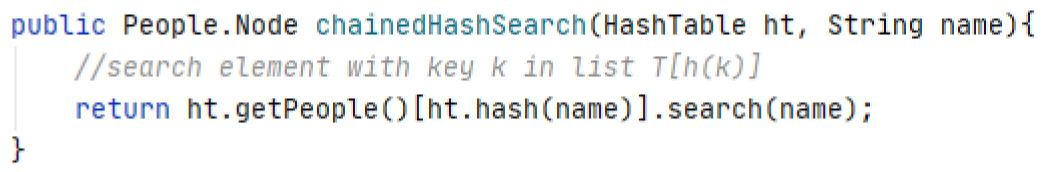
Design:



```
Printing hash table...
Index: 0
   Noah Jacob Charlotte Penelope
Index: 1
   Ava Ella Chloe
Index: 2
   Daniel Joseph Carter Evelyn Madison Scarlett
Index: 3
   Avery Aria Riley
Index: 4
   Benjamin Mason Elijah Oliver Michael Alexander David Isabella Elizabeth Victoria
Index: 5
   Matthew Mia Amelia Sofia Grace
Index: 6
   Logan Wyatt Abigail Harper
Index: 7
   William James Lucas Henry Emily Camila
Index: 8
   Aiden Jackson Samuel Sebastian
Index: 9
   Olivia
Calling chained-hash-search on: Riley, Camilia, and Scarlett
Found Riley at index 3
  Person 3 in the list
Found Camila at index 7
  Person 6 in the list
Found Scarlett at index 2
  Person 6 in the list
```

The function searches the global hash table for a node containing a Person object containing the given name by using the hash function on the given name. The linked list containing the name we're searching for is found using the hashing function on the given name, and then we search for the name using linked list search method within the linked list at that index. As shown in the figure above, the index of the person is found using the hashing function, and then the search() linked list method is used to find the person in that linked list.

Implementation:



```
public People.Node chainedHashSearch(HashTable ht, String name){
    //search element with key k in list T[h(k)]
    return ht.getPeople()[ht.hash(name)].search(name);
}
```

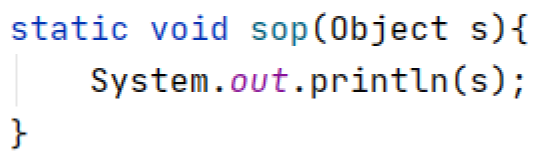
The function takes in the global hash table and a given string name. The name is converted to a natural number using our hashing function, and if the Person with the given name has already been inserted into the hash table, the hashing function will always give us the same index, meaning that the Person object with the given name is guaranteed to be in the linked list at that index of the hash table. The search() linked list method is used to search for a Person object with the given name in the linked list at that index.

2. List of Classes/Subroutines/Function calls:

2.1 Main Class

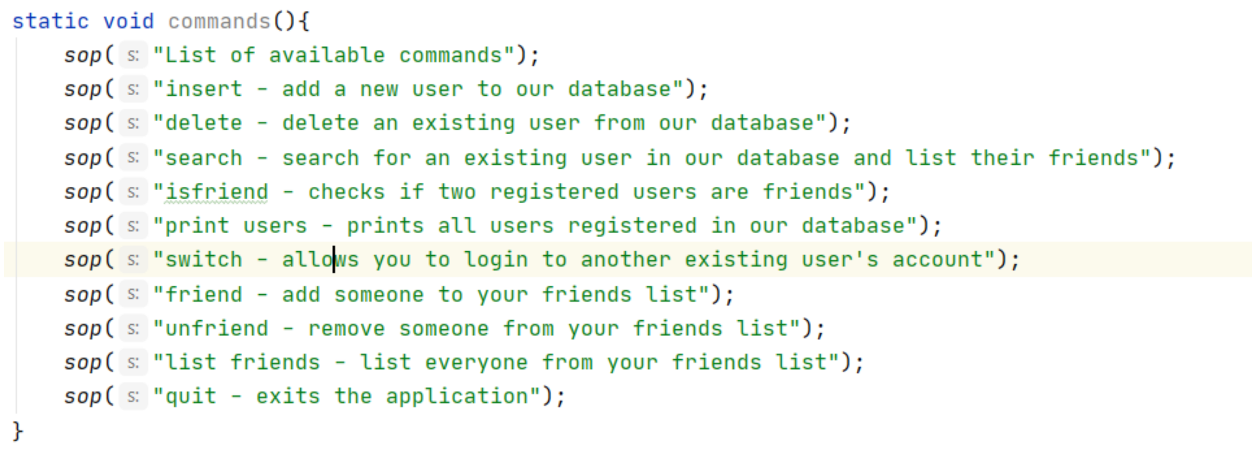
The main class runs our social media application. A user interface is generated for the user to interact with the functions in the other classes.

Functions:



```java
static void sop(Object s){
    System.out.println(s);
}
```

Serves as a short-hand for System.out.println(s) to save time writing the code.

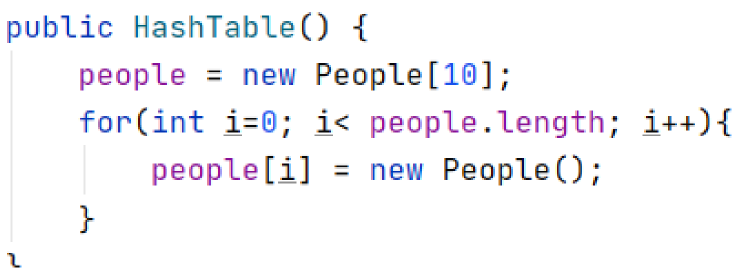


```java
static void commands(){
    sop( s: "List of available commands");
    sop( s: "insert - add a new user to our database");
    sop( s: "delete - delete an existing user from our database");
    sop( s: "search - search for an existing user in our database and list their friends");
    sop( s: "isfriend - checks if two registered users are friends");
    sop( s: "print users - prints all users registered in our database");
    sop( s: "switch - allows you to login to another existing user's account");
    sop( s: "friend - add someone to your friends list");
    sop( s: "unfriend - remove someone from your friends list");
    sop( s: "list friends - list everyone from your friends list");
    sop( s: "quit - exits the application");
}
```

Prints out a list of available commands so the user knows how to interact and call functions in our social media application.

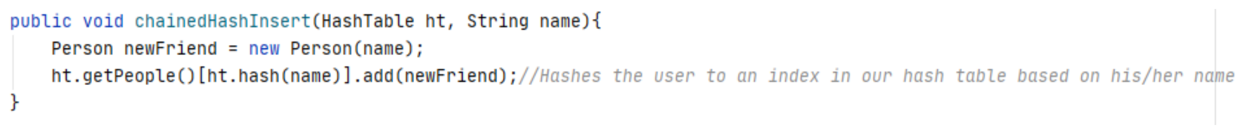
2.2 Hash Table class

The hash table class represents a data structure (hash table) containing registered users in a social media application. Changes to the database can be made using its methods.



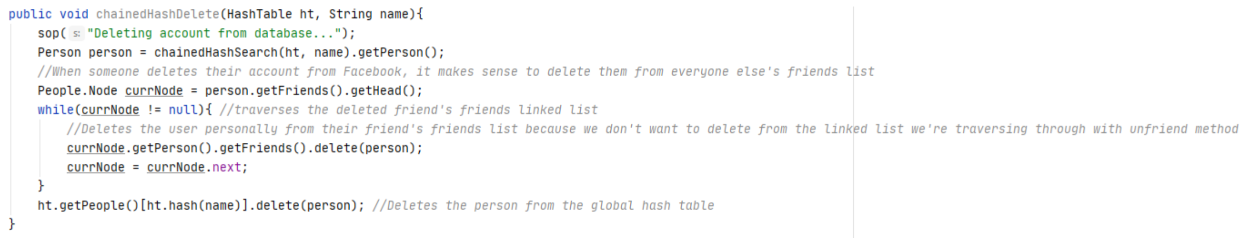
```java
public HashTable() {
    people = new People[10];
    for(int i=0; i< people.length; i++){
        people[i] = new People();
    }
}
```

Constructor for the HashTable class. Initializes the size of the hash table as 10 and creates a new linked list from our People class at each index to deal with collisions when hashing.



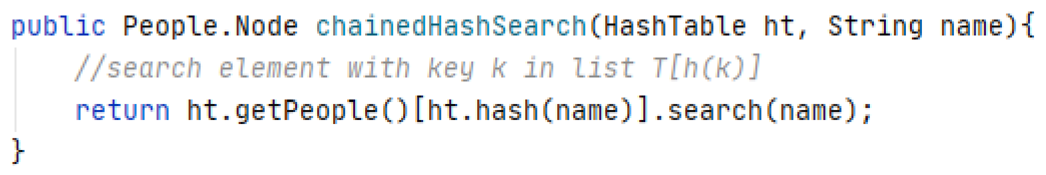
```java
public void chainedHashInsert(HashTable ht, String name){
    Person newFriend = new Person(name);
    ht.getPeople()[ht.hash(name)].add(newFriend);//Hashes the user to an index in our hash table based on his/her name
}
```

The function takes in the global hash table containing registered users and a given name that isn't already in the hash table. A new Person object is made using the given name and is then inserted into the hash table based on our hashing function with their name.



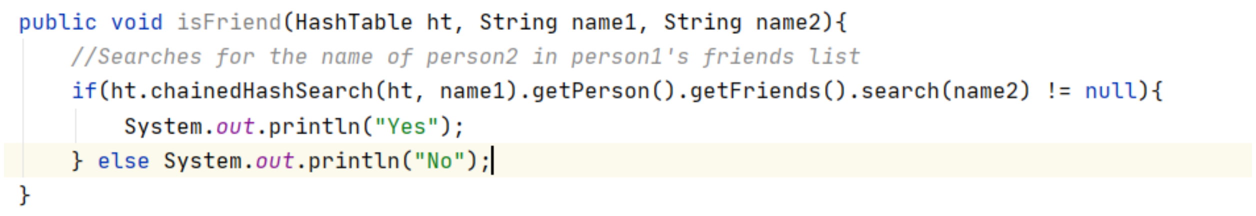
```java
public void chainedHashDelete(HashTable ht, String name){
    sop( s: "Deleting account from database...");
    Person person = chainedHashSearch(ht, name).getPerson();
    //When someone deletes their account from Facebook, it makes sense to delete them from everyone else's friends list
    People.Node currNode = person.getFriends().getHead();
    while(currNode != null){ //traverses the deleted friend's friends linked list
        //Deletes the user personally from their friend's friends list because we don't want to delete from the linked list we're traversing through with unfriend method
        currNode.getPerson().getFriends().delete(person);
        currNode = currNode.next;
    }
    ht.getPeople()[ht.hash(name)].delete(person); //Deletes the person from the global hash table
}
```

The function starts by using chainedHashSearch() to find the person we want to delete using the given string name within our hash table. A while loop is used with a dummy node to traverse through the deleted user's friends list. We don't call unfriend and directly delete the deleted friend from the traversed node's friends list to avoid null pointers since we don't want to delete nodes from the linked list we're traversing. After the deleted user is deleted from all of his/her friends' friends list, we finally delete the person from the hash table.



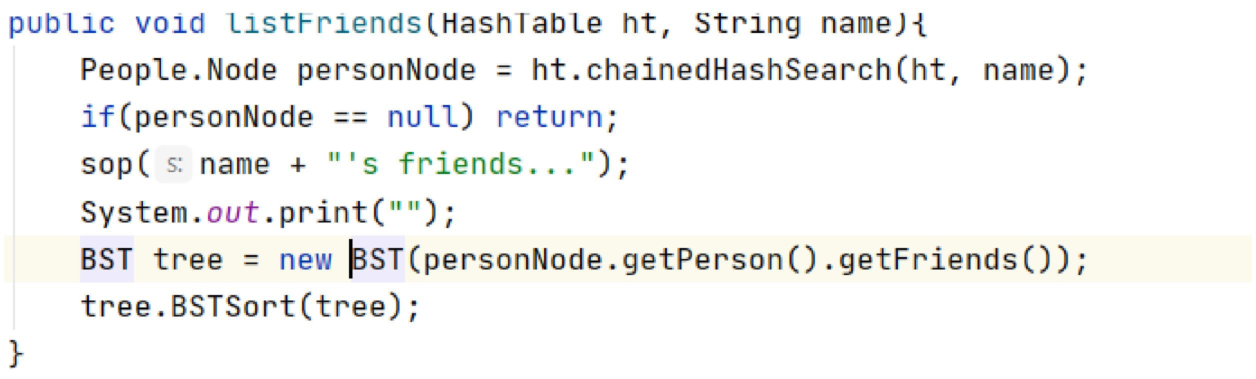
```java
public People.Node chainedHashSearch(HashTable ht, String name){
    //search element with key k in list T[h(k)]
    return ht.getPeople()[ht.hash(name)].search(name);
}
```

The function takes in the global hash table and a given string name. The name is converted to a natural number using our hashing function, and if the Person with the given name has already been inserted into the hash table, the hashing function will always give us the same index, meaning that the Person object with the given name is guaranteed to be in the linked list at that index of the hash table. The search() linked list method is used to search for a Person object with the given name in the linked list at that index.



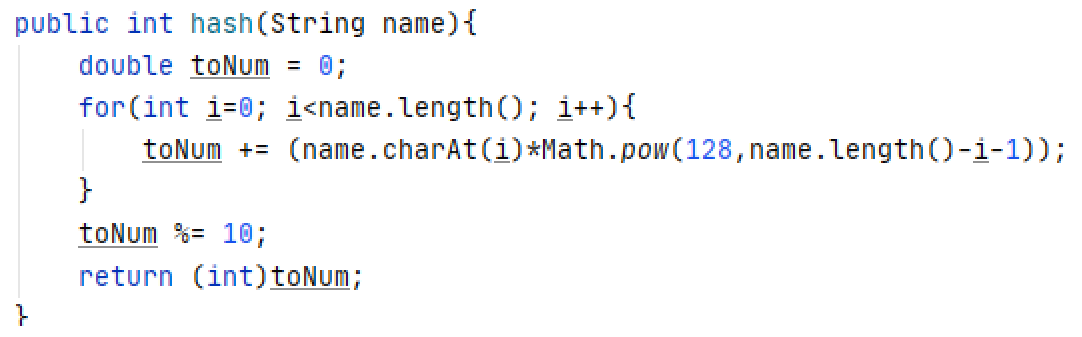
```java
public void isFriend(HashTable ht, String name1, String name2){
    //Searches for the name of person2 in person1's friends list
    if(ht.chainedHashSearch(ht, name1).getPerson().getFriends().search(name2) != null){
        System.out.println("Yes");
    } else System.out.println("No");
}
```

Used to check if two people are friends. Uses chainedHashSearch() to see if the names are registered users first in our hash table, and then calls search() linked list method to see if the names appear in each other’s friends list. Prints ‘Yes’ if the people with the given names are friends, Prints ‘No’ if not.



```java
public void listFriends(HashTable ht, String name){
    People.Node personNode = ht.chainedHashSearch(ht, name);
    if(personNode == null) return;
    sop( s: name + "'s friends...");
    System.out.print("");
    BST tree = new BST(personNode.getPerson().getFriends());
    tree.BSTSort(tree);
}
```

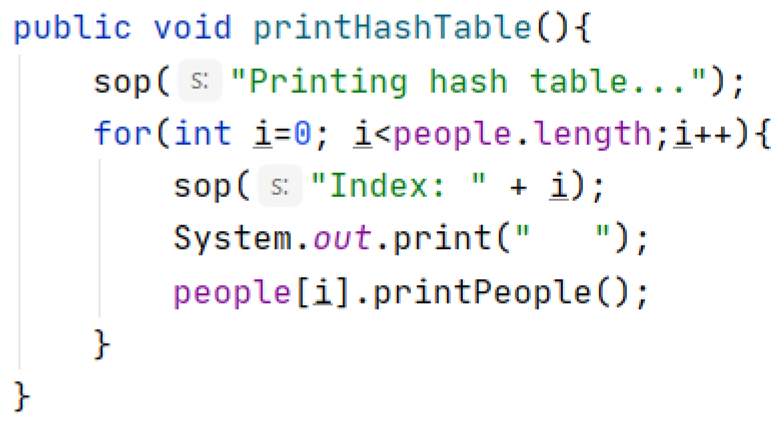
Lists the user’s friends using our BST sort class’s function BSTSort(). It does an in-order-walk of our constructed BST using the user’s friends list to print out the friends list in alphabetical order.



```java
public int hash(String name){
    double toNum = 0;
    for(int i=0; i<name.length(); i++){
        toNum += (name.charAt(i)*Math.pow(128,name.length()-i-1));
    }
    toNum %= 10;
    return (int)toNum;
}
```

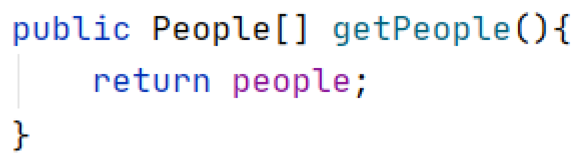
The purpose of this function is to turn a given string name is turn into a natural number based on radix notation to be used in other hashing functions. Our formula for the radix notation is as follows:

hash(name) = ((first digit as ASCII number) *128^(num of digits in name -1) + (second digit as ASCII number)*128^(num of digits in name -2) + … + (last digit as ASCII number)*128^(0)) mod 10, which gives us a double that is casted as an integer and then returned.



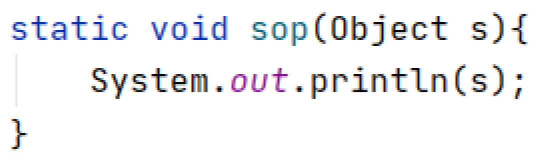
```java
public void printHashTable(){
    sop("Printing hash table...");
    for(int i=0; i<people.length;i++){
        sop("Index: " + i);
        System.out.print("    ");
        people[i].printPeople();
    }
}
```

Prints the names of the Nodes containing People objects in the hash table, as well as the corresponding index each name is under.



```java
public People[] getPeople(){
    return people;
}
```

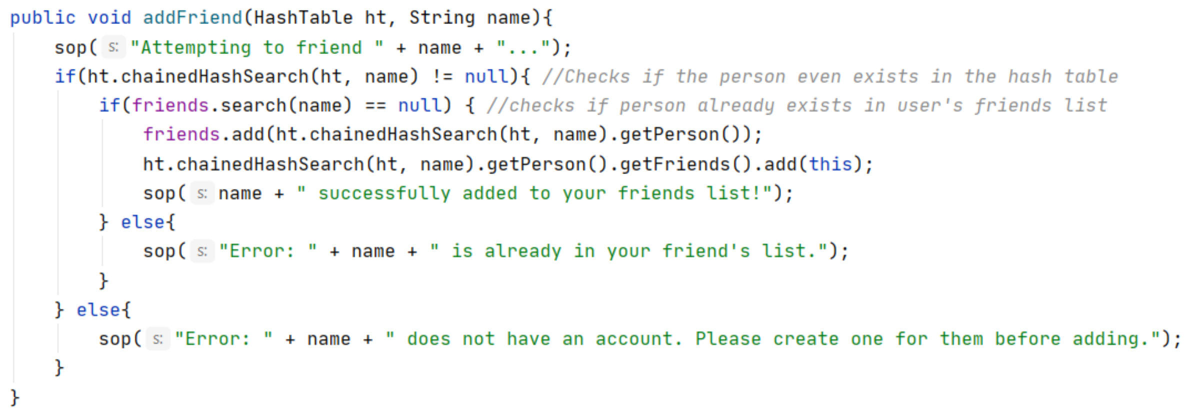
Getter for hash table array.



```java
static void sop(Object s){
    System.out.println(s);
}
```

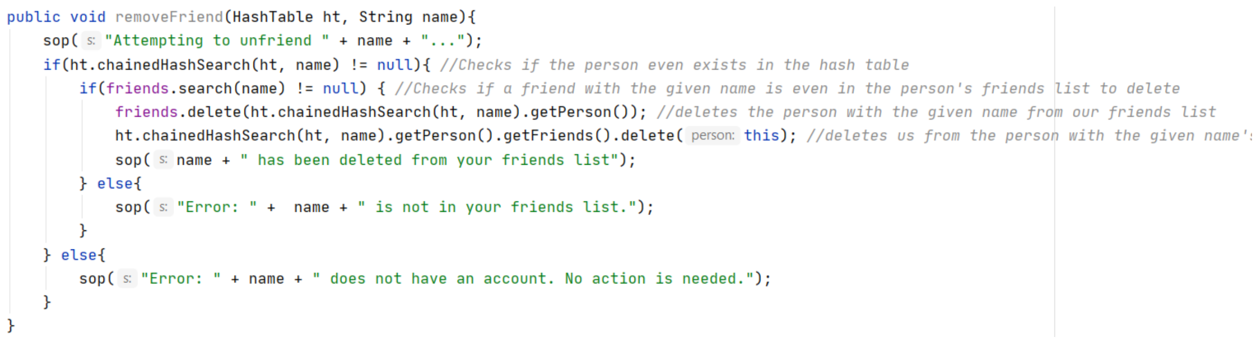
Serves as a short-hand for System.out.println(s) to save time writing the code.

2.3 Person class



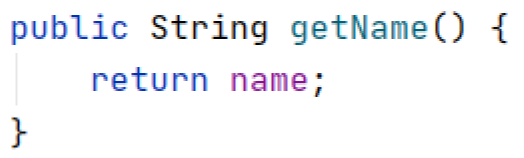
```java
public void addFriend(HashTable ht, String name){
    sop( s: "Attempting to friend " + name + "...");
    if(ht.chainedHashSearch(ht, name) != null){ //Checks if the person even exists in the hash table
        if(friends.search(name) == null) { //checks if person already exists in user's friends list
            friends.add(ht.chainedHashSearch(ht, name).getPerson());
            ht.chainedHashSearch(ht, name).getPerson().getFriends().add(this);
            sop( s: name + " successfully added to your friends list!");
        } else{
            sop( s: "Error: " + name + " is already in your friend's list.");
        }
    } else{
        sop( s: "Error: " + name + " does not have an account. Please create one for them before adding.");
    }
}
```

Retrieves a person from our hash table based on a given name and adds them to our friends list. This also adds the user to their friends lists. ChainedHashSearch() I used to check if the person even exists in the hash table. A secondary check is done to see if the person is already in the user’s friends list before adding them to the friends list of each user. The else statement covers the case where the person is not a registered user in the hash table.



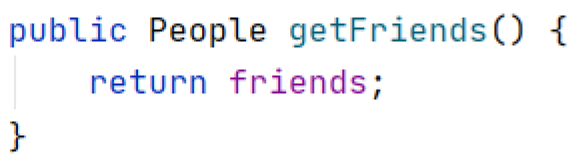
```java
public void removeFriend(HashTable ht, String name){
    sop( s: "Attempting to unfriend " + name + "...");
    if(ht.chainedHashSearch(ht, name) != null){ //Checks if the person even exists in the hash table
        if(friends.search(name) != null) { //Checks if a friend with the given name is even in the person's friends list to delete
            friends.delete(ht.chainedHashSearch(ht, name).getPerson()); //deletes the person with the given name from our friends list
            ht.chainedHashSearch(ht, name).getPerson().getFriends().delete( person: this); //deletes us from the person with the given name'
            sop( s: name + " has been deleted from your friends list");
        } else{
            sop( s: "Error: " +  name + " is not in your friends list.");
        }
    } else{
        sop( s: "Error: " + name + " does not have an account. No action is needed.");
    }
}
```

Retrieves a person from our hash table based on a given name and deletes them from our friends list. This also deletes us from their friends list. ChainedHashSearch() is called to check if the person exists in the hash table before removal. Another check is done to check if the friend with the given name is even in the person’s friends list to delete. If these checks pass, then the deletion commences. Otherwise, a print statement is used to tell the user that the person is either not in their friends list or the person doesn’t own an account.



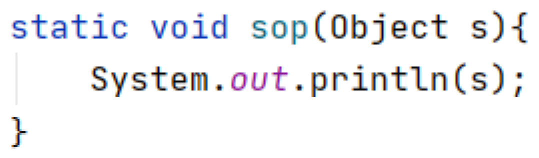
```java
public String getName() {
    return name;
}
```

Getting for the name of the Person object.



```java
public People getFriends() {
    return friends;
}
```

Getter for the Person object's linked list of friends

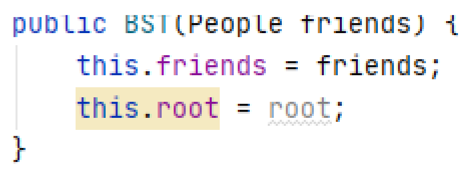


```java
static void sop(Object s){
    System.out.println(s);
}
```

Serves as a short-hand for System.out.println(s) to save time writing the code.

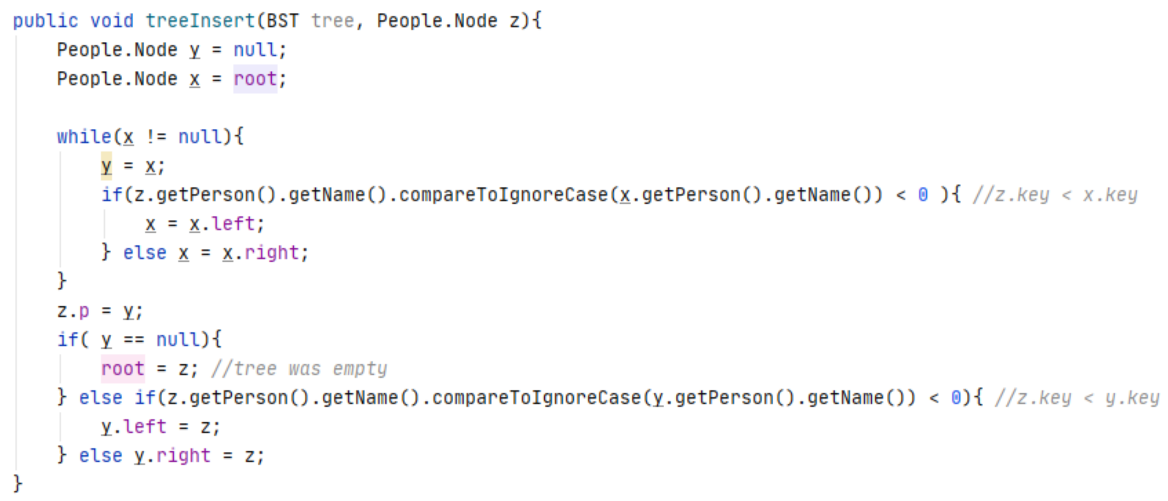
2.4 BST class

Represents a binary search tree data structure.



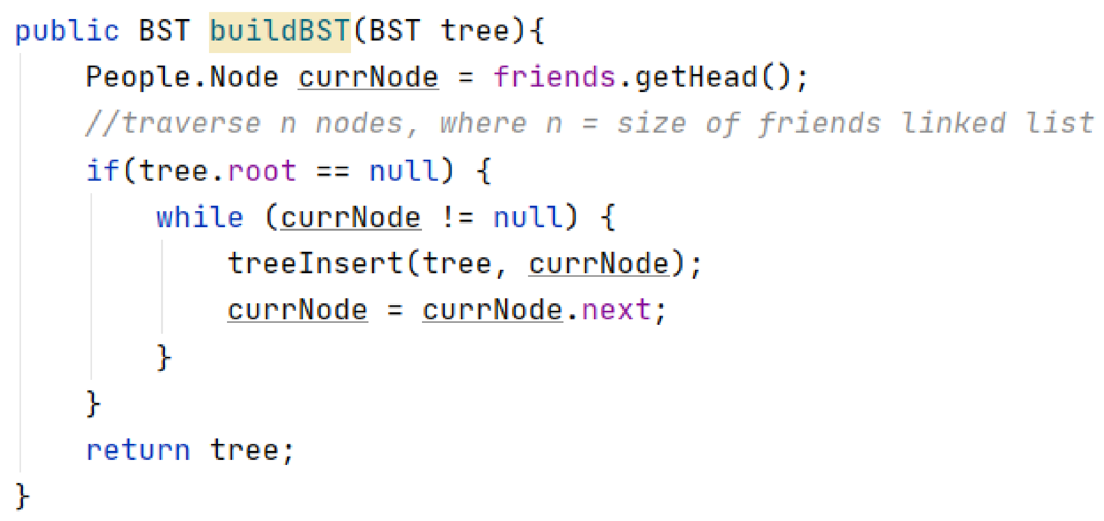
```java
public BST(People friends) {
    this.friends = friends;
    this.root = root;
}
```

Constructor for the BST class. Initializes the tree and the root of the tree.



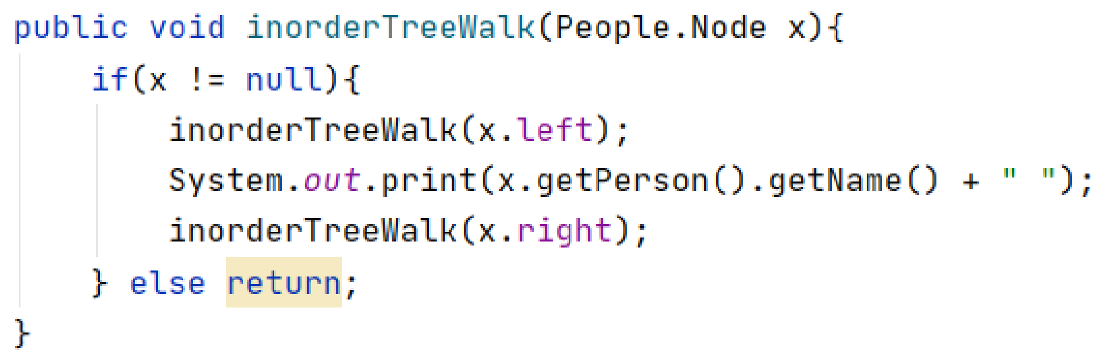
```java
public void treeInsert(BST tree, People.Node z){
    People.Node y = null;
    People.Node x = root;

    while(x != null){
        y = x;
        if(z.getPerson().getName().compareToIgnoreCase(x.getPerson().getName()) < 0 ){ //z.key < x.key
            x = x.left;
        } else x = x.right;
    }
    z.p = y;
    if( y == null){
        root = z; //tree was empty
    } else if(z.getPerson().getName().compareToIgnoreCase(y.getPerson().getName()) < 0){ //z.key < y.key
        y.left = z;
    } else y.right = z;
}
```

Inserts a node containing a person object into the given binary search tree. Y acts as trailer always hovering above x. The first while loop helps us find a NIL location for our node z based on the binary search tree property. After we find this spot, we set the parent of z to y, and then we set z to be the left or right of y depending on which side of y z is on. If the tree is empty, then we just set the root to be z. The pseudocode uses keys, but in our case we're comparing names in alphabetical order, so I used the String class's building .compareToIgnoreCase() to compare the values with the binary search tree property in mind.



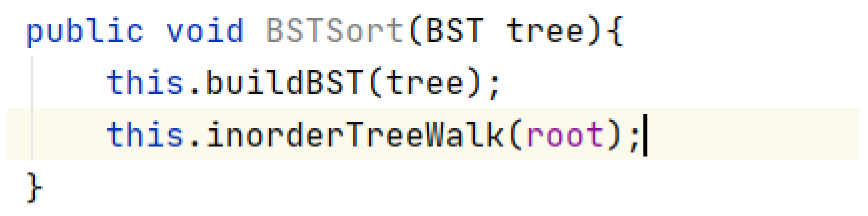
```java
public BST buildBST(BST tree){
    People.Node currNode = friends.getHead();
    //traverse n nodes, where n = size of friends linked list
    if(tree.root == null) {
        while (currNode != null) {
            treeInsert(tree, currNode);
            currNode = currNode.next;
        }
    }
    return tree;
}
```

Builds the binary search tree by inserting the currNode as we traverse through the friends linked list to the tree in order. Returns the BST at the end.



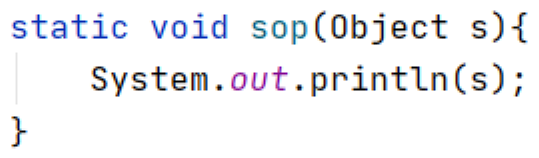
```java
public void inorderTreeWalk(People.Node x){
    if(x != null){
        inorderTreeWalk(x.left);
        System.out.print(x.getPerson().getName() + " ");
        inorderTreeWalk(x.right);
    } else return;
}
```

Does the inorder tree walk starting at a given node. Prints out the names of the node in the order of the way we walk. It traverses the nodes using recursion.



```java
public void BSTSort(BST tree){
    this.buildBST(tree);
    this.inorderTreeWalk(root);
}
```

Constructs a BST by calling buildBST(tree) and calling inorderTreeWalk() on the root of the tree. BST sort prints out the sorted order of the friends list that came with the tree.

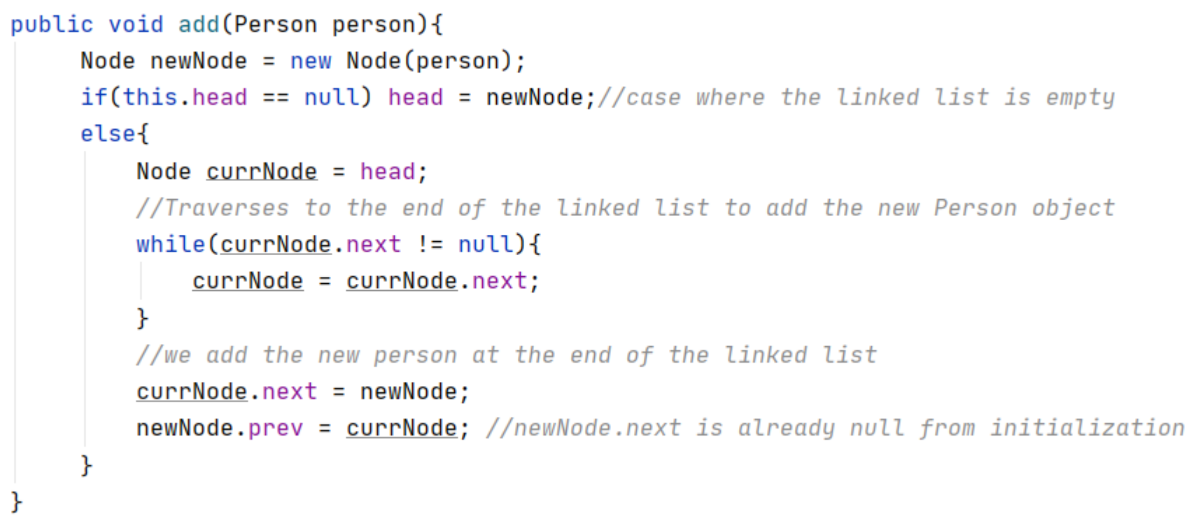


```java
static void sop(Object s){
    System.out.println(s);
}
```

Serves as a short-hand for System.out.println(s) to save time writing the code.

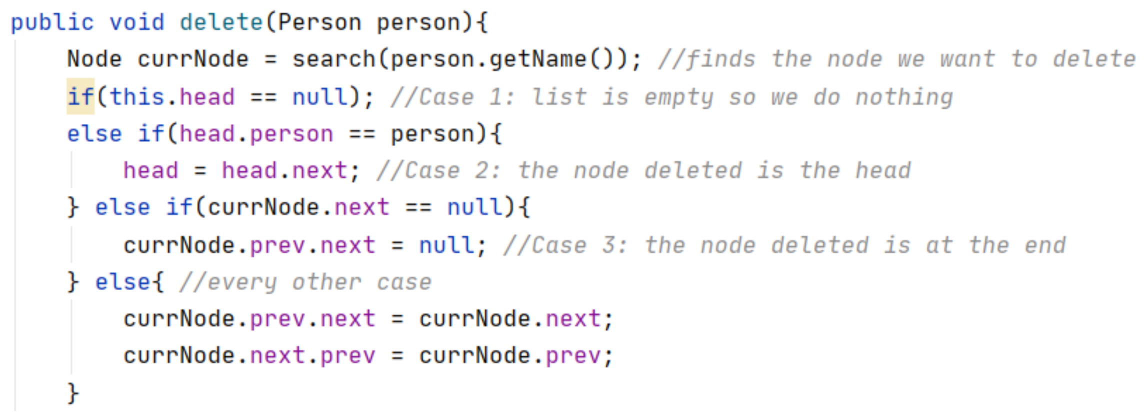
2.5 People class

A class representing a linked list data structure. Elements in the linked list are composed of Node objects.



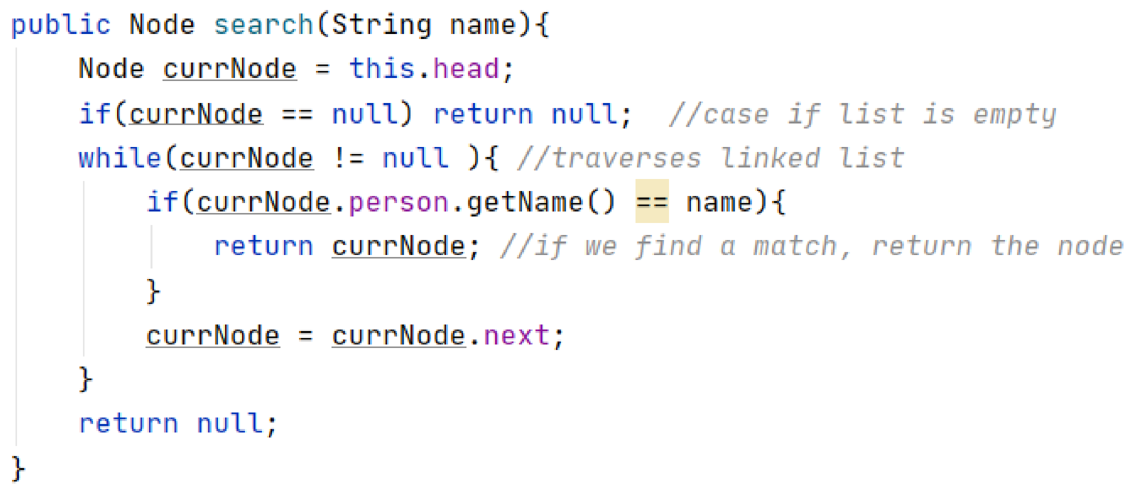
```java
public void add(Person person){
    Node newNode = new Node(person);
    if(this.head == null) head = newNode;//case where the linked list is empty
    else{
        Node currNode = head;
        //Traverses to the end of the linked list to add the new Person object
        while(currNode.next != null){
            currNode = currNode.next;
        }
        //we add the new person at the end of the linked list
        currNode.next = newNode;
        newNode.prev = currNode; //newNode.next is already null from initialization
    }
}
```

Adds a node containing the given person object to the end of the linked list. The first if statement covers the case where the list is empty so we just make the given person the head. If it goes to the else statement, currNode is set to the head and used to traverse to the end of the linked list. Then we add the given person to the end of the list where currNode stops.



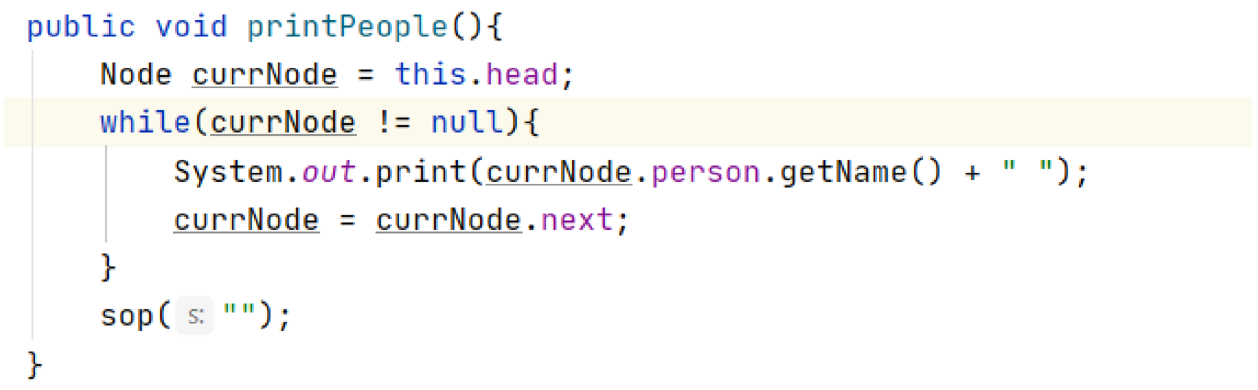
```java
public void delete(Person person){
    Node currNode = search(person.getName()); //finds the node we want to delete
    if(this.head == null); //Case 1: list is empty so we do nothing
    else if(head.person == person){
        head = head.next; //Case 2: the node deleted is the head
    } else if(currNode.next == null){
        currNode.prev.next = null; //Case 3: the node deleted is at the end
    } else{ //every other case
        currNode.prev.next = currNode.next;
        currNode.next.prev = currNode.prev;
    }
```

Deletes a node containing the given Person object from the linked list. The search() method Is used to find the node we want to delete. A different case applies to each situation. Case 1 covers an empty list so there is node to delete. Case 2 covers the node we want to delete being at the head of the list. Case 3 covers the case where the node at the very end of the linked list is deleted so we just set the next pointer of the second-to-last node to null. And finally, the else statement covers every other case for standard deletion.



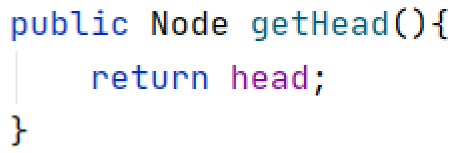
```
public Node search(String name){
    Node currNode = this.head;
    if(currNode == null) return null;  //case if list is empty
    while(currNode != null ){ //traverses linked list
        if(currNode.person.getName() == name){
            return currNode; //if we find a match, return the node
        }
        currNode = currNode.next;
    }
    return null;
}
```

Searches for a node containing a Person object with the given name within the linked list. The first case covers an empty list so we return null. The while loop and currNode is used to traverse the linked list. If any of the nodes containing a Person object have a matching name to the one we're looking for, we return that node since we have a correct match. Otherwise, we return null since we couldn't find the name in the linked list.



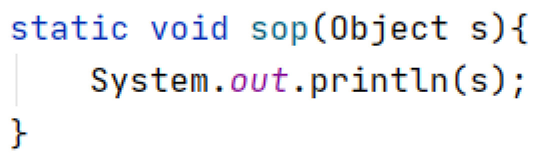
```
public void printPeople(){
    Node currNode = this.head;
    while(currNode != null){
        System.out.print(currNode.person.getName() + " ");
        currNode = currNode.next;
    }
    sop( s: "");
}
```

Prints the names of the people in the linked list in the order they were added.



```
public Node getHead(){
    return head;
}
```

Getting for the head of the linked list.

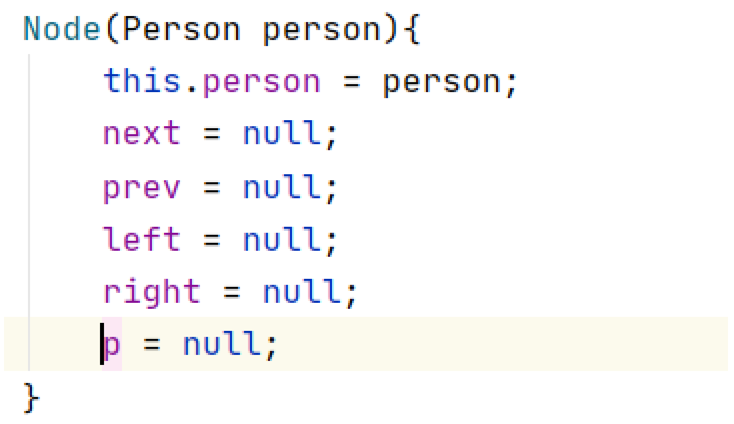


```java
static void sop(Object s){
    System.out.println(s);
}
```

Serves as a short-hand for System.out.println(s) to save time writing the code.

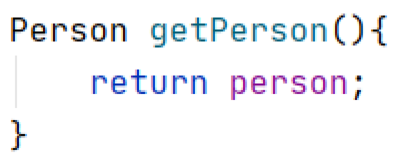
Node Inner Class

A node class to be used in type People linked lists and BST data structures.



```java
Node(Person person){
    this.person = person;
    next = null;
    prev = null;
    left = null;
    right = null;
    p = null;
}
```

A node representing the given Person object. Initializes next and prev nodes for a standard People linked list, and left, right and p for the BST data structure.



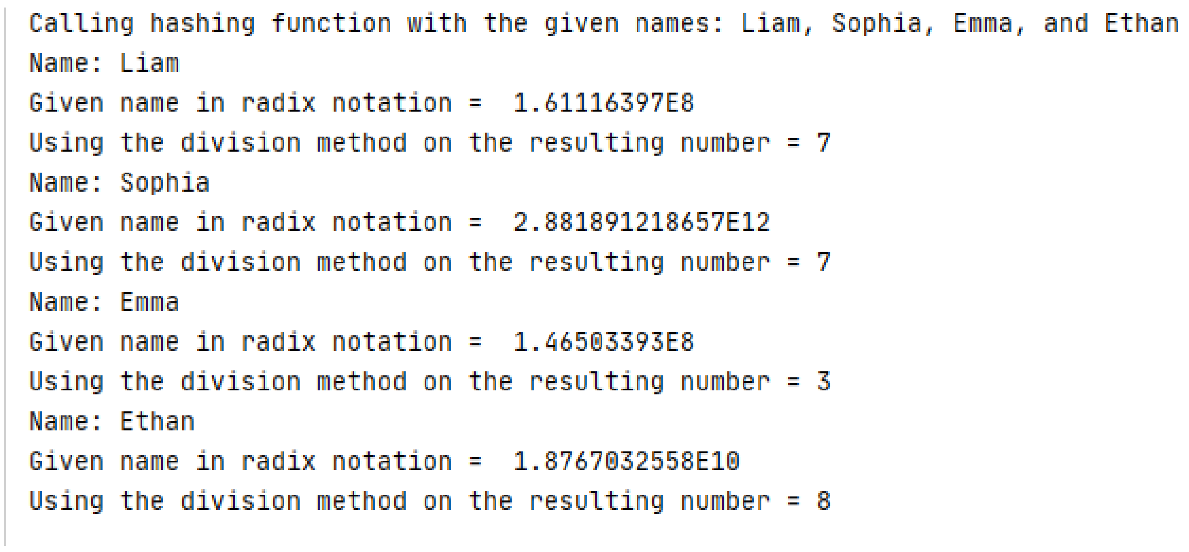
```java
Person getPerson(){
    return person;
}
```

Getter for the Person contained in the node.

3. Self-testing Screenshots:

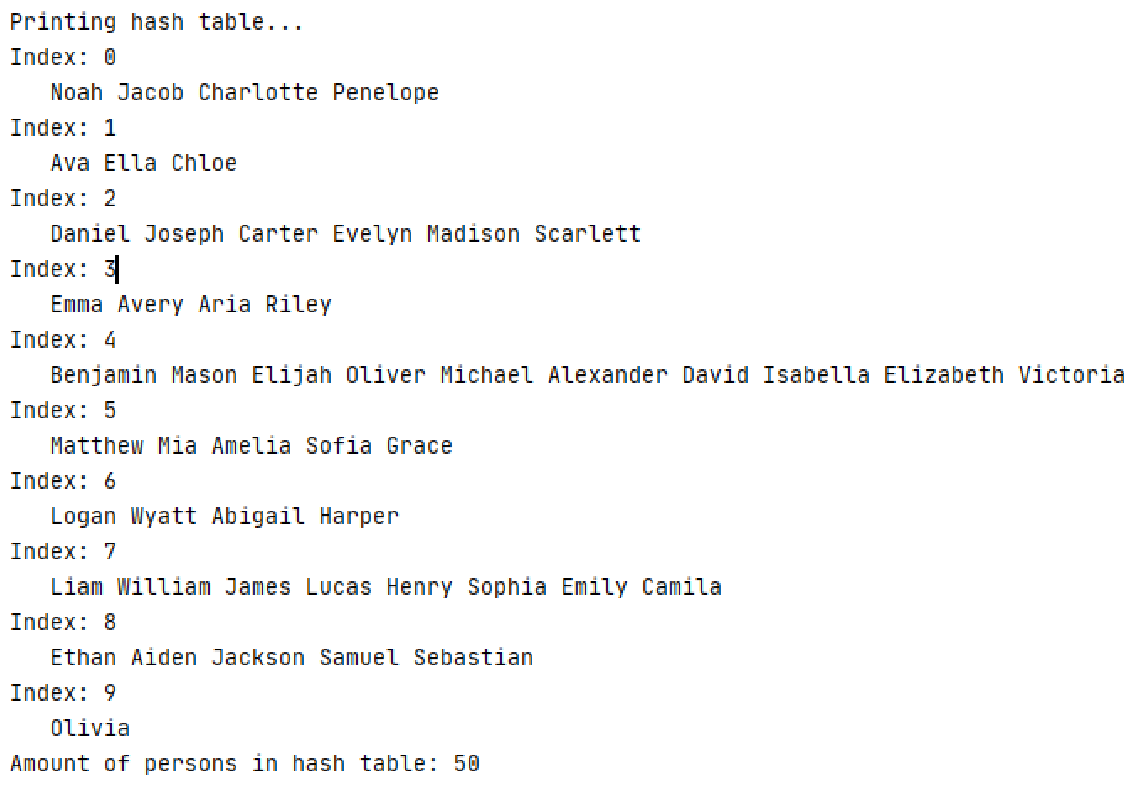
3.1 Hash Table class functions

Hash function on the given names



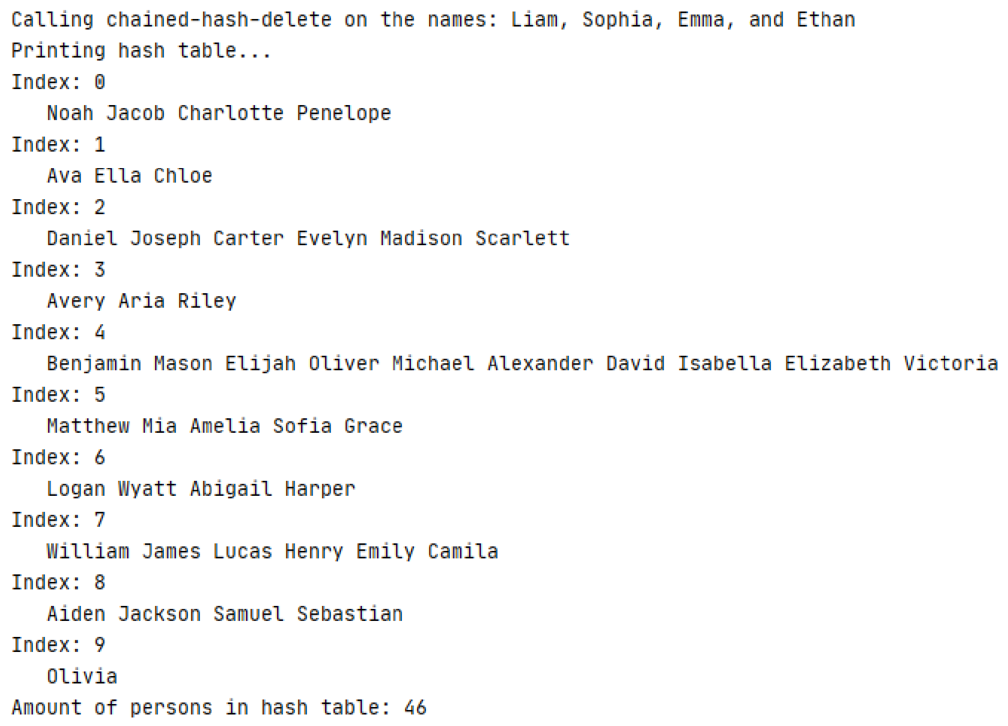
```
Calling hashing function with the given names: Liam, Sophia, Emma, and Ethan
Name: Liam
Given name in radix notation =  1.61116397E8
Using the division method on the resulting number = 7
Name: Sophia
Given name in radix notation =  2.881891218657E12
Using the division method on the resulting number = 7
Name: Emma
Given name in radix notation =  1.46503393E8
Using the division method on the resulting number = 3
Name: Ethan
Given name in radix notation =  1.8767032558E10
Using the division method on the resulting number = 8
```

ChainedHashInsert() on the 50 names given in PA3



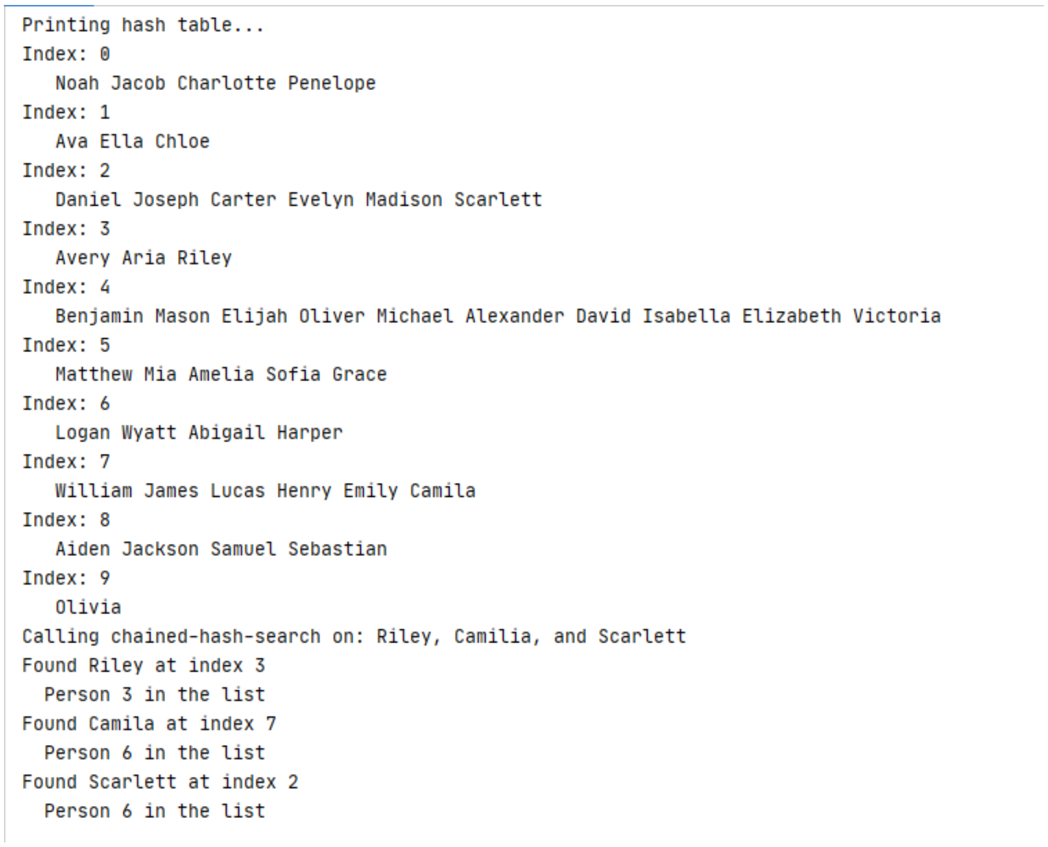
```
Printing hash table...
Index: 0
   Noah Jacob Charlotte Penelope
Index: 1
   Ava Ella Chloe
Index: 2
   Daniel Joseph Carter Evelyn Madison Scarlett
Index: 3
   Emma Avery Aria Riley
Index: 4
   Benjamin Mason Elijah Oliver Michael Alexander David Isabella Elizabeth Victoria
Index: 5
   Matthew Mia Amelia Sofia Grace
Index: 6
   Logan Wyatt Abigail Harper
Index: 7
   Liam William James Lucas Henry Sophia Emily Camila
Index: 8
   Ethan Aiden Jackson Samuel Sebastian
Index: 9
   Olivia
Amount of persons in hash table: 50
```

ChainedHashDelete() on the given names



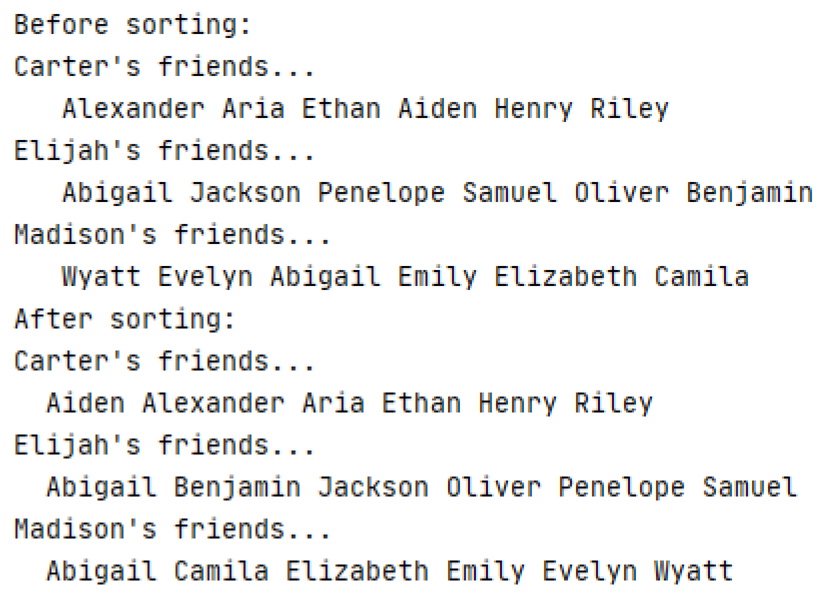
```
Calling chained-hash-delete on the names: Liam, Sophia, Emma, and Ethan
Printing hash table...
Index: 0
   Noah Jacob Charlotte Penelope
Index: 1
   Ava Ella Chloe
Index: 2
   Daniel Joseph Carter Evelyn Madison Scarlett
Index: 3
   Avery Aria Riley
Index: 4
   Benjamin Mason Elijah Oliver Michael Alexander David Isabella Elizabeth Victoria
Index: 5
   Matthew Mia Amelia Sofia Grace
Index: 6
   Logan Wyatt Abigail Harper
Index: 7
   William James Lucas Henry Emily Camila
Index: 8
   Aiden Jackson Samuel Sebastian
Index: 9
   Olivia
Amount of persons in hash table: 46
```

ChainedHashSearch() on the names: Riley, Camila, and Scarlett



```
Printing hash table...
Index: 0
   Noah Jacob Charlotte Penelope
Index: 1
   Ava Ella Chloe
Index: 2
   Daniel Joseph Carter Evelyn Madison Scarlett
Index: 3
   Avery Aria Riley
Index: 4
   Benjamin Mason Elijah Oliver Michael Alexander David Isabella Elizabeth Victoria
Index: 5
   Matthew Mia Amelia Sofia Grace
Index: 6
   Logan Wyatt Abigail Harper
Index: 7
   William James Lucas Henry Emily Camila
Index: 8
   Aiden Jackson Samuel Sebastian
Index: 9
   Olivia
Calling chained-hash-search on: Riley, Camilia, and Scarlett
Found Riley at index 3
  Person 3 in the list
Found Camila at index 7
  Person 6 in the list
Found Scarlett at index 2
  Person 6 in the list
```

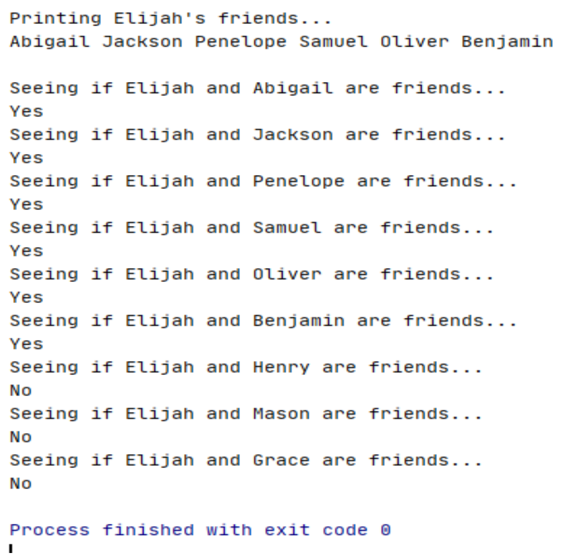
listFriends()



```
Before sorting:
Carter's friends...
   Alexander Aria Ethan Aiden Henry Riley
Elijah's friends...
   Abigail Jackson Penelope Samuel Oliver Benjamin
Madison's friends...
   Wyatt Evelyn Abigail Emily Elizabeth Camila
After sorting:
Carter's friends...
  Aiden Alexander Aria Ethan Henry Riley
Elijah's friends...
  Abigail Benjamin Jackson Oliver Penelope Samuel
Madison's friends...
  Abigail Camila Elizabeth Emily Evelyn Wyatt
```

Note how after the sort, the friends are printed out in alphabetical order.

isFriend() on Elijah

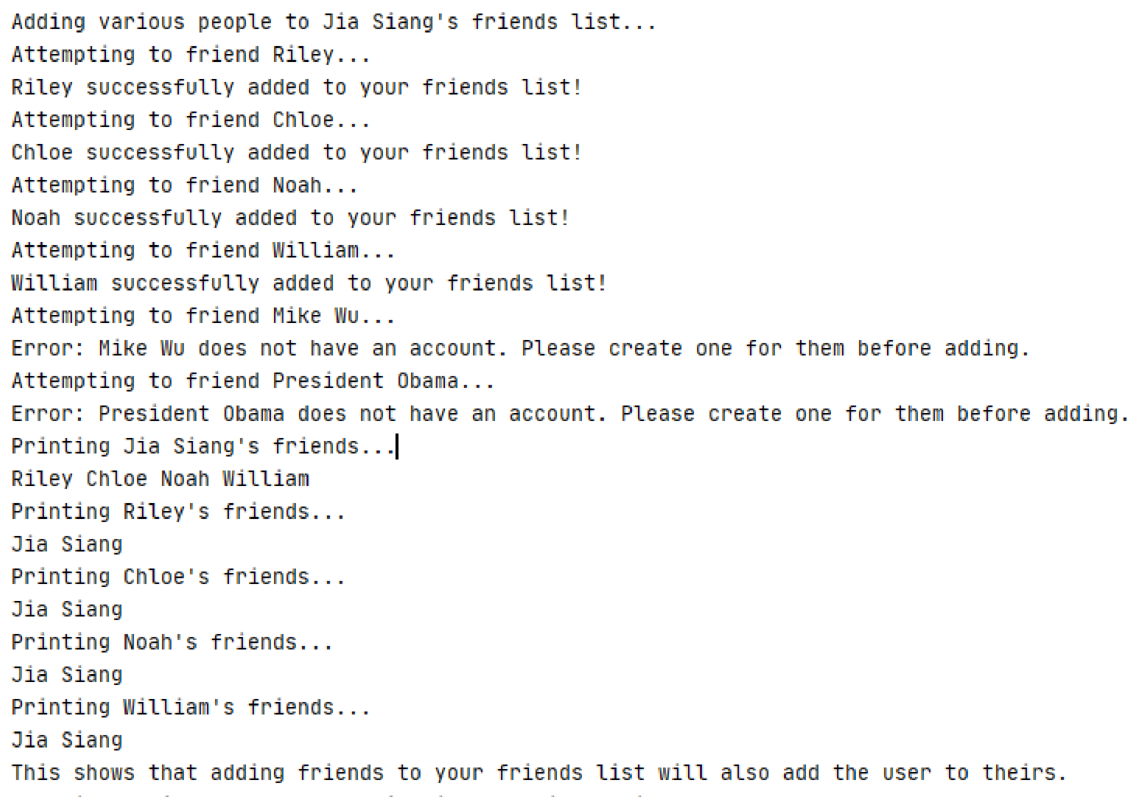


```
Printing Elijah's friends...
Abigail Jackson Penelope Samuel Oliver Benjamin

Seeing if Elijah and Abigail are friends...
Yes
Seeing if Elijah and Jackson are friends...
Yes
Seeing if Elijah and Penelope are friends...
Yes
Seeing if Elijah and Samuel are friends...
Yes
Seeing if Elijah and Oliver are friends...
Yes
Seeing if Elijah and Benjamin are friends...
Yes
Seeing if Elijah and Henry are friends...
No
Seeing if Elijah and Mason are friends...
No
Seeing if Elijah and Grace are friends...
No

Process finished with exit code 0
```

3.2 Person class functions

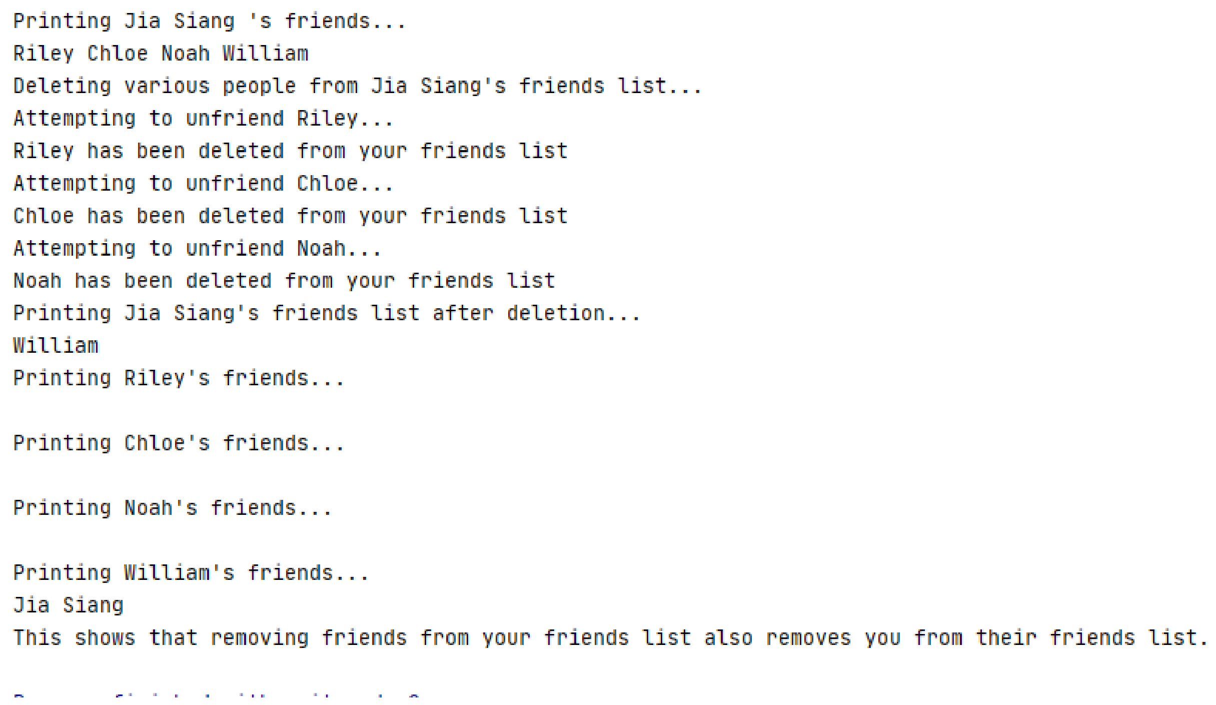
The functions in this section use the operations add() delete() search() from the People (linked list) class so these screenshots show testing for both classes.

addFriend() to Jia Siang’s friends list



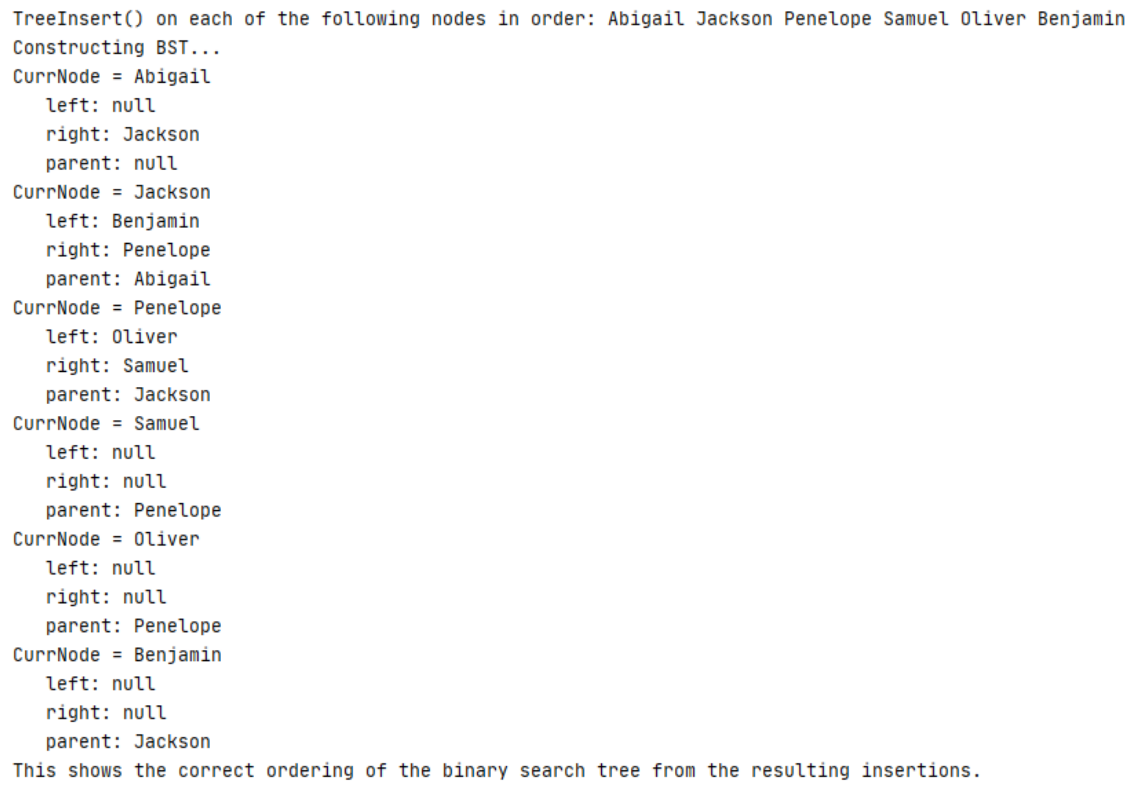
```
Adding various people to Jia Siang's friends list...
Attempting to friend Riley...
Riley successfully added to your friends list!
Attempting to friend Chloe...
Chloe successfully added to your friends list!
Attempting to friend Noah...
Noah successfully added to your friends list!
Attempting to friend William...
William successfully added to your friends list!
Attempting to friend Mike Wu...
Error: Mike Wu does not have an account. Please create one for them before adding.
Attempting to friend President Obama...
Error: President Obama does not have an account. Please create one for them before adding.
Printing Jia Siang's friends...
Riley Chloe Noah William
Printing Riley's friends...
Jia Siang
Printing Chloe's friends...
Jia Siang
Printing Noah's friends...
Jia Siang
Printing William's friends...
Jia Siang
This shows that adding friends to your friends list will also add the user to theirs.
```

removeFriend() on Jia Siang’s friends list



```
Printing Jia Siang 's friends...
Riley Chloe Noah William
Deleting various people from Jia Siang's friends list...
Attempting to unfriend Riley...
Riley has been deleted from your friends list
Attempting to unfriend Chloe...
Chloe has been deleted from your friends list
Attempting to unfriend Noah...
Noah has been deleted from your friends list
Printing Jia Siang's friends list after deletion...
William
Printing Riley's friends...

Printing Chloe's friends...

Printing Noah's friends...

Printing William's friends...
Jia Siang
This shows that removing friends from your friends list also removes you from their friends list.
```

3.4 BST class



```
TreeInsert() on each of the following nodes in order: Abigail Jackson Penelope Samuel Oliver Benjamin
Constructing BST...
CurrNode = Abigail
   left: null
   right: Jackson
   parent: null
CurrNode = Jackson
   left: Benjamin
   right: Penelope
   parent: Abigail
CurrNode = Penelope
   left: Oliver
   right: Samuel
   parent: Jackson
CurrNode = Samuel
   left: null
   right: null
   parent: Penelope
CurrNode = Oliver
   left: null
   right: null
   parent: Penelope
CurrNode = Benjamin
   left: null
   right: null
   parent: Jackson
This shows the correct ordering of the binary search tree from the resulting insertions.
```

The rest of the code is already tested using the listFriends() method in 3.2

4. Setup Procedure (Windows):

1) Download the .zip file named ‘PA3-Fung’ containing PA3.jar, PA3_Report_Fung.pdf, and the corresponding java class files for the code implementation off the site I uploaded it to.

2) Extract the .zip file using a program such as 7Zip.

3) To run the program, open the command line – if you don’t know where it is, type ‘cmd’ in the Windows search bar.

4) Type in “java -jar ” and enter the file directory of PA3.jar, for example “C:\Users\File_Path\PA3.jar”

5) Upon starting the program, 50 registered users will be in the database already. Some will have friends, some may not. You can make a new user, and after you can switch to an existing user by entering the corresponding commands.

6) Type ‘commands’ for a guide on what you functions you can use in this program

5. Problems Encountered:

I ran into an issue in my hashing function method. The initial plan was to use the same radix notation done in the power point slides. I was returning the same value after the mod operation no matter what the string was. Turns out, I hit the max range for the primitive type so adding by another char's ASCII value * a power of 128 would still give us the same number. I fixed this by setting the number we add to as a double and then casting it as an int after we do the mod operation.

I forgot how to make a linked list class so I had to relearn old concepts. Initially had null pointer from not covering the different cases in deletion. Fixed that by have a set of if/elseif/else statements going through each possible case in the deletion process.

Another issue I ran into was regarding design choice for how I wanted to construct the friends linked list and the linked list at each index of the hash table. You did say we could use a variable string array, but I didn't go with that since it can produce ArrayOutofBoundsErrors if the person has enough friends in their friends list. I considered having two different parameters for the Node inner class – one for String, the other for Person. One problem with this is there is a lot of repeat code. Since I had to make each method in my linked list work for each type of Node so I basically had two versions of each method in the class. It made more sense in the end to just have type Person for both, shortening the code by a lot and I can still pull the name out from each Person object in the friends linked list anyway.

There was a major problem with the user interface I that I came across. I used a Scanner type interface where the user would enter inputs and those inputs would be fed into if statements that called certain commands. I know what the source of the problem is, but after a few hours, I could not fix it. The problem was that even though the user input was stored in a variable for example, String name1 = scanner.nextLine(), it would actually return a null value instead of the input value. As a result, the functions I call using the input parameter name1 would just cause a nullpointerexception since the name is null no matter what. Calling the functions with a string directly causes no issues. This leads me to believe there is a confliction with the Scanner class that can't be avoided.

Sorting with BSTSort more than once leads to a massive bug that repeats printed characters endlessly. I started from the ground up testing, treeInsert() first all the way up to BSTSort() to see where in particular the bug was coming from. After many attempts, I could not figure out the source of the bug.

6. Lessons Learned:

I relearned how to make a linked list class from scratch. It didn't take long since I did learn it before, but it's been a while since I've done it, so I had to look back at the pseudocode in our textbook for that data structure.

I’ve learned how to construct a BST. I wasn’t sure how to do it initially since it didn’t seem like the book ever mentioned how to construct one with a given list, but then I thought back to how we made the heap by just calling heap-insert on all the elements from our PA1 and did the same for the BST. I also learned how to write the methods for sorting a binary search tree from scratch. It takes me back to PA1, where we have like five different functions that lead up to the final sorting algorithm. This gives me inspiration in learning more complex algorithms because even if they look unapproachable at first, if I just break them down into multiple series of simple steps, it becomes much easier to understand.

I’ve gotten better at designing software and seeing the bigger picture where all the classes and how the main function that makes up the user interface all come together. I’m getting better at organizing where my methods should go.

Usually once I turn in an assignment, I consider it finished and drop it, but seeing what more I can add to this program makes me want to continue working on it in the future. Working on clones of existing applications that people actually use is really fun since there’s usually a lot of room to add functionality. I can also look at the original and try to implement some of their functions.

I’ve learned how to implement hashing in a realistic application. I understand it conceptually from reading the power point slides, but actually implementing one of the hashing methods into a fleshed-out program helped solidify my understanding of it.