**Leo Le - Programming assignment 4**

**Table of Contents**

[**How to use / Basic tests**](#_3fbnab8rnlwa) **2**

[**Design + Implementation**](#_riy3sa1qzq3k) **7**

[Person class](#_o1b0mm17p7bi) 8

[FriendsList & TableList classes](#_u9c2af98irmn) 10

[HashTable class](#_9pacemuyy6uz) 11

[Program class](#_y0si9h9mc6xk) 12

[**Lessons Learned**](#_58wn2d18tcv9) **17**

[**Problems Encountered**](#_hl4p1nq9drcx) **17**

[**More Tests**](#_1tgsubelijtx) **18**

[new hash function building tests (p.263)](#_o2q2e4b5p8kt) 25

[Miscellaneous tests - Testing with OLD hash function / unfixed bug. You can ignore these.](#_gvtud35z3o55) 31

**File download**

<https://drive.google.com/open?id=1EZxPxKzxBpXHzvRLa3Mn7yTUCXjSp8jU>

#### 

#### 

#### 

#### 

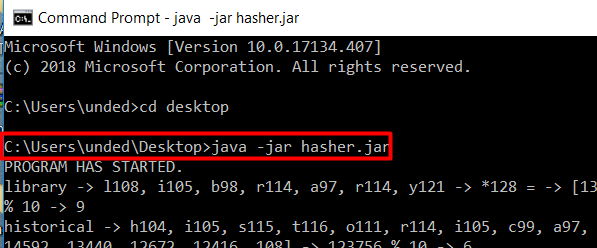
#### 

#### 

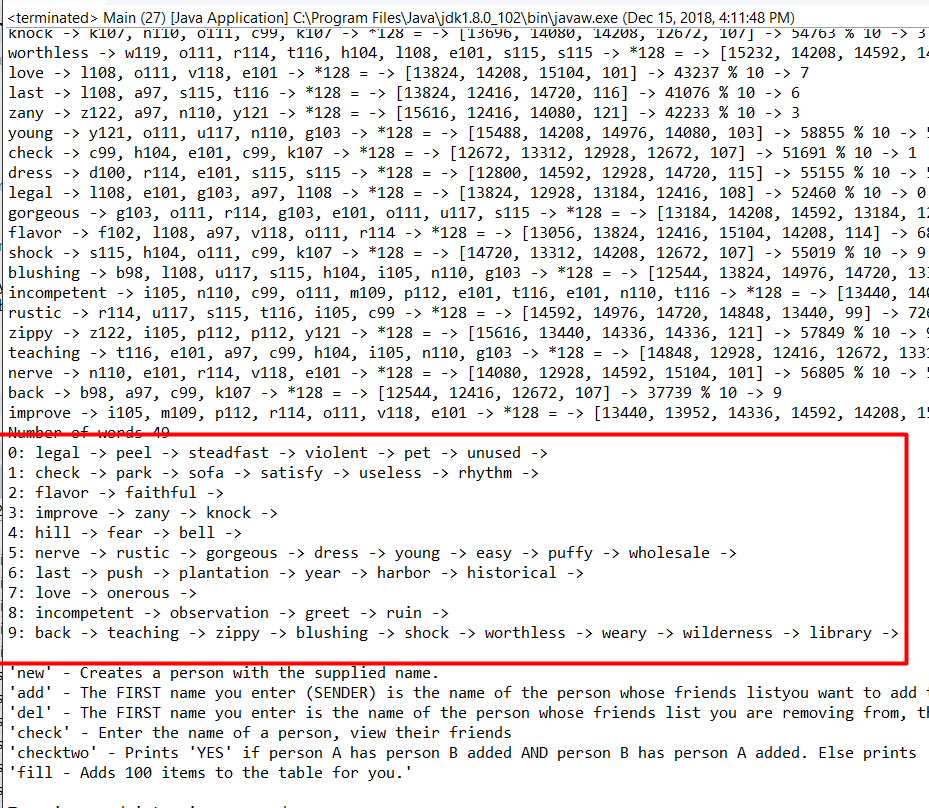
#### 

#### How to use / Basic tests

1. Extract the files (With winrar, right click the .zip and click “extract here”)
2. Open command prompt
3. Navigate to directory where hasher.jar is. I held it in my desktop
4. Type “**java -jar hasher.jar**”
5. The program will start in your command prompt

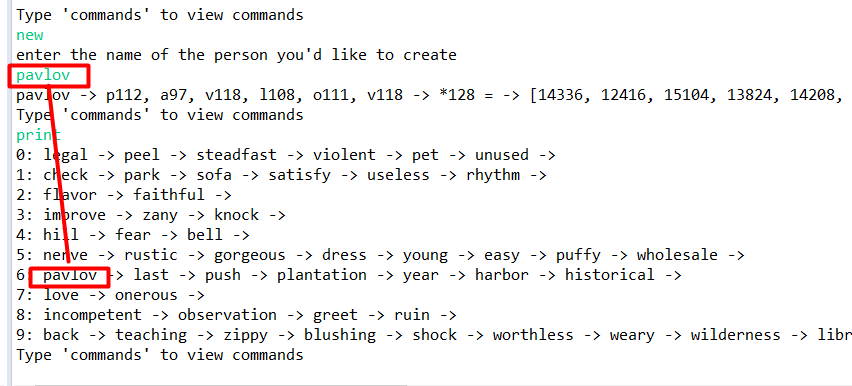


You start off with 49 items in the list of size 10. This is to test collisions and situations in the middle of a list.



^The words highlighted in red are the information inside the Hash Table (names of Person objects).

To add an item to the list, type in “**new**”, and then the person’s name. Type “**print**” to view your results.

****

To add to friends list, pick a user. Lets pick from list 6, the user named “last”



Let’s add “push”,” rustic”, and “year” to “last”’s friends list.



Type “**add**”

Type “**last**” → selects user “last”’s friends list.

Type “**push**” → adds “push” to “last”’s friends list.

Now “push” is added to “last”’s friends list.

Repeat with “rustic” and “year”.

Type “**add**”

Type “**last**”

Type “**rustic**”

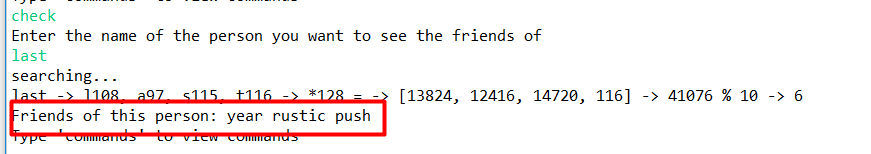
Type “**add**”

Type “**last**”

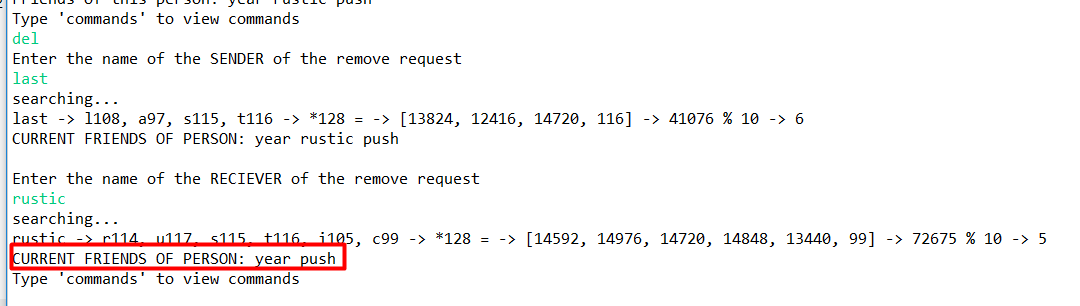
Type “**year**”

Now push, rustic, and year are all a part of last’s friends list

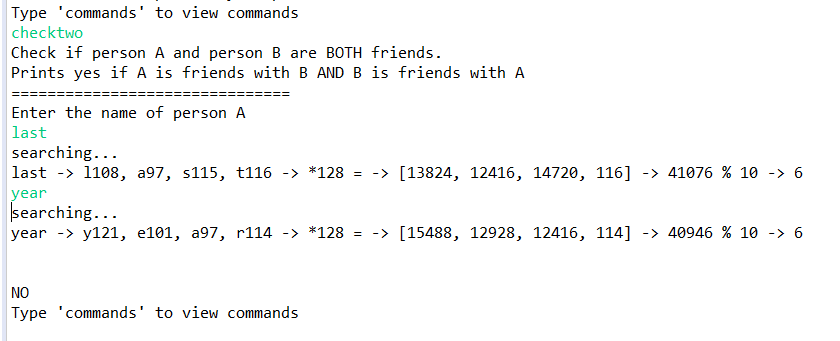
To check “last”’s friends, type “**check**”, then “**last**”



To remove a friend, type “**del**”. We want to delete “rustic” from “last”’s friends list, so after you’ve typed “del”, type “**last**” to select “last”’s friends list, then type “**rustic**” to delete “rustic” from last’s friends list.

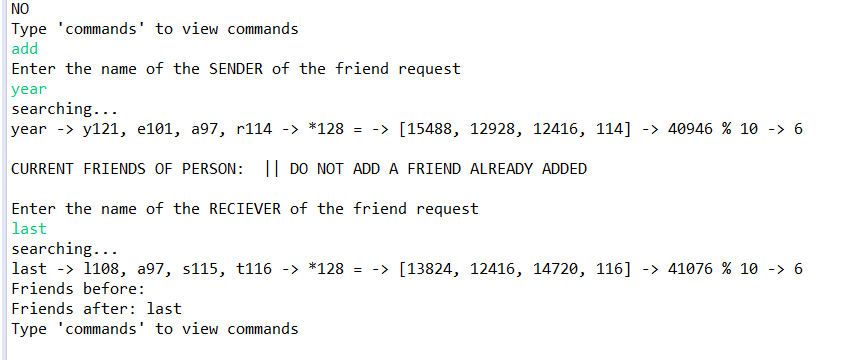


To check if two people are friends with one another, that is, if A is friends with B AND if B is friends with A, type “**checktwo**”. Enter the names of the two people you want to check.

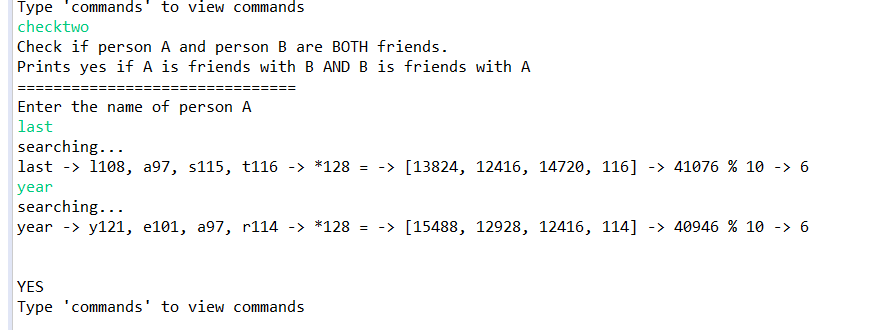


It prints “NO” because “last” is friends with “year” BUT “year” is NOT friends with “last”.

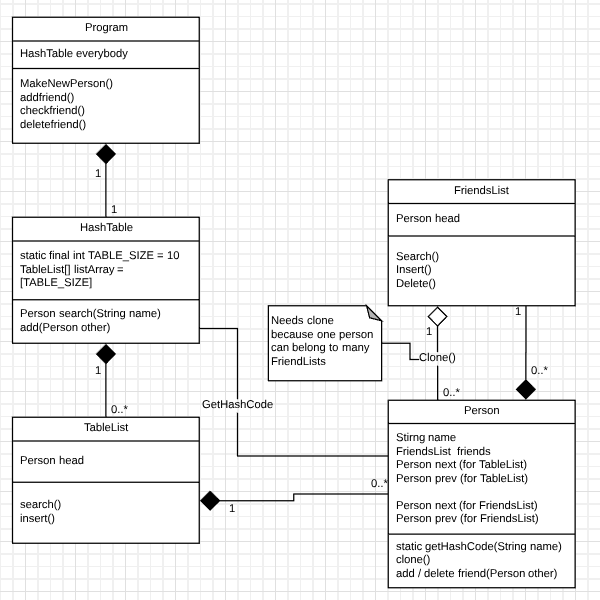
Make it print yes but adding “last” to “year”’s friends list, as shown below.



Now type “**checktwo**” and type “**last**” and “**year**” again



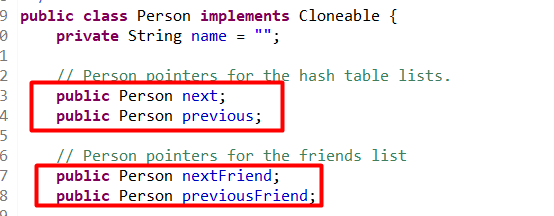
#### Design + Implementation



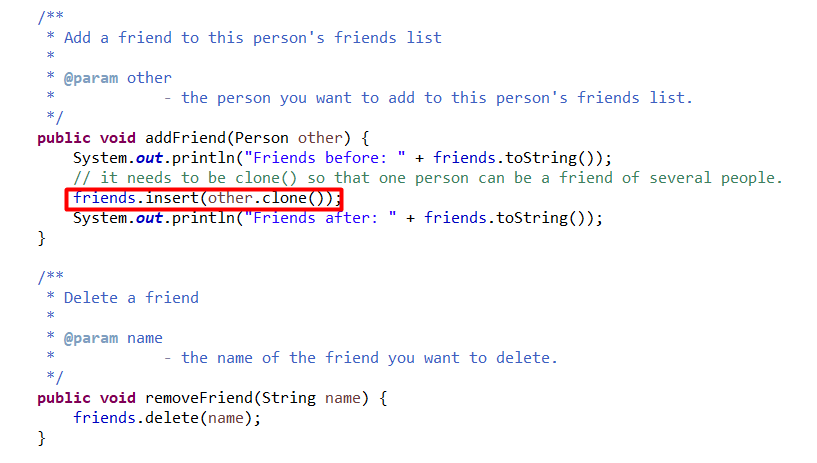
Program takes user input. Program has a HashTable, which has an array of TableLists. These TableLists are linked lists, so collisions are handled. Each linked list stores Person Objects. Therefore, the HashTable stores all the Person objects that have been created. When the user wants to tell HashTable to search in / add to the hash table, HashTable calls a Person.HashCode() method to get the hashcode (slot) where the person should be, and takes the linkedlist at that slot to search for the person within the linked list.

##### **Person** c**lass**

Role in the implementation: It stores the name variable for people who make accounts. It has a linked list of Person objects which it uses to keep track of its friends. It is stored in TableLists inside the HashTable and other people’s friends lists. It has two sets of prev/next pointers because it can be a part of TableList from HashTable and FriendsList from other Persons.

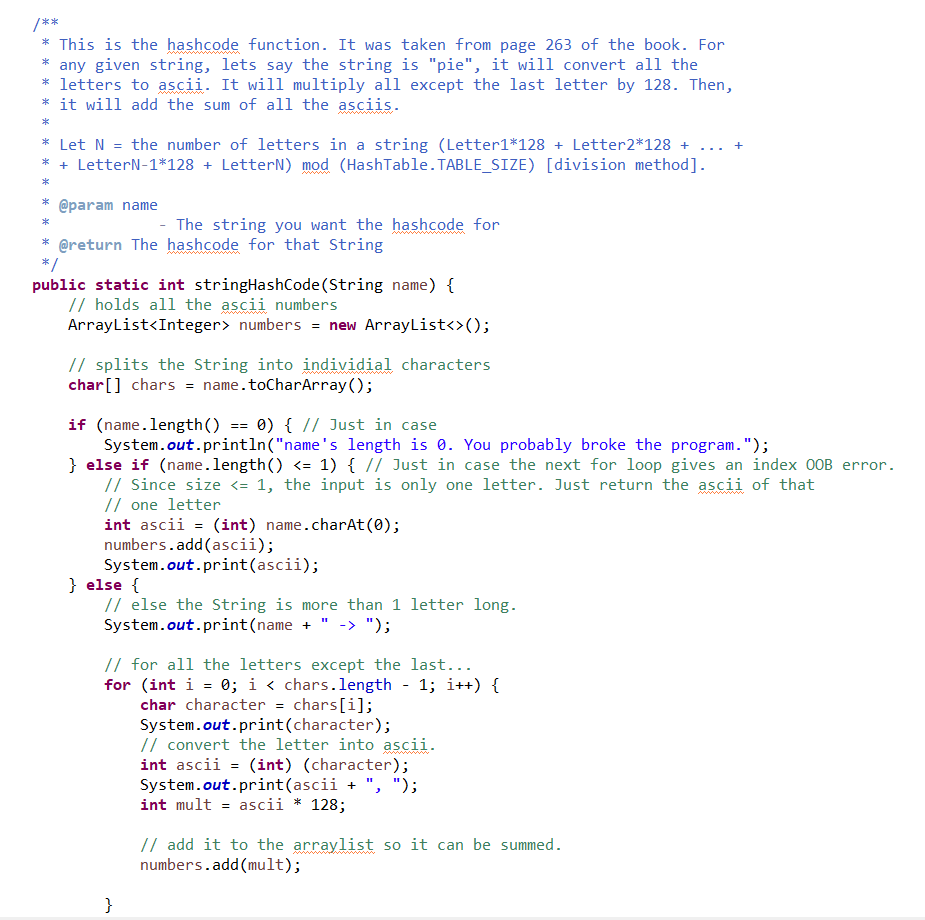
****

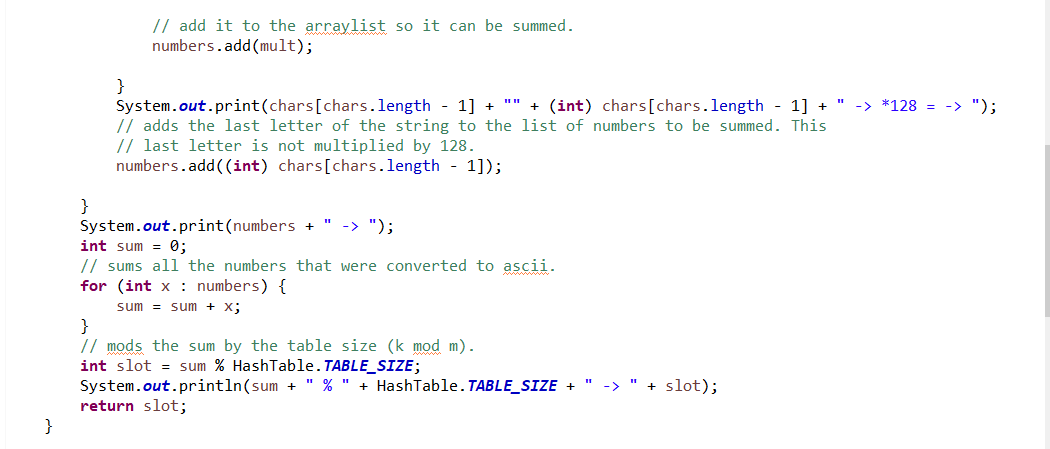
Individual Person objects hold the methods to delete their friends and add new ones to their own friends list. When a Person adds a friend to their list, they add a cloned Person object. This is because one person can be a part of several friends lists, but each Person object only has one set of nextFriend / PreviousFriend pointers.



,

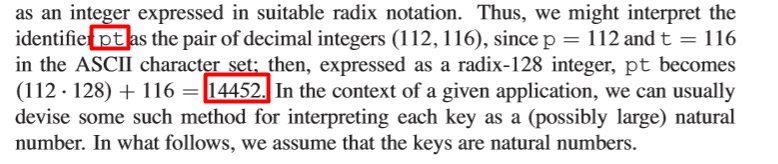
The blue comment above the method explains how the hashcode function works.



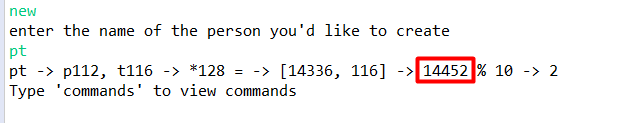


^In the pie example, we would get pie = [112, 105, 101]. All the integers except the last one are 112 and 105. So the function would do 112\*128 + 105\*128 + 101 = 278277. 278277 % TABLE\_SIZE = 278277 % 10 = 7. Therefore, “pie” goes in the linked list at 7th slot.

In the book example, they use “pt”.



If we calculate the hashcode for pt, we get the same thing

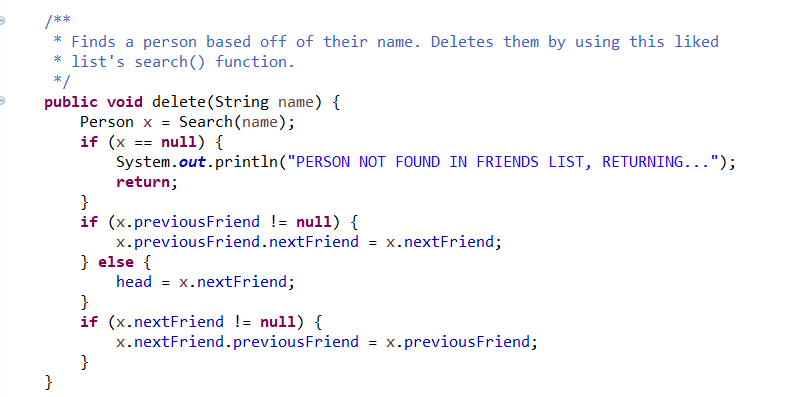


##### **FriendsList & TableList** c**lasses**

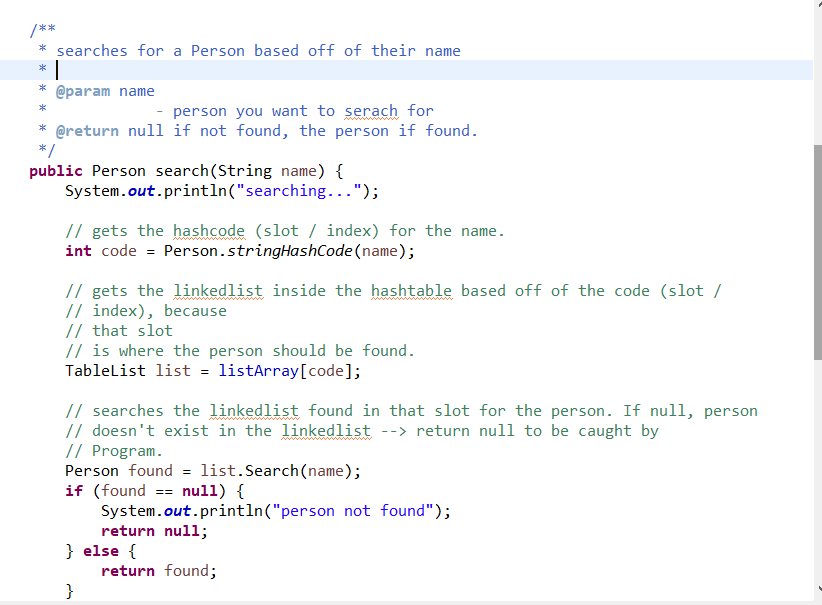
Both these classes are linked lists written using the book’s code.

Role in implementation: FriendsList is used by Person objects to keep track of their friends. TableList is used by the HashTable to deal with collisions among Person objects being stored inside it. HashTables store TableLists, but People have a FriendsList.

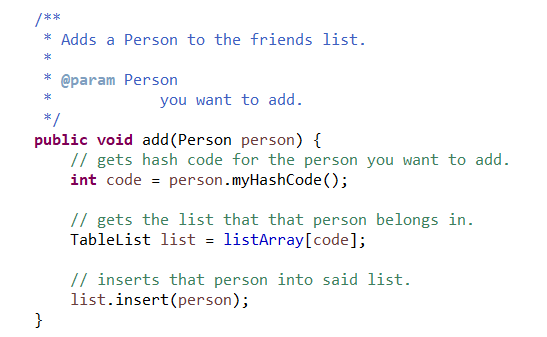
The only notable change I made to the code was in the delete method for FriendsList. It deletes based off of name by using the Search() method inside its own class. It will return null if nobody is found, which will be caught by the Program class to inform the user



##### **HashTable class**

Role in implementation: It holds an array of TableList objects. TableList objects hold Person objects. Therefore, this class holds all the Person objects. The TableLists are linked lists used to deal with collisions. It picks which slot within the array to add to / get information from by using Person.stringHashCode(String name).

Adding a Person to the list of Persons is done in a similar way than search. Use Person’s static hashcode function to pick the slot the user be added to. Then search uses the same hashcode function to calculate the slot the Person should be at when you want to pull information.

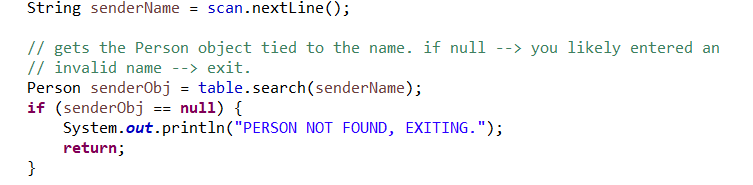


##### 

##### Program class

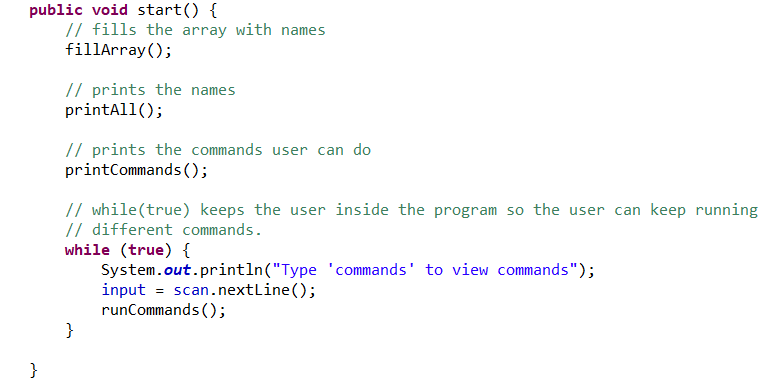
Role in implementation: Handles all the user input. Stores the HashTable. HashTable stores all the Person objects. User enters which command they want to perform, and how they want to perform it, then the appropriate methods are called.

**Important note:** Whenever you see code that looks like this...

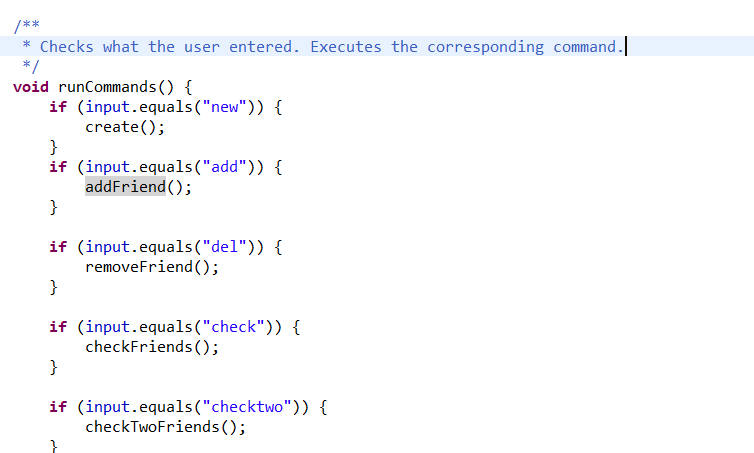


It just means “check if the person exists within the HashTable / FriendsList”. If a person is not found, then their Person pointer will be null, so the program will exit executing that command.

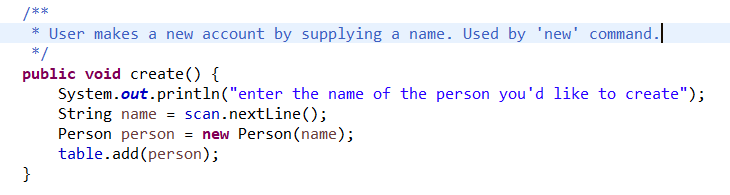
When the program starts, it will be thrown into an infinite while loop.



The infinite while loop will prompt you for user input. The user input is checked by runCommands(). If you enter a valid user input, which I refer to as “command”, then it will execute the appropriate method to perform your action, then it will start from the beginning of the while loop when it is done. Else, if the input is not a valid command, it will go through all the if() statements in runCommands(), not execute anything, and restart at the top of the while loop anyways.



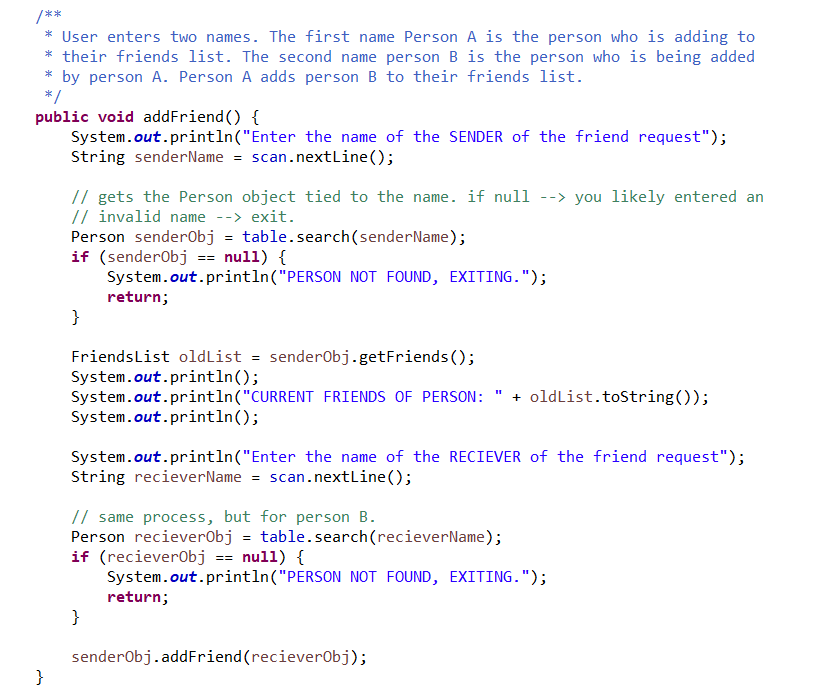
For example, if the user types “**new**”, this method will run



^It will prompt the user to enter a name, and a person will be added to the HashTable based off of their name.

To add a friend, we use HashTable’s search(String name) function to get a reference to the Person object we want to add to. The Person object will have their friends list attached.

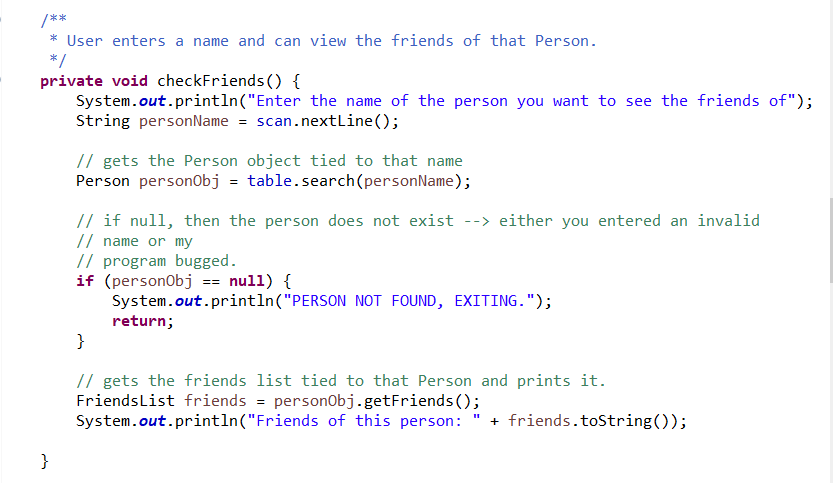
To pick which friend we want to add, the user enters their name. We follow the same process of using the hash function to get the table slot the person should be at, then get a reference to that Person object to add to the friends list. I make a clone of the person to add to the friends list instead of making a new Person() object because if we want to upgrade Person to have more data than name in the future, we can just clone them. Also because we need to check if the person exists in Facebook before we add them to a friends list, and if we get returned null by the search function, then we know that they don’t exist.



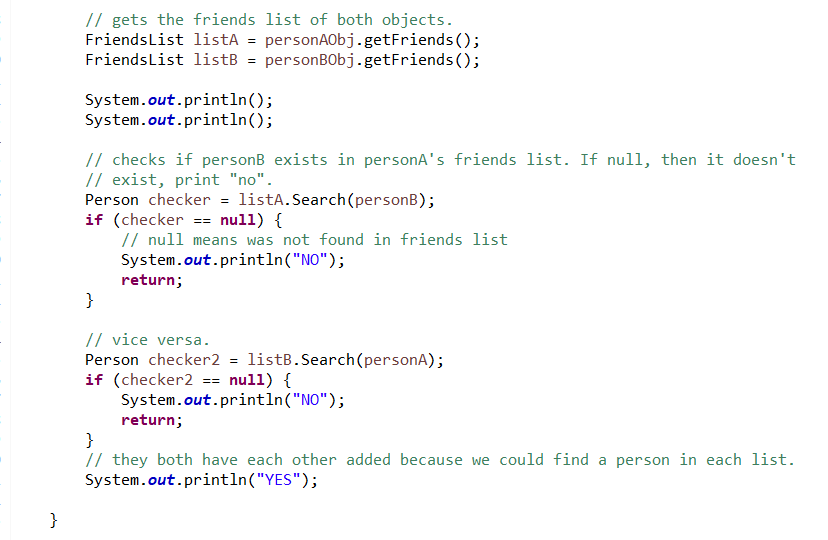
removeFriend() method follows the same pattern, except instead of calling “senderObj.addFriend(recieverObj);”, it calls “senderObj.removeFriend(recieverName);”.

If user does not exist in the friends list, it will be null, so the program will exit.

checkFriends() method follows the same process of getting a reference to the Person object. Then, it pulls the FriendsList object of the Person and prints all their friends.



checkTwoFriends() follows the same process as checkFriends(), but instead of pulling one FriendsList object, it pulls the FriendsList of both Persons and checks if Person A is in Person B’s FriendsList, then if Person B is in Person A’s FriendsList.



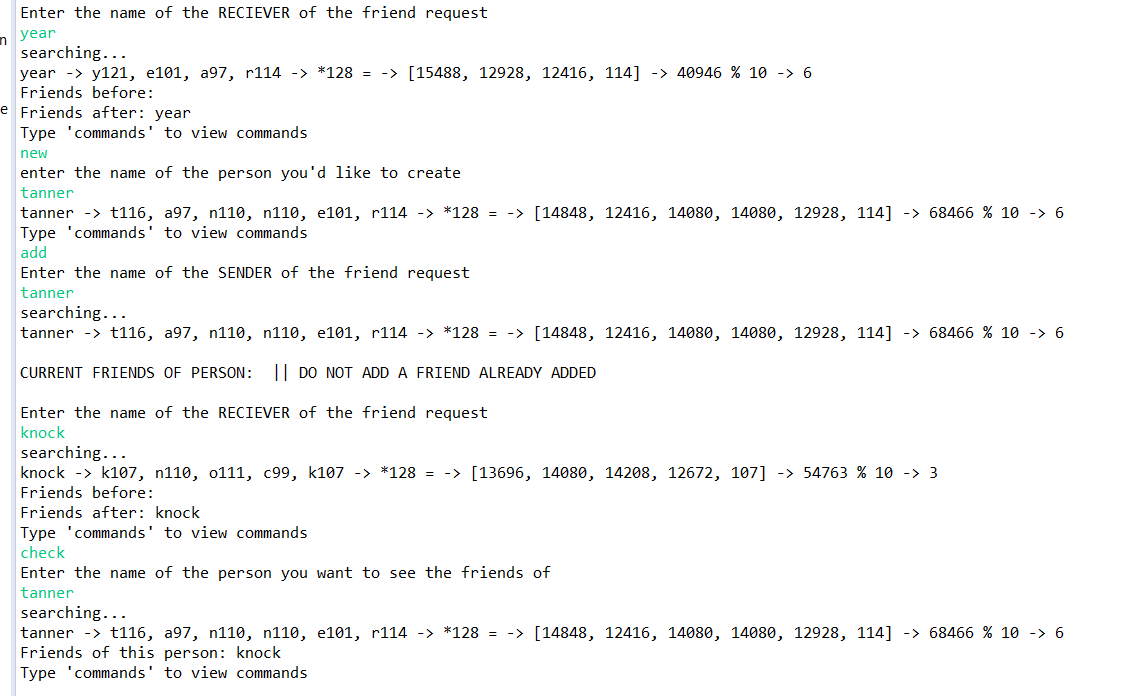
#### Lessons Learned

* How to implement a linked list to handle collisions within a hash table through the chaining method instead of using probing techniques.
* Implementing a hash function with an array so that strings, through the hash function, can be stored and retrieved through the hash function calculation
  + A good hash function that has very little collisions with a hash table at least as big as your user base would probably be a very fast way to access data.

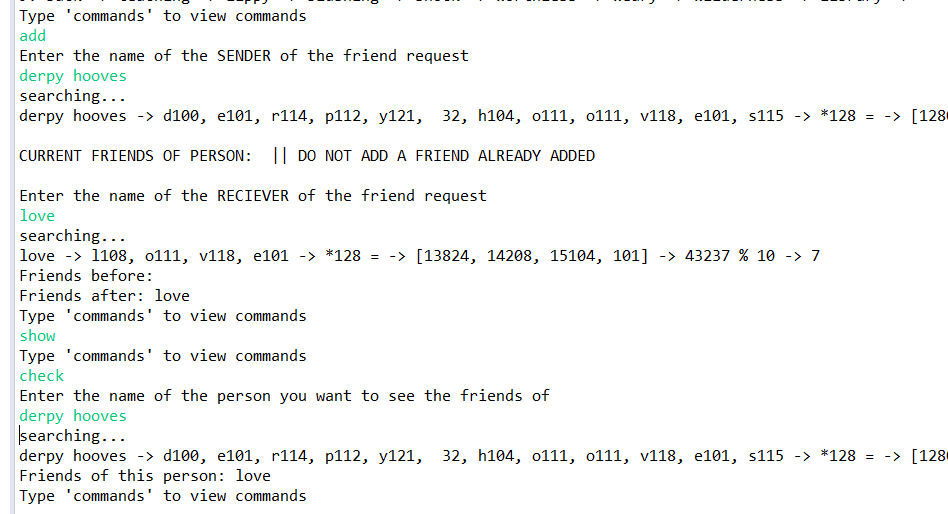
#### Problems Encountered

* There was a bug where if you added a person to the middle of a linked list in the HashTable, and then added them to a linked list in the friends list, then their pointers would change because of shared pointers between the two list classes. I fixed that by making the TableList and FriendsList classes, then made two sets of pointers for each Person, one set of pointers for FriendsList to use and another set of pointers for TableList to use.
  + The same bug happened, except this time, if person A had a friends list of (van, bad, volcano), and person B added “bad” to their friends list, volcano and everything after would be deleted from the friends list of person A. This bug happened because one Person needs to be able to be the friend of several people, but each person only had one set of friend pointers. To fix this, when I add a Person to the FriendsList of another Person, I add a clone. When I delete a friend, I search for that clone.
    - Because of the new clone implementation, I had to change FriendsList’s delete method to delete by name instead of by Person object.

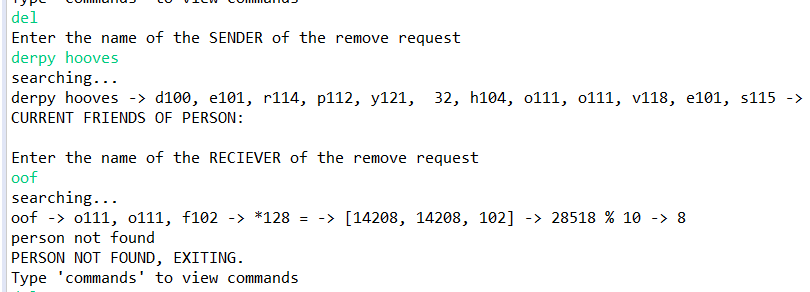
#### More Tests



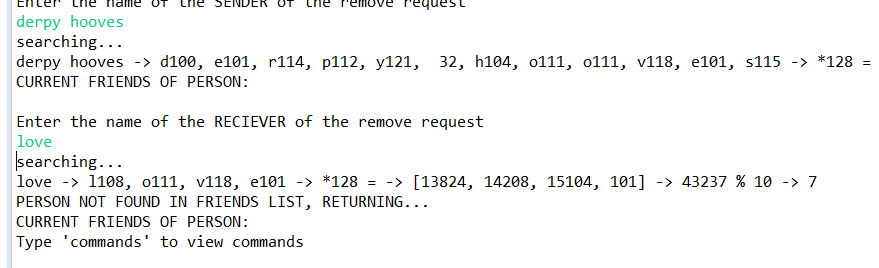
^testing how to handle collisions if two Person with the name “tanner” are added to the hash table. It will only manipulate the most recently added one. It can handle collisions under an accidental add (won’t crash), but I am assuming there will be none.



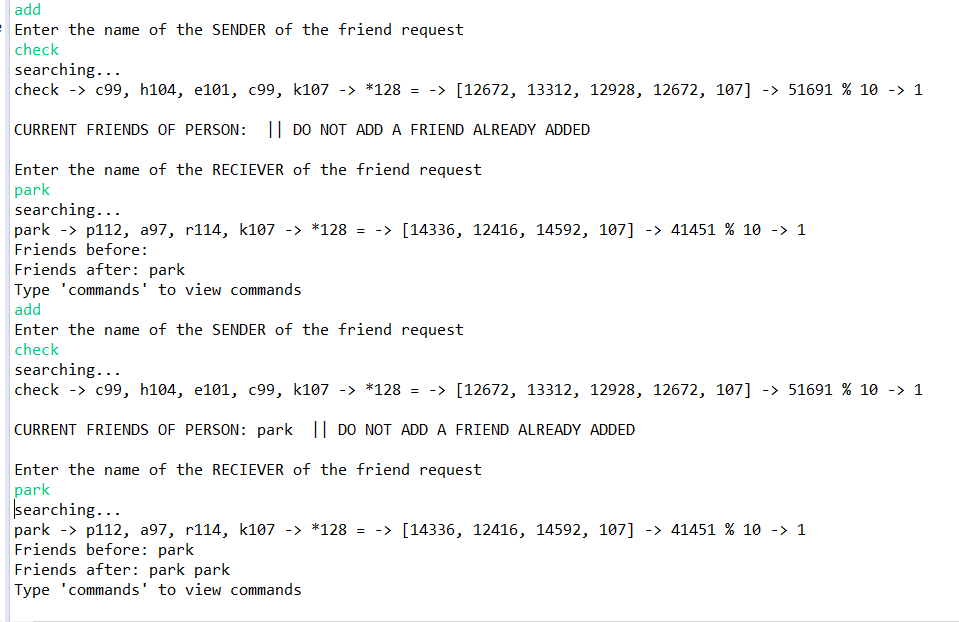
^I’ve been using the default provided names. Making sure the “**new**” command and adding onto Person objects created by “**new**” works by adding a Person named “derpy hooves” and adding friends to and removing friends from that person.



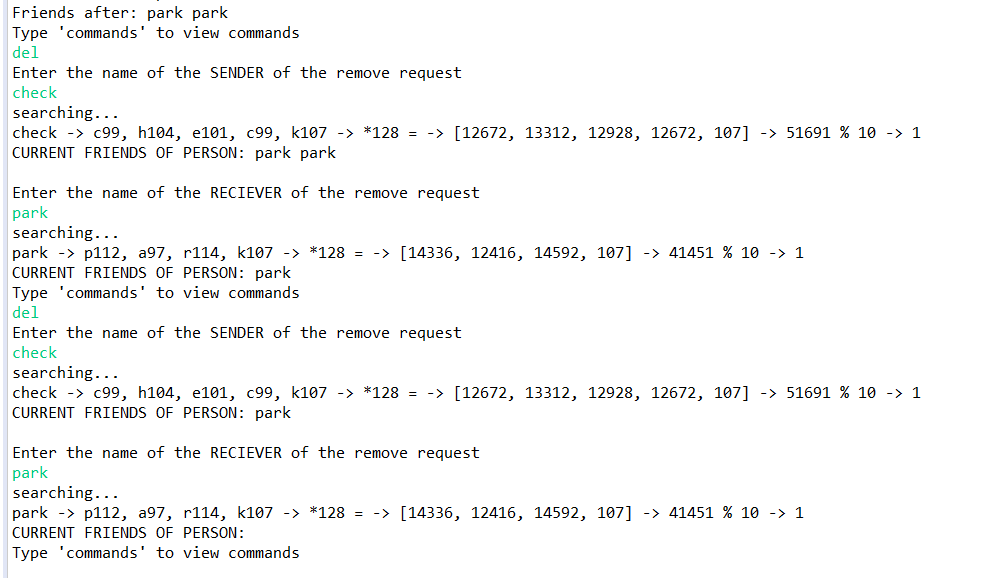
^testing exit if we take a valid Person, but try to delete a name that doesn’t exist



^testing what would happen if you try to delete a person that doesn’t exist in their friends list



^testing that would happen if you try to add the same friend twice. They appear twice in your friends list. To prevent this from happening, I would need to either search through the list for whether it already exists in O(n) time or write a hash table for the friends list instead of just using a linked list. The reason this happens if because a clone of the person is added to the friends list. Previously, without cloning, if you tried to add the same person twice, it would be stuck in an infinite loop.

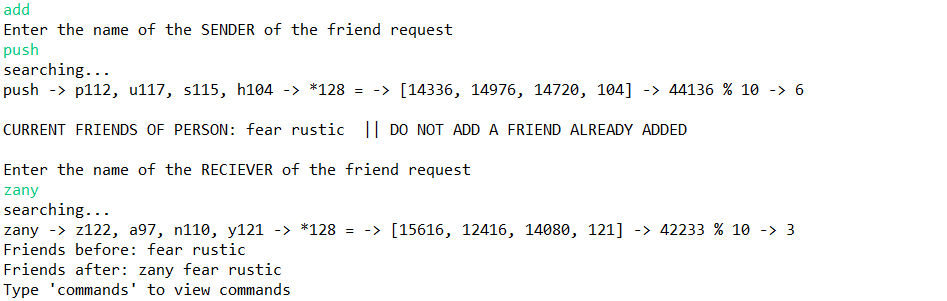


^you can delete both instances of the same friend that you created from the list

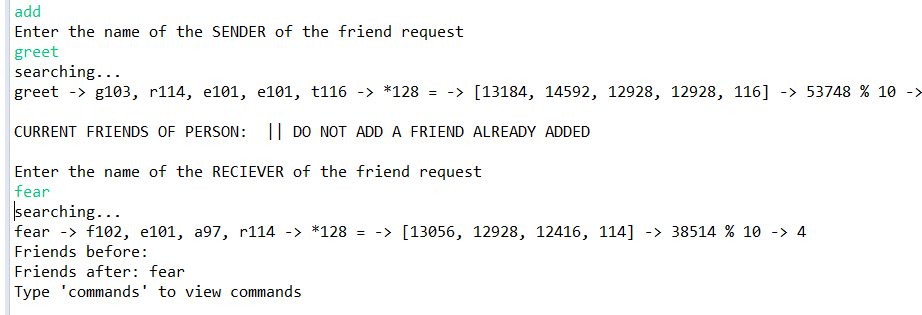
========================

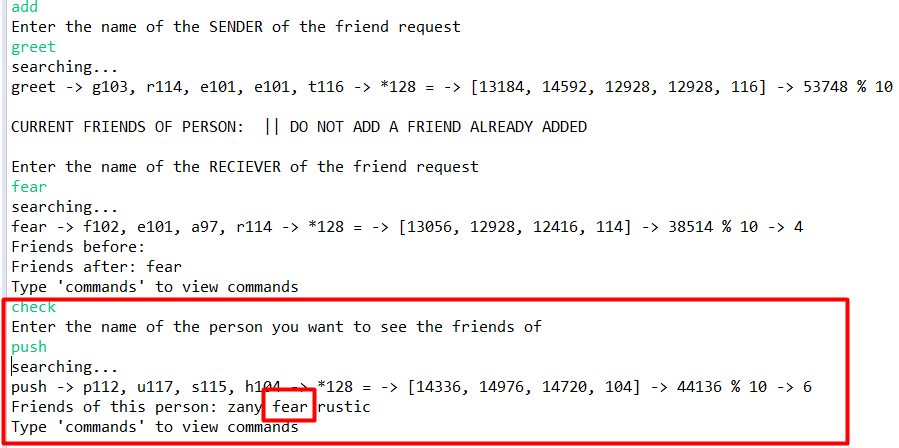
Below are a set of screenshots for one test

First, for user named “push”’s friends list, add “zany”, “fear”, and “rustic”.



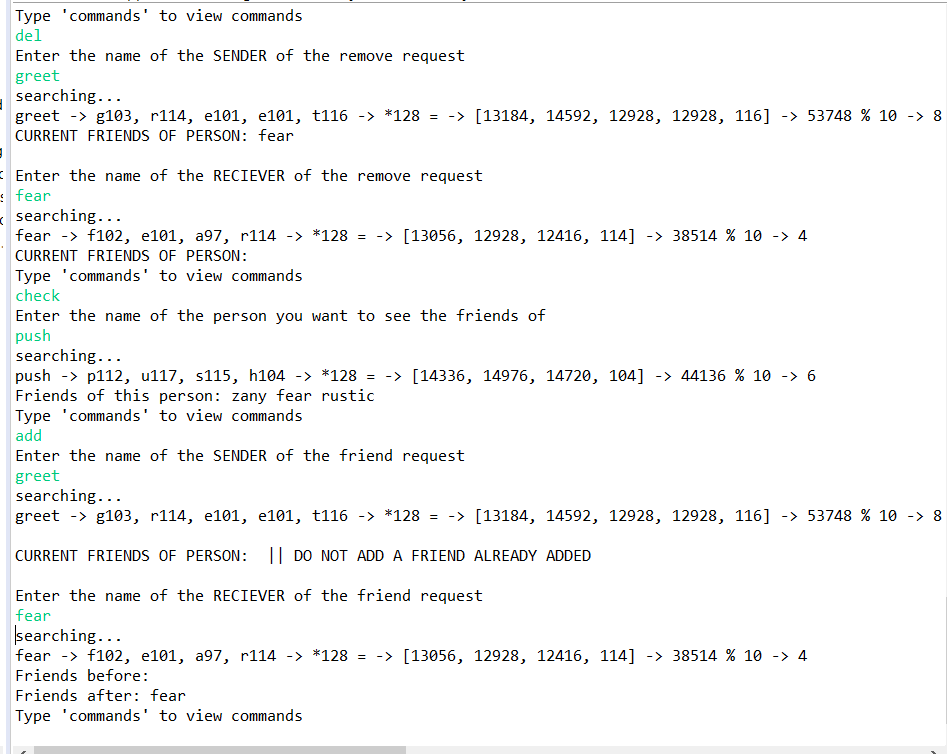
Next, to “greet”’s friends list, add “fear”, since “fear” is in the middle of “push”’s friends list. I am testing to see if manipulating “fear”’s pointers will manipulate its placement in “push”’s friends list. It should not because of my adding a clone implementation.

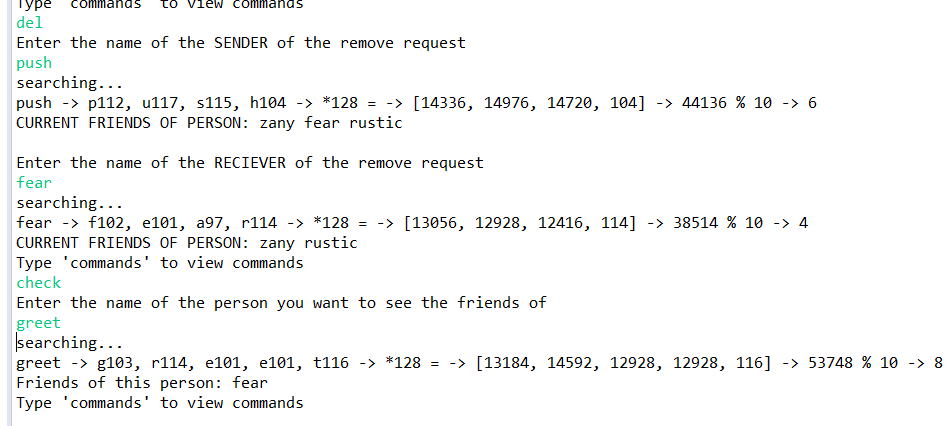




^This means that one Person can be the friend of several people.

Just to be safe, test it with remove. Remove “fear” from “greet”’s friends list. Then, check “push”’s friends list to make sure nothing happened with the “fear” in that other list. Then add it back for testing it the other way around. Delete “fear” from “push”’s friends list and make sure “fear” is still in “greet”’s friends list.

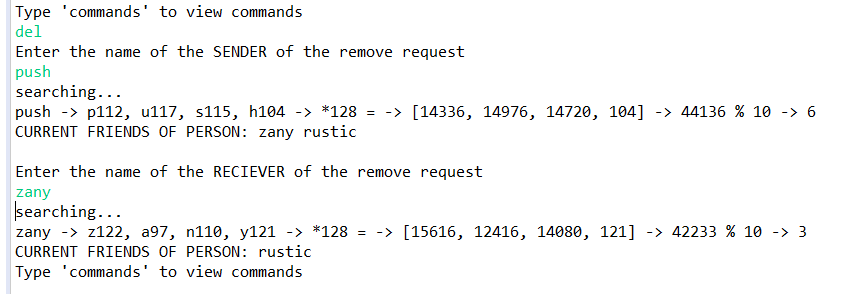




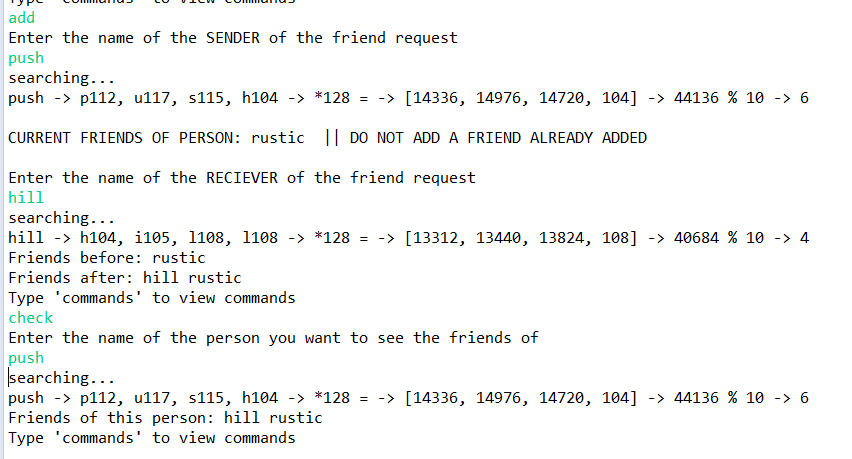
^This test shows that delete is safe even though “fear” belonged to two friends lists. It also shows an important case of the linked lists, that I can delete from the middle.

=================================

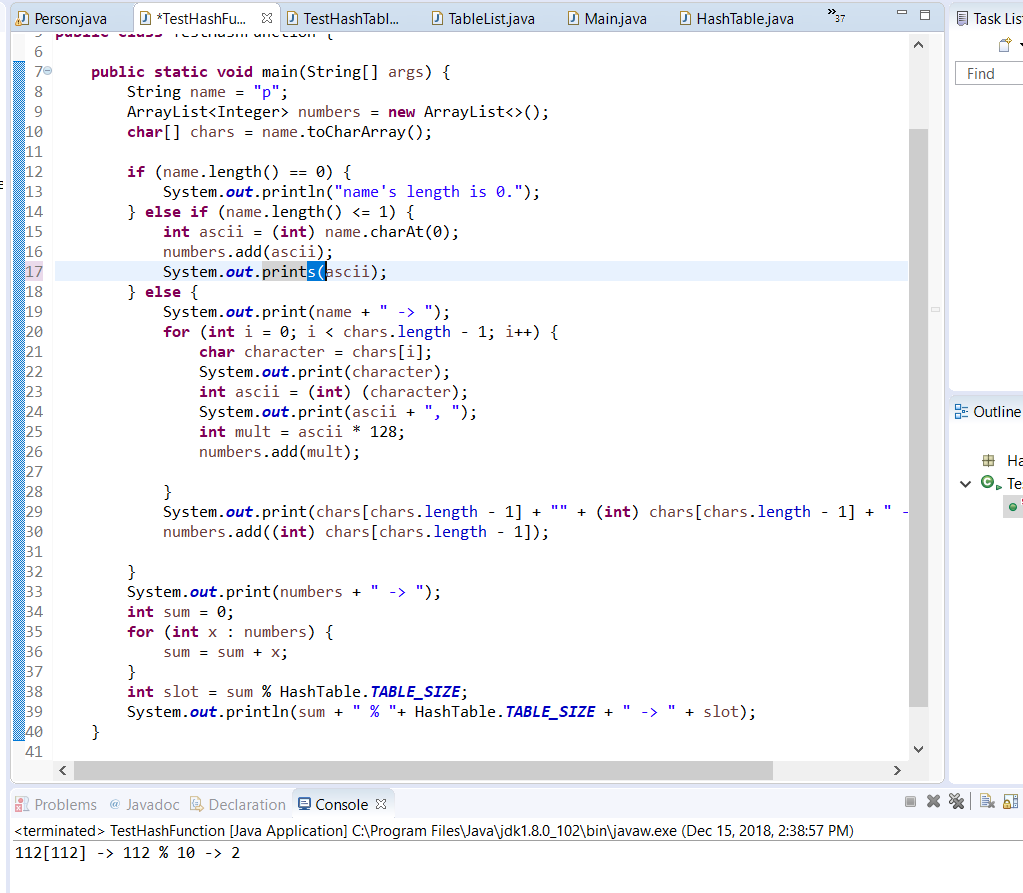
In the previous set of test, I deleted a friend from the middle of a list. Now I want to delete from the beginning of the same list, delete “zany” from “push”’s friends list.



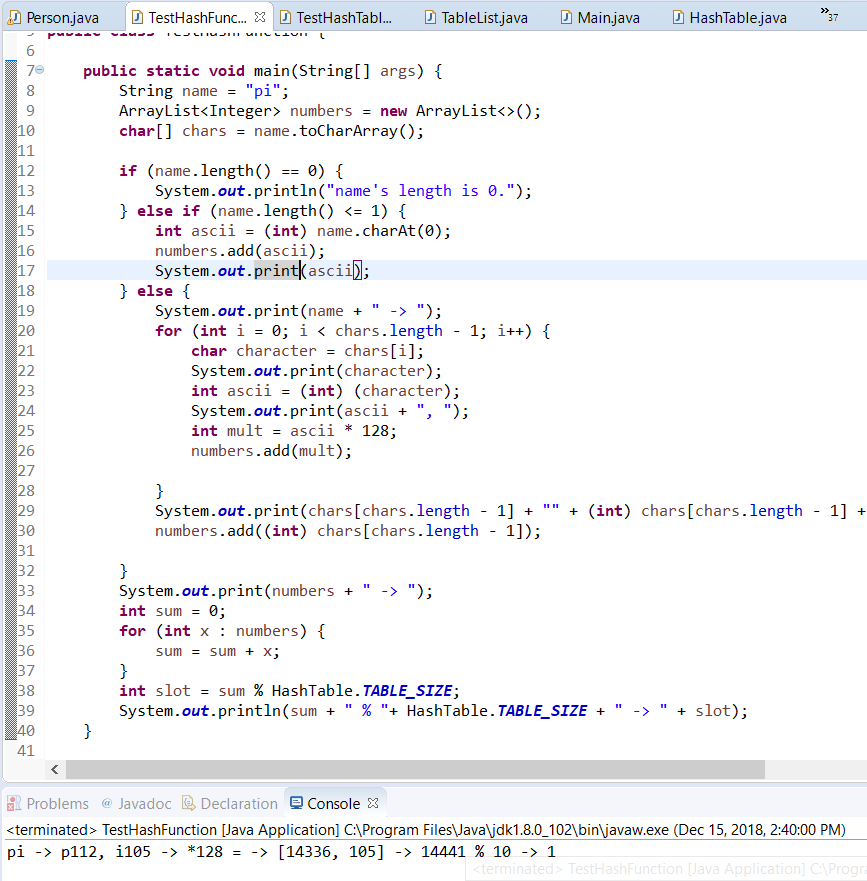
I want to make sure the list isn’t corrupted, so add back to it.



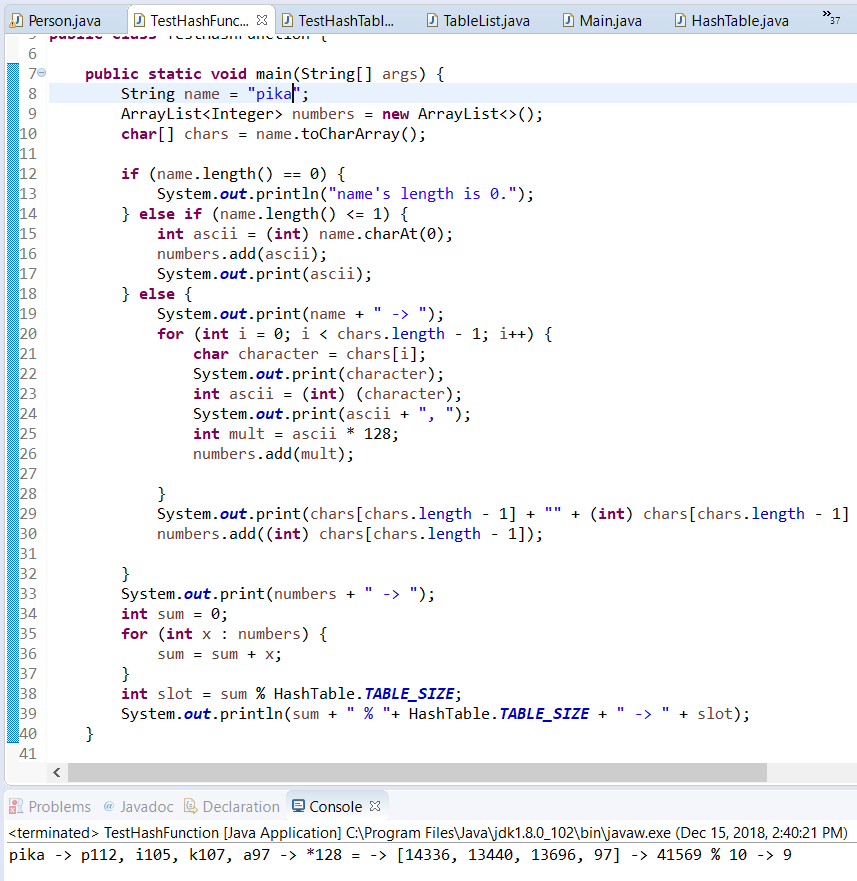
##### newh**ash function building tests (p.263)**



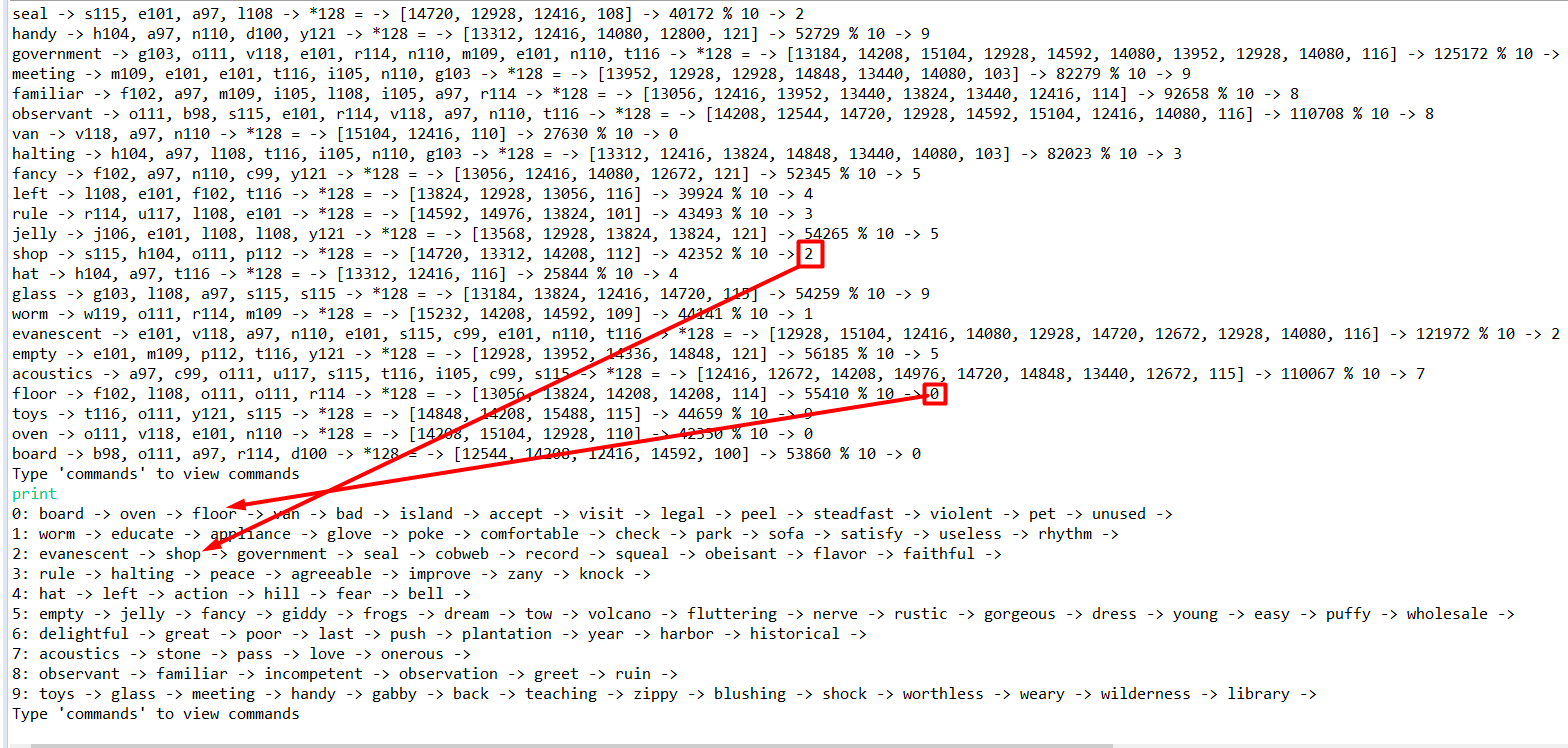
^testing with one letter, p



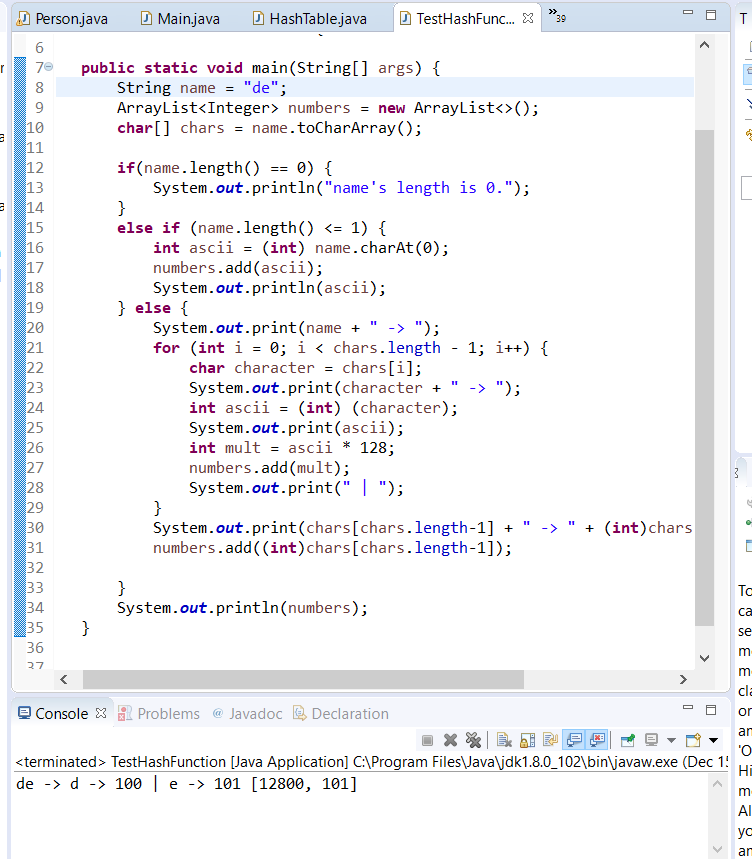
^testing with two letters, pi

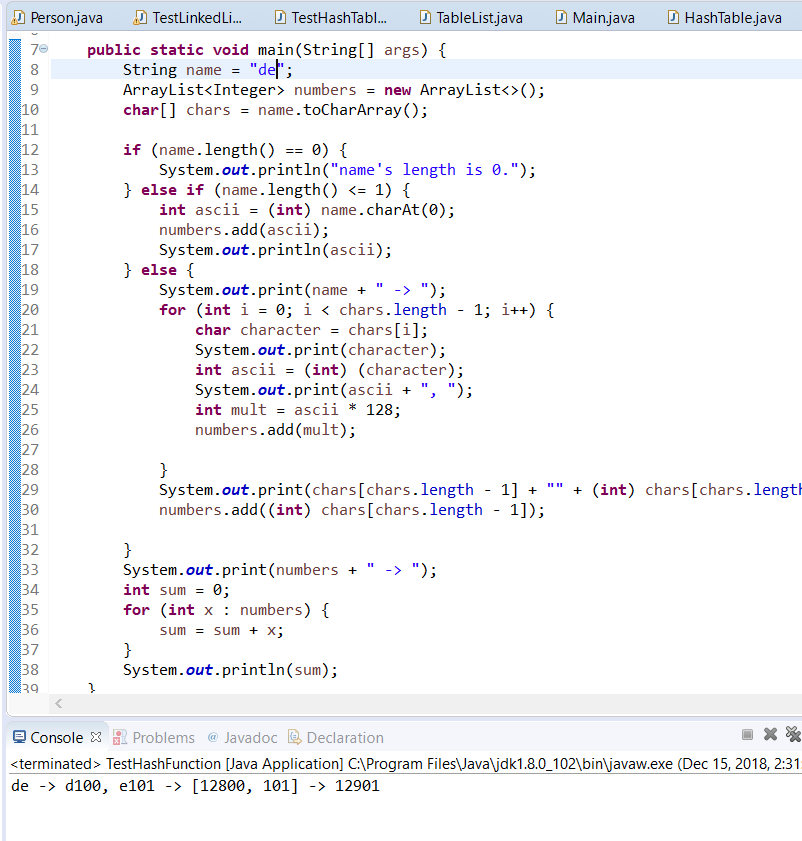


^testing with 4 letters, pika

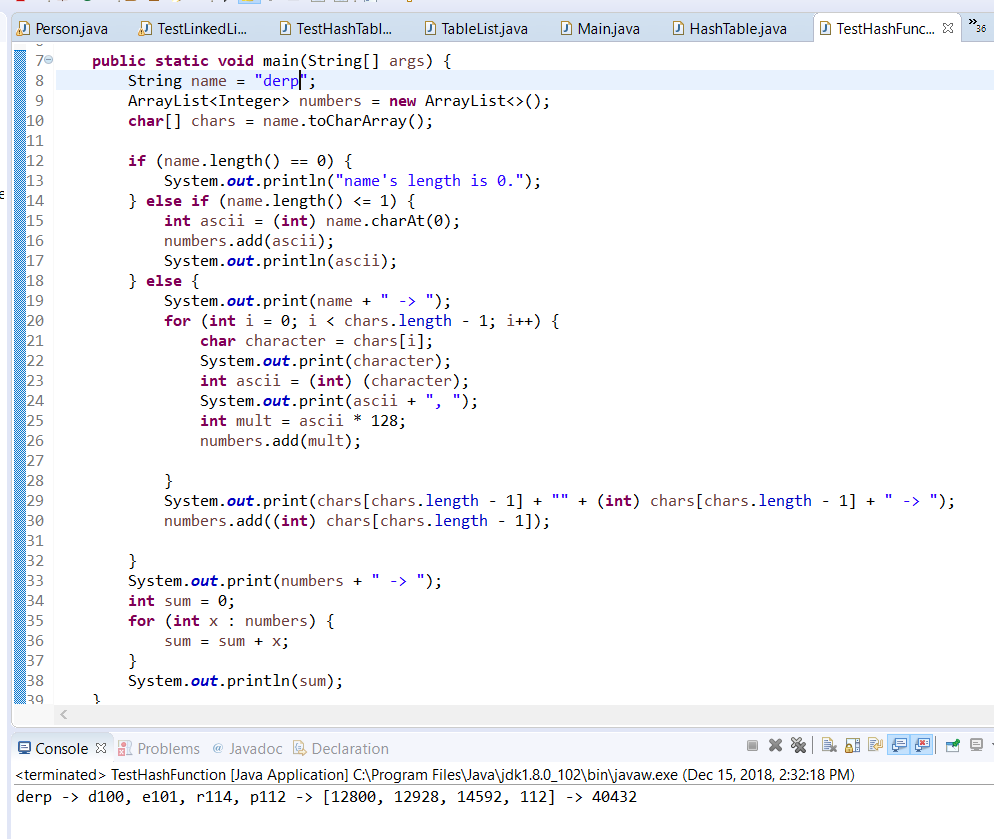


^filling the array with 100 random words and not getting a thrown exception.





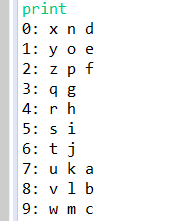
^testing with de



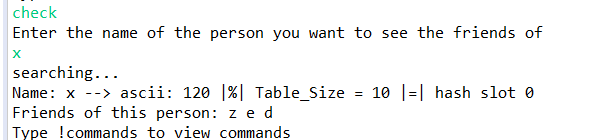
^testing with derp

##### Miscellaneous tests - **Testing with OLD hash function / unfixed bug**. You can ignore these.

Other Tests

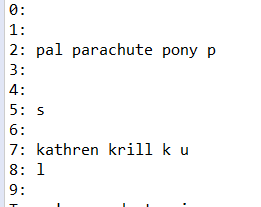


To x’s friends list, I add the letters z, e , and d

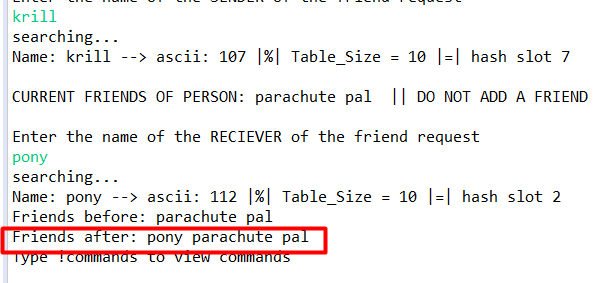


I want to make sure my friends list can

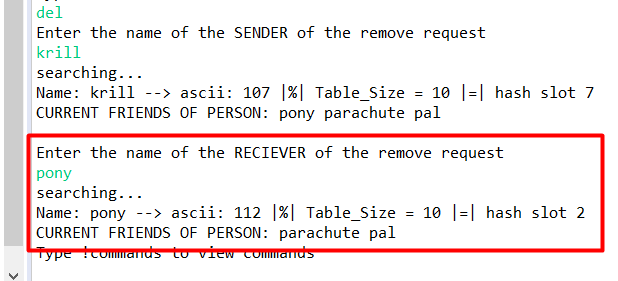
TESTING FRIENDS LIST



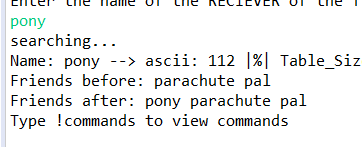
Take krill and to krill’s friends list, add pal, parachute, and pony to his friends list



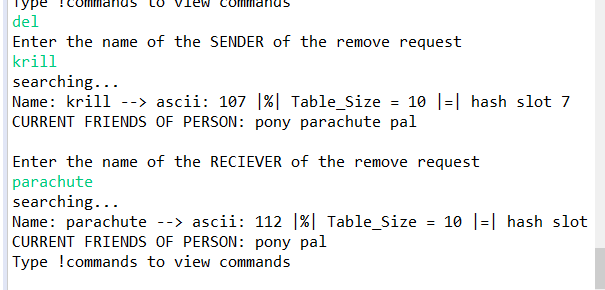
Test if from krill’s friends list, delete first element, pony



Re-add pony to the friends list

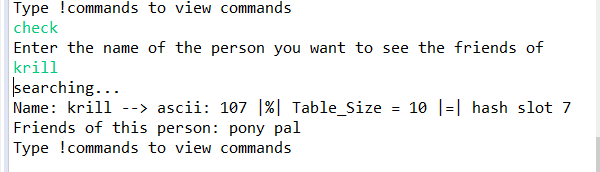


Now delete from the middle, delete “parachute”

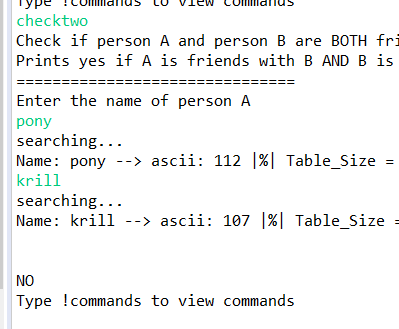


Now it is verified that we can delete friends from the first and middle.

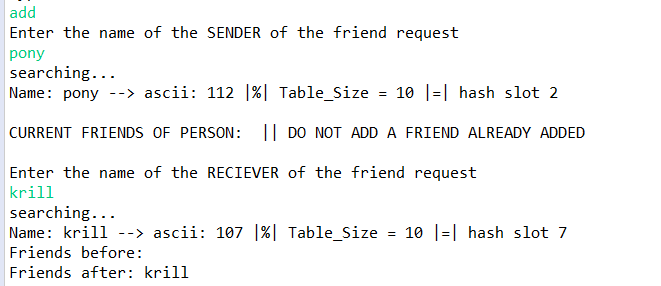
This is currently krill’s friend list



Try checktwo with pony and krill, you will get “NO” because pony does not have krill added to pony’s friends list. Only krill has pony added.

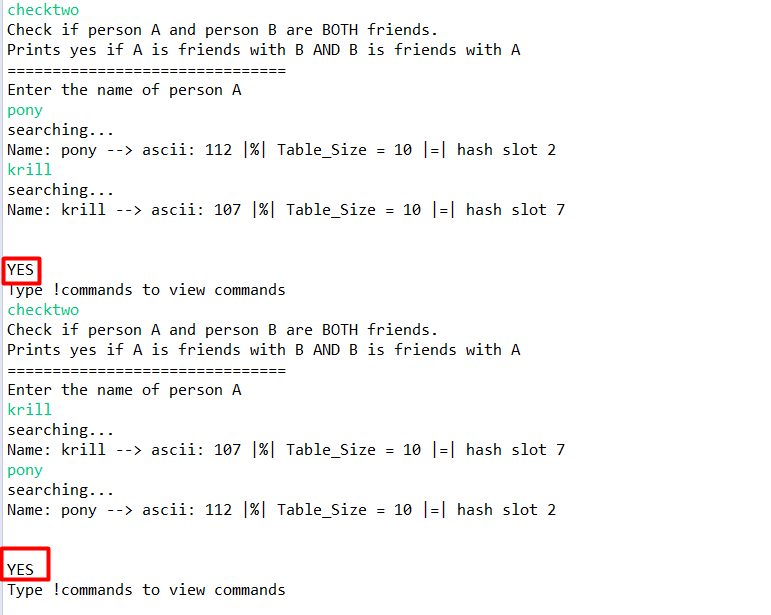


So lets change that by getting pony’s friends list and adding krill to it

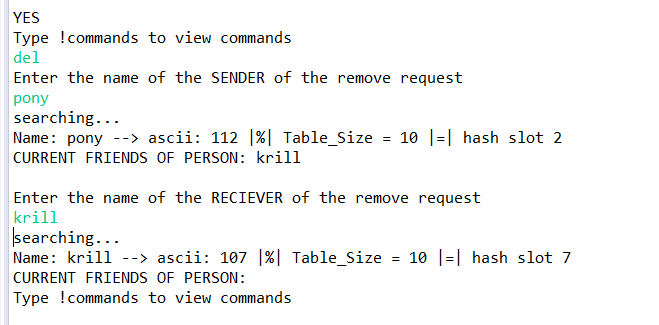


Now krill is added to pony’s friends list

Checktwo only prints yes if and only if person A and person B are BOTH friends with each other, not one sided.



If we remove krill from pony’s friends list, as follows…



Now it prints “NO” , because pony has no friends.

