Homework 9

Question 1 (5 pt.)

Extend the implementation of the hash table with chaining presented in class in order to support a *successor* operation. Notice that this operation is not part of a standard hash table interface, as it cannot be implemented efficiently on this type of container. Since the hash table does not maintain a relative order among keys, obtaining the successor of a given key involves traversing all elements and returning the minimum element that is greater than the given key.

Follow these steps to complete this implementation:

- Create a directory named `q1` and copy files `Node.java`, `HashTable.java`, and `Person.java` presented in class.

- In the present implementation class `Node`, a key is defined using type `Object`. But the successor operation requires keys to define a relative order beyond a simple equality comparison, and thus, they must support the `compareTo()` operation. Change the node definition to use type `Comparable` for field `key`, and modify every function where keys are passed as arguments accordingly.

- Write a public function in class `HashTable` named `GetSuccessor()`. The function has the same exact interface and behavior as the function with the same name in class `BinarySearchTree`. It takes a key as an argument and returns the data associated with the successor key, or `null` if the given key does not have a successor.

- Write a main program that inserts 4 persons into the hash table, using their names as the keys. The program should print the successor of at least two persons, one of which should be the person with the last name in alphabetical order (whose successor should be `null`).

Assuming that your program inserts 4 persons named Adam, John, Mary, and Zack, this is a possible sample output:

```
$ java Test

Successor of Mary: Zack (22 years old)
Successor of Zack: null
```

Create a ZIP file named `q1.zip` containing directory `q1`, and submit it on Canvas.

Question 2 (5 pt.)

A hash table relies on a high-quality hash function that spreads keys evenly across the array in order to minimize collisions. Research shows that keeping the size of the array equal to a prime number helps minimize collisions, especially when the keys are integer numbers whose hash function simply applies a modulo operation.

Modify the implementation of the hash table presented in class in order to make the array size become a prime number whenever the array has to grow. Follow these steps:

- Create a directory named `q2` and copy files `Node.java`, `HashTable.java`, and `Person.java` presented in class.

- Add a private function called `IsPrime()` in class `HashTable`. This function takes an integer value as an input argument and returns a Boolean value indicating whether the input is a prime number.

- Add a private function called `NextPrime()` in class `HashTable`. This function takes an integer number *n* as an input argument and returns the smallest prime number that is greater or equal to *n*. This function can conveniently call `IsPrime()` while searching for the next prime number.

- Modify function `Grow()` in class `HashTable` in such a way that the array grows to at least twice its original capacity, rounding this value up to the nearest prime number. You can conveniently call `NextPrime()` as part of the computation of the new size. Make the `Grow()` function print a message indicating the new array size.

- Write a main program that inserts 1000 integer values set to 0, 1, …, 999 into a hash table. The data value associated with each key can be set `null`. Execute your program and verify that the array is grown to sizes that are prime numbers.

This is a sample execution of your program:

```
$ java Test

Growing hash table to 11 entries
Growing hash table to 23 entries
...
```

Create a ZIP file named `q2.zip` containing directory `q2`, and submit it on Canvas.