```
#pragma once
#include <iostream>
#include "KeyValuePair.h"
#include "HashTableExceptions.h"
#include "LinkedList/LinkedList.h"
#include "SafeArray/SafeArray.h"
#include <cmath>

using namespace std;

template <class T>
class HashTable
{
 public:
    HashTable();
    ~HashTable();

    void insert(const string& k, const T& v);

    bool remove(const string& k);

    bool find(const string& k);

    T& retrieve(const string& k);

    void getKeys(SafeArray <string>& allKeys);

    void getValues(SafeArray <T>& allValues);

    void print();

    void printCollisionInfo();

    T& operator [] (string key);

    bool isEmpty();

 private:
    int hash(const string& k);

    int rehash(int newSize, const string& k);

    bool isPrime(int num);

    int findPrime(int num);

    void resize();

    SafeArray < LinkedList <KeyValuePair <T>* > > * table;

    int numElements;

};

//ctor
template <class T>
HashTable <T> :: HashTable()
{
    //initialize table with size 101
    table = new SafeArray < LinkedList <KeyValuePair <T>* > > (101);

    numElements = 0;
}
```

```
//dtor
template <class T>
HashTable <T> :: ~HashTable()
{
    //delete all of the kvps we have allocated and stored in linked list
    for(int i = 0; i < table->cap(); i++){

        if(!((*table)[i].isEmpty())){

            for(int j = 0; j < (*table)[i].size(); j++){

                delete (*table)[i][j];
            }
        }
    }

    delete table;
}

//insert
template <class T>
void HashTable <T> :: insert(const string& k, const T& v)
{
    //hash the key
    int index = hash(k);

    //is the kvp already in the table?
    if(find(k)){
        //HashTableException error;
        //throw error;

        cout << "Ugly exception here, kvp with this key is already in table" << endl;
    }
    else{
        //create a new KeyValuePair on the heap
        KeyValuePair <T>* p_kvp = new KeyValuePair <T> (k,v);

        //insert into the table
        (*table)[index].insert(p_kvp);

        numElements++;
    }

    if(numElements > (int) (((double)table->cap())*0.9)){
        //print collision info
        printCollisionInfo();
        //resize the table
        resize();
        //print collision info
        printCollisionInfo();
    }
}

//remove
template <class T>
bool HashTable <T> :: remove(const string& k)
{
    bool retVal = false;

    //hash the key
    int index = hash(k);
```

```
    //is linked list at index empty?
    if(((*table)[index]).isEmpty()){
        HashTableNullLinkedListException error;
        throw error;
    }
    //there is a linked list at the index
    else{

        //helper kvp object
        KeyValuePair <T> * val;

        (*table)[index].print();

        bool justRemovedItem = true;

        while(justRemovedItem){
            //position in linked list
            int count = 0;

            //go through list, keeping track of position
            //LinkedList.remove takes in a position

            if(!((*table)[index].isEmpty())){
                //if the list is not empty
                if((*table)[index].first(val)){

                    cout << val->getValue() << endl;

                    //if the first element was of interest
                    if(val->getKey() == k){
                        cout << "Element has key of interest" << endl;
                        //remove element at this position in list
                        (*table)[index].remove(count);


                        //switch retVal
                        retVal = true;
                    }
                    //continue to see if any of...
                    //next elements are of interest
                    else{
                        //checking next elements
                        while((*table)[index].next(val)){
                            //we are at next position in list...
                            count++;
                            cout << val->getValue() << endl;

                            //if the element at count is of interest
                            if(val->getKey() == k){
                                cout << "Element has key of interest" << endl;
                                //remove element at this position
                                (*table)[index].remove(count);

                                //switch retVal
                                retVal = true;
                                //break to start loop over correctly
                                break;
                            }

                            //if we have gone through entire list without already breaking...
                            if(count == (*table)[index].size() - 1){
                                justRemovedItem = false;
                            }
```

```
                        }
                    }
                }
            }
            else{
                justRemovedItem = false;
            }
        }
    }

    return retVal;

}

//find
template <class T>
bool HashTable <T> :: find(const string& k)
{
    bool retVal = false;

    //find index of list in array
    int index = hash(k);

    //search through linked list at hash(k)
    for(int i = 0; i < (*table)[index].size(); i++){

        //does the linked list have a kvp with key k?
        if((*table)[index].at(i)->getKey() == k){
            retVal = true;
        }

    }

    return retVal;
}

//retrieve
template <class T>
T& HashTable <T> :: retrieve(const string& k)
{
    if(!(find(k))){
        cout << "Obnoxious Exception thrown here" << endl;
    }
    else{
        //find index of list in array
        int index = hash(k);

        //search through linked list at hash(k)
        for(int i = 0; i < (*table)[index].size(); i++){
            //if kvp we are looking for?
            if((*table)[index].at(i)->getKey() == k){
                return (*table)[index].at(i)->getValue();
            }
        }
    }

}

//getKeys
template <class T>
void HashTable <T> :: getKeys(SafeArray <string>& aK)
{
    for(int i = 0; i < table->cap(); i++){
```

```
        //if the list at position i isn't empty
        if(!((*table)[i].isEmpty())){

            //for each element in the list
            for(int j = 0; j < (*table)[i].size(); j++){

                aK.push_back((*table)[i].at(j)->getKey());

            }
        }
    }

}

//getValues
template <class T>
void HashTable <T> :: getValues(SafeArray <T>& aV)
{
    for(int i = 0; i < table->cap(); i++){

        //if the list at position i isn't empty
        if(!((*table)[i].isEmpty())){

            //for each element in the list
            for(int j = 0; j < (*table)[i].size(); j++){

                aV.push_back((*table)[i].at(j)->getValue());

            }
        }
    }
}

//operator []
template <class T>
T& HashTable <T> :: operator [] (string key)
{
    return retrieve(key);
}

//print
template <class T>
void HashTable <T> :: print()
{
    for(int i = 0; i < table->cap(); i++){

        if(!((*table)[i].isEmpty())){

            for(int j = 0; j < (*table)[i].size(); j++){

                cout << (*table)[i][j]->getKey() << " ::: " << (*table)[i][j]->getValue() << e
ndl;

            }
        }
    }
}

//printCollisionInfo
template <class T>
void HashTable <T> :: printCollisionInfo()
{
```

```
    int runningTotal = 0;
    int averageSize;
    int longest = 0;
    int numWith = 0;
    int numEmpty = 0;

    for(int i = 0; i < table->cap(); i++){

        if(!((*table)[i].isEmpty())){

            runningTotal = runningTotal + (*table)[i].size();
            numWith++;

            if((*table)[i].size() > longest){

                longest = (*table)[i].size();

            }
        }
        else{
            numEmpty++;
        }
    }

    averageSize = runningTotal/numWith;

    cout << "Average length of non-empty lists: " << averageSize << endl
         << "Size of longest list: " << longest << endl
         << "Number of empty lists: " << numEmpty << endl << endl;


}

//isEmpty
template <class T>
bool HashTable <T> :: isEmpty()
{
    return (numElements == 0);
}

//hash
template <class T>
int HashTable <T> :: hash(const string& k)
{
    int index;

    unsigned int hashVal = 0;

    for(int i = 0; i < k.size(); i++){

        //find the hash value before mod
        hashVal = (hashVal + (int) k.at(i))*33;

    }

    //mod hash value by current size of table
    index = hashVal % table->cap();

    return index;

}

//rehash
```

```
template <class T>
int HashTable <T> :: rehash(int newSize, const string& k)
{
    int index;

    unsigned int hashVal = 0;

    for(int i = 0; i < k.size(); i++){

        //find the hash value before mod
        hashVal = (hashVal + (int) k.at(i))*33;

    }

    //mod hash value by current size of table
    index = hashVal % newSize;

    return index;

}

//isPrime
template <class T>
bool HashTable <T> :: isPrime(int num)
{
    bool retVal = true;

    double numDouble = (double) num;
    double root = sqrt(numDouble);
    int intRoot = (int) root;

    for(int i = 2; i < intRoot; i++){

        if((num % i) == 0){
            retVal = false;
            break;
        }
    }

    return retVal;
}

//findPrime
template <class T>
int HashTable <T> :: findPrime(int num)
{
    bool notPrime = true;

    while(notPrime){

        if(isPrime(num)){
            notPrime = false;
        }
        else
            num++;
    }

    return num;

}

//resize
template <class T>
```

```
void HashTable <T> :: resize()
{
    //first find new size
    int newSize = findPrime(2*(table->cap()));

    //make sure the newSize is prime
    if(!(isPrime(newSize))){
        HashTableInvalidSizeException error;
        throw error;
    }

    //temp pointer
    SafeArray < LinkedList <KeyValuePair <T>* > > * p_tableTemp;

    //otherwise, initialize new table-type with size newSize
    p_tableTemp = new SafeArray < LinkedList <KeyValuePair <T>* > > (newSize);

    //need to transfer old data into new table container
    for(int i = 0; i < table->cap(); i++){

        if(!((*table)[i].isEmpty())){

            for(int j = 0; j < (*table)[i].size(); j++){

                KeyValuePair <T> * p_kvpTemp = (*table)[i][j];

                //get key of this kvp, rehash it
                int newIndex = rehash(newSize, p_kvpTemp->getKey());

                //insert kvp into new table at index
                (*p_tableTemp)[newIndex].insert(p_kvpTemp);

            }
        }
    }

    //now make sure table pointer points to new table
    SafeArray < LinkedList <KeyValuePair <T>* > > * p_toDelete;

    p_toDelete = table;

    table = p_tableTemp;

    delete p_toDelete;

    p_toDelete = 0;
    p_tableTemp = 0;


}
```