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

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();

 private:
    int hash(const string& k);

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

};

//ctor
template <class T>
HashTable <T> :: HashTable() : table(101)
{


}

//dtor
template <class T>
HashTable <T> :: ~HashTable()
{


}

//insert
template <class T>
void HashTable <T> :: insert(const string& k, const T& v)
{
    //create a new KeyValuePair on the heap
    KeyValuePair <T>* p_kvp = new KeyValuePair <T> (k,v);

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

```
    //add the kvp to the list of kvps
    allKVPs.push_back(p_kvp);

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

}

//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 == 0?
    if((table[index]).isEmpty()){
        HashTableNullLinkedListException error;
        throw error;
    }
    //there is a linked list at the index
    else{

        //helper kvp object
        KeyValuePair <T> * val;
        //helper variable for list
        LinkedList <KeyValuePair <T>*> list = table[index];

        list.print();

        bool wasInList = true;

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

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

            if(!(list.isEmpty())){
                //if the list is not empty
                if(list.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
                        list.remove(count);
                        //remove kvp from list of kvps
                        allKVPs.removeElement(val);
                        //switch retVal
                        retVal = true;
                    }
                    else{
                        //checking next elements
                        while(list.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
                                list.remove(count);
                                //remove kvp from list of kvps
                                allKVPs.removeElement(val);
                                //switch retVal
                                retVal = true;
                                //break to start loop over correctly
                                break;
                            }

                            //if we have gone through entire list without already breaking...
                            if(count == list.size()){
                                wasInList = false;
                            }
                        }
                    }
                }
                else{
                    wasInList = false;
                }
            }
        }

        return retVal;

}

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

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

        if(allKVPs[i]->getKey() == k){
            retVal = true;
        }
    }

    return retVal;
}

//retrieve
template <class T>
T& HashTable <T> :: retrieve(const string& k)
{


}

//getKeys
template <class T>
void HashTable <T> :: getKeys(SafeArray <string>& aK)
{
    for(int i = 0; i < allKVPs.size(); i++){

        aK.push_back(allKVPs[i]->getKey());
```

```
    }
}

//getValues
template <class T>
void HashTable <T> :: getValues(SafeArray <T>& aV)
{
    for(int i = 0; i < allKVPs.size(); i++){

        aV.push_back(allKVPs[i]->getValue());

    }
}

//print
template <class T>
void HashTable <T> :: print()
{
    for(int i = 0; i < allKVPs.size(); i++){

        cout << allKVPs[i]->getKey() << " " << allKVPs[i]->getValue() << endl;

    }

}

//hash
template <class T>
int HashTable <T> :: hash(const string& k)
{
    unsigned int hashVal = 0;
    int index;

    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.size();

    return index;

}
```