```
/*Author: Eric Gustin
  Assignment: CPSC223-01 HW07 hash_table_collection.h
  Description: Hash table implementation of the collection
  class. Insert, remove, and find functions have become much faster than in
  previous implementations of the purely abstract class. Uses the standard
  library's hash function.
*/


#ifndef HASH_TABLE_COLLECTION_H
#define HASH_TABLE_COLLECTION_H

#include <vector>
#include <algorithm>
#include <functional>
#include "collection.h"

template<typename K, typename V>
class HashTableCollection : public Collection<K,V>
{
public:

 // create an empty linked list
 HashTableCollection();

 // copy a linked list
 HashTableCollection(const HashTableCollection<K,V>& rhs);

// assign a linked list
 HashTableCollection<K,V>& operator=(const HashTableCollection<K,V>& rhs);

 // delete a linked list
 ~HashTableCollection();

 // insert a key - value pair into the collection
 void insert(const K& key, const V& val);

 // remove a key - value pair from the collection
 void remove(const K& key);

 // find the value associated with the key
 bool find(const K& key, V& val) const;

 // find the keys associated with the range
 void find(const K& k1, const K& k2, std::vector<K>& keys) const;

 // return all keys in the collection
 void keys(std::vector<K>& keys) const;

 // return collection keys in sorted order
 void sort(std::vector<K>& keys) const;

 // return the number of keys in collection
 int size() const;

private:

 // helper to empty entire hash table
 void make_empty();

 // resize and rehash the hash table
 void resize_and_rehash();

 // linked list node structure
 struct Node {
  K key;
  V value;
  Node* next;
 };

 // number of k-v pairs in the collection
 int collection_size;
```

```
 // number of hash table buckets ( default is 16)
 int table_capacity;

 // hash table array load factor ( set at 75% for resizing )
 const double load_factor_threshold;

 // hash table array
 Node** hash_table;
};

template<typename K, typename V>
HashTableCollection<K,V>::HashTableCollection():
collection_size(0), table_capacity(16), load_factor_threshold(0.75)
{
 // dynamically allocate the hash table array
 hash_table = new Node*[table_capacity];
 // initialize the hash table chains
 for(int i = 0; i < table_capacity; ++ i)
  hash_table[i] = nullptr;
}

template<typename K, typename V>
void HashTableCollection<K,V>::make_empty()
{
 // make sure hash table exists
 if (hash_table != nullptr) {
 // remove each node
 Node* curr = nullptr;
 for (int i = 0; i < table_capacity; ++i) {
  while (hash_table[i] != nullptr) {
   curr = hash_table[i];
   hash_table[i] = hash_table[i]->next;
   delete curr;
   curr = nullptr;
   --collection_size;
  }
 }
 // remove the hash table
 delete hash_table;
 }
}

template<typename K, typename V>
HashTableCollection<K,V>::~HashTableCollection()
{
 make_empty();
}
template<typename K, typename V>
HashTableCollection<K,V>::HashTableCollection(const HashTableCollection<K,V>& rhs)
: hash_table(nullptr), load_factor_threshold(rhs.load_factor_threshold),
  collection_size(0)
{
 *this = rhs;
}

template<typename K, typename V>
HashTableCollection<K,V>&
HashTableCollection<K,V>::operator=(const HashTableCollection<K,V>& rhs)
{
 // check if rhs is current object and return currect object
 if (this == &rhs)
  return *this;
 // delete current object
 make_empty();
 // initialize current object
 // create the hash_table
 table_capacity = rhs.table_capacity;
 hash_table = new Node*[table_capacity];
 for(int i = 0; i < table_capacity; ++ i)
  hash_table[i] = nullptr;
 // do the copy
 Node* rhs_curr = nullptr;
 for (int i = 0; i < table_capacity; ++i) {
```

```
  rhs_curr = rhs.hash_table[i];
  while (rhs_curr != nullptr) {
   insert(rhs_curr->key, rhs_curr->value);
   rhs_curr = rhs_curr->next;
  }
 }
 return *this;
}

template<typename K, typename V>
void HashTableCollection<K,V>::resize_and_rehash()
{
 // setup new table
 int new_capacity = table_capacity * 2;
 int new_collection_size = collection_size;
 // dynamically allocate the new table
 Node** new_table = new Node*[new_capacity];
 // initialize the hash table chains
 for(int i = 0; i < new_capacity; ++i)
  new_table[i] = nullptr;
 // insert key values
 std::vector<K> ks;
 keys(ks);

 std::hash<K> hash_fun; // create hash function for the type
 std::size_t hcode; // get hash code
 std::size_t new_index; // find corresponding index for new table
 V val;
 for(K key : ks) {
  find(key, val); // find the value that is paired with the current key

  hcode = hash_fun(key);
  new_index = hcode % new_capacity;

  Node* curr = new Node;
  curr->key = key;
  curr->value = val;
  curr->next = new_table[new_index];
  new_table[new_index] = curr;
 }

 // clear the current data
 make_empty();

 hash_table = new_table;
 table_capacity = new_capacity;
 collection_size = new_collection_size;
}

template<typename K, typename V>
void HashTableCollection<K,V>::insert(const K& key, const V& val)
{
 // check current load factor versus load factor threshold ,
 // and resize and copy if necessary by calling resize_and_rehash ()
 if ((collection_size / table_capacity) > load_factor_threshold)
  resize_and_rehash();
 // hash the key
 std::hash<K> hash_fun; // create hash function for the type
 std::size_t hcode = hash_fun(key);
 std::size_t index = hcode % table_capacity;
 // create the new node
 Node* curr = new Node;
 curr->key = key;
 curr->value = val;
 curr->next = hash_table[index];
 hash_table[index] = curr;
 ++collection_size;
}

template<typename K, typename V>
void HashTableCollection<K,V>::remove(const K& key)
{
 if (collection_size > 0) {
```

```
  std::hash<K> hash_fun; // create hash function for the type
  std::size_t hcode = hash_fun(key); // get hash code
  std::size_t index = hcode % table_capacity; // find corresponding index

  Node* prev = hash_table[index];
  Node* curr = hash_table[index];
  if (hash_table[index] != nullptr && hash_table[index]->key == key) {
   hash_table[index] = hash_table[index]->next;
   delete prev;
   prev = nullptr;
   curr = nullptr;
   --collection_size;
  }
  while (curr != nullptr) {
   if (curr->key == key) {
    prev->next = curr->next;
    delete curr;
    curr = nullptr;
    --collection_size;
   }
   else {
    prev = curr;
    curr = curr->next;
   }
  }
 }
}

template<typename K, typename V>
bool HashTableCollection<K,V>::find(const K& key, V& val) const
{
 std::hash<K> hash_fun; // create hash function for the type
 std::size_t hcode = hash_fun(key); // get hash code
 std::size_t index = hcode % table_capacity; // find corresponding index

 Node* curr = hash_table[index];

 while (curr != nullptr) {
  if (curr->key != key)
   curr = curr->next;
  else {
   val = curr->value;
   return true;
  }
 }
 return false;
}

template<typename K, typename V>
void HashTableCollection<K,V>::find(const K& k1, const K& k2, std::vector<K>& keys) const
{

 // I create a vector of keys, sort it, then find range.
 std::vector<K> temp_vctr;
 sort(temp_vctr);
 for (K temp_key : temp_vctr)
  if (temp_key >= k1 && temp_key <= k2)
   keys.push_back(temp_key);
}

template<typename K, typename V>
void HashTableCollection<K,V>::keys(std::vector<K>& keys) const
{
 Node* ptr;
 // for-loop: iterate through each bin
 for (int i = 0; i < table_capacity; ++i) {
  ptr = hash_table[i];
  // while-loop: traverse a single bin
  while (ptr != nullptr) {

   keys.push_back(ptr->key);
   ptr = ptr->next;
  }
```

```
 }
}

template<typename K, typename V>
void HashTableCollection<K,V>::sort(std::vector<K>& ks) const
{
 keys(ks);
 std::sort(ks.begin(), ks.end());
}

template<typename K, typename V>
int HashTableCollection<K,V>::size() const
{
 return collection_size ;
}

#endif
```