```
/*Author: Eric Gustin
  Assignment: CPSC223-01 HW04
  Description: Contains LinkedListCollection class definition and
  implementation. This class inherits from the pure abstract
  class called Collection. Follows the rule of three by having a copy constructor
  destructor, assignment operator. Makes use of a the Node struct to represent
  key-value pairs. Includes functionality to insert a node with a key-value pair,
  remove a node with a key-value pair, find and return a value associated with a
  key, find a range of keys that are between two other keys. return all of the keys
  in the list, return the size, and sort the list in ascending order of keys.*/

#ifndef LINKED_LIST_COLLECTION_H
#define LINKED_LIST_COLLECTION_H

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


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

 // create an empty linked list
 LinkedListCollection();

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

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

 // delete a linked list
 ~LinkedListCollection();

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

 Node* head; // pointer to first list node
 Node* tail; // pointer to last list node
 int length; // number of linked list nodes in list

 // helper function for destructor & assignment operator
 void make_empty(LinkedListCollection<K,V>& list);
};

template <typename K, typename V>
LinkedListCollection<K,V>::LinkedListCollection()
: head(nullptr), tail(nullptr), length(0)
{}

template <typename K, typename V>
LinkedListCollection<K,V>::LinkedListCollection(const LinkedListCollection<K,V>& rhs)
: head(nullptr), tail(nullptr), length(0)
{
```

```
 *this = rhs;
}

template <typename K, typename V>
LinkedListCollection<K,V>& LinkedListCollection<K,V>::operator=(const LinkedListCollection<K,V>& rhs)
{
 // ensure that an object is not being assigned to itself
 if (this != &rhs) {
  // empty contents of this object
  make_empty(*this);
  // copy contents into this object
  Node* curr = rhs.head;
  while (curr != nullptr) {
   insert(curr->key, curr->value);
   curr = curr->next;
  }
  delete curr;
  curr = nullptr;
 }
 return *this;
}

template <typename K, typename V>
LinkedListCollection<K,V>::~LinkedListCollection()
{
 make_empty(*this);
}

template <typename K, typename V>
void LinkedListCollection<K,V>::make_empty(LinkedListCollection<K,V>& list) {
 Node* curr = head;
 while (head != nullptr) {
  head = head->next;
  delete curr;
  curr = nullptr;
  --length;
 }
 head = nullptr;
 tail = nullptr;
}

template <typename K, typename V>
void LinkedListCollection<K,V>::insert(const K& key, const V& val) {
 Node* curr = new Node;
 // edge cases
 if (size() == 0)
  head = curr;
 if (size() > 0)
  tail->next = curr;

 tail = curr;
 // assign values to the current node's members variables
 curr->key = key;
 curr->value = val;
 curr->next = nullptr;

 ++length;
}

template <typename K, typename V>
void LinkedListCollection<K,V>::remove(const K& key){
 if (size() > 0) {
   Node* prev = head;
   Node* curr = head->next;
   // edge case where desired key is contained in head
   if (head->key == key) {
    head = head->next;
    delete prev;
    prev = nullptr;
    --length;
   }
   else {
    while (curr != nullptr) {
     if (curr->key == key) {
      //edge case where desired key is at end of list
      if (curr->next == nullptr)
       tail = prev;
      prev->next = curr->next;
      delete curr;
      curr = nullptr;
      curr = nullptr;
      --length;
```

```
      break;
     }
     prev = curr;
     curr = curr->next;
    }
   }
  }
 }

template <typename K, typename V>
bool LinkedListCollection<K,V>::find(const K& key, V& val) const
{
 Node* curr = head;
 // iterate through linked list until desired key is found or end of list is reached
 while (curr != nullptr) {
  if (curr->key == key) {
   val = curr->value;
   return true;
  }
  curr = curr->next;
 }
 return false;
}

template <typename K, typename V>
void LinkedListCollection<K,V>::find(const K& k1, const K& k2, std::vector<K>& keys) const
{
 //for-loop variable to keep track of current index of keys vector
 unsigned int curr_index = 0;
 // set keys vector equal to kv_list
 this->keys(keys);
 //iterate through keys and remove pairs that dont meet requirements
 //curr_index is not incremented if an element is erased since all
 //elements coming after will be moved forward 1 index.
 for (int i = 0; i < size(); ++i) {
  if ((keys[curr_index] < k1) || (keys[curr_index] > k2)) {
   keys.erase(keys.begin()+curr_index);
  }
  else
   ++curr_index;
 }
}

template <typename K, typename V>
void LinkedListCollection<K,V>::keys(std::vector<K>& keys) const
{
 // inserts keys of the list into a vector
 Node* cur = head;
 while (cur != nullptr) {
  keys.push_back(cur->key);
  cur = cur->next;
 }
}

template <typename K, typename V>
void LinkedListCollection<K,V>::sort(std::vector<K>& keys) const
{
 Node* ptr = head;
 while(ptr != nullptr) {
  keys.push_back(ptr->key);
  ptr = ptr->next;
 }
 std::sort(keys.begin(), keys.end());
}

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

#endif
```