Project 1 - Hashing
CSCI 230 T Th 11:10 am
Compiler: g++
OS: Linux

Michael Morikawa

March 29, 2020

Source Code

main.cpp

```
#include "HashCode.hpp"
#include "ChainHashMap.hpp"
#include "OpenAddressMap.hpp"
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

struct RecordData
{
    int population;
    std::string name;
    RecordData(int n = 0, std::string s = "") : population(n), name(s) {}
    friend std::ostream &operator<<(std::ostream &os, const RecordData &data)
    {
        os << data.population << ' ' << data.name;
        return os;
    }
};

//returns number of the choice
int menu();

//Create vector of entries from a properly formatted input file
std::vector<Entry<int, RecordData>> processFile(std::string filePath);

int main()
{
    std::string filepath;
    std::vector<Entry<int, RecordData>> entryList;
    std::cout << "Please enter the file path for the input file you wish to use:\n";
    std::cin >> filepath;
    std::cout << filepath << std::endl; //!Alert: Delete later
    entryList = processFile(filepath);
    if (entryList.empty())
    {
        std::cout << "File not found or file is empty.";
        return -1;
    }

    int code, population;
    std::string name;
    int choice;
    double loadFactor;
    std::cout << "\n1. Chain Hashing\n"
              << "2. Linear Probe Hashing\n"
              << "3. Double Hashing\n"
              << "Please choose a hashing scheme: ";
    std::cin >> choice;
    std::cout << choice << std::endl; //!Alert: Delete later
    std::cout << "\nPlease enter a load factor between 0-1: ";
    std::cin >> loadFactor;
    std::cout << loadFactor << std::endl; //!Alert: Deletle later
    int operation = 0;
    if (choice == 1)
    {
```

```
    ChainHashMap<int, RecordData, HashCode> table(entryList, loadFactor);
    while (operation != 5)
    {
        operation = menu();
        switch (operation)
        {
        case 1:
        {
            std::cout << "Please enter the state code: ";
            std::cin >> code;
            std::cout << code << std::endl; //!ALERT: For file redirection delete later
            auto it = table.find(code, true);
            if (it == table.end())
            {
                std::cout << "Could not find the record\n";
            }
            else
            {
                std::cout << "Record Found\n";
                std::cout << (*it).value() << '\n';
            }
        }
        break;
        case 2:
        {
            std::cout << "Please enter the state code, population and name separated by spaces:\n";
            std::cin >> code >> population;
            std::getline(std::cin, name);
            std::cout << code << " " << population << " " << name << std::endl; //!Alert: Delete later
            table.put(code, RecordData(population, name), true);
        }
        break;
        case 3:
        {
            std::cout << "Please enter the state code: ";
            std::cin >> code;
            std::cout << code << std::endl; //!Alert: Delete later
            table.erase(code, true);
        }
        break;
        case 4:
        {
            table.printAll();
        }
        break;
        default:
            break;
        }
    }
}
else
{
    OpenAddressMap<int, RecordData, HashCode> *table2;
    if (choice == 2)
    {
        table2 = new OpenAddressMap<int, RecordData, HashCode>;
    }
    else
    {
        table2 = new DoubleHashMap<int, RecordData, HashCode>;
```

```
        }
        table2->createFromEntryList(entryList, loadFactor);
        while (operation != 5)
        {
            operation = menu();
            switch (operation)
            {
            case 1:
            {
                std::cout << "Please enter the state code: ";
                std::cin >> code;
                std::cout << code << std::endl; //!Alert: Delete later
                auto it = table2->find(code, true);
                if (it == table2->end())
                {
                    std::cout << "Could not find the record\n";
                }
                else
                {
                    std::cout << (*it).value() << '\n';
                }
            }
            break;
            case 2:
            {
                std::cout << "Please enter the state code, population and name separated by spaces:\n";
                std::cin >> code >> population;
                std::getline(std::cin, name);
                std::cout << code << " " << population << " " << name << std::endl; //!Alert: Delete later
                table2->put(code, RecordData(population, name), true);
            }
            break;
            case 3:
            {
                std::cout << "Please enter the state code: ";
                std::cin >> code;
                std::cout << code << std::endl; //!Alert: Delete later
                table2->erase(code, true);
            }
            break;
            case 4:
            {
                table2->printAll();
            }
            break;
            default:
                break;
            }
        }
    }
}

int menu()
{
    std::cout << "\n1. Search for a record\n"
              << "2. Insert a record\n"
              << "3. Delete a record\n"
              << "4. List all records\n"
              << "5. Exit\n\n"
              << "Please Select an option: ";
```

```
    int choice;
    std::cin >> choice;
    std::cout << choice << std::endl; //!Alert delete later
    if (std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore();
        return 0;
    }
    return choice;
}

std::vector<Entry<int, RecordData>> processFile(std::string filePath)
{
    std::vector<Entry<int, RecordData>> entryList;
    Entry<int, RecordData> inputEntry;
    std::ifstream inputFile;
    inputFile.open(filePath);
    if (inputFile.fail())
    {
        return entryList;
    }
    int size;
    inputFile >> size;
    std::string stateCode, population, name;
    for (int i = 0; i < size; i++)
    {
        std::getline(inputFile, stateCode, ',');
        std::getline(inputFile, population, ',');
        std::getline(inputFile, name);
        inputEntry.setKey(std::stoi(stateCode));
        inputEntry.setValue(RecordData(std::stoi(population), name));
        entryList.push_back(inputEntry);
    }
    return entryList;
}
```

HashCode.hpp

```
#pragma once
class HashCode
{
public:
    int operator()(int key)
    {
        return key;
    }
};
```

Entry.hpp

```
#pragma once
template <typename K, typename V>
class Entry
{
public:
    Entry(const K& k = K(), const V& v = V())
        : _key(k), _value(v) {}
    const K& key() const
    {
```

```
        return _key;
    }
    const V& value() const
    {
        return _value;
    }
    void setKey(const K& k)
    {
        _key = k;
    }
    void setValue(const V& v)
    {
        _value = v;
    }

private:
    K _key;
    V _value;
};
```

ChainHashMap.hpp

```
#pragma once
#include "Entry.hpp"
#include <cmath>
#include <iostream>
#include <list>
#include <vector>

//Templated on key, value and hash function
template <typename K, typename V, typename H>
class ChainHashMap
{

public:
    class Iterator;

public:
    ChainHashMap(int capacity = 100)
        : n(0), table(capacity), probes(0) {}
    ChainHashMap(const std::vector<Entry<K, V>>& eList, float loadFactor);
    int size() const
    {
        return n;
    }
    bool empty() const
    {
        return n == 0;
    }
    Iterator find(const K& k, bool diagnostic = false);
    //diagnostic will print the probes want to turn off for loading file
    Iterator put(const K& k, const V& v, bool diagnostic = false);
    Iterator put(const Entry<K, V>& e, bool diagnostic = false);
    void erase(const K& k, bool diagnostic = false);
    void erase(const Iterator& p, bool diagnostic = false);
    void printAll();
    Iterator begin();
    Iterator end();

protected:
```

```
    typedef std::list<Entry<K, V>> Bucket;
    typedef std::vector<Bucket> BktArray;
    typedef typename BktArray::iterator BktIter;
    typedef typename Bucket::iterator ListIter;
    Iterator finder(const K& k);
    Iterator inserter(const Iterator& p, const Entry<K, V>& e);
    void eraser(const Iterator& p);
    static void nextEntry(Iterator& p)
    {
        ++p.ent;
    }
    static bool endOfBkt(Iterator& p)
    {
        return p.ent == p.bkt->end();
    }
    static bool isPrime(int n)
    {
        if (n < 2 || n % 2 == 0)
        {
            return false;
        }
        if (n == 2)
        {
            return true;
        }
        for (int div = 3; div <= sqrt(n); div += 2)
        {
            if (n % div == 0)
            {
                return false;
            }
        }
        return true;
    }

private:
    int n;
    H hash;
    BktArray table;
    int probes; //keeps track of probes during operations

public:
    class Iterator
    {
    private:
        BktIter bkt;
        ListIter ent;
        const BktArray* ba;

    public:
        Iterator(const BktArray& ba, const BktIter& bIt,
                 const ListIter& entIt = ListIter())
            : ba(&ba), bkt(bIt), ent(entIt) {}
        Entry<K, V>& operator*() const;
        bool operator==(const Iterator& p) const;
        Iterator& operator++();
        friend class ChainHashMap;
    };
};
```

```
//Iterator Class Definitions

template <typename K, typename V, typename H>
Entry<K, V>& ChainHashMap<K, V, H>::Iterator::operator*() const
{
    return *ent;
}

template <typename K, typename V, typename H>
bool ChainHashMap<K, V, H>::Iterator::operator==(const Iterator& p) const
{
    if (ba != p.ba || bkt != p.bkt)
    {
        return false;
    }
    else if (bkt == ba->end())
    {
        return true;
    }
    else
    {
        return (ent == p.ent);
    }
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator& ChainHashMap<K, V, H>::Iterator::operator++()
{
    ++ent;
    if (endOfBkt(*this))
    {
        ++bkt; //check next bucket
        while (bkt != ba->end() && bkt->empty())
        {
            ++bkt;
        }
        if (bkt == ba->end())
        {
            return *this;
        }
        ent = bkt->begin();
    }
    return *this;
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::end()
{
    return Iterator(table, table.end());
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::begin()
{
    if (empty())
    {
        return end();
    }
    BktIter bkt = table.begin();
    while (bkt->empty())
```

```
    {
        ++bkt;
    }
    return Iterator(table, bkt, bkt->begin());
}

template <typename K, typename V, typename H>
ChainHashMap<K, V, H>::ChainHashMap(const std::vector<Entry<K, V>>& eList, float loadFactor)
{

    int capacity = eList.size() / loadFactor;
    //finding size for table
    if (capacity % 2 == 0)
    {
        capacity++; //make it odd for easier prime checking
    }
    while (!isPrime(capacity))
    {
        capacity += 2;
    }
    table.resize(capacity);
    int probeSum = 0;
    int probeMax = 0;
    for (auto e : eList)
    {
        put(e);
        probeSum += probes;
        probeMax = std::max(probes, probeMax);
    }
    std::cout << "Table Size: " << table.size()
              << "\nAverage number of probes: " << (float)probeSum / eList.size()
              << "\nMax Probes: " << probeMax << std::endl;
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::finder(const K& k)
{
    probes = 1; //set to 1 because the initial index counts as a probe
    int i = hash(k) % table.size();
    BktIter bkt = table.begin() + i;
    Iterator p(table, bkt, bkt->begin());
    while (!endOfBkt(p) && (*p).key() != k)
    {
        probes++;
        nextEntry(p);
    }
    return p;
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::find(const K& k, bool diagnostic)
{
    Iterator p = finder(k);
    if (diagnostic)
    {
        std::cout << "Found/Not Found in " << probes << " probes.\n";
    }
    if (endOfBkt(p))
        return end();
    else
```

```
        return p;
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::inserter(const Iterator& p, const Entry<K, V>&
{
    ListIter ins = p.bkt->insert(p.ent, e);
    n++;
    return Iterator(table, p.bkt, ins);
}

template <typename K, typename V, typename H> // insert/replace (v,k)
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::put(const K& k, const V& v, bool diagnostic)
{
    Iterator p = finder(k);
    if (diagnostic)
    {
        std::cout << "Inserted in " << probes << " probes.\n";
    }
    if (endOfBkt(p))
    {
        return inserter(p, Entry<K, V>(k, v));
    }
    else
    {
        p.ent->setValue(v);
        return p;
    }
}

template <typename K, typename V, typename H>
typename ChainHashMap<K, V, H>::Iterator ChainHashMap<K, V, H>::put(const Entry<K, V>& e, bool diagnostic)
{

    return put(e.key(), e.value(), diagnostic);
}

template <typename K, typename V, typename H>
void ChainHashMap<K, V, H>::erase(const Iterator& p, bool diagnostic)
{
    eraser(p);
    if (diagnostic)
    {
        std::cout << "Removed in " << probes << " probes\n";
    }
}

template <typename K, typename V, typename H>
void ChainHashMap<K, V, H>::eraser(const Iterator& p)
{
    p.bkt->erase(p.ent);
    n--;
}

template <typename K, typename V, typename H>
void ChainHashMap<K, V, H>::erase(const K& k, bool diagnostic)
{
    Iterator p = finder(k);

    if (diagnostic)
```

```
    {
        std::cout << "Removed/searched in " << probes << " probes\n";
    }
    if (endOfBkt(p))
    {
        return;
    }
    eraser(p);
}

template <typename K, typename V, typename H>
void ChainHashMap<K, V, H>::printAll()
{
    Iterator p = begin();
    Iterator stop = end();
    while (!(p == stop))
    {
        std::cout << (*p).key() << ' ' << (*p).value() << '\n';
        ++p;
    }
}
```

OpenAddressMap.hpp

```
#include "Entry.hpp"
#include <vector>

template <typename K, typename V>
class VisitEntry : public Entry<K, V>
{
public:
    VisitEntry()
        : Entry<K, V>(), empty(true), available(true) {}

    VisitEntry(const K &k, const V &v)
        : Entry<K, V>(k, v), empty(false), available(false) {}

    bool available;
    bool empty;
};

template <typename K, typename V, typename H>
class OpenAddressMap
{
public:
    class Iterator;

    //data
protected:
    std::vector<VisitEntry<K, V>> table;
    int n;
    int probes;
    H hash;

    //helper functions
protected:
    //skips over erased entries
    virtual Iterator finder(const K &k);

    //different function used to find where to insert
```

```
    //will return first open spot
    virtual Iterator insertionFinder(const K &k);
    virtual void eraser(const Iterator &p);

    static bool isPrime(int n)
    {
        if (n < 2 || n % 2 == 0)
        {
            return false;
        }
        if (n == 2)
        {
            return true;
        }
        for (int div = 3; div <= sqrt(n); div += 2)
        {
            if (n % div == 0)
            {
                return false;
            }
        }
        return true;
    }

public:
    OpenAddressMap(int size = 11) : n(0), table(size){};

    //Can't be a constructor since it calls a virutal function
    //Also outputs average probes and max probes for insertion from list
    // to console for project
    virtual void createFromEntryList(const std::vector<Entry<K, V>> &eList, float loadFactor);
    Iterator find(const K &k, bool diagnositc = false);
    Iterator put(const Entry<K, V> &e, bool diagnostic = false);
    Iterator put(const K &k, const V &v, bool diagnostic = false);
    void erase(const Iterator &p, bool diagnostic = false);
    void erase(const K &k, bool diagnostic = false);
    bool empty() const
    {
        return n == 0;
    }
    int size()
    {
        return n;
    }
    Iterator end();
    Iterator begin();
    void printAll();

public:
    class Iterator
    {
    protected:
        typedef typename std::vector<VisitEntry<K, V>>::iterator vecItor;

    private:
        const std::vector<VisitEntry<K, V>> *tableRef;
        vecItor bktIt;

    public:
        Iterator(const std::vector<VisitEntry<K, V>> &table, const vecItor &it)
```

```
            : tableRef(&table), bktIt(it) {}
        Iterator &
        operator++();
        VisitEntry<K, V> &operator*() const;
        bool operator==(const Iterator &p) const;
        friend class OpenAddressMap;
    };
};

template <typename K, typename V, typename H>
VisitEntry<K, V> &OpenAddressMap<K, V, H>::Iterator::operator*() const
{
    return *bktIt;
}

template <typename K, typename V, typename H>
bool OpenAddressMap<K, V, H>::Iterator::operator==(const Iterator &p) const
{
    return (tableRef == p.tableRef && bktIt == p.bktIt);
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator &OpenAddressMap<K, V, H>::Iterator::operator++()
{
    ++bktIt;
    if (bktIt->available)
    {
        while (bktIt != tableRef->end() && bktIt->available)
        {
            ++bktIt;
        }
    }
    return *this;
}

template <typename K, typename V, typename H>
void OpenAddressMap<K, V, H>::createFromEntryList(const std::vector<Entry<K, V>> &eList, float loadFactor)
{
    int capacity = eList.size() / loadFactor;
    //finding size for table
    if (capacity % 2 == 0)
    {
        capacity++; //make it odd for easier prime checking
    }
    while (!isPrime(capacity))
    {
        capacity += 2;
    }
    table.resize(capacity);
    int probeSum = 0;
    int probeMax = 0;

    for (auto e : eList)
    {
        put(e);
        probeSum += probes;
        probeMax = std::max(probes, probeMax);
    }
    std::cout << "Table Size: " << table.size()
              << "\nAverage number of probes: " << (float)probeSum / eList.size()
```

```
              << "\nMax Probes: " << probeMax << std::endl;
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::end()
{
    return Iterator(table, table.end());
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::begin()
{
    if (empty())
    {
        return end();
    }
    auto it = table.begin();
    while (it->empty)
    {
        ++it;
    }
    return Iterator(table, it);
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::finder(const K &k)
{
    probes = 1;
    int i = hash(k) % table.size();
    auto it = table.begin() + i;
    while (!it->empty && it->key() != k)
    {
        probes++;
        it++;
        if (it == table.end())
        {
            it = table.begin();
        }
    }
    return Iterator(table, it);
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::insertionFinder(const K &k)
{
    probes = 1;
    int i = hash(k) % table.size();
    auto it = table.begin() + i;
    while (!it->available && it->key() != k)
    {
        probes++;
        it++;
        if (it == table.end())
        {
            it = table.begin();
        }
    }
    return Iterator(table, it);
}
```

```
template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::find(const K &k, bool diagnostic)
{
    Iterator p = finder(k);

    if (diagnostic)
    {
        std::cout << "Found/Not Found in " << probes << " probes.\n";
    }

    if ((*p).empty)
    {
        return end();
    }
    else
    {
        return p;
    }
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::put(const K &k, const V &v, bool diagnosti
{
    n++;
    Iterator p = insertionFinder(k);
    (*p) = VisitEntry<K, V>(k, v);
    p.bktIt->empty = false;
    p.bktIt->available = false;
    return p;
}

template <typename K, typename V, typename H>
typename OpenAddressMap<K, V, H>::Iterator OpenAddressMap<K, V, H>::put(const Entry<K, V> &e, bool diagnostic)
{
    return put(e.key(), e.value(), diagnostic);
}

template <typename K, typename V, typename H>
void OpenAddressMap<K, V, H>::eraser(const Iterator &p)
{
    //make it usable again
    (*p).available = true;

    n--;

    //reset entry to default
    (*p).setKey(K());
    (*p).setValue(V());
}

template <typename K, typename V, typename H>
void OpenAddressMap<K, V, H>::erase(const Iterator &p, bool diagnostic)
{
    eraser(p);
}

template <typename K, typename V, typename H>
void OpenAddressMap<K, V, H>::erase(const K &k, bool diagnostic)
{
    Iterator p = finder(k);
```

```
    eraser(p);
}

template <typename K, typename V, typename H>
void OpenAddressMap<K, V, H>::printAll()
{
    Iterator p = begin();
    Iterator stop = end();
    while (!(p == stop))
    {
        std::cout << (*p).key() << ' ' << (*p).value() << '\n';
        ++p;
    }
}

template <typename K, typename V, typename H>
class DoubleHashMap : public OpenAddressMap<K, V, H>
{
public:
    void createFromEntryList(const std::vector<Entry<K, V>> &eList, float loadFactor) override
    {
        int capacity = eList.size() / loadFactor;
        //finding size for table
        if (capacity % 2 == 0)
        {
            capacity++; //make it odd for easier prime checking
        }
        while (!this->isPrime(capacity))
        {
            capacity += 2;
        }
        this->table.resize(capacity);
        int probeSum = 0;
        int probeMax = 0;

        collisionPrime = (this->table.size()) - 1;
        if (collisionPrime % 2 == 0)
        {
            collisionPrime--;
        }
        while (!this->isPrime(collisionPrime))
        {
            collisionPrime -= 2;
        }

        for (auto e : eList)
        {
            this->put(e);
            probeSum += this->probes;
            probeMax = std::max(this->probes, probeMax);
        }
        std::cout << "Table Size: " << this->table.size()
                  << "\nAverage number of probes: " << (float)probeSum / eList.size()
                  << "\nMax Probes: " << probeMax << std::endl;
    }

protected:
    typedef typename OpenAddressMap<K, V, H>::Iterator Iterator;
    Iterator finder(const K &k) override
    {
```

```
        this->probes = 1;
        int index = k % this->table.size();
        auto currentEntry = this->table[index];
        while (!currentEntry.empty && currentEntry.key() != k)
        {
            this->probes++;
            index = (index + (collisionPrime - (k % collisionPrime))) % this->table.size();
            currentEntry = this->table[index];
        }
        return Iterator(this->table, this->table.begin() + index);
    }

    Iterator insertionFinder(const K &k) override
    {
        this->probes = 1;
        int index = k % this->table.size();
        auto currentEntry = this->table[index];
        while (!currentEntry.available && currentEntry.key() != k)
        {
            this->probes++;
            index = (index + (collisionPrime - (k % collisionPrime))) % this->table.size();
            currentEntry = this->table[index];
        }
        return Iterator(this->table, this->table.begin() + index);
    }
    /*
    Iterator put(const K& k, const V& v, bool diagnostic = false) override
    {
        this->n++;
        Iterator p = insertionFinder(k);
        (*p) = VisitEntry<K, V>(k, v);
        (*p).empty = false;
        (*p).available = false;
        return p;
    }

    Iterator put(const Entry<K, V>& e, bool diagnostic = false) override
    {
        return put(e.key(), e.value(), diagnostic);
    }
    */

private:
    int collisionPrime;
};
```

Output