```cpp
/***************************************************************************
 * AUTHOR           : Nick Reardon
 * Assignment #7    : Hashing Algorithms
 * CLASS            : CS1D
 * SECTION          : MW - 2:30p
 * DUE DATE         : 03 / 04 / 20
 ***************************************************************************/
#ifndef _DOUBLEHASH_H_
#define _DOUBLEHASH_H_
#include <string>
#include <iostream>
#include <iomanip>

namespace errorType
{
    enum errors
    {
        DEFUALT,
        EMPTY,
        FULL

    };

    std::string errorString[]
    {
        "Error - An error occured",
        "Error - Map is empty",
        "Error - Map is full"
    };
}

enum indexLabel
{
    EMPTY,
    FULL,
    AVAILABLE
};

template <class T_key, class T_value>
struct T_struct
{
    T_key key;
    T_value value;

    enum indexLabel label = EMPTY;

    T_struct<T_key, T_value>()
    {
        key = -1;
        label = EMPTY;
    }

    T_struct<T_key, T_value>(const T_key& key, const T_value& value)
    {
        this->key = key;
        this->value = value;

        label = EMPTY;
    }

    T_struct<T_key, T_value>(const T_struct<T_key, T_value>& rhs)
    {
        this->key = rhs.key;
        this->value = rhs.value;

        this->label = rhs.label;
    }

    T_struct<T_key, T_value>& operator=(const T_struct<T_key, T_value>& rhs)
    {
        this->key = rhs.key;
        this->value = rhs.value;

        this->label = rhs.label;

        return *this;
    }
};

template <class T_key, class T_value>
T_struct<T_key, T_value> make_struct(T_key newKey, T_value newValue)
{
    return T_struct<T_key, T_value>(newKey, newValue);
}



template <class T_key, class T_value>
class DoubleHashMap
{
private:

    T_struct<T_key, T_value>* map;

    int currentSize;
    int capacity;

    // ostream member? Assign it in constructor or method???
    // set to NULL?

protected:

    int DoubleHash(const int givenKey, const int collisionCount) const
    {
```

```cpp
        int hashKey;

        int j = collisionCount;
        int k = givenKey;
        int N = capacity;

        /*
        int hk;
        int hk2;

        hk = (k % N);
        hk2 = (k % 13);
        hk2 = 13 - hk2;
        hk2 = j * hk2;

        hashKey = hk + hk2;

        hashKey = hashKey % N;
        */

        hashKey = ((( k % N ) + (j * (13 - (k % 13))) ) % N);

        return hashKey;
    }

    int DoubleHash(const T_struct<T_key, T_value>& toInsert, const int collisionCount) const
    {
        DoubleHash(toInsert.key, collisionCount);
    }

public:

    DoubleHashMap(const int newCapacity)
    {
        map = new  T_struct<T_key, T_value>[newCapacity];

        currentSize = 0;

        capacity = newCapacity;
    }

    ~DoubleHashMap()
    {
        delete[] map;
    }

    void insert(const T_struct<T_key, T_value>& toInsert)
    {
        if (full())
        {
            throw(errorType::FULL, errorType::errorString[FULL], 5);
        }

        int hashKey;
        std::string output = std::to_string(toInsert.key);
        int collisionCount = 0;
        bool stopHash = false;
        bool success = false;

        while (stopHash == false)
        {
            hashKey = DoubleHash(toInsert.key, collisionCount);

            if (map[hashKey].label == EMPTY ||
                map[hashKey].label == AVAILABLE)
            {
                stopHash = true;
                success = true;

            }
            else
            {
                if (map[hashKey].key == toInsert.key)
                {
                    stopHash = true;
                    success = true;
                }

                collisionCount++;
            }

            output += "->" + std::to_string(hashKey);

        }


        if (success)
        {
            map[hashKey] = toInsert;
            map[hashKey].label = FULL;

            currentSize++;

            std::cout << "Inserting: " << '(' << toInsert.key << ", " <<
              toInsert.value << ')'
                << '\n' << "Hashed Key: " << output << '\n' << '\n';
        }
    }

    /**
     * @fn  void DoubleHashMap::remove(const T_key key)
     * @brief   Removes the given key
     *
     * @exception   errorType::FULL,    Thrown when a full, error condition
       occurs.
     *
     * @param   key The key to remove.
     */
    void remove(const T_key key)
    {
        if (empty())
        {
            throw(errorType::FULL, errorType::errorString[FULL], 5);
        }

        int hashKey;
        std::string output = std::to_string(key);
        int collisionCount = 0;
        bool stopHash = false;
        bool success = false;
        while (stopHash == false)
        {
            hashKey = DoubleHash(key, collisionCount);

            if (map[hashKey].label == FULL ||
                map[hashKey].label == AVAILABLE)
            {
                if (map[hashKey].key == key)
                {
                    stopHash = true;
                    success = true;
                }
                else
                {
                    collisionCount++;

                }
            }
            else
            {
                stopHash = true;
            }
            output += "->" + std::to_string(hashKey);
        }

        if (success)
        {
            map[hashKey].key = -1;
            map[hashKey].value = "";
            map[hashKey].label = AVAILABLE;

            currentSize--;

            std::cout << "Removing key: " << key << '\n'
                << "Hashed Key: " << output << '\n' << '\n';
```

```cpp


        }


    }

    bool full()
    {
        return currentSize == capacity;
    }

    bool empty()
    {
        return currentSize == 0;
    }

    int size()
    {
        return currentSize;
    }

    void printAll(std::ostream& output)
    {
        if (empty())
        {
            throw(errorType::EMPTY, errorType::errorString[EMPTY], 5);
        }

        output << "  Index  | LABEL |  Key  |  Value" << '\n'
               << "_________|_______|_______|___________________"
            << '\n';

        for (int i = 0; i < capacity; i++)
        {
            output << std::right
                << " [" << std::setw(5) << i << "] | ";
            switch (map[i].label)
            {
            case EMPTY:
                output << "EMPTY |";
                break;

            case FULL:
                output << "FULL  |";
                break;

            case AVAILABLE:
                output << "AVAIL |";
                break;

            }
```

```
                output << ' ' << std::setw(4) << map[i].key << " |";

                output  << std::left
                    << ' ' << map[i].value
                    << '\n';
            }
            output << "\n\n";
        }
};

//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||


template <class T_key, class T_value>
class QuadraticHashMap
{
private:

    T_struct<T_key, T_value>* map;

    int currentSize;
    int capacity;

    // ostream member? Assign it in constructor or method???
    // set to NULL?

protected:

    int QuadraticHash(const int givenKey, const int collisionCount) const
    {
        int hashKey;

        int j = collisionCount;
        int k = givenKey;
        int N = capacity;

        /*
        int hk;
        int hk2;

        hk = (k % N);
        hk2 = (k % 13);
        hk2 = 13 - hk2;
        hk2 = j * hk2;

        hashKey = hk + hk2;

        hashKey = hashKey % N;
        */



        if (j > 0)
        {
            hashKey = (((k % N) + (j * j)) % N);
        }
        else
        {
            hashKey = (k % N);
        }

        return hashKey;
    }

    int QuadraticHash(const T_struct<T_key, T_value>& toInsert, const int
      collisionCount) const
    {
        QuadraticHash(toInsert.key, collisionCount);
    }

public:

    QuadraticHashMap(const int newCapacity)
    {
        map = new  T_struct<T_key, T_value>[newCapacity];

        currentSize = 0;

        capacity = newCapacity;
    }

    ~QuadraticHashMap()
    {
        delete[] map;
    }

    void insert(const T_struct<T_key, T_value>& toInsert)
    {
        if (full())
        {
            throw(errorType::FULL, errorType::errorString[FULL], 5);
        }

        int hashKey;
        std::string output = std::to_string(toInsert.key);
        int collisionCount = 0;
        bool stopHash = false;
        bool success = false;

        while (stopHash == false)
        {
            hashKey = QuadraticHash(toInsert.key, collisionCount);

            if (map[hashKey].label == EMPTY ||
                map[hashKey].label == AVAILABLE)
            {
                stopHash = true;
                success = true;

            }
            else
            {
                if (map[hashKey].key == toInsert.key)
                {
                    stopHash = true;
                    success = true;
                }

                collisionCount++;
            }

            output += "->" + std::to_string(hashKey);

        }


        if (success)
        {
            map[hashKey] = toInsert;
            map[hashKey].label = FULL;

            currentSize++;

            std::cout << "Inserting: " << '(' << toInsert.key << ", " <<
              toInsert.value <<  ')'
                << '\n' << "Hashed Key: "<< output << '\n' << '\n';
        }
    }

    void remove(const T_key key)
    {
        if (empty())
        {
            throw(errorType::FULL, errorType::errorString[FULL], 5);
        }

        int hashKey;
        std::string output = std::to_string(key);
        int collisionCount = 0;
        bool stopHash = false;
        bool success = false;
        while (stopHash == false)
        {
            hashKey = QuadraticHash(key, collisionCount);

            if (map[hashKey].label == FULL ||
                map[hashKey].label == AVAILABLE)
            {
                if (map[hashKey].key == key)
                {
                    stopHash = true;
                    success = true;
                }
                else
                {
                    collisionCount++;

                }
            }
            else
            {
                stopHash = true;
            }
            output += "->" + std::to_string(hashKey);
        }

        if (success)
        {
            map[hashKey].key = -1;
            map[hashKey].value = "";
            map[hashKey].label = AVAILABLE;

            currentSize--;

            std::cout << "Removing key: " << key << '\n'
                << "Hashed Key: " << output << '\n' << '\n';


        }


    }

    bool full()
    {
        return currentSize == capacity;
    }

    bool empty()
    {
        return currentSize == 0;
    }

    int size()
    {
        return currentSize;
    }

```

```
    void printAll(std::ostream& output)
    {
        if (empty())
        {
            throw(errorType::EMPTY, errorType::errorString[EMPTY], 5);
        }

        output << "  Index   | LABEL |  Key   |  Value" << '\n'
            << "_________|_______|_______|___________________"
            << '\n';

        for (int i = 0; i < capacity; i++)
        {
            output << std::right
                << " [" << std::setw(5) << i << "] | ";
            switch (map[i].label)
            {
            case EMPTY:
                output << "EMPTY |";
                break;

            case FULL:
                output << "FULL  |";
                break;

            case AVAILABLE:
                output << "AVAIL |";
                break;

            }
            output << ' ' << std::setw(4) << map[i].key << " |";

            output << std::left
                << ' ' << map[i].value
                << '\n';
        }
        output << "\n\n";
    }
};







#endif //!_DOUBLEHASH_H_
```