CISS350: Data Structures and Advanced Algorithms
Assignment 13

Objectives:

1. Implement as hashtable using separate chaining with `std::vector` and `std::list`.
2. Implement a set using a hashtable.
3. Implement a sparse matrix using a hashtable.

Warning: As always skeleton code is meant to get you started. Skeleton code needs to be corrected and completed.

SPRING 2019: OMIT Q2-Q5.

Submission. Email your work to `yliow@ccis.edu`. Use your college email account. The subject line of the email must be `ciss350 a13`.

Q1. Implementation of hashtable with separate chaining using linked lists

Implement a hashtable where each record has a key of type `std::string` and value of type `double`. Each bucket is a `std::list` of (key, value) pairs.

The following is skeleton code that you should begin with (HT is hashtable for short to save on writing):

```
#include <iostream>
#include <vector>
#include <list>

//============================================================================
// hash(std::string) -> unsigned int is a simple hash function for std::string
// objects.
//
// WARNING: Remmeber to mod by the size of your table!
//============================================================================
unsigned int hash(const std::string & s)
{
    unsigned int h = 0;
    unsigned int power = 1;
    for (unsigned int i = 0; i < s.length(); i++)
    {
        h += int(s[i]) * power;
        power *= 10;
    }
    return h;
}


//============================================================================
// Each record in the hash table is a (key, value) pair of type (std::string,
// double).
//============================================================================
class HTRow
{
public:
    // TODO
private:
    std::string key_;
    double value_;
};


//============================================================================
// This exception class is used to throw an exception in the event that a key
// is not found in the hash table during a search or a remove.
//============================================================================
class KeyError
```

```
{};


//=============================================================================
// Hashtable of (key, value) of type (std::string, double).
//
// When the percentage of get collisions is > threshold, the hashtable
// resizes by doubling its size. See num_gets_, num_gets_collisions_,
// and threshood_.
//
// The number of get() invocation is stored in _num_gets and the number
// of get collisions is stored in _num_get_collisions. The threshold for
// resize is stored in _threshold. A get collision is accessing a (key,
// value) pair that is not what you want. Therefore if you hash to a linked
// linked of three nodes and the node you really want is the last, then
// there are two collisions.
//=============================================================================
class HT
{
public:

    // The instance variable table is initialized to a vector of linked list
    // of HTRows objects. The size of the table is the given size.
    // When the percentage collision for get() operations is greater than
    // threshold, the table doubles its size.
    //
    // num_gets_ and num_get_collisions_ must be initialized to 0.
    // The default threshold is 0.75.
    HT(int size=10, double threshold=0.75)
    {}

    // 1. Search for record with key_ matching key
    // 2. If the record is found, modify value_ to value
    // 3. If the record is not found, insert a new record
    void set(std::string key, double value)
    {
    }

    // 1. Search for record with key_ matching key
    // 2. If the record is found, return the value
    // 3. If the record is not found, KeyError is thrown
    double get(const std::string & key)
    {
    }

    // 1. Removes record with key_ matching key
    // 2. If record is not found, KeyError is thrown
    void remove(const std::string & key)
    {
    }
```

```
    // Clear all entries in all linked lists.
    void clear()
    {
    }

    // Returns a std::vector<std::string> of keys
    std::vector<std::string> keys()
    {}

    // Resizes the hashtable
    void resize(int size)
    {
        // Resize the table (you need to recreate the hashtable structure.
        // Remember to reset num_gets_ and num_get_collisions_ to 0.
    }

private:
    std::vector< std::list< HTRow > > table_;

    int num_get_collisions_; // Records number of collisions in get()
    int num_gets_;           // Records number of get()
    double threshold_;       // If num_get_collisions_/num_gets_ > threshold
                             // (when num_gets_ > 0) the table must resize
};


std::ostream & operator<<(std::ostream & cout, const HT & d)
{
    // TODO
    return cout;
}

int main()
{
    HT d0;     // 10 linked lists (i.e. 10 buckets)
    HT d1(20); // 20 linked lists (i.e. 20 buckets)

    d0.set("John", 5.2);
    std::cout << d0 << std::endl; // prints {John:5.2}

    // Add more test cases to test your class.

    return 0;
}
```

Q2. Implementation of template hashtable with separate chaining

The hashtable in Q1 accepts keys of `std::string` type and values of `double` type. Modify the hashtable from Q1 so that is a class template of the following form:

```
class < typename Key, typename Value >
class HT
{
    ...
};
```

so that

```
HT< std::string, double >
```

becomes the class in Q1.

But wait!

Instead of having hash functions all over the place, it's better to tie hash functions into the classes that we're using as `Key`.

In fact, for practice, create a `Hashable` abstract base class and create a class `HashableString` that subclasses `std::string` and `Hashable`. `Hashable` has a pure virtual method `hash`. In that case

```
HT< HashableString, double >
```

is what you have in Q1. By doing it this way, you can have different hash functions for different hashtables for even the same key type. For instance suppoer you have a hashtable with keys made up of SSN and the SSNs you're putting into your hashtable always begin with the same 3 characters – you might want to ignore the first 3 characters when computing the hash.

[Note: You should also modify your `HTRow` class as well. Note also that for different `Key` types you need different hash functions.]

SPOILERS ON NEXT PAGE ...

```
#include <string>

class Hashable
{
public:
    virtual unsigned int hash() = 0;
};


class HashableString: public std::string, Hashable
{
public:
    HashableString(const std::string & s)
        : std::string(s)
    {}

    unsigned int hash()
    {
        unsigned int h = 0;
        unsigned int power = 1;
        for (unsigned int i = 0; i < length(); i++)
        {
            h += operator[](i) * power; power *= 10;
        }
        return h;
    }
};
```

Q3. Implementation of sets using hashtables

There are times when you might want a container to keep keys instead of (key, value) pairs. Think for instance of a set. A set (just like a set in your math classes – you first saw sets in precalc) is just a container of value where you want to:

1. Check if a value in the set (or not).
2. Add a value into the set.
3. Remove a value from the set.

(and possibly other basic operations such as the size of the set, etc.) Note that you are only interested in whether a value is in the set – you don't really care how many times a value occurs in the set. Therefore sets do not have duplicates. You also do not care about the ordering of values in the set – you never ask what is the value before or after this value.

You can redevelop the hashtable or you can also do this: For a set of `std::string` objects, you are looking at a hashtable with `std::string` for key type and your value type be any any type and you can just put in some dummy data for the value part. (Obviously you want to put in something that's small – like an integer. As a personal project, you can optimize by redoing Q1 without the value part.)

```
class < typename T >
class Set
{
public:

    // Adds x to d is x is not in d.
    void insert(const T & x)
    {}

    // Removes x from d if x is in d. Otherwise KeyError is thrown.
    void remove(const T & x)
    {}

    // Returns true if x is in d
    bool has_element(const T & x) const
    {}

    // Returns true if each value in d is also in set.d
    bool issubset(const Set< T > & set) const
    {}

    // Same as issubset
    bool operator<=(const Set< T > & set) const
    {}
```

```
    // Returns true if each value in set.d is also in d
    bool issuperset(const Set< T > & set) const
    {}

    // Same as issuperset
    bool operator>=(const Set< T > & set) const
    {}

    // Returns true if *this and set contains the same values
    bool operator==(const Set< T > & set) const
    {}

private:
    HT< T, char > d;
};

// If set contains "a", "b", print as {a, b}
template < typename T >
std::ostream & operator<<(std::stream & cout, const Set< T > set)
{}
```

Notes.

1. C++ STL set (`std::set`) is implemented as a tree and not hashtable.
2. Python implements its `set` class (and also `frozenset` class) by using its dictionary (which is a hashtable) with the value part occupied by some dummy value.

Q4. Comparing hashtable set and BST set

Using the implementation from Q3, create a set of 100,000 random integers and find the time taken to find 1,000 values in this set. Next, use your BST and do the same. Next repeat the above with a set of 200,000 random integers. Etc. Compare the time and plot a graph.

Q5. Implementation of sparse matrix using hashtable

Matrices are basically 2d arrays. They appear everywhere in CS and math. In real-life applications matrices are extremely huge, possibly going up a size of to thousands by thousands. Sometimes a matrix can be full of zeroes – this is very common in real life applications. Such matrices are called sparse matrices. There are many different ways to represent sparse matrices – see linked list notes for an implementation using doubly linked list. Another way to represent a sparse matrix is to use hashtables. In this case the key is the (row,column) of a value. For instance if the matrix is a 2-by-2:

$$\{\{5,0\},\{0,7\}\}$$

Then at (row,column) = $(0, 0)$, the value is 5: the key is $(0, 0)$ and the value is 5.

Create a class `SparseMatrix` with doubles as values.

```
SparseMatrix m(1000, 2000); // 1000-by-2000
m.set(10, 20, 3.1415);      // set value at row 10, column 20 to 3.1415
std::cout << m.get(10, 20); // prints 3.1415
std::cout << m.get(0, 0);   // prints 0 -- the default value
```

The implementation should use hashtables with separate chaining. (Modify Q1.) For the hash function, do the following: say the (row, column) is (4, 5). First convert the key to the string

```
"4,5"
```

(note the comma) and then use the method from Q1. Attempt to access a value with row or column outside the sizes specified using constructor call will result in the exception object from class `IndexError` being thrown.

Note several things:

- The runtime to access a value in the sparse matrix is (in the best case) $O(1)$ and the runtime to access the matrix as a 2D array is also $O(1)$.
- However if the matrix has a large rowsize and columnsize, then the 2d array method will use more memory.
- In fact if the rowsize and columnsize is too huge, the total size of the array (i.e., rowsize $\times$ columnsize) could be too big for the computer to handle – there's a limit to how much memory is allowed for an array. In this case the hashtable can actually work.