Contents

2 ex6/README.md

```
# ex6-michaelmerzin
```

3 ex6/Dictionary.hpp

```

// Created by yurim on 6/12/2022.
//
#include "HashMap.hpp"
#ifndef _DICTIONARY_HPP_
#define _DICTIONARY_HPP_
class InvalidKey:public std::invalid_argument
{
 public:
  InvalidKey():std::invalid_argument("")
  {}
  InvalidKey(const std::string &a):std::invalid_argument(a)
  {}
};
 class Dictionary: public HashMap<std::string ,std::string>
{
 private:
 public :
  Dictionary()= default;
  Dictionary(std::vector<std::string> first, std::vector<std::string> second)
  {
   if(first.size()!=second.size())
   {
     throw std::out_of_range ("Error size");
   }
    for (unsigned  i = 0; i < first.size(); ++i)
    {
      this->insert (first[i],second[i]);
    }

  }
  bool erase(std::string key) override
  {
      bool check = HashMap<std::string, std::string>::erase(key);
      if(!check)
      {
        throw InvalidKey("error");
      }
      return true;
  }
   template<class Itertaor>
  void update(Itertaor a
              ,Itertaor b)
  {
    for (auto i = a; i !=b; ++i)
    {
      std::hash<std::string> hash_func;
      int check = hash_func (i->first) & (capacity1-1);
      bool check2 =this->contains_key (i->first);
      if(check2)
      {

        for (unsigned  j = 0; j < this->table[check].size(); ++j)
        {
          if( this->table[check][j].first==i->first)
          {
            this->table[check][j].second=i->second;
          }
        }

      }
      else
      {
        table[check].push_back ({i->first,i->second});
        M++;
      }
    }

  }
};

#endif //_DICTIONARY_HPP_
```

4 ex6/HashMap.hpp

```
#ifndef _HASHMAP_HPP_
#define _HASHMAP_HPP_
#include <algorithm>
#include <utility>
#include <iostream>
#include <vector>
#define DEFSIZE 16
#define LOWER 0.25
#define UPPER 0.75
#define A12 12
template<typename KeyT, typename ValT>
class HashMap
{
 protected:
  int capacity1;
  int M;
  std::vector<std::pair<KeyT, ValT >> *table;
 public :
  friend class Iterator;
  friend class ConstIterator;
  HashMap ()
  {
    table = new std::vector<std::pair<KeyT, ValT>>[DEFSIZE];
    capacity1 = DEFSIZE;
    M = 0;

  }
  HashMap (HashMap<KeyT, ValT> const &a)
  {
    capacity1 = a.capacity1;
    M = a.M;
    this->table = new std::vector<std::pair<KeyT, ValT>>[a.capacity1];
    for (int i = 0; i < a.capacity1; ++i)
    {
      table[i] = a.table[i];
    }

  }
  void help_when_if(std::vector<KeyT> first, std::vector<ValT> second)
  {
    table = new std::vector<std::pair<KeyT, ValT>>[DEFSIZE];
    capacity1 = DEFSIZE;
    M = 0;
    for (unsigned int i = 0; i < first.size (); i++)
    {
      int check = 0;
      std::hash<KeyT> hash_func;
      int check2 = hash_func (first[i]) & (capacity1-1);

      int size_help = table[check2].size ();
      for (int j = 0; j < size_help; ++j)
      {
        KeyT helper = table[check2][j].first;
        if (helper == first[i])
        {
          if (table[check2][j].second != second[i])
          {
            table[check2][j].second = second[i];
          }
          check++;
        }

      }
      if (check == 0)
      {
        int index = hash_func (first[i]) & (capacity1-1);

        table[index].push_back ({first[i], second[i]});
        M++;
      }
    }
    if (M > A12)
    {
      make_new_hash ();
    }
  }
  void help_when_else(std::vector<KeyT> first, std::vector<ValT> second)
  {
    capacity1 = new_size (first.size ());
    table = new std::vector<std::pair<KeyT, ValT>>[capacity1];
    M = 0;
    for (int i = 0; i < DEFSIZE; i++)
    {
      for (unsigned int i = 0; i < first.size (); i++)
      {
        int check = 0;
        std::hash<KeyT> hash_func;
        int size_help = table[hash_func (first[i]) & (capacity1-1)].size ();
        for (int j = 0; j < size_help; ++j)
        {
          KeyT helper = table[hash_func (first[i]) & (capacity1-1)][j].first;
          if (helper == first[i])
          {
            if (table[hash_func (first[i]) & (capacity1-1)][j].second
                != second[i])
            {
              table[hash_func (first[i]) &
              (capacity1-1)][j].second = second[i];
            }
            check++;
          }
        }
        if (check == 0)
        {
          table[hash_func (first[i]) & (capacity1-1)].push_back ({first[i],
                                                                 second[i]});
          M++;
        }
      }
    }
  }
  int find_first_size (int size)
  {
    int x = 2;
    while (x < size || x <= capacity1)
    {
      x = x * 2;
    }
    return x;
  }
  HashMap (std::vector<KeyT> first, std::vector<ValT> second)
  {
    if (first.size () != second.size ())
    {
      throw std::out_of_range ("Error size");
    }
    if (first.size () <= DEFSIZE)
    {
      help_when_if (first,second);
    }
    else
    {
      help_when_else( first,second);
    }
  }
  int new_size (int vec_size)
  {
    vec_size++;
    int x = 2;
    if (get_load_factor () > UPPER)
    {
      return capacity1 * 2;

    }
    if (get_load_factor () < LOWER)
    {
      if (capacity1 == 1)
      { return 1; }
      return capacity1 / 2;

    }
    return x;
  }
  virtual ~HashMap ()/// check if  it must
/// be arry or it can be vector
  {
    if(table!=NULL){delete[] table;}

  }

  int size () const
  {
    return M;
  }

  int capacity () const
  {
    return capacity1;
  }

  bool empty () const
  {
    return M == 0;
  }
  bool contains_key (const KeyT& key2)const
  {
    std::hash<KeyT> hash_func;
    int check = hash_func (key2) & (capacity1-1);


    for (unsigned int j = 0; j < table[check].size (); ++j)
    {
      if (table[check][j].first == key2)
      {
        return true;
      }
    }

    return false;
  }
  void make_new_hash ()
  {
    int new_size1 = new_size (M);
    std::vector<std::pair<KeyT, ValT >> *new_table =
        new std::vector<std::pair<KeyT, ValT>>[new_size1];
    for (int i = 0; i < capacity1; ++i)
    {
      for (unsigned int j = 0; j < table[i].size (); ++j)
      {
        if (table[i].size () != 0)
        {
          KeyT thekey = table[i][j].first;
          std::hash<KeyT> hash_func;
          int check = hash_func (thekey) & (new_size1-1);

          new_table[check].push_back
              ({table[i][j].first, table[i][j].second});
        }

      }

    }
    delete[] table;
    this->capacity1 = new_size1;
    this->table = new_table;
  }


  ValT &at (const KeyT &key)
  {
    std::hash<KeyT> hash_func;
    int check = hash_func (key) & (capacity1-1);


    for (std::pair<KeyT, ValT>& elem :  table[check])
    {
      if (elem.first == key)
      {
        return elem.second;
      }
    }

    throw std::out_of_range ("Error size");

  }
  ValT at (const KeyT& key)const
  {
    std::hash<KeyT> hash_func;
    int check = hash_func (key) & (capacity1-1);

    int vecsize = table[check].size ();
    for (int i = 0; i < vecsize; i++)
    {
      if (table[check][i].first == key)
      {
        return table[check][i].second;
      }
    }

    throw std::out_of_range ("Error size");

  }
  bool insert (const KeyT& key, const ValT& value)
  {
    if (!contains_key(key))
    {
      std::hash<KeyT> hash_func;
      int check2 = hash_func (key) & (capacity1-1);

      std::pair<KeyT,ValT> pair;
      pair.first=key;
      pair.second=value;
      if(value==ValT())
      {table[check2].push_back (std::make_pair(key,ValT()));}
      else{table[check2].push_back (std::make_pair(key,value));}

      M++;
      if (get_load_factor () > UPPER)
      {
        make_new_hash ();
      }
      return true;
    }
    else
    {

      return false;
    }
  }

  virtual bool erase (KeyT key)
  {
    std::hash<KeyT> hash_func;
    int check = hash_func (key) & (capacity1-1);
    typename std::vector<std::pair<KeyT, ValT>>::
    iterator del = this->table[check].end ();
    int check_if_find = 0;
    for (auto i = table[check].begin (); i != table[check].end (); i++)
    {
      if (key == i->first)
      {
        del = i;
        check_if_find++;
      }
    }
    if (check_if_find != 0)
    {
      table[check].erase (del);
      M--;

      double fac = get_load_factor ();
      if (fac < LOWER)
      {
        make_new_hash ();
      }
      return true;
    }
    return false;
  }

  double get_load_factor ()const
  {
    return (double) M / capacity1;
  }
  void clear ()
  {
    for (int i = 0; i < capacity1; ++i)
    {
      M = M - table[i].size ();
      table[i].clear ();

    }
  }
  int bucket_size (const KeyT& key)const
  {
    std::hash<KeyT> hash_func;
    int check = hash_func (key) & (capacity1-1);
    return table[check].size ();
  }

  int bucket_index (const KeyT &key)const
  {
    std::hash<KeyT> hash_func;
    return hash_func (key) & (capacity1-1);
  }

  ValT operator[] (const KeyT &key) const
  {
    std::hash<KeyT> hash_func;
    int check_if_we_found = 0;
    int check = hash_func (key) & (capacity1-1);

    for (int i = 0; i < table[check].size (); ++i)
    {
      if (table[check][i].first == key)
      {
        check_if_we_found++;
        return table[check][i].second;
      }
    }
    return ValT();
  }

  ValT &operator[] (const KeyT &key)
  {

    if(!contains_key(key))
    {
      insert (key, ValT());
    }

    ValT &ret = this->at (key);
    return ret;

  }

  HashMap<KeyT, ValT> &operator= (const HashMap a)
  {
    if(a==*this)
    {return *this;}
    this->capacity1 = a.capacity1;
    this->M = a.M;
    delete[] this->table;
    table = new std::vector<std::pair<KeyT, ValT>>[capacity1];
    for (int i = 0; i < a.capacity1; ++i)
    {
      table[i] = a.table[i];

    }
    return *this;
  }
//  bool equalhelp(const HashMap a)
//  {}
  bool operator== (const HashMap a) const
  {
    std::hash<KeyT> hash_func;
    if (this->M != a.M)
    {return false;}
    for( int i = 0;i<capacity1;i++)
    {
      for (unsigned int j = 0; j < table[i].size(); ++j)
      {
        int check_if_found=0;
        int check = hash_func (table[i][j].first) & (a.capacity1-1);
        for (unsigned int k = 0; k < a.table[check].size(); ++k)
        {
          if(table[i][j].first==a.table[check][k].first&&
             table[i][j].second==a.table[check][k].second)
          {check_if_found++;}
        }
        if(check_if_found==0)
        {return false;}
      }
    }
    for( int i = 0;i<a.capacity1;i++)
    {
      for (unsigned int j = 0; j < a.table[i].size(); ++j)
      {
        int check_if_found=0;
        int check = hash_func (a.table[i][j].first) & (a.capacity1-1);

        for (unsigned int k = 0; k < table[check].size(); ++k)
        {
          if(a.table[i][j].first==table[check][k].first&&a.
              table[i][j].second==table[check][k].second)
          {
            check_if_found++;
          }
        }
        if(check_if_found==0)
        {return false;}
      }
    }
    return true;
  }
  bool operator!= (const HashMap a) const
  {
    if (this->capacity1 != a.capacity1)
    {
      return true;
    }
    for (int i = 0; i < capacity1; ++i)
    {
      if (table[i].size () != a.table[i].size ())
      { return true; }
      if (table[i] != a.table[i])
      { return true; }

    }
    return false;

  }


  class ConstIterator
  {

    friend class Hashmap;
    int index_for_vec1;
    int index_in_vec1;
    const HashMap &a;

   public:

    typedef std::pair<KeyT, ValT> value_type;
    typedef std::pair<KeyT, ValT> &reference;
    typedef std::pair<KeyT, ValT> *pointer;
    typedef std::ptrdiff_t difference_type;
    typedef std::forward_iterator_tag iterator_category;

    // Constructor
    ConstIterator (int index_for_vec, int index_in_vec, const HashMap &Ha)
        : index_for_vec1 (index_for_vec), index_in_vec1 (index_in_vec),a (Ha)
    {

    }
    ConstIterator &operator++ ()
    {
      if (index_for_vec1 == a.capacity1)
      {
        int check2 = -1;
        for (int i = 0; i < a.size (); ++i)
        {
          if (a.size () != 0 && check2 == -1)
          {
            check2++;
            index_for_vec1 = i;
          }
        }

      }
      if (a.capacity1 == index_for_vec1
          && index_in_vec1 == a.table[index_for_vec1].size () - 1)
      {
        std::cout << "error";
      }
      if (index_in_vec1 < a.table[index_for_vec1].size () - 1)
      {
        index_in_vec1++;
        return *this;
      }
      if (index_in_vec1 == a.table[index_for_vec1].size ()
                           - 1)/// check if we in the end of the vector
      {
        int check = -1;
        //// check if we find new vector
        for (int i = index_for_vec1 + 1; i < a.capacity1; ++i)
        {
          if (check == -1 && a.table[i].size () != 0)
          {
            check = i;
            index_in_vec1 = 0;
            index_for_vec1 = i;
          }
        }
        if (check == -1)
        {
          this->index_for_vec1= a.capacity1 + 1;
          this->index_in_vec1 = 0;
          return *this;
        }
      }
      return *this;
    }

    ConstIterator operator++ (int)
    {
      if (index_for_vec1 == a.capacity1)
      {
        int check2 = -1;
        for (int i = 0; i < a.size (); ++i)
        {
          if (a.size () != 0 && check2 == -1)
          {
            check2++;
            index_for_vec1 = i;
          }
        }

      }
      if (a.capacity1 == index_for_vec1
          && index_in_vec1 == a.table[index_for_vec1].size () - 1)
      {
        std::cout << "error";
      }
      if (index_in_vec1 < a.table[index_for_vec1].size () - 1)
      {
        index_in_vec1++;
        return *this;
      }
      if (index_in_vec1 == a.table[index_for_vec1].size ()
                           - 1)/// check if we in the end of the vector
      {
        int check = -1;
        //// check if we find new vector
        for (int i = index_for_vec1 + 1; i < a.capacity1; ++i)
        {
          if (check == -1 && a.table[i].size () != 0)
          {
            check = i;
            index_in_vec1 = 0;
            index_for_vec1 = i;
          }
        }
        if (check == -1)
        {
          this->index_for_vec1= a.capacity1 + 1;
          this->index_in_vec1 = 0;
          return *this;
        }
      }
      return *this;
    }

    bool operator== (const ConstIterator &rhs) const
    {

      return index_for_vec1 == rhs.index_for_vec1
             && index_in_vec1 == rhs.index_in_vec1;
    }

    bool operator!= (const ConstIterator &rhs) const
    {
      return index_for_vec1 != rhs.index_for_vec1
             || index_in_vec1 != rhs.index_in_vec1||rhs.a!=this->a;
    }

    reference operator* ()
    {
      return a.table[index_for_vec1][index_in_vec1];
    }

    pointer operator-> ()
    { return &(operator* ()); }
  };

  // using iterator = Iterator;
  using const_iterator = ConstIterator;

//  Iterator begin ()
//  {
//    if (M == 0)
//    {
//      return Iterator (capacity1 + 1, 0, *this);
//    }
//    for (unsigned int i = 0; i < table->size (); ++i)
//    {
//      if (table[i].size () != 0)
//      {
//        return Iterator (i, 0, *this);
//      }
//    }
//    return Iterator (capacity1 + 1, 0, *this);
```

```
//
//
//  }

  const_iterator begin () const
  {

    if (M == 0)
    {
      return const_iterator (capacity1 + 1, 0, *this);
    }
    for ( int i = 0; i < this->capacity1; ++i)
    {
      if (table[i].size () != 0)
      {
        return const_iterator (i, 0, *this);
      }
    }
    return const_iterator (capacity1 + 1, 0, *this);

  }

  const_iterator cbegin () const
  {
    if (M == 0)
    {
      return const_iterator (capacity1 + 1, 0, *this);
    }
    for ( long unsigned int i = 0; i < table->capacity(); ++i)
    {
      if (table[i].size () != 0)
      {
        return const_iterator (i, 0, *this);
      }
    }
    return const_iterator (capacity1 + 1, 0, *this);
  }

//  Iterator end ()
//  { return Iterator (capacity1 + 1, 0, *this); }
  const_iterator end () const
  { return const_iterator (this->capacity1 + 1, 0, *this); }
  const_iterator cend () const
  { return const_iterator (this->capacity1 + 1, 0, *this); }



};


#endif //_HASHMAP_HPP_
```