**ENSF 408**

**Lab 3**

**Shamin Rahman(30037908)**

**11 oct 2019**

**B01**

**EX A:**

Output:

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Source code: (iterator.cpp)**

*// iterator.cpp*

*// ENSF 480 - Fall 2019 - Lab 3, Ex A*

*// M. Moussavi: Sept 26, 2018*

#include <iostream>

#include <assert.h>

#include <string>

#include <string.h>

#include "mystring2.h"

using namespace std;

template <class T>

class Vector{

public:

    class VectIter{

        friend class Vector<T>;

    private:

        Vector<T> \*v;*// points to a vector object of type T*

        int index;*// represents the subscript number of the vector's*

*// array.*

    public:

        VectIter(Vector<T> &x);

        T operator++();*// same as ++x*

*//PROMISES: increments the iterator's indes and return the*

*//          value of the element at the index position. If*

*//          index exceeds the size of the array it will*

*//          be set to zero. Which means it will be circulated*

*//          back to the first element of the vector.*

        T operator++(int);*//same as x++*

*// PRIMISES: returns the value of the element at the index*

*//           position, then increments the index. If*

*//           index exceeds the size of the array it will*

*//           be set to zero. Which means it will be circulated*

*//           back to the first element of the vector.*

        T operator--();*//same as --X*

*// PROMISES: decrements the iterator index, and return the*

*//           the value of the element at the index. If*

*//           index is less than zero it will be set to the*

*//           last element in the aray. Which means it will be*

*//           circulated to the last element of the vector.*

        T operator--(int);

*// PRIMISES: returns the value of the element at the index*

*//           position, then decrements the index. If*

*//           index is less than zero it will be set to the*

*//           last element in the aray. Which means it will be*

*//           circulated to the last element of the vector.*

        T operator\*();

*// PRIMISES: returns the value of the element at the current*

*//           index position.*

    };

    Vector(int sz);

    ~Vector();

    int &operator[](int i);

*// PRIMISES: returns existing value in the ith element of*

*//           array or sets a new value to  the ith element in*

*//           array.*

    void ascending\_sort();

*// PRIMISES: sorts the vector values in ascending order.*

private:

    T \*array;*// points to the first element of an array of T*

    int size;*// size of array*

    void swap(T &, T &);*// swaps the values of two elements in array*

};

template<>

class Vector<const char\*>{

    private:

    int size;

    const char\*\* array;

    public:

    Vector(int sz);

    ~Vector();

    void ascending\_sort();

    void swap(const char\*  &a, const char\* &b);

    const char\*& operator[](int i){

        return array[i];

    }

*// const char\*& operator =(const char\* rhs){*

*// }*

    class VectIter{

        friend class Vector<const char\*>;

        private:

        Vector<const char\*> \*v;*// points to a vector object of type T*

        int index;

        public:

        VectIter(Vector<const char\*> &x){

            v = &x;

            index = 0;

        }

        const char\* operator++(){*//++x*

            index++;

            if(index >= v->size)

                index = 0;

            return v->array[index];

        }

        const char\* operator++(int){*//x++*

            const char\* res = v->array[index];

            index++;

            if(index >= v->size)

                index = 0;

            return res;

        }

        const char\* operator--(){*//--x*

            index--;

            if(index < 0)

                index = 0;

            return v->array[index];

        }

        const char\* operator--(int){*//x--*

            const char\* res = v->array[index];

            index--;

            if(index < 0)

                index = 0;

            return res;

        }

        const char\* operator\*(){

            return v->array[index];

        }

    };

};

template<>

class Vector<Mystring>{

    private:

    int size;

    Mystring \*array;

    public:

    Vector(int sz);

    ~Vector();

    friend ostream& operator << (ostream &out, const Mystring &c);

    void ascending\_sort();

    void swap(Mystring  &a, Mystring &b);

    Mystring &operator[](int i){

        return array[i];

    }

    class VectIter{

        friend class Vector<Mystring>;

        private:

        Vector<Mystring> \*v;*// points to a vector object of type T*

        int index;

        public:

        VectIter(Vector<Mystring> &x){

            v = &x;

            index = 0;

        }

        Mystring operator++(){*//++x*

            index++;

            if(index >= v->size)

                index = 0;

            return v->array[index];

        }

        Mystring operator++(int){*//x++*

            Mystring res = v->array[index];

            index++;

            if(index >= v->size)

                index = 0;

            return res;

        }

        Mystring operator--(){*//--x*

            index--;

            if(index < 0)

                index = v->size - 1;

            return v->array[index];

        }

        Mystring operator--(int){*//x--*

            Mystring res = v->array[index];

            index--;

            if(index < 0)

                index =  v->size - 1;

            return res;

        }

        Mystring operator\*(){

            return v->array[index];

        }

    };

};

*// ostream& operator << (ostream &out, const Mystring &c){*

*//  out << c.c\_str();*

*//  return out;*

*// }*

Vector<Mystring>::Vector(int sz){

    size = sz;

    array = new Mystring[size];

    assert(array != nullptr);

*//// strcpy(array, sz);*

}

Vector<Mystring>::~Vector(){

*// for(int i = 0; i < size; i++){*

*//  Mystring\* yuh = array[0];*

*//  delete [] yuh;*

*// }*

    delete [] array;

    array = NULL;

}

Vector<const char\*>::Vector(int sz){

    size = sz;

    array = new const char\*[size];

    assert(array != nullptr);

*//// strcpy(array, sz);*

}

Vector<const char\*>::~Vector(){

*// for(int i = 0; i < size; i++){*

*//  delete [] array[i];*

*// }*

    delete [] array;

    array = NULL;

}

void Vector<const char\*>::ascending\_sort(){

    for (int i = 0; i < size - 1; i++)

        for (int j = i + 1; j < size; j++)

            if (strcmp(array[i], array[j]) > 0)*//array[i] > array[j]*

                swap(array[i], array[j]);

}

void Vector<Mystring>::ascending\_sort(){

    for (int i = 0; i < size - 1; i++)

        for (int j = i + 1; j < size; j++)

            if (array[i].isGreater(array[j]))*//array[i] > array[j]*

                swap(array[i], array[j]);

}

template <class T>

T Vector<T>::VectIter::operator++()

{*// same as ++x*

    index++;

    if (index >= v->size)

    {

        index = 0;

    }

    return v->array[index];

}

template <class T>

T Vector<T>::VectIter::operator++(int)

{*// same as x++*

    int result = v->array[index];

    index++;

    if (index >= v->size)

    {

        index = 0;

    }

    return result;

}

template <class T>

T Vector<T>::VectIter::operator--()

{*//same as --X*

    index--;

    if (index < 0)

    {

        index = v->size - 1;

    }

    return v->array[index];

}

template <class T>

T Vector<T>::VectIter::operator--(int)

{*//same as X--*

    int result = v->array[index];

    index--;

    if (index < 0)

    {

        index = v->size - 1;

    }

    return result;

}

template <class T>

T Vector<T>::VectIter::operator\*()

{

    return v->array[index];

}

template <class T>*//TODO prob have to do specialization here*

void Vector<T>::ascending\_sort()

{

    for (int i = 0; i < size - 1; i++)

        for (int j = i + 1; j < size; j++)

            if (array[i] > array[j])

                swap(array[i], array[j]);

}

template <class T>

void Vector<T>::swap(T &a, T &b)

{

    T tmp = a;

    a = b;

    b = tmp;

}

*// template <>*

void Vector<Mystring>::swap(Mystring &a, Mystring &b){

    Mystring tmp = a;

    a = b;

    b = tmp;

}

void Vector<const char\*>::swap(const char\* &a, const char\* &b){

    const char\* tmp = a;

    a = b;

    b = tmp;

}

template <class T>

Vector<T>::VectIter::VectIter(Vector<T> &x)

{

    v = &x;

    index = 0;

}

template <class T>

Vector<T>::Vector(int sz)

{

    size = sz;

    array = new T[sz];

    assert(array != NULL);

}

template <class T>

Vector<T>::~Vector()

{

    delete[] array;

    array = NULL;

}

template <class T>

int &Vector<T>::operator[](int i)

{

    return array[i];

}

int main()

{

    Vector<int> x(3);

    x[0] = 999;

    x[1] = -77;

    x[2] = 88;

    Vector<int>::VectIter iter(x);

    cout << "\n\nThe first element of vector x contains: " << \*iter;

*// the code between the  #if 0 and #endif is ignored by*

*// compiler. If you change it to #if 1, it will be compiled*

#if 1

    cout << "\nTesting an <int> Vector: " << endl;;

    cout << "\n\nTesting sort";

    x.ascending\_sort();

    for (int i=0; i<3 ; i++)

        cout << endl << iter++;

    cout << "\n\nTesting Prefix --:";

    for (int i=0; i<3 ; i++)

        cout << endl << --iter;

    cout << "\n\nTesting Prefix ++:";

    for (int i=0; i<3 ; i++)

        cout << endl << ++iter;

    cout << "\n\nTesting Postfix --";

    for (int i=0; i<3 ; i++)

        cout << endl << iter--;

    cout << endl;

    cout << "Testing a <Mystring> Vector: " << endl;

    Vector<Mystring> y(3);

    y[0] = "Bar";

    y[1] = "Foo";

    y[2] = "All";;

    Vector<Mystring>::VectIter iters(y);

    cout << "\n\nTesting sort";

    y.ascending\_sort();

    for (int i=0; i<3 ; i++)

        cout << endl << iters++;

*// cout << endl << y;*

    cout << "\n\nTesting Prefix --:";

    for (int i=0; i<3 ; i++)

        cout << endl << --iters;

    cout << "\n\nTesting Prefix ++:";

    for (int i=0; i<3 ; i++)

        cout << endl << ++iters;

    cout << "\n\nTesting Postfix --";

    for (int i=0; i<3 ; i++)

        cout << endl << iters--;

    cout << endl; cout << "Testing a <char \*> Vector: " << endl;

    Vector<const char\*> z(3);

    z[0] = "Orange";

    z[1] = "Pear";

    z[2] = "Apple";

*// creating copy of z called iterchar*

    Vector<const char\*>::VectIter iterchar(z);

    cout << "\n\nTesting sort";

    z.ascending\_sort();

    for (int i=0; i<3 ; i++)

        cout << endl << iterchar++;

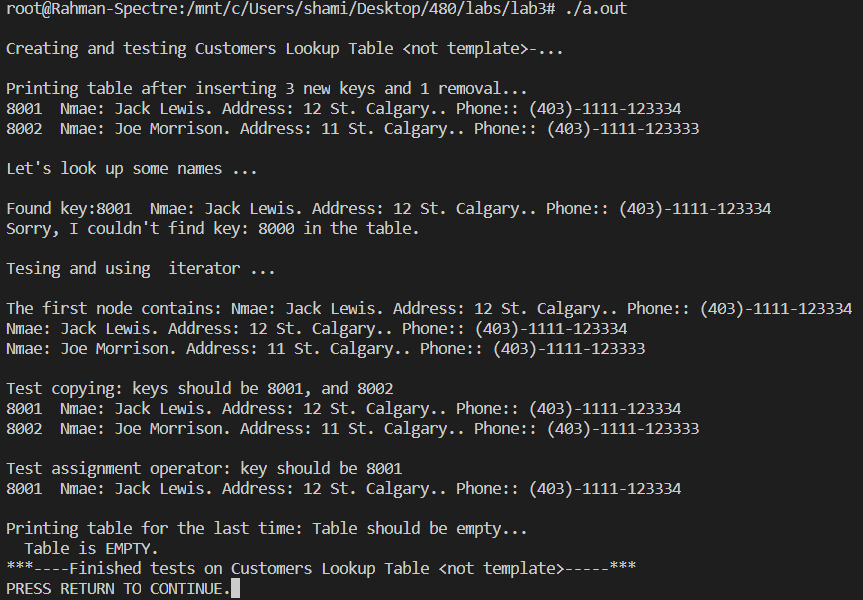
#endif

    cout << "\nPrgram Terminated Successfully." << endl;

    return 0;

}

**EX B**



A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Source code: {lookuptable.h}**

*// LookupTable.h*

*// ENSF 480 - Fall 2019 - Lab 3, Ex A*

#ifndef LOOKUPTABLE\_H

#define LOOKUPTABLE\_H

#include <iostream>

using namespace std;

*// class LookupTable: GENERAL CONCEPTS*

*//*

*//    key/datum pairs are ordered.  The first pair is the pair with*

*//    the lowest key, the second pair is the pair with the second*

*//    lowest key, and so on.  This implies that you must be able to*

*//    compare two keys with the < operator.*

*//*

*//    Each LookupTable has an embedded iterator class that allows users*

*//    of the class to traverse trhough the list and  have acess to each*

*//    node.*

#include "customer.h"

template <class T, class K> class LookupTable;

template <class T, class K> struct Pair;

template <class T, class K> class LT\_Node;

*//    In this version of the LookupTable a new struct type called Pair*

*//    is introduced which represents a key/data pair.*

typedef int LT\_Key;

typedef Customer LT\_Datum;

template <class T, class K>

struct Pair{

    Pair(K keyA, T datumA) : key(keyA), datum(datumA){}

    K key;

    T datum;

};

template <class T, class K>

class LT\_Node{

    friend class LookupTable<T, K>;

private:

    Pair<T, K> pairM;

    LT\_Node \*nextM;

*// private:*

*// This ctor should be convenient in insert and copy operations.*

    LT\_Node(const Pair<T, K> &pairA, LT\_Node<T, K> \*nextA);

};

template <class T, class K>

class LookupTable{

    private:

    int sizeM;

    LT\_Node<T, K> \*headM;

    LT\_Node<T, K> \*cursorM;

    void destroy();

*// Deallocate all nodes, set headM to zero.*

    void copy(const LookupTable<T, K> &source);

*// Establishes \*this as a copy of source.  Cursor of \*this will*

*// point to the twin of whatever the source's cursor points to.*

    public:

    class Iterator{

        friend class LookupTable;

        LookupTable<T, K> \*LT;

*//    LT\_Node\* cursor;*

        public:

        Iterator() : LT(0) {}

        Iterator(LookupTable<T, K> &x) : LT(&x) {}

        const T &operator\*();

        const T &operator++();

        const T &operator++(int);

        int operator!();*//TODO dsas*

        void step\_fwd(){

            assert(LT->cursor\_ok());

            LT->step\_fwd();

        }

    };

    LookupTable();

    LookupTable(const LookupTable<T, K> &source);

    LookupTable<T, K> &operator=(const LookupTable<T, K> &rhs);

    ~LookupTable();

    LookupTable &begin();

    int size() const;

*// PROMISES: Returns number of keys in the table.*

    int cursor\_ok() const;

*// PROMISES:*

*//   Returns 1 if the cursor is attached to a key/datum pair,*

*//   and 0 if the cursor is in the off-list state.*

    const K &cursor\_key() const;

*// REQUIRES: cursor\_ok()*

*// PROMISES: Returns key of key/datum pair to which cursor is attached.*

    const T &cursor\_datum() const;

*// REQUIRES: cursor\_ok()*

*// PROMISES: Returns datum of key/datum pair to which cursor is attached.*

    void insert(const Pair<T, K> &pariA);

*// PROMISES:*

*//   If keyA matches a key in the table, the datum for that*

*//   key is set equal to datumA.*

*//   If keyA does not match an existing key, keyA and datumM are*

*//   used to create a new key/datum pair in the table.*

*//   In either case, the cursor goes to the off-list state.*

    void remove(const K &keyA);

*// PROMISES:*

*//   If keyA matches a key in the table, the corresponding*

*//   key/datum pair is removed from the table.*

*//   If keyA does not match an existing key, the table is unchanged.*

*//   In either case, the cursor goes to the off-list state.*

    void find(const K &keyA);

*// PROMISES:*

*//   If keyA matches a key in the table, the cursor is attached*

*//   to the corresponding key/datum pair.*

*//   If keyA does not match an existing key, the cursor is put in*

*//   the off-list state.*

    void go\_to\_first();

*// PROMISES: If size() > 0, cursor is moved to the first key/datum pair*

*//   in the table.*

    void step\_fwd();

*// REQUIRES: cursor\_ok()*

*// PROMISES:*

*//   If cursor is at the last key/datum pair in the list, cursor*

*//   goes to the off-list state.*

*//   Otherwise the cursor moves forward from one pair to the next.*

    void make\_empty();

*// PROMISES: size() == 0.*

*// friend  ostream& operator << <K,D> (ostream& os,const LookupTable<K,D>& lt);*

    friend ostream &operator << (ostream &os, const LookupTable<T, K> &lt){

        if (lt.cursor\_ok())

            os << lt.cursor\_key() << "  " << lt.cursor\_datum();

        else

            os << "Not Found.";

        return os;

    }

};

#endif

template <class T, class K>

LookupTable<T, K>& LookupTable<T, K>::begin()

{

    cursorM = headM;

    return \*this;

}

template <class T, class K>

LT\_Node<T, K>::LT\_Node(const Pair<T, K> &pairA, LT\_Node<T, K> \*nextA)

        : pairM(pairA), nextM(nextA)

{

}

template <class T, class K>

LookupTable<T, K>::LookupTable(): sizeM(0), headM(0), cursorM(0){}

template <class T, class K>

LookupTable<T, K>::LookupTable(const LookupTable<T, K> &source){

    copy(source);

}

template <class T, class K>

LookupTable<T, K> &LookupTable<T, K>::operator=(const LookupTable<T, K> &rhs){

    if (this != &rhs){

        destroy();

        copy(rhs);

    }

    return \*this;

}

template <class T, class K>

LookupTable<T, K>::~LookupTable(){

    destroy();

}

template <class T, class K>

int LookupTable<T, K>::size() const{

    return sizeM;

}

template <class T, class K>

int LookupTable<T, K>::cursor\_ok() const{

    return cursorM != 0;

}

template <class T, class K>

const K &LookupTable<T, K>::cursor\_key() const{

    assert(cursor\_ok());

    return cursorM->pairM.key;

}

template <class T, class K>

const T &LookupTable<T, K>::cursor\_datum() const{

    assert(cursor\_ok());

    return cursorM->pairM.datum;

}

template <class T, class K>

void LookupTable<T, K>::insert(const Pair<T, K> &pairA){

*// Add new node at head?*

    if (headM == 0 || pairA.key < headM->pairM.key){

        headM = new LT\_Node<T, K>(pairA, headM);

        sizeM++;

    }

*// Overwrite datum at head?*

    else if (pairA.key == headM->pairM.key)

        headM->pairM.datum = pairA.datum;

*// Have to search ...*

    else{

        LT\_Node<T, K> \*before = headM;

        LT\_Node<T, K> \*after = headM->nextM;

        while (after != NULL && (pairA.key > after->pairM.key))

        {

            before = after;

            after = after->nextM;

        }

        if (after != NULL && pairA.key == after->pairM.key)

        {

            after->pairM.datum = pairA.datum;

        }

        else

        {

            before->nextM = new LT\_Node<T, K>(pairA, before->nextM);

            sizeM++;

        }

    }

}

template <class T, class K>

void LookupTable<T, K>::remove(const K &keyA){

    if (headM == 0 || keyA < headM->pairM.key)

        return;

    LT\_Node<T, K> \*doomed\_node = 0;

    if (keyA == headM->pairM.key)

    {

        doomed\_node = headM;

        headM = headM->nextM;

        sizeM--;

    }

    else

    {

        LT\_Node<T, K> \*before = headM;

        LT\_Node<T, K> \*maybe\_doomed = headM->nextM;

        while (maybe\_doomed != 0 && keyA > maybe\_doomed->pairM.key)

        {

            before = maybe\_doomed;

            maybe\_doomed = maybe\_doomed->nextM;

        }

        if (maybe\_doomed != 0 && maybe\_doomed->pairM.key == keyA)

        {

            doomed\_node = maybe\_doomed;

            before->nextM = maybe\_doomed->nextM;

            sizeM--;

        }

    }

    delete doomed\_node;*// Does nothing if doomed\_node == 0.*

}

template <class T, class K>

void LookupTable<T, K>::find(const K &keyA){

    LT\_Node<T, K> \*ptr = headM;

    while (ptr != NULL && ptr->pairM.key != keyA)

    {

        ptr = ptr->nextM;

    }

    cursorM = ptr;

}

template <class T, class K>

void LookupTable<T, K>::go\_to\_first(){

    cursorM = headM;

}

template <class T, class K>

void LookupTable<T, K>::step\_fwd(){

    assert(cursor\_ok());

    cursorM = cursorM->nextM;

}

template <class T, class K>

void LookupTable<T, K>::make\_empty(){

    destroy();

    sizeM = 0;

    cursorM = 0;

}

template <class T, class K>

void LookupTable<T, K>::destroy(){

    LT\_Node<T, K> \*ptr = headM;

    while (ptr != NULL)

    {

        headM = headM->nextM;

        delete ptr;

        ptr = headM;

    }

    cursorM = NULL;

    sizeM = 0;

}

template <class T, class K>

void LookupTable<T, K>::copy(const LookupTable<T, K> &source){

    headM = 0;

    cursorM = 0;

    if (source.headM == 0)

        return;

    for (LT\_Node<T, K> \*p = source.headM; p != 0; p = p->nextM)

    {

        insert(Pair<T, K>(p->pairM.key, p->pairM.datum));

        if (source.cursorM == p)

            find(p->pairM.key);

    }

}

*// template <class T, class K>*

*// ostream &operator << <T, K>(ostream &os, const LookupTable<T, K> &lt){*

*//  if (lt.cursor\_ok())*

*//      os << lt.cursor\_key() << "  " << lt.cursor\_datum();*

*//  else*

*//      os << "Not Found.";*

*//  return os;*

*// }*

template <class T, class K>

const T &LookupTable<T, K>::Iterator::operator\*(){

    assert(LT->cursor\_ok());

    return LT->cursor\_datum();

}

template <class T, class K>

const T &LookupTable<T, K>::Iterator::operator++(){

    assert(LT->cursor\_ok());

    const T &x = LT->cursor\_datum();

    LT->step\_fwd();

    return x;

}

template <class T, class K>

const T &LookupTable<T, K>::Iterator::operator++(int){

    assert(LT->cursor\_ok());

    LT->step\_fwd();

    return LT->cursor\_datum();

}

template <class T, class K>

int LookupTable<T, K>::Iterator::operator!(){

    return (LT->cursor\_ok());

}

**Source code: {mainLab3ExB.h}**

*// ENSF 480 - Fall 2019 - Lab 3, Ex A*

*// M. Moussavi: Sept 26, 2019*

#include <assert.h>

#include <iostream>

#include "lookupTable.h"

#include "customer.h"

#include <cstring>

using namespace std;

template <class T, class K>

void print(LookupTable<T, K> &lt);

template <class T, class K>

void try\_to\_find(LookupTable<T, K> &lt, K key);

void test\_Customer();

void test\_String();

void  test\_integer();

*//Uncomment the following function calls when ready to test template class LookupTable*

*//void test\_String();*

*//void test\_integer();*

int main()

{

*//create and test a lookup table with an integer key value and Customer datum*

  test\_Customer();

*// Uncomment the following function calls when ready to test template class LookupTable*

*// create and test a a lookup table of type <int, String>*

  test\_String();

*// Uncomment the following function calls when ready to test template class LookupTable*

*// create and test a a lookup table of type <int, int>*

  test\_integer();

  cout << "\n\nProgram terminated successfully.\n\n";

  return 0;

}

template <class T, class K>

void print(LookupTable<T, K> &lt){

  if (lt.size() == 0)

    cout << "  Table is EMPTY.\n";

  for (lt.go\_to\_first(); lt.cursor\_ok(); lt.step\_fwd()){

    cout << lt << endl;

  }

}

template <class T, class K>

void try\_to\_find(LookupTable<T, K> &lt, K key)

{

  lt.find(key);

  if (lt.cursor\_ok())

    cout << "\nFound key:" << lt;

  else

    cout << "\nSorry, I couldn't find key: " << key << " in the table.\n";

}

void test\_Customer()

*//creating a lookup table for customer objects.*

{

  cout << "\nCreating and testing Customers Lookup Table <not template>-...\n";

  LookupTable <Customer, int>lt;

*// Insert using new keys.*

  Customer a("Joe", "Morrison", "11 St. Calgary.", "(403)-1111-123333");

  Customer b("Jack", "Lewis", "12 St. Calgary.", "(403)-1111-123334");

  Customer c("Tim", "Hardy", "13 St. Calgary.", "(403)-1111-123335");

  lt.insert(Pair<Customer, int>(8002, a));

  lt.insert(Pair<Customer, int>(8004, c));

  lt.insert(Pair<Customer, int>(8001, b));

  assert(lt.size() == 3);

  lt.remove(8004);

  assert(lt.size() == 2);

  cout << "\nPrinting table after inserting 3 new keys and 1 removal...\n";

  print(lt);

*// Pretend that a user is trying to look up customers info.*

  cout << "\nLet's look up some names ...\n";

  try\_to\_find(lt, 8001);

  try\_to\_find(lt, 8000);

*// test Iterator*

  cout << "\nTesing and using  iterator ...\n";

  LookupTable<Customer, int>::Iterator it = lt.begin();

  cout << "\nThe first node contains: " << \*it << endl;

  while (!it)

  {

    cout << ++it << endl;

  }

*//test copying*

  lt.go\_to\_first();

  lt.step\_fwd();

  LookupTable <Customer, int>clt(lt);

  assert(strcmp(clt.cursor\_datum().getFname(), "Joe") == 0);

  cout << "\nTest copying: keys should be 8001, and 8002\n";

  print(clt);

  lt.remove(8002);

*//Assignment operator check.*

  clt = lt;

  cout << "\nTest assignment operator: key should be 8001\n";

  print(clt);

*//Wipe out the entries in the table.*

  lt.make\_empty();

  cout << "\nPrinting table for the last time: Table should be empty...\n";

  print(lt);

  cout << "\*\*\*----Finished tests on Customers Lookup Table <not template>-----\*\*\*\n";

  cout << "PRESS RETURN TO CONTINUE.";

  cin.get();

}

*// Uncomment and modify the following funciton when ready to test LookupTable<int,Mystring>*

void test\_String()

*// creating lookuptable for Mystring objects*

  {

    cout<<"\nCreating and testing LookupTable <int, Mystring> .....\n";

    LookupTable <Mystring, int>lt;*// <Mystring, int>;*

*// Insert using new keys.*

    Mystring a("I am an ENEL-409 student.");

    Mystring b("C++ is a powerful language for engineers but it's not easy.");

    Mystring c ("Winter 2004");

    lt.insert(Pair<Mystring, int> (8002,a));

    lt.insert(Pair<Mystring, int> (8001,b));

    lt.insert(Pair<Mystring, int> (8004,c));

*//assert(lt.size() == 3);*

*//lt.remove(8004);*

*//assert(lt.size() == 2);*

    cout << "\nPrinting table after inserting 3 new keys and  and 1 removal...\n";

    print(lt);

*// Pretend that a user is trying to look up customers info.*

    cout << "\nLet's look up some names ...\n";

    try\_to\_find(lt, 8001);

    try\_to\_find(lt, 8000);

*// test Iterator*

    LookupTable<Mystring, int>::Iterator it = lt.begin();

    cout <<"\nThe first node contains: " <<\*it <<endl;

    while (!it) {

      cout <<++it << endl;

    }

*//test copying*

    lt.go\_to\_first();

    lt.step\_fwd();

    LookupTable  <Mystring, int>clt(lt);

    assert(strcmp(clt.cursor\_datum().c\_str(),"I am an ENEL-409 student.")==0);

    cout << "\nTest copying: keys should be 8001, and 8002\n";

    print(clt);

    lt.remove(8002);

*//Assignment operator check.*

    clt= lt;

    cout << "\nTest assignment operator: key should be 8001\n";

    print(clt);

*// Wipe out the entries in the table.*

    lt.make\_empty();

    cout << "\nPrinting table for the last time: Table should be empty ...\n";

    print(lt);

    cout << "\*\*\*----Finished Lab 4 tests on <int> <Mystring>-----\*\*\*\n";

    cout << "PRESS RETURN TO CONTINUE.";

    cin.get();

  }

*// Uncomment and modify the following funciton when ready to test LookupTable<int,int>*

void test\_integer()

*//creating look table of integers*

  {

    cout<<"\nCreating and testing LookupTable <int, int> .....\n";

    LookupTable <int, int> lt;

*// Insert using new keys.*

    lt.insert(Pair<int, int>(8002,9999));

    lt.insert(Pair<int, int>(8001,8888));

    lt.insert(Pair<int, int>(8004,8888));

    assert(lt.size() == 3);

    lt.remove(8004);

    assert(lt.size() == 2);

    cout << "\nPrinting table after inserting 3 new keys and  and 1 removal...\n";

    print(lt);

*// Pretend that a user is trying to look up customers info.*

    cout << "\nLet's look up some names ...\n";

    try\_to\_find(lt, 8001);

    try\_to\_find(lt, 8000);

*// test Iterator*

    LookupTable<int, int>::Iterator it = lt.begin();

    while (!it) {

      cout <<++it << endl;

    }

*//test copying*

    lt.go\_to\_first();

    lt.step\_fwd();

    LookupTable<int, int> clt(lt);

    assert(clt.cursor\_datum()== 9999);

    cout << "\nTest copying: keys should be 8001, and 8002\n";

    print(clt);

    lt.remove(8002);

*//Assignment operator check.*

    clt= lt;

    cout << "\nTest assignment operator: key should be 8001\n";

    print(clt);

*// Wipe out the entries in the table.*

    lt.make\_empty();

    cout << "\nPrinting table for the last time: Table should be empty ...\n";

    print(lt);

    cout << "\*\*\*----Finished Lab 4 tests on <int> <int>-----\*\*\*\n";

}