/\* Assignment: Hash Program

Author: Ryan Wood

Created On: April 14, 2018

Purpose: Read a data file filled with students and

their phone numbers and store them in a

Hash Table. Print the final table and

demonstrate looking up keys that are there

and keys that are not in the table. Demonstrate

the ability to look up a name and get their phone

number

\*/

/\*\* Hash Table Main file\*\*/

#include "HashTable.h"

#include <iostream>

#include <fstream>

#include "../LinkedList/Student.h"

using namespace std;

ifstream inFile("HashNamesAndPhone.txt");

ofstream outFile("HashOutput.out");

HashTable<Student, 51> table;

/\* Function: isNumber(char)

Purpose: determines whether the given character

falls within the ascii rang 48 and 57

which is numbers 0 through 9

Parameters: the character

Return: whether the character is a number

\*/

bool isNumber(char c)

{

return (c > 47 && c < 58);//0 - 9 ASCII

}

/\* Function: trim(string)

Purpose: trims the trailing space, tab, or

newline characters from the given string

Parameters: a reference to a string

\*/

void trim(string &str)

{

str.erase(str.find\_last\_not\_of(" \n\r\t")+1);

}

/\* Function: fillTable()

Purpose: reads the data from the file an creates a student for

each line it finds. Places the student into the table

\*/

void fillTable()

{

Student \*student = NULL;

int index = 0;

string name;

string phone;

while(getline(inFile, name))

{

//test for invalid line

if(name.length() < 2)

continue;

/\* get an index to the string at which we should divide

between the end of the name and the beginning of the phone number\*/

index = 0;

while(!isNumber(name[index]) && index < name.length())

{

index++;

}

//split the line into phone and name

phone = name.substr(index, name.length());

name = name.substr(0, index);

//trim any trailing space/newlines

trim(name);

trim(phone);

//create a student. This function sets the key variable too

student = new Student(name, phone);

table.insert(student);

}

}

int main()

{

if(!inFile)

{

cout << "No data file found!" << endl;

outFile << "No data file found!" << endl;

return -1;

}

//fill the table with data

fillTable();

//print the table

cout << table << endl;

outFile << table << endl;

//print some stats

cout << "table size = " << table.size() << endl;

outFile << "table size = " << table.size() << endl;

cout << "num data holes = " << table.getNumDataHoles() << endl;

outFile << "num data holes = " << table.getNumDataHoles() << endl;

cout << "num collisions = " << table.getNumCollisions() << endl;

outFile << "num collisions = " << table.getNumCollisions() << endl;

//find a student

Student \*mighty = NULL;

if(table.isPresent("Mighty Mouse"))

{

cout << "Mighty Mouse is in the list" << endl;

outFile << "Mighty Mouse is in the list" << endl;

mighty = table.getElement("Mighty Mouse");

if(NULL == mighty)

{

cout << "Failed to get item Mighty Mouse" << endl;

outFile << "Failed to get item Mighty Mouse" << endl;

}

else

{

cout << "His phone number is " << mighty->phoneNumber << endl;

outFile << "His phone number is " << mighty->phoneNumber << endl;

}

//remove a student

cout << "removing Mighty Mouse" << endl;

outFile << "removing Mighty Mouse" << endl;

table.remove("Mighty Mouse");

if(table.isPresent("Mighty Mouse"))

{

cout << "failed to remove Mighty Mouse" << endl;

outFile << "failed to remove Mighty Mouse" << endl;

}

else

{

cout << "succesfully removed Mighty Mouse" << endl;

outFile << "succesfully removed Mighty Mouse" << endl;

}

}

//reprint size after removal

cout << "Now table size is " << table.size() << endl;

outFile << "Now table size is " << table.size() << endl;

//search for student not in the table

if(table.isPresent("Heather Wood"))

{

cout << "Somenow my wife ended up in this table." << endl;

outFile << "Somenow my wife ended up in this table." << endl;

}

else

{

cout << "seems the lookup function works right." << endl;

outFile << "seems the lookup function works right." << endl;

}

outFile.close();

return 0;

}

/\* Class: HashTable

Author: Ryan Wood

Purpose: demonstrate an understanding of hash tables and how

they work to make data storage faster and simpler.

In order to use this Hash Table, the datatype must

include a string member named key and a member named nodeLink,

which is a pointer to another item of the same data type

Created On: April 14, 2018

\*/

/\*Hash Table definition\*/

#ifndef HASH\_TABLE\_H

#define HASH\_TABLE\_H

#include "../List/List.h"

#include "../LinkedList/LinkedList.h"

template<typename type, size\_t CAPACITY>

class HashTable

{

typedef type\* element;

public:

HashTable();

/\* Function: insert(element)

Purpose: inserts the given element into the table

Parameters: an item that includes a public member named key

\*/

void insert(element);

/\* Function: remove(element)

Purpose: removes an element from the table that has the

given key.

Parameters: a key that may be in the table

\*/

void remove(string key);

/\* Function: getElement(string) const

Purpose: retrieves the element from the list having the

given key

Parameters: the key

Return: an element from the table or a NULL element

\*/

type\* getElement(string key) const;

/\* Function: isPresent(string) const

Purpose: determines whether there is an element in

the table having the given key

Parameters: a key that may be in the list

Return: whether the key was found

\*/

bool isPresent(string key) const;

/\* Function: size() const

Purpose: determines the number of elements in the list

by going through each element of the list and

retrieving the size of the element's Linked List

and adding those totals together

Return: the number of items in the list

\*/

size\_t size() const;

/\* Function: print(ostream&) const

Purpose: This function prints the table using the given

output stream, printing each element on its own

line and printing "Data Hole" for those parts of

the table that have no items stored

Parameters: the output stream

Return: the output stream

\*/

std::ostream &print(ostream&) const;

/\* Function: print(ofstream& const

Purpose: prints the list to the given file output

stream, printing each element on its own line

and printing "Data hole" for those parts of the

table that have no items stored

Parameters: the file stream

Return: the file stream

\*/

std::ofstream &print(ofstream&) const;

/\* Function: print() const

Purpose: prints the list to the console,

printing each element on its own line

and printing "data hole" for those parts

of the list that have nothing stored

\*/

void print() const;

/\* Function: getNumCollisions() const

Purpose: retrieves the number of LinkedList elements in

the table that have more than one element stored

in them and determines how many elements are stored

in them, accumulating a total number of element that

ended up with the same hash index

Return: the number of elements that ended up with the same hash

\*/

int getNumCollisions() const;

/\* Function: getNumDataHoles() const

Purpose: determines the number of LinkedList elements in the

table have no items stored

Return: the number of holes in the table

\*/

int getNumDataHoles() const;

private:

/\* Function: hash(string) const

Purpose: creates an index from the data in the given

key, that is likely to be unique, but will

generate some keys that are non-unique

Parameters: the key

Return: the index to the table

\*/

int hash(std::string) const;

static const int numListVals = 10;

List<LinkedList<type, numListVals>\*, CAPACITY> \*list;//the table

int numElements;//number of data values

};

#include "HashTable.template"

#endif

/\* Class: HashTable

Author: Ryan Wood

Purpose: demonstrate an understanding of hash tables and how

they work to make data storage faster and simpler.

In order to use this Hash Table, the datatype must

include a string member named key and a member named nodeLink,

which is a pointer to another item of the same data type

Created On: April 14, 2018

\*/

/\*Hash Table implementation\*/

using namespace std;

template<typename type, size\_t CAPACITY>

HashTable<type, CAPACITY>::HashTable()

{

int index = 0;

list = new List<LinkedList<type, numListVals>\*, CAPACITY>;

numElements = 0;

for(index = 0; index < CAPACITY; index++)

{

LinkedList<type, numListVals> \*linky = new LinkedList<type, numListVals>;

list->insertAfter(linky);

list->next();

}

}

template<typename type, size\_t CAPACITY>

void HashTable<type, CAPACITY>::insert(element item)

{

int index = 0;

string key;

key = item->key;

index = hash(key);

if(!(list->getElement(index)->isFull()))

{

list->getElement(index)->insert(item);

numElements++;

}

else

{

cout << "Unable to add the element " << \*item

<< "to the Hash Table" << endl;

}

}

template<typename type, size\_t CAPACITY>

type\* HashTable<type, CAPACITY>::getElement(string key) const

{

element item = NULL;

int index = 0;

index = hash(key);

item = list->getElement(index)->find(key);

return item;

}

template<typename type, size\_t CAPACITY>

void HashTable<type, CAPACITY>::remove(string key)

{

int index = 0;

element item = NULL;

index = hash(key);

item = list->getElement(index)->find(key);

if(NULL != item)

{

list->getElement(index)->remove(item);

numElements--;

}

}

template<typename type, size\_t CAPACITY>

bool HashTable<type, CAPACITY>::isPresent(string key) const

{

int index = 0;

element item = NULL;

index = hash(key);

item = list->getElement(index)->find(key);

return(NULL != item);

}

template<typename type, size\_t CAPACITY>

size\_t HashTable<type, CAPACITY>::size() const

{

return numElements;

}

template<typename type, size\_t CAPACITY>

int HashTable<type, CAPACITY>::hash(string key) const

{

int sumKey = 0;

int keySize = 0;

int iHash = 0;

int count = 0;

keySize = key.length();

for(count = 0; count < keySize; count++)

{

sumKey = sumKey \*7 + key[count];

}

sumKey\*= count;

iHash = sumKey % CAPACITY;

return iHash;

}

template<typename type, size\_t CAPACITY>

int HashTable<type, CAPACITY>::getNumCollisions() const

{

int count = 0;

list->first();

while(list->getPos() < list->getCapacity()-1)

{

if(list->getElement()->size() > 1)

count+= list->getElement()->size()-1;

list->next();

}

if(list->getElement()->size() > 1)

count+= list->getElement()->size()-1;

return count;

}

template<typename type, size\_t CAPACITY>

int HashTable<type, CAPACITY>::getNumDataHoles() const

{

int count = 0;

list->first();

while(list->getPos() < list->getCapacity()-1)

{

if(list->getElement()->isEmpty())

count++;

list->next();

}

if(list->getElement()->isEmpty())

count++;

return count;

}

template<typename type, size\_t CAPACITY>

void HashTable<type, CAPACITY>::print() const

{

list->first();

while(list->getPos() < list->getCapacity()-1)

{

if(!list->getElement()->isEmpty())

cout << \*list->getElement();

else

cout << "Data Hole" << endl;

list->next();

}

if(!list->getElement()->isEmpty())

cout << \*list->getElement();

else

cout << "Data Hole" << endl;

}

template<typename type, size\_t CAPACITY>

std::ostream &HashTable<type, CAPACITY>::print(std::ostream &out) const

{

list->first();

while(list->getPos() < list->getCapacity() -1)

{

out.flush();

if(!list->getElement()->isEmpty())

out << \*(list->getElement());

else

out << "Data Hole" << endl;

list->next();

}

return out;

}

template<typename type, size\_t CAPACITY>

std::ofstream &HashTable<type, CAPACITY>::print(std::ofstream &out) const

{

list->first();

while(list->getPos() < list->getCapacity() -1)

{

out.flush();

if(!list->getElement()->isEmpty())

out << \*(list->getElement());

else

out << "Data Hole" << endl;

list->next();

}

return out;

}

template<typename type, size\_t CAPACITY>

std::ostream &operator<<(std::ostream &out, const HashTable<type, CAPACITY> &table)

{

return table.print(out);

}

template<typename type, size\_t CAPACITY>

std::ofstream &operator<<(std::ofstream &out, const HashTable<type, CAPACITY> &table)

{

return table.print(out);

}

/\* Assignment: Hash Table

Author: Ryan Wood

Created On: April 14, 2018

\*/

/\*Student\*/

#ifndef STUDENT\_H

#define STUDENT\_H

#include <iostream>

#include <fstream>

/\*the Student structure for our Linked List\*/

class Student

{

/\* What I really want is a stuct, so I'm making members public,

but I also want to easilly associate the name with the key,

so I wrote the constructor function, I want to be able to

easily compare so I wrote the comparison functions,

I want to be able to easily copy a student so I

overloaded the equal operator, and I want to be able to

easilly output to a file or the console so I overloaded the

insertion operator

\*/

public:

std::string name;

std::string key;

std::string phoneNumber;

Student \*nodeLink;//Link to order list by name

/\* Function: Student(string, string)

Purpose: creates a student having the given name

and phone number, sets the key variable

to the value of name, and sets the nodeLink NULL

Parameters: the name, the phone number

Return: a student instance

\*/

Student(std::string strName, std::string strPhone)

{

name = strName;

key = strName;

phoneNumber = strPhone;

nodeLink = NULL;

}

/\* Function: Student()

Purpose: sets the nodeLink variable NULL

Return: an empty student

\*/

Student()

{

nodeLink = NULL;

}

/\* Function: operator =(Student)

Purpose: sets this student's member variables to be

the values of the given student's

member variables (copy)

Parameters: the other student

\*/

void operator = (const Student &stud)

{

nodeLink = stud.nodeLink;

key = stud.key;

name = stud.name;

phoneNumber = stud.phoneNumber;

}

/\* Function: operator >(const Student&) const

Purpose: determines whether the given student has a

name variable value that is less than this

student's name variable value

Parameters: the other student

Return: whether the other student is less

\*/

bool operator >(const Student &stud) const

{

return name > stud.name;

}

/\* Function: operator <(const Student &) const

Purpose: determines whether the given student has a

name variable value that is greater then this

student's name variable value

Parameters: the other student

Return: whether the other student is greater

\*/

bool operator <(const Student &stud) const

{

return name < stud.name;

}

/\* Function: operator ==(const Student &) const

Purpose: determines whether the given student has

the same name variable value as this student

Parameters: the student

Return: whether the students are equal

\*/

bool operator ==(const Student &stud) const

{

return name == stud.name;

}

/\* Function: operator <<(ofstream&, const Student)

Purpose: outputs the given student using the given file output stream

printing the name followed by a colon, followed by the

student phone number to the file

Parameters: the file stream, the student

Return: the file stream

\*/

friend std::ofstream& operator <<(std::ofstream& out, const Student &student)

{

out.flush();

out << student.name << " : " << student.phoneNumber;

return out;

}

/\* Function: operator <<(ostream&, const Student &)

Purpose: outputs the given student using the given output stream,

printing the name followed by a colon, followed by the

student phone number

Parameters: the output stream, the student

Return: the output stream

\*/

friend std::ostream& operator <<(std::ostream& out, const Student &student)

{

out.flush();

out << student.name << " : " << student.phoneNumber;

return out;

}

};

#endif

/\* Assignment: General List Program

Author: Ryan Wood

Created On: 1/28/2018

Last Modified: 4/15/2018

\*/

/\*List definition file\*/

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

#include <fstream>

#include <string>

template <typename et, size\_t CAPACITY>

class List

{

public:

/\* Function: List()

Purpose: default constructor. Creates a List with a capacity of 20

Return: an empty List

\*/

List();

/\* Function: List()

Purpose: copy constructor

Paramters: the List to copy

Return: a new List with the same values as

the one given

\*/

List(const List &lstA);

/\* Function: emtpy()

Purpose: determines whether the list has no values

Return: whether elements are used

\*/

bool empty() const;

/\* Function: full()

Purpose: determines whether the list is unable

to take on any new values

Return: whether used is equal to CAPACITY

\*/

bool full() const;

/\* Function: first()

Purpose: sets the position to the first element of the array, 0

\*/

void first();

/\* Function: last()

Purpose: sets the position of the array to last element

in the array, used-1

\*/

void last();

/\* Function: prev()

Purpose: sets the position back by one index

\*/

void prev();

/\* Function: next()

Purpose: sets the postition forward by one index

\*/

void next();

/\* Function: getPosition() const

Purpose: retrieves the current position index

Return: the index

\*/

int getPos() const;

/\* Funcrtion: setPos(int)

Purpose: sets the current position to the given

index if it is valid

Parameters: an index less than the size of the List

\*/

void setPos(int);

/\* Function: insertBefore(et)

Purpose: inserts the given value before the current

position in the list

Parameters: the value

\*/

void insertBefore(et);

/\* Function: insertAfter(et)

Purpose: inserts the given value after the

current position in the List

Parameters: the value

\*/

void insertAfter(et);

/\* Function: getElement() const

Purpose: retrieves the value at the current position

in the list

Return: the value at the current index

\*/

et getElement() const;

/\* Function: getElement(int)

Purpose: retrieves the value at the given index to the List

if it is valid for the list

Parameters: the index

Return: the value

\*/

et getElement(int pos) const;

/\* Function: size()

Purpose: gets the size of the list, the number of

elements that have been used

Return: the number of elements used

\*/

size\_t size() const;

/\* Function: replace(et)

Purpose: replaces the value at the current index

to the list with the value given

Parameters: the value

\*/

void replace(et);

/\* Function: erase()

Purpose: erases the element at the current index to

the list, shifting all other elements down

to take the empty place

\*/

void erase();

/\* Function: clear()

Purpose: deletes all elements in the list, setting

pos and used both to 0

\*/

void clear();

/\* Function: swap(int, int)

Purpose: swaps the element at the given first index

with the element at the given second index

if neither index is greater than the current

number of values used

Parameters: the first index, the second index

\*/

void swap(int indexA, int indexB);

/\* Function: max(int, int)

Purpose: determines which of the two given indexes

to the list contains the largest value,

returning the index with the largest value.

If one index has a null value, returns the

other. If they are both null returns -1

Parameters: the first index, the second index

Return: the index holding the larger value

\*/

int max(int indexA, int indexB);

/\* Function: getCapacity()

Purpose: returns the maximum number of elements the

list is able to hold

Return: capacity

\*/

size\_t getCapacity() const;

private:

/\* Function: copy(List&)

Purpose: copies the given list to the current list,

element by element, copying the values of

the array and the value of pos and used

Parameters: a valid list

\*/

void copy(const List &lstA);

et \*arry;//the list itself

int used;//how many elements have been used

int pos;//the current pointer to the list

};//List class

template <typename et, size\_t CAPACITY>

inline List<et, CAPACITY>::List()

{

used = 0;

pos = 0;

arry = new et[CAPACITY];

}

template <typename et, size\_t CAPACITY>

inline List<et, CAPACITY>::List(const List &lstA)

{

copy(lstA);

}

/\* Function: operator<<(ostream&, List&)

Purpose: overloads the stream insertion operator using the given

output to the console

Parameters: the console output stream, the List to print

Return: a reference to the output stream

\*/

template <typename et, size\_t CAPACITY>

std::ostream& operator <<(std::ostream &out, const List<et, CAPACITY> &lst);

/\* Function: operator <<(ofstream&, List&)

Purpose: overloads the given file output stream and prints the

given list to the file

Parameters: a reference to the file output stream,

a reference to the LIst to print

Return: a reference to the file output stream

\*/

template <typename et, size\_t CAPACITY>

std::ofstream& operator <<(std::ofstream &out, const List<et, CAPACITY> &lst);

/\* Function: operator==(List&, List&)

Purpose: determines whether the given lists are equal

Parameters: the first list, the second list

Return: whether the lists have the same values

\*/

template <typename et, size\_t CAPACITY>

bool operator ==(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB);

/\* Function: operator !=(List&, List&)

Purpose: determines whether the given lists are not equal

Parameters: the first list, the second list

Return: whether the two lists do not have the same values

\*/

template <typename et, size\_t CAPACITY>

bool operator !=(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB);

/\* Function: operator +(List&, List&)

Purpose: adds the values of the second list into the remaining values

of the first list, creating a new list with the results

Parameters: the first list, the second List

Return: a list with values of both lists

\*/

template <typename et, size\_t CAPACITY>

List<et, CAPACITY> operator +(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB);

template<typename et, size\_t CAPACITY>

inline bool List<et, CAPACITY>::empty() const

{

return(used == 0);

}

template<typename et, size\_t CAPACITY>

inline bool List<et, CAPACITY>::full() const

{

return(used == CAPACITY);

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::first()

{

pos = 0;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::last()

{

pos = used-1;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::prev()

{

if(pos > 0)

pos--;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::next()

{

if(pos < CAPACITY-1 && pos < used-1)

pos++;

}

template<typename et, size\_t CAPACITY>

inline int List<et, CAPACITY>::getPos() const

{

return pos;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::setPos(int val)

{

if(val < used && val > -1)

pos = val;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::insertBefore(et val)

{

et \*current;

et \*tmp;

et \*next;

//case used = 0

if(used < CAPACITY)

{

if(used == 0)

{

arry[pos] = val;

}

else

{

//set current to position address

current = &arry[pos];

//set a temp var to the next to last value

tmp = &arry[used-1];

next = tmp+1;

//loop backward up to our current position

while(tmp >= current)

{

//set the next value to be the value before it

\*next = \*tmp;

tmp--;

next--;

}

//now set the current value to the given value

\*current = val;

}

//increment how many values there are

used++;

}

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::insertAfter(et val)

{

et \*current;

et \*tmp;

if(used < CAPACITY)

{

//case used = 0;

if(used == 0)

arry[pos] = val;

else

{

current = &arry[pos];

tmp = &arry[used-1];

//from the last item to the current item, loop

//backward through array, moving elements forward by one

while(tmp >= current)

{

\*(tmp+1) = \*tmp;

tmp--;

}

current++;

\*current = val;

pos++;

}

used++;

}

}

template<typename et, size\_t CAPACITY>

inline et List<et, CAPACITY>::getElement(int ePos) const

{

et val;

if(pos >= 0 && pos < used)

{

val = arry[ePos];

}

return val;

}

template<typename et, size\_t CAPACITY>

inline et List<et, CAPACITY>::getElement() const

{

et val;

if(used > 0)

{

val = arry[pos];

}

return val;

}

template<typename et, size\_t CAPACITY>

inline size\_t List<et, CAPACITY>::size() const

{

return used;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::replace(et val)

{

arry[pos] = val;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::erase()

{

int count = 0;

if(used > 0)

{

count = pos;

while(count < used)

{

arry[count] = arry[count+1];

count++;

}

used--;

pos--;

}

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::swap(int indexA, int indexB)

{

et tmp;

if(indexA <= used && indexB <= used)

{

tmp = arry[indexA];

arry[indexA] = arry[indexB];

arry[indexB] = tmp;

}

}

template<typename et, size\_t CAPACITY>

inline int List<et, CAPACITY>::max(int indexA, int indexB)

{

int maxIndex = -1;

et valA;

et valB;

if(indexA >= used || indexB >= used)

{

if(indexA >= used && indexB >= used)

maxIndex = -1;

else if(indexA >= used)

maxIndex = indexB;

else

maxIndex = indexA;

}

else

{

valA = arry[indexA];

valB = arry[indexB];

if(valA > valB)

maxIndex = indexA;

else

maxIndex = indexB;

}

return maxIndex;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::clear()

{

int count = 0;

et null;

for(count = 0; count < used; count++)

{

arry[count] = null;

}

used = 0;

pos = 1;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::copy(const List &lstA)

{

int count = 0;

et \*iter = NULL;

CAPACITY = lstA.getCapacity();

arry = new et[CAPACITY];

used = 0;

pos = 0;

if(!lstA.empty())

{

iter = arry;

for(count = 0; count < lstA.size(); count++)

{

\*iter = lstA.getElement(count);

iter++;

}

}

pos = lstA.getPos();

used = lstA.size();

}

template<typename et, size\_t CAPACITY>

inline size\_t List<et, CAPACITY>::getCapacity() const

{

return CAPACITY;

}

template<typename et, size\_t CAPACITY>

inline std::ostream& operator <<(std::ostream &out, const List<et, CAPACITY> &lst)

{

int count = 0;

for(count = 0; count < lst.size(); count++)

{

if(count == 0)

out << lst.getElement(count);

else

out << ", " << lst.getElement(count);

}

return out;

}

template<typename et, size\_t CAPACITY>

inline std::ofstream& operator <<(std::ofstream &out, const List<et, CAPACITY> &lst)

{

int count = 0;

for(count = 0; count < lst.size(); count++)

{

out << lst.getElement(count) << " ";

}

return out;

}

template<typename et, size\_t CAPACITY>

inline bool operator ==(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB)

{

int count = 0;

if(lstA.size() != lstB.size())

return false;

for(count = 0; count < lstA.size(); count++)

{

if(lstA.getElement(count) != lstB.getElement(count))

return false;

}

return true;

}

template<typename et, size\_t CAPACITY>

inline bool operator !=(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB)

{

return(!(lstA == lstB));

}

template<typename et, size\_t CAPACITY>

inline List<et, CAPACITY> operator +(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB)

{

int count = 0;

List<et, CAPACITY> lstC;

for(count = 0; count < lstA.size(); count++)

{

lstC.insertAfter(lstA.getElement(count));

}

for(count = 0; count < lstB.size(); count++)

{

lstC.insertAfter(lstA.getElement(count));

}

return lstC;

}

#endif

/\* Class: LinkedList

Author: Ryan Wood

Created On: February 11, 2018

Last Modified: April 14, 2018

Purpose: demonstrate use of Linked List structure.

In order to utilize this data structure

the type of data stored in the LinkedList

must have a member named key and a member

named nodeLink, which is a pointer to another

item of the same datatype

\*/

/\*Linked List Definition file\*/

#ifndef LINKED\_LIST\_H

#define LINKED\_LIST\_H

#include<iostream>

template<typename type, std::size\_t CAPACITY>

class LinkedList

{

typedef type\* node;//make the pointer easier to deal with

public:

/\* Function: LinkedList()

Purpose: default constructor. sets head to null

and opens the output file

Return: an instance of an empty LinkedList

\*/

LinkedList();

/\* Function: ~LinkedList()

Purpose: default destructor. Closes the output file

\*/

~LinkedList();

/\* Function: size()

Purpose: retrieves the number of nodes in the list

Return: the number of elements

\*/

std::size\_t size() const;

/\* Function: insert(type)

Purpose: inserts a new node into the list

Parameters: the data

\*/

void insert(node);

/\* Function: remove(node)

Purpose: removes the given item from the LinkedList

if it is in the list, Linking what was

previously linked to it up to whatever it

is currently linked to and deleting the

item, freeing its allocated memory

Parameters: the item

\*/

void remove(node);

/\* Function: isFull() const

Purpose: determines whether the Linked List data

sturcture is full, unable to take more values

Return: whether the list is full

\*/

bool isFull() const;

/\* Function: isEmpty() const

Purpose: determines whether the Linked List has

no data stored in it

Return: whether the list is empty

\*/

bool isEmpty() const;

/\* Function: find(string)

Purpose: finds an item in the list that has a

value for a member named key that matches

the given key value

Paramters: the key

Return: the item or a null element

\*/

node find(std::string key) const;

/\* Function: print(ostream&) const

Purpose: prints the list using the given ostream handle,

printing the items with a newline after each entry

Paramters: the output stream

Return: the output stream

\*/

std::ostream& print(std::ostream&) const;

/\* Function: print(ofstream) const

Purpose: prints the list using the given file output stream

handle, printing each element one line at a time

Parameters: the file output stream

Return: the file output stream

\*/

std::ofstream& print(std::ofstream&) const;

/\* Function: print() const

Purpose: prints the list to the console, one item per line

\*/

void print() const;

private:

/\* Function: clear(node)

Purpose: deletes the given node and recursively

deletes the items linked to it via nodeLink.

head may be passed to the function, deleting

all data in the list

Parameters: the node

\*/

void clear(node);

node head;//pointer to the first data element

};

#include "LinkedList.template"

#endif

/\* Class: LinkedList

Author: Ryan Wood

Created On: February 11, 2018

Last Modified: April 14, 2018

Purpose: demonstrate use of Linked List structure

\*/

/\*Linked List implementation file\*/

#include <iostream>

#include <fstream>

#include <iomanip>

#include <string>

using namespace std;

template<typename type, size\_t CAPACITY>

LinkedList<type, CAPACITY>::LinkedList()

{

head = NULL;

}

template<typename type, size\_t CAPACITY>

void LinkedList<type, CAPACITY>::clear(node data)

{

if(NULL == data)

return;

clear(data->nodeLink);

delete(data);

}

template<typename type, size\_t CAPACITY>

LinkedList<type, CAPACITY>::~LinkedList()

{

clear(head);

}

template<typename type, size\_t CAPACITY>

size\_t LinkedList<type, CAPACITY>::size() const

{

int total = 0;

node nodePointer = NULL;

if(NULL == head)

return 0;

nodePointer = head->nodeLink;

while(NULL != nodePointer)

{

total++;

nodePointer = nodePointer->nodeLink;

}

return total;

}

template<typename type, size\_t CAPACITY>

bool LinkedList<type, CAPACITY>::isFull() const

{

return (size() == CAPACITY);

}

template<typename type, size\_t CAPACITY>

void LinkedList<type, CAPACITY>::insert(node data)

{

node newNode = NULL;

node nodePtr = NULL;

node prevNode = NULL;

if(NULL == head)

{

head = new type;

head->nodeLink = NULL;

}

//get the first element

nodePtr = head->nodeLink;

while(NULL != nodePtr && nodePtr > data)

{

prevNode = nodePtr;

nodePtr = nodePtr->nodeLink;

}

if(data == nodePtr)

return;

newNode = data;

newNode->nodeLink = NULL;

newNode->nodeLink = nodePtr;

if(NULL == prevNode)

head->nodeLink = newNode;

else

prevNode->nodeLink = newNode;

}

template<typename type, size\_t CAPACITY>

std::ofstream& LinkedList<type, CAPACITY>::print(std::ofstream& out) const

{

node nodePtr = NULL;

if(NULL == head)

return out;

nodePtr = head->nodeLink;

while(NULL != nodePtr)

{

out << \*nodePtr << endl;

nodePtr = nodePtr->nodeLink;

}

return out;

}

template<typename type, size\_t CAPACITY>

std::ostream& LinkedList<type, CAPACITY>::print(std::ostream& out) const

{

node nodePtr = NULL;

if(NULL == head)

return out;

nodePtr = head->nodeLink;

while(NULL != nodePtr)

{

out << \*nodePtr << endl;

nodePtr = nodePtr->nodeLink;

}

return out;

}

template<typename type, size\_t CAPACITY>

void LinkedList<type, CAPACITY>::print() const

{

node nodePtr = NULL;

if(NULL == head)

return;

nodePtr = head->nodeLink;

while(NULL != nodePtr)

{

cout << \*nodePtr << endl;

nodePtr = nodePtr->nodeLink;

}

}

template<typename type, size\_t CAPACITY>

void LinkedList<type, CAPACITY>::remove(node data)

{

node nodePtr = NULL;

node prev = NULL;

if(NULL == data)

return;

if(NULL == head)

return;

nodePtr = head;

while(NULL != nodePtr && !(nodePtr == data))

{

prev = nodePtr;

nodePtr = nodePtr->nodeLink;

}

if(NULL == prev)

head->nodeLink = nodePtr->nodeLink;

else

prev->nodeLink = nodePtr->nodeLink;

delete(nodePtr);

}

template<typename type, size\_t CAPACITY>

type\* LinkedList<type, CAPACITY>::find(string key) const

{

node item = NULL;

item = head;

while(NULL != item && (item->key != key))

{

item = item->nodeLink;

}

return item;

}

template<typename type, size\_t CAPACITY>

bool LinkedList<type, CAPACITY>::isEmpty() const

{

return (NULL == head);

}

template<typename type, size\_t CAPACITY>

std::ostream &operator <<(std::ostream &out, const LinkedList<type, CAPACITY> &list)

{

return list.print(out);

}

template<typename type, size\_t CAPACITY>

std::ofstream &operator <<(std::ofstream &out, const LinkedList<type, CAPACITY> &list)

{

return list.print(out);

}

TABLE OUTPUT CONVERTED TO MS WORD TABLE FROM NEWLINE DELINEATED LIST

|  |  |  |
| --- | --- | --- |
| Hoos R. Dadie : 818-3821 | Hofstra M. : 601-3225 | james wilis thomas : 261-8342 |
| Big Tow : 384-5624 | Data Hole | Data Hole |
| Znot Noz Oozey : 531-4353 | Currie W. D. : 701-4281 | Tau Jam : 532-6871 |
| Data Hole | Mister Rogers : 924-7221 | Morier G. E. : 544-2319 |
| Hewy Lewy Dewy : 623-9921 | Data Hole | Smelly Tau : 707-7281 |
| Mighty Mouse : 222-2222 | Booger Runs : 822-7724 | Trigger Phanger : 421-3435 |
| Data Hole | Data Hole | Data Hole |
| Data Hole | Man Oh Mann : 313-7422 | Data Hole |
| Data Hole | Chipoff E Oblock : 773-4152 | LessGo Eat : 233-1984 |
| SixOne Other : 843-2343 | Tobe Ore Knott : 613-2414 | Sid MacGriddy : 882-2424 |
| Data Hole | Cray Z. Cyco : 134-7242 | G P Morier : 832-4562 |
| Taylor marie : 939-1512 | Malioneyh P. J. : 287-4344 | Data Hole |
| Zevent Heaven : 834-2563 | Legg T. : 587-2839 | Data Hole |
| E Lec Shaun : 709-7424 | Antoney Wenner : 999-3255 | TooB OrNot : 283-5214 |
| Hasey Moto : 823-8000 | Hauser Yauser : 606-2940 | john marshall : 888-2891 |
| mack russell : 123-1234 | Twoseeor knocksee : 823-8321 | Oe vey : 177-1423 |
| Data Hole | Data Hole | Data Hole |
| Fanny Pac Jac : 842-3242 | stephens reynolds : 696-9231 | Data Hole |
| Data Hole | lea high lee : 266-8324 | Data Hole |
| Sticky Finger McRidder : 829-9853 | Juan Legged : 882-6246 | Tyson Chicken : 602-3152 |
| Hack N Zakkers : 231-4449 | Lewis Michelle Tee : 828-2148 | Stone Rock : 544-2372 |

table size = 45

num data holes = 19

num collisions = 13

Mighty Mouse is in the list

His phone number is 222-2222

removing Mighty Mouse

succesfully removed Mighty Mouse

Now table size is 44

seems the lookup function works right.