/\* Assignment: General List Program

Author: Ryan Wood

Created On: 1/28/2018

Last Modified: 2/24/2018

\*/

/\*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(et)

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: 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;

et null;

if(used > 0)

{

//delete current element

arry[pos] = null;

//left justify the rest of the array

count = pos;

while(count < used)

{

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

count++;

}

used--;

pos--;

}

}

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++)

{

if(count == 0)

{

out << lst.getElement(count);

}

else

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: Queue

Purpsoe: this container class takes elements using the push function

into a List that may be of any size the use wishes. The

elements are treated with a first in first out (FIFO)

order scheme. The pop function may be used to remove elements

and the top function may be used to get the element on

the top of the queue.

Author: Ryan Wood

Created On: February 17, 2018

Last Modified: February 24, 2018

\*/

/\*Queue definition file\*/

#ifndef QUEUE\_H

#define QUEUE\_H

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

#include <string>

template <typename et, size\_t CAPACITY>

class Queue

{

public:

/\* Function: Queue()

Purpose: default constructor. Creates an empty

Queue with a capacity of 100

Return: an instance of a Queue

\*/

Queue();

/\* Function: Queue()

Purpose: Destructor. Deletes the List, releasing the

RAM allocated for the Queue

\*/

~Queue();

/\* Function: push(et)

Purpose: pushes the given value into the back of the Queue

Parameters: the value

\*/

void push(et);

/\* Function: pop()

Purpose: removes the value that is at the top of the Queue

\*/

void pop();

/\* Function: top()

Purpose: retrieves the value that is at the top of the Que

Return: the top value of the Queue

\*/

et top() const;

/\* Function: print() const

Purpose: prints the values of the Queue in the

First in First Out order n which they would

be popped off.

\*/

void print() const;

/\* Function: size() const

Purpose: retrieves the number of elements that have been

used in the queue

Return: the size of the Queue

\*/

size\_t size() const;

/\* Function: empty() const

Purpose: determines whether the Queue has no used values

Return: whether size is zero

\*/

bool empty() const;

/\* Function: full() const

Purpose: determines whether the Queue can hold no more values

Return: whether size is equal to capacity

\*/

bool full() const;

private:

List<et, CAPACITY> \*qList;

};//Queue class

template <typename et, size\_t CAPACITY>

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

{

qList = new List<et, CAPACITY>;

}

template <typename et, size\_t CAPACITY>

inline Queue<et, CAPACITY>::~Queue()

{

delete(qList);

}

template <typename et, size\_t CAPACITY>

inline void Queue<et, CAPACITY>::push(et val)

{

if(!qList->full())

{

qList->insertBefore(val);

}

}

template <typename et, size\_t CAPACITY>

inline void Queue<et, CAPACITY>::pop()

{

if(!qList->empty())

{

qList->last();

qList->erase();

qList->first();

}

}

template <typename et, size\_t CAPACITY>

inline void Queue<et, CAPACITY>::print() const

{

int count = 0;

int vals = 0;

for(count = qList->size()-1; count >= 0; count--)

{

std::cout << qList->getElement(count) << " ";

vals++;

if(vals%10 == 0)

std::cout << std::endl;

}

}

template <typename et, size\_t CAPACITY>

inline et Queue<et, CAPACITY>::top() const

{

et val;

if(!qList->empty())

{

val = qList->getElement(qList->size()-1);

}

return val;

}

template <typename et, size\_t CAPACITY>

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

{

return qList->size();

}

template <typename et, size\_t CAPACITY>

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

{

return qList->empty();

}

template <typename et, size\_t CAPACITY>

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

{

return qList->full();

}

#endif

/\* Class: Stack

Purpose: this container class stores as many elements as the user

determines in the parameterized constructore in a Last in First Out

(FIFO) order scheme. Elements are added to the stack using push,

retrieved using top and removed using pop.

Author: Ryan Wood

Created On: February 17, 2018

Last Modified: February 24, 2018

\*/

/\*Stack definition file\*/

#ifndef STACK\_H

#define STACK\_H

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

#include <iostream>

template <typename et, size\_t CAPACITY>

class Stack

{

public:

/\* Function: stack()

Purpose: default constructor. Creates an instance of

an empty stack having a capacity of 100

Return: an instance of a stack

\*/

Stack();

/\* Function: ~Stack()

Purpose: destructor. Deletes the private List member variable,

freeing the memory allocated for the Stack

\*/

~Stack();

/\* Function: push(et)

Purpose: pushes the given element onto the top of the stack. The

last value given to this function will be the one retrieved

by the top function and deleted by the pop function

Parameters: the value

\*/

void push(et);

/\* Function: pop()

Purpose: removes the element of the Stack that is at the top

\*/

void pop();

/\* Function: top() const

Purpose: retrives the element of the stack that is on the top. This

is the last element that was placed in the stack using push

Return: the top element

\*/

et top() const;

/\* Function: print() const

Purpose: prints the stack in the order that the elements would be

popped off, last in first out

\*/

void print() const;

/\* Function: size() const

Purpose: determines the number of elements that are currently

in use, have a value in the stack

Return: the size

\*/

size\_t size() const;

/\* Function: empty() const

Purpose: determines whether the stack has no values in it

Return: whether size is zero

\*/

bool empty() const;

/\* Function: full() const

Purpose: determines whether the list can take no more elements

Return: whether size is equal to capacity

\*/

bool full() const;

private:

List<et, CAPACITY> \*stackList;

};//Stack class

template <typename et, size\_t CAPACITY>

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

{

stackList = new List<et, CAPACITY>;

}

template <typename et, size\_t CAPACITY>

inline Stack<et, CAPACITY>::~Stack()

{

delete(stackList);

}

template <typename et, size\_t CAPACITY>

inline void Stack<et, CAPACITY>::push(et val)

{

if(!stackList->full())

{

stackList->insertAfter(val);

}

}

template <typename et, size\_t CAPACITY>

inline void Stack<et, CAPACITY>::pop()

{

if(!stackList->empty())

{

stackList->erase();

}

}

template <typename et, size\_t CAPACITY>

inline void Stack<et, CAPACITY>::print() const

{

int count = 0;

for(count = 0; count < stackList->size(); count++)

{

std::cout << stackList->getElement() << " ";

if(count > 0 && count % 10 == 0)

std::cout << std::endl;

}

}

template <typename et, size\_t CAPACITY>

inline et Stack<et, CAPACITY>::top() const

{

et val;

if(!stackList->empty())

{

val = stackList->getElement();

}

return val;

}

template <typename et, size\_t CAPACITY>

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

{

return stackList->size();

}

template <typename et, size\_t CAPACITY>

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

{

return stackList->empty();

}

template <typename et, size\_t CAPACITY>

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

{

return stackList->full();

}

#endif

/\* Assignment: Queue and Stack Program

Author: Ryan Wood

Created On: February 18, 2018

Requirements: Decode the "CodedMsg.txt" file that was encoded using

the algorithm described in document QueueInterLeaveDecode.doc,

posted to Blackboard by Dr. Rimes on February 15, 2018.

\*/

/\*Stack, Queue, List Main file\*/

//include my containers

#include "../Containers/GenericContainers/List/List.h"

#include "../Containers/GenericContainers/Stack/Stack.h"

#include "../Containers/GenericContainers/Queue/Queue.h"

#include <string>

#include <iostream>

#include <fstream>

using namespace std;

ifstream inFile("CodedMsg.txt");

ofstream outFile("DecodeMsg.txt");

const size\_t CAPACITY = 200;

typedef string et;

Stack<et, CAPACITY> stack;

Queue<et, CAPACITY> que;

/\* Function: transferHalfQueueToStack()

Purpose: transfers the front half of the que to the stack

\*/

void transferHalfQueueToStack()

{

int half = 0;

int count = 0;

int qSize = 0;

et strVal = "";

qSize = que.size();

half = qSize/2;

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

{

strVal = que.top();

que.pop();

stack.push(strVal);

}

}

/\* Function: shiftHalfFrontQueueBack()

Purpose: shifts the front half of the queue

to the back

\*/

void shiftHalfFrontQueueBack()

{

string strVal = "";

int queSize = 0;

int half = 0;

int count = 0;

queSize = que.size();

half = queSize/2;

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

{

strVal = que.top();

que.pop();

que.push(strVal);

}

}

/\* Function: interLeave()

Purpose: this function loops through all elements of the

queue. Uses the stack as a temporary storage location

so it can flip the value of each successive element

in the queue.

\*/

void interLeave()

{

et strVal = "";

int count = 0;

int qSize = 0;

qSize = que.size();

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

{

strVal = que.top();

que.pop();

if(count %2 == 0)

{

stack.push(strVal);

}

else

{

que.push(strVal);

if(!stack.empty())

{

strVal = stack.top();

stack.pop();

que.push(strVal);

}

}

}

}

/\* Function: pushHalfAlternateQueueToStack()

Purpose: This function steps through half of the elements

of the queue and pushes every other element to the

stack. The elements it does not push to the stack,

it pushes to the beginning of the queue (itself).

In either case, pops the value off the top. Only

goes through half of the queue.

\*/

void pushHalfAlternatQueueToStack()

{

int queueSize = 0;

int count = 0;

int half = 0;

et strVal = "";

queueSize = que.size();

half = queueSize/2;

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

{

strVal = que.top();

if(count %2 == 0)

stack.push(strVal);

else

que.push(strVal);

que.pop();

}

}

/\* Function: pushAlternateQueueToStack()

Purpose: this function steps through all the elements

of the queue and pushes every other element

to the stack. The elements it does not push

to the stack, it pushes to the back of the

queue (itself). In both instances, pops the

value off the top, ensuring we get no duplicates

\*/

void pushAlternatQueueToStack()

{

int queuSize = 0;

int count = 0;

et strVal = "";

queuSize = que.size();

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

{

strVal = que.top();

if(count %2 == 0)

stack.push(strVal);

else

que.push(strVal);

que.pop();

}

}

/\* Function: transferStackToQueue()

Purpose: this function transfers the contents of the stack

beginning with top to the back of the queue

\*/

void transferStackToQueue()

{

et strVal = "";

int count = 0;

while(!stack.empty())

{

strVal = stack.top();

que.push(strVal);

stack.pop();

}

}

/\* Function: writeDecodedToFile()

Purpose: this function outputs the final queue values

to the file DecodedMsg.txt as the decoded message

\*/

void writeDecodedToFile()

{

et strVal = "";

while(!que.empty())

{

strVal = que.top();

que.pop();

outFile << strVal << " ";

}

}

/\* Function: loadFileToQueue()

Purpose: loads the contents of the CodedMsg.txt file

into the Queue container class

\*/

void loadFileToQueue()

{

et strVal;

while(inFile >> strVal)

{

if(!que.full())

{

que.push(strVal);

}

else

{

cout << "We're gonna need a bigger boat!" << endl;

return;

}

}

}

int main()

{

if(!inFile)

{

cout << "Could not load the input file." << endl;

return -1;

}

loadFileToQueue();

interLeave();

pushAlternatQueueToStack();

transferStackToQueue();

shiftHalfFrontQueueBack();

transferHalfQueueToStack();

transferStackToQueue();

shiftHalfFrontQueueBack();

que.print();

cout << endl;

cout << "Number of elements in Queue = " << que.size() << endl;

writeDecodedToFile();

inFile.close();

outFile.close();

return 0;

}

CODE MESSAGE

The CS input 372 will Program be Assignment "wordlist.txt" Queue which Program consists Interleave of the a first list half of of words the that queue have with been second encoded half as My described Encoding above. Algorithm Restrictions: Following You are must the use steps aggregate used Stack to and encode Queue a class. list To of solve words: this 1. problem, Push you the can first have half ONE elements Queue of and queue ONE to Stack stack. and 2. only Enqueue one back of the each. stack The elements. input 3. data Dequeue will the be first loaded half into elements a of Queue the to queue start and with enqueue and them from back. the 4. Queue Again start push your the algorithm first to half decode elements the into text. the (Decoding stack. as 5. you Interleave read the in elements your of data queue will and not stack. be Your acceptable.) problem loop is cin to >> decode str; the yourqueue.enqueue(str); encoded endloop data.

DECODED MESSAGE

CS 372 Program Assignment Queue Program Interleave the first half of the queue with second half My Encoding Algorithm Following are the steps used to encode a list of words: 1. Push the first half elements of queue to stack. 2. Enqueue back the stack elements. 3. Dequeue the first half elements of the queue and enqueue them back. 4. Again push the first half elements into the stack. 5. Interleave the elements of queue and stack. Your problem is to decode the encoded data. The input will be "wordlist.txt" which consists of a list of words that have been encoded as described above. Restrictions: You must use aggregate Stack and Queue class. To solve this problem, you can have ONE Queue and ONE Stack and only one of each. The input data will be loaded into a Queue to start with and from the Queue start your algorithm to decode the text. (Decoding as you read in your data will not be acceptable.) loop cin >> str; yourqueue.enqueue(str); endloop