**CSE 330 Lab 7 Report**

Daniel Meyer

Data Structures

Fall 2017

**Status:** 100%

**Time Complexity:** O(n)

**Storage Complexity:** O(n)

**Source Code:** Pages 2 - 8

**Sample Run:** Page 9

/\*

Daniel Meyer

11-6-17

CSE 330

Fall 2017

Lab 7: Deque

Problem: Implement a double-ended queue class that has the functionality of a STL deque

Algorithm: Using two vectors as each end of the deque, functions of a STL deque are

implemented such as an index operator[], erase function and insert function. A DequeIterator

is also implmented to traverse the Deque class and the two vectors.

Status: 100%

Time Complexity: O(n) Storage Complexity: O(n)

\*/

#ifndef DEQUE\_H

#define DEQUE\_H

#include <vector>

using namespace std;

template <class T> class DequeIterator;

//Deque class that has the functions of a double-ended queue of two vectors

template <class T>

class Deque

{

public:

typedef DequeIterator<T> iterator;

Deque() : vecOne(), vecTwo() { }

Deque(const unsigned int size, const T& initial) : vecOne(size / 2, initial), vecTwo(size - (size / 2), initial) { }

Deque(const Deque<T> & d) : vecOne(d.vecOne), vecTwo(d.vecTwo) { }

~Deque() { } // destructors for vecOne and vecTwo are automatically called

// never call a destructor explicitly

Deque & operator=(const Deque<T> & d);

T & operator[](unsigned int);

T & front();

T & back();

bool empty();

iterator begin();

iterator end();

void erase(const iterator &);

void erase(const iterator &, const iterator &); //delete everything between two iterators

void insert(const iterator &, const T &);

int size();

void push\_front(const T & value);

void push\_back(const T & value);

void pop\_front();

void pop\_back();

protected:

vector<T> vecOne;

vector<T> vecTwo;

};

//Overloaded equality operator for custom Deque class

template <class T>

Deque<T>& Deque<T>::operator=(const Deque<T> &d)

{

vecOne = d.vecOne;

vecTwo = d.vecTwo;

return \*this;

}

//Overloaded index operator for custom Deque class

template <class T>

T& Deque<T>::operator[](unsigned int index)

{

int n = vecOne.size();

if (index < n) {

return vecOne[(n - 1) - index];

}

else {

return vecTwo[index - n];

}

}

//Returns the back of first vector if its not empty, else front of second vector

template <class T>

T& Deque<T>::front()

{

if (!vecOne.empty()) {

return vecOne.back();

}

else {

return vecTwo.front();

}

}

//Returns the back of second vector if its not empty, else front of first vector

template <class T>

T& Deque<T>::back()

{

if (!vecTwo.empty()) {

return vecTwo.back();

}

else {

return vecOne.front();

}

}

//Returns true if both vectors are empty

template <class T>

bool Deque<T>::empty()

{

return (vecOne.empty() && vecTwo.empty());

}

//Returns an iterator pointing to the front of the Deque

template<class T>

DequeIterator<T> Deque<T>::begin()

{

return iterator(this, 0);

}

////Returns an iterator pointing to the back of the Deque

template <class T>

DequeIterator<T> Deque<T>::end()

{

return iterator(this, size());

}

//Removes value at index that iterator is pointing to

template <class T>

void Deque<T>::erase(const iterator &d)

{

int index = d.index;

int n = vecOne.size();

if (index < n) {

vecOne.erase(vecOne.begin() + (index - (n - 1)));

}

else {

vecTwo.erase(vecTwo.begin() + (index - n));

}

}

//Removes values from all index between start and end pointers

template <class T>

void Deque<T>::erase(const iterator &d1, const iterator &d2)

{

//Three Cases

//1 iterator is in each vector then -

// - split into two erase() if erasing from both vectors

//int start = d1.index;

//int end = d2.index;

//int n = vecOne.size();

/\*

if (start < n && end < n) { //If both pointers are in 1st vector

while (start != end) {

vecOne.erase(vecOne.begin() + (start - (n - 1)));

start++;

}

}

else if (start > n && end > n) { //If both pointers are in 2nd vector

while (start != end) {

vecTwo.erase(vecTwo.begin() + (start - n));

start++;

}

}

else {

int vecOne\_start = start;

int vecOne\_end = n - start;

int vecTwo\_start = vecTwo.begin();

int vecTwo\_end = end;

while (vecOne\_start != vecOne\_end) {

vecOne.erase(vecOne.begin() + (vecOne\_start - (n - 1)));

vecOne\_start++;

}

while (vecTwo\_start != vecTwo\_end) {

vecTwo.erase(vecTwo.begin() + (vecTwo\_start - n));

vecTwo\_start++;

}

}

\*/

//Would this recursive call work?

DequeIterator<T> i = d2;

while(i != d1) {

erase(i - 1);

i--;

}

}

//Inserts a value behind index the iterator is pointing to

template <class T>

void Deque<T>::insert(const iterator &d, const T &x)

{

int index = d.index;

int n = vecOne.size();

if (index < n) {

vecOne.insert(vecOne.begin() + (n - index), x);

}

else {

vecTwo.insert(vecTwo.begin() + (index - n), x);

}

}

//Returns the combined size of both vectors

template <class T>

int Deque<T>::size()

{

return (vecOne.size() + vecTwo.size());

}

//Pushes a value to the back of the first vector

template <class T>

void Deque<T>::push\_front(const T &value)

{

vecOne.push\_back(value);

}

//Pushes a value to the back of the second vector

template <class T>

void Deque<T>::push\_back(const T &value)

{

vecTwo.push\_back(value);

}

//Pops value from back of first vector if it's not empty, else erases begining of second vector

template <class T>

void Deque<T>::pop\_front()

{

if (!vecOne.empty()) {

vecOne.pop\_back();

}

else {

vecTwo.erase(vecTwo.begin());

}

}

//Pops value from back of first vector if it's not empty, else erases begining of first vector

template <class T>

void Deque<T>::pop\_back()

{

if (!vecTwo.empty()) {

vecTwo.pop\_back();

}

else {

vecOne.erase(vecOne.begin());

}

}

//Custom iterator used for traversal of the Deque class

template <class T> class DequeIterator

{

friend class Deque<T>;

typedef DequeIterator<T> iterator;

public:

DequeIterator() : theDeque(0), index(0) { }

DequeIterator(Deque<T> \* d, int i) : theDeque(d), index(i) { }

DequeIterator(const iterator & d) : theDeque(d.theDeque), index(d.index) { }

T & operator\*();

iterator & operator++();

iterator operator++(int);

iterator & operator--();

iterator operator--(int);

bool operator==(const iterator & r);

bool operator!=(const iterator & r);

bool operator<(const iterator & r);

T & operator[](unsigned int i);

iterator operator=(const iterator & r);

iterator operator+(int i);

iterator operator-(int i);

protected:

Deque<T> \* theDeque;

int index;

};

//Overloaded dereference operator for custom Deque iterator

template <class T>

T& DequeIterator<T>::operator\*()

{

return (\*theDeque)[index];

}

//Overloaded pre-increment for custom Deque iterator

template <class T>

DequeIterator<T>& DequeIterator<T>::operator++()

{

++index;

return \*this;

}

//Overloaded post-increment for custom Deque iterator

template <class T>

DequeIterator<T> DequeIterator<T>::operator++(int)

{

DequeIterator<T> temp(theDeque, index);

index++;

return temp;

}

//Overloaded pre-decrement for custom Deque iterator

template <class T>

DequeIterator<T>& DequeIterator<T>::operator--()

{

--index;

return \*this;

}

//Overloaded post-decrement for custom Deque iterator

template <class T>

DequeIterator<T> DequeIterator<T>::operator--(int)

{

DequeIterator<T> temp(theDeque, index);

index--;

return temp;

}

//Overloaded is equal to operator for custom Deque iterator

template <class T>

bool DequeIterator<T>::operator==(const iterator &r)

{

return (theDeque == r.theDeque && index == r.index);

}

//Overloaded not equal to operator for custom Deque iterator

template <class T>

bool DequeIterator<T>::operator!=(const iterator &r)

{

return (theDeque != r.theDeque || index != r.index);

}

//Overloaded less than operator for custom Deque iterator

template <class T>

bool DequeIterator<T>::operator<(const iterator &r)

{

return (theDeque == r.theDeque && index < r.index);

}

//Overloaded index operator for custom Deque iterator

template <class T>

T& DequeIterator<T>::operator[](unsigned int i)

{

return (\*theDeque)[index + i];

}

//Overloaded equality operator for custom Deque iterator

template <class T>

DequeIterator<T> DequeIterator<T>::operator=(const iterator &r)

{

theDeque = r.theDeque;

index = r.index;

return (iterator(theDeque, index));

}

//Overloaded addition operator for custom Deque iterator

template <class T>

DequeIterator<T> DequeIterator<T>::operator+(int i)

{

return (iterator(theDeque, index + i));

}

//Overloaded subtraction operator for custom Deque iterator

template <class T>

DequeIterator<T> DequeIterator<T>::operator-(int i)

{

return (iterator(theDeque, index - i));

}

#endif

**Sample Run**

Script started on Sat 11 Nov 2017 10:37:24 PM UTC

To run a command as administrator (user "root"), use "sudo <command>".

See "man sudo\_root" for details.

]0;ubuntu@ubuntu: ~[01;32mubuntu@ubuntu[00m:[01;34m~[00m$ cd Desktop

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ g++ Deque\_test.cpp Deque.h

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ ./a.out

3 2 1 0

0 1 2 3

SUCCESS

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ exit

Script done on Sat 11 Nov 2017 10:37:45 PM UTC