**CSE 330 Homework 4 Report**

Daniel Meyer

Data Structures

Fall 2017

**Status:** 100%

**Time Complexity:** O(n)

**Storage Complexity:** O(n)

**Source Code:** test.cpp: Page 2

List.h**:** Pages 3 - 12

**Sample Run:** Page 13

// test.cpp - a simple test program for List.h with reverse\_iterator

#include <iostream>

#include "List.h"

using namespace std;

int main()

{

List<int> l;

l.push\_back(44); // list = 44

l.push\_back(33); // list = 44, 33

l.push\_back(11); // list = 44, 33, 11

l.push\_back(22); // list = 44, 33, 11, 22

List<int> m(l);

cout << "List in order" << endl;

for (int i = 0; i < m.size(); i++) {

cout << m[i] << endl;

}

m.reverse(); // list = 22, 11, 33, 44

cout << "List in order after being reversed" << endl;

List<int>::iterator itr(m.begin());

while (itr != m.end()) {

cout << \*itr << endl;

itr++;

}

cout << "List in reverse order" << endl;

List<int>::reverse\_iterator rItr(m.rbegin());

while (rItr != m.rend()) {

cout << \*rItr << endl;

rItr++;

}

return 0;

}

/\*

Daniel Meyer

10-31-17

CSE 330

Fall 2017

Homework 4: List w/ Reverse

Problem: Add reverse and accessor functionality to List class

Algorithm: Added reverse\_iterator functionality with necesarry functions such a

rbegin() and rend() as well as the ability to use the operator[] and a function that

reverses the values of the List.

Status: 100%

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

\*/

#ifndef LIST\_H

#define LIST\_H

#include <algorithm>

#include <iostream>

using namespace std;

template <class T> class Link;

template <class T> class List\_iterator;

template <class T> class List\_iterator\_reverse;

//List class that contains each Link with a size value and pointer to first and last Link

template <class T>

class List

{

protected:

Link<T> \*first\_link;

Link<T> \*last\_link;

unsigned int my\_size;

public:

typedef List\_iterator<T> iterator;

typedef List\_iterator\_reverse<T> reverse\_iterator;

List();

List(const List<T> &l);

~List();

void free();

bool empty() const;

unsigned int size() const;

T& back() const;

T& front() const;

void push\_front(const T &x);

void push\_back(const T &x);

void pop\_front();

void pop\_back();

iterator begin() const;

iterator end() const;

void insert(iterator pos, const T &x); //insert x before pos

void erase(iterator &pos); //pos must be valid after erase() returns

List<T> operator=(const List<T> &l);

T& operator[](unsigned int);

reverse\_iterator rbegin();

reverse\_iterator rend();

void reverse();

};

//Default Constructor that sets first/last link and size to 0.

template <class T>

List<T>::List()

{

first\_link = 0;

last\_link = 0;

my\_size = 0;

}

//Overloaded constructor that sets current List to contents of existing List

template <class T>

List<T>::List(const List<T> &l)

{

first\_link = 0;

last\_link = 0;

my\_size = 0;

for (Link<T> \*current = l.first\_link; current != 0; current = current->next\_link)

push\_back(current->value);

}

//Destructor that calls free() function

template <class T>

List<T>::~List()

{

free();

}

//Functions that removes all elements in List

template <class T>

void List<T>::free()

{

while (my\_size != 0)

{

pop\_back();

}

}

//Function to check if List is empty

template <class T>

bool List<T>::empty() const

{

return (my\_size == 0);

}

//Accessor function to return size of List

template <class T>

unsigned int List<T>::size() const

{

return my\_size;

}

//Returns value of the first element in List

template <class T>

T& List<T>::front() const

{

return first\_link->value;

}

//Returns value of the last element in List

template <class T>

T& List<T>::back() const

{

return last\_link->value;

}

//Adds element to front of List, sets new first\_link, increments my\_size

template <class T>

void List<T>::push\_front(const T &x)

{

Link<T> \*new\_link = new Link<T>(x);

if (new\_link == 0) {

exit(1);

}

if (empty()) {

first\_link = last\_link = new\_link;

}

else {

first\_link->prev\_link = new\_link; //n <- f

new\_link->next\_link = first\_link; //n -> f

first\_link = new\_link; //n = f

}

my\_size++;

}

//Adds element to back of List, sets new last\_link, increments my\_size

template <class T>

void List<T>::push\_back(const T &x)

{

Link<T> \*new\_link = new Link<T>(x);

if (new\_link == 0) {

exit(1);

}

if (first\_link == 0) {

first\_link = last\_link = new\_link;

}

else {

last\_link->next\_link = new\_link;//l -> n

new\_link->prev\_link = last\_link;//l <- n

last\_link = new\_link; //l = n

}

my\_size++;

}

//Removes element from front of List, sets new first\_link, decrements my\_size

template <class T>

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

{

Link<T> \*new\_link = first\_link;

first\_link = first\_link->next\_link;

if (empty()) {

return;

}

else if (my\_size == 1) {

delete first\_link;

first\_link = last\_link = 0;

}

else if (first\_link != 0) {

first\_link->prev\_link = 0;

}

else {

last\_link = 0;

}

my\_size--;

delete new\_link;

}

//Removes element from back of List, sets new last\_link, decrements my\_size

template <class T>

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

{

Link<T> \*new\_link = last\_link;

last\_link = last\_link->prev\_link;

if (empty()) {

return;

}

else if (my\_size == 1) {

delete last\_link;

first\_link = last\_link = 0;

}

else if (last\_link != 0) {

last\_link->next\_link = 0;

}

else {

first\_link = 0;

}

my\_size--;

delete new\_link;

}

//Returns pointer to first element in List

template <class T>

typename List<T>::iterator List<T>::begin() const

{

return iterator(first\_link);

}

//Returns pointer to last element in List

template <class T>

typename List<T>::iterator List<T>::end() const

{

return iterator(last\_link->next\_link);

}

//Inserts a new element before the pointer and increments my\_size

template<class T>

inline void List<T>::insert(iterator pos, const T &x)

{

Link<T> \*new\_link = new Link<T>(x);

Link<T> \*p = (pos.current\_link)->prev\_link;

if (empty()) {

return;

}

else if (pos == begin()) { //first element

push\_front(x);

}

else if (pos == ++end()) { //last element

push\_back(x);

}

else

{

new\_link->next\_link = (pos.current\_link);

new\_link->prev\_link = p;

p->next\_link = new\_link;

(pos.current\_link)->prev\_link = new\_link;

my\_size++;

}

}

//Removes element at pointer and adjusts previous and next element

template<class T>

inline void List<T>::erase(iterator &pos)

{

if (empty()) {

return;

}

else if (pos == begin()) { //first element

pop\_front();

}

else if (pos == end()) { //last element

pop\_back();

}

else {

Link<T> \*n = (pos.current\_link)->next\_link;

Link<T> \*p = (pos.current\_link)->prev\_link;

n->prev\_link = p;

p->next\_link = n;

delete (pos.current\_link);

}

my\_size--;

}

//Overloaded assignment operator that erases current List content and adds contents from new List

template <class T>

List<T> List<T>::operator=(const List<T> &l)

{

free();

iterator temp = l.begin();

while (my\_size != l.my\_size)

{

push\_back((temp.current\_link)->value);

temp++;

}

return \*this;

}

//Overloaded index operator that returns value at specified index

template<class T>

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

{

if (index < (my\_size / 2)) {

iterator iter = begin();

for (int i = 0; i != index; i++) {

iter++;

}

return \*iter;

}

else {

reverse\_iterator rIter = rbegin();

for (int i = my\_size - 1; i != index; i--) {

rIter++;

}

return \*rIter;

}

}

////Returns a reverse\_iterator pointer to last element

template<class T>

List\_iterator\_reverse<T> List<T>::rbegin()

{

return reverse\_iterator(last\_link);

}

//Returns a reverse\_iterator pointer to first element

template<class T>

List\_iterator\_reverse<T> List<T>::rend()

{

return reverse\_iterator(first\_link->prev\_link);

}

//Reverses each value, not Link, in List

template<class T>

void List<T>::reverse()

{

iterator start = begin();

reverse\_iterator last = rbegin();

T temp1, temp2;

for (unsigned int i = 0; i < my\_size / 2; i++) {

temp1 = \*start;

temp2 = \*last;

start.current\_link->value = temp2;

last.current\_link->value = temp1;

start++;

last++;

}

}

//Link class that contains data and pointers to neighboring Links

template <class T>

class Link

{

private:

Link(const T &x) : value(x), next\_link(0), prev\_link(0) {}

T value;

Link<T> \*next\_link;

Link<T> \*prev\_link;

friend class List<T>;

friend class List\_iterator<T>;

friend class List\_iterator\_reverse<T>;

};

//List\_iterator class that is used to traverse the List

template <class T> class List\_iterator

{

protected:

Link<T> \*current\_link;

friend class List<T>;

public:

typedef List\_iterator<T> iterator;

List\_iterator(Link<T> \*source\_link) : current\_link(source\_link) {}

List\_iterator() : current\_link(0) {}

List\_iterator(List\_iterator<T> \*source\_iterator) : current\_link(source\_iterator->current\_link) { }

T& operator\*(); // dereferencing operator

iterator operator=(const iterator &rhs);

bool operator==(const iterator &rhs) const;

bool operator!=(const iterator &rhs) const;

iterator& operator++(); // pre-increment, ex. ++it

iterator operator++(int); // post-increment, ex. it++

iterator& operator--(); // pre-decrement

iterator operator--(int); // post-decrement

};

//Overloaded dereference operator to get value stored in current Link

template <class T>

T& List\_iterator<T>::operator\*()

{

return current\_link->value;

}

//Overloaded assignment operator that sets value of current Link to the value in new Link

template <class T>

List\_iterator<T> List\_iterator<T>::operator=(const iterator &rhs)

{

current\_link = rhs.current\_link;

return this;

}

//Overloaded equal to operator that checks if two Links are equal

template <class T>

bool List\_iterator<T>::operator==(const iterator &rhs) const

{

return (current\_link == rhs.current\_link);

}

//Overloaded not equal to operator that checks if two Links are not equal

template <class T>

bool List\_iterator<T>::operator!=(const iterator &rhs) const

{

return (current\_link != rhs.current\_link);

}

//Overloaded pre-increment operator that increments pointer to next Link

template <class T>

List\_iterator<T>& List\_iterator<T>::operator++() // pre-increment

{

current\_link = current\_link->next\_link;

return \*this;

}

//Overloaded post-increment operator that increments pointer to next Link

template <class T>

List\_iterator<T> List\_iterator<T>::operator++(int) // post-increment

{

List\_iterator<T> new\_link = current\_link;

current\_link = current\_link->next\_link;

return new\_link;

}

//Overloaded pre-decrement operator that decrements pointer to previous Link

template <class T>

List\_iterator<T>& List\_iterator<T>::operator--() // pre-decrement

{

current\_link = current\_link->prev\_link;

return \*this;

}

//Overloaded post-decrement operator that decrements pointer to previous Link

template <class T>

List\_iterator<T> List\_iterator<T>::operator--(int) // post-decrement

{

List\_iterator<T> new\_link = current\_link;

current\_link = current\_link->prev\_link;

return new\_link;

}

//List\_iterator class that is used to traverse the List in reverse

template <class T> class List\_iterator\_reverse

{

protected:

Link<T> \*current\_link;

friend class List<T>;

public:

typedef List\_iterator\_reverse<T> reverse\_iterator;

List\_iterator\_reverse(Link<T> \*source\_link) : current\_link(source\_link) {}

List\_iterator\_reverse() : current\_link(0) {}

List\_iterator\_reverse(List\_iterator\_reverse<T> \*source\_iterator) : current\_link(source\_iterator->current\_link) { }

T& operator\*(); // dereferencing operator

reverse\_iterator operator=(const reverse\_iterator &lhs);

bool operator==(const reverse\_iterator &lhs) const;

bool operator!=(const reverse\_iterator &lhs) const;

reverse\_iterator& operator++(); // pre-increment, ex. ++it

reverse\_iterator operator++(int); // post-increment, ex. it++

reverse\_iterator& operator--(); // pre-decrement

reverse\_iterator operator--(int); // post-decrement

};

#endif

//Overloaded dereference operator to get value stored in current Link

template<class T>

T& List\_iterator\_reverse<T>::operator\*()

{

return current\_link->value;

}

//Overloaded assignment operator that sets value of current Link to the value in new Link

template<class T>

List\_iterator\_reverse<T> List\_iterator\_reverse<T>::operator=(const reverse\_iterator & lhs)

{

current\_link = lhs.current\_link;

return this;

}

//Overloaded equal to operator that checks if two Links are equal

template<class T>

bool List\_iterator\_reverse<T>::operator==(const reverse\_iterator & lhs) const

{

return (current\_link == lhs.current\_link);

}

//Overloaded not equal to operator that checks if two Links are not equal

template<class T>

bool List\_iterator\_reverse<T>::operator!=(const reverse\_iterator & lhs) const

{

return (current\_link != lhs.current\_link);

}

//Overloaded pre-increment operator that increments pointer to previous Link

template<class T>

List\_iterator\_reverse<T>& List\_iterator\_reverse<T>::operator++()

{

current\_link = current\_link->prev\_link;

return \*this;

}

//Overloaded post-increment operator that increments pointer to previous Link

template<class T>

List\_iterator\_reverse<T> List\_iterator\_reverse<T>::operator++(int)

{

List\_iterator\_reverse<T> new\_link = current\_link;

current\_link = current\_link->prev\_link;

return new\_link;

}

//Overloaded pre-decrement operator that decrements pointer to next Link

template<class T>

List\_iterator\_reverse<T>& List\_iterator\_reverse<T>::operator--()

{

current\_link = current\_link->next\_link;

return \*this;

}

//Overloaded post-decrement operator that decrements pointer to next Link

template<class T>

List\_iterator\_reverse<T> List\_iterator\_reverse<T>::operator--(int)

{

List\_iterator\_reverse<T> new\_link = current\_link;

current\_link = current\_link->next\_link;

return new\_link;

}

**Sample Run:**

Script started on Tue 28 Nov 2017 07:44:45 PM UTC

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

See "man sudo\_root" for details.

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

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

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

List in order

44

33

11

22

List in order after being reversed

22

11

33

44

List in reverse order

44

33

11

22

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

Script done on Tue 28 Nov 2017 07:45:01 PM UTC