Eric Blasko

Data Structures CSE330

Winter 2018

Lab 5 - List

* **Status**

100% complete

* **Complexity Analaysis**

Class List

The following functions are O(1) as they are constant in time

* List()
* List & operator=(const List & rhs)
* List(List && rhs);
* List & operator=(List && rhs)
* init()
* iterator begin()
* const\_iterator begin() const
* iterator end()
* const\_iterator end() const
* int size() const
* bool empty() const
* void push\_front(const Object & x)
* void push\_front(Object && x)
* void push\_back(const Object & x)
* void push\_back(Object && x)
* iterator insert(iterator, const Object &)
* iterator insert(iterator, Object &&)
* void pop\_front()
* void pop\_back()
* iterator erase(iterator)

The following functions are O(n), were n is the size of the list

* ~List()
* List(const List & rhs)
* void clear()
* iterator erase(iterator, iterator);

Class const\_iterator

All functions for class const\_iterator are O(1) as they are constant in time

* const\_iterator()
* const Object & operator\*() const
* const\_iterator & operator++()
* const\_iterator operator++(int)
* const\_iterator & operator--()
* const\_iterator operator--(int)
* bool operator==(const const\_iterator & rhs) const
* bool operator!=(const const\_iterator & rhs) const
* Object & retrieve() const
* const\_iterator(Node \*p)

Class iterator

All functions for class iterator are O(1) as they are constant in time

* iterator(Node \*p)
* iterator()
* Object & operator\*()
* const Object & operator\*() const
* iterator & operator++();
* iterator operator++(int);
* iterator & operator--();
* iterator operator--(int);
* **Source Code**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Eric Blasko

\* List.h

\* 02/14/2018

\* This class minics the main functions found in the STL class list. There is two

\* nested class, const\_iterator and iterator, along with a struct Node.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef LIST\_H

#define LIST\_H

using namespace std;

//Function for class list

template <typename Object>

class List

{

struct Node;

int theSize;

Node \*head;

Node \*tail;

void init();

public:

class const\_iterator;

class iterator;

List() {init();}

List(const List & rhs);

List(List && rhs);

~List();

List & operator=(const List & rhs);

List & operator=(List && rhs);

iterator begin() {return head->next;}

const\_iterator begin() const {return head->next;}

iterator end() {return tail;}

const\_iterator end() const {return tail;}

int size() const {return theSize;}

bool empty() const {return theSize == 0;}

void clear();

Object & front() {return \*begin();}

const Object & front() const {return \*begin();}

Object & back() {return \*--end();}

const Object & back() const {return \*--end();}

void push\_front(const Object & x) {insert(begin(),x);}

void push\_front(Object && x) {insert(begin(),move(x));}

void push\_back(const Object & x) {insert(end(), x);}

void push\_back(Object && x) {insert(end(),move(x));}

void pop\_front() {erase(begin());}

void pop\_back() {erase(--end());}

iterator insert(iterator, const Object &);

iterator insert(iterator, Object &&);

iterator erase(iterator);

iterator erase(iterator, iterator);

};

//Sets defaults during creation of new Node

template <typename Object>

void List<Object>::init()

{

theSize = 0;

head = new Node();

tail = new Node();

head->next = tail;

tail->prev = head;

}

//copy constructor

template <typename Object>

List<Object>::List(const List & rhs)

{

init();

for(auto & x: rhs)

push\_back(x);

}

//move operator

template <typename Object>

List<Object> & List<Object>::operator=(List && rhs)

{

swap(theSize, rhs.theSize);

swap(head, rhs.head);

swap(tail, rhs.tail);

return \*this;

}

//insert with a const object (copy)

template <typename Object>

typename List<Object>::iterator List<Object>::insert(iterator itr, const Object & x)

{

Node \*p = itr.current;

theSize++;

return {p->prev = p->prev->next = new Node{x, p->prev, p}};

}

//move constructor

template <typename Object>

List<Object>::List(List && rhs)

:theSize{rhs.theSize}, head{rhs.head}, tail{rhs.tail}

{

rhs.theSize = 0;

rhs.head = nullptr;

rhs.tail = nullptr;

}

//destructor

template <typename Object>

List<Object>::~List()

{

clear();

delete head;

delete tail;

}

//copy operator

template <typename Object>

List<Object> & List<Object>::operator=(const List & rhs)

{

List copy = rhs;

std::swap(\*this,copy);

return \*this;

}

//clear entire content of list

template <typename Object>

void List<Object>::clear()

{

while(!empty())

pop\_front();

}

//insert with rvalue (move)

template <typename Object>

typename List<Object>::iterator List<Object>::insert(iterator itr, Object && x)

{

Node \*p = itr.current;

theSize++;

return {p->prev = p->prev->next = new Node{move(x),p->prev, p}};

}

//erase node at iterator.iterator returns pointer to next node

template <typename Object>

typename List<Object>::iterator List<Object>::erase(iterator itr)

{

Node \*p = itr.current;

iterator retVal{p->next};

p->prev->next = p->next;

p->next->prev = p->prev;

delete p;

theSize--;

return retVal;

}

//erase from iterator to iterator

template <typename Object>

typename List<Object>::iterator List<Object>::erase(iterator from, iterator to)

{

for( iterator itr = from; itr != to;)

itr = erase(itr);

return to;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Struct Node

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <typename Object>

struct List<Object>::Node

{

Object data;

Node \*prev;

Node \*next;

Node(const Object & d = Object{}, Node \* p = nullptr, Node \* n = nullptr)

: data{d}, prev{p}, next{n} {}

Node(Object && d, Node \*p = nullptr, Node \*n = nullptr)

: data{move(d)}, prev{p}, next{n} {}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Class const\_iterator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <typename Object>

class List<Object>::const\_iterator

{

public:

const\_iterator(): current{nullptr} {}

const Object & operator\*() const {return retrieve();}

const\_iterator & operator++();

const\_iterator operator++(int);

const\_iterator & operator--();

const\_iterator operator--(int);

bool operator==(const const\_iterator & rhs) const {return current == rhs.current;}

bool operator!=(const const\_iterator & rhs) const {return current != rhs.current;}

protected:

Node \*current;

Object & retrieve() const {return current->data;}

const\_iterator(Node \*p): current{p} {}

friend class List<Object>;

};

//increase current position of iterator(post)

template <typename Object>

typename List<Object>::const\_iterator & List<Object>::const\_iterator::operator++()

{

current = current->next;

return \*this;

}

//increase current position of iterator(pre)

template <typename Object>

typename List<Object>::const\_iterator List<Object>::const\_iterator::operator++(int)

{

const\_iterator old = \*this;

++(\*this);

return old;

}

//decrease current position of iterator(post)

template <typename Object>

typename List<Object>::const\_iterator & List<Object>::const\_iterator::operator--()

{

current = current->prev;

return \*this;

}

//decrease current position of iterator(pre)

template <typename Object>

typename List<Object>::const\_iterator List<Object>::const\_iterator::operator--(int)

{

const\_iterator old = \*this;

--(\*this);

return old;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Class iterator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <typename Object>

class List<Object>::iterator: public List<Object>::const\_iterator

{

public:

iterator() {}

Object & operator\*() {return List<Object>::const\_iterator::retrieve();}

const Object & operator\*() const {return List<Object>::const\_iterator::operator\*();}

iterator & operator++();

iterator operator++(int);

iterator & operator--();

iterator operator--(int);

protected:

iterator(Node \*p): List<Object>::const\_iterator{p} {}

friend class List<Object>;

};

//increase current position of iterator(post)

template <typename Object>

typename List<Object>::iterator & List<Object>::iterator::operator++()

{

this->current = this->current->next;

return \*this;

}

//increase current position of iterator(pre)

template <typename Object>

typename List<Object>::iterator List<Object>::iterator::operator++(int)

{

iterator old = \*this;

++(\*this);

return old;

}

//decrease current postion of iterator(post)

template <typename Object>

typename List<Object>::iterator & List<Object>::iterator::operator--()

{

this->current = this->current->prev;

return \*this;

}

//decrease current position of iterator(pre)

template <typename Object>

typename List<Object>::iterator List<Object>::iterator::operator--(int)

{

iterator old = \*this;

--(\*this);

return old;

}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Eric Blasko

\* List\_test1.cpp

\* 02/14/2018

\* The program test the functions within the class List.h. Different ways of

\* adding and removing data to list objects will be test, returning assert error

\* if the test fails. If successful, end of test will print successs to console

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <cassert>

#include "List.h"

using namespace std;

//Main function that test function methods from the class List.h

int main()

{

List<int> l;

assert(l.size() == 0);

assert(l.empty());

l.push\_front(44);

assert(!l.empty());

assert(l.front() == 44);

assert(l.back() == 44);

l.push\_front(33);

assert(l.size() == 2);

assert(l.front() == 33);

assert(l.back() == 44);

l.push\_front(22);

List<int>::iterator it = l.begin();

l.insert(it, 11);

it = l.begin();

assert(\*it == 11);

assert(\*++it == 22);

assert(\*++it == 33);

assert(\*++it == 44);

it = l.begin();

++it;

++it;

++it;

l.insert(it, 38);

List<int>::iterator it2 = l.begin();

assert(\*it2 == 11);

assert(\*++it2 == 22);

assert(\*++it2 == 33);

assert(\*++it2 == 38);

assert(\*++it2 == 44);

l.pop\_front();

it2 = l.begin();

assert(\*it2 == 22);

assert(\*++it2 == 33);

assert(\*++it2 == 38);

assert(\*++it2 == 44);

l.pop\_back();

List<int> copy = l;

assert(copy.size() == 3);

List<int>::iterator it3 = copy.begin();

assert(\*it3 == 22);

assert(\*++it3 = 33);

copy.erase(it3);

assert(copy.size() == 2);

it3 = copy.begin();

assert(\*it3 == 22);

assert(\*++it3 == 38);

cout << "SUCCESS\n";

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Eric Blasko

\* List\_test2.cpp

\* 02/14/2018

\* This program test the functionality of the class List.h. Main functions to be

\* tested are the push\_back, pop\_front, pop\_back functions and iterator functions.

\* If program works properly it will display the contents of the list

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include "List.h"

using namespace std;

//Main function that tests the methods of the class List.h

int main()

{

List<int> l;

l.push\_back(44);

l.push\_back(33);

l.push\_back(11);

l.push\_back(22);

List<int> m(l);

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

for( ; itr != m.end(); itr++)

cout << \*itr << " ";

cout << "\n------------\n";

l.pop\_back();

m = l;

for(itr = m.begin(); itr != m.end(); itr++)

cout << \*itr << " ";

cout << "\n------------\n";

m.pop\_front();

for(itr = m.begin(); itr != m.end(); itr++)

cout << \*itr << " ";

cout << "\n------------\n";

m.clear();

for(itr = m.begin(); itr != m.end(); itr++)

cout << \*itr << " ";

cout << endl;

}

* **Sample Runs**

**Test 1**

Script started on 2018-02-13 11:30:18-0800

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ g++ -c List)[K\_ts[Kest1.cpp

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ g++ List\_test1.cpp[K[K[Ko

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ ./a.out

SUCCESS

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ exit

Script done on 2018-02-13 11:30:59-0800

**Test 2**

Script started on 2018-02-13 11:31:12-0800

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ g++ -c List\_test2.cpp

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ g++ te[K[KList\_test2.o

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ ./a.out

44 33 11 22

------------

44 33 11

------------

33 11

------------

]0;005670557@csusb.edu@csevnc:~/cse330/lab5[005670557@csusb.edu@csevnc lab5]$ exit

Script done on 2018-02-13 11:31:38-0800

* **Discussion**

2. What is the output of List\_test2.cpp? Is it what you expected?

Output was:

44 33 11 22

------------

44 33 11

------------

33 11

------------

This is what was expected. The first line was what was original data pushed onto the list. The second line is with the last number popped off. The third line is with the first number popped off. The final loop did not print anything because m.clear() removed all contents from the list.

3. What happens if accessor iterator::operator\*() is commented out? Answer: bottom of page 97.

If iterator::operator\*() was commented out, the original implementation in const\_iterator would be hidden .