

[Assignment 7 & 20152008, KUSDavletov Ernar]

[OOP]Assign7

Student ID: 20152008

Name: KUSDavletov Ernar

Date: 2017.06.13

1. Descriptions

In this project, we had deal with buffer, which is implemented as a circular doubly-linked list (LL or II) and an iterator to traverse the LL.

Circular doubly-linked list – a linked list is a data structure that can grow and shrink dynamically and allows direct access to elements using indexing. In a circular doubly linked list, each node contains two pointers: one to the next node in the list, and the other to the previous node in the list. Moreover, the last node points back to the first node in the list, not to NULL, as the next node, and the first node points back to the last node in the list, not to NULL, as the previous node.

Iterator – a pointer-like object that accesses the elements in a circular doubly-linked list. The buffer class supports two major operations of produce and consume. For a produce operation, a node is created and added to the linked list, while for a consume operation, a node is deleted from the list.

In this project, we use 3 classes: Node, Buffer, Iterator.

2. Code

Node.cpp

```
#include "Node.h"
#include <iostream>

Node::Node(){ // default constructor
    next = NULL;
    prev = NULL;
    word.clear();
}

Node::Node(std::string data){ // constructor with single argument
    next = this;
    prev = this;
    word = data;
}

Node::Node(std::string data, Node *value){ // constructor with 2 arguments, second argument is a pointer to prev
    next = value->next;
    value->next = this;
    prev = value;
    word = data;
    if (value->prev == value){
        value->prev = this;
    }
}

std::string& Node::getWord(){ // accessor for word - returns contents of word
    return word;
}

Node* Node::getNext() const{ // accessor for next - returns pointer to node to which next is pointing
    return next;
}

Node* Node::getPrev() const{ // accessor for next - returns pointer to node to which prev is pointing
    return prev;
}

void Node::setWord(std::string data){ // mutator for word - changes string to value passed in
    word = data;
}

void Node::setNext(Node *value){ // mutator for next - changes pointer to value passed in
    next = value;
}

void Node::setPrev(Node *value){ // mutator for prev - changes pointer to value passed in
    prev = value;
}
```

Iterator.cpp

```
#include "Iterator.h"
#include <iostream>

Iterator::Iterator(Node *currIn){ // initializes curr to currIn
    curr = currIn;
}

const std::string Iterator::operator*() const{ // dereference curr, returns plain text string
    return curr->getWord();
}

Iterator& Iterator::operator++(){ // pre-increment operator
    curr = curr->getNext();
    return *this;
}

Iterator Iterator::operator++(int){ // post-increment operator
    Iterator temp = *this;
    temp.curr = temp.curr->getNext();
    return temp;
}

Iterator Iterator::operator+(const int &a){ // for iterator math - for example it = it + 2;
    Iterator temp = *this;
    for (int i = 1; i <= a; i++)
        temp.curr = temp.curr->getNext();
    return temp;
}

Iterator Iterator::operator+=(const int &a){ // for iterator math - for example it += 2;
    for (int i = 1; i <= a; i++)
        curr = curr->getNext();
    return *this;
}

Iterator& Iterator::operator--(){ // pre-decrement operator
    curr = curr->getPrev();
    return *this;
}

Iterator Iterator::operator--(int a){ // post-decrement operator
    Iterator temp = *this;
    temp.curr = temp.curr->getPrev();
    return temp;
}

Iterator Iterator::operator-(const int &a){ // for iterator math - for example it = it - 2;
    Iterator temp = *this;
    for (int i = 1; i <= a; i++)
        temp.curr = temp.curr->getPrev();
    return temp;
}

Iterator Iterator::operator-=(const int &a){ // for iterator math - for example it -= 2;
    for (int i = 1; i <= a; i++)
        curr = curr->getPrev();
}
```

```
    return *this;
}
bool Iterator::operator==(const Iterator &other){ // true if 2 iterators are pointing to the same node
    if (curr == other.curr)
        return true;
    else
        return false;
}
bool Iterator::operator!=(const Iterator &other){ // true if 2 iterators are pointing to different nodes
    if (curr != other.curr)
        return true;
    else
        return false;
}
Node*& Iterator::getCurr(){ // returns the node to which the iterator points
    return curr;
}
```

Buffer.cpp

```
#include "Buffer.h"
#include <iostream>

Buffer::Buffer(){ // default constructor - creates empty linked list of count = 0
    head = NULL;
    count = 0;
}
Buffer::Buffer(const Buffer &other){ // copy constructor for buffer creates a deep copy
    head = NULL;
    count = 0;
    copy(other);
}
Buffer::~Buffer(){ // destructor - calls clear();
    clear();
}
Buffer& Buffer::operator=(const Buffer &original){ // overloaded assignment operator - calls copy()
    copy(original);
    return *this;
}
Node* Buffer::getHead(){ // accessor - returns the head
    return head;
}
Node* Buffer::getTail(){ // accessor - returns the tail node
    return head->getPrev();
}

// returns the string in the head node of the ll, this operation is not valid for an empty ll (i.e. buffer)
// (If it is called on an empty buffer, return an empty string "")
std::string Buffer::getHeadElement(){
    if (count == 0)
        return std::string("");
    else
        return head->getWord();
}
// returns the string in the tail node of the ll, this operation is not valid for an empty ll (i.e. buffer)
// (If it is called on an empty buffer, return an empty string "")
std::string Buffer::getTailElement(){
    if (count == 0)
        return std::string("");
    else
        return getTail()->getWord();
}
// creates a node containing str, adds the string at the head of the ll
// (i.e. before the current head node), after the operation, head node is the newly added one
void Buffer::produceAtHead(const std::string str){
    if (count == 0){
        Node *new_head = new Node(str);
```

```

        head = new_head;
        count += 1;
        return;
    }
    Node *new_head = new Node(str);
    new_head->setNext(getHead());
    new_head->setPrev(getTail());
    getTail()->setNext(new_head);
    getHead()->setPrev(new_head);
    head = new_head;
    count += 1;
}
// creates a node containing str, adds the string at the end of the ll
// (i.e. after the current tail node)
void Buffer::produceAtTail(const std::string str){
    if (count == 0){
        Node *new_head = new Node(str);
        head = new_head;
        count += 1;
        return;
    }
    Node *new_tail = new Node(str);
    new_tail->setNext(getHead());
    new_tail->setPrev(getTail());
    getTail()->setNext(new_tail);
    getHead()->setPrev(new_tail);
    count += 1;
}
// inserts Node containing "str" into the ll in front of "pos"
// return an iterator that points to the the newly inserted elements.
// any iterators after pos are considered invalid
Iterator Buffer::produceAtMiddle(Iterator pos, const std::string str){
    Node *new_middle = new Node(str);
    new_middle->setNext(pos.getCurr());
    new_middle->setPrev((pos - 1).getCurr());
    (pos - 1).getCurr()->setNext(new_middle);
    pos.getCurr()->setPrev(new_middle);
    count += 1;
    return pos;
}
// buffer must not be empty before calling this, deletes the first element (i.e. current head node)
void Buffer::consumeAtHead(){
    if (count == 0)
        return;
    else if (count == 1){
        delete head;
        count -= 1;
        return;
    }
}

```

```

    else{
        head->getNext()->setPrev(head->getPrev());
        head->getPrev()->setNext(head->getNext());
        Node *temp = head;
        head = head->getNext();
        delete temp;
        count -= 1;
    }
}
// buffer must not be empty before calling this, deletes the last element (i.e. current tail node)
void Buffer::consumeAtTail(){
    if (count == 0)
        return;
    else if (count == 1){
        delete head;
        count -= 1;
        return;
    }
    else{
        Node *temp = getTail();
        getTail()->getPrev()->setNext(head);
        head->setPrev(getTail()->getPrev());
        delete temp;
        count -= 1;
    }
}
void Buffer::consume(Iterator it){ // "it" is pointing to the element to be removed
    Node *temp = it.getCurr();
    (it - 1).getCurr()->setNext((it + 1).getCurr());
    (it + 1).getCurr()->setPrev((it - 1).getCurr());
    delete temp;
    count -= 1;
}
// erases all nodes in the range [s,t), s is erased and all nodes up to but not including t are erased
void Buffer::consume(Iterator s, Iterator t){
    if (s == t){
        return;
    }
    consume(s + 1, t);
    consume(s);
}
unsigned Buffer::size() const{ // returns the number of elements in the ll
    return count;
}
bool Buffer::isEmpty(){ // returns true if the ll is empty, otherwise it returns false
    if (count == 0)
        return true;
    else
        return false;
}

```



```

}
// brackets operator allows indexing into the ll, "fakes" random access to the ith element of the ll
// returns contents of ith element of the ll, if ll looks like: we -> are -> family ->
// then element 2 would be "family", remember the first element is the 0th element
std::string& Buffer::operator[](int i) const{
    Iterator it = head;
    it += i;
    return it.getCurr()->getWord();
}
Iterator Buffer::getHeadltr(){ // returns the iterator pointing to the first node in the ll
    return Iterator(head);
}
Iterator Buffer::getTailltr(){ // returns an iterator pointing the tail node in ll
    return Iterator(head->getPrev());
}
Iterator Buffer::getNextltr(Iterator it){ // returns an iterator of the next node pointed by it
    return (it + 1);
}
Iterator Buffer::getPrevltr(Iterator it){ // returns an iterator of the previous node pointed by it
    return (it - 1);
}
// prints out the ll like the following: contents of node, followed by a space,
// followed by -> followed by a space after list is printed, it skips a line
// ie has 2 endls, if list is empty, it prints "->" followed by 2 endls
void Buffer::print(){
    if(count == 0)
        std::cout << "->";
    else{
        Node *p = head;
        while (p != getTail()){
            std::cout << p->getWord() << " -> ";
            p = p->getNext();
        }
        std::cout << p->getWord();
    }
    std::cout << "\n\n";
}
// makes a deep copy of the ll for copy constructor and assignment operator "other" is copied
void Buffer::copy(const Buffer &other){
    clear();
    if (other.size() == 0)
        return;
    head = new Node(other[0]);
    count = 1;
    Node *p = other.head;
    unsigned i = 1;
    unsigned n = other.size();
    while(i < n){
        p = p->getNext();
    }
}

```

```

        produceAtTail(p->getWord());
        i += 1;
    }
}
// clears the ll by deleting all nodes, calls the recursive function deleteBuffer()
void Buffer::clear(){
    deleteBuffer(head);
}
// recursively deletes the ll p, THIS FUNCTION MUST BE RECURSIVE
void Buffer::deleteBuffer(Node *p){
    if (count == 0 || p == NULL)
        return;
    else if (p->getNext() == p){
        delete p;
        count -= 1;
        return;
    }
    if (count >= 1){
        count -= 1;
        deleteBuffer(p->getNext());
    }
    delete p;
}
// non-member function, locates "str" in the range [first,last]
// if "str" is found in the range, return true, if "str" is not found, return false
bool findNode(Iterator first, Iterator last, std::string str){
    Iterator it;
    it = first;
    while(true){
        if (it.getCurr()->getWord() == str)
            return true;
        it += 1;
        if (it == last){
            if (it.getCurr()->getWord() == str)
                return true;
            break;
        }
    }
}
return false;
}

```

3. Sample input

testNode input:

```
#include <iostream>
#include "Node.h"

using namespace std;

int main() {
    Node n1;
    Node n2("Node2");
    Node n3("Node3", &n1);
    Node n4("Node4", &n2);
    cout << "Node1: " << n1.getWord() << endl;
    cout << "Node2: " << n2.getWord() << endl;
    cout << "Node3: " << n3.getWord() << endl;
    cout << "Node4: " << n4.getWord() << endl;
    if (n3.getPrev() == &n1 && n1.getNext() == &n3) {
        cout << "Node1 -> Node3" << endl;
    }
    if (n4.getPrev() == &n2 && n2.getNext() == &n4) {
        cout << "Node2 -> Node4" << endl;
    }
    n1.setWord("Node1");
    cout << "Node1: " << n1.getWord() << endl;
    n3.setNext(&n2);
    n2.setPrev(&n3);
    if (n3.getPrev() == &n1 && n1.getNext() == &n3 && n3.getNext() == &n2 && n2.getPrev() == &n3 && n2.getNext() == &n4 && n4.getPrev() == &n2) {
        cout << "Node1 -> Node3 -> Node2 -> Node4" << endl;
    }
    return 0;
}
```

testIterator input:

```
#include <iostream>
#include "Iterator.h"

using namespace std;

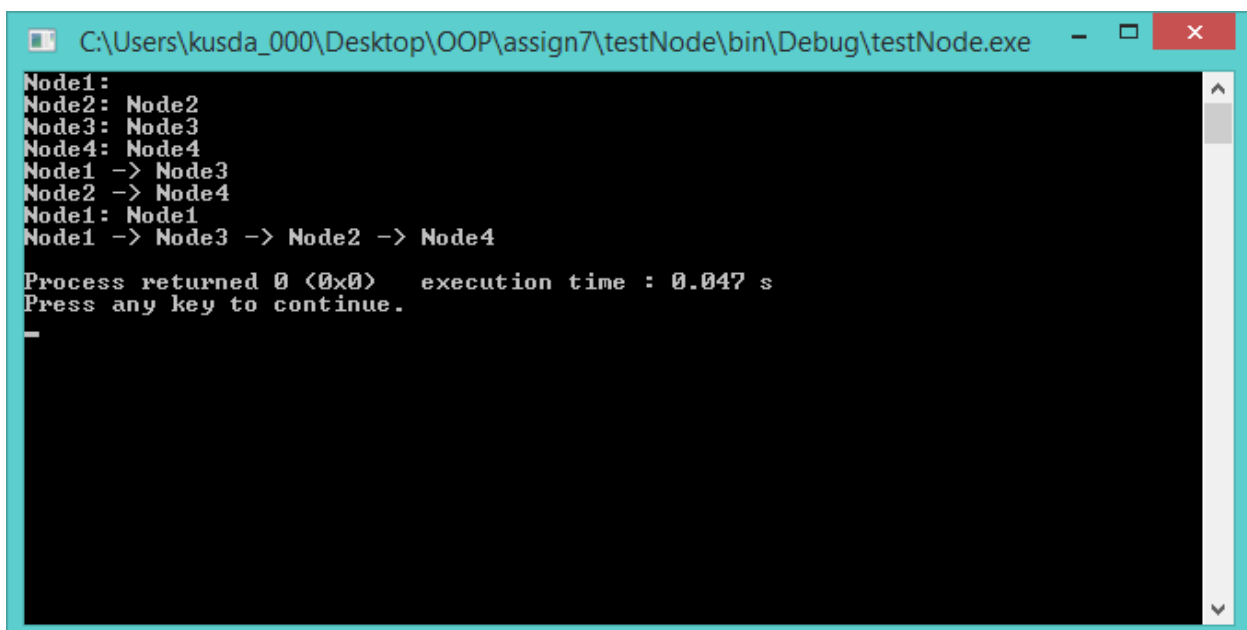
int main() {
    Node n1("Node1 for Iterator");
    Node n2("Node2 for Iterator");
    Iterator t1(&n1);
    Iterator t2(&n2);
    cout << *t1 << endl;
    cout << *t2 << endl;
    if (t1.getCurr() == &n1 && t2.getCurr() == &n2) {
        cout << "getCurr works!" << endl;
    }
    if (t1 != t2) {
        cout << "!= operator works!" << endl;
    }
    if (t1 == t1) {
        cout << "== operator works!" << endl;
    }
    return 0;
}
```

testBuffer input:

```
int main() {
    Buffer v1;
    Iterator it;
    v1.produceAtHead("Is");
    v1.produceAtHead("Name");
    v1.produceAtHead("My");
    v1.produceAtTail("James");
    v1.produceAtTail("Bond");
    cout << "count = " << v1.size() << endl; // print "5"
    cout << "first string= " << v1.getHeadElement() << endl; // print "My"
    cout << "last string= " << v1.getTailElement() << endl; // print "Bond"
    v1.print(); // My -> Name -> Is -> James -> Bond
    v1.consumeAtHead();
    v1.consumeAtTail();
    v1.print(); // Name -> Is -> James
    it = v1.getHeadItr();
    it = v1.getNextItr(it);
    v1.consume(it);
    v1.print(); // Name -> James
    it = v1.getTailItr();
    it = v1.produceAtMiddle(it, "Porter"); // insert in middle
    v1.print(); // Name -> Porter -> James
    cout << findNode(v1.getHeadItr(), v1.getTailItr(), "Bond") << endl; // test find
    v1.consumeAtHead();
    v1.consumeAtHead();
    v1.consumeAtHead();
    cout << "count = " << v1.size() << endl; // 0
    v1.print(); // ->
    if (v1.isEmpty()) // test isEmpty()
        cout << "empty buffer" << endl;
    return 0;
}
```

4. Sample output

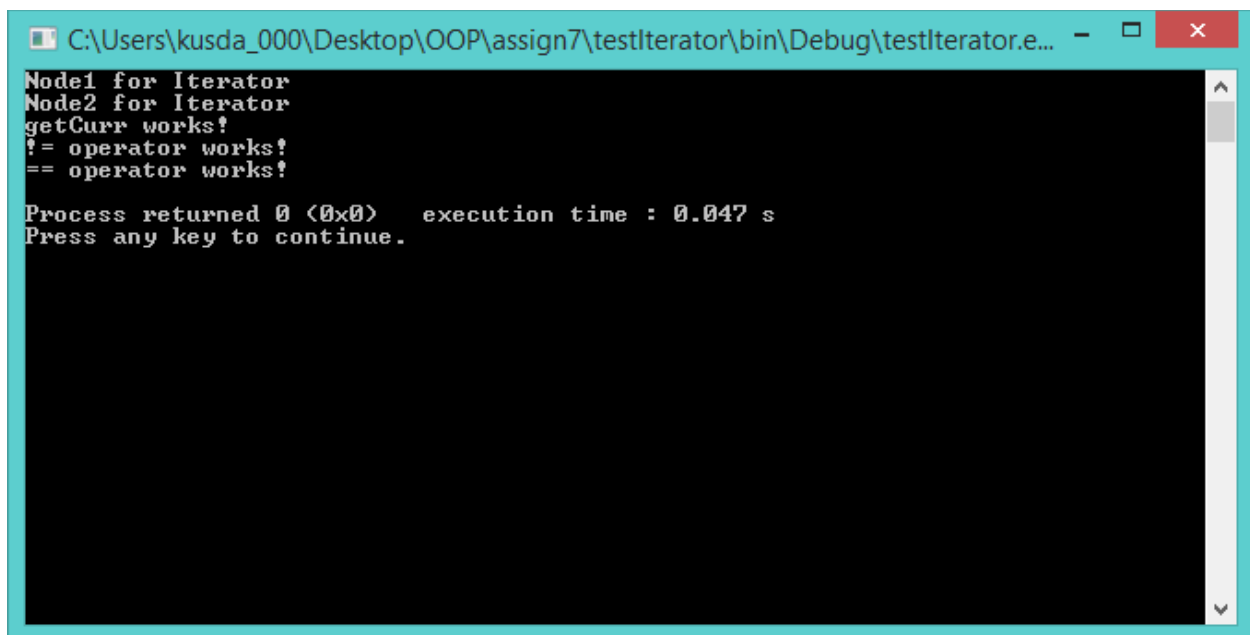
testNode output:



```
C:\Users\kusda_000\Desktop\OOP\assign7\testNode\bin\Debug\testNode.exe
Node1:
Node2: Node2
Node3: Node3
Node4: Node4
Node1 -> Node3
Node2 -> Node4
Node1: Node1
Node1 -> Node3 -> Node2 -> Node4

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

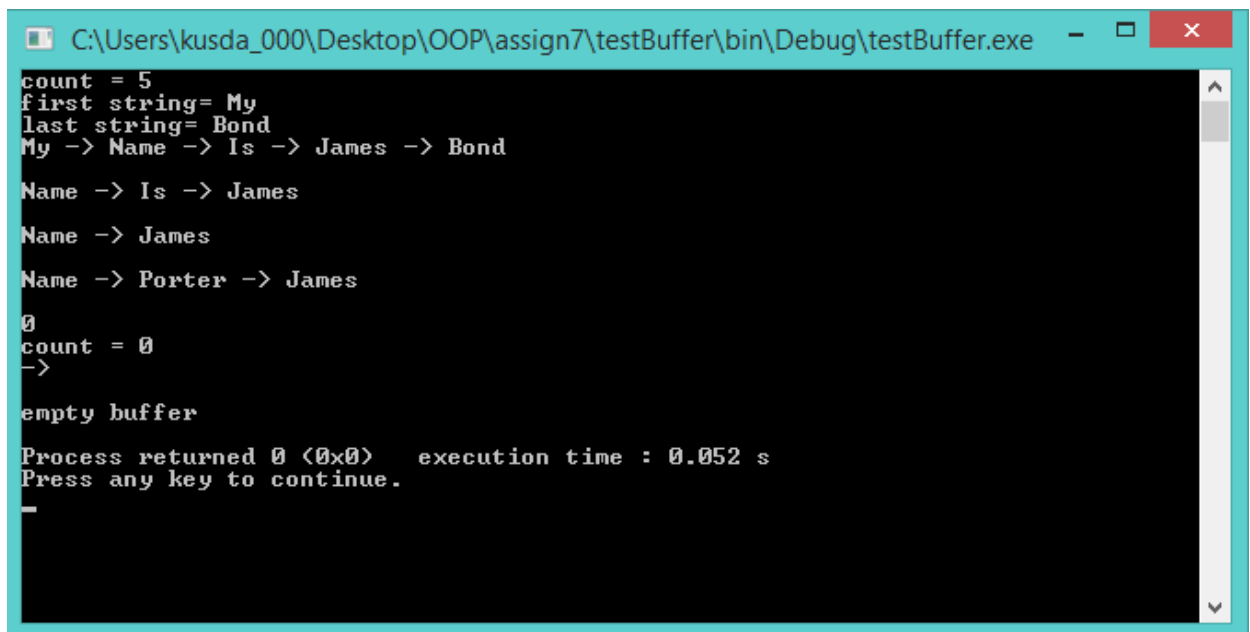
testIterator output:



```
C:\Users\kusda_000\Desktop\OOP\assign7\testIterator\bin\Debug\testIterator.e... - [ ] [X]
Node1 for Iterator
Node2 for Iterator
getCurr works!
!= operator works!
== operator works!

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

testBuffer output:



```
C:\Users\kusda_000\Desktop\OOP\assign7\testBuffer\bin\Debug\testBuffer.exe - [ ] [X]
count = 5
first string= My
last string= Bond
My -> Name -> Is -> James -> Bond

Name -> Is -> James
Name -> James
Name -> Porter -> James

0
count = 0
->

empty buffer

Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
-
```