資料結構報告 HW3

楊道恩

August 24, 2024

CONTENTS

解題說明

Implement Polynomial with **circular list with header node.** Available list is implemented to delete polynomial classes efficiently.
The input is assumed to be a sequence of integers in the form $n, c_1, e_1, c_2, e_2, c_3, e_3, \ldots, c_n, e_n$. Where c_i = coefficients, e_i = exponents, n = number of terms in a polynomial. The exponents are in decreasing order.

Write and test the following functions:

(a) *istream*& **operator>>**(*istream*& *is*, *Polynomial*& *x*): Read in an input polynomial and convert it to its circular list representation using a header node.

(b) *ostream*& **operator<<**(*ostream*& *os*, *Polynomial*& *x*): Convert *x* from its linked list representation to its external representation and output it.

(c) *Polynomial*::*Polynomial*(**const** *Polynomial*& *a*) [Copy Constructor]: Initialize the polynomial ***this** to the polynomial *a*.

(d) **const** *Polynomial*& *Polynomial*::**operator=**(**const** *Polynomial*& *a*) **const** [Assignment Operator]: Assign polynomial *a* to ***this**.

(e) *Polynomial*::~*Polynomial*() [Destructor]: Return all nodes of the polynomial ***this** to the available-space list.

(f) *Polynomial* **operator+** (**const** *Polynomial*& *b*) **const** [Addition]: Create and return the polynomial ***this** + *b*.

(g) *Polynomial* **operator−** (**const** *Polynomial*& *b*) **const** [Subtraction] : Create and return the polynomial ***this** − *b*.

(h) *Polynomial* **operator***(**const** *Polynomial*& *b*) **const** [Multiplication]: Create and return the polynomial ***this** * *b*.

(i) **float** *Polynomial*::*Evaluate*(**float** *x*) **const**: Evaluate the polynomial ***this** at *x* and return the result.

Figure 1. Functions to implement in Polynomial class

設計與實作

For full info, check the source code LinkedPolynomial.cpp

```cpp
class Polynomial; //forward declaration
template <class T> class circularList;

//Circular list w/Header
template <class T>
class chainNode {
	friend class circularList<T>;
private:
	T data;
	chainNode<T>* link;
public:
	chainNode() { link = nullptr; }
	chainNode(const T& d) { data = d; link = nullptr; }
	chainNode(const T& d, chainNode<T>* next) { data = d; link = next; }
};
```

Figure 2.1 The class chainNode

```cpp
template <class T>
class circularList {
public:
	circularList() { //constructor
		head = new chainNode<T>();
		head->link = head;
		last = head;
		av = new chainNode<T>();
	}
	~circularList();
	//forward iterator
	class chainIterator {
	public:
		//typedefs
		//constructor
		chainIterator(chainNode<T>* startNode = 0) {
			current = startNode;
		}
		//dereferencing operators * and ->
		T& operator *() const { return current->data; }
		T* operator ->() const { return &current->data; }

		//increment
		chainIterator& operator ++() {//preincrement
			current = current->link; return *this;
		}
```

```
		chainIterator operator ++(int) {//postincrement
			chainIterator old = *this;
			current = current->link;
			return old;
		}

		//equality testing
		bool operator!=(const chainIterator right)const {
			return current != right.current;
		}
		bool operator==(const chainIterator right)const {
			return current == right.current;
		}
	private:
		chainNode<T>* current;
	};
	chainIterator header() const { return chainIterator(head); }
	chainIterator begin() const { return chainIterator(head->link); }
	chainIterator end() const { return chainIterator(last); }
	//chain manipulation
	void insertBack(const T& e);
	void insertFront(const T& e);
	//void reverse();
private:
	chainNode<T>* head; //header node
	//chainNode<T>* first;
	chainNode<T>* last;
	chainNode<T>* av;
};
```

Figure 2.2 The class circularList with header node, and its iterator

```
template <class T>
circularList<T>::~circularList() {
	if (last) {
		chainNode<T>* f = last->link;
		last->link = av;
		av = f;
		last = head;
	}
}
```

Figure 2.3 Deconstructor of circularList by using available space list

```
template <class T>
void circularList<T>::insertBack(const T& e) {
	//inserting at back
	chainNode<T>* newNode = new chainNode<T>(e);
	newNode->link = head;
	last->link = newNode;
	last = newNode;
```

```
}

template<class T>
void circularList<T>::insertFront(const T& e) {
	//insert element at the 'front' of the circular list
	//*this, where last points to the last node in the list
	chainNode<T>* newNode = new chainNode<T>(e);
	if (last) { //non empty list
		newNode->link = last->link;
		last->link = newNode;
	}
	else { //empty list
		last = newNode;
		newNode->link = newNode;
	}
}
```

Figure 2.4 Insertion to circularList

```
struct term {
	float coef = 0;
	int exp = 0;
	term set(float c, int e) { coef = c; exp = e; return *this; };
};

class Polynomial {
public:
	Polynomial();
	Polynomial(const Polynomial& a); //copy constructor
	~Polynomial();
	friend istream& operator>>(istream& in, Polynomial& x);
	friend ostream& operator<<(ostream& out, Polynomial& x);
	const Polynomial& operator=(const Polynomial& a);
	Polynomial operator+(const Polynomial& b) const;
	Polynomial operator-(const Polynomial& b) const;
	Polynomial operator*(const Polynomial& b) const;
	float Eval(float x) const;
private:
	circularList<term> poly;
};
```

Figure 2.5 The class Polynomial

```
Polynomial::Polynomial() {
	poly.header()->exp = -1;
}

Polynomial::Polynomial(const Polynomial& a) {
	poly.header()->exp = -1;
	term temp;
	circularList<term>::chainIterator ai = a.poly.begin();
```

```
        while (ai->exp!=-1) {
                poly.insertBack(temp.set(ai->coef, ai->exp));
                ai++;
        }
}

Polynomial::~Polynomial() {
        //Destructor. Return all nodes of Polynomial *this and return
        //to the available space list
        poly.~circularList();
}
```

Figure 2.6 The constructor and destructor of Polynomial

```
istream& operator>>(istream& in, Polynomial& x){
        //Read in an input polynomial and convert it to circular list
        term t;
        in >> t.coef >> t.exp;
        x.poly.insertBack(t);
        return in;
}

ostream& operator<<(ostream& out, Polynomial& x) {
        //Convert x from its linked list representation to its
        //external expression and output it
        circularList<term>::chainIterator a = x.poly.begin();
        while (1) {
                if (a->exp == -1) { return out; }
                if (a->exp > 1 || a->exp < 0) {
                        out << a->coef << "x^" << a->exp;
                }
                else if(a->exp == 1) {
                        out << a->coef << "x";
                }
                else {
                        out << a->coef;
                }
                a++;
                if (a->coef > 0 && a->exp != -1) {
                        //output "+" if the term after first one is > 0
                        out << "+";
                }
        }
}
```

Figure 2.7 Input and output Polynomial via overloading ‘>>’ and ‘<<’

```
const Polynomial& Polynomial::operator=(const Polynomial& a) {
	//assign polynomial a to *this
	if (poly.end()!=nullptr) {
		this->Polynomial::~Polynomial();
	}
	term temp;
	circularList<term>::chainIterator ai = a.poly.begin();
	while (1) {
		if (ai->exp == -1) return *this;
		poly.insertBack(temp.set(ai->coef, ai->exp));
		ai++;
	}
}
```

Figure 2.8 Assignment operator

```
Polynomial Polynomial::operator+(const Polynomial& b) const {
	//add *this (a) and b and the sum returned
	term temp;
	circularList<term>::chainIterator ai = poly.begin(),
		bi = b.poly.begin();
	Polynomial c;
	while (1) {
		if (ai->exp == bi->exp) {
			if (ai->exp == -1) return c;
			float sum = ai->coef + bi->coef;
			if (sum) c.poly.insertBack(temp.set(sum, ai->exp));
			ai++; bi++; //advance to next erm
		}
		else if (ai->exp < bi->exp) {
			c.poly.insertBack(temp.set(bi->coef, bi->exp));
			bi++; //next term of b
		}
		else {
			c.poly.insertBack(temp.set(ai->coef, ai->exp));
			ai++; //next term of a
		}
	}
}
```

Figure 2.9 Adding Polynomials

```
Polynomial Polynomial::operator-(const Polynomial& b) const{
	//subtract *this (a) and b and the sum returned
	term temp;
	circularList<term>::chainIterator ai = poly.begin(),
		bi = b.poly.begin();
	Polynomial c;
	while (1) {
		if (ai->exp == bi->exp) {
			if (ai->exp == -1) return c;
```

```
                float sum = ai->coef - bi->coef;
                if (sum!=0) c.poly.insertBack(temp.set(sum, ai->exp));
                ai++; bi++; //advance to next erm
            }
            else if (ai->exp < bi->exp) {
                //0 - bi->coef
                c.poly.insertBack(temp.set(bi->coef*-1, bi->exp));
                bi++; //next term of b
            }
            else {
                c.poly.insertBack(temp.set(ai->coef, ai->exp));
                ai++; //next term of a
            }
        }
}
```

Figure 2.10 Subtracting Polynomials

```
Polynomial Polynomial::operator*(const Polynomial& b) const {
        //multiply *this (a) and b
        term temp;
        circularList<term>::chainIterator ai = poly.begin(),
                bi = b.poly.begin(), ci;
        Polynomial c;
        while (1) {
                if (ai->exp == -1) return c;
                while (bi->exp != -1) {
                        bool added = false; //check if the term for c is added
                        float theCoeff = ai->coef * bi->coef;
                        int theExp = ai->exp + bi->exp;
                        if (theExp) {
                                //Run down the existing nodes of C to check if there
                                //is an element that has the same exponent as theCoeff
                                ci = c.poly.begin();
                                while (ci->exp != -1) {
                                        if (ci->exp == theExp) {
                                                ci->coef += theCoeff;
                                                added = true;
                                        }
                                        ci++;
                                }
                                //Insert the term if the current term isn't added
                                if (added == false)
                                        c.poly.insertBack(temp.set(theCoeff, theExp));
                        }
                        else {
                                c.poly.insertBack(temp.set(theCoeff, 0));
                        }
                        bi++;
                }
                bi = b.poly.begin();
                ai++;
        }
}
```

Figure 2.11 Multiplying Polynomials

```
float Polynomial::Eval(float x) const {
	circularList<term>::chainIterator e = poly.begin();
	float sum = 0;
	while (e->exp!=-1) {
		sum += e->coef * (float)pow(x, e->exp);
		e++;
	}
	return sum;
}
```

Figure 2.12 Evaluation of Polynomial

The main() tests out the functions implemented in Polynomial class

```
int main() {
	Polynomial A;
	Polynomial B;
	int m, n;
	cout << "Enter terms of A, B: ";
	cin >> m >> n;
	cout << "Enter coefficients, exponents of A:\n";
	for (int i = 0; i < m; i++) {
		cin >> A;
	}
	cout << "Enter coefficients, exponents of B:\n";
	for (int j = 0; j < n; j++) {
		cin >> B;
	}
	//Output A and B
	cout << "A = " << A << '\n';
	cout << "B = " << B << '\n';
	//Addition
	Polynomial C  = A + B;
	cout << "C = A + B = " << C << '\n';
	//Subtraction
	Polynomial D = A - B;
	cout << "D = A - B = " << D << '\n';
	//Multiplication
	Polynomial E = C; //Assign polynomial C to E
	E = E * B;
	cout << "E = C * B = " << E << '\n';
	//Evaluate the polynomial C = A+B with user input
	float ex;
	cout << "Input a number to evaluate C: ";
	cin >> ex;
	cout << "The result after evaluating C = " << C.Eval(ex) << '\n';

	system("pause");
	return 0;
}
```

Figure 2.13 The main section of LinkedPolynomial.cpp

效能分析

時間複雜度

istream: $O(n)$

Where n = number of terms

ostream: $O(n)$

n = number of terms

Copy Constructor: $O(n)$

n = number of nodes in linked list

Assignment Operator: $O(n)$

n = number of nodes in linked list

Destructor: $O(1)$

Since the available-space list is used, the operation takes $O(1)$ time

Addition: $O(m+n)$

m, n are number of terms in Polynomial

Subtraction: $O(m+n)$

m, n are number of terms in Polynomial

Multiplication: $O(m\times n)$

m, n are number of terms in Polynomial

Evaluation: $O(n)$

n = number of terms

空間複雜度

istream: $O(1)$

1 call to input a term

ostream: $O(n)$

1 linked list, n calls

Copy Constructor: $O(n)$

1 linked list, n calls to copy

Assignment Operator: $O(n)$

1 linked list, n calls to assign (copy)

Destructor: $O(1)$

1 call to the available-space list

Addition: $O(m+n)$

1 linked list (Polynomial C), m+n calls

Subtraction: $O(m+n)$

1 linked list, m+n calls

Multiplication: $O(m\times n)$

1 linked list, m*n calls

Evaluation: $O(1)$

1 space for sum variable

測試與過程

Sample Input

`Enter terms of A, B: 3 2`
`Enter coefficients, exponents of A:`
`2 3`
`3 1`
`1 0`
`Enter coefficients, exponents of B:`
`2 2`
`3 0`
`Input a number to evaluate C: 2`

Sample Output

`A = 2x^3+3x+1`
`B = 2x^2+3`
`C = A + B = 2x^3+2x^2+3x+4`
`D = A - B = 2x^3-2x^2+3x-2`
`E = C * B = 4x^5+12x^3+4x^4+14x^2+9x+12`
`The result after evaluating C is 34`

Table 1. Sample input and output

驗證

Addition: $(2x^3+3x+1) + (2x^2+3) = 2x^3+2x^2+3x+4$

Subtraction: $(2x^3+3x+1) - (2x^2+3) = 2x^3-2x^2+3x-2$

Multiplication: $(2x^3+2x^2+3x+4)(2x^2+3) = 4x^5+6x^3+4x^4+6x^2+6x^3+9x+8x^2+12$

$=4x^5+12x^3+4x^4+14x^2+9x+12$

Evaluation: $2 \times 2^3 + 2 \times 2^2 + 3 \times 2 + 4 = 16 + 8 + 6 + 4 = 34$