# HW3

41243117 吳承璿

1.7 2025

# 解題說明

1. 使用鏈結串列儲存多項式：
   - 每個多項式儲存在一個 Chain 資料結構中。
   - 每個節點 (ChainNode) 包含一個係數與次方數的 Term 物件。
   - 串列設計為循環鏈結串列，方便實現多項操作。
2. 多項式基本運算：
   - 加法 (operator+)：將兩個多項式的相同次方數合併，其餘依次插入結果。
   - 減法 (operator-)：類似加法，差別在於其中一方係數取反。
   - 乘法 (operator*)：兩多項式逐項相乘，使用線性哈希處理重複次方數的項。
   - 求值 (Evaluate)：使用 Horner's 法則逐項計算 f(x)f(x)f(x)。
3. 迭代器設計：

- 使用 ChainIterator 類別方便訪問鏈結串列中的每個節點。

4. **多載運算子：**

- >>：用於讀取多項式。
- <<：用於輸出多項式。

# 演算法與設計

```cpp
#include<iostream>
using namespace std;


template<class T>
class Chain;
template<class T>
class ChainNode
{
    friend class Chain<T>;
private:
    T data;
    ChainNode<T>* link;
public:
    ChainNode() {}
    ChainNode(const T& data)
    {
        this->data = data;
    }
    ChainNode(const T& data, ChainNode<T>* link)
    {
        this->data = data;//ChainNode 的資料
        this->link = link;//下一個node位址
    }
};

template<class T>
class Chain
{
public:
    Chain() { first = &head; first->link = &head; itemCount = 0; }
    ~Chain() {}
    bool IsEmpty()const { return first == head.link; }//判斷是否為空
    void sethead(const T& e) { head.data = e; head.link = &head; first = &head; }//設定第0項
    class ChainIterator //Chain 迭代器
    {
    public:
        friend class ChainNode<T>;
        ChainIterator(ChainNode<T>* startNode = 0) { current = startNode; };
        T& operator*() const { return current->data; }//多載*
        T* operator->() const { return &current->data; }//多載->
        ChainIterator& operator++();
        ChainIterator& operator++(int);
        bool operator!=(const ChainIterator right) const   //多載*
        {
            return current != right.current;
        }
        bool operator==(const ChainIterator right) const  //多載==
        {
            return current == right.current;
        }
        bool operator&&(const ChainIterator right)//多載&&
        {
            return current == nullptr && right.current == nullptr;
        }
    private:
        ChainNode<T>* current;
    };
    Chain<T>::ChainIterator begin() const;//回傳起始項位置
    Chain<T>::ChainIterator end() const;//回傳最後項位置
    void InsertBack(const T& e);//在最後插入一項
    int getCount()const { return itemCount; }//回傳項數
private:
    ChainNode<T>* first;
    ChainNode<T> head;
    int itemCount;
};
```

```cpp
template<class T>
typename Chain<T>::ChainIterator Chain<T>::begin() const
{
    return typename Chain<T>::ChainIterator(first->link);
}

template<class T>
typename Chain<T>::ChainIterator Chain<T>::end() const
{
    ChainNode<T>* p = first;
    while (p->link != first)//找最後一項
    {
        p = p->link;
    }
    return p->link;
}



template <class T>
typename Chain<T>::ChainIterator& Chain<T>::ChainIterator::operator++()
{
    current = current->link;
    return *this;
}

template <class T>
typename Chain<T>::ChainIterator& Chain<T>::ChainIterator::operator++(int)
{
    ChainIterator old = *this;
    current = current->link;
    return old;
}

template <class T>
void Chain<T>::InsertBack(const T& e)
{
    itemCount++;
    if (!IsEmpty())
    {
        ChainNode<T>* p = first;
        while (p->link != first)
        {
            p = p->link;
        }
        p->link = new ChainNode<T>(e, first);
    }
    else
        first->link = new ChainNode<T>(e, first);
}


class Polynomial;
class Term {
    friend Polynomial;
    friend ostream& operator<<(ostream& os, Polynomial& p);//多載輸出運算子
    friend istream& operator>>(istream& os, Polynomial& p);//多載輸入運算子
    Term Set(float c, int e) { coef = c; exp = e; return *this; };
private:
    float coef;
    int exp;
};
```

```
132  class Polynomial {
133  private:
134      Chain<Term> poly;
135  public:
136      Polynomial();
137      Polynomial(const Polynomial&);
138      ~Polynomial();
139      void newPoly(float c, int e);
140      Polynomial operator+(const Polynomial& b) const;//多載+
141      Polynomial operator-(const Polynomial& b) const;//多載-
142      Polynomial operator*(const Polynomial& b) const;//多載*
143      float Evaluate(float x)const;//求值
144      friend ostream& operator<<(ostream& os, Polynomial& p);//多載輸出運算子
145      friend istream& operator>>(istream& os, Polynomial& p);//多載輸入運算子
146  };
147
148  Polynomial::Polynomial()
149  {
150      Term tmp;
151      poly.sethead(tmp.Set(-1, -1));//將poly初始化
152  }
153  Polynomial::~Polynomial()
154  {
155  }
156  Polynomial::Polynomial(const Polynomial& a) :poly(a.poly) {}
157  void Polynomial::newPoly(float c, int e)
158  {
159      Term tmp;
160      poly.InsertBack(tmp.Set(c, e));
161  }
162
163  Polynomial Polynomial::operator+(const Polynomial& b) const
164  {
165      Term tmp;
166      Chain<Term>::ChainIterator ai = poly.begin(), bi = b.poly.begin();
167      Polynomial c;
168      while (1)//次方相同係數相加
169      {
170          if (ai->exp == bi->exp)
171          {
172              if (ai->exp == -1)return c;
173              float sum = ai->coef + bi->coef;
174              if (sum)
175              {
176                  c.poly.InsertBack(tmp.Set(sum, ai->exp));
177              }
178              ai++;
179              bi++;
180          }
181          else if (ai->exp < bi->exp)//poly1次方<poly2次方，poly2加入結果多項式中
182          {
183              c.poly.InsertBack(tmp.Set(bi->coef, bi->exp));
184              bi++;
185          }
186          else//poly1次方>poly2次方，poly1加入結果多項式中
187          {
188              c.poly.InsertBack(tmp.Set(ai->coef, ai->exp));
189              ai++;
190          }
191      }
192      return c;
193  }
```

```cpp
Polynomial Polynomial::operator-(const Polynomial& b) const
{
    Term tmp;
    Chain<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 + (-1 * bi->coef);
            if (sum)
            {
                c.poly.InsertBack(tmp.Set(sum, ai->exp));
            }
            ai++;
            bi++;
        }
        else if (ai->exp < bi->exp)//poly1次方<poly2次方，poly2加入結果多項式中
        {
            c.poly.InsertBack(tmp.Set(-bi->coef, bi->exp));
            bi++;
        }
        else//poly1次方>poly2次方，poly1加入結果多項式中
        {
            c.poly.InsertBack(tmp.Set(ai->coef, ai->exp));
            ai++;
        }
    }
    return c;
}
int hash_function(int x, int t, int* u)//計算線性哈希函數
{
    if (u[x % t] == 0 || u[x % t] - 1 == x)
    {
        u[x % t] = x + 1;
        return x % t;
    }
    else
    {
        for (int i = 1; i < t; i++)
        {
            if (u[(x + i) % t] == 0)
            {
                u[(x + i) % t] = x + 1;
                return (x + i) % t;
            }
        }
    }
}
```

```cpp
Polynomial Polynomial::operator*(const Polynomial& b) const
{
    Polynomial res;
    int use_cap = poly.getCount() * b.poly.getCount(); //已用過的次方旗標陣列大小
    bool* use = new bool[use_cap]();//用於檢查是否有重複次方數的項
    int* hash = new int[use_cap]();//紀錄哈希表中對應的值
    Chain<Term>::ChainIterator ai = poly.begin(), bi = b.poly.begin();
    for (int i = 0; i < poly.getCount(); i++)//每項一一相乘
    {
        bi = b.poly.begin();
        for (int j = 0; j < b.poly.getCount(); j++)
        {
            int mult_exp = ai->exp + bi->exp;//相乘後的次方
            if (use[hash_function(mult_exp, use_cap, hash)])//已用過，將相乘完的項使用加法功能加入多項式
            {
                Term tmp;
                tmp.coef = ai->coef * bi->coef;
                tmp.exp = mult_exp;
                Polynomial temp;
                temp.poly.InsertBack(tmp);
                res = res + temp;
            }
            else//未用過，將相乘完的項加入多項式
            {
                Term tmp;
                tmp.coef = ai->coef * bi->coef;
                tmp.exp = mult_exp;
                res.poly.InsertBack(tmp);
                use[mult_exp] = true;
            }
            bi++;
        }
        ai++;
    }
    return res;
}

float Polynomial::Evaluate(float x)const
{
    Chain<Term>::ChainIterator ai = poly.begin();
    float total = 0;
    while (ai != poly.end())
    {
        total += ai->coef * powf(x, ai->exp);
        ai++;
    }
    return total;
}
ostream& operator<<(ostream& os, Polynomial& p) {//多載輸出運算子
    Chain<Term>::ChainIterator o = p.poly.begin();
    while (o != p.poly.end())
    {
        if (o->exp == 0)
            os << abs(o->coef);
        else if (o->exp == 1)
            os << abs(o->coef) << "X";
        else
            os << abs(o->coef) << "X^" << o->exp;
        if (++o != p.poly.end())
            if (o->coef < 0)
                os << " - ";
            else
                os << " + ";
    }
    return os;
}
```

```cpp
istream& operator>>(istream& os, Polynomial& p) {//多載輸入運算子
    float coeftmp;//暫存coef
    int exptmp;//暫存exp
    os.ignore();//忽略(
    bool next = 0;
    while (1)
    {
        os >> coeftmp;//讀入係數
        os.ignore(2);//忽略 X^
        os >> exptmp;//讀入次方
        if (next)
            p.newPoly(-coeftmp, exptmp);
        else
            p.newPoly(coeftmp, exptmp);
        char c;
        os >> c;
        if (c == ')')break;//讀到括號尾跳出
        else if (c == '-')//判斷下一項是否為負
        {
            next = 1;
        }
    }
    os.get();//吃掉換行
    return os;
}

int main() {
    Polynomial poly1;
    Polynomial poly2;
    Polynomial poly3;
    cout << "多項式1:";
    cin >> poly1;
    cout << "多項式2:";
    cin >> poly2;

    poly3 = (poly1 + poly2);
    cout << "(" << poly1 << ") + (" << poly2 << ") = ";
    cout << poly3 << endl;

    cout << "(" << poly1 << ") - (" << poly2 << ") = ";
    poly3 = (poly1 - poly2);
    cout << poly3 << endl;

    cout << "(" << poly1 << ") * (" << poly2 << ") = ";
    poly3 = (poly1 * poly2);
    cout << poly3 << endl;

    cout << "Evaluate(2):";
    cout << poly3.Evaluate(2) << endl;

    cout << "poly1=poly2:" << endl;
    poly1 = poly2;
    cout << poly1 << endl << poly2;
    return 0;
}
```

# 效能分析

**1. 空間複雜度：**

多項式以鏈結串列儲存：

每個節點存儲一個 Term（包含兩個值：float coef 和 int exp）。

節點還有一個指向下一節點的指標（鏈結）。

空間複雜度 O(n)O(n)O(n)，其中 nnn 是多項式項數。

運算時使用臨時資料結構，如哈希表（線性陣列）和布林陣列，這些額外空間依賴於多項式項數的乘積，為 O(n⋅m)O(n \cdot m)O(n⋅m)，其中 mmm 和 nnn 分別是兩多項式的項數。

**2. 時間複雜度：**

- **加法與減法 (operator+, operator-)：**

  遍歷兩個多項式串列進行合併，時間複雜度為 O(n+m)O(n + m)O(n+m)。

- **乘法 (operator*)：**

  需要對兩個多項式的每一項進行相乘操作，時間複雜度為 O(n⋅m)O(n \cdot m)O(n⋅m)。

線性哈希檢查重複次方數會增加額外的 O(k)O(k)O(k) 時間，其中 kkk 是哈希表大小（通常為 n⋅mn \cdot mn⋅m）。

整體複雜度約為 O(n⋅m+n⋅m)=O(n⋅m)O(n \cdot m + n \cdot m) = O(n \cdot m)O(n⋅m+n⋅m)=O(n⋅m)。

- **求值 (Evaluate)：**

  計算多項式值的時間複雜度為 O(n)O(n)O(n)，其中 nnn 是多項式項數。

- **輸出與輸入 (<<, >>)：**

  時間複雜度為 O(n)O(n)O(n)（輸出每一項需要遍歷鏈結串列）。