队列的应用：多项式类

一元多项式

$$p_n(x) = p_n x^n + p_{n-1} x^{n-1} + \ldots + p_2 x^2 + p_1 x + p_0$$

该多项式可表示成一个线性表：

$P = (p_n, p_{n-1}, \ldots, p_0)$

但如果是这样的多项式呢？该如何表示？

$$\mathbf{S(x) = -2x^{20000} + 3x^{10000} + 1}$$

一元多项式

任意系数的多项式:

$$P_n(x) = p_m x^{em} + \ldots\ \ p_i x^{ei} + \ \ldots + p_1 x^{e1} + p_0 x^{e0}$$

其中：p_i 是指数为e_i 的项的非零系数，

$$0 \leq e_0 < e_1 < \ \ldots < e_m = n$$

可以下列线性表表示：

（$(p_m,e_m) \ldots$（p_1, e_1），(p_0, e_0)）

一元多项式示例

$$P_{999}(x) = -8x^{999} - 2x^{12} + 7x^{3}$$

可以用线性结构表示:

((-8, 999), (-2, 12), (7, 3))

一元多项式的数据结构

- 多项式中的一项(term)：系数+幂指数
- 多项式：若干项(term)构成
  - 多项式可表示成由若干项(term)构成的有序线性表

一元多项式的数据结构

```
struct Term {
    int degree;
    double coefficient;
    Term (int exponent = 0, double scalar = 0);
};
Term :: Term(int exponent, double scalar)
/* Post: The Term is initialized with the given coefficient and
exponent, or with default parameter values of 0. */
{
    degree = exponent;
    coefficient = scalar;
}
```

一元多项式的数据结构

- 多项式的数据结构：一个栈？队列？或者普通线性表？
- 多项式加法

$$A(x) = 5x^{17} + 9x^{8} + 3x + 7$$

$$B(x) = -9x^{8} + 22x^{7} + 8x$$

$$C(x) = A(x) + B(x)$$
$$= 5x^{17} + 22x^{7} + 11x + 7$$

➢ 上述例子可以看到：从多项式A和B对应列表的头部移出每一项，运算得到的结果依次追加到列表C中

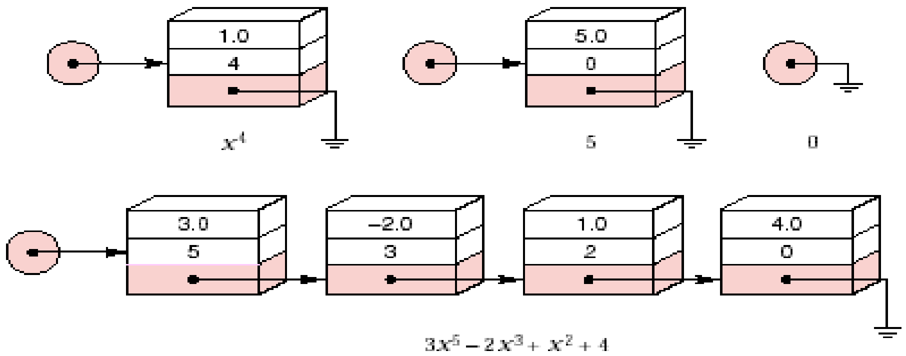
➢ 在一端进行删除，另一端进行插入——FIFO

➢ 多项式适合用队列，准确地说是扩展的队列进行描述

一元多项式的数据结构

- 多项式实现时，选择顺序队列，还是链式队列？

  事先不知道多项式的长度，多项式系数不连续，建议采用链式结构。



- 结点表示多项式的一项，由系数和幂指数构成，指针域指向下一项；
- 多项式中各项是按照幂指数的降序排列，不存在幂指数相同的多项；
- 系数为零的多项式项无需存储；
- 如果多项式就是数值0，可用空队表示。

一元多项式

```
class Polynomial: private Extended_queue { // Use private inheritance.
public:
    void read( );
    void print( ) const;
    void equals_sum(Polynomial p, Polynomial q);
    void equals_difference(Polynomial p, Polynomial q);
    void equals_product(Polynomial p, Polynomial q);
    Error_code equals_quotient(Polynomial p, Polynomial q);
    int degree( ) const;//最高次项的指数
private:
    void mult_term(Polynomial p, Term t);
};
```

一元多项式——输出

```
void Polynomial :: print( ) const
/* Post: The Polynomial is printed to cout. */   {
Node *print_node = front;
bool first_term = true;
while (print_node != NULL) {
      Term &print_term = print_node->entry;
      if (print_term.coefficient < 0) cout << "- ";
      if (first_term)  // In this case, suppress printing an initial '+'.
                  first_term = false;
      else
                  if (print_term.coefficient >= 0)  cout << " + ";
```

-9x^5 + x^4 -2x^2 + 2x+1

9x^5 + x^4 -2x^2 + 2x+4

養天地正氣 法古今完人

一元多项式——输出

```
double r = (print_term.coefficient >= 0)// r is abs of the coefficient
      ? print_term.coefficient : -(print_term.coefficient);
if (r != 1) cout << r;
if (print_term.degree > 1) cout << " X^" << print_term.degree;
if (print_term.degree == 1) cout << " X";
if (r == 1 && print_term.degree == 0) cout << " 1";
print_node = print_node->next;
}
if (first_term)
      cout << "0"; // Print 0 for an empty Polynomial.
cout << endl;
}
```

-9x^5 + x^4 -2x^2 + 2x+1

9x^5 + x^4 -2x^2 + 2x+4

一元多项式——输入

```
void Polynomial :: read( ) /* Post: The Polynomial is read from cin. */
{
clear( );
double coefficient;
int last_exponent, exponent;
bool first_term = true;
cout << "Enter the coefficients and exponents for the polynomial, "
<< "one pair per line. Exponents must be in descending order." << endl
<< "Enter a coefficient of 0 or an exponent of 0 to terminate." << endl;
do {
      cout << "coefficient? " << flush;
      cin >> coefficient;
      if (coefficient != 0.0) {
                  cout << "exponent? " << flush;
                  cin >> exponent;
```

一元多项式——输入

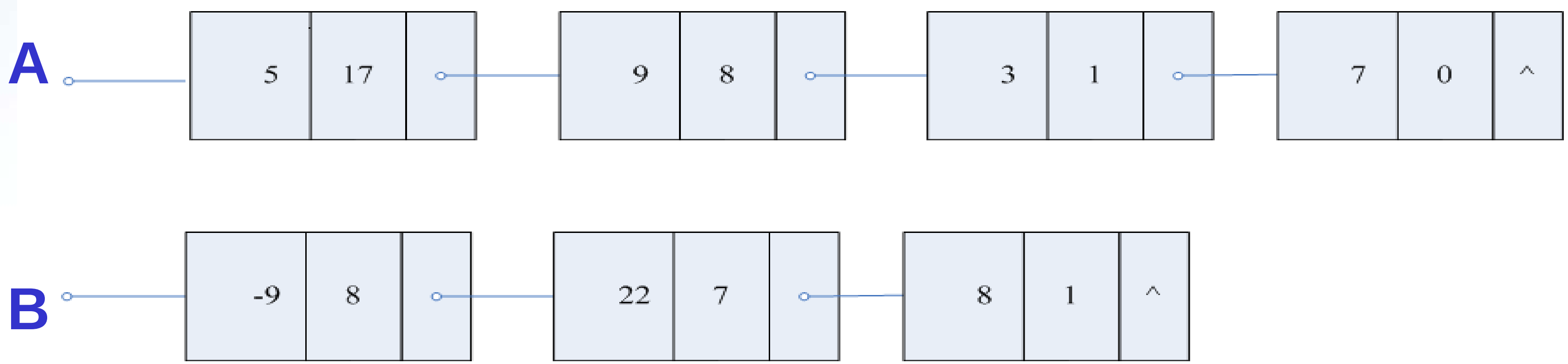
```
    if ((!first_term && exponent >= last_exponent) || exponent < 0) {
        exponent = 0;
        cout << "Bad exponent: Polynomial terminates without its last
        term."<< endl;//the input is invalid!
    }
  else {
          Term new_term(exponent, coefficient);
          append(new_term);
          first_term = false;
  }
  last_exponent = exponent;
  }
} while (coefficient != 0.0 && exponent != 0);
}
```

一元多项式——加法

$A(x) = 5x^{17} + 9x^{8} + 3x + 7$

$B(x) = -9x^{8} + 22x^{7} + 8x$

$C(x) = A(x) + B(x) = 5x^{17} + 22x^{7} + 11x + 7$



一元多项式——加法

```
void Polynomial :: equals_sum(Polynomial p, Polynomial q)
/* Post: The Polynomial object is reset as the sum of the two
parameters. */
{
clear( );
while (!p.empty( ) || !q.empty( )) {
Term p_term, q_term;
if (p.degree( ) > q.degree( )) {
        p.serve_and_retrieve(p_term);
        append(p_term);
        }
```

一元多项式——加法

```
else if (q.degree( ) > p.degree( )) {
        q.serve_and_retrieve(q_term);
        append(q_term);
}
else {
        p.serve_and_retrieve(p_term);
        q.serve_and_retrieve(q_term);
if (p_term.coefficient + q_term.coefficient != 0) {
        Term answer_term(p_term.degree,
        p_term.coefficient + q_term.coefficient);
        append(answer_term);
}
    }
   }
}
```

一元多项式——加法

- determine degree

```
int Polynomial :: degree( ) const
/* Post: If the Polynomial is identically 0, a result of -1 is returned.
Otherwise the degree of the Polynomial is returned. */
{
    if (empty( )) return -1;
    Term lead;
    retrieve(lead);
    return lead.degree;
}
```

一元多项式——乘法

```
void Polynomial::equals_product(Polynomial p,Polynomial q){
    clear();
    Polynomial temp;
    Term p_term;
    while (!p.empty()){
        p.serve_and_retrieve(p_term);
        temp.mult_term(q,p_term);
        equal_sum(*this,temp);
    }
}
```

多项式与单项相乘

```
void Polynomial::mult_term(Polynomial p,Term t) {
        Term p_term;
        clear();
        while(!p.empty()){
                p.serve_and_retrieve(p_term);
                append(Term(p_term.degree+t.degree,
                            p_term.coefficient*t.coefficient));
        }
}
```