Chương 15 – ỨNG DỤNG CỦA HÀNG ĐỢI

CTDL hàng đợi được sử dụng rất rộng rãi trong các chương trình máy tính. Đặc biệt là trong công việc của hệ điều hành khi cần xử lý các công việc một cách tuần tự. Hàng đợi trong chương 3 là một khái niệm FIFO đơn giản. Trong thực tế, người ta thường rất hay sử dụng các hàng đợi ưu tiên được trình bày trong chương 11. Chương này minh họa một số ứng dụng đơn giản của hàng đợi.

15.1. Các dịch vụ

Chúng ta có thể viết một chương trình mô phỏng việc cung cấp các dịch vụ. Chẳng hạn tại quầy bán vé các tuyến bay, có nhiều người đang đến và đang sắp hàng chờ để mua vé. Có khả năng chỉ có một nhân viên bán vé, hoặc có nhiều nhân viên bán vé đồng thời. Sinh viên hãy xem đây như là một gợi ý để viết thành một ứng dụng cho CTDL hàng đợi. Những điều thường được quan tâm là:

- Thời gian chờ đợi trung bình (*queue time*) của một khách hàng từ lúc đến cho đến lúc được bắt đầu được phục vụ.
- Thời gian phục vụ trung bình (*service time*) mà một dịch vụ được thực hiện.
- Thời gian đáp ứng trung bình (*response time*) của một khách hàng từ lúc đến cho đến lúc rời khỏi quầy (chính bằng tổng hai thời gian trên).
- Tần suất đến của khách hàng.

Dựa vào những điều trên người ta có thể điều chỉnh các kế hoạch phục vụ cho thích hợp hơn.

15.2. Phân loại

Một ví dụ về phân loại là việc sử dụng nhiều hàng đợi cùng một lúc. Tùy theo đặc điểm của các yêu cầu công việc, mỗi hàng đợi chỉ nhận các công việc cùng đặc điểm mà thôi. Như vậy đầu ra của mỗi hàng đợi sẽ là những công việc có chung đặc điểm. Việc sử dụng hàng đợi ở đây giúp ta phân loại được công việc, đồng thời trong các công việc cùng loại, chúng vẫn giữ nguyên thứ tự ban đầu giữa chúng. Hình ảnh dễ thấy nhất chính là ví dụ trên với mỗi hàng người đợi sẽ được mua vé đi cùng một tuyến bay nào đó (một ô cửa chỉ bán cho một tuyến bay nhất định). Một ví dụ khác về sự phân loại các bưu kiện tại một trung tâm chuyển phát: các bưu kiện sẽ được phân loại vào các hàng đợi dựa vào thể tích, trọng lượng, nơi đến,...., mà thứ tự giữa chúng trong một hàng đợi là không thay đổi.

15.3. Phương pháp sắp thứ tự Radix Sort

Ứng dụng hàng liên kết trong phương pháp sắp thứ tự Radix Sort được trình bày trong chương 8.

15.4. Tính trị cho biểu thức *prefix*

Để tính trị cho biểu thức dạng *prefix* người ta dùng hàng đợi. Việc tính được thực hiện lặp lại nhiều lần, mỗi lần luôn xử lý cho biểu thức từ trái sang phải như dưới đây:

- + * 9 **+ 2 8** * **+ 4 8** 6 3

- + *** 9 10** *** 12 6** 3

- **+ 90 72** 3

- 162 3

159

Mỗi lần duyệt biểu thức, chúng ta thay thế mọi toán tử mà có đủ hai toán hạng đứng ngay sau bằng kết quả của phép tính cho toán tử đó. Do thứ tự duyệt qua biểu thức luôn là từ trái sang phải, nên chúng ta có thể lưu biểu thức vào hàng đợi và sử dụng các phương thức của hàng đợi để lấy từng phần tử cũng như đưa từng phần tử vào hàng. Hiện thực chương trình được xem như bài tập cho sinh viên.

15.5. Ứng dụng phép tính trên đa thức

Đây là một ứng dụng có sử dụng CTDL ngăn xếp và hàng đợi. Trong ứng dụng này chúng ta có dịp nhìn thấy công dụng của các CTDL đã được thiết kế hoàn chỉnh. Có nhiều bài toán có thể sử dụng các CTDL hoàn chỉnh để phát triển thêm các lớp thừa kế rất tiện lợi. Ngoài ra, phương pháp phát triển dần thành một chương trình hoàn chỉnh được trình bày dưới đây cũng là một tham khảo rất tốt cho sinh viên khi làm quen với các kỹ năng thiết kế và lập trình.

15.5.1. Mục đích của ứng dụng

Trong phần 14.3 chúng ta đã viết chương trình mô phỏng một máy tính đơn giản thực hiện các phép cộng, trừ, nhân, chia và một số phép tính khác tương tự. Phần này sẽ mô phỏng một máy tính tương tự thực hiện cách tính Balan ngược cho các phép tính trên đa thức. Ý tưởng chính của giải thuật là sử dụng ngăn xếp để lưu các toán hạng. Còn hàng đợi sẽ được sử dụng để mô phỏng đa thức.

15.5.2. Chương trình

Chúng ta sẽ hiện thực một lớp đa thức (`Polynomial`) để sử dụng trong chương trình. Việc hiện thực chương trình sẽ trở nên rất đơn giản. Chương trình chính

cần khai báo một ngăn xếp để chứa các đa thức, nhận các yêu cầu tính và thực hiện.

```
int main()
/*
post: Chương trình thực hiện các phép tính toán số học cho các đa thức do người sử dụng nhập
      vào.
uses: Các lớp Stack, Polynomial và các hàm introduction, instructions,
      do_command, get_command.
*/
{
   Stack<Polynomial> stored_polynomials;
   introduction();
   instructions();
   while (do_command(get_command(), stored_polynomials));
}
```

Chương trình này hầu như tương tự với chương trình chính ở phần 14.3, hàm phụ trợ `get_command` tương tự hàm đã có.

15.5.2.1. Các phương thức của lớp `Polynomial`

Tương tự phần 14.3, thay vì nhập một con số, dấu ? chờ người sử dụng nhập vào một đa thức.

Phần lớn các việc cần làm trong chương trình này là việc hiện thực các phương thức của `Polynomial`. Chẳng hạn chúng ta cần phương thức `equals_sum` để cộng hai đa thức. Cho p, q, r là các đối tượng đa thức, `p.equals_sum(q,r)` sẽ thay p bởi đa thức tổng của hai đa thức q và r. Tương tự chúng ta có các phương thức `equals_difference`, `equals_product`, và `equals_quotient` để thực hiện các phép tính số học khác trên đa thức.

Ngoài ra, lớp `Polynomial` còn hai phương thức không thông số là `print` và `read` để xuất và đọc đa thức.

15.5.2.2. Thực hiện các yêu cầu

Hàm `do_command` nhận đối tượng ngăn xếp là tham biến do ngăn xếp cần được biến đổi trong hàm.

```
bool do_command(char command, Stack<Polynomial> &stored_polynomials)
/*
pre:  command chứa ký tự hợp lệ biểu diễn yêu cầu của người sử dụng.
post: Yêu cầu của người sử dụng được thực hiện đối với các đa thức trong ngăn xếp, hàm trả về
      true ngoại trừ trường hợp command='q' thì hàm trả về false.
uses: Các lớp Stack và Polynomial.
*/
```

```
{
   Polynomial p, q, r;
   switch (command) {

   case '?':
      p.read();
      if (stored_polynomials.push(p) == overflow)
         cout << "Warning: Stack full, lost polynomial" << endl;
      break;

   case '=':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         p.print();
      }
      break;

   case '+':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         stored_polynomials.pop();
         if (stored_polynomials.empty()) {
            cout << "Stack has just one polynomial" << endl;
            stored_polynomials.push(p);
         }
         else {
            stored_polynomials.top(q);
            stored_polynomials.pop();
            r.equals_sum(q, p);
            if (stored_polynomials.push(r) == overflow)
               cout << "Warning: Stack full, lost polynomial" << endl;
         }
      }
      break;
   //    Cần bổ sung các dòng lệnh để thực hiện các phép tính còn lại.

    case 'q':
      cout << "Calculation finished." << endl;
      return false;
   }
   return true;
}
```

15.5.2.3. Các mẩu chương trình và công việc kiểm tra

Cách làm sau đây minh họa ý tưởng sử dụng các mẩu tạm trong chương trình như đã trình bày trong phần 1.5.3 (các kỹ năng lập trình).

Hiện tại chúng ta chỉ phát triển chương trình vừa đủ để có thể dịch, chỉnh sửa lỗi, và kiểm tra tính đúng đắn của những phần đã viết.

Để dịch thử chương trình, chúng ta cần tạo các mẩu cho mọi phần tử còn thiếu của chương trình. Phần tử còn thiếu ở đây là lớp `Polynomial`. Do chúng ta còn chưa quyết định sẽ hiện thực các đối tượng đa thức như thế nào, chúng ta hãy chạy chương trình như một máy tính Balan ngược dành cho các số thực. Thay vào chỗ cần khai báo dữ liệu cho lớp `Polynomial`, chúng ta khai báo số thực.

```
class Polynomial {
public:
   void read();
   void print();
   void equals_sum(Polynomial p, Polynomial q);
   void equals_difference(Polynomial p, Polynomial q);
   void equals_product(Polynomial p, Polynomial q);
   ErrorCode equals_quotient(Polynomial p, Polynomial q);
private:
   double value; // mẩu tạm dùng để thử chương trình.
};
```

Phương thức `equals_quotient` cần kiểm tra phép chia 0 nên cần trả về `ErrorCode`. Hàm sau đây là một điển hình cho mẩu phương thức dùng để thử chương trình.

```
void Polynomial::equals_sum(Polynomial p, Polynomial q)
{
   value = p.value + q.value; // Chỉ viết tạm, sau sẽ viết lại đúng cho đa thức.
}
```

Việc tạo ra một bộ khung chương trình tại thời điểm này cho phép chúng ta kiểm tra xem ngăn xếp và các gói tiện ích (chứa các hàm phụ trợ) đã được kết nối một cách đúng đắn trong chương trình hay chưa. Chương trình, cùng các mẩu thử của nó, phải thực thi một cách chính xác cả với hiện thực ngăn xếp liên tục lẫn ngăn xếp liên kết.

15.5.3. Cấu trúc dữ liệu của đa thức

Chúng ta hãy quay lại nhiệm vụ chính của chúng ta là chọn lựa cách biểu diễn đa thức và viết các phương thức xử lý cho chúng. Các đa thức có dạng sau

$$3x^5 - 2x^3 + x^2 + 4$$

Thông tin quan trọng trong một đa thức là các hệ số và các số mũ của x, còn bản thân x chỉ là một biến. Chúng ta xem đa thức được tạo thành từ các số hạng (`term`), mỗi số hạng chứa một hệ số và một số mũ. Trong máy tính có thể xem đa thức là một danh sách các cặp gồm hệ số và số mũ. Chúng ta dùng `struct` để khai báo số hạng

```
struct Term {
   int degree;
   double coefficient;
   Term (int exponent = 0, double scalar = 0);
};
```

```
Term::Term(int exponent, double scalar)
/*
post: Term được khởi tạo bởi hệ số và số mũ nhận được, nếu không có thông số truyền vào thì
      hai số này được gán mặc định là 0.
*/
{
   degree = exponent;
   coefficient = scalar;
}
```

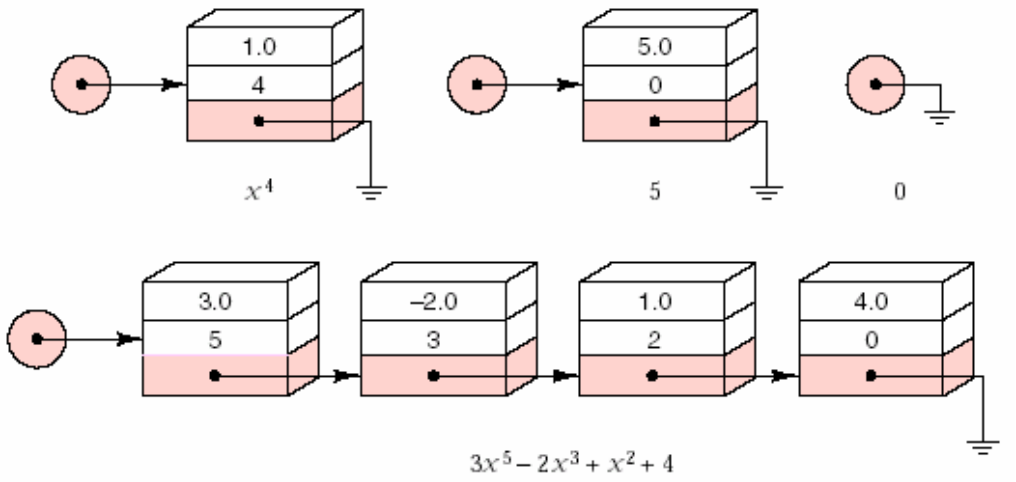
Chúng ta chưa lưu tâm về thứ tự của các số hạng trong đa thức, tuy nhiên nếu cho phép các số hạng có một thứ tự tùy ý thì chúng ta sẽ gặp khó khăn trong việc nhận ra các đa thức bằng nhau cũng như trong việc tính toán trên các đa thức. Chúng ta chọn phương án buộc các số hạng trong một đa thức phải có thứ tự giảm dần theo số mũ, đồng thời cũng không cho phép hai số hạng có cùng số mũ hoặc số hạng có hệ số bằng không. Với sự ràng buộc này, khi thực hiện một phép tính số học nào đó trên hai đa thức, chúng ta thường lần lượt xử lý cho từng cặp số hạng của hai đa thức từ trái sang phải. Các số hạng của đa thức kết quả cũng được ghi vào đa thức theo thứ tự này. Một đa thức được biểu diễn bằng một danh sách các `Term`. Như vậy, các phép tính số học có thể xem đa thức như là một `Queue`, hay chính xác hơn, như là `Extended_queue`, vì chúng ta thường xuyên cần đến các phương thức `clear` và `serve_and_retrieve`.

Chúng ta thử hỏi nên dùng hàng liên tục hay hàng liên kết. Nếu biết trước bậc của đa thức và các đa thức ít có hệ số bằng không thì chúng ta nên dùng hàng liên tục. Ngược lại, nếu không biết trước bậc, hoặc đa thức có rất ít hệ số khác không thì hàng liên kết thích hợp hơn. Hình 15.1 minh họa đa thức hiện thực bằng hàng liên kết.

Mỗi phần tử trong hàng chứa một số hạng của đa thức và chỉ có những số hạng có hệ số khác không mới được lưu vào hàng. Đa thức bằng 0 biểu diễn bởi hàng rỗng.

Với cấu trúc dữ liệu được chọn cho đa thức như trên chúng ta xây dựng lớp `Polynomial` là lớp dẫn xuất từ lớp `Extended_Queue`, chúng ta chỉ việc bổ sung các phương thức riêng của đa thức.

Để hiện thực cụ thể cho lớp dẫn xuất `Polynomial`, chúng ta cần đặt câu hỏi: một đa thức có hẳn là một `Extended_Queue` hay không?



Hình 15.1- Biểu diễn đa thức bởi một hàng liên kết các số hạng

Một `Extended_Queue` cung cấp các phương thức như `serve` chẳng hạn, phương thức này không nhất thiết và cũng không nên là phương thức của `Polynomial`. Sẽ là một điều không hay khi chúng ta cho phép người sử dụng gọi phương thức `serve` để loại đi một phần tử của `Polynomial`. Đa thức nên là một đối tượng đóng kín. Vì vậy một `Polynomial` không hẳn là một `Extended_Queue`. Mặc dù rất có lợi khi sử dụng lại các thuộc tính và các phương thức từ `Extended_Queue` cho `Polynomial`, nhưng chúng ta không thể sử dụng phép thừa kế đơn giản, vì quan hệ của hai lớp này không phải là quan hệ "*is-a*".

Quan hệ "*is-a*" là dạng thừa kế `public` trong C++. Nếu khai báo thừa kế `public` thì người sử dụng có thể xem một đa thức cũng là một hàng đợi. C++ cung cấp một dạng thừa kế thứ hai, gọi là thừa kế riêng (*private inheritance*), đây chính là điều chúng ta mong muốn. Sự thừa kế riêng cho phép lớp dẫn xuất được hiện thực bằng cách sử dụng lại lớp cơ sở, nhưng người sử dụng không gọi được các phương thức của lớp cơ sở thông qua đối tượng của lớp dẫn xuất. Quan hệ này còn được gọi là quan hệ "*is implemented of*". Chúng ta sẽ định nghĩa lớp `Polynomial` thừa kế riêng từ lớp `Extended_Queue`: các thuộc tính và các phương thức của `Extended_Queue` có thể được sử dụng trong hiện thực của lớp `Polynomial`, nhưng chúng không được nhìn thấy bởi người sử dụng.

```
class Polynomial: private Extended_queue<Term> {  //  Thừa kế riêng.

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;
private:
   void mult_term(Polynomial p, Term t);
};
```

Định nghĩa trên bổ sung thêm các phương thức như `degree` trả về bậc của đa thức và `mult_term` để nhân một đa thức với một `term`.

15.5.4. Đọc và ghi các đa thức

Việc in đa thức đơn giản chỉ là duyệt qua các phần tử của hàng và in dữ liệu trong mỗi phần tử. Phương thức dưới đây in đa thức với một số xử lý cho các trường hợp đặc biệt như loại bỏ dấu + (nếu có) ở đầu đa thức, không in hệ số hoặc số mũ nếu bằng 1 và không in x^0.

```
void Polynomial::print() const
/*
post: Đa thức được in ra cout.
*/
{
   Node *print_node = front;
   bool first_term = true;

   while (print_node != NULL) {
      Term &print_term = print_node->entry;
      if (first_term) {                    // Không in dấu '+' đầu đa thức.
         first_term = false;
         if (print_term.coefficient < 0) cout << "- ";
      }
      else if (print_term.coefficient < 0) cout << " - ";
      else cout << " + ";
      double r = (print_term.coefficient >= 0)
                 ? 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;
}
```

Phương thức đọc đa thức thực hiện việc kiểm tra để các số hạng nhập vào thỏa điều kiện số mũ giảm dần và một số ràng buộc trong việc biểu diễn đa thức như đã nêu trên. Việc nhập kết thúc khi hệ số và số mũ đều nhận trị 0.

```
void Polynomial::read()
/*
post: Đa thức được đọc từ 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;
         if ((!first_term && exponent >= last_exponent) || exponent < 0) {
            exponent = 0;

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

15.5.5. Phép cộng đa thức

Phần này chúng ta chỉ hiện thực phép cộng đa thức, các phép tính khác xem như bài tập.

Do các số hạng trong cả hai đa thức có số mũ giảm dần nên phép cộng rất đơn giản. Chúng ta chỉ cần duyệt qua một lần và đồng thời đối với cả hai đa thức. Nếu gặp hai số hạng của hai đa thức có số mũ bằng nhau, chúng ta cộng hai hệ số và kết quả đưa vào đa thức tổng, ngược lại, số hạng của đa thức nào có số mũ cao hơn được đưa vào tổng, phép duyệt chỉ dịch chuyển tới một bước trên đa thức này. Chúng ta chỉ phải lưu ý đến trường hợp tổng hai hệ số của hai số hạng bằng 0 thì sẽ không có phần tử mới được đưa vào đa thức tổng. Ngoài ra, vì phương thức này sẽ phá hủy các đa thức toán hạng được đưa vào nên chúng được gởi bằng tham trị.

```
void Polynomial::equals_sum(Polynomial p, Polynomial q)
/*
post: Đối tượng đa thức sẽ có trị bằng tổng hai đa thức nhận vào từ thông số.
*/
{
   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);
      }
```

```
      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);
         }
      }
   }
}
```

Phương thức trên bắt đầu bằng việc dọn dẹp mọi số hạng trong đa thức sẽ chứa kết quả của phép cộng. Phương thức `degree` được gọi để trả về bậc của đa thức, nếu đa thức rỗng, `degree` sẽ trả về -1.

15.5.6. Hoàn tất chương trình

15.5.6.1. Các phương thức còn thiếu

Phép trừ hai đa thức hoàn toàn tương tự phép cộng. Đối với phép nhân, trước hết chúng ta phải viết hàm nhân một đa thức với một số hạng, sau đó kết hợp với phương thức cộng hai đa thức để hoàn tất phép nhân. Phép chia đa thức phức tạp hơn rất nhiều, phép chia đa thức cho một kết quả là đa thức thương và một là đa thức số dư.