CHƯƠNG 4

DANH SÁCH

Danh sách là cấu trúc dữ liệu tuyến tính, danh sách được tạo nên từ các phần tử dữ liệu được sắp xếp theo một thứ tự xác định. Danh sách là một trong các cấu trúc dữ liệu cơ bản được sử dụng thường xuyên nhất trong các thuật toán. Danh sách còn được sử dụng để cài đặt nhiều KDLTT khác. Trong chương này, chúng ta sẽ xác định KDLTT danh sách và nghiên cứu phương pháp cài đặt KDLTT danh sách bởi mảng. Sau đó, chúng ta sẽ sử dụng danh sách để cài đặt KDLTT tập động.

4.1 KIỂU DỮ LIỆU TRỪU TƯỢNG DANH SÁCH

Danh sách là một khái niệm được sử dụng thường xuyên trong thực tiễn. Chẳng hạn, chúng ta thường nói đến danh sách sinh viên của một lớp, danh sách các số điện thoại, danh sách thí sinh trúng tuyển, …

Danh sách được định nghĩa là một dãy hữu hạn các phần tử:

$L = (a_1, a_2, \dots , a_n)$

trong đó a_i (i = 1, …, n) là phần tử thứ i của danh sách. Cần lưu ý rằng, số phần tử của danh sách, được gọi là **độ dài** của danh sách, có thể thay đổi theo thời gian. Và một phần tử dữ liệu có thể xuất hiện nhiều lần trong danh sách ở các vị trí khác nhau. Chẳng hạn, trong danh sách các số nguyên sau:

L = (3, 5, 5, 0, 7, 5)

số nguyên 5 xuất hiện 3 lần ở các vị trí 2, 3, và 6. Một đặc điểm quan trọng khác của danh sách là các phần tử của nó có thứ tự tuyến tính, thứ tự này được xác định bởi vị trí của các phần tử trong danh sách. Khi độ dài của danh sách bằng không (n = 0), ta nói danh sách rỗng. Nếu danh sách không rỗng ($n \geq 1$), thì phần tử đầu tiên a_1 được gọi là **đầu** của danh sách, còn phần tử cuối cùng a_n được gọi là **đuôi** của danh sách.

Không có hạn chế nào trên kiểu dữ liệu của các phần tử trong danh sách. Khi mà tất cả các phần tử của danh sách cùng một kiểu, ta nói danh sách là danh sách thuần nhất. Trong trường hợp tổng quát, một danh sách có thể chứa các phần tử có kiểu khác nhau, đặc biệt một phần tử của danh sách có thể lại là một danh sách. Chẳng hạn

L = (An, (20, 7, 1985), 8321067)

Trong danh sách này, phần tử đầu tiên là một xâu ký tự, phần tử thứ hai là danh sách các số nguyên, phần tử thứ ba là số nguyên. Danh sách này có thể sử dụng để biểu diễn, chẳng hạn, một sinh viên có tên là An, sinh ngày 20/7/1985, có số điện thoại 8321067. Danh sách (tổng quát) là cấu trúc dữ liệu cơ bản trong các ngôn ngữ lập trình chuyên dụng cho các xử lý dữ liệu phi số, chẳng hạn Prolog, Lisp. Trong sách này, chúng ta chỉ quan tâm tới các danh sách thuần nhất, tức là khi nói đến danh sách thì cần được hiểu đó là danh sách mà tất cả các phần tử của nó cùng một kiểu.

Khi sử dụng danh sách trong thiết kế thuật toán, chúng ta cần dùng đến một tập các phép toán rất đa dạng trên danh sách. Sau đây là một số phép toán chính. Trong các phép toán này, L ký hiệu một danh sách bất kỳ có độ dài n ≥ 0, x ký hiệu một phần tử bất kỳ cùng kiểu với các phần tử của L và i là số nguyên dương chỉ vị trí.

1. Empty(L). Hàm trả về true nếu L rỗng và false nếu L không rỗng.
2. Length(L). Hàm trả về độ dài của danh sách L.
3. Insert(L, x, i). Thêm phần tử x vào danh sách L tại vị trí i. Nếu thành công thì các phần tử $a_i, a_{i+1}, \dots, a_n$ trở thành các phần tử $a_{i+1}, a_{i+2}, \dots, a_{n+1}$ tương ứng, và độ dài danh sách là n+1. Điều kiện để phép toán xen có thể thực hiện được là i phải là vị trí hợp lý, tức $1 \leq i \leq n+1$, và không gian nhớ dành để lưu danh sách L còn chỗ.
4. Append(L, x). Thêm x vào đuôi danh sách L, độ dài danh sách tăng lên 1.
5. Delete(L, i). Loại phần tử ở vị trí thứ i trong danh sách L. Nếu thành công, các phần tử $a_{i+1}, a_{i+2}, \dots, a_n$ trở thành các phần tử $a_i, a_{i+1}, \dots, a_{n-1}$ tương ứng, và độ dài danh sách là n-1. Phép toán loại

chỉ được thực hiện thành công khi mà danh sách không rỗng và i là vị trí thực sự trong danh sách, tức là $1 \leq i \leq n$.

6. Element(L, i). Tìm biết phần tử ở vị trí thứ i của L. Nếu thành công hàm trả về phần tử ở vị trí i. Điều kiện để phép toán tìm thực hiện thành công cũng giống như đối với phép toán loại.

Chúng ta quan tâm đến các phép toán trên, vì chúng là các phép toán được sử dụng thường xuyên khi xử lý danh sách. Hơn nữa, đó còn là các phép toán sẽ được sử dụng để cài đặt nhiều KDLTT khác như tập động, từ điển, ngăn xếp, hàng đợi, hàng ưu tiên. Nhưng trong các chương trình có sử dụng danh sách, nhiều trường hợp chúng ta cần thực hiện các phép toán đa dạng khác trên danh sách, đặc biệt chúng ta thường phải đi qua danh sách (duyệt danh sách) để xem xét lần lượt từng phần tử của danh sách từ phần tử đầu tiên đến phần tử cuối cùng và tiến hành các xử lý cần thiết với mỗi phần tử của danh sách. Để cho quá trình duyệt danh sách được thực hiện thuận tiện, hiệu quả, chúng ta xác định các phép toán sau đây. Các phép toán này tạo thành bộ công cụ lặp (iteration). Tại mỗi thời điểm, phần tử đang được xem xét của danh sách được gọi là **phần tử hiện thời** và vị trí của nó trong danh sách được gọi là **vị trí hiện thời.**

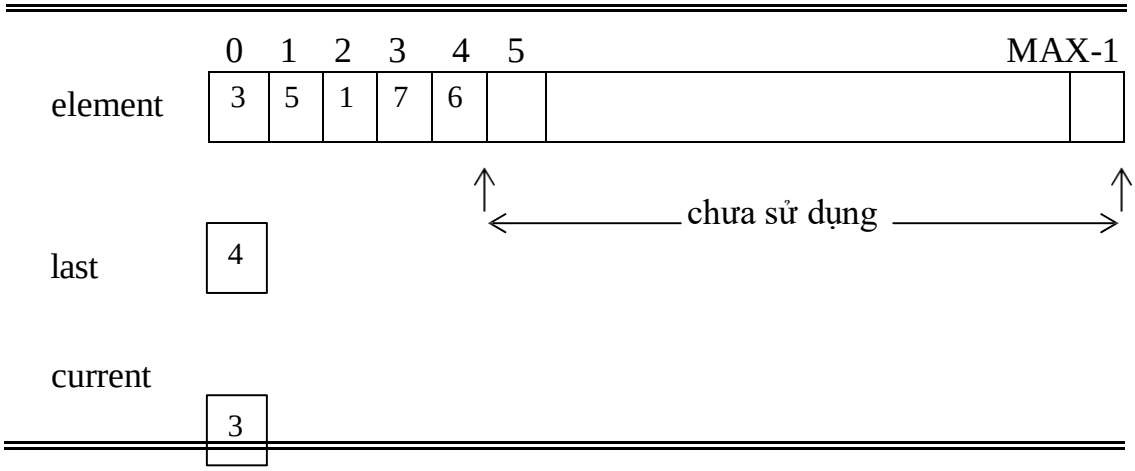
7. Start(L). Đặt vị trí hiện thời là vị trí đầu tiên trong danh sách L.
8. Valid(L). Trả về true, nếu vị trí hiện thời có chứa phần tử của danh sách L, nó trả về false nếu không.
9. Advance(L). Chuyển vị trí hiện thời tới vị trí tiếp theo trong danh sách L.
10. Current(L). Trả về phần tử tại vị trí hiện thời trong L.
11. Add(L, x). Thêm phần tử x vào trước phần tử hiện thời, phần tử hiện thời vẫn còn là phần tử hiện thời.
12. Remove(L). Loại phần tử hiện thời khỏi L. Phần tử đi sau phần bị loại trở thành phần tử hiện thời.

Chúng ta đã đặc tả KDLTT danh sách. Bây giờ chuyển sang giai đoạn cài đặt danh sách.

4.2 CÀI ĐẶT DANH SÁCH BỞI MẢNG

Chúng ta sẽ cài đặt KDLTT danh sách bởi các lớp C + +. Có nhiều cách thiết kế lớp cho KDLTT danh sách, điều đó trước hết là do danh sách có thể biểu diễn bởi các cấu trúc dữ liệu khác nhau. Các thiết kế lớp khác nhau cho danh sách sẽ được trình bày trong chương này và chương sau. Trong mục này chúng ta sẽ trình bày cách cài đặt danh sách bởi mảng (mảng tĩnh). Đây là phương pháp cài đặt đơn giản và tự nhiên nhất.

Chúng ta sẽ sử dụng một mảng element có cỡ là MAX để lưu các phần tử của danh sách. Các phần tử của danh sách sẽ được lần lượt lưu trong các thành phần của mảng element[0], element[1], … , element[n-1], nếu danh sách có n phần tử. Tức là, danh sách được lưu trong đoạn đầu element[0…n-1] của mảng, đoạn sau của mảng, element[n.. MAX-1], là không gian chưa được sử dụng. Cần lưu ý rằng, phần tử thứ i của danh sách (i = 1, 2, …) được lưu trong thành phần element[i-1] của mảng. Cần có một biến last ghi lại chỉ số sau cùng mà tại đó mảng có chứa phần tử của danh sách. Vị trí hiện thời được xác định bởi biến current, nó là chỉ số mà element[current] chứa phần tử hiện thời của danh sách. Chẳng hạn, giả sử chúng ta có danh sách các số nguyên L = (3, 5, 1, 7, 6) và phần tử hiện thời đứng ở vị trí thứ 4 trong danh sách, khi đó danh sách L được biểu diễn bởi cấu trúc dữ liệu được mô tả trong hình 4.1.



Hình 4.1.Biểu diễn danh sách bởi mảng.

Chúng ta muốn thiết kế lớp danh sách sao cho người lập trình có thể sử dụng nó để biểu diễn danh sách với các phần tử có kiểu tuỳ ý. Do đó, lớp danh sách được thiết kế là lớp khuôn phụ thuộc tham biến kiểu Item như sau:

```
template    <class Item>
class       List
{
 public :
      static  const  int  MAX = 50; // khai báo cỡ của mảng.
      // các hàm thành phần.
 private :
      Item    element[MAX] ;
      int     last ;
      int     current ;
} ;
```

Bây giờ cần phải thiết kế các hàm thành phần của lớp List. Ngoài các hàm thành phần tương ứng với các phép toán trên danh sách, chúng ta đưa vào một hàm kiến tạo mặc định, hàm này khởi tạo nên danh sách rỗng. Lớp List chứa ba biến thành phần đã khai báo như trên, nên không cần thiết phải đưa vào hàm kiến tạo copy, hàm huỷ và hàm toán tử gán, vì chỉ cần sử dụng các hàm kiến tạo copy tự động, … do chương trình dịch cung cấp là đủ. Định nghĩa đầy đủ của lớp List được cho trong hình 4.2.

```
// File đầu list.h
# ifndef  LIST_H
# define LIST_H
# include <assert.h>
```

```
template   <class, Item>
class    List
{
  public :
     static  const  int    MAX = 50 ;
     List ( )
     // Khởi tạo danh sách rỗng.
     { last = -1; current = -1; }
     bool  Empty( )  const
     // Kiểm tra danh sách có rỗng không.
     // Postcondition: Hàm trả về true nếu danh sách rỗng và false
     // nếu không
     { return   last < 0; }
     int  Length( )  const
     // Xác định độ dài danh sách.
     // Postcondition: Trả về số phần tử trong danh sách.
     {return   last+1; }
     void  Insert(const Item&x, int i);
     // Xen phần tử x vào vị trí thứ i trong danh sách.
     // Precondition: Length( ) < MAX và 1 ≤ i ≤ Length( )
     // Postcondition: các phần tử của danh sách kể từ vị trí thứ i
     // được đẩy ra sau một vị trí, x nằm ở vị trí i.
     void  Append(const  Item&x);
     // Thêm phần tử x vào đuôi danh sách.
     // Precondition : Length( ) < MAX
     // Postcondition : x là đuôi của danh sách.

     void  Delete(int i)
     // Loại khỏi danh sách phần tử ở vị trí i.
     // Precondition: 1 ≤ i ≤ Length( )
     // Postcondition: phần tử ở vị trí i bị loại khỏi danh sách,
```

```
// các phần tử đi sau được đẩy lên trước một vị trí.

Item &   Element(int i)  const
// Tìm phần tử ở vị trí thứ i.
// Precondition: 1 ≤ i ≤ Length( )
// Postcondition: Trả về phần tử ở vị trí i.
  { assert (1<= i && i <= Length( ) ); return element[i - 1]; }
// Các hàm bộ công cụ lặp:

void   start( )
// Postcondition: vị trí hiện thời là vị trí đầu tiên của danh sách.
  {current = 0; }
bool   Valid( ) const
// Postcondition: Trả về true nếu tại vị trí hiện thời có phần tử
// trong danh sách, và trả về false nếu không.
  {return   current >= 0 && current <= last; }
void   Advance( )
// Precondition: Hàm Valid( ) trả về true.
// Postcondition: Vị trí hiện thời là vị trí tiếp theo trong danh
// sách.
  { assert (Valid( )); assert (Valid( )); current + +;}
Item &   Current( )  const
// Precondition: Hàm Valid( ) trả về true.
// Postcondition: Trả về phần tử hiện thời của danh sách.
  { assert (Valid( ));  return element[current]; }
void    Add(const Item& x);
// Precondition: Length( ) < MAX và hàm Valid( ) trả về true
// Postcondition: Phần tử x được xen vào trước phần tử
// hiện thời, phần tử hiện thời vẫn còn là phần tử hiện thời.

void   Remove( );
```

```
        // Precondition: hàm Valid( ) trả về true.
        // Postcondition: Phần tử hiện thời bị loại khỏi danh sách,
        // phần tử đi sau nó trở thành phần tử hiện thời.

    private :
        Item   element[MAX];
        int   last;
        int   current;
  };
# include "list.template"
# endif
```

Hình 4.2. Định nghĩa lớp List.

Bước tiếp theo chúng ta cần cài đặt các hàm thành phần của lớp List. Trước hết, nói về hàm kiến tạo mặc định, hàm này cần tạo ra một danh sách rỗng, do vậy chỉ cần đặt giá trị cho biến last là –1, giá trị của biến current cũng là –1. Hàm này được cài đặt là hàm inline. Với cách khởi tạo này, mỗi khi cần thêm phần tử mới vào đuôi danh sách (kể cả khi danh sách rỗng), ta chỉ cần tăng chỉ số last lên 1 và đặt phần tử cần thêm vào thành phần mảng element[last].

Hàm Append được định nghĩa như sau:

```
template  <class  Item>
void   List<Item> :: Append (const Item& x)
{ assert (Length( ) < MAX);
  last + + ;
  element[last] = x;
}
```

Hàm Insert. Để xen phần tử x vào vị trí thứ i của danh sách, tức là x cần đặt vào thành phần element[i - 1] trong mảng, chúng ta cần dịch chuyển đoạn element[i – 1 … last] ra sau một vị trí để có chỗ cho phần tử x. Hàm Insert có nội dung như sau:

```
template  <class  Item>
void   List<Item> :: Insert(const Item& x, int i)
{
  assert (Length( ) < MAX && 1 <= i && i <= Length( ));
  last + +;
  for (int k = last; k >= i; k - -)
       element[k] = element[k - 1];
  element[i - 1] = x;
}
```

Hàm Add. Hàm này cũng tương tự như hàm Insert, nó có nhiệm vụ xen phần tử mới x vào vị trí của phần tử hiện thời. Vì phần tử hiện thời được đẩy ra sau một vị trí, nên chỉ số hiện thời phải được tăng lên 1.

```
template <class  Item>
void   List<Item> :: Add(const Item& x)
{
  assert (Length( ) < MAX && Valid( ));
  last + +;
  for(int k = last; k > current; k - -)
       element[k] = element[k - 1];
  element[current] = x;
  current + +;
}
```

Hàm Delete. Muốn loại khỏi danh sách phần tử ở vị trí thứ i, chúng ta cần phải đẩy đoạn cuối của danh sách kể từ vị trí i + 1 lên trước 1 vị trí, và giảm chỉ số last đi 1.

```
template  <class  Item>
void   List<Item> :: Delete(int i)
{
  assert (1 <= i && i <= Length( ));
  for (int k = i – 1; k < last; k + +)
        element[k] = element[k + 1];
  last - - ;
}
```

Hàm Remove. Hàm này được cài đặt tương tự như hàm Delete.

```
template  <class  Item>
void   List<Item> :: Remove( )
{
  assert(Valid( ));
  for (int k = current; k < last; k + +)
  element[k] = element[k + 1];
  last - - ;
}
```

Tất cả các hàm còn lại đều rất đơn giản và được cài đặt là hàm inline.

Bây giờ chúng ta đánh giá hiệu quả của các phép toán của KDLTT danh sách, khi mà danh sách được cài đặt bởi mảng. Ưu điểm của cách cài đặt này là nó cho phép ta truy cập trực tiếp tới từng phần tử của danh sách, vì phần tử thứ i của danh sách được lưu trong thành phần mảng element[i - 1]. Nhờ đó mà thời gian của phép toán Element(i) là O(1). Giả sử danh sách có độ dài n, để xen phần tử mới vào vị trí thứ i trong danh sách, chúng ta cần

đẩy các phần tử lưu ở các thành phần mảng từ element[i -1] đến element[n - 1] ra sau một vị trí. Trong trường hợp xấu nhất (khi xen vào vị trí đầu tiên trong danh sách), cần n lần đẩy, vì vậy thời gian của phép toán Insert là O(n). Phân tích một cách tượng tự, chúng ta thấy rằng thời gian chạy của các phép toán Delete, Add và Remove cũng là O(n), trong đó n là độ dài của danh sách. Thời gian của tất cả các phép toán còn lại đều là O(1).

Như chúng ta đã nói, các phép toán trong bộ công cụ lặp cho phép chúng ta có thể tiến hành dễ dàng các xử lý danh sách cần đến duyệt danh sách. Chẳng hạn, giả sử L là danh sách các số nguyên, để loại khỏi danh sách L tất cả các số nguyên bằng 3, chúng ta có thể viết:

```
L. Start( );
while (L.Valid( ))
    if (L.Current( ) = = 3)
                    L.Remove( ) ;
    else    L.Advance( ) ;
```

Chúng ta cũng có thể in ra tất cả các phần tử của danh sách bằng cách sử dụng vòng lặp for như sau:

```
for (L.Start( ); L.Valid( ); L.Advance( ))
        cout << L.Current( ) << endl;
```

Nhận xét. Cài đặt danh sách bởi mảng có ưu điểm cơ bản là nó cho phép truy cập trực tiếp tới từng phần tử của danh sách. Nhờ đó chúng ta có thể cài đặt rất thuận tiện phép toán tìm kiếm trên danh sách, đặc biệt khi danh sách là danh sách được sắp (các phần tử của nó được sắp xếp theo thứ tự không tăng hoặc không giảm), nếu lưu danh sách được sắp trong mảng, chúng ta có thể cài đặt dễ dàng phương pháp tìm kiếm nhị phân. Phương pháp tìm kiếm nhị phân là một kỹ thuật tìm kiếm rất hiệu quả và sẽ được nghiên cứu trong mục 4.4.

Chúng ta đã cài đặt danh sách bởi mảng tĩnh, mảng có cỡ MAX cố định. Khi danh sách phát triển, tới lúc nào đó mảng sẽ đầy, và lúc đó các phép toán Insert, Append, Add sẽ không thể thực hiện được. Đó là nhược điểm chính của cách cài đặt danh sách bởi mảng tĩnh.

Bây giờ nói về cách thiết kế lớp List. Trong lớp List này, tất cả các hàm trong bộ công cụ lặp được đưa vào các hàm thành phần của lớp và biến lưu vị trí hiện thời cũng là một biến thành phần của lớp. Thiết kế này có vấn đề: chỉ có một biến hiện thời, do đó chúng ta không thể tiến hành đồng thời hai hoặc nhiều phép lặp khác nhau trên cùng một danh sách.

Mục sau sẽ trình bày một phương pháp thiết kế lớp List khác, nó khắc phục được các nhược điểm đã nêu trên.

4.3 CÀI ĐẶT DANH SÁCH BỞI MẢNG ĐỘNG

Trong mục này chúng ta sẽ thiết kế một lớp khác cài đặt KDLTT danh sách, lớp này được đặt tên là Dlist (Dynamic List). Lớp Dlist khác với lớp List đã được trình bày trong mục 4.2 ở hai điểm. Thứ nhất, danh sách được lưu trong mảng được cấp phát động. Thứ hai, các hàm trong bộ công cụ lặp được tách ra và đưa vào một lớp riêng: lớp công cụ lặp trên danh sách, chúng ta sẽ gọi lớp này là DlistIterator. Với cách thiết kế này, chúng ta sẽ khắc phục được các nhược điểm của lớp List đã được nêu ra trong nhận xét ở cuối mục 4.2.

Lớp Dlist chưa ba thành phần dữ liệu: Biến con trỏ element trỏ tới mảng được cấp phát động để lưu các phần tử của danh sách. Biến size lưu cỡ của mảng, và biến last lưu chỉ số cuối cùng mà tại đó mảng chứa phần tử của danh sách.

Lớp Dlist chứa tất cả các hàm thành phần thực hiện các phép toán trên danh sách giống như trong lớp List, trừ ra các hàm công cụ lặp (các hàm này sẽ được đưa vào lớp DlistIterator). Chúng ta đưa vào lớp Dlist hai hàm kiến tạo. Hàm kiến tạo một tham biến nguyên là cỡ của mảng được cấp phát động và hàm kiến tạo copy. Chúng ta cần phải đưa vào lớp Dlist hàm huỷ để thu

hồi bộ nhớ đã cấp phát cho mảng element, khi mà đối tượng không còn cần thiết nữa. Lớp Dlist cũng cần có hàm toán tử gán. Định nghĩa lớp Dlist được cho trong hình 4.3.

```
template <class  Item>
class   DlistIterator ;
// Khai báo trước lớp DlistIterator.

template <class  Item>
class    Dlist
{
  public:
      friend class DlistIterator<Item>;
      Dlist( )
      {element = NULL; size = 0; last = -1; }
      DList (int m);
      Dlist (const Dlist & L);
      ~ Dlist( )
      {delete [ ] element; }
      Dlist & operator = (const Dlist & L);
      inline  bool  Empty( ) const;
      inline  int  Length( ) const;
      void   Insert(const Item & x, int i);
      void   Append(const Item & x);
      void   Delete(int i);
      inline  Item & Element(int i);
  private:
      Item*   element;
      Int    size;
      Int    last;
```

```
};
```

Hình 4.3. Định nghĩa lớp Dlist

Chú ý rằng, trước khi định nghĩa lớp Dlist, chúng ta đã khai báo trước lớp DlistIterator. Khai báo này là cần thiết, bởi vì trong định nghĩa lớp Dlist, chúng ta xác định lớp DlistIterator là bạn của nó.

Sau đây chúng ta sẽ xem xét sự cài đặt các hàm thành phần của lớp Dlist. Các hàm Empty, Length, Delete và Retrieve là hàm hoàn toàn giống như trong lớp List.

Các hàm kiến tạo. Trước hết nói đến hàm kiến tạo có một tham biến nguyên m. Nhiệm vụ chính của nó là cấp phát một mảng động có cỡ là m để lưu các phần tử của danh sách.

```
template <class  Item>
Dlist<Item> :: Dlist(int m)
// Precondition. m là số nguyên dương.
// Postcondition. Một danh sách rỗng được khởi tạo, với khả năng tối
// đa chứa được m phần tử.
{
    element = new Item[m];
    assert (element != NULL);
    size = m;
    last = -1;
}
```

Hàm kiến tạo copy có trách nhiệm tạo ra một danh sách mới là bản sao của danh sách đã có L. Trước hết ta cần cấp phát một mảng động có cỡ là cỡ của mảng trong danh sách L, sau đó sao chép từng thành phần của mảng trong danh sách L sang mảng mới.

```
template <class Item>
Dlist<Item> :: Dlist(const Dlist<Item> & L)
{
    element = new Item[L.size];
    size = L.size;
    last = L.last;
    for (int i = 0; i <= last; i + +)
        element[i] = L.element[i];
}
```

Toán tử gán. Toán tử gán được cài đặt gần giống như hàm kiến tạo copy. Chỉ có điều cần lưu ý là, khi cỡ của mảng trong hai danh sách khác nhau, chúng ta mới cần cấp phát một mảng động có cỡ là cỡ của mảng trong danh sách nguồn, rồi thực hiện sao chép giống như trong hàm kiến tạo copy.

```
Dlist<Item> & Dlist<Item> :: operator = (const Dlist<Item> & L)
{
 if (size != L.size)
 {
    delete [ ] element;
    element = new Item[L.size];
    size = L.size;
 }
 last = L.last;
 for(int i = 0; i <= last; i + +)
   element[i] = L.element[i];
 return   *this;
}
```

Hàm Insert. Ưu điểm của cách cài đặt danh sách bởi mảng động là các hành động thêm phần tử mới vào danh sách luôn luôn được thực hiện. Bởi vì khi mà mảng đầy, chúng ta sẽ cấp phát một mảng động mới có cỡ gấp đôi cỡ của mảng cũ. Sau đó sao chép đoạn đầu [0… i-1] của mảng cũ sang mảng mới, đưa phần tử cần xen vào mảng mới, rồi lại chép đoạn còn lại của mảng cũ sang mảng mới. Hàm Insert được định nghĩa như sau:

```
template <class  Item>
void   Dlist<Item> :: Insert(const Item&x, int i)
{
   assert(i >= 0 && i <= last);
   if (Length( ) < size)
   {
     last + +;
     for (int k = last; k >= i; k - -)
       element[k] = element[k -1];
       element[i-1] = x;
   }
   else  // mảng element đầy
   {
      Item*   newArray = new Item[2 * size + 1]
      assert (newArray != NULL);
      for (int k = 0; k < i –1; k + + )
               newArray[k] = element[k];
      newArray[i - 1] = x;
      for (int k = i; k <= last + 1; k + + )
               newArray[k] = element[k - 1];
      delete [ ] element;
      element = newArray;
      last + +;
      size = 2 * size + 1;
```

```
    }
}
```

Hàm Append được cài đặt tương tự như hàm Insert, nhưng đơn giản hơn. Chúng tôi để lại cho độc giả, xem như bài tập.

Hàm Delete được cài đặt như trong lớp List (xem mục 4.2).

Bây giờ chúng ta thiết kế lớp công cụ lặp trên danh sách được cài đặt bởi mảng: lớp DlistIterator. Khai báo lớp công cụ lặp được sử dụng với lớp Dlist được cho trong hình 4.4. Lớp này chứa các phép toán của KDLTT danh sách liên quan tới phép lặp qua các phần tử của danh sách. Lớp DlistIterator chứa hai thành phần dữ liệu: biến current lưu số nguyên biểu diễn vị trí hiện thời, nó là chỉ số mà tại đó mảng lưu phần tử hiện thời; và biến LPtr lưu một con trỏ hằng trỏ tới đối tượng của lớp Dlist mà các phép toán công cụ lặp sẽ thực hiện trên nó.

```
# include  <assert.h>
template   <class Item>
class    DlistIterator
{
  public :
      DlistIterator (const Dlist<Item> & L) // Hàm kiến tạo
      { LPtr = &L; current = -1; }
      void   Start( )
      { current = 0; }
      bool   Valid( )  const
      { return 0 <= current && current <= LPtr ⟶ last; }
      void   Advance( )
      { assert (Valid( )); current + +;}
      Item &   Current( ) const
      { assert(Valid( )); return LPtr  ⟶ element[current]; }
```

```
        void    Add(const Item & x);
        void    Remove( );
    private :
        const   Dlist<Item>*    LPtr;
        int     current;
};
```

Hình 4.4. Định nghĩa lớp công cụ lặp.

Chú ý rằng, chúng ta đã khai báo lớp DlistIterator là bạn của lớp Dlist. Điều này là để khi cài đặt các hàm thành phần của lớp DlistIterator chúng ta có quyền truy cập trực tiếp tới các thành phần dữ liệu của lớp Dlist. Bây giờ chúng ta cài đặt hàm Add, hàm này được cài đặt hoàn toàn tương tự như hàm Insert trong lớp Dlist. Chỉ cần để ý rằng, ở đây chúng ta cần xen một phần tử mới vào vị trí hiện thời trong danh sách mà con trỏ LPtr trỏ tới.

```
template <class Item>
void   DlistIterator<Item> :: Add(const Item & x)
{
  if (LPtr →Length( ) < LPtr → size)
    {
      LPtr → last + +;
      for (int k = LPtr → last; k > current; k - - )
              LPtr → element[k] = LPtr → element[k - 1];
      LPtr →element[current] = x;
    }
  else {
       Item* newArray = new Item[ 2 * (LPtr → size + 1) ];
       assert(newArray != NULL);
       for(int i = 0; i < current; i + + )
              newArray[i] = LPtr →element[i];
```

```
        newArray[current] = x;
        for(int i = current + 1; i <= LPtr →Length( ); i + +)
                newArray[i] = LPtr →element[i - 1];
        delete [ ] LPtr → element;
        LPtr → element = newArray;
        LPtr → size = 2 * (LPtr → size) + 1;
        LPtr → last + +;
        }
        current + +;
}
```

Dễ dàng thấy rằng, mặc dầu các phép toán Insert, Append và Add cài đặt phức tạp hơn các phép toán tương ứng khi danh sách được cài đặt bởi mảng tĩnh, song thời gian thực hiện các phép toán đó vẫn chỉ là O(n).

Chúng ta đưa ra một ví dụ minh hoạ cách sử dụng lớp công cụ lặp. Giả sử L là danh sách các số nguyên. Chúng ta cần thêm số nguyên 4 vào trước số nguyên 5 xuất hiện lần đầu trong danh sách, nếu L không chứa 5 thì thêm 4 vào đuôi danh sách. Trước hết, chúng ta phải tạo ra một đối tượng của lớp công cụ lặp gắn với danh sách L, và cần nhớ rằng, mọi phép lặp cần đến thao tác đầu tiên là Start. Xem đoạn mã sau:

```
Dlist<int>  L(100); // Danh sách L có thể chứa được tối đa 100
                    // số nguyên.
….
DlistIterator<Int>  itr(L); // khởi tạo đối tượng lặp itr gắn với
                            // danh sách L.
itr.Start( );
while(itr.Valid( ))
   if (itr.Current( ) = = 5)
           { itr.Add(4); break; }
   else    itr.Advance( );
if (!itr.Valid( ))
```

L. Append(4);

4.4 CÀI ĐẶT TẬP ĐỘNG BỞI DANH SÁCH. TÌM KIẾM TUẦN TỰ VÀ TÌM KIẾM NHỊ PHÂN

Nhớ lại rằng, mỗi đối tượng của KDLTT tập động là một tập các phần tử dữ liệu , và các phần tử dữ liệu chứa một thành phần dữ liệu được gọi là khoá. Chúng ta giả thiết rằng, trên các giá trị khoá có quan hệ thứ tự và các phần tử dữ liệu khác nhau có giá trị khoá khác nhau. Trên tập các phần tử dữ liệu S, chúng ta xác định các phép toán cơ bản sau:

1. Insert(S, x). Xen phần tử dữ liệu x vào tập S.
2. Delete(S, k). Loại khỏi tập S phần tử dữ liệu có khoá k.
3. Search(S, k). Tìm phần tử dữ liệu có giá trị khoá là k trong tập S. Hàm trả về true nếu tìm thấy và false nếu ngược lại.
4. Max(S). Hàm trả về phần tử dữ liệu có giá trị khoá lớn nhất trong tập S.
5. Min(S). Hàm trả về phần tử dữ liệu có giá trị khoá nhỏ nhất trong tập S.

Để viết cho ngắn gọn, chúng ta giả sử rằng các phần tử dữ liệu có kiểu là Item và có thể truy cập trực tiếp tới thành phần khoá của Item, chẳng hạn trong các áp dụng thông thường Item là một cấu trúc:

```
struct   Item
{
  keyType   key; // khoá của dữ liệu.
  // các thành phần khác.
};
```

4.4.1 Cài đặt bởi danh sách không được sắp. Tìm kiếm tuần tự

Chúng ta có thể sắp xếp các phần tử của tập động (theo một thứ tự tuỳ ý) thành một danh sách, và do đó dễ dàng thấy rằng, ta có thể sử dụng cách

cài đặt danh sách bởi mảng động để cài đặt KDLTT tập động. Lớp tập động Dset (Dynamic Set) được thiết kế bằng cách sử dụng lớp Dlist (xem mục 4.3) làm lớp cơ sở với dạng thừa kế private. Với cách thiết kế này , các hàm thành phần của lớp Dlist trở thành các hàm thành phần private của lớp DSet và chúng ta sẽ sử dụng chúng để cài đặt các phép toán tập động. Lớp Dset chỉ chứa các thành phần dữ liệu được thừa kế từ lớp Dlist. Định nghĩa lớp Dset được cho trong hình 4.5.

```
template <class Item>
class Dset : private Dlist<Item>
{
  public :
     DSet(int m = 1)
     // khởi tạo ra tập rỗng, có thể chứa được nhiều nhất là m phần tử
     // dữ liệu.
     : Dlist(m) { }
     void DSetInsert(const Item & x);
     // Postcondition: phần tử x trở thành thành viên của tập động.
     void DSetDelete(keyType k);
     // Postcondition: phần tử với khoá k không có trong tập động.
     bool Search(keyType k);
     // Postcondition: Trả về true nếu phần tử với khoá k có trong
     // tập động và trả về false nếu không.
     Item & Max( );
     // Precondition: Danh sách không rỗng.
     // Postcondition: Trả về phần tử có khoá lớn nhất trong tập động.
     Item & Min( );
     // Precondition: Danh sách không rỗng.
     // Postcondition: Trả về phần tử có khoá nhỏ nhất trong tập động.
};
```

Hình 4.5. Định nghĩa lớp DSet.

Sau đây chúng ta cài đặt các phép toán trên tập động bằng cách sử dụng các hàm được thừa kế từ lớp danh sách.

Hàm DSetInsert. Hàm này được thực hiện bằng cách thêm phần tử mới vào đuôi danh sách:

```
template   <class  Item>
void    DSet<Item> ::  DSetInsert(const Item & x)
{
  if (! Search(x.key))
                Append(x);
}
```

Hàm DSetDelete. Để loại phần tử có khóa k khỏi tập động, chúng ta cần xem xét từng phần tử trong danh sách, khi gặp phần tử cần loại ở vị trí thứ i trong danh sách thì sử dụng hàm Delete(i) thừa kế từ lớp Dlist.

```
template <class  Item>
void   DSet<Item> : DSetDelete(keyType  k)
{
  for(int i = 1; i <= Length(  ); i + +)
        if (Element(i).key = = k)
                  {
                    Delete(i);
                     break;
                   }
}
```

Hàm Search. Cần lưu ý rằng, các phần tử của tập động đã được lưu dưới dạng một danh sách. Do đó để tìm một phần tử với khoá cho trước có

trong tập động hay không, chúng ta xem xét lần lượt từng phần tử của danh sách bắt đầu từ phần tử đầu tiên, cho tới khi gặp phần tử cần tìm hoặc đi đến hết danh sách mà không thấy. Kỹ thuật tìm kiếm này được gọi là **tìm kiếm tuần tự (sequential search)**, đó là một dạng của tìm kiếm vét cạn.

```
template <class  Item>
bool   DSet<Item> :: Search(keyType    k)
{
  for(int i = 1; i <= Length(  ); i + +)
          if (Element(i).key = = k)
                          return   true;
  return     false;
}
```

Phép toán Max, Min cũng được cài đặt rất đơn giản. Chúng ta chỉ cần đi qua danh sách và lưu lại phần tử có khoá lớn nhất (nhỏ nhất).

Thời gian thực hiện các phép toán tập động trong cách cài đặt này. Để thực hiện các phép toán DSetInsert, DSetDelete, Search, Max, Min chúng ta đều cần phải đi qua từng phần tử của danh sách, kể từ phần tử đầu tiên (vòng lặp for). Khi tìm kiếm, trong trường hợp xấu nhất (chẳng hạn phần tử cần tìm không có trong danh sách), cần phải đi hết danh sách, do đó số lần lặp tối đa là n (n là độ dài danh sách). Vì danh sách được lưu trong mảng, nên phép toán Element(i) chỉ tốn thời gian O(1). Do đó thời gian thực hiện tìm kiếm là O(n), trong đó n là độ dài danh sách. Phân tích tương tự ta thấy rằng, thời gian của các phép toán khác cũng là O(n).

4.4.2 Cài đặt bởi danh sách được sắp. Tìm kiếm nhị phân

Bây giờ chúng ta xét một cách cài đặt tập động khác, trong cách này, tập động cũng được biểu diễn dưới dạng danh sách, nhưng các phần tử của danh sách được sắp xếp theo thứ tự tăng của các giá trị khoá. Phần tử có

khoá nhỏ nhất của tập động đứng ở vị trí đầu tiên trong danh sách, còn phần tử có khoá lớn nhất của tập động đứng sau cùng trong danh sách. Do đó, các phép toán Max, Min được cài đặt rất đơn giản và chỉ cần thời gian O(1). Chẳng hạn, phép toán Min được xác định như sau:

```
template <class  Item>
Item &   DSet<Item> :: Min(  )
{
    assert( ! Empty(  ));
    return    Element(1);
}
```

Bây giờ chúng ta viết hàm DSetInsert. Vì tập động được biểu diễn dưới dạng danh sách được sắp, nên khi xen một phần tử mới vào tập động, chúng ta cần phải đặt nó vào vị trí thích hợp trong danh sách, để đảm bảo sau khi xen, danh sách vẫn còn là danh sách được sắp. DSetInsert được cài đặt như sau:

```
template <class  Item>
void   DSet<Item> :: DSetInsert(const Item & x)
{
  int   i;
  for(i = 1; i < = Length(  ); i + +)
      if (x.key < Element(i).key)
         { Insert(x, i); break; }
      else if (x.key = = Element(i).key)
               break;
      if (i > Length(  ))      // danh sách rỗng hoặc x có khóa lớn hơn
              Append(x);     // khoá của mọi phần tử trong danh sách.
}
```

Phép toán DSetDelete được cài đặt giống như trong trường hợp danh sách không được sắp, nhưng cần lưu ý rằng, khi gặp một phần tử trong danh sách có khoá lớn hơn khoá k có nghĩa là trong danh sách không chứa phần tử cần loại và do đó ta có thể dừng lại mà không cần đi hết danh sách.

```
template <class Item>
void DSet<Item> :: DSetDelete(keyType k)
{
  for (int i = 1; i <= Length(  ); i + +)
    if (Element(i).key = = k)
            {
              Delete(i);
              break;
             }
    else if (Element(i).key > k)
              break;
}
```

Ưu điểm lớn nhất của phương pháp cài đặt tập động bởi danh sách được sắp là chúng ta có thể sử dụng kỹ thuật tìm kiếm khác hiệu quả hơn tìm kiếm tuần tự.

Tìm kiếm nhị phân (Binary Search)

Tư tưởng của kỹ thuật tìm kiếm nhị phân là như sau: xét phần tử đứng giữa danh sách, nó chia danh sách thành hai phần: nửa đầu và nửa sau. (Chú ý rằng, vì danh sách được lưu trong mảng, nên ta có thể tìm thấy phần tử ở giữa danh sách chỉ với thời gian O(1)). Do danh sách được sắp xếp theo thứ tự tăng của khoá, nên tất cả các phần tử ở nửa đầu danh sách đều có khoá nhỏ hơn khoá của phần tử đứng giữa danh sách và khoá của phần tử này nhỏ hơn khoá của tất cả các phần tử ở nửa sau danh sách. Nếu khoá k

của phần tử cần tìm bằng khoá của phần tử đứng giữa danh sách có nghĩa là ta đã tìm thấy; còn nếu k khác khoá của phần tử đứng giữa, ta tiếp tục tìm kiếm ở nửa đầu (nửa sau) danh sách tuỳ thuộc khoá k nhỏ hơn (lớn hơn) khóa của phần tử đứng giữa danh sách. Quá trình tìm kiếm ở nửa đầu (hoặc nửa sau) được thực hiện bằng cách lặp lại thủ tục trên. Chúng ta có thể cài đặt thuật toán tìm kiếm nhị phân bởi hàm đệ quy hoặc không đệ quy. Trong hình 4.6 là hàm Search không đệ quy.

```
template <class  Item>
bool   DSet<Item> :: Search(keyType k)
{
    int   bottom, top, mid;
    bottom = 1;
    top = Length(  );
    while (bottom <= top)
    {
       mid = (bottom + top) / 2;
       if (k = = Element(mid).key)
                      return    true;
       else if (k  < Element(mid).key)
                     top = mid – 1;
            else    bottom = mid + 1;
    }
    return    false;
}
```

Hình 4.6.Tìm kiếm nhị phân.

Thời gian tìm kiếm nhị phân. Bây giờ chúng ta đánh giá thời gian chạy của thuật toán tìm kiếm nhị phân theo độ dài n của danh sách. Dễ dàng thấy rằng, thời gian thực hiện thân của vòng lặp while là O(1), bởi vì việc

truy cập tới phần tử đứng giữa danh sách, Element(mid), chỉ cần thời gian O(1). Do đó chúng ta chỉ cần đánh giá số lần lặp. Mỗi lần lặp là một lần chia danh sách đang xét thành hai phần: nửa đầu và nửa sau, và danh sách được xét tới ở lần lặp sau là một trong hai phần đó. Số lần lặp tối đa là số tối đa lần chia danh sách ban đầu cho tới khi nhận được danh sách cần xem xét chỉ gồm một phần tử (bottom = top). Trường hợp xấu nhất này sẽ xảy ra khi mà phần tử cần tìm là phần tử đầu tiên (phần tử cuối cùng) của danh sách, hoặc không có trong danh sách. Nếu n là chẵn, thì khi chia đôi ta nhận được danh sách ở bên trái phần tử đứng giữa có độ dài n/2 –1, còn danh sách con bên phải có độ dài n/2. Xét trường hợp $n = 2^r$. Nếu phần tử cần tìm ở cuối cùng trong danh sách hoặc khoá k lớn hơn khoá của tất cả các phần tử trong danh sách, thì số lần chia là r. Đó là trường hợp xấu nhất. Vì vậy, trong trường hợp $n = 2^r$, thì số lần chia nhiều nhất là r và $r = \log_2 n$.

Còn nếu $n \neq 2^r$ thì sao? Chúng ta có thể tìm được số nguyên r nhỏ nhất sao cho:

$2^{r-1} < n < 2^r$

$r - 1 < \log_2 n < r$

$r < 1 + \log_2 n < r + 1$

Do đó, trong trường hợp $n \neq 2^r$, thì số lần chia tối đa để dẫn đến danh sách gồm một phần tử không vượt quá $r < 1 + \log_2 n$.

Như vậy, thời gian của thuật toán tìm kiếm nhị phân là O(logn).

So sánh hai phương pháp cài đặt.

Nếu chúng ta cài đặt tập động bởi danh sách không được sắp (các phần tử của tập động được xếp thành danh sách theo trật tự tuỳ ý), thì thời gian thực hiện các phép toán trên tập động là O(n). (Cần lưu ý rằng, để xen phần tử mới vào tập động, chúng ta chỉ cần thêm nó vào đuôi danh sách, nhưng chúng ta cần phải kiểm tra nó đã có trong tập động chưa, tức là vẫn cần duyệt danh sách, do đó phép toán DSetInsert cũng đòi hỏi thời gian O(n)). Trong khi đó, nếu tập động được biểu diễn bởi danh sách được sắp

(các phần tử của tập động được sắp xếp thành danh sách theo thứ tự tăng (hoặc giảm) của các giá trị khoá), thì các phép toán DSetInsert, DSetDelete vẫn cần thời gian O(n), phép toán Min, Max chỉ cần thời gian O(1); đặc biệt phép toán Search, do áp dụng kỹ thuật tìm kiếm nhị phân, nên chỉ đòi hỏi thời gian O(logn).

Trên đây chúng ta đã trình bày một cách thiết kế lớp tập động DSet sử dụng lớp Dlist như lớp cơ sở private. Chúng ta có thể đưa ra một cách thiết kế lớp DSet khác, thay cho sử dụng lớp cơ sở Dlist, lớp DSet sẽ chứa một thành phần dữ liệu là đối tượng của lớp Dlist. Lớp DSet này sẽ có dạng như sau:

```
template   <class  Item>
class    DSet1
{
  public :
    // Các hàm thành phần như trong lớp DSet
  private :
       Dlist    L;
};
```

Chúng ta còn có thể thiết kế lớp DSet một cách khác không cần sử dụng đến lớp Dlist, mà trực tiếp sử dụng ba thành phần dữ liệu: biến con trỏ element trỏ tới mảng được cấp phát động để lưu các phần tử của danh sách biểu diễn tập động, biến size lưu cỡ của mảng động và biến last lưu chỉ số của mảng chứa phần tử cuối cùng của danh sách. Trong cách thiết kế này, lớp DSet có dạng sau:

```
template <class    Item>
class    DSet2
{
  public :
```

```
    // Các hàm kiến tạo mặc định, copy
    // Hàm huỷ
    // Toán tử gán, …
    // Các hàm cho các phép toán tập động.
  private :
    Item*   element;
    int   size;
    int   last;
};
```

Độc giả có thể cài đặt dễ dàng các lớp DSet1 và DSet2 (bài tập).

4.5 ỨNG DỤNG

Danh sách là một cấu trúc dữ liệu tuyến tính được sử dụng nhiều trong thiết kế các thuật toán. Nhiều loại đối tượng dữ liệu có thể biểu diễn dưới dạng danh sách, và do đó chúng ta có thể sử dụng danh sách để cài đặt nhiều KDLTT khác. Chẳng hạn, trong mục 4.4, chúng ta đã sử dụng danh sách để cài đặt KDLTT tập động. Trong mục này chúng ta sẽ xét một vài ứng dụng khác: sử dụng danh sách để cài đặt đa thức và các phép toán đa thức, và sử dụng danh sách để cài đặt các ma trận thưa (ma trận chỉ chứa một số ít các thành phần khác không).

Đa thức và các phép toán đa thức.

Một đa thức là một biểu thức có dạng:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

Mỗi hạng thức $a_k x^k$, $a_k \neq 0$ có thể biểu diễn bởi một cặp hai thành phần: hệ số (coef) a_k và số mũ (expo) k. Và do đó, chúng ta có thể biểu diễn đa thức như là một danh sách các cặp hệ số, số mũ. Chẳng hạn, đa thức:

$$17x^5 - 25x^2 + 14x - 32$$

được biểu diễn dưới dạng danh sách sau:

((17, 5), (-25, 2), (14, 1), (-32, 0))

Các đa thức cùng với các phép toán quen biết trên đa thức tạo thành KDLTT đa thức. Chúng ta sẽ cài đặt lớp đa thức (class Poly), bằng cách biểu diễn đa thức dưới dạng danh sách các hạng thức như đã trình bày ở trên. Lớp đa thức chứa một thành phần dữ liệu là danh sách các hạng thức. Do đó, trước khi định nghĩa lớp đa thức, chúng ta cần xác định lớp hạng thức (class Term).

Định nghĩa lớp đa thức được cho trong hình 4.7. Để cho ngắn gọn, trong đó chúng ta chỉ xác định một hàm toán tử thực hiện phép cộng đa thức, các hàm thành phần cần thiết khác độc giả tự xác định lấy.

```
class   Poly; // khai báo trước lớp đa thức.
class   Term  // Lớp hạng thức.
{
  public :
    friend class Poly;
    Term(double a = 0, int k = 0);
    // Kiến tạo nên hạng thức có hệ số a, số mũ k.
  private:
        double   coef;
        int   expo;
};
class Poly  // Lớp đa thức.
{
  public :
            ……
    Poly& operator + (const Poly & p);
    // Toán tử cộng hai đa thức.
            ……
  private :
```

```
    Dlist<Term>     TermList; // Danh sách các hạng thức.
};
```

Hình 4.7.Lớp đa thức.

Bây giờ chúng ta cài đặt một hàm thành phần: hàm cộng đa thức. Thuật toán cộng hai đa thức như sau: sử dụng hai phép lặp, mỗi phép lặp chạy trên một danh sách các hạng thức của một đa thức. Chúng ta so sánh hai hạng thức hiện thời, nếu chúng có cùng số mũ thì tạo ra một hạng thức mới với số mũ là số mũ đó, còn hệ số là tổng các hệ số của hai hạng thức hiện thời (nếu tổng này khác không), và đưa hạng thức mới vào đuôi danh sách hạng thức của đa thức kết quả, đồng thời dịch chuyển hai vị trí hiện thời đến các vị trí tiếp theo trong hai danh sách. Nếu hạng thức hiện thời của một trong hai danh sách có số mũ lớn hơn số mũ của hạng thức hiện thời kia, thì ta đưa hạng hiện thời có số mũ lớn hơn vào đuôi danh sách của đa thức kết quả. Khi mà một vị trí hiện thời đã chạy hết danh sách, chúng ta tiếp tục cho vị trí hiện thời kia chạy trên danh sách còn lại, để ghi các hạng thức còn lại của danh sách này vào đuôi danh sách của đa thức kết quả. Hàm toán tử cộng đa thức được cho trong hình 4.8.

```
Poly &  Poly :: operator + (const Poly & p)
{
   Term    term1, term2, term3;
   Poly result;
   DListIterator<Term>  itr1(TermList);
   DlistIterator<Term>    itr2(p.TermList);
   itr1.Start( );
   itr2.Start( );
   while (itr1.Valid( ) && itr2.Valid( ))
   {
     term1 = itr1.Current( );
```

```
  term2 = itr2.Current( 0;
  if (term1. expo = = term2. expo)
  {
     if (term1.coef + term2.coef ! = 0)
     {
         term3.coef = term1.coef + term2.coef;
         term3.expo = term1.expo;
         result.TermList.Append(term3);
         // Thêm hạng thức term3 vào đuôi danh sách các hạng thức của
         // đa thức kết quả.
     }
     itr1.Advance( );
     itr2.Advance( );
  }
  else if (term1.expo > term2.expo)
     {
         result.TermList.Append(term1);
         itr1.Advance( );
      }
  else {
         result.TermList.Append(term2);
         itr2.Advance( );
         }
}; // Hết while
while (itr1.Valid( ))
{
   term1 = itr1.Current( );
   result.TermList.Append(term1);
   itr1.Advance( );
}
while (itr2.Valid( ))
```

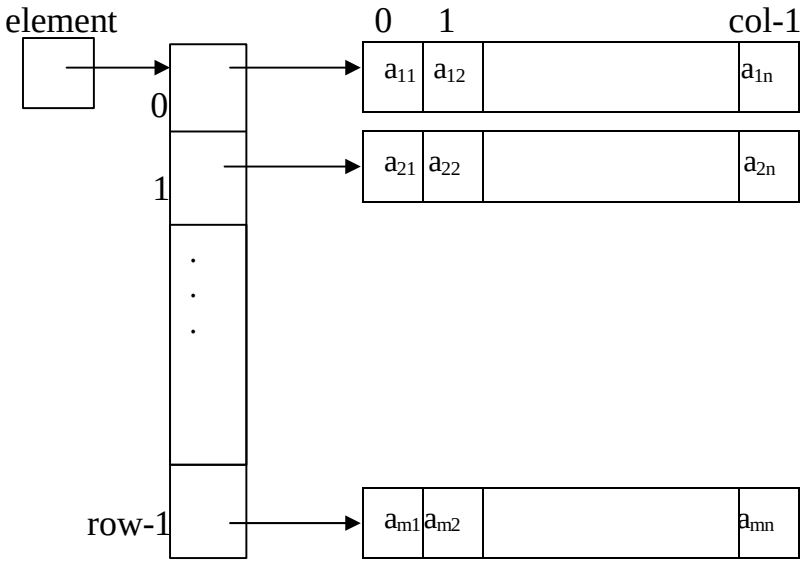
```
    {
      term2 = itr2.Current( );
      result.TermList.Append(term2);
      itr2.Advance( );
    }

    return    result;

}
```

Hình 4.8.Hàm cộng đa thức.

Ma trân thưa. Một ma trận là một bảng hình chữ nhật chứa m x n phần tử được sắp xếp thành m dòng, n cột:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

trong đó a_{ij} là phần tử đứng ở dòng thứ i, cột thứ j. Tập hợp các ma trận cùng với các phép toán quen biết trên ma trận lập thành KDLTT ma trận. Để cài đặt KDLTT ma trận, trước hết chúng ta cần biểu diễn ma trận bởi cấu trúc dữ liệu thích hợp. Cách tự nhiên nhất là cài đặt ma trận bởi mảng hai chiều. Nhưng để có thể biểu diễn được ma trận có cỡ m x n bất kỳ, chúng ta sử dụng mảng được cấp phát động. Do đó, để biểu diễn ma trận chúng ta sử dụng ba biến: biến row lưu số dòng và biến col lưu số cột của ma trận, và con trỏ element trỏ tới mảng động cỡ row, mảng này chứa các con trỏ trỏ tới các mảng động cỡ col để lưu các phần tử của ma trận, như trong hình sau:



Với cách biểu diễn ma trận như trên, lớp ma trận sẽ có nội dung như sau:

```
template <class  Item>
class    Matrix
{
private :
   int   row, col;
   Item**   element;
public :
   Matrix(  )
   { row = 0; col = 0; element = NULL; }
   Matrix (int m, int n);
   // Kiến tạo ma trận không m dòng, n cột.
   Matrix (const Matrix & M); // Kiến tạo copy.
   ~ Matrix(  ); // Hàm huỷ.
   Matrix &  operator = (const Matrix & M); // Toán tử gán.
   int   Row( ) const
   { return   row; }
   int   Col( ) const
```

```
    { return   col; }
    friend Matrix<Item> & operator + (const Matrix<Item> & M1
                                      const Matrix<Item> & M2);
    // Các hàm toán tử cho các phép toán khác trên ma trận.
};
```

Bây giờ chúng ta xét các ma trận thưa: ma trận chỉ chứa một số ít các phần tử khác không. Chẳng hạn ma trận sau:

$$M = \begin{bmatrix} 0 & 7 & 0 & 0 & 3 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 9 & 0 \end{bmatrix}$$

Với các ma trận thưa, nếu biểu diễn ma trận bởi mảng sẽ rất lãng phí bộ nhớ. Chẳng hạn, với ma trận cỡ 1000 x 2000 chúng ta cần cấp phát bộ nhớ để lưu 2 triệu phần tử. Song nếu ma trận đó chỉ chứa 2000 phần tử khác không thì chúng ta mong muốn chỉ cần cấp phát bộ nhớ để lưu 2000 phần tử đó. Cách tiếp cận biểu diễn ma trận bởi danh sách cho phép ta làm được điều đó.

Mỗi phần tử của ma trận được mô tả bởi một cặp: chỉ số cột của phần tử đó và giá trị của nó. Mỗi dòng của ma trận sẽ được biểu diễn bởi danh sách các cặp (chỉ số, giá trị). Chẳng hạn, dòng thứ nhất của ma trận M ở trên được biểu diễn bởi danh sách:

((2,7), (5,3))

Một ma trận có thể xem như danh sách các dòng. Và do đó, chúng ta có thể biểu diễn ma trận bởi danh sách của các danh sách các cặp (chỉ số, giá trị). Chẳng hạn, ma trận M ở trên được biểu diễn bởi danh sách sau:

(((2,7), (5,3)), ((3,8)), ((2,5), (4,9)))

Với cách biểu diễn ma trận bởi danh sách, để xây dựng lớp các ma trận thưa (SpMatrix), trước hết chúng ta cần xác định lớp các phần tử của ma trận (Elem).

```
template <class  Item>
class    MaRow; // Lớp dòng của ma trận.
template <class  Item>
class    SpMatrix; // Lớp ma trận thưa.
template <class  Item>
class    Elem // Lớp phần tử của ma trận.
{
   friend   class MaRow<Item>;
   friend   class SpMatrix<Item>;
public :
    Elem (int j, Item a)
    // Kiến tạo một phần tử ở cột j, có giá trị a.
    { colIndex = j; value = a; }
private :
   int    colIndex; // Chỉ số cột.
   Item    value; // giá trị.
};
```

Lớp dòng của ma trận (MaRow) chứa hai thành phần dữ liệu: chỉ số dòng của ma trận (rowIndex), và một thành phần dữ liệu khác là danh sách các phần tử trong dòng đó, danh sách này được ký hiệu là eleList và được cài đặt bởi danh sách động Dlist (xem mục 4.3). Trong lớp MaRow, chúng ta cần đưa vào các hàm phục vụ cho việc thực hiện các phép toán trên ma trận: đó là các hàm toán tử + (cộng hai dòng), - (trừ hai dòng), * (nhân một dòng với một giá trị), * (nhân một dòng với một ma trận), … Lớp MaRow được xác định như sau:

```
template <class  Item>
class    MaRow
{
   friend   class   SpMatrix;
```

```
public :
    MaRow(  ); // khởi tạo dòng rỗng.
    MaRow & operator + (const MaRow & R);
    MaRow & operator + = (const MaRow & R);
    MaRow & operator – (const MaRow & R);
    MaRow & operator - = (const MaRow & R);
    MaRow & operator * (const Item & a);
    // nhân một dòng với giá trị a.
    MaRow & operator * (const SpMatrix<Item> & M);
    // nhân một dòng với ma trận M.
  private :
    Dlist <Elem<Item>>   eleList; // Danh sách các phần tử.
    int  rowIndex; // chỉ số dòng.
};
```

Tại sao trong lớp MaRow chúng ta cần đưa vào các hàm toán tử trên? Để cộng (trừ) hai ma trận, chúng ta cần cộng (trừ) các dòng tương ứng của hai ma trận, vì vậy trong lớp MaRow chúng ta đã đưa vào các toán tử +, + =, -, - =. Chú ý rằng, mỗi dòng của ma trận được biểu diễn bởi danh sách các phần tử được sắp xếp theo thứ tự tăng dần theo chỉ số cột, vì vậy phép cộng (trừ) các dòng được thực hiện theo thuật toán hoàn toàn giống như khi ta thực hiện cộng hai đa thức. Phép toán nhân dòng ma trận với một giá trị được đưa vào để cài đặt phép nhân ma trận với một giá trị. Bây giờ chúng ta thảo luận về phép nhân một dòng với một ma trận. Tại sao lại cần phép toán này? Giả sử cần tính tích của hai ma trận A và B: A * B = C. Phần tử c_{ij} của ma trận C được tính theo công thức:

$c_{ij} = \sum_k a_{ik} b_{kj}$, trong đó A = (a_{ik}), B = (b_{kj}).

Tính theo công thức này sẽ kém hiệu quả, vì chúng ta phải duyệt dòng thứ i của ma trận A, và với mỗi phần tử ở cột k, chúng ta cần tìm đến dòng

thứ k của ma trận B để tìm phần tử ở cột j. Song phân tích công thức trên, chúng ta sẽ thấy rằng, dòng thứ i của ma trận C là dòng thứ i của ma trận A nhân với ma trận B. Trong đó phép nhân một dòng với ma trận B được định nghĩa như sau: Lấy phần tử ở cột k của dòng nhân với dòng thứ k của ma trận B, rồi cộng tất cả các dòng thu được ta nhận được dòng kết quả. Chẳng hạn:

$$[5, 0, 1, 3]* \begin{bmatrix} 0 & 3 & 2 \\ 1 & 0 & 4 \\ 0 & 6 & 0 \\ 2 & 0 & 0 \end{bmatrix} = 5* [0\ \ 3\ \ 2] + 0* [1\ \ 0\ \ 4] + 1* [0\ \ 6\ \ 0] + 3* [2\ \ 0\ \ 0]$$

$$= [6\ \ 21\ \ 10].$$

Đến đây, chúng ta có thể định nghĩa lớp ma trận thưa SpMatrix. Lớp này chứa ba thành phần dữ liệu: số dòng (row), số cột (col) và danh sách các dòng của ma trận (rowList)

```
template <class  Item>
class    SpMatrix
{
public:
     // Hàm kiến tạo và các hàm cần thiết khác.
     // Các hàm toán tử cho các phép toán ma trận.
private:
     int   row; // Số dòng.
     int   col; // Số cột.
     Dlist<MaRow<Elem<Item>>>     rowList;
     // Danh sách các dòng của ma trận.
};
```

Cài đặt chi tiết các lớp MaRow, SpMatrix để lại cho độc giả xem như bài tập.

BÀI TẬP

1. Trong lớp Dlist, hãy cài đặt hàm truy cập tới phần tử thứ i trong danh sách (hàm Element(i)) bởi hàm toán tử Operator [ ].

2. Hãy cài đặt hàm Advance( ) trong lớp DlistIterator bởi hàm toán tử Operator + +.

3. Giả sử trong lớp Dset, tập động được cài đặt bởi danh sách được sắp theo giá trị khoá tăng dần. Hãy cài đặt hàm Search(k) bởi hàm đệ quy theo kỹ thuật tìm kiếm nhị phân.

4. Viết chương trình sử dụng danh sách các số nguyên L (L là đối tượng của lớp Dlist). Chương trình cần thực hiện các nhiệm vụ sau:
    a. Tạo ra danh sách các số nguyên được đưa vào từ bàn phím.
    b. Loại khỏi danh sách tất cả các số nguyên bằng một số nguyên cho trước.
    c. Thêm vào danh sách số nguyên n sau số nguyên m xuất hiện đầu tiên trong danh sách, nếu m không có trong danh sách thì thêm m vào đuôi danh sách.
    d. In ra các số nguyên trong danh sách.

5. Cho hai danh sách số nguyên L_1 và L_2 là đối tượng của lớp Dlist. Hãy viết hàm toán tử Operator + thực hiện nối hai danh sách đó thành một danh sách, cần giữ nguyên thứ tự của các phần tử và các phần tử của L_2 đi sau các phần tử của L_1.