CHƯƠNG 6

NGĂN XẾP

Trong chương này, chúng ta sẽ trình bày KDLTT ngăn xếp. Cũng giống như danh sách, ngăn xếp là CTDL tuyến tính, nó gồm các đối tượng dữ liệu được sắp thứ tự. Nhưng đối với danh sách, các phép toán xen, loại và truy cập có thể thực hiện ở vị trí bất kỳ của danh sách, còn đối với ngăn xếp các phép toán đó chỉ được thực hiện ở một đầu. Mặc dù các phép toán trên ngăn xếp là rất đơn giản, song ngăn xếp là một trong các CTDL quan trọng nhất. Trong chương này chúng ta sẽ đặc tả KDLTT ngăn xếp, sau đó sẽ nghiên cứu các phương pháp cài đặt ngăn xếp. Cuối cùng chúng ta sẽ trình bày một số ứng dụng của ngăn xếp.

6.1 KIỂU DỮ LIỆU TRỪU TƯỢNG NGĂN XẾP

Chúng ta có thể xem một chồng sách là một ngăn xếp. Trong chồng sách, các quyển sách đã được sắp xếp theo thứ tự trên - dưới, quyển sách nằm trên cùng được xem là ở đỉnh của chồng sách. Chúng ta có thể dễ dàng đặt một quyển sách mới lên đỉnh chồng sách và lấy quyển sách ở đỉnh ra khỏi chồng sách. Và như thế quyển sách được lấy ra khỏi chồng là quyển sách được đặt vào chồng sau cùng.

Ngăn xếp (**stack** hoặc đôi khi **pushdown store**) là một cấu trúc dữ liệu bao gồm các đối tượng dữ liệu được sắp xếp theo thứ tự tuyến tính, một trong hai đầu được gọi là **đỉnh** của ngăn xếp. Chúng ta chỉ có thể truy cập tới đối tượng dữ liệu ở đỉnh của ngăn xếp, thêm một đối tượng dữ liệu mới vào đỉnh của ngăn xếp và loại đối tượng dữ liệu ở đỉnh ra khỏi ngăn xếp.

Sau đây chúng ta sẽ đặc tả chính xác hơn các phép toán ngăn xếp. Chúng ta sẽ mô tả các phép toán ngăn xếp bởi các hàm, trong đó S ký hiệu một ngăn xếp và x là một đối tượng dữ liệu cùng kiểu với các đối tượng trong ngăn xếp S.

Các phép toán ngăn xếp:

1. Empty(S): Hàm trả về true nếu ngăn xếp S rỗng và false nếu ngược lại.
2. Push(S,x): Đẩy x vào đỉnh của ngăn xếp S. Chẳng hạn, S là ngăn xếp các số nguyên, S = [5, 3, 6, 4, 7], và đầu bên phải là đỉnh ngăn xếp, tức là 7 ở đỉnh của ngăn xếp. Nếu chúng ta đẩy số nguyên x = 2 vào đỉnh ngăn xếp, thì ngăn xếp trở thành S = [5, 3, 6, 4, 7, 2].
3. Pop(S): Loại đối tượng ở đỉnh của ngăn xếp S. Ví dụ, nếu S là ngăn xếp S = [5, 3, 6, 4, 7], thì khi loại phần tử ở đỉnh ngăn xếp, ngăn xếp trở thành S = [5, 3, 6, 4].

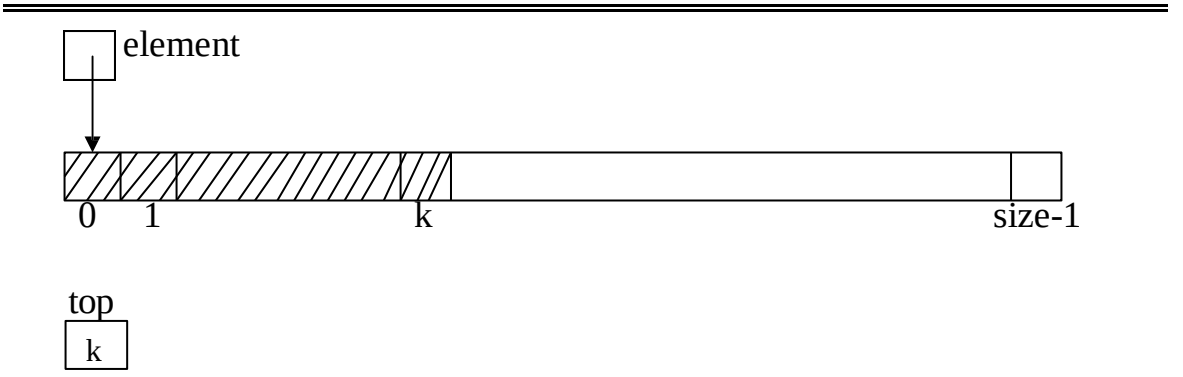
4. GetTop(S): Hàm trả về đối tượng ở đỉnh của S, ngăn xếp S không thay đổi.

Ngăn xếp được gọi là cấu trúc dữ liệu LIFO (viết tắt của cụm từ Last-In- First- Out), có nghĩa là đối tượng sau cùng được đưa vào ngăn xếp sẽ là đối tượng đầu tiên được lấy ra khỏi ngăn xếp.

Cũng giống như danh sách, chúng ta có thể cài đặt ngăn xếp bởi mảng hoặc bởi DSLK. Sau đây chúng ta sẽ nghiên cứu mỗi cách cài đặt đó.

6.2 CÀI ĐẶT NGĂN XẾP BỞI MẢNG

Chúng ta sử dụng một mảng element (mảng tĩnh hoặc động) để lưu giữ các đối tượng dữ liệu của ngăn xếp. Các thành phần mảng element[0], element[1], …, element[k] sẽ lần lượt lưu giữ các đối tượng của ngăn xếp. Đỉnh của ngăn xếp được lưu ở element[0] hay element[k] ? Nếu đỉnh của ngăn xếp ở element[0], thì khi thực hiện phép toán Push (Pop) chúng ta sẽ phải đẩy (dồn) các đối tượng được lưu trong đoạn mảng element[1…k] ra sau (lên trên) một vị trí. Do đó, hợp lý hơn, chúng ta chọn đỉnh ngăn xếp ở vị trí k trong mảng. Hình 6.1 mô tả cấu trúc dữ liệu cài đặt ngăn xếp sử dụng một mảng động element.



Hình 6.1. Ngăn xếp cài đặt bởi mảng động

Nếu cài đặt ngăn xếp bởi mảng như trên, thì các phép toán ngăn xếp là các trường hợp riêng của các phép toán trên danh sách: việc đẩy đối tượng x vào đỉnh của ngăn xếp là xen x vào đuôi danh sách (phép toán Append trên danh sách), còn việc loại (hoặc truy cập) phần tử ở đỉnh của ngăn xếp là loại (truy cập) phần tử ở vị trí thứ k + 1 của danh sách. Do đó chúng ta có thể sử dụng lớp DList (xem mục 4.3) để cài đặt lớp ngăn xếp Stack. Sẽ có hai cách lựa chọn:

- Xây dựng lớp Stack là lớp dẫn xuất từ lớp cơ sở DList (tương tự như chúng ta xây dựng lớp tập động DSet sử dụng lớp DList như lớp cơ sở private, xem mục 4.4).

- Thay cho sử dụng lớp DList làm lớp cơ sở private, chúng ta có thể thiết kế lớp Stack một cách khác: lớp Stack chứa một thành phần dữ liệu là đối tượng của lớp DList.

Cả hai cách trên cho phép ta sử dụng các hàm thành phần của lớp DList (các hàm Append, Delete, Element) để cài đặt các hàm thực hiện các phép toán ngăn xếp. Cài đặt lớp Stack theo các phương án trên để lại cho độc giả, xem như bài tập.

Bởi vì các phép toán ngăn xếp là rất đơn giản, cho nên chúng ta sẽ cài đặt lớp Stack trực tiếp, không sử dụng lớp DList.

Lớp Stack được thiết kế sau đây là lớp khuôn phụ thuộc tham biến kiểu Item, Item là kiểu của các phần tử trong ngăn xếp. Lớp Stack chứa ba thành phần dữ liệu: biến con trỏ element trỏ tới mảng được cấp phát động để lưu các phần tử của ngăn xếp, biến size lưu cỡ của mảng động, và biến top lưu chỉ số mảng là đỉnh ngăn xếp. Định nghĩa lớp Stack được cho trong hình 6.2.

```
template <class Item>
class    Stack
{
  public :
    Stack (int m = 1);
    //khởi tạo ngăn xếp rỗng với dung lượng m, m là số nguyên dương
    Stack (const Stack & S) ;
    // Hàm kiến tạo copy.
    ~ Stack( ) ; // Hàm huỷ.
    Stack &  operator = (const Stack & S);
    // Toán tử gán.
    // Các phép toán ngăn xếp:
    bool   Empty( )  const;
    // Xác định ngăn xếp có rỗng không.
    // Postcondition : Hàm trả về true nếu ngăn xếp rỗng, và trả về
    // false nếu không.
    void   Push(const Item & x);
    // Đẩy phần tử x vào đỉnh của ngăn xếp.
    // Postcondition: phần tử x trở thành đỉnh của ngăn xếp.
    Item &   Pop( );
    // Loại phần tử ở đỉnh của ngăn xếp.
    // Precondition: ngăn xếp không rỗng.
    // Postcondition: phần tử ở đỉnh ngăn xếp bị loại khỏi ngăn xếp,
    // và hàm trả về phần tử này.
    Item &   GetTop( )  const ;
    // Truy cập đỉnh ngăn xếp.
```

```
    // Precondition : ngăn xếp không rỗng.
    // Postcondition: Hàm trả về phần tử ở đỉnh ngăn xếp, ngăn xếp
    // không thay đổi.
  private:
    Item *   element ;
    int    size ;
    int    top ;
} ;
```

Hình 6.2. Lớp Stack.

Sau đây chúng ta cài đặt các hàm thành phần của lớp Stack. Trước hết nói về hàm kiến tạo Stack(int m), hàm này làm nhiệm vụ cấp phát một mảng động có cỡ m để lưu các phần tử của ngăn xếp, vì ngăn xếp rỗng mảng không chứa phần tử nào cả, biến top được đặt bằng –1.

```
template <class   Item>
Stack<Item>   Stack(int m)
{
  element = new Item[m] ;
  size = m ;
  top = -1 ;
}
```

Các hàm kiến tạo copy, hàm huỷ, toán tử gán được cài đặt tương tự như trong lớp DList (xem mục 4.3). Các hàm thực hiện các phép toán ngăn xếp được cài đặt rất đơn giản như sau:

```
template <class  Item>
bool    Stack<Item> :: Empty( )
{
 return    top = = -1 ;
}

template <class Item>
void   Stack<Item> : : Push(const Item & x)
{
 if  (top = = size –1) // mảng đầy
 {
     Item * A = new Item[2 * size] ;
     Assert (A! = NULL) ;
     for (int i = 0 ; i < = top ; i + +)
         A[i] = element[i] ;
```

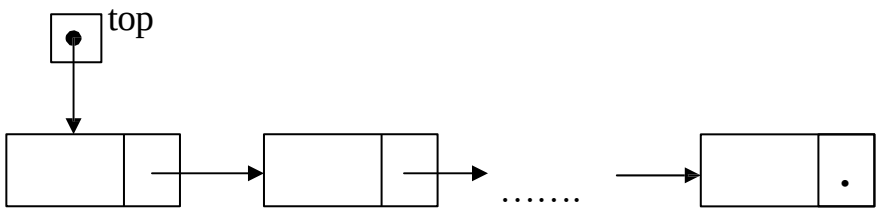
```
        size = 2 * size ;
        delete [ ] element ;
        element = A ;
    } ;
    element [+ + top] = x ;
}

template <class Item>
Item & Stack<Item> : : Pop( )
{
    assert (top > = 0) ;
    return   element[top - -] ;
}

template <class Item>
Item & Stack<Item> : : GetTop( )
{
    assert (top >= 0) ;
    return    element[top];
}
```

Chúng ta có nhận xét rằng, tất cả các phép toán ngăn xếp chỉ đòi hỏi thời gian O(1), trừ khi chúng ta thực hiện phép toán Push và mảng đã đầy. Khi đó chúng ta phải cấp phát một mảng động mới với cỡ gấp đôi mảng cũ và sao chép dữ liệu từ mảng cũ sang mảng mới. Do đó trong trường hợp mảng đầy, phép toán Push đòi hỏi thời gian O(n), với n là số phần tử trong ngăn xếp.

6.3 CÀI ĐẶT NGĂN XẾP BỞI DSLK

Ngăn xếp cũng có thể cài đặt bởi DSLK, và chúng ta có thể sử dụng lớp LList (xem mục 5.4) làm lớp cơ sở private để xây dựng lớp Stack. Tuy nhiên, khi lưu giữ các phần tử của ngăn xếp trong các thành phần của DSLK thì các phép toán ngăn xếp cũng được thực hiện rất đơn giản. Do đó chúng ta sẽ cài đặt trực tiếp lớp Stack, không sử dụng tới lớp LList.

Chúng ta sẽ cài đặt ngăn xếp bởi DSLK, đỉnh của ngăn xếp được lưu trong thành phần đầu tiên của DSLK, một con trỏ ngoài top trỏ tới thành phần này, xem hình 6.3.



Hình 6.3. Cài đặt ngăn xếp bởi DSLK.

Lớp Stack cài đặt KDLTT ngăn xếp sử dụng DSLK được mô tả trong hình 6.4. Lớp Stack này chỉ chứa một thành phần dữ liệu là con trỏ top trỏ tới thành phần đầu tiên của DSLK (như trong hình 6.3). Các thành phần của DSLK là cấu trúc Node gồm biến data có kiểu Item (Item là kiểu của các phần tử trong ngăn xếp), và biến con trỏ next trỏ tới thành phần đi sau. Lớp Stack ở đây chứa các hàm thành phần với khai báo và chú thích giống hệt như các hàm thành phần trong lớp Stack ở mục 6.2 (xem hình 6.2), trừ hàm kiến tạo.

```
template  <class Item>
class     Stack
{
  public :
     Stack(  ) // Hàm kiến tạo mặc định khởi tạo ngăn xếp rỗng.
     { top = NULL; }
     Stack (const  Stack & S) ;
     ~ Stack(  ) ;
     Stack &   operator = (const Stack & S) ;
     bool    Empty( )   const
             { return  top = = NULL; }
     void    Push (const Item & x) ;
     Item &    Pop( ) ;
     Item &    GetTop( )   const ;
  private :
     struct  Node
     {
        Item    data ;
        Node *   next;
        Node (const Item & x)
        : data(x), next (NULL) {  }
     } ;
```

```
    Node *   top ;
} ;
```

Hình 6.4. Lớp Stack cài đặt bởi DSLK.

Bây giờ chúng ta cài đặt các hàm thành phần của lớp Stack. Hàm kiến tạo mặc định nhằm khởi tạo nên một ngăn xếp rỗng, muốn vậy chỉ cần đặt giá trị của con trỏ top là hằng NULL. Hàm này đã được cài đặt inline.

Hàm kiến tạo copy. Cho trước một ngăn xếp L, công việc của hàm kiến tạo copy là tạo ra một ngăn xếp mới là bản sao của L. Cụ thể hơn, phải tạo ra một DSLK với con trỏ ngoài top là bản sao của DSLK với con trỏ ngoài L.top. Nếu ngăn xếp L không rỗng, đầu tiên ta khởi tạo ra DSLK top chỉ có một thành phần chứa dữ liệu như trong thành phần đầu tiên của DSLK L.top, tức là:

top = new Node (L.top → data) ;

Sau đó, ta nối dài DSLK top bằng cách thêm vào các thành phần chứa dữ liệu như trong các thành phần tương ứng ở DSLK L.top. Điều đó được thực hiện bởi vòng lặp với con trỏ P chạy trên DSLK L.top và con trỏ Q chạy trên DSLK top. Ở mỗi bước lặp, cần thực hiện:

Q = Q → next = new Node (P → data) ;

Hàm kiến tạo copy được viết như sau:

```
template <class Item>
Stack <Item> : : Stack (const Stack & S)
{
  if (S.Empty( ))
          top = NULL ;
  else {
          Node * P = S.top ;
          top = new Node (P → data) ;
          Node * Q = top ;
          for  (P = P → next ; P! = NULL ; P = P → next)
                Q = Q → next = new Node (P →data) ;
        }
}
```

Hàm huỷ. Hàm cần thực hiện nhiệm vụ thu hồi bộ nhớ đã cấp phát cho từng thành phần của DSLK, lần lượt từ thành phần đầu tiên.

```
template <class Item>
Stack<Item> : : ~ Stack( )
{
  if (top! = NULL)
```

```
    {
        Node * P = top ;
        while (top ! = NULL)
        {
          P = top ;
          top = top → next ;
          delete P ;
        }
    }
}
```

Toán tử gán. Hàm này gồm hai phần. Đầu tiên ta huỷ DSLK top với các dòng lệnh như trong hàm ~Stack( ), sau đó tạo ra DSLK top là bản sao của DSLK S.top với các dòng lệnh như trong hàm kiến tạo copy.

Các hàm thực hiện các phép toán ngăn xếp được cài đặt rất đơn giản: việc đẩy một phần tử mới x vào đỉnh ngăn xếp chẳng qua là xen một thành phần chứa dữ liệu x vào đầu DSLK, còn việc loại (truy cập) phần tử ở đỉnh ngăn xếp đơn giản là loại (truy cập) thành phần đầu tiên của DSLK. Các phép toán ngăn xếp được cài đặt như sau:

```
template  <class Item>
void    Stack<Item> : : Push (const Item & x)
{
  Node* P = new Node(x) ;
  if  (top = = NULL)
          top = P ;
  else  {
          P → next = top ;
          top = P ;
        }
}


template  <class Item>
Item &   Stack<Item> : : Pop( )
{
   assert (top! = NULL) ;
   Item object = top → data ;
   Node* P = top ;
   top = top → next ;
   delete P ;
   return   object ;
}
```

```
template <class Item>
Item &  Stack<Item> : : GetTop( )
{
  assert( top! = NULL) ;
  return   top → data ;
}
```

Rõ ràng là khi cài đặt ngăn xếp bởi DSLK thì các phép toán ngăn xếp Push, Pop, GetTop chỉ cần thời gian O(1).

Sau đây chúng ta sẽ trình bày một số ứng dụng của ngăn xếp.

6.4 BIỂU THỨC DẤU NGOẶC CÂN XỨNG

Ngăn xếp được sử dụng nhiều trong các chương trình dịch (compiler). Trong mục này chúng ta sẽ trình bày vấn đề: sử dụng ngăn xếp để kiểm tra tính cân xứng của các dấu ngoặc trong chương trình nguồn.

Trong chương trình ở dạng mã nguồn, chẳng hạn chương trình viết bằng ngôn ngữ C + +, chúng ta sử dụng nhiều các dấu mở ngoặc "{" và đóng ngoặc "}", mỗi dấu "{" cần phải có một dấu "}" tương ứng đi sau. Tương tự, trong các biểu thức số học (hoặc logic) chúng ta cũng sử dụng các dấu mở ngoặc "(" và đóng ngoặc ")", mỗi dấu "(" cần phải tương ứng với một dấu ")". Nếu chúng ta loại bỏ tất cả các ký hiệu khác, chỉ giữ lại các dấu mở ngoặc và đóng ngoặc thì chúng ta sẽ có một dãy các dấu mở ngoặc và đóng ngoặc mà ta gọi là **biểu thức dấu ngoặc** và nó cần phải cân xứng. Độc giả có thể đưa ra định nghĩa chính xác thế nào là biểu thức dấu ngoặc cân xứng.

Để minh hoạ, ta xét biểu thức số học (((

a – b) * (5 + c) + x) / (x + y))

Loại bỏ đi tất cả các toán hạng và các dấu phép toán ta nhận được biểu thức dấu ngoặc:

```
( (  ( )  ( ) )  ( ) )
1 2 3 4  5 6 7 8 9 10
```

Biểu thức dấu ngoặc trên là cân xứng: các cặp dấu ngoặc tương ứng là 1 – 10, 2 – 7, 3 – 4, 5 – 6 và 8 – 9. Biểu thức dấu ngoặc (( ) ( ) là không cân xứng. Vấn đề đặt ra là làm thế nào để cho biết một biểu thức dấu ngoặc là cân xứng hay không cân xứng.

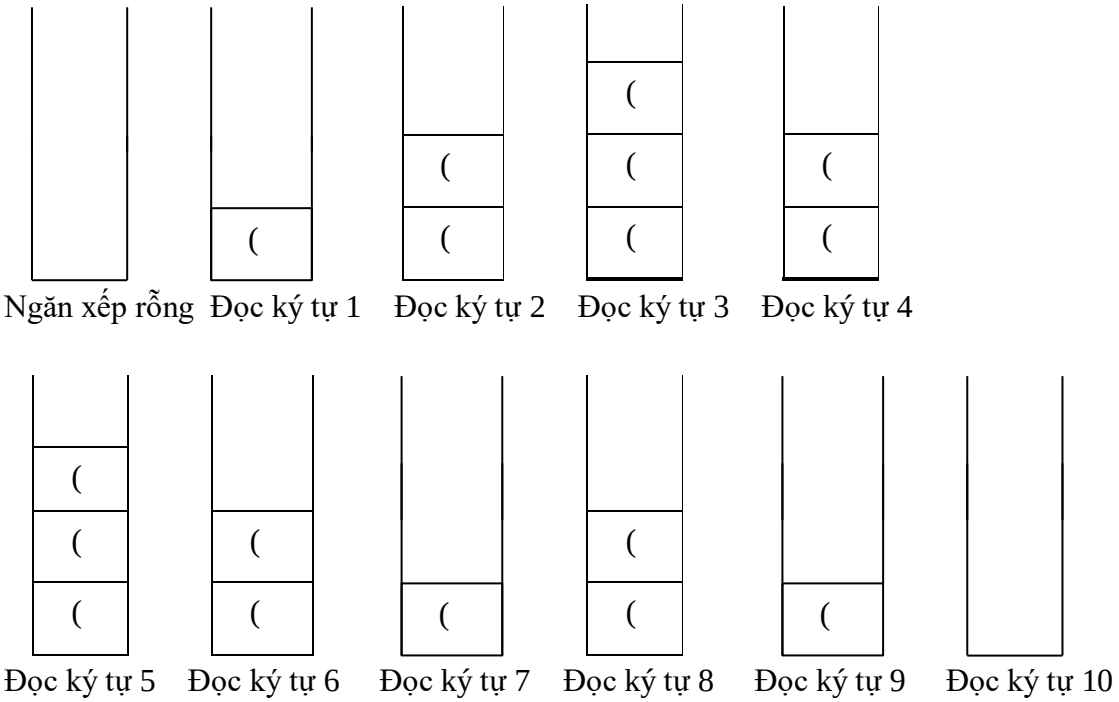
Sử dụng ngăn xếp, chúng ta dễ dàng thiết kế được thuật toán kiểm tra tính cân xứng của biểu thức dấu ngoặc. Một biểu thức dấu ngoặc được xem như một xâu ký tự được tạo thành từ hai ký tự mở ngoặc và đóng ngoặc. Ngăn xếp được sử dụng để lưu các dấu mở ngoặc. Thuật toán gồm các bước sau:

1. Khởi tạo một ngăn xếp rỗng.
2. Đọc lần lượt các ký tự trong biểu thức dấu ngoặc
    a. Nếu ký tự là dấu mở ngoặc thì đẩy nó vào ngăn xếp.
    b. Nếu ký tự là dấu đóng ngoặc thì:

- Nếu ngăn xếp rỗng thì thông báo biểu thức dấu ngoặc không cân xứng và dừng.
- Nếu ngăn xếp không rỗng thì loại dấu mở ngoặc ở đỉnh ngăn xếp.

3. Sau khi ký tự cuối cùng trong biểu thức dấu ngoặc đã được đọc, nếu ngăn xếp rỗng thì thông báo biểu thức dấu ngoặc cân xứng.

Để thấy được thuật toán trên làm việc như thế nào, ta xét biểu thức dấu ngoặc đã đưa ra ở trên: ((( ) ( )) ( )). Biểu thức dấu ngoặc này là một xâu gồm 10 ký tự. Ban đầu ngăn xếp rỗng. Đọc ký tự đầu tiên, nó là dấu mở ngoặc và được đẩy vào ngăn xếp. Ký tự thứ hai và ba cũng là dấu mở ngoặc, nên cũng được đẩy vào ngăn xếp, và như vậy đến đây ngăn xếp chứa ba dấu mở ngoặc. Ký tự thứ tư là dấu đóng ngoặc, do đó dấu mở ngoặc ở đỉnh ngăn xếp bị loại. Ký tự thứ năm là dấu mở ngoặc, nó lại được đẩy vào ngăn xếp. Tiếp tục, sau khi ký tự cuối cùng được đọc, ta thấy ngăn xếp rỗng, do đó biểu thức dấu ngoặc đã xét là cân xứng. Hình 6.4 minh hoạ các trạng thái của ngăn xếp tương ứng với mỗi ký tự được đọc.



Hình 6.4. Các trạng thái của ngăn xếp khi đọc biểu thức ((( ) ( )) ( ))

6.5 ĐÁNH GIÁ BIỂU THỨC SỐ HỌC

Trong các chương trình viết bằng ngôn ngữ bậc cao, chẳng hạn Pascal, C/ C + +, thường có mặt các biểu thức số học (hoặc logic). Khi chương trình được thực hiện, các biểu thức sẽ được đánh giá. Trong mục này chúng ta sẽ trình bày thuật toán tính giá trị của biểu thức, thuật toán này được cài đặt trong module thực hiện nhiệm vụ đánh giá biểu thức trong các chương trình dịch.

Biểu thức dạng postfix (biểu thức Balan). Để đơn giản cho trình bày, chúng ta giả thiết rằng biểu thức chỉ chứa các phép toán có hai toán hạng, chẳng hạn các phép toán +, -, *, /, … Trong các chương trình, các biểu thức được viết theo cách truyền thống như trong toán học, tức là ký hiệu phép toán đứng giữa hai toán hạng, chẳng hạn biểu thức:

(a + b) * c

Các biểu thức được viết theo cách truyền thống được gọi là **biểu thức dạng infix**. Trong các biểu thức dạng infix có thể có các dấu ngoặc, chẳng hạn biểu thức trên. Các dấu ngoặc được đưa vào biểu thức để thay đổi thứ tự tính toán. Các biểu thức còn có thể được viết dưới một dạng khác: **dạng postfix**. Trong các biểu thức dạng postfix (biểu thức Balan) ký hiệu phép toán đứng sau hai toán hạng, chẳng hạn dạng postfix của biểu thức (a + b) * c là biểu thức:

a b + c *

Cần lưu ý rằng, trong các biểu thức postfix không có các dấu ngoặc. Sau đây chúng ta sẽ thấy việc đánh giá các biểu thức postfix là rất dễ dàng. Vì vậy, việc đánh giá các biểu thức infix được thực hiện qua hai giai đoạn:

- Chuyển biểu thức infix thành biểu thức postfix.
- Đánh giá biểu thức postfix.

Trước hết chúng ta trình bày thuật toán đánh giá biểu thức postfix.

6.5.1 Đánh giá biểu thức postfix

Trong biểu thức postfix, các phép toán được thực hiện theo thứ tự mà chúng xuất hiện trong biểu thức kể từ trái sang phải. Chẳng hạn, xét biểu thức postfix:

5 8 3 1 - / 3 * + (1)

Phép toán được thực hiện đầu tiên là -, rồi đến /, tiếp theo là * và sau cùng là +. Mỗi phép toán được thực hiện với hai toán hạng đứng ngay trước nó. Chẳng hạn, phép – được thực hiện với các toán hạng là 1 và 3: 3 – 1 = 2. Sau khi thực hiện phép - biểu thức (1) trở thành 5 8 2 / 3 * +. Phép / được thực hiện với hai toán hạng đứng ngay trước nó là 2 và 8: 8 / 2 = 4. Tiếp tục, ta có 5 8 2 / 3 * + = 5 4 3 * + = 5 12 + = 17. Như vậy, mỗi khi thực hiện một phép toán trong biểu thức postfix chúng ta cần có khả năng tìm được các toán hạng của nó. Cách dễ nhất để làm được điều đó là sử dụng một ngăn xếp.

Chúng ta sẽ sử dụng một ngăn xếp S để lưu các toán hạng, ban đầu ngăn xếp rỗng.

Thuật toán đánh giá biểu thức postfix là như sau. Xem biểu thức postfix như một dãy các thành phần (mỗi thành phần hoặc là toán hạng, hoặc là ký hiệu phép toán). Đọc lần lượt các thành phần của biểu thức từ trái sang phải, với mỗi thành phần được đọc thực hiện các bước sau:

1. Nếu thành phần được đọc là toán hạng od thì đẩy nó vào ngăn xếp, tức là S. Push (od).
2. Nếu thành phần được đọc là phép toán op thì lấy ra các toán hạng ở đỉnh ngăn xếp:

   od 2 = S.Pop( )

   od 1 = S.Pop( )

   Thực hiện phép toán op với các toán hạng là od 1 và od 2, kết quả được đẩy vào ngăn xếp:

   r = od 1 op od 2

   S. Push(r)

   Lặp lại hai bước trên cho tới khi thành phần cuối cùng của biểu thức được đọc qua. Khi đó ngăn xếp chứa kết quả của biểu thức.

   Để thấy được thuật toán trên làm việc như thế nào, chúng ta lại xét biểu thức postfix (1). Khi bốn toán hạng đầu tiên được đọc, chúng được đẩy vào ngăn xếp và ngăn xếp có nội dung như sau:

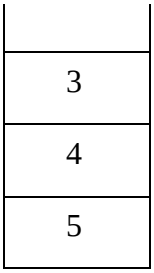
1
3
8
5

Tiếp theo, thành phần được đọc là phép " – ", hai toán hạng ở đỉnh của ngăn xếp là 1 và 3 được lấy ra khỏi ngăn xếp và thực hiện phép " – " với các toán hạng đó, ta thu được kết quả là 2, đẩy 2 vào ngăn xếp:

2
8
5

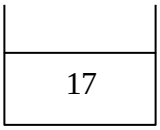
Thành phần được đọc tiếp theo là phép toán /, do đó các toán hạng 2 và 8 được lấy ra khỏi ngăn xếp, và thương của chúng là 4 lại được đẩy vào ngăn xếp:

Toán hạng tiếp theo 3 được đẩy vào ngăn xếp:



Thành phần tiếp theo là phép x, do đó 3 và 4 được lấy ra khỏi ngăn xếp, và 4 * 3 = 12 được đẩy vào ngăn xếp:

Cuối cùng là phép + được đọc, nên 12 và 5 được lấy ra khỏi ngăn xếp và kết quả của biểu thức là 5 + 12 = 17 được đẩy vào ngăn xếp:



6.5.2 Chuyển biểu thức infix thành biểu thức postfix

Để chuyển biểu thức infix thành biểu thức postfix, trước hết chúng ta cần chú ý tới thứ tự thực hiện các phép toán trong biểu thức infix. Nhớ lại rằng, trong biểu thức infix, các phép toán * và / cần được thực hiện trước các phép toán + và - , chẳng hạn 1 + 2 * 3 = 1 + 6 = 7. Dó đó chúng ta nói rằng các phép toán * và / có quyền ưu tiên cao hơn các phép toán + và -, các phép toán * và / cùng mức ưu tiên, các phép toán + và – cùng mức ưu tiên. Mặt khác nếu các phép toán cùng mức ưu tiên đứng kề nhau, thì thứ tự thực hiện chúng là từ trái qua phải, chẳng hạn 8 – 2 + 3 = 6 + 3 = 9. Các cặp dấu ngoặc

được đưa vào biểu thức infix để thay đổi thứ tự thực hiện các phép toán, các phép toán trong cặp dấu ngoặc cần được thực hiện trước, chẳng hạn 8 – (2 + 3) = 8 – 5 = 3. Vì vậy vấn đề then chốt trong việc chuyển biểu thức infix thành biểu thức postfix là xác định được thứ tự thực hiện các phép toán trong biểu thức infix là đặt các phép toán đó vào đúng vị trí của chúng trong biểu thức postfix.

Ý tưởng của thuật toán chuyển biểu thức infix thành biểu thức postfix là như sau: Chúng ta xem biểu thức infix như dãy các thành phần (mỗi thành phần (mỗi thành phần là toán hạng hoặc ký hiệu phép toán hoặc dấu mở ngoặc hoặc dấu đóng ngoặc). Thuật toán có đầu vào là biểu thức infix đã cho và đầu ra là biểu thức postfix. Chúng ta đã sử dụng một ngăn xếp S để lưu các phép toán và dấu mở ngoặc. Đọc lần lượt từng thành phần của biểu thức infix từ trái sang phải, cứ mỗi lần gặp toán hạng, chúng ta viết nó vào biểu thức postfix. Khi đọc tới một phép toán (phép toán hiện thời), chúng ta xét các phép toán ở đỉnh ngăn xếp. Nếu các phép toán ở đỉnh ngăn xếp có quyền ưu tiên cao hơn hoặc bằng phép toán hiện thời, thì chúng được kéo ra khỏi ngăn xếp và được viết vào biểu thức postfix. Khi mà phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời, thì phép toán hiện thời được đẩy vào ngăn xếp. Bởi vì các phép toán trong cặp dấu ngoặc cần được thực hiện trước tiên, do đó chúng ta xem dấu mở ngoặc như phép toán có quyền ưu tiên cao hơn mọi phép toán khác. Vì vậy, mỗi khi gặp dấu mở ngoặc thì nó được đẩy vào ngăn xếp, và nó chỉ bị loại khỏi ngăn xếp khi ta đọc tới dấu đóng ngoặc tương ứng và tất cả các phép toán trong cặp dấu ngoặc đó đã được viết vào biểu thức postfix.

1. Nếu thành phần được đọc là toán hạng thì viết nó vào biểu thức postfix.
2. Nếu thành phần được đọc là phép toán (phép toán hiện thời), thì thực hiện các bước sau:

Nếu ngăn xếp không rỗng thì nếu phần tử ở đỉnh ngăn xếp là phép toán có quyền ưu tiên cao hơn hay bằng phép toán hiện thời, thì phép toán đó được kéo ra khỏi ngăn xếp và viết vào biểu thức postfix. Lặp lại bước này.

Nếu ngăn xếp rỗng hoặc phần tử ở đỉnh ngăn xếp là dấu mở ngoặc hoặc phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời, thì phép toán hiện thời được đẩy vào ngăn xếp.

3. Nếu thành phần được đọc là dấu mở ngoặc thì nó được đẩy vào ngăn xếp.
4. Nếu thành phần được đọc là dấu đóng ngoặc thì thực hiện các bước sau:

(Bước lặp) Loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức postfix cho tới khi đỉnh ngăn xếp là dấu mở ngoặc.

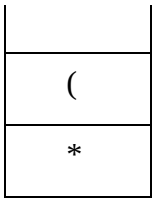
Loại dấu mở ngoặc khỏi ngăn xếp.

5. Sau khi toàn bộ biểu thức infix được đọc, loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức postfix cho tới khi ngăn xếp rỗng.

Ví dụ. Xét biểu thức infix:

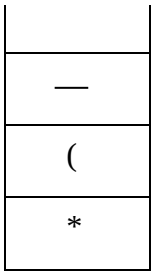
a * (b – c + d) + e / f

Đầu tiên ký hiệu được đọc là toán hạng a, nó được viết vào biểu thức postfix. Tiếp theo ký hiệu phép toán * được đọc, và ngăn xếp rỗng, nên nó được đẩy vào ngăn xếp. Đọc thành phần tiếp theo, đó là dấu mở ngoặc nên nó được đẩy vào ngăn xếp. Tới đây ta có trạng thái ngăn xếp và biểu thức postfix như sau:



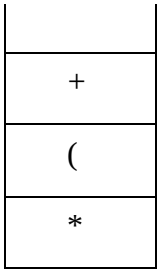
Biểu thức postfix: a

Đọc tiếp toán hạng b, nó được viết vào biểu thức postfix. Thành phần tiếp theo là phép -, lúc này đỉnh ngăn xếp là dấu mở ngoặc, nên ký hiệu - được đẩy vào ngăn xếp. Ta có:



Biểu thức postfix: a b

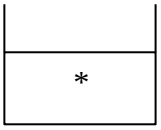
Thành phần được đọc tiếp theo là toán hạng c, nó được viết vào biểu thức postfix. Ký hiệu được đọc tiếp là phép +. Phép toán ở đỉnh ngăn xếp là phép – cùng quyền ưu tiên với phép +, nên nó được kéo ra khỏi ngăn xếp và viết vào đầu ra. Lúc này, đỉnh ngăn xếp là dấu mở ngoặc, nên phép + được đẩy vào ngăn xếp và ta có:



Biểu thức postfix
a b c -

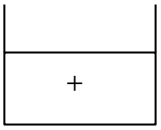
Đọc đến toán hạng d, nó được viết vào biểu thức postfix. Đọc tiếp ký hiệu tiếp theo: dấu đóng ngoặc. Lúc này ta cần loại lần lượt các phép toán ở đỉnh

ngăn xếp và viết chúng vào biểu thức postfix, cho tới khi đỉnh ngăn xếp là dấu mở ngoặc thì loại nó. Ta có:



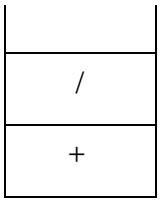
Biểu thức postfix
a b c – d +

Ký hiệu được đọc tiếp theo là phép +. Đỉnh ngăn xếp là phép *, nó có quyền ưu tiên cao hơn phép +, do đó nó được loại khỏi ngăn xếp và viết vào biểu thức postfix. Đến đây ngăn xếp rỗng, nên phép toán hiện thời + được đẩy vào ngăn xếp và ta có:



Biểu thức postfix
a b c – d + *

Đọc tiếp toán hạng e, nó được viết vào đầu ra. Ký hiệu được đọc tiếp là phép chia / , nó có quyền ưu tiên cao hơn phép + ở đỉnh ngăn xếp, nên phép / được đẩy vào ngăn xếp. Ký hiệu cuối cùng được đọc là toán hạng f, nó được viết vào đầu ra, ta có:



Biểu thức postfx
a b c – d + * e f

Loại các phép toán ở đỉnh ngăn xếp và viết vào đầu ra cho tới khi ngăn xếp rỗng, chúng ta nhận được biểu thức postfix kết quả là:

a b c – d + * e f / +

6.6 NGĂN XẾP VÀ ĐỆ QUY

Một trong các ứng dụng quan trọng của ngăn xếp là sử dụng ngăn xếp để chuyển thuật toán đệ quy thành thuật toán không đệ quy. Với nhiều vấn đề, có thể tồn tại cả thuật toán đệ quy và thuật toán không đệ quy để giải quyết. Trong nhiều trường hợp, thuật toán đệ quy kém hiệu quả cả về thời gian và bộ nhớ. Chúng ta sẽ nghiên cứu kỹ hơn các thuật toán đệ quy trong mục 16.2. Việc loại bỏ các lời gọi đệ quy trong các hàm đệ quy có thể cải thiện đáng kể thời gian chạy và bộ nhớ đòi hỏi. Trong mục này chúng ta sẽ trình bày các kỹ thuật sử dụng ngăn xếp để chuyển đổi các hàm đệ quy thành hàm không đệ quy.

Trước hết cần biết rằng, trong một hàm đệ quy thì lời gọi đệ quy ở dòng cuối cùng của hàm được gọi là lời gọi **đệ quy đuôi** (tail recursion). Lời

gọi đệ quy đuôi dễ dàng được thay thế bởi một vòng lặp, mà không cần sử dụng ngăn xếp. Chẳng hạn, xét hàm đệ quy sắp xếp mảng QuickSort, hàm này sẽ được nghiên cứu trong mục 17.3. Thuật toán sắp xếp nhanh được thiết kế theo chiến lược chia - để - trị: để sắp xếp mảng A[a..b] theo thứ tự tăng dần, chúng ta phân hoạch đoạn [a ..b] thành hai đoạn con bởi một chỉ số k, a $\leq$ k $\leq$ b, sao cho với mọi chỉ số i trong đoạn con dưới [a, k - 1] và với mọi chỉ số j trong đoạn con trên [k + 1….b] ta có A[i] $\leq$ A[k] $\leq$ A[j]. Nếu thực hiện được sự phân hoạch như thế, thì việc sắp xếp mảng A[a…b] được quy về việc sắp hai mảng con A[a... k - 1] và A[k+1 …b]. Do đó, hàm đệ quy QuickSort như sau:

```
void     QuickSort(item  A[ ], int a, int b)
{
  if (a < b)
  {
    k = Partition(A, a, b); // hàm phân hoạch
    QuickSort(A, a, k – 1) ;
    QuickSort(A, k+1, b); // lời gọi đệ quy đuôi
  }
}
```

Thực hiện lời gọi đệ quy đuôi (ở dòng lệnh cuối cùng) có nghĩa là chúng ta cần thực hiện lặp lại các dòng lệnh từ đầu hàm đến dòng lệnh đứng trên kề lời gọi đệ quy đuôi. Vì vậy đưa dãy các lệnh đó vào một vòng lặp, chúng ta loại bỏ được lời gọi đệ quy đuôi. Hàm QuickSort có thể viết lại như sau:

```
void    QuickSort(item  A[  ], int a, int b)
{
  k = Partition(A, a, b);
  while (a < b)
  {
    QuickSort(A, a, k – 1);
    a = k + 1;
    k = b + 1;
  }
}
```

Việc loại bỏ các lời gọi đệ quy không phải là đuôi không đơn giản như đệ quy đuôi. Chúng ta cần sử dụng ngăn xếp để lưu lại các đối số trong các lời gọi đệ quy. Phân tích sự thực hiện hàm đệ quy QuickSort, chúng ta sẽ thấy quá trình thực hiện hàm sẽ diễn ra như sau: phân hoạch đoạn [a …b] bởi chỉ số k, rồi phân hoạch đoạn con dưới [a ..k - 1] bởi chỉ số k_1, rồi lại phân hoạch đoạn con dưới [a .. k_1 - 1]… cho tới khi đoạn con dưới chỉ gồm một

chỉ số, lúc đó ta mới phân hoạch đoạn con trên ứng với đoạn con dưới đó. Việc phân hoạch một đoạn bất kỳ lại diễn ra như trên. Vì vậy, chúng ta sẽ sử dụng một ngăn xếp để lưu các chỉ số đầu và chỉ số cuối của các đoạn con trên khi phân hoạch. Cụ thể là ngăn xếp sẽ lưu k + 1, b, rồi k_1 + 1, k – 1, … Hàm đệ quy QuickSort được chuyển thành hàm không đệ quy sau:

```
void QuickSort2 (item  A[  ], int a, int b)
{
  Stack S;
  S.Push(a) ;
  S.Push(b) ;
  do
      b = S.Pop( ) ;
      a = S.Pop( ) ;
      while (a < b)
          {
              k = Partition(A, a, b) ;
              S.Push(k + 1) ;
              S.Push(b) ;
              b = k – 1 ;
           }
  while (! S.Empty( ) ) ;
}
```

Sau này chúng ta sẽ thấy, ngăn xếp được sử dụng để thiết kế các hàm không đệ quy thực hiện các phép toán trên cây.

BÀI TẬP

1. Hãy cài đặt lớp Stack bằng cách sử dụng lớp Dlist (hoặc lớp Llist) làm lớp cơ sở với dạng thừ kế private.

2. Cho ngăn xếp S chứa các phần tử. Sử dụng ngăn xếp T rỗng, hãy đưa ra thuật toán (chỉ được sử dụng các phép toán ngăn xếp) thực hiện các nhiệm vụ sau:
   a. Đếm số phần tử trong ngăn xếp S, ngăn xếp S không thay đổi.
   b. Loại bỏ tất cả các phần tử trong ngăn xếp S bằng một phần tử cho trước, thứ tự các phần tử còn lại trong S không thay đổi.

3. Giả sử biểu thức dấu ngoặc chứa ba loại dấu ngoặc ( ), [ ], { }. Biểu thức [( ) ( )] { } được xem là cân xứng, còn biểu thức {[( ]) ( ) là không cân xứng. Hãy đưa ra định nghĩa biểu thức dấu ngoặc cân xứng

và đưa ra thuật toán cho biết biểu thức dấu ngoặc có cân xứng hay không.

4. Áp dụng thuật toán chuyển biểu thức dạng infix thành biểu thức dạng postfix (xem mục 6.5.2), hãy chuyển các biểu thức infix sau thành biểu thức postfix, cần chỉ ra nội dung của ngăn xếp sau mỗi bước của thuật toán.
    a. A / B /C – (D + E) * F
    b. A – (B + C * D) / E
    c. A * (B / C / D) + E

5. Thiết kế thuật toán đoán nhận các xâu ký tự có dạng w $ w', trong đó w' là đảo ngược của xâu w, chẳng hạn nếu w = a c d b thì w' = b d c a.

6. Cho đỉnh A (đỉnh xuất phát) và đỉnh B (đỉnh đích) trong đồ thị định hướng G (đồ thị có thể có chu trình). Sử dụng ngăn xếp, hãy thiết kế thuật toán tìm đường đi từ A đến B. (Sử dụng ngăn xếp để lưu vết của đường đi từ A đến B).

7. Sử dụng các ngăn xếp, hãy đưa ra thuật toán không đệ quy cho bài toán tháp Hà Nội.