Chương 4 – DANH SÁCH

Chúng ta đã làm quen với các danh sách hạn chế như ngăn xếp và hàng, trong đó việc thêm/ bớt dữ liệu chỉ thực hiện ở các đầu của danh sách. Trong chương này chúng ta tìm hiểu các danh sách thông thường hơn mà trong đó việc thêm, loại hoặc truy xuất phần tử có thể thực hiện tại bất kỳ vị trí nào trong danh sách.

4.1. Định nghĩa danh sách

Chúng ta bắt đầu bằng việc định nghĩa kiểu cấu trúc dữ liệu trừu tượng gọi là danh sách (*list*). Cũng giống như ngăn xếp và hàng, danh sách bao gồm một chuỗi nối tiếp các phần tử dữ liệu. Tuy nhiên, khác với ngăn xếp và hàng, danh sách cho phép thao tác trên mọi phần tử.

Định nghĩa: Danh sách các phần tử kiểu T là một chuỗi nối tiếp hữu hạn các phần tử kiểu T cùng các tác vụ sau:

1. Tạo một danh sách rỗng.
2. Xác định danh sách có rỗng hay không.
3. Xác định danh sách có đầy hay chưa.
4. Tìm số phần tử của danh sách.
5. Làm rỗng danh sách.
6. Thêm phần tử vào một vị trí nào đó của danh sách.
7. Loại phần tử tại một vị trí nào đó của danh sách.
8. Truy xuất phần tử tại một vị trí nào đó của danh sách.
9. Thay thế phần tử tại một vị trí nào đó của danh sách.
10. Duyệt danh sách, thực hiện một công việc cho trước trên mỗi phần tử.

Ngoài ra còn một số tác vụ khác có thể áp lên một chuỗi nối tiếp các phần tử. Chúng ta có thể xây dựng rất nhiều dạng khác nhau cho các kiểu cấu trúc dữ liệu trừu tượng tương tự bằng cách sử dụng các gói tác vụ khác nhau. Bất kỳ một trong các dạng này đều có thể được định nghĩa cho tên gọi CTDL danh sách. Tuy nhiên, chúng ta chỉ tập trung vào một danh sách cụ thể mà các tác vụ của nó có thể được xem như một khuôn mẫu để minh họa ý tưởng và các vấn đề cần giải quyết trên danh sách.

4.2. Đặc tả các phương thức cho danh sách

Khi bắt đầu tìm hiểu ngăn xếp, chúng ta nhấn mạnh việc che dấu thông tin bằng cách phân biệt giữa việc sử dụng ngăn xếp và việc lập trình cho các tác vụ trên ngăn xếp. Đối với hàng, chúng ta tiếp tục ý tưởng này và đã nhanh chóng tìm được rất nhiều cách hiện thực có thể có. Các danh sách thông dụng cho phép truy xuất và thay đổi bất kỳ phần tử nào. Do đó nguyên tắc che dấu thông tin đối

CuuDuongThanCong.com https://fb.com/tailieudientucntt

với danh sách càng quan trọng hơn nhiều so với ngăn xếp và hàng. Chúng ta hãy đặc tả cho các tác vụ trên danh sách:

Constructor cần có trước khi danh sách được sử dụng:

```
template <class Entry>
List<Entry>::List();
```
post: đối tượng danh sách rỗng đã được tạo.

Tác vụ thực hiện trên một danh sách đã có và làm rỗng danh sách:

```
template <class Entry>
void List<Entry>::clear();
```
post: Mọi phần tử của danh sách đã được giải phóng, danh sách trở nên rỗng.

Các tác vụ xác định trạng thái của danh sách:

```
template <class Entry>
bool List<Entry>::empty() const;
```
post: trả về *true* nếu danh sách rỗng, ngược lại trả về *false*. Danh sách không đổi.

```
template <class Entry>
bool List<Entry>::full() const;
```
post: trả về *true* nếu danh sách đầy, ngược lại trả về *false*. Danh sách không đổi.

```
template <class Entry>
int List<Entry>::size() const;
```
post: trả về số phần tử của danh sách. Danh sách không đổi.

Chúng ta xem xét tiếp các tác vụ truy xuất các phần tử của danh sách. Tương tự như đối với ngăn xếp và hàng, các tác vụ này sẽ trả về `ErrorCode` khi cần thiết.

Chúng ta dùng một số nguyên để chỉ vị trí (`position`) của phần tử trong danh sách. Vị trí ở đây được hiểu là thứ tự của phần tử trong danh sách. Các vị trí trong danh sách được đánh số 0, 1, 2, ...Việc xác định một phần tử trong danh sách thông qua vị trí rất giống với sự sử dụng chỉ số trong dãy, tuy nhiên vẫn có một số điểm khác nhau quan trọng. Nếu chúng ta thêm một phần tử vào một vị trí nào đó trong danh sách thì vị trí của tất cả các phần tử phía sau sẽ tăng lên 1. Nếu loại một phần tử thì vị trí các phần tử phía sau giảm 1. Vị trí của các phần tử trong danh sách được xác định không xét đến cách hiện thực. Đối với danh sách liên tục, hiện thực bằng dãy, vị trí phần tử rõ ràng là chỉ số của phần tử trong dãy. Nhưng chúng ta cũng vẫn thông qua vị trí để tìm các phần tử trong danh sách liên kết dù rằng danh sách liên kết không có chỉ số.

CuuDuongThanCong.com https://fb.com/tailieudientucntt

Chúng ta sẽ đặc tả chính xác các phương thức liên quan đến chỉ một phần tử của danh sách dưới đây.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x);
```
post: Nếu danh sách chưa đầy và `0 ≤ position ≤ n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: mọi phần tử từ `position` đến cuối danh sách sẽ có vị trí tăng lên 1, x được thêm vào tại `position`; ngược lại, danh sách không đổi, `ErrorCode` sẽ cho biết lỗi cụ thể.

Phương thức `insert` chấp nhận `position` bằng `n` vì nó chấp nhận thêm phần tử mới ngay sau phần tử cuối. Tuy nhiên, các phương thức sau chỉ chấp nhận `position<n`, vì chúng chỉ thực hiện trên những phần tử đã có sẵn.

```
template <class Entry>
ErrorCode List<Entry>::remove(int position, Entry &x);
```
post: Nếu `0 ≤ position < n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: phần tử tại `position` được loại khỏi danh sách, trị của nó được chép vào x, các phần tử phía sau giảm vị trí bớt 1; ngược lại, danh sách không đổi, `ErrorCode` sẽ cho biết lỗi cụ thể.

```
template <class Entry>
ErrorCode List<Entry>::retrieve(int position, Entry &x) const;
```
post: Nếu 0 ≤ `position < n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: phần tử tại `position` được chép vào x, danh sách không đổi; ngược lại, `ErrorCode` sẽ cho biết lỗi cụ thể. Cả hai trường hợp danh sách đều không đổi.

```
template <class Entry>
ErrorCode List<Entry>::replace(int position, const Entry &x);
```
post: Nếu 0 ≤ `position < n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: phần tử tại `position` được thay thế bởi x; ngược lại, danh sách không đổi, `ErrorCode` sẽ cho biết lỗi cụ thể.

Phương thức duyệt danh sách để thực hiện một nhiệm vụ nào đó cho từng phần tử của danh sách thường tỏ ra có lợi, đặc biệt cho mục đích kiểm tra. Người sử dụng gọi phương thức này khi muốn thực hiện một công việc gì đó trên từng phần tử của danh sách. Chẳng hạn, người sử dụng có hai hàm

`void` **`update`**`(List_Entry &x)` và `void` **`modify`**`(List_Entry &x),`

và một đối tượng `the_list` của lớp `List`, có thể sử dụng lệnh

`the_list.`**`traverse`**`(update)` hoặc `the_list.`**`traverse`**`(modify)`

CuuDuongThanCong.com https://fb.com/tailieudientucntt

để thực hiện một trong hai hàm trên lên mỗi phần tử của danh sách. Nếu người sử dụng muốn in mọi phần tử của danh sách thì gọi như sau:

```
the_list.traverse(print)
```

với `void print(Entry &x)` là một hàm dùng để in một phần tử của danh sách.

Khi gọi phương thức `traverse`, người sử dụng gởi tên của hàm làm thông số. Trong C++, tên của hàm mà không có cặp dấu ngoặc chính là con trỏ chỉ đến hàm. Thông số hình thức **`visit`** dưới đây của phương thức **`traverse`** cần được khai báo như một con trỏ chỉ đến hàm. Ngoài ra, khai báo con trỏ hàm làm thông số phải có kiểu trả về là `void` và có thông số tham chiếu đến `Entry`.

```
template <class Entry>
void List<Entry>::traverse(void(*visit)(Entry &x));
```
post: Công việc đặc tả bởi hàm `*visit` được thực hiện lần lượt trên từng phần tử của danh sách, bắt đầu từ phần tử thứ 0.

Cũng giống như mọi thông số khác, `visit` chỉ là tên hình thức và chỉ được gán bởi một con trỏ thực sự khi `traverse` bắt đầu thực thi. Biểu diễn **`*visit`** thay mặt cho hàm sẽ được sử dụng để xử lý cho từng phần tử của danh sách khi `traverse` thực thi.

Trong phần kế tiếp chúng ta sẽ hiện thực các phương thức này.

4.3. Hiện thực danh sách

Chúng ta đã đặc tả đầy đủ các tác vụ mong muốn đối với danh sách. Phần này sẽ hiện thực chi tiết chúng trong C++. Ngăn xếp và hàng đã được hiện thực cả hai dạng liên tục và liên kết. Chúng ta cũng sẽ làm tương tự cho danh sách.

4.3.1. Hiện thực danh sách liên tục

Trong hiện thực danh sách liên tục (*contiguous list*), các phần tử của danh sách có kiểu là `Entry` được chứa trong dãy kích thước là `max_list`. Cũng giống như hiện thực ngăn xếp liên tục, ở đây chúng ta cần một biến `count` đếm số phần tử hiện có trong danh sách. Sau đây là định nghĩa lớp `List` có hai thuộc tính thành phần và tất cả các phương thức mà chúng ta đã đặc tả.

```
template <class Entry>
class List {
public:
//  Các phương thức của kiểu dữ liệu trừu tượng danh sách
   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
   void traverse(void (*visit)(Entry &));
   ErrorCode retrieve(int position, Entry &x) const;
   ErrorCode replace(int position, const Entry &x);
   ErrorCode remove(int position, Entry &x);
   ErrorCode insert(int position, const Entry &x);

protected:
// Các thuộc tính cho hiện thực danh sách liên tục
   int count;
   Entry entry[max_list];
};
```

Hầu hết các phương thức (`List, clear, empty, full, size, retrieve`) rất dễ hiện thực.

```
template <class Entry>
int List<Entry>::size() const
/*
post: trả về số phần tử của danh sách. Danh sách không đổi.
*/
{
   return count;
}
```

Chúng ta dành các phương thức đơn giản khác lại cho phần bài tập. Ở đây chúng ta sẽ tập trung vào các phương thức truy xuất dữ liệu. Khi thêm một phần tử mới, các phần tử trong dãy phải được di chuyển để nhường chỗ.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 ≤ position ≤ n, n là số phần tử hiện có của danh sách,
      phương thức trả về success: mọi phần tử từ position đến cuối danh sách sẽ có vị trí
      tăng lên 1, x được thêm vào tại position; ngược lại, danh sách không đổi, ErrorCode sẽ
      cho biết lỗi cụ thể.
*/
{
   if (full())
      return overflow;
   if (position < 0 || position > count)
      return range_error;
   for (int i = count - 1; i >= position; i--)
      entry[i + 1] = entry[i];
   entry[position] = x;
   count++;
   return success;
}
```

Có bao nhiêu công việc mà hàm trên cần phải làm? Nếu phần tử mới được thêm vào cuối danh sách thì hàm chỉ phải thực hiện một số không đổi các lệnh. Trong trường hợp ngược lại, nếu phần tử được thêm vào đầu danh sách, hàm sẽ phải dịch chuyển một số phần tử lớn nhất để tạo chỗ trống, nếu danh sách đã

CuuDuongThanCong.com https://fb.com/tailieudientucntt

khá dài thì công việc cần làm rất nhiều. Xét bình quân, nếu chúng ta giả sử mọi vị trí trong danh sách đều có khả năng thêm phần tử mới như nhau, hàm trên sẽ phải dịch chuyển một nửa số phần tử trong danh sách. Chúng ta nói rằng số việc cần làm trong hàm tỉ lệ với chiều dài n của danh sách.

Tương tự, việc loại phần tử trong danh sách cũng cần phải dịch chuyển các phần tử để lấp chỗ trống và việc loại này cũng tốn thời gian tỉ lệ với chiều dài n của danh sách.

Khác với hai trường hợp trên, hầu hết các phương thức còn lại không cần thực hiện vòng lặp nào và thời gian thực hiện là hằng số. Tóm lại,

Trong xử lý danh sách liên tục có n phần tử:

- **`insert`** và **`remove`** cần thời gian tỉ lệ với n.
- **`List`**, **`clear`**, **`empty`**, **`full`**, **`size`**, **`replace`** và **`retrieve`** thực hiện trong thời gian không đổi.

Chúng ta chưa kể ra đây phương thức **`traverse`** vì thời gian thực hiện còn phụ thuộc vào thông số hàm visit. Riêng `traverse` thì ít nhất cũng cần thời gian tỉ lệ với n do phải có vòng lặp để duyệt qua hết các phần tử của danh sách. Tuy nhiên, với cùng một hàm `visit` thì `traverse` cần thời gian tỉ lệ với n.

```
template <class Entry>
void List<Entry>::traverse(void (*visit)(Entry &))
/*
post: Công việc đặc tả bởi hàm *visit được thực hiện lần lượt trên từng phần tử của danh sách,
      bắt đầu từ phần tử thứ 0.
*/
{
   for (int i = 0; i < count; i++)
      (*visit)(entry[i]);
}
```

4.3.2. Hiện thực danh sách liên kết đơn giản

4.3.2.1. Các khai báo

Để hiện thực danh sách liên kết (*linked list*), chúng ta bắt đầu với khai báo `Node`. `Node` dưới đây cũng tương tự như trong ngăn xếp liên kết và hàng liên kết.

```
template <class Entry>
struct Node {
//  Các thuộc tính
   Entry entry;
   Node<Entry> *next;
//  constructors
   Node();
   Node(Entry item, Node<Entry> *link = NULL);
};
template <class Entry>
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
class List {
public:
//  Các phương thức của danh sách liên kết (cũng giống như của danh sách liên tục)
//  Các phương thức bảo đảm tính an toàn cho CTDL có chứa thuộc tính con trỏ.
   ~List();
   List(const List<Entry> &copy);
   void operator =(const List<Entry> &copy);
protected:
//  Các thuộc tính cho hiện thực liên kết của danh sách
   int count;
   Node<Entry> *head; // Con trỏ chỉ phần tử đầu của danh sách.

//  The following auxiliary function is used to locate list positions
   Node<Entry> *set_position(int position) const;
};
```

Trong định nghĩa trên chúng ta không liệt kê lại các phương thức của danh sách liên kết vì chúng cũng tương tự như đối với danh sách liên tục. Trong phần `protected` chúng ta có bổ sung phương thức `set_position` mà chúng ta sẽ thấy ích lợi của nó trong khi hiện thực các phương thức `public` khác.

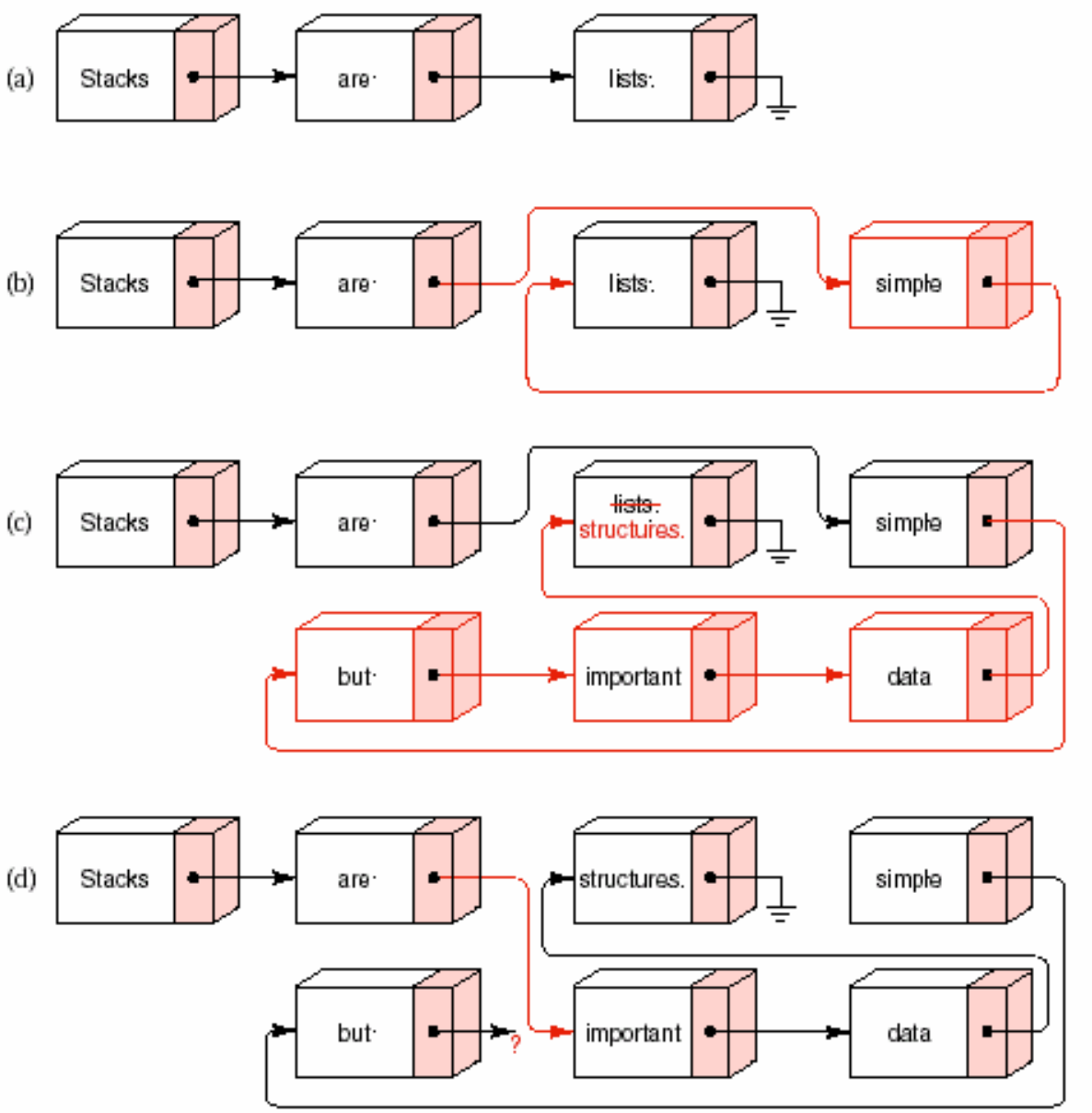
4.3.2.2. Ví dụ

Hình 4.1 minh họa việc thêm bớt dữ liệu trong danh sách qua một ví dụ sửa đổi văn bản. Mỗi phần tử trong danh sách chứa một từ và một tham chiếu đến phần tử kế. Hình a là danh sách chứa câu ban đầu là "*Stacks are lists*" . Nếu chúng ta thêm từ "*simple*" trước từ "*lists*" chúng ta có danh sách như hình b. Tiếp theo chúng ta quyết định thay thế từ "*lists*" bởi từ "*structures*" và thêm ba từ "*but important data*" thì có hình c. Cuối cùng chúng ta lại quyết định bỏ đi các từ "*simple but*" để có được câu cuối cùng "*Stacks are important data structures*".

4.3.2.3. Tìm đến một vị trí trong danh sách

Chúng ta thiết kế một hàm **`set_position`** để được gọi trong một vài phương thức. Hàm này nhận thông số là **`position`** (một số nguyên chỉ vị trí phần tử trong danh sách) và trả về con trỏ tham chiếu đến phần tử tương ứng trong danh sách.

CuuDuongThanCong.com https://fb.com/tailieudientucntt



Hình 4.1- Các thao tác trên danh sách liên kết.

Nếu người sử dụng nhìn thấy được `set_position` thì họ sẽ có thể truy xuất đến mọi phần tử trong danh sách. Vì vậy, để duy trì tính đóng kín của dữ liệu, chúng ta sẽ không cho phép người sử dụng nhìn thấy hàm `set_position`. Bằng cách khai báo `protected` chúng ta bảo đảm rằng hàm này chỉ được gọi trong các phương thức khác của danh sách.

Cách dễ nhất để xây dựng hàm `set_position` là bắt đầu duyệt từ đầu của danh sách cho đến phần tử mà chúng ta muốn tìm.

```
template <class Entry>
Node<Entry> *List<Entry>::set_position(int position) const
/*
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
Pre:  position phải hợp lệ; 0 <= position < count.
Post: trả về địa chỉ của phần tử tại position.
*/
{
   Node<Entry> *q = head;
   for (int i = 0; i < position; i++) q = q->next;
   return q;
}
```

Do chúng ta nắm được chính xác các phương thức nào cần gọi đến `set_position`, trong hàm này chúng ta không cần kiểm tra lỗi. Thay vào đó chúng ta bảo đảm bằng `precondition` cho nó. Có nghĩa là các phương thức trước khi gọi `set_position` sẽ kiểm tra trước và chỉ gọi khi điều kiện hợp lệ. Việc kiểm tra sẽ không phải lặp lại trong hàm này, chương trình sẽ hiệu quả hơn.

Nếu mọi phần tử được truy xuất với xác suất ngang nhau thì trung bình hàm `set_position` sẽ phải duyệt qua một nửa số phần tử trong danh sách để đến được vị trí cần thiết. Thời gian này tỉ lệ với chiều dài n của danh sách.

4.3.2.4. Thêm phần tử vào danh sách

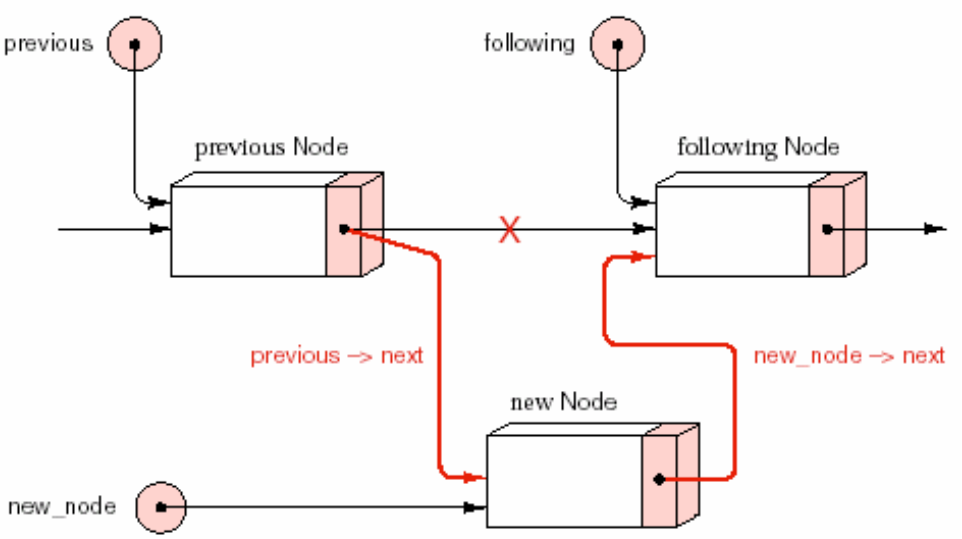
Tiếp theo chúng ta sẽ xem xét vấn đề thêm một phần tử mới vào danh sách. Nếu chúng ta có một phần tử mới và chúng ta muốn chèn phần tử này vào một vị trí nào đó trong danh sách, ngoại trừ vị trí đầu danh sách, như hình 4.2, chúng ta cần có hai con trỏ **`previous`** và **`following`** chỉ đến hai phần tử trước và sau vị trí cần chèn. Nếu con trỏ `new_node` đang chỉ phần tử mới cần chèn thì các lệnh gán sau sẽ chèn được phần tử mới vào danh sách:

```
new_node->next = following;
previous->next = new_node;
```

Trong phương thức `insert` dưới đây phép gán `new_node->next= following` được thực hiện thông qua *constructor* có nhận thông số thứ hai là `following`.

Việc thêm phần tử vào đầu danh sách cần được xử lý riêng, do trường hợp này không có phần tử nào nằm trước phần tử mới nên chúng ta không sử dụng con trỏ `previous`, thay vào đó thuộc tính `head` chỉ đến phần tử đầu của danh sách phải được gán lại.

CuuDuongThanCong.com https://fb.com/tailieudientucntt



Hình 4.2- Thêm phần tử vào danh sách liên kết.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 ≤ position ≤ n, n là số phần tử hiện có của danh sách,
      phương thức trả về success: mọi phần tử từ position đến cuối danh sách sẽ có vị trí tăng
      lên 1, x được thêm vào tại position; ngược lại, danh sách không đổi, ErrorCode sẽ cho
      biết lỗi cụ thể.
*/
{
   if (position < 0 || position > count)
      return range_error;
   Node<Entry> *new_node, *previous, *following;
   if (position == 0) // Trường hợp đặc biệt: phần tử mới thêm vào đầu danh sách.
      following = head;
   else {             // Trường hợp tổng quát.
      previous = set_position(position - 1); // Tìm phần tử phía trước vị trí cần thêm
                                             phần tử mới.
      following = previous->next;
   }

   new_node = new Node<Entry>(x, following);
   if (new_node == NULL)
      return overflow;
   if (position == 0) // Trường hợp đặc biệt: phần tử mới thêm vào đầu danh sách.
      head = new_node;
   else               // Trường hợp tổng quát.
      previous->next = new_node;
   count++;
   return success;
}
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

Ngoài lệnh gọi hàm `set_position`, các lệnh còn lại trong `insert` không phụ thuộc vào chiều dài n của danh sách. Do đó `insert`, cũng giống như `set_position`, sẽ có thời gian thực hiện tỉ lệ với chiều dài n của danh sách.

4.3.2.5. Các tác vụ khác

Các phương thức còn lại của danh sách liên kết xem như bài tập. Việc tìm kiếm một phần tử nào đó trong các phương thức luôn phải gọi hàm `set_position`. Hầu hết các phương thức này cũng giống như `insert`, sử dụng các lệnh chiếm thời gian không đổi, ngoại trừ lúc gọi hàm `set_position`. Chỉ có phương thức `clear` và `traverse` là phải duyệt qua các phần tử của danh sách. Chúng ta có kết luận như sau:

Trong việc xử lý một danh sách liên kết có n phần tử:

- **`insert`**, **`remove`**, **`retrieve`** và **`replace`** cần thời gian tỉ lệ với n.
- **`List`**, **`empty`**, **`full`** và **`size`** thực hiện với thời gian không đổi.

Một lần nữa, chúng ta chưa kể đến phương thức **`traverse`** ở đây, vì thời gian nó cần còn phụ thuộc vào thông số `visit`. Tuy nhiên, cũng như phần trước, với cùng một hàm visit thì `traverse` cần thời gian tỉ lệ với n.

4.3.3. Lưu lại vị trí hiện tại

Đa số các ứng dụng truy xuất các phần tử của danh sách theo thứ tự các phần tử. Nhiều ứng dụng khác truy xuất cùng một phần tử nhiều lần, thực hiện các tác vụ truy xuất hoặc thay thế trước khi chuyển qua phần tử khác. Đối với tất cả các ứng dụng này, cách hiện thực danh sách hiện tại của chúng ta tỏ ra không hiệu quả, do mỗi lần truy xuất một phần tử, hàm set_position đều phải tìm từ đầu danh sách đến phần tử mong muốn. Nếu chúng ta có thể nhớ lại phần tử vừa được truy xuất trong danh sách, và tác vụ mà ứng dụng yêu cầu tiếp theo cũng xem xét phần tử này hoặc phần tử kế thì việc tìm kiếm bắt đầu từ vị trí được nhớ này nhanh hơn rất nhiều.

Tuy nhiên, không phải việc nhớ lại vị trí vừa được truy xuất này luôn có hiệu lực đối với mọi ứng dụng. Chẳng hạn với ứng dụng truy xuất các phần tử trong danh sách theo thứ tự ngược, mọi truy xuất đều phải bắt đầu từ đầu danh sách do các tham chiếu trong các phần tử chỉ có một chiều.

Chúng ta dùng thuộc tính **`current_position`** để lưu vị trí vừa nói trên. Thuộc tính này sẽ được `set_position` sử dụng cũng như sẽ cập nhật lại mỗi khi hàm này được gọi. Điều cần lưu ý là `set_position` được gọi trong các phương thức khác của danh sách, trong đó có một số phương thức được đặc tả là `const` có nghĩa là không được làm thay đổi danh sách, trong khi đó `current_position` phải được thay đổi. Như vậy, chúng ta sẽ dùng từ

CuuDuongThanCong.com https://fb.com/tailieudientucntt

khóa **mutable** của C++ nhưng lưu ý rằng không phải từ khóa này luôn được cung cấp bởi mọi trình biên dịch C++. Khi một thuộc tính của một lớp được khai báo là `mutable` thì nó có thể được thay đổi ngay cả trong các hàm được khai báo là `const`.

Định nghĩa danh sách mới như sau:

```
template <class Entry>
class List {
public:
//   Các phương thức của danh sách liên kết (cũng giống như của danh sách liên tục)
//   Các phương thức bảo đảm tính an toàn cho CTDL có chứa thuộc tính con trỏ.

protected:
//   Các thuộc tính cho hiện thực liên kết của danh sách có lưu vị trí hiện tại.
   int count;
   mutable int current_position;
   Node<Entry> *head;
   mutable Node<Entry> *current;

//   Hàm phụ trợ để tìm một phần tử.
   void set_position(int position) const;
};
```

Hai thuộc tính được thêm vào **current_position** và **current** đều được khai báo `protected`, do đó đối với người sử dụng lớp `List` vẫn không có gì thay đổi so với định nghĩa cũ.

Hàm `set_position` được viết lại như sau:

```
template <class Entry>
void List<Entry>::set_position(int position) const
/*
pre:  position hợp lệ: 0 <= position < count.
post: Thuộc tính current  chứa địa chỉ phần tử được tìm thấy tại position,
      current_position được cập nhật tương ứng.
*/
{
   if (position < current_position) { // Trường hợp phải tìm từ đầu danh sách
      current_position = 0;
      current = head;
   }
   for ( ; current_position != position; current_position++)
      current = current->next;
}
```

Nếu một phần tử trong danh sách được truy xuất lập lại nhiều lần thì các lệnh trong `if` cũng như trong vòng `for` của hàm trên đều không phải thực hiện, hàm sẽ không hề chiếm thời gian chạy. Nếu phần tử kế được truy xuất, các lệnh trong vòng `for` chỉ chạy một lần, hàm vẫn thực hiện rất nhanh. Trong trường hợp xấu

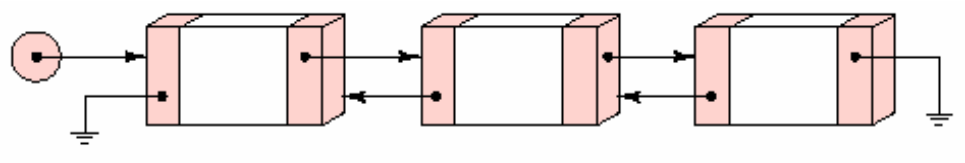
CuuDuongThanCong.com https://fb.com/tailieudientucntt

nhất, nếu cần phải bắt đầu từ đầu danh sách, hàm cũng sẽ làm việc giống như cách chúng ta đã hiện thực trước đây.

4.3.4. Danh sách liên kết kép

Một vài ứng dụng thường xuyên yêu cầu dịch chuyển tới và lui trên danh sách. Trong phần trước chúng ta đã giải quyết việc dịch chuyển theo một chiều trong quá trình duyệt danh sách. Nhưng việc lập trình hơi khó khăn và thời gian chạy chương trình phụ thuộc vào danh sách, nhất là khi danh sách có nhiều phần tử.

Để khắc phục vấn đề này, có nhiều chiến lược khác nhằm giải quyết việc tìm phần tử nằm trước phần tử hiện tại trong danh sách. Trong phần này chúng ta tìm hiểu một chiến lược đơn giản nhất nhưng cũng linh động và phù hợp trong rất nhiều trường hợp.



Hình 4.3- Danh sách liên kết kép.

4.3.4.1. Các khai báo cho danh sách liên kết kép

Như hình 4.3 minh họa, ý tưởng ở đây là việc lưu cả hai con trỏ chỉ hai hướng ngược nhau trong cùng một node của danh sách. Bằng cách dịch chuyển theo tham chiếu thích hợp chúng ta có thể duyệt danh sách theo hướng mong muốn. CTDL này được gọi là danh sách liên kết kép (*doubly linked list*).

```
template <class Node_entry>
struct Node {
//  Các thuộc tính
   Node_entry entry;
   Node<Node_entry> *next;
   Node<Node_entry> *back;
//  constructors
   Node();
   Node(Node_entry, Node<Node_entry> *link_back = NULL,
                    Node<Node_entry> *link_next = NULL);
};
```

Constructor cho Node của danh sách liên kết kép gần giống *constructor* cho Node của danh sách liên kết đơn. Dưới đây là đặc tả cho lớp danh sách liên kết kép:

CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
template <class Entry>
class List {
public:
// Các phương thức thông thường của danh sách.
// Các phương thức bảo đảm tính an toàn cho CTDL có thuộc tính con trỏ.
protected:
// Các thuộc tính
   int count;
   mutable int current_position;
   mutable Node<Entry> *current;

// Hàm phụ trợ để tìm đến một phần tử trong danh sách
   void set_position(int position) const;
};
```

Trong cách hiện thực này chúng ta chỉ cần giữ một con trỏ tham chiếu đến một phần tử nào đó trong danh sách là chúng ta có thể duyệt danh sách theo hướng này hoặc hướng kia. Như vậy, chúng ta dùng luôn con trỏ **`current`** chỉ đến phần tử hiện tại để làm nhiệm vụ này, và chúng ta không cần giữ con trỏ chỉ đến đầu hoặc cuối danh sách.

4.3.4.2. Các tác vụ trên danh sách liên kết kép

Trong danh sách liên kết kép, việc duyệt danh sách theo cả hai hướng để tìm một phần tử, việc thêm hoặc loại phần tử tại vị trí nào đó có thể được thực hiện dễ dàng. Một vài tác vụ có thay đổi chút ít so với danh sách liên kết đơn, như `insert` và `delete`, do phải cập nhật đầy đủ cả hai con trỏ theo hai hướng của danh sách .

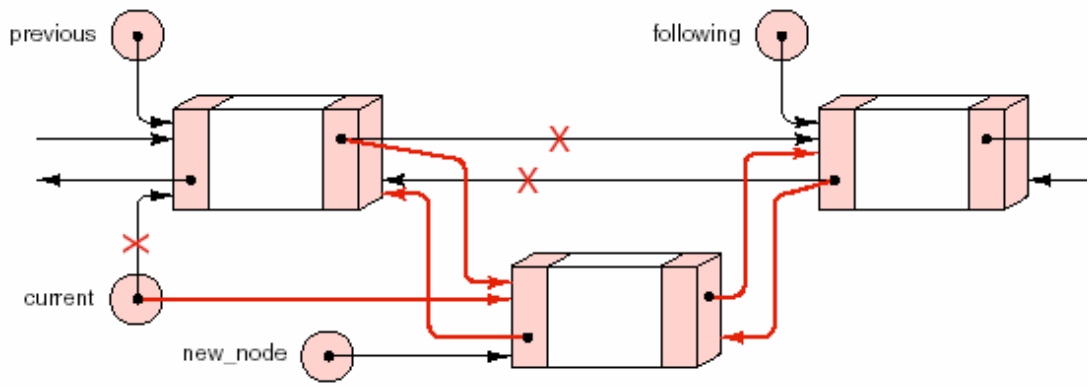
Trước hết, để tìm một vị trí nào đó, chúng ta chỉ cần quyết định nên duyệt theo hướng nào trong danh sách bắt đầu từ con trỏ `current`.

```
template <class Entry>
void List<Entry>::set_position(int position) const
/*
pre:  position hợp lệ: 0 <= position < count.
post: thuộc tính current chứa địa chỉ của phần tử tại position.
*/
{
   if (current_position <= position)
      for ( ; current_position != position; current_position++)
         current = current->next;
   else
      for ( ; current_position != position; current_position--)
         current = current->back;
}
```

Với hàm này chúng ta viết tác vụ `insert` sau đây. Hình 4.4 minh họa các con trỏ cần phải cập nhật. Chúng ta cũng đặc biệt chú ý hai trường hợp hơi đặc biệt,

CuuDuongThanCong.com https://fb.com/tailieudientucntt

đó là khi thêm phần tử vào một trong hai đầu của danh sách hoặc thêm vào một danh sách rỗng.



Hình 4.4- Thêm phần tử và danh sách liên kết kép.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 ≤ position ≤ n, n là số phần tử hiện có của danh sách,
      phương thức trả về success: mọi phần tử từ position đến cuối danh sách sẽ có vị trí
      tăng lên 1, x được thêm vào tại position; ngược lại, danh sách không đổi, ErrorCode sẽ
      cho biết lỗi cụ thể.
*/
{
   Node<Entry> *new_node, *following, *preceding;
   if (position < 0 || position > count) return range_error;
   if (position == 0) {
      if (count == 0) following = NULL;    // Trường hợp đặc biệt: danh sách đang
                                              rỗng.
      else {
         set_position(0);
         following = current;
      }
      preceding = NULL;                    // Trường hợp đặc biệt: thêm phần tử mới
                                              vào đầu danh sách, không có phần tử
                                              đứng trước.
   }
   else {
      set_position(position - 1);
      preceding = current;
      following = preceding->next;         // Trường hợp tổng quát: thêm phần tử
                                              vào giữa, nhưng gộp cả trường hợp
                                              thêm vào cuối (position = n)
                                              following sẽ nhận trị NULL.
   }
   new_node = new Node<Entry>(x, preceding, following);

   if (new_node == NULL) return overflow;
   if (preceding != NULL) preceding->next = new_node;
   if (following != NULL) following->back = new_node;
   current = new_node;
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
   current_position = position;
   count++;
   return success;
}
```

Giá phải trả đối với danh sách liên kết kép là vùng nhớ cho tham chiếu thứ hai trong mỗi `Node`. Với phần lớn ứng dụng, do vùng **`entry`** cần chứa thông tin trong `Node` lớn hơn nhiều vùng nhớ dành cho các con trỏ, nên tổng dung lượng bộ nhớ tăng không đáng kể.

4.4. So sánh các cách hiện thực của danh sách

Chúng ta đã xem xét một vài giải thuật xử lý cho danh sách liên kết và một vài biến thể về cấu trúc và cách hiện thực. Trong phần này chúng ta sẽ phân tích các ưu nhược điểm của danh sách liên kết và danh sách liên tục.

Ưu điểm lớn nhất của danh sách liên kết trong bộ nhớ động là tính mềm dẻo. Không có vấn đề tràn bộ nhớ trừ khi bộ nhớ máy tính thực sự đã được sử dụng hết. Đặc biệt khi một `entry` chứa thông tin quá lớn, chúng ta khó có thể xác định tổng dung lượng vùng nhớ như thế nào cho vừa đủ để khai báo một dãy, trong khi chúng ta cũng cần phải xét đến phần bộ nhớ còn lại sao cho đủ để dành cho các biến khác. Trong bộ nhớ động, chúng ta không cần phải quyết định điều này trước khi chương trình chạy.

Trong danh sách liên kết, việc thêm hay loại phần tử có thể thực hiện nhanh hơn so với trong danh sách liên tục. Đối với các CTDL lớn, việc thay đổi một vài con trỏ nhanh hơn rất nhiều so với việc chép dữ liệu sang chỗ khác.

Nhược điểm đầu tiên của danh sách liên kết là tốn vùng nhớ cho các con trỏ. Trong phần lớn hệ thống, một con trỏ chiếm vùng nhớ bằng vùng nhớ của một số nguyên. Như vậy một danh sách liên kết các số nguyên sẽ đòi hỏi vùng nhớ gấp đôi một danh sách liên tục các số nguyên.

Trong nhiều ứng dụng thực tiễn, một `node` trong danh sách thường lớn, dữ liệu thường chứa hàng trăm *bytes*, việc sử dụng danh sách liên kết chỉ tốn thêm khoản một phần trăm vùng nhớ. Thực ra, danh sách liên kết tiết kiệm vùng nhớ hơn nhiều, nếu xét đến những vùng nhớ được khai báo dự trữ sẵn cho việc thêm phần tử trong danh sách liên tục mà chưa được dùng đến. Nếu mỗi `entry` chiếm 100 *bytes* thì vùng nhớ liên tục chỉ tiết kiệm khi số phần tử sử dụng thực sự trong dãy lên đến 99 *bytes*.

Nhược điểm chính của danh sách liên kết là nó không thích hợp với việc truy xuất ngẫu nhiên. Trong vùng nhớ liên tục, việc truy xuất đến bất kỳ vị trí nào cũng rất nhanh và không khác gì so với những vị trí khác. Trong danh sách liên kết, có thể phải duyệt cả chặng đường dài mới đến được phần tử mong muốn. Việc

CuuDuongThanCong.com https://fb.com/tailieudientucntt

truy xuất một node trong vùng nhớ liên kết cũng chiếm hơn một chút thời gian vì trước hết phải có được con trỏ và sau đó mới đến được địa chỉ cần tìm, tuy nhiên điều này thường không quan trọng. Ngoài ra, các tác vụ xử lý trong danh sách liên kết thường phải lập trình công phu hơn.

Danh sách liên tục, nói chung, thường được chọn khi:

- Mỗi entry rất nhỏ.
- Kích thước của danh sách được biết trước khi lập trình.
- Ít có nhu cầu thêm hoặc loại phần tử trừ trường hợp phần tử cuối danh sách.
- Việc truy xuất ngẫu nhiên thường xảy ra.

Danh sách liên kết tỏ ra ưu thế khi:

- Mỗi entry lớn.
- Kích thước của danh sách không được biết trước khi ứng dụng chạy.
- Có yêu cầu về tính linh hoạt: thêm, loại phần tử hoặc tổ chức lại các phần tử.

Để chọn lựa CTDL với cách hiện thực thích hợp, người lập trình cần xem xét các tác vụ nào sẽ được thực hiện trên cấu trúc đó, tác vụ nào trong số đó là quan trọng nhất. Việc truy xuất là cục bộ nếu một phần tử được truy xuất, nó có thể được truy xuất lần nữa. Và nếu các phần tử thường được truy xuất theo thứ tự, thì nên nhớ lại vị trí phần tử vừa được truy xuất như là một thuộc tính của danh sách. Còn nếu việc truy xuất theo hai hướng của danh sách là cần thiết thì nên chọn cách hiện thực danh sách liên kết kép.

4.5. Danh sách liên kết trong mảng liên tục

Một vài ngôn ngữ tuy xưa nhưng rất phổ biến như Fortran, Cobol và Basic không cung cấp khả năng sử dụng bộ nhớ động hoặc con trỏ. Nếu cần sử dụng các ngôn ngữ này để giải quyết các bài toán mà trong đó các tác vụ trên danh sách liên kết (DSLK) tỏ ra có ưu thế hơn hẳn trên danh sách liên tục (việc thay đổi một vài con trỏ dễ dàng và nhanh chóng hơn nhiều việc phải chép lại một số lượng lớn dữ liệu), chúng ta vẫn có thể sử dụng mảng liên tục để mô phỏng DSLK. Trong phần này chúng ta sẽ tìm hiểu một hiện thực của DSLK mà không cần con trỏ. Hay nói cách khác, chúng ta không dùng con trỏ chứa địa chỉ, mà sẽ dùng con trỏ là một số nguyên, và DSLK sẽ được hiện thực trong một mảng liên tục.

4.5.1. Phương pháp

Ý tưởng chính ở đây bắt đầu từ một mảng liên tục dùng để chứa các phần tử của một DSLK. Chúng ta xem mảng này như một vùng nhớ chưa sử dụng và chúng ta sẽ tự phân phối lấy. Chúng ta sẽ xây dựng một số hàm để quản lý mảng

CuuDuongThanCong.com https://fb.com/tailieudientucntt

này: nhận biết vùng nào trong mảng chưa được sử dụng, nối kết các phần tử trong mảng theo một thứ tự mong muốn.

Một đặc điểm của DSLK mà chúng ta phải bỏ qua trong phần này là việc định vị bộ nhớ động, ngay từ đầu chúng ta phải xác định kích thước cần thiết cho mảng. Mọi ưu điểm còn lại khác của DSLK đều được giữ nguyên, như tính mềm dẻo trong việc tổ chức lại vùng nhớ cho các phần tử có kích thước lớn, hoặc tính dễ dàng và hiệu quả trong việc thêm hay bớt bất cứ phần tử nào trong danh sách.

Hiện thực DSLK trong mảng liên tục cũng tỏ ra rất hiệu quả trong những ngôn ngữ tựa C++ có cung cấp con trỏ và cách định vị bộ nhớ động. Các ứng dụng dưới đây được xem là thích hợp khi sử dụng DSLK trong mảng liên tục :

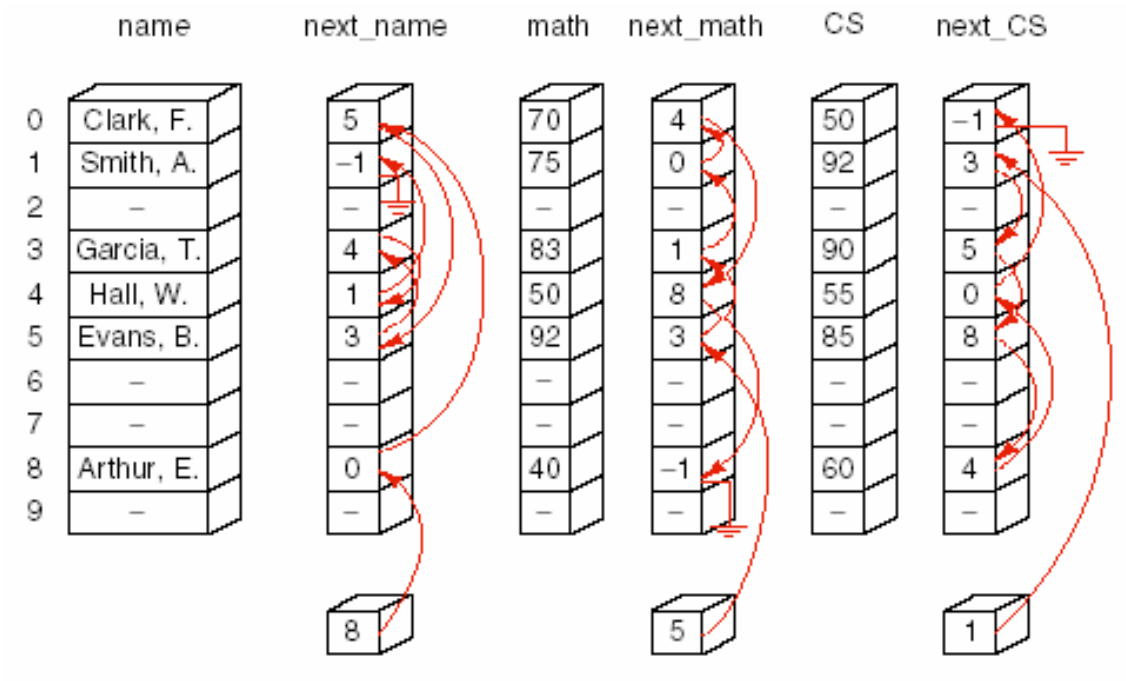
- Số phần tử tối đa trong danh sách được biết trước.
- Các tham chiếu thường xuyên được tổ chức lại, nhưng việc thêm hoặc loại phần tử tương đối ít xảy ra.
- Cùng dữ liệu nhưng có khi cần xử lý như DSLK có khi lại cần xử lý như danh sách liên tục.

Hình 4.5 là một ví dụ minh họa cho những ứng dụng như vậy. Đây là một phần dữ liệu chứa các thông tin về sinh viên. Mã sinh viên được gán cho các sinh viên theo thứ tự nhập trường, tên và điểm số không theo một thứ tự đặc biệt nào. Thông tin về sinh viên có thể được tìm thấy nhanh chóng dựa vào mã sinh viên do số này được dùng như chỉ số để tìm trong mảng liên tục. Tuy nhiên, thỉnh thoảng chúng ta cần in danh sách sinh viên có thứ tự theo tên, và điều này có thể làm được bằng cách lần theo **các tham chiếu được lưu trong mảng `next_name`**. Tương tự, các điểm số cũng có thể sắp thứ tự nhờ các tham chiếu trong các mảng tương ứng.

Để thấy được cách hiện thực này của DSLK làm việc như thế nào, chúng ta hãy duyệt DSLK `next_name` trong phần đầu của hình 4.5. Đầu vào của danh sách chứa trị là 8, có nghĩa là phần tử trong vị trí 8, Arthur, E., là phần tử đầu tiên của danh sách. Vị trí 8 của `next_name` chứa trị 0, có nghĩa là tên ở vị trí 0, Clark, F., là phần tử thứ hai. Vị trí 0 của `next_name` chứ trị 5, vậy Evans, B., là phần tử kế tiếp. Vị trí 5 chỉ đến vị trí 3 (Garcia, T.), vị trí 3 lại chỉ ví trí 4 (Hall, W.), và vị trí 4 chỉ vị trí 1 (Smith, A.). Tại vị trí 1, `next_name` chứa trị -1, có nghĩa là vị trí 1 là phần tử cuối cùng của danh sách.

Tương tự, mảng `next_math` biểu diễn một DSLK, cho phép truy xuất mảng `math` theo thứ tự giảm dần. Phần tử đầu tiên tại vị trí 5, kế đến là 3, 1, 0, 4, 8. Thứ tự các phần tử xuất hiện trong DSLK biểu diễn bởi `next_CS` là 1, 3, 5, 8, 4, 0.

CuuDuongThanCong.com https://fb.com/tailieudientucntt



Hình 4.5- DSLK trong mảng liên tục.

Như ví dụ trong hình 4.5, hiện thực của DSLK trong mảng liên tục có được tính linh hoạt của DSLK đối với những sự thay đổi. Ngoài ra nó còn có khả năng chia sẻ thông tin (chẳng hạn tên sinh viên) giữa các DSLK khác nhau. Hiện thực này cũng còn có ưu điểm của danh sách liên tục là có thể truy xuất ngẫu nhiên các phần tử nhờ cách sử dụng chỉ số truy xuất trực tiếp.

Trong hiện thực của DSLK trong mảng liên tục, các con trỏ trở thành các chỉ số tương đối so với điểm bắt đầu của danh sách. Các tham chiếu của danh sách chứa trong một mảng, mỗi phần tử của mảng chứa một số nguyên chỉ đến vị trí của phần tử kế của danh sách trong mảng chứa dữ liệu. Để phân biệt với các con trỏ (*pointer*) của DSLK trong bộ nhớ động, chúng ta sẽ dùng từ **chỉ số** (*index*) để gọi các tham chiếu này.

Chúng ta cần khai báo hai mảng liên tục cho mỗi DSLK, **`entry[ ]`** để chứa dữ liệu, và **`next_node[ ]`** để chứa chỉ số chỉ đến phần tử kế. Đối với phần lớn các ứng dụng, **`entry`** là một mảng mà mỗi phần tử là một cấu trúc, hoặc một vài mảng trong trường hợp ngôn ngữ lập trình không cung cấp kiểu cấu trúc. Cả hai mảng **`entry`** và **`next_node`** cần đánh chỉ số từ 0 đến **`max_list-1`**, **`max_list`** là một hằng số biết trước.

Do chúng ta dùng chỉ số bắt đầu từ 0, chúng ta sẽ dùng trị đặc biệt –1 để biểu diễn danh sách đã kết thúc, tương tự như trị `NULL` của con trỏ trong bộ nhớ động.

CuuDuongThanCong.com https://fb.com/tailieudientucntt

4.5.2. Các tác vụ quản lý vùng nhớ

Nhiệm vụ đầu tiên của chúng ta là nắm được các vùng nhớ còn trống để viết một số hàm tìm một vị trí mới hay trả một vị trí không sử dụng nữa về lại vùng nhớ trống. Khác với phần 4.3.2, toàn bộ vùng nhớ mà chúng ta sẽ dùng ở đây là một mảng liên tục gọi là **workspace**, các phần tử của nó tương ứng với các phần tử trong DSLK (hình 4.6). Chúng ta cũng sẽ gọi các phần tử trong workspace là node và sẽ khai báo **Node** để chứa dữ liệu. Mỗi **Node** là một cấu trúc gồm hai phần: **entry** kiểu **Entry** chứa dữ liệu, và **next** kiểu **index**. Kiểu index được hiện thực bằng số nguyên, có các trị biểu diễn vị trí các phần tử trong mảng liên tục, và như vậy nó thay thế kiểu con trỏ như trong các DSLK trước đây.

Các vị trí trống trong workspace có hai dạng:

- Các node chưa được sử dụng tới.
- Các node đã từng được sử dụng nhưng đã được giải phóng.

Chúng ta sẽ bắt đầu sử dụng từ đầu mảng liên tục và dùng chỉ số **last_used** chứa vị trí cuối vừa mới sử dụng trong mảng. Các vị trí trong mảng có chỉ số lớn hơn last_used là các vị trí chưa hề được sử dụng.

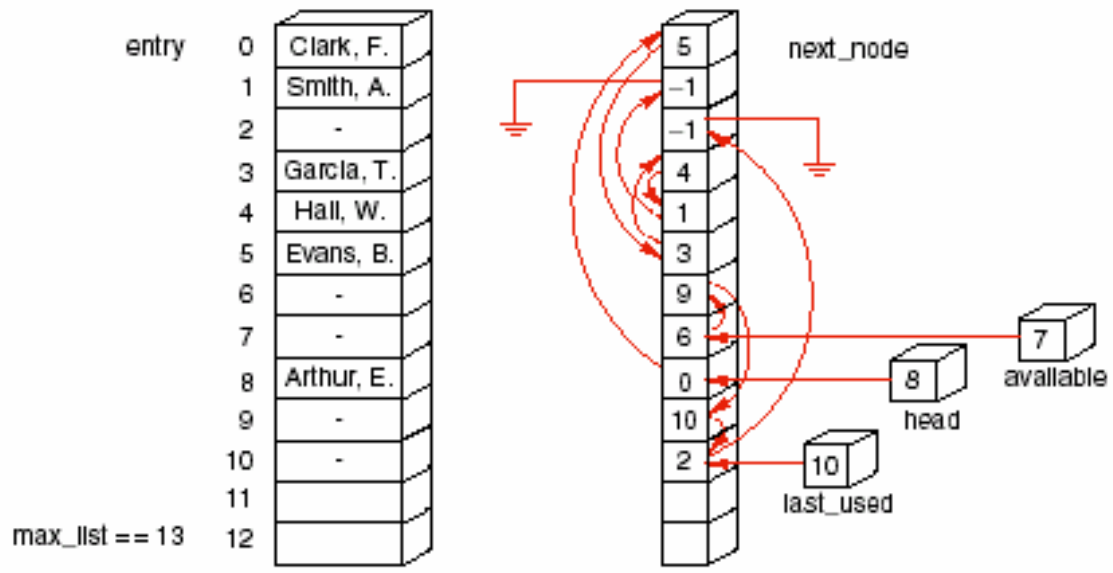
Các node đang chứa dữ liệu sẽ thuộc một DSLK có đầu vào là **head**. Head chứa vị trí của phần tử đầu của DSLK trong mảng. Các node kế tiếp trong DSLK này được truy xuất thông qua các chỉ số trong thành phần **next** của các **node**, biểu diễn bởi các mũi tên bên trái của **next_node** trong hình 4.6**.** Node cuối cùng của DSLK có chỉ số là –1. Chúng ta đọc được danh sách có các tên sắp theo thứ tự *alphabet* bắt đầu từ head = 8, Arthur, E. nằm đầu danh sách, rồi đến các tên tại các vị trí 0, 5, 3, 4, 1 của mảng; Smith, A. là tên nằm cuối danh sách.

Đối với những node đã từng sử dụng và đã được giải phóng, chúng ta sẽ dùng một dạng của cấu trúc liên kết để nối kết chúng lại với nhau và để có thể truy xuất đến, từ node này đến node kế. Do ngăn xếp liên kết là một dạng đơn giản nhất của cấu trúc liên kết, chúng ta sẽ dùng ngăn xếp liên kết cho trường hợp này. Ngăn xếp liên kết cũng được nối kết nhờ chỉ số **next** trong mỗi node, **available** là một chỉ số chứa **top** của ngăn xếp.

Các mũi tên bên phải của **next_node** trong hình 4.6, với đầu vào là **available**, chỉ các node trong một ngăn xếp bao gồm các node đã từng được sử dụng và đã được giải phóng. Chú ý rằng chỉ số trong các vùng **next** của ngăn xếp liên kết này chính xác là các số ≤ **last_used**, đó cũng là các vị trí trong mảng hiện tại không có dữ liệu. Bắt đầu từ **available** = 7, rồi đến 6, 9, 10, 2. Còn các vị trí từ **last_used**+1 trở đi là các vị trí chưa hề có dữ liệu.

CuuDuongThanCong.com https://fb.com/tailieudientucntt

Khi có một node bị loại khỏi DSLK chứa dữ liệu (chẳng hạn loại tên một sinh viên ra khỏi danh sách), vị trí của nó trong mảng được giải phóng và được push vào ngăn xếp liên kết. Ngược lại, khi cần thêm một node mới vào DSLK, chúng ta tìm vị trí trống bằng cách pop một phần tử ra khỏi ngăn xếp liên kết: trong hình 4.6 thì node mới sẽ được thêm vào vị trí 7 của mảng, còn **available** được cập nhật lại là 6. Chỉ khi cần thêm một node mới vào DSLK mà **available**=-1 (ngăn xếp liên kết rỗng), chúng ta mới dùng đến một node mới chưa hề sử dụng đến, đó là vị trí **last_used+1**, **last_used** được cập nhật lại. Khi **last_used** đạt **max_list-1**, mà **available=-1**, thì workspace đầy, danh sách của chúng ta không cho phép thêm node mới nữa.



Hình 4.6- DSLK trong mảng liên tục và ngăn xếp liên kết chứa các vùng có thể sử dụng.

Khi đối tượng **List** được khởi tạo, **available** và **last_used** phải được gán -1: **available=-1** chỉ ra rằng ngăn xếp chứa các vùng nhớ đã từng được sử dụng và đã được giải phóng là rỗng, **last_used=-1** chỉ rằng chưa có vùng nhớ nào trong mảng đã được sử dụng.

Chúng ta có khai báo DSLK trong mảng liên tục như sau:

```
typedef int index;
const int max_list = 7; //  giá trị nhỏ dành cho việc kiểm tra CTDL.
template <class Entry>
class Node {
public:
   Entry entry;
   index next;
};

template <class Entry>
class List {
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
public:
//  Methods of the list ADT
   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(Entry &));
   Error_code retrieve(int position, Entry &x) const;
   Error_code replace(int position, const Entry &x);
   Error_code remove(int position, Entry &x);
   Error_code insert(int position, const Entry &x);

protected:
//  Các thuộc tính
   Node<Entry> workspace[max_list];
   index available, last_used, head;
   int count;

//  Các hàm phụ trợ
   index new_node();
   void delete_node(index n);
   int current_position(index n) const;
   index set_position(int position) const;
};
```

Các phương thức `public` trên được đặc tả hoàn toàn giống với các hiện thực khác của `List` trước đây. Điều này có nghĩa là hiện thực mới của chúng ta có thể thay thế bất kỳ một hiện thực nào khác của CTDL trừu tượng `List` trong các ứng dụng. Một số hàm phụ trợ `protected` được bổ sung để quản lý các `node` trong `workspace`. Chúng được sử dụng để xây dựng các phương thức `public` nhưng không được nhìn thấy bởi người sử dụng. Các hàm **`new_node`** và **`delete_node`** thay thế cho các tác vụ **`new`** và **`delete`** của C++. Chẳng hạn, **`new_node`** trả về chỉ số của một vùng đang trống của `workspace` (để thêm phần tử mới cho danh sách).

```
template <class Entry>
index List<Entry>::new_node()
/*
post: Trả về chỉ số của phần tử đầu tiên có thể sử dụng trong workspace; các thuộc tính
      available, last_used, và workspace được cập nhật nếu cần thiết .
      Nếu workspace thực sự đầy, trả về -1.
*/

{
   index new_index;

   if (available != -1) {                        // ngăn xếp chưa rỗng.
      new_index = available;                     // pop từ ngăn xếp.
      available = workspace[available].next; // cập nhật cho ngăn xếp.
   } else if (last_used<max_list - 1){// ngăn xếp rỗng và workspace chưa đầy.
      new_index = ++last_used;
   } else return -1;
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
   workspace[new_index].next = -1;
   return new_index;
}
```

```
template <class Entry>
void List<Entry>::delete_node(index old_index)
/*
pre:  Danh sách có một phần tử lưu tại chỉ số old_index.
post: Vùng nhớ có chỉ số old_index được đẩy vào ngăn xếp các chỗ trống có thể sử dụng lại;
      các thuộc tính available, last_used, và workspace được cập nhật nếu cần thiết.
*/
{
   index previous;
   if (old_index == head) head = workspace[old_index].next;
   else {
      previous = set_position(current_position(old_index) - 1);
      workspace[previous].next = workspace[old_index].next;
   }
   workspace[old_index].next = available;        // đẩy vào ngăn xếp.
   available = old_index;
}
```

Cả hai hàm này thực ra là thực hiện việc push và pop ngăn xếp. Nếu muốn chúng ta có thể viết các hàm xử lý ngăn xếp riêng rồi mới viết hàm sử dụng chúng.

Các hàm phụ trợ protected khác là **set_position** và **current_position**. Cũng giống như các hiện thực trước, **set_position** nhận vị trí (thứ tự) của phần tử trong danh sách và trả về chỉ số phần tử đó trong workspace. Hàm **current_position** nhận chỉ số phần tử trong workspace và trả về vị trí (thứ tự) của phần tử trong danh sách. Hiện thực của chúng được xem như bài tập.

```
index List<Entry>::set_position(int position) const;
pre: position là vị trí hợp lý trong danh sách; 0 ≤ position < count.
post: trả về chỉ số trong workspace của node tại vị trí position trong danh sách..
```

```
int List<Entry>::current_position(index n) const;
post: trả về vị trí trong danh sách của node tại chỉ số n trong workspace, hoặc –1 nếu không có
      node này.
```

4.5.3. Các tác vụ khác

Chúng ta hiện thực các phương thức xử lý cho DSLK trong mảng liên tục bằng cách thay đổi các phương thức đã có của DSLK trong phần 4.3.2. Phần lớn việc hiện thực này được dành lại xem như bài tập, ở đây chúng ta sẽ viết hàm duyệt danh sách và thêm phần tử mới vào danh sách.

```
template <class Entry>
void List<Entry>::traverse(void (*visit)(Entry &))
```


CuuDuongThanCong.com https://fb.com/tailieudientucntt

```
/*
post: Công việc cần làm bởi hàm *visit được thực hiện trên từng phần tử của danh sách, bắt
      đầu tử phần tử thứ 0.
*/
{
   for (index n = head; n != -1; n = workspace[n].next)
      (*visit)(workspace[n].entry);
}
```

So sánh phương thức này với phương thức tương ứng đối với DSLK dùng con trỏ và bộ nhớ động trong phần 4.3.2, chúng ta dễ dàng nhận thấy mỗi dòng lệnh là một sự chuyển đổi đơn giản một dòng lệnh của hiện thực trước. Bằng cách chuyển đổi tương tự chúng ta cũng có được phương thức thêm một phần tử mới vào DSLK trong mảng liên tục.

```
template <class Entry>
Error_code List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 <= position <= n, với n là số phần tử hiện có trong
      danh sách, phương thức sẽ thực hiện thành công việc chèn x vào tại position và các
      phần tử phía sau bị đẩy lùi thứ tự bởi 1 đơn vị. Ngược lại, phương thức trả về mã lỗi thích
      hợp.
*/
{
   index new_index, previous, following;

   if (position < 0 || position > count) return range_error;

   if (position > 0) {                                  } Tìm phần tử trước và sau vị trí
      previous = set_position(position - 1);            } cần thêm phần tử mới.
      following = workspace[previous].next;             }
   }
   else following = head;
   if ((new_index = new_node()) == -1) return overflow; // Tìm vùng trống.
   workspace[new_index].entry = x;              // Cập nhật dữ liệu vào vùng trống.
   workspace[new_index].next = following;

   if (position == 0)
      head = new_index; // Trường hợp đặc biệt: thêm                    }
                        vào đầu DSLK.                                   } Thêm phần tử mới
   else                                                                 } vào DSLK.
      workspace[previous].next = new_index;                             }
   count++;
   return success;
}
```

4.5.4. Các biến thể của danh sách liên kết trong mảng liên tục

Mảng liên tục cùng các chỉ số không những được dùng cho DSLK, chúng còn có hiệu quả tương tự đối với DSLK kép hoặc với vài biến thể khác. Đối với DSLK kép, khả năng áp dụng phép tính số học cho các chỉ số cho phép một hiện thực mà trong đó các tham chiếu tới và lui chỉ cần chứa trong một vùng chỉ số đơn (sử dụng cả trị âm lẫn trị dương cho các chỉ số).

CuuDuongThanCong.com https://fb.com/tailieudientucntt