Chương II
DANH SÁCH LIÊN KẾT

Mảng là một cấu trúc dữ liệu cơ bản được sử dụng trong các ngôn ngữ lập trình. Tuy nhiên, nó có ba hạn chế:

1) Kích thước của mảng phải được biết trước tại thời gian biên dịch;

2) Các phần tử được lưu trữ kế tiếp với kích thước như nhau, do đó nếu ta muốn chèn một phần tử vào trong mảng thì phải dịch chuyển các phần tử dữ liệu khác trong mảng;

3) Bộ nhớ được cấp phát cho mảng trong suốt cả quá trình thực hiện chương trình kể cả khi mảng không còn cần thiết nữa cũng không được dùng để cấp phát cho mục đích khác.

Để khắc phục các nhược điểm này chúng ta sử dụng các cấu trúc dữ liệu danh sách liên kết (*linked lists*). Danh sách liên kết là một tập hợp các các phần tử chứa dữ liệu và liên kết giữa các phần tử đó. Theo cách tổ chức này, các phần tử của danh sách có thể được lưu ở bất kì đâu trong bộ nhớ. Có nhiều cách cài đặt danh sách liên kết, nhưng cách phổ biến nhất là sử dụng con trỏ.

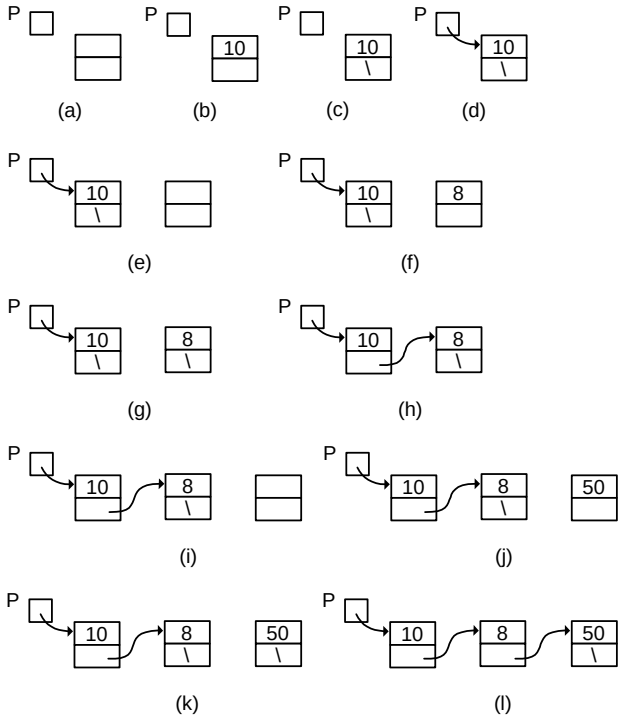
I. Danh sách liên kết đơn

Một danh sách liên kết bao gồm tập các nút (phần tử), mỗi nút chứa một số thông tin và một con trỏ trỏ tới một nút khác trong danh sách. Nếu mỗi nút chỉ có một liên kết tới nút tiếp theo trong danh sách (còn được gọi là hậu duệ của nó), thì danh sách được gọi là danh sách liên kết đơn. Một ví dụ về danh sách liên kết đơn được chỉ ra trong hình 1. Ở đây, nút cuối cùng trong danh sách được nhận biết qua con trỏ có giá trị *null*.

Mỗi nút trong hình là một thực thể (*instance*) của lớp đối tượng được định nghĩa như sau:

```
class IntNode{
   public:
         IntNode() {
               next = NULL;
         }
         IntNode(int i, IntNode* in = NULL){
               info = i;
               next = in;
         }
         int info;
```

```
        IntNode* next;
};
```



Hình 1. Danh sách liên kết đơn

Trong hình 1 mỗi nút bao gồm hai thành phần: *info* và *next*. Trường *info* được sử dụng để lưu trữ dữ liệu, còn trường *next* lưu trữ con trỏ trỏ tới nút tiếp theo trong danh sách. Trường này là một thành phần dữ liệu phụ được sử dụng để duy trì danh sách liên kết. Thành phần này không thể thiếu được trong việc cài đặt danh sách liên kết nhưng nó không có ý nghĩa về mặt thông tin đối với người dùng. Ở đây, lớp *IntNode* được định nghĩa qua chính nó bởi vì *next* là một con trỏ trỏ tới chính lớp *IntNode* đang được định nghĩa.

Lớp *IntNode* có hai hàm khởi tạo, hàm thứ nhất khởi tạo giá trị *null* cho con trỏ *next* và *info* có giá trị không xác định. Hàm khởi tạo thứ hai có hai tham số, tham số đầu tiên được sử dụng để khởi tạo giá trị cho biến *info* và tham số còn lại khởi tạo giá trị cho biến *next*. Hàm khởi tạo thứ hai sử dụng tham số mặc định có thể gồm cả trường hợp người dùng chỉ truyền vào một tham số có kiểu số nguyên thì trong trường hợp này biến *info* sẽ được gán giá trị là giá trị của tham số, còn biến *next* được gán giá trị *null*.

Để tạo ra danh sách liên kết đơn gồm 3 phần tử {10, 8, 50} như trong hình 1, trước tiên chúng ta tạo ra các nút một cách riêng biệt sau đó chèn nó vào danh sách. Quá trình diễn ra như sau:

IntNode p = new IntNode(10);*

Câu lệnh này sẽ tạo ra nút đầu tiên trong danh sách và tạo ra biến con trỏ *p* trỏ tới nút đó, câu lệnh này được thực hiện qua 4 bước:

Bước 1: Một đối tượng *IntNode* mới được tạo (hình 1a).

Bước 2: Thành phần *info* của nút này được gán giá trị 10 (hình 1b).

Bước 3: Trường *next* của nút trỏ tới *null* (hình 1c).

Ở đây con trỏ *null* được đánh dấu bằng dấu sổ chéo (\). Trong đó, bước 2 và 3 khởi tạo các giá trị cho các trường dữ liệu của đối tượng *IntNode* mới qua lời gọi hàm khởi tạo *IntNode(10)* được chuyển đổi thành *IntNode(10, NULL)*.

Bước 4: Gán con trỏ *p* trỏ tới nút mới được tạo ra (hình 1d). Con trỏ này là địa chỉ của nút và được chỉ ra bằng mũi tên đi từ *p* tới nút mới.

Nút thứ hai được tạo ra bằng câu lệnh: *p-> next = new IntNode(8);* trong đó *p->next* là trường *next* của nút được *p* trỏ tới. Quá trình lặp lại 3 bước như trên (hình 1e, f, g), chỉ khác là tại bước thứ tư ta đưa nút mới vào danh sách bằng cách gán con trỏ *next* của nút đầu tiên trỏ tới nút mới (hình 1h). Đối với nút *IntNode(50),* ta làm tương tự: *p->next->next = new IntNode(50);* trong đó *p->next->next* là thành phần *next* của nút thứ hai.

Trong ví dụ này, ta có thể thấy rõ sự bất cập trong việc sử dụng các con trỏ nếu như danh sách liên kết của ta có nhiều phần tử hơn và nếu như ta muốn truy cập tới phần tử thứ *n* nào đó, thì ta sẽ phải nhập tới *n* lần từ *next* như *p->next->next…->next* hoặc phải dùng một vòng lặp để tìm nút cuối cùng.

Để khắc phục điều này, chúng ta sử dụng hai con trỏ cho danh sách liên kết: một con trỏ cho nút đầu tiên và một con trỏ cho nút cuối cùng, như chương trình cài đặt sau đây:

```
//*********************intSLLst.h ******************
// Danh sách liên kết đơn các số nguyên
#ifndef INT_LINKED_LIST
#define INT_LINKED_LIST
class IntNode{
public:
   int info;
   IntNode* next;
```

```
   IntNode(int el, IntNode* ptr = NULL) {
         info = el;
         next = ptr;
   }
};
class IntSLList {
public:
   IntSLList(){
         head = tail = NULL;
   }
   ~IntSLList();
   int isEmpty(){
         return head == NULL;
   }
   void addToHead(int);
   void addToTail(int);
   int deleteFromHead(); //Xoá phần tử ở đầu danh sách
   int deleteFromTail(); //Xoá phần tử ở cuối danh sách
   void deleteNode(int);
   bool isInList(int) const;
private:
   IntNode* head, * tail;
};
#endif

//*******************intSLLst.cpp *********************
#include <iostream.h>
#include "intSLLst.h"
IntSLList:: ~IntSLList(){
   for (IntNode* p; !isEmpty(); ){
         p = head -> next;
         delete head;
         head = p;
   }
}
void IntSLList:: addToHead(){
   head = new IntNode(el, head);
   if (NULL == tail)
         tail = head;
}
void IntSLList:: addToTail(){
   if (NULL != tail){       //Nếu danh sách không rỗng
         tail->next = IntNode(el);
         tail = tail->next;
   }
   else head = tail = new IntNode(el);

}
int IntSLList:: deleteFromHead(){
   int el = head->info;
   IntNode* tmp = head;
```

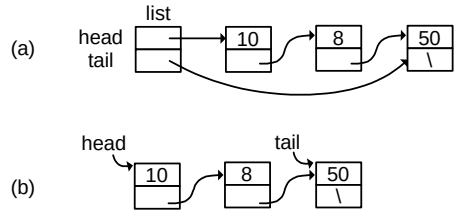
```
   if (head == tail) // Nếu chỉ có một phần tử trong danh sách
         head = tail = NULL;
   else head = head->next;
   delete tmp;
   return el;
}
int IntSLList:: deleteFromTail(){
   int el = tail->info;
   if (head == tail){       // Nếu chỉ có một phần tử trong danh sách
         delete head;
         head = tail = NULL;
   }
   else{              // Nếu trong danh sách có nhiều hơn một phần tử

         IntNode* tmp; // tìm nút trước tail
         for (tmp = head; temp->next != tail; tmp = tmp->next);
         delete tail;
         tail = tmp; // lùi tail lên trước
         tail->next = NULL;
   }
   return el;
}
void IntSLList:: deleteNode(int el) {
   if (NULL != head) // Nếu danh sách không rỗng
         if (head == tail && el == head->info){
// Nếu chỉ có một phần tử trong danh sách
               delete head;
               head = tail = NULL;
         }
         else if (el == head->info){
// Nếu trong danh sách có nhiều hơn một phần tử
               IntNode* tmp = head->next;
               head = head->next;
               delete tmp; // xóa head cũ
         }
         else{        // Nếu trong danh sách có nhiều hơn một phần tử
               IntNode* pred, * tmp;

               // Tìm nút cần xóa
               for (pred = head, tmp = head->next;
               tmp != NULL && !(tmp->info == el);
               pred = pred->next, tmp = tmp->next);

               // Xóa nút
               if (NULL != tmp){
                     pred->next = tmp->next;
                     if (tmp == tail)
                           tail = pred;
                     delete tmp;
```

```
            }
        }
}
bool IntSLList::isInList(int el) const {
   IntNode* tmp;
   for (tmp = head; tmp != NULL && !(tmp->info == el);
         tmp = tmp->next);
   return tmp != NULL;
}
```

Cài đặt danh sách liên kết đơn trên sử dụng hai lớp: lớp *IntNode* lưu trữ các nút trong danh sách và lớp *IntSLList* để truy cập danh sách. Lớp *IntSLList* định nghĩa hai thành phần dữ liệu con trỏ là *head* và *tail*, trỏ tới đầu và cuối danh sách. Để các phương thức của lớp *IntSLList* truy cập được thành phần dữ liệu trong *IntNode* thì các thành phần này phải được khai báo là **`public`**. Một giải pháp khác trong C++ là khai báo *IntSLList* là *lớp bạn* của *IntNode* bằng từ khoá **`friend`**. Do các nút của danh sách có thể truy cập thông qua các con trỏ, nên để đảm bảo nguyên lí che dấu thông tin các biến *head* và *tail* của *IntSLList* được khai báo là **`private`**.

Ví dụ về danh sách liên kết được chỉ ra trong hình 2a, hình 2b là cách vẽ giản lược tương ứng. Danh sách này được khai báo bằng câu lệnh:

```
IntSLList list;
```



Hình 2. Danh sách liên kết đơn các số nguyên

Ngoài hai thuộc tính *head* và *tail*, lớp *IntSLList* còn định nghĩa các hàm thành viên cho phép chúng ta thao tác với danh sách. Sau đây chúng ta đi sâu vào tìm hiểu một số thao tác cơ bản trên danh sách được minh hoạ trong chương trình cài đặt trên.

1. Chèn phần tử vào danh sách liên kết

Để chèn một phần tử vào đầu danh sách liên kết, ta thực hiện theo 4 bước:

Bước1: Một nút rỗng được tạo ra theo nghĩa chương trình không gán bất cứ giá trị nào cho các thành phần dữ liệu của nút (hình 3a).

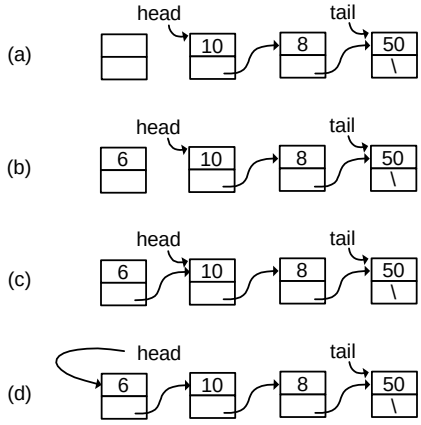
Bước2: Thành phần *info* được gán một giá trị cụ thể (hình 3b).

Bước3: Do nút được gắn vào đầu danh sách, nên thành phần *next* của nó phải trỏ tới nút đầu tiên trong danh sách và đó cũng là giá trị hiện tại của *head* (hình 3c).

Bước 4: Nút mới này nằm trước tất cả các nút trong danh sách, do đó *head* được cập nhật để trỏ tới nút mới này (hình 3d).

Bốn bước này được thực thi bởi hàm thành viên *addToHead* (chương trình cài đặt danh sách đơn các số nguyên). Hàm này thực thi 3 bước đầu gián tiếp thông qua lời gọi hàm khởi tạo *IntNode(el, head)*. Bước cuối cùng được thực thi một cách trực tiếp trong hàm bằng cách gán cho *head* địa chỉ của nút mới được tạo ra.

Hàm *addToHead()* còn xử lí một trường hợp đặc biệt, đó là chèn một nút vào danh sách liên kết rỗng. Trong danh sách liên kết rỗng, cả hai con trỏ *head* và *tail* đều là *null*, do đó cả hai đều trỏ tới cùng một nút trong danh sách mới. Khi chèn vào đầu danh sách liên kết không rỗng, chỉ có con trỏ *head* cần được cập nhật.



Hình 3. Chèn một nút mới vào đầu danh sách liên kết đơn

Quá trình chèn một nút vào cuối danh sách liên kết bao gồm 5 bước:

Bước 1: Tạo một nút rỗng (hình 4a).

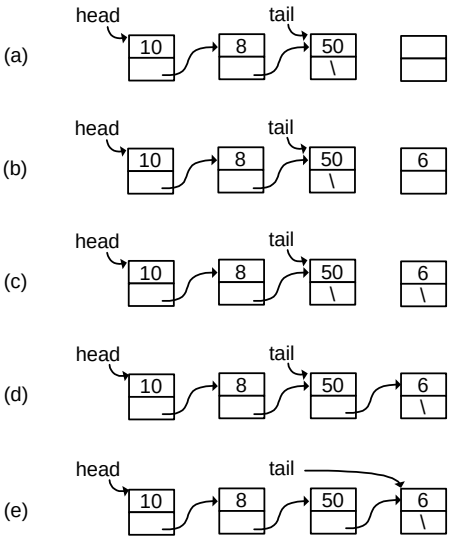
Bước 2: Thành phần *info* của nút được khởi tạo giá trị *el* (hình 4b).

Bước 3: Do nút được chèn vào cuối danh sách liên kết, nên thành phần *next* được gán bằng *null* (hình 4c).

Bước 4: Nút được chèn vào danh sách liên kết bằng cách cho thành phần *next* của nút cuối trong danh sách trỏ tới nút mới được tạo (hình 4d).

Bước 5: Nút mới đi sau tất cả các nút trong danh sách, nên con trỏ *tail* bây giờ trỏ tới nút mới (hình 4e).

Tất cả các bước được thực thi trong mệnh đề **`if`** của *addToTail*() (chương trình cài đặt danh sách liên kết đơn các số nguyên). Mệnh đề **`else`** chỉ được thực thi khi danh sách liên kết là rỗng. Nếu không đưa trường hợp này vào thì chương trình có thể bị phá huỷ bởi vì trong mệnh đề **`if`** ta dùng thành phần *next* của con trỏ *tail* mà trong trường hợp danh sách liên kết rỗng, nó là một con trỏ *null*.



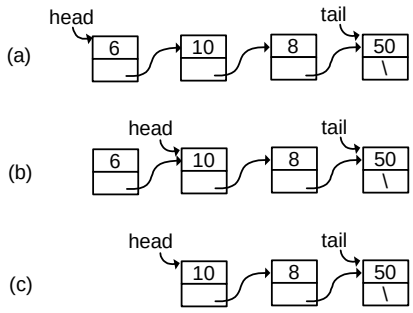
Hình 4. Chèn một nút mới vào cuối danh sách liên kết đơn

Quá trình chèn một nút vào đầu danh sách liên kết tương tự như quá trình chèn một nút vào cuối danh sách liên kết. Bởi vì việc cài đặt *IntSLList* sử dụng hai thành phần biến con trỏ là *head* và *tail*. Vì lí do này, cả hai hàm *addToHead()* và *addToTail*() đều có thời gian thực thi là O(1), nghĩa là không phụ thuộc vào số lượng nút trong danh sách, số lượng các toán tử được thực hiện bởi hai hàm này không vượt quá một hằng số nào đó.

2. Xoá nút khỏi danh sách liên kết

a) Xoá nút đầu danh sách liên kết

Phép toán xoá một nút khỏi đầu danh sách liên kết và trả lại giá trị chứa trong nút đó được cài đặt bởi hàm *deleteFromHead()*. Trong phép toán này, thông tin của nút đầu tiên được lưu trữ tạm thời trong biến cục bộ *el* và sau đó biến *head* được thiết lập lại sao cho nút thứ hai trở thành nút đầu tiên của danh sách. Theo cách này, nút đầu tiên bị xoá với thời gian thực thi là O(1) (hình 5).



Hình 5. Xoá một nút khỏi đầu danh sách liên kết đơn

Khi xoá nút ở đầu danh sách, chúng ta cần xem xét hai trường hợp ngoại lệ là xoá nút từ danh sách rỗng và xoá nút từ danh sách có duy nhất một nút.

Trường hợp thứ nhất, chúng ta cố gắng loại một nút khỏi danh sách liên kết rỗng, nếu không có đoạn mã xử lí riêng, rất có thể chương trình sẽ thao tác sai và gây ra các lỗi nghiêm trọng. Vấn đề tiếp theo phải giải quyết là trong trường hợp đó thì giá trị trả về là như thế nào để chương trình gọi nó tiếp tục hoạt động bình thường. Có một vài cách để tiếp cận để giải quyết vấn đề này nhưng về cơ bản là trước khi xoá chúng ta cần kiểm tra xem danh sách có rỗng hay không. Ví dụ, thêm hàm *isEmpty()* vào lớp *IntSLList* và sử dụng nó như sau:

```
if (!List.isEmpty())
   n = list.deleteFromHead();
else { // không xoá
…
}
```

Một giải pháp an toàn hơn mà người lập trình C++ ưa thích là dùng cấu trúc ném và bắt ngoại lệ, như sau:

```
int IntSLList:: deleteFromHead() {
   if (isEmpty())
         throw("Empty");
   int el = head->info;
   ……
   return el;
}
```

Lệnh *throw* ném ra một ngoại lệ kiểu xâu kí tự cần phải đi cùng với lệnh *catch* với tham số kiểu tương ứng để bắt ngoại lệ đó như trong đoạn mã sau:

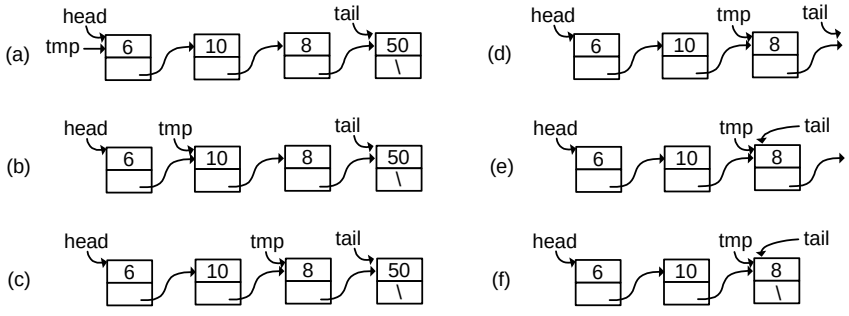
```
void foo(){
   ……
   try{
         n = list.deleteFromHead();
         // thao tác với n
```

```
        ……
    } catch(char* s){
        cerr << "Error: " << s << endl;
    }
    ……
}
```

Trường hợp đặc biệt thứ hai cần xét là ta cần xoá một nút khỏi danh sách liên kết chỉ có một phần tử. Trong trường hợp này, hai con trỏ *head* và *tail* phải được đặt là *null*.

b) Xoá nút cuối danh sách liên kết

Xoá một nút ở cuối danh sách liên kết và trả về giá trị của nó được thực hiện qua hàm *deleteFromTail*(). Sau khi xoá một nút, *tail* phải trỏ tới đuôi mới của danh sách, nghĩa là *tail* phải di chuyển ngược trở lại một nút. Nhưng việc di chuyển ngược lại là không thể vì không có liên kết trực tiếp nào từ nút cuối cùng tới nút trước nó. Do đó, để tìm ra nút trước nó ta phải duyệt từ đầu danh sách liên kết và dừng ngay trước nút do *tail* trỏ tới. Ở đây ta sử dụng biến tạm thời *tmp* để quét danh sách liên kết trong vòng lặp *for*. Biến *tmp* được khởi tạo trỏ tới đầu danh sách và sau đó trong mỗi bước lặp, nó đứng trước nút tiếp theo. Nếu danh sách liên kết như trong hình 6a thì đầu tiên *tmp* sẽ trỏ tới nút đầu danh sách chứa số (6). Sau khi thực thi câu lệnh gán *tmp=tmp->next*, *tmp* trỏ tới nút thứ hai (hình 6b). Sau bước lặp thứ hai, *tmp* trỏ tới nút thứ ba (hình 6c). Bởi vì nút này cũng là nút đứng cạnh nút cuối cùng, nên vòng lặp thoát ra ngoài, sau đó nút cuối cùng được xoá (hình 6d). Bởi vì *tail* bây giờ đang trỏ tới một nút không tồn tại, nên ngay lập tức nó được thiết lập để trỏ đến nút nằm liền kề nút cuối cùng trong danh sách liên kết được tham chiếu bởi *tmp* (hình 6e). Để đánh dấu đây là nút cuối cùng trong danh sách liên kết thì trường *next* của nút này phải được đặt là *null*.



Hình 6. Xoá một nút khỏi đuôi danh sách liên kết đơn

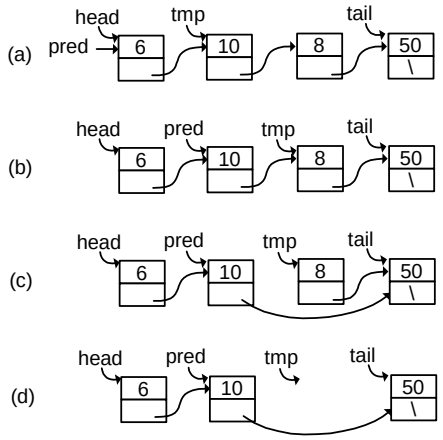
Trong việc loại bỏ nút cuối cùng, có hai trường hợp đặc biệt. Nếu danh sách liên kết rỗng thì không có nút nào được loại khỏi danh sách, như vậy ở đây người dùng phải xử lí các ngoại lệ có thể xảy ra. Nếu danh sách liên kết có duy nhất một phần tử, thì sau khi loại bỏ phần tử đó, các biến con trỏ *head* và *tail* phải được gán giá trị *null*.

Phép toán tốn nhiều thời gian trong hàm *deleteFromTail()* là việc duyệt danh sách liên kết để tìm ra nút đứng ngay trước nút cuối cùng. Rõ ràng là nếu danh sách liên kết có *n* nút thì vòng lặp sẽ phải thực hiện *n* - 1 bước lặp, do đó hàm này mất một khoảng thời gian là O(*n*) để xoá nút cuối cùng.

c) Xoá nút ở vị trí bất kì

Phép toán xoá thứ ba là xoá một nút có giá trị cho trước. Một cách ngắn gọn, trước tiên ta phải định vị nút đó trong danh sách thông qua giá trị của nó, sau đó loại nút này khỏi danh sách bằng cách liên kết nút trước nó và nút sau nó. Do ta không biết chính xác vị trí của nút trong danh sách, nên quá trình tìm và xoá một nút có giá trị cho trước sẽ phức tạp hơn nhiều so với hai thao tác xoá đã trình bày. Hàm *deleteNode()* (chương trình cài đặt danh sách liên kết đơn các số nguyên) là một cài đặt cho phép toán này.

Do danh sách liên kết đi theo một chiều, vì vậy chúng ta cần tìm định vị cả nút cần xoá và nút đứng trước nó. Để thực hiện điều này, như trong hình 7, giả sử ta muốn xoá nút có giá trị (8) ta sử dụng hai con trỏ *pred* và *tmp* được khởi tạo giá trị trong vòng lặp *for* sao cho chúng trỏ tới nút đầu tiên và thứ hai của danh sách (hình 7a). Do nút *tmp* có giá trị (5), nên bước lặp đầu tiên được thực thi, cả hai biến *pred* và *tmp* được chuyển sang các nút tiếp theo (hình 7b). Bởi vì điều kiện bây giờ của vòng lặp là *true*, nên vòng lặp thoát và câu lệnh *pred->next = tmp->next* được thực hiện (hình 7c). Việc gán này loại bỏ hoàn toàn nút có giá trị (8) khỏi danh sách, nhưng nó vẫn có thể được truy cập thông qua biến *tmp*. Câu lệnh *delete tmp* dùng để giải phóng thực sự không gian nhớ đã được cấp cho nút này (hình 7d).



Hình 7. Xoá một nút khỏi danh sách liên kết đơn

Các sơ đồ trong hình trên chỉ đề cập đến trường hợp thông thường, dưới đây là các trường hợp đặc biệt:

1. Xoá một nút khỏi danh sách liên kết rỗng, trong trường hợp đó hàm thoát ra ngoài ngay lập tức.
2. Xoá một nút khỏi danh sách liên kết một phần tử: cả hai con trỏ *head* và *tail* phải được thiết lập là *null*.
3. Xoá nút đầu tiên khỏi danh sách liên kết có ít nhất hai phần tử, yêu cầu cập nhật con trỏ *head*.
4. Không có nút có giá trị như giá trị cần xoá: không làm gì cả.

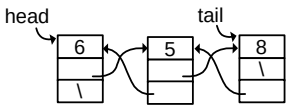
3. Tìm kiếm

Phép toán tìm kiếm thực hiện việc duyệt danh sách liên kết để xác định liệu một số có nằm trong danh sách liên kết hay không. Chúng ta cài đặt phép toán này bởi hàm Boolean *isInList()*. Hàm này sử dụng biến tạm thời *tmp* để đi qua danh sách liên kết từ đầu danh sách. Giá trị nằm trong mỗi nút được so sánh với số đang tìm, nếu hai giá trị này bằng nhau thì vòng lặp thoát. Ngược lại, biến *tmp* chuyển sang nút tiếp theo *tmp->next* để tiếp tục quá trình tìm kiếm. Sau khi chạm tới nút cuối cùng và thực hiện phép gán *tmp=tmp->next*, *tmp* trở thành *null*, điều đó đồng nghĩa với việc số *el* không nằm trong danh sách liên kết. Đó là lí do tại sao hàm *isInList()* trả lại kết quả của phép so sánh *tmp != NULL*. Nếu *tmp* không bằng *null*, *el* được tìm thấy và giá trị *true* được trả lại, ngược lại tìm kiếm thất bại và giá trị *false* được trả lại.

Lập luận tương tự phần trên ta thấy, hàm *isInList()* có thời gian thực hiện trong trường hợp tốt nhất là O(1) và trong trường hợp xấu nhất là O(*n*).

II. Danh sách liên kết kép

Hàm *deleteFromTail*() chỉ ra một vấn đề trong danh sách liên kết đơn là không thể truy cập trực tiếp tới phần tử đứng trước nút cuối cùng trong danh sách liên kết, mà phải duyệt toàn bộ danh sách để tìm ra nút đứng trước đó dẫn đến tăng thời gian xử lí trên danh sách. Để khắc phục trở ngại này, danh sách liên kết được định nghĩa lại sao cho mỗi nút trong danh sách liên kết có hai con trỏ, một con trỏ trỏ tới nút đứng trước và một con trỏ tới nút đứng sau nút đó. Danh sách liên kết kiểu này gọi là danh sách liên kết kép được minh hoạ trong hình 8.



Hình 8. Danh sách liên kết kép

Dưới đây là chương trình cài đặt danh sách liên kết kép *DoublyLinkedList* tổng quát. Hàm này viết bằng C++ sử dụng khuôn mẫu (*template*).

```
#ifndef DOUBLY_LINKED_LIST
#define DOUBLY_LINKED_LIST
#include <iostream.h>
template<class T>
class Node{
public:
   Node(){
         next = prev = NULL;
   }
   Node(const T& el, Node* n = NULL, Node* p = NULL){
         info = el; next = n; prev = p;
   }
   T info;
   Node* next, * prev;
};
template<class T>
class DoublyLinkedList(){
public:
   DoublyLinkedList(){
         head = tail = NULL;
   }
   void addToDLLTail(const T&);
   T deleteFromDLLTail();
   ……
protected:
   Node<T>* head, * tail;
```

```
};

template<class T>
void DoublyLinkedList<T>:: addToDLLTail(const T& el){
   if (tail != NULL){
         tail = new Node<T>(el, NULL, tail);
         tail->prev->next = tail;
   }
   else head = tail = new Node<T>(el);
}
template<class T>
T DoublyLinkedList<T>::deleteFromDLLTail(){
   T el = tail->info;
   if (head == tail){       // Nếu trong danh sách chỉ có một phần tử
         delete head;
         head = tail = NULL;
   }
   else {             // Nếu trong danh sách có nhiều hơn một phần tử
         tail = tail->prev;
         delete tail->next;
         tail->next = NULL;
   }
   return el;
}
……
#endif
```

Các hàm xử lí danh sách liên kết kép phức tạp hơn so với danh sách liên kết đơn bởi vì có nhiều hơn một con trỏ. Ở đây chúng ta chỉ xem chi tiết về hai hàm: hàm chèn một nút vào cuối danh sách liên kết và hàm loại bỏ phần tử ở cuối danh sách liên kết.

1. Thêm nút vào danh sách liên kết kép

Để thêm vào một nút trong danh sách liên kết, một nút mới phải được tạo ra, các thành phần dữ liệu của nó được khởi tạo thích hợp và sau đó nút được chèn vào danh sách. Việc chèn một nút vào cuối danh sách liên kết kép được thực hiện bởi hàm *addToDLLTail*() được minh hoạ trong hình 9. Quá trình chèn được thực hiện qua 6 bước:

Bước 1: Một nút mới được tạo ra (hình 9a) và sau đó 3 thành phần dữ liệu của nó được khởi tạo.

Bước 2: Thành phần *info* được khởi tạo giá trị *el* là giá trị cần chèn (hình 9b).

Bước 3: Thành phần *next* được gán bằng *null* (hình 9c).

Bước 4: Thành phần *prev* được gán giá trị của *tail* sao cho con trỏ này chỉ tới nút cuối cùng trong danh sách (hình 9d).