Giải Thuật Lập Trình

Nơi tổng hợp và chia sẻ những kiến thức liên quan tới giải thuật nói chung và lý thuyết khoa học máy tính nói riêng.

< Giải thuật cho hệ thống gợi ý với dữ liệu lớn -- A Recommendation Algorithm for Big Data • Lý thuyết phổ đồ thị I -- Spectral Graph Theory I >

Danh sách liên kết đơn và các biến thể -- Singly linked list and its variants

July 24, 2016 in Uncategorized | No comments

Danh sách liên kết (DSLK) đơn (Singly linked list) là một cấu trúc dữ liệu cơ bản và có cực kì nhiều ứng dụng. Ý tưởng của DSLK đơn khá đơn giản. Tuy nhiên, để cài đặt cấu trúc DSLK đơn một cách ngắn gọn, hiệu quả thì không phải là điều hiển nhiên. Trong bài này, chúng ta sẽ thảo luận các cách cài đặt danh sách liên kết (trong C) sao cho **hiệu quả** và **ngắn gọn**.

Cài đặt DSLK trong C không thể tránh khỏi sử dụng con trỏ (pointer) và struct. Con trỏ là một kiểu dữ liệu cực kì mạnh của ngôn ngữ C. Nó cho phép người lập trình thao tác bộ nhớ rất hiệu quả. Những lập trình viên giỏi luôn là những lập trình viên biết thao tác con trỏ một cách thuần thục. Tuy nhiên, điểm yếu của con trỏ là khó học, khó nắm bắt và cũng khó debug các chương trình có con trỏ, nhất là đối với những người lần đầu tiếp xúc với khái niệm con trỏ.

Trong bài này, mình cũng không có gắng giải thích ý nghĩa, mặc dù có nhắc lại, con trỏ. Mình khuyến khích các bạn tự tìm hiểu, vì có rất nhiều nguồn, kể cả tiếng Anh và tiếng Việt. Phần cuối bài mình sẽ liên kết một số bài viết hay về con trỏ. Mục tiêu của bài này là minh họa tối đa khả năng kì diệu của con trỏ trong thao danh sách liên kết. Các bài khác trong blog mình thường minh họa bằng giả mã. Tuy nhiên, bài này mình sẽ trực tiếp sử dụng code C.

Con trỏ và struct

Một con trỏ (pointer) là một biến dùng để lưu trữ **địa chỉ** của một biến khác. Biến khác ở đây có thể là một biến thông thường như biến số nguyên, biến số thực, hoặc có thể là một mảng, một hàm, và có khi cũng chính là một con trỏ khác. Vì con trỏ là một biến nên nó cũng cần phải được lưu trữ ở đâu đó trong bộ nhớ; có nghĩa là bản thân biến con trỏ cũng có một địa chỉ trong bộ nhớ. Do đó, ta có thể dùng con trỏ để lưu địa chỉ của một con trỏ khác. Tính chất này cũng chính là sự kì diệu của con trỏ mà mình sẽ minh họa trong bài này.

Cú pháp khai báo con trỏ trong C, nói một cách đơn giản (nhưng không hoàn toàn chính xác), gồm 3 phần: (i) kiểu của biến khác mà con trỏ lưu trữ địa chỉ, (2) dấu * và (3) tên của con trỏ. Con trỏ hàm thì khai báo hơi khác một chút, nhưng đây không phải vấn đề trọng tâm của bài viết. Ví dụ, khai báo một số con trỏ:

```
int *a; // pointer to an integer
float *b; // pointer to a float
char *c; // pointer to character
```

Khai báo thì như vậy, nhưng ý nghĩa của con trỏ thì không phải lúc nào cũng nhất quán. Ví dụ con trỏ **int *a** vừa có thể hiểu là con trỏ tới một biến số nguyên, vừa có thể hiểu là con trỏ tới phần tử đầu tiên của một mảng số nguyên.

Struct trong C cho phép chúng ta tập hợp một hoặc một vài kiểu dữ liêu khác nhau thành một kiểu dữ liệu mới. Các kiểu dữ liệu thành phần của một struct có thể là kiểu dữ liệu sẵn có như số nguyên, số thực, con trỏ, hoặc một kiểu struct đã định nghĩa trước đó. Tóm lại, struct cho phép chúng ta "gộp" một số kiểu dữ liệu đã có lại với nhau để tiện thao tác.

Trong DSLK, struct được sử dụng để khai báo một "mắt xích" của danh sách. Ví dụ ta muốn khai báo một mắt xích của một danh sách liên kết các biến kiểu int thì ta có thể khai báo như sau:

```
typedef struct llnode{
    int x;
    struct llnode *next;
} llnode;
```

Biến x trong khai báo trên là dữ liệu của mỗi mắt xích và nó có kiểu int. Kiểu của x còn có thể là con trỏ, hay một struct. Mỗi mắt xích thường được minh họa như trong Figure 1, trong đó, mũi tên ám chỉ con trỏ lưu địa chỉ của mắt xích tiếp theo (con trỏ next). Con trỏ này trong mắt xích cuối cùng của danh sách thường là NULL.

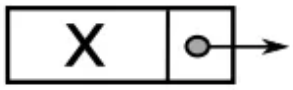

Figure 1: Biểu diễn một mắt xích của danh sách liên kết.

Để khởi tạo một mắt xích, ta sẽ dùng hàm malloc (viết tắt của **m**emory **alloc**ation). Malloc là một hàm quản lí bộ nhớ của C. Bản thân hàm này cũng khá phức tạp. Bạn đọc xem thêm tại liên kết ở cuối bài.

```
llnode * create_new_node(int datum){
  llnode * node = (llnode *)malloc(sizeof(llnode)); // allocate m
  node->x = datum;
  node->next = NULL;
  return node;
}
```

Danh sách liên kết đơn

Danh sách liên kết đơn, về mặt trực quan, là cấu trúc dữ liệu tuyến tính giống như một cái xích dài liên kết các "mắt xích" với nhau. Mỗi mắt xích có dạng khai báo llnode ở trên. Figure 2 minh họa một danh sách liên kết với 5 phần tử.

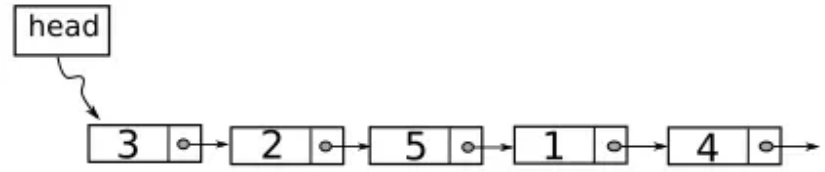

Figure 2: Một danh sách liên kết với 5 phần tử.

Để theo dõi danh sách, ta sẽ dùng một con trỏ đặc biệt, gọi là con trỏ head. Con trỏ này lưu trữ **địa chỉ** của mắt xích đầu tiên của danh sách. Như đã nói ở trên, con trỏ của mắt xích cuối cùng của danh sách sẽ có giá trị NULL. Ta có thể duyệt qua danh sách này bằng cách bắt đầu từ phần tử đầu tiên, đi theo con trỏ liên kết để đi tới nút tiếp theo. Đến khi ta gặp con trỏ NULL thì ta đã duyệt xong danh sách. Đoạn code dưới đây là thủ tục walk_down để duyệt danh sách.

```
1  void walk_down(llnode *head){
2     while(head->next != NULL){
3        printf("%d," head->x);
4        head = head->next;   // walk to the next node
5    }
6  }
```

Trong đoạn code trên, liệu ta có bị mất con trỏ head? Câu trả lời là không vì mỗi lần gọi một hàm walk_down như trên, một bản "copy" của con trỏ head sẽ được tạo ra, và hàm walk_down sẽ thay đổi bản copy này. Bản gốc vẫn không thay đổi. Nếu bạn thay đổi con trỏ head trong hàm main (hàm mà bạn tạo ra con trỏ này) hoặc con trỏ head là một biến toàn cục, thì con trỏ head sẽ bị thay đổi, hay bị mất. Điều gì sẽ xảy ra nếu như ta "làm mất" con trỏ head? Bạn sẽ mất dấu của danh sách và do đó, sẽ không thể thao tác được với danh sách nữa. Phần bộ nhớ của danh sách lúc này sẽ vẫn bị chiếm bởi danh sách, do đó, tạo ra rác (garbage) trong hệ thống. Phần rác này tồn tại cho đến khi chương trình kết thúc. Bài học rút ra: khi nào bạn không còn cần danh sách nữa thì sử dụng hàm free để giải phóng bộ nhớ, đừng bao giờ để mất dấu con trỏ head vì nó sẽ tạo ra rác.

Thêm phần tử mới vào danh sách liên kết

Khi thêm phần tử mới vào danh sách, nếu bài toán không có yêu cầu gì đặc biệt, thì bạn nên **thêm vào đầu** của danh sách. Nếu bạn thêm vào đuôi của danh sách (theo tư duy thông thường), thì ngoài phải trả thêm bộ nhớ để lưu trữ con trỏ đuôi của danh sách (nếu không lưu con trỏ này thì sẽ phải duyệt rất tốn thời gian), bạn phải xét trường hợp khi danh sách rỗng (con trỏ đuôi là NULL), i.e, thêm if-then trong code. Điều này làm code vừa chậm vừa "tối sủa" (xem code ví dụ dưới đây).

```
1  llnode * head; // head is a global variable
2  llnode* insert_to_tail(llnode *tail, int datum){
3     llnode *node = create_new_node(datum); // create a new node
4     if( tail == NULL) { // the list is empty
5         tail = node;
6         head = tail;
7     } else{
8         tail->next = node;
9         tail= node;
10    }
11    return tail;
12 }
```

Tóm lại, luôn thêm vào đầu danh sách nếu có thể. Đoạn code sau minh họa thao tác chèn vào đầu. Rõ ràng đoạn code dưới đây đẹp hơn rất nhiều.

```
1  llnode * head; // head is a global variable
2  void insert_to_head(int datum){
3     llnode *node = create_new_node(datum); // create a new node
4     node->next = head; // dont need to care whether head is null o
5     head = node;
6  }
```

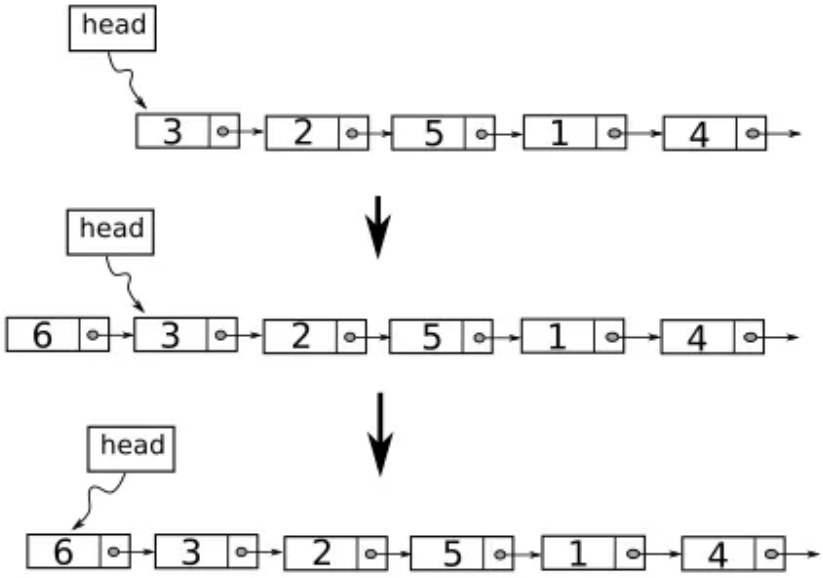

Figure 3: Các bước chèn một mắt xích có giá trị 6 vào đầu DSLK trong Figure 2.

Xóa một phần tử khỏi danh sách liên kết

Giả sử bạn muốn xóa một mắt xích có trường dữ liệu có giá trị y ra khỏi danh sách, bạn có thể làm như sau (xem minh họa trong Figure 4):

1. Duyệt danh sách (từ đầu) để tìm ra mắt xích có giá trị y. Trong quá trình duyệt, bạn sẽ lưu trữ con trỏ prev, để trỏ tới mắt xích cha của mắt xích hiện tại. Mắt xích cha được hiểu là mắt xích có con trỏ trỏ vào nút hiện tại.
2. Cập nhật con trỏ prev để trỏ vào mắt xích con của mắt xích có dữ liệu y.

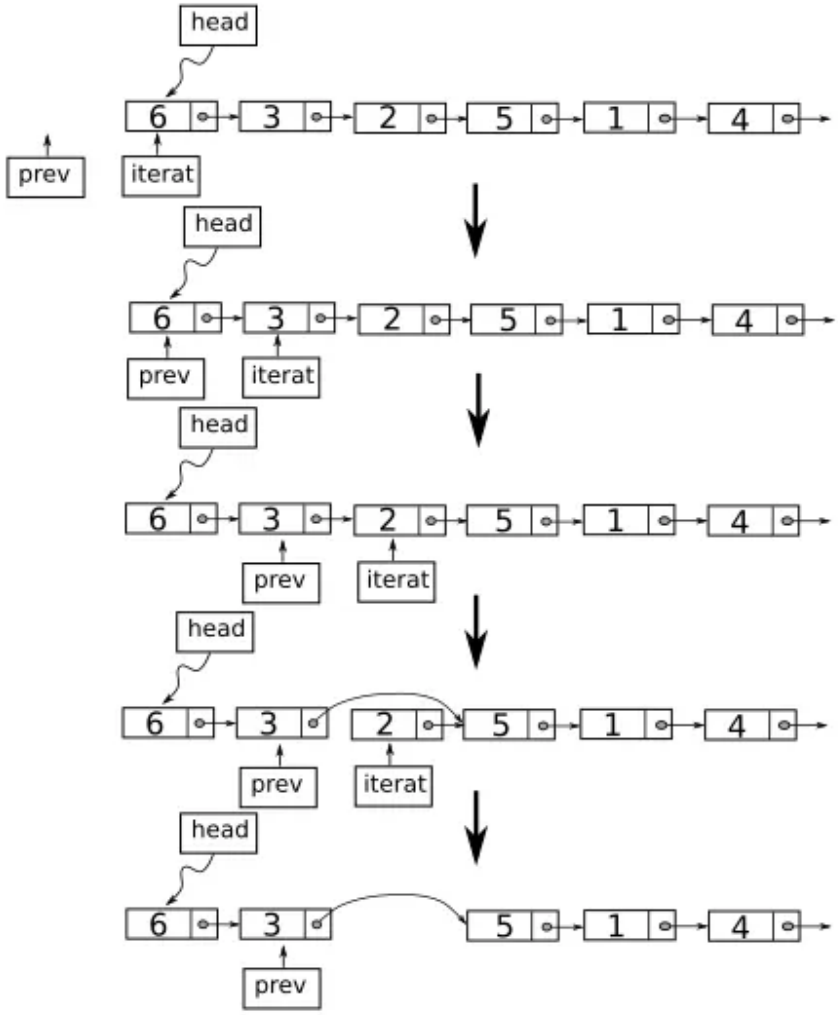

Figure 4: Các bước xóa mắt xích có giá trị 2 ra khỏi danh sách.
Code:

```
1  // Warning: this piece of code is BUGGY
2  void buggy_remove(int datum){
3     llnode *prev = NULL;
4     llnode *iterator = head;
5     while(iterator->x != datum){
6        prev = iterator;
7        head = iterator->next;
8     }
9     prev->next = iterator->next;
10    free(iterator); // officially remove the node having value da
11 }
```

Đoạn code trên chỉ minh họa ý tưởng, và nó là đoạn code có bug, nghĩa là nó chỉ thành công trong một số trường hợp. Điều gì sẽ xảy ra nếu ta chạy đoạn code trên trong trường hợp mắt xích đầu tiên có giá trị datum? Khi đó đoạn code trong vòng while sẽ không thực hiện, và kết quả là prev vẫn là NULL sau vòng lặp while. Đến đây bạn có thể gặp lỗi Segmentation fault. Bạn có thể sửa đổi code trên như sau:

```
1  // Warning: this piece of code is possibly buggy
2  void good_but_not_clean_remove( int datum){
3     llnode *prev = NULL;
4     llnode *iterator = head;
5     while(iterator->x != datum){
6        prev = iterator;
7        iterator = iterator->next;
```

```
    }
    if( prev == NULL){ // datum is the head
        head = head->next;
    } else {
        prev->next = iterator->next;
    }
    free(iterator); // officially remove the node having value da
}
```

Đoạn code này ok, có thể không có lỗi, nhưng trông không được "sạch sẽ" và chậm vì có if-then. Cách tốt hơn, không có if-then, đó là thao tác trên **địa chỉ của con trỏ**. Phương pháp này minh họa cách sử dụng con trỏ rất thông minh. Hiểu đoạn code này có thể khó nhưng một khi đã hiểu thì bạn sẽ thấy cái hay của nó.

```
void remove( int datum){
   llnode **iterator = &(head); // take the address of the head p
   while(*(iterator)->x != datum){
      iterator = &((*iterator)->next);
   }
   *iterator = (*iterator)->next;
}
```

Nhắc lại, toán tử & là toán tử lấy địa chỉ của một biến, còn toán tử * là toán tử lấy giá trị của biến có địa chỉ lưu trong con trỏ. Nhìn vào đoạn code trên, sự khác biệt so với đoạn code good_but_not_clean_remove là chúng ta thao tác trên **địa chỉ** của con trỏ, thay vì thao tác trên con trỏ. Đoạn code trên được sửa đổi từ bài viết ở đây (https://medium.com/@bartobri/applying-the-linus-tarvolds-good-taste-coding-requirement-99749f37684a#.hrfndzt1s) .

Đôi khi trong một số trường hợp, ta muốn xóa một nút khỏi danh sách khi mà ta biết trước địa chỉ của nút đó. Cụ thể, ta muốn đoạn code tương tự như sau:

```
void remove_current_node(llnode * current_node){
   // delete currrent_node
}
```

Nếu chỉ đơn giản gán current_node = NULL thì đoạn code trên sẽ không thực hiện đúng như mong muốn (bạn đọc nên thử và giải thích tại sao). Có hai cách để làm. Cách làm không mấy sạch sẽ là copy dữ liệu từ nút mắt xích sau đó vào nút hiện tại và ta xóa nút sau đó.

```
void remove_by_copy_data(llnode * current_node){
   llnode *next_node = current_node->next;
   current_node->x = next_node->x;  // copy data from next to cur
   current_node->next = next_node->next;
   free(next_node); // delete next_node
}
```

Tuy nhiên, cách này không áp dụng được nếu current_node là mắt xích cuối cùng của DSLK (bạn đọc có thể thử để kiểm tra). Cách tốt hơn, lấy ý tưởng hàm remove ở trên sử dụng địa chỉ của con trỏ, ta sẽ truyên vào hàm địa chỉ của current_node.

```
void remove_by_copy_address(llnode **current_node){
   *current_node = (*current_node)->next;
}
```

Cách này áp dụng được ngay cả khi current_node là mắt xích cuối cùng của DSLK.

Một số biến thể của DSLK đơn

Danh sách liên kết đôi

Biến thể đầu tiên là DSLK đôi (doubly linked list), trong đó, mỗi mắt xích có 2 con trỏ. Con trỏ đầu tiên trỏ vào mắt xích trước nó trong DSLK và con trỏ thứ hai trỏ vào mắt xích sau đó trong DSLK.

```
1  typedef struct dllnode{
2      int x;
3          struct dllnode *prev;
4      struct dllnode *next;
5  } dllnode;
```

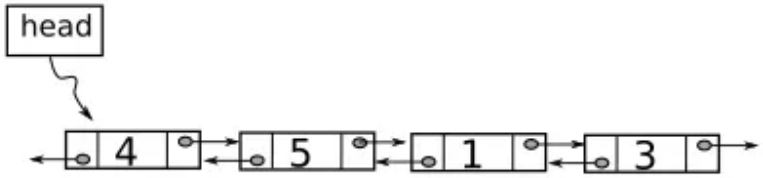

Figure 5: Một DSLK đôi với 4 mắt xích.

DSLK đôi so với DSLK đơn cần nhiều bộ nhớ hơn để lưu trữ thêm một con trỏ. Các thao tác xóa và thêm mới cũng lâu hơn vì ta còn phải cập nhật cả con trỏ prev mỗi khi ta cần xóa hoặc thêm vào DSLK. Về điểm mạnh, DSLK đôi mềm dẻo hơn vì từ một nút, ta có thể đi đến nút trước nó mà không phải duyệt từ đầu DSLK. Tính mềm dẻo này được Knuth [2] khai thác triệt để trong thiết kế cấu trúc Dancing Links, một cấu trúc dữ liệu hỗ trợ các thuật toán quay lui cho bài toán liệt kê tổ hợp.

Ngoài ra, ta có thể không cần con trỏ head để xác định nút đầu tiên của danh sách như DSLK đơn (tại sao?). Nếu con trỏ head bị mất, ta vẫn có thể tìm được phần tử đầu tiên của DSLK nếu ta được phép truy nhập đến một phần tử bất kì của DSLK. DSLK đôi cũng chính là cấu trúc đằng sau LinkedList (http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html) của Java.

Danh sách liên kết vòng

Danh sách liên kết vòng (circularly linked list) là một biến thể khác của DSLK đơn, trong đó con trỏ next của mắt xích cuối cùng của danh sách trỏ vào mắt xích đầu tiên của danh sách để tạo thành một vòng tròn (xem Figure 6). Với cấu trúc này, khái niệm đầu và cuối không thực sự có ý nghĩa. Do đó, ta thường chỉ lưu một con trỏ đặc biệt để trỏ vào một nút nào đó trong danh sách.

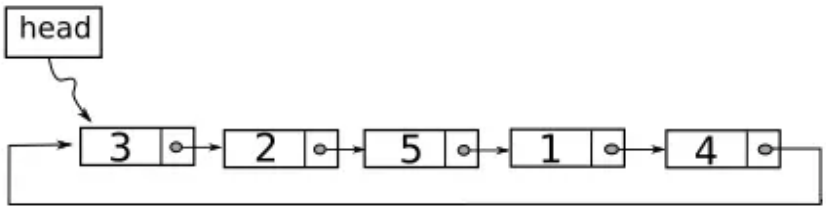

Figure 6: Một DSLK vòng với 5 mắt xích.

Danh sách liên kết vòng thường được dùng trong các ứng dụng trong đó các thành phần tham gia được thực thi theo lượt. Ví dụ trong các game đánh bài khi mỗi người chơi được phép đi một lượt theo vòng. Một bài tập hay ứng dụng danh sách liên kết vòng là bài toán Josephus (https://vi.wikipedia.org/wiki/B%C3%A0i_to%C3%A1n_Josephus).

Liên kết về con trỏ trong C

Chú ý: Phần này sẽ liên tục được cập nhật. Bạn nào có tài liệu hay về con trỏ cũng như quản lí bộ nhớ thì comment xuống dưới để mình liên kết tới bạn đọc.

Tutorial về con trỏ và ý nghĩa của con trỏ:
http://www.tutorialspoint.com/cprogramming/c_pointers.htm
(http://www.tutorialspoint.com/cprogramming/c_pointers.htm).
Malloc làm việc như thế nào:
http://www.inf.udec.cl/%7Eleo/Malloc_tutorial.pdf
(http://www.inf.udec.cl/%7Eleo/Malloc_tutorial.pdf)
Con trỏ hàm: http://www.learncpp.com/cpp-tutorial/78-function-pointers/
(http://www.learncpp.com/cpp-tutorial/78-function-pointers/)

Tham khảo

[1] T.H Cormen, C.E. Leiserson, R. Rivest, C. Stein. *Introduction to Algorithms (2nd ed.)*, Chapter 10 . MIT Press and McGraw-Hill (2001). ISBN 0-262-03293-7.
[2] D.E. Knuth, Donald E. *Dancing links (https://arxiv.org/pdf/cs/0011047v1.pdf)*. arXiv preprint cs/0011047 (2000).

--- Extendable Array, Stack and Queue (http://www.giaithua...p=1883)
February 15, 2017
In "armotize-analysis"

Theory (http://www.giaithua...p=553)
September 26, 2015
In "bfs"

Hashing and collision handling (http://www.giaithua...p=967)
March 31, 2016
In "ball-and-bin"

Tags: circularly-linked-list, data structure, doubly-linked-list, linked-list, memory-managment, pointers

No comments

Comments feed for this article

Trackback link: http://www.giaithuatlaptrinh.com/wp-trackback.php?p=1326

Reply

Your email address will not be published. Required fields are marked *

Your comment

Name *

Email *

Website

Post Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.