**Cơ bản về danh sách liên kết đơn (ôn lại để dễ hiểu)**

* **Danh sách liên kết đơn** giống như một chuỗi người nắm tay nhau. Mỗi người (nút, Node) có:
  + **Dữ liệu** (data): Một giá trị, ví dụ số 10.
  + **Con trỏ next**: Chỉ đến người tiếp theo (nút tiếp theo). Nút cuối có next = NULL.
* **Head**: Con trỏ trỏ đến nút đầu tiên của danh sách.
* **Ví dụ**: Danh sách [10 -> 20 -> 30] có:
  + Nút 1: data = 10, next trỏ đến nút chứa 20.
  + Nút 2: data = 20, next trỏ đến nút chứa 30.
  + Nút 3: data = 30, next = NULL.

**1. Hàm delete\_node**

c

CollapseWrap

Copy

void delete\_node(struct Node\*\* head\_ref, int data) {

struct Node\* current = \*head\_ref;

struct Node\* prev = NULL;

*// Trường hợp danh sách rỗng*

if (current == NULL) {

return;

}

*// Trường hợp nút đầu tiên chứa data*

if (current->data == data) {

\*head\_ref = current->next;

free(current);

return;

}

*// Tìm nút chứa data*

while (current != NULL && current->data != data) {

prev = current;

current = current->next;

}

*// Nếu không tìm thấy data*

if (current == NULL) {

return;

}

*// Xóa nút chứa data*

prev->next = current->next;

free(current);

}

**Mục đích của hàm**

Hàm này tìm và xóa **nút đầu tiên** trong danh sách liên kết có giá trị bằng data. Nếu không tìm thấy data hoặc danh sách rỗng, hàm không làm gì.

**Giải thích từng dòng và lý do**

1. **Dòng 1: struct Node\*\* head\_ref, int data**
   * **Giải thích**:
     + head\_ref là con trỏ đến con trỏ (Node\*\*), trỏ đến biến head (con trỏ đến nút đầu tiên của danh sách).
     + data là giá trị cần tìm và xóa.
   * **Tại sao**:
     + Dùng Node\*\* vì hàm cần thay đổi con trỏ head nếu nút đầu tiên bị xóa. Nếu chỉ dùng Node\* head, thay đổi head trong hàm sẽ không ảnh hưởng đến head ở ngoài hàm (do truyền tham trị, chỉ sao chép giá trị con trỏ).
     + Ví dụ: Nếu danh sách là [20 -> 10 -> 30] và xóa data = 20, head phải được cập nhật để trỏ đến nút 10. Điều này chỉ làm được với Node\*\*.
2. **Dòng 2: struct Node\* current = \*head\_ref;**
   * **Giải thích**: Tạo con trỏ current trỏ đến nút đầu tiên của danh sách (\*head\_ref là head).
   * **Tại sao**:
     + Cần một con trỏ để duyệt qua danh sách và kiểm tra từng nút. Bắt đầu từ nút đầu tiên vì ta cần tìm nút đầu tiên chứa data.
     + **Ví dụ**: Với danh sách [10 -> 20 -> 30], current trỏ đến nút 10.
3. **Dòng 3: struct Node\* prev = NULL;**
   * **Giải thích**: Tạo con trỏ prev để theo dõi nút **trước** nút current. Ban đầu, vì current ở nút đầu, không có nút trước nên prev = NULL.
   * **Tại sao**:
     + Khi xóa một nút ở giữa hoặc cuối danh sách, ta cần liên kết nút trước với nút sau. prev giúp lưu vị trí nút trước để thực hiện liên kết này.
     + **Ví dụ**: Nếu xóa nút 20 trong [10 -> 20 -> 30], cần prev trỏ đến nút 10 để gán 10->next trỏ đến nút 30.
4. **Dòng 5-7: Trường hợp danh sách rỗng**

c

CollapseWrap

Copy

if (current == NULL) {

return;

}

* + **Giải thích**: Kiểm tra xem danh sách có rỗng không (current == NULL nghĩa là head không trỏ đến nút nào). Nếu rỗng, thoát hàm.
  + **Tại sao**:
    - Nếu danh sách rỗng, không có nút nào để xóa, nên thoát ngay để tránh lỗi (như truy cập current->data khi current là NULL).
    - **Ví dụ**: Nếu head = NULL, hàm dừng ngay để đảm bảo an toàn.

1. **Dòng 9-12: Trường hợp nút đầu chứa data**

c

CollapseWrap

Copy

if (current->data == data) {

\*head\_ref = current->next;

free(current);

return;

}

* + **Giải thích**:
    - Kiểm tra xem nút đầu (current) có chứa data không.
    - Nếu đúng, cập nhật head để trỏ đến nút tiếp theo (current->next).
    - Giải phóng bộ nhớ của nút đầu (free(current)).
    - Thoát hàm vì đã xóa xong.
  + **Tại sao**:
    - Nút đầu là trường hợp đặc biệt vì không có nút trước (prev không cần dùng). Chỉ cần cập nhật head và giải phóng nút.
    - Giải phóng bộ nhớ (free) là cần thiết để tránh rò rỉ bộ nhớ (memory leak).
    - Thoát hàm ngay để tránh xử lý tiếp, vì nút cần xóa đã được xử lý.
    - **Ví dụ**: Với [20 -> 10 -> 30] và data = 20:
      * current->data == 20, gán \*head\_ref = current->next (tức head trỏ đến nút 10).
      * Giải phóng nút 20.
      * Kết quả: [10 -> 30].

1. **Dòng 14-17: Tìm nút chứa data**

c

CollapseWrap

Copy

while (current != NULL && current->data != data) {

prev = current;

current = current->next;

}

* + **Giải thích**:
    - Vòng lặp duyệt danh sách để tìm nút chứa data.
    - Mỗi lần lặp:
      * Lưu current vào prev (nút trước).
      * Di chuyển current sang nút tiếp theo.
    - Thoát khi tìm thấy data hoặc đến cuối danh sách (current == NULL).
  + **Tại sao**:
    - Cần duyệt danh sách để tìm nút chứa data. Kiểm tra current != NULL để tránh lỗi truy cập bộ nhớ không hợp lệ.
    - Lưu prev để biết nút trước nút cần xóa, giúp liên kết lại danh sách sau khi xóa.
    - **Ví dụ**: Với [10 -> 20 -> 30] và data = 20:
      * Lần 1: current tại 10, 10 != 20, gán prev = current (nút 10), current sang nút 20.
      * Thoát vì current->data == 20.

1. **Dòng 19-21: Nếu không tìm thấy data**

c

CollapseWrap

Copy

if (current == NULL) {

return;

}

* + **Giải thích**: Nếu vòng lặp kết thúc với current == NULL, nghĩa là không tìm thấy data. Thoát hàm.
  + **Tại sao**:
    - Nếu không tìm thấy data, yêu cầu là "không làm gì", nên thoát để giữ danh sách nguyên vẹn.
    - Kiểm tra này cần thiết để tránh lỗi khi cố gắng truy cập current->next ở bước sau.

1. **Dòng 23-24: Xóa nút chứa data**

c

CollapseWrap

Copy

prev->next = current->next;

free(current);

* + **Giải thích**:
    - Liên kết nút trước (prev) với nút sau (current->next), bỏ qua nút current.
    - Giải phóng bộ nhớ của nút current.
  + **Tại sao**:
    - Để xóa nút, ta phải "nối" nút trước với nút sau để duy trì cấu trúc danh sách.
    - Giải phóng current để tránh rò rỉ bộ nhớ.
    - **Ví dụ**: Với [10 -> 20 -> 30] và data = 20:
      * prev tại 10, current tại 20.
      * Gán prev->next = current->next (tức 10->next trỏ đến 30).
      * Giải phóng nút 20.
      * Kết quả: [10 -> 30].

**Hình ảnh minh họa**

Danh sách: [10 -> 20 -> 30], xóa data = 20:

* Trước: head -> [10 -> 20 -> 30]
* Sau khi tìm: prev -> [10], current -> [20 -> 30]
* Sau xóa: 10->next = 30, giải phóng nút 20
* Kết quả: head -> [10 -> 30]

**Tại sao cần xử lý các trường hợp riêng?**

* **Danh sách rỗng**: Tránh lỗi truy cập bộ nhớ.
* **Nút đầu**: Cần cập nhật head, không cần prev.
* **Nút giữa/cuối**: Cần prev để liên kết lại danh sách.
* **Không tìm thấy data**: Đáp ứng yêu cầu "không làm gì".

**2. Hàm search\_node**

c

CollapseWrap

Copy

struct Node\* search\_node(struct Node\* head, int data) {

struct Node\* current = head;

while (current != NULL) {

if (current->data == data) {

return current;

}

current = current->next;

}

return NULL;

}

**Mục đích của hàm**

Tìm **nút đầu tiên** chứa giá trị data và trả về con trỏ đến nút đó. Nếu không tìm thấy, trả về NULL.

**Giải thích từng dòng và lý do**

1. **Dòng 1: struct Node\* head, int data**
   * **Giải thích**:
     + head là con trỏ đến nút đầu tiên của danh sách.
     + data là giá trị cần tìm.
   * **Tại sao**:
     + Chỉ cần Node\* head (không phải Node\*\*) vì hàm chỉ đọc danh sách, không sửa đổi head.
     + **Ví dụ**: Với [10 -> 20 -> 30] và data = 20, hàm trả về con trỏ đến nút 20.
2. **Dòng 2: struct Node\* current = head;**
   * **Giải thích**: Tạo con trỏ current trỏ đến nút đầu tiên để bắt đầu duyệt.
   * **Tại sao**:
     + Cần một con trỏ để duyệt từng nút, bắt đầu từ đầu danh sách để kiểm tra từ nút đầu tiên.
     + **Ví dụ**: current trỏ đến nút 10.
3. **Dòng 3-7: Duyệt danh sách**

c

CollapseWrap

Copy

while (current != NULL) {

if (current->data == data) {

return current;

}

current = current->next;

}

* + **Giải thích**:
    - Vòng lặp kiểm tra từng nút:
      * Nếu current->data == data, trả về con trỏ current.
      * Nếu không, di chuyển current đến nút tiếp theo.
    - Thoát khi hết danh sách (current == NULL).
  + **Tại sao**:
    - Cần kiểm tra từng nút để tìm data. Kiểm tra current != NULL để tránh lỗi truy cập bộ nhớ.
    - Trả về ngay khi tìm thấy để đảm bảo trả về **nút đầu tiên** chứa data.
    - **Ví dụ**: Với [10 -> 20 -> 30] và data = 20:
      * Lần 1: current tại 10, 10 != 20, current sang nút 20.
      * Lần 2: 20 == 20, trả về con trỏ đến nút 20.

1. **Dòng 8: return NULL;**
   * **Giải thích**: Nếu duyệt hết danh sách mà không tìm thấy data, trả về NULL.
   * **Tại sao**:
     + Yêu cầu hàm trả về NULL nếu không tìm thấy data, giúp người gọi biết kết quả.
     + **Ví dụ**: Với [10 -> 20 -> 30] và data = 50, duyệt hết danh sách, trả về NULL.

**Hình ảnh minh họa**

Danh sách: [10 -> 20 -> 30], tìm data = 20:

* Bắt đầu: current -> [10]
* Lần 1: 10 != 20, current -> [20]
* Lần 2: 20 == 20, trả về con trỏ đến nút [20]

**Tại sao hàm đơn giản vậy?**

* Chỉ cần duyệt tuyến tính và trả về ngay khi tìm thấy, vì mục tiêu là tìm **nút đầu tiên**.
* Không cần Node\*\* vì hàm không thay đổi cấu trúc danh sách.

**3. Hàm main**

c

CollapseWrapRun

Copy

int main() {

struct Node\* head = NULL;

*// Tạo danh sách [10, 20, 30, 40]*

insert\_at\_head(&head, 40);

insert\_at\_head(&head, 30);

insert\_at\_head(&head, 20);

insert\_at\_head(&head, 10);

printf("Danh sach ban dau: ");

print\_list(head);

*// Tìm kiếm số 30*

struct Node\* found = search\_node(head, 30);

if (found != NULL) {

printf("Tim thay %d trong danh sach\n", found->data);

} else {

printf("Khong tim thay 30 trong danh sach\n");

}

*// Xóa số 20*

delete\_node(&head, 20);

printf("Danh sach sau khi xoa 20: ");

print\_list(head);

*// Giải phóng bộ nhớ*

struct Node\* current = head;

while (current != NULL) {

struct Node\* temp = current;

current = current->next;

free(temp);

}

return 0;

}

**Giải thích từng bước và lý do**

1. **Dòng 2: struct Node\* head = NULL;**
   * **Giải thích**: Khởi tạo con trỏ head trỏ đến NULL, nghĩa là danh sách rỗng.
   * **Tại sao**:
     + Ban đầu, danh sách chưa có nút nào, nên head phải là NULL để biểu thị danh sách rỗng.
     + Điều này cũng giúp các hàm như insert\_at\_head xử lý trường hợp danh sách rỗng.
2. **Dòng 4-7: Tạo danh sách [10, 20, 30, 40]**

c

CollapseWrap

Copy

insert\_at\_head(&head, 40);

insert\_at\_head(&head, 30);

insert\_at\_head(&head, 20);

insert\_at\_head(&head, 10);

* + **Giải thích**: Gọi insert\_at\_head 4 lần để tạo danh sách. Vì chèn vào đầu, thứ tự gọi ngược lại tạo ra [10 -> 20 -> 30 -> 40].
  + **Tại sao**:
    - insert\_at\_head được chọn vì nó đơn giản và nhanh (không cần duyệt danh sách).
    - Truyền &head (địa chỉ của head) để hàm có thể cập nhật head.
    - **Ví dụ**:
      * Sau insert\_at\_head(&head, 40): [40]
      * Sau insert\_at\_head(&head, 30): [30 -> 40]
      * Sau insert\_at\_head(&head, 20): [20 -> 30 -> 40]
      * Sau insert\_at\_head(&head, 10): [10 -> 20 -> 30 -> 40]

1. **Dòng 9-10: In danh sách ban đầu**

c

CollapseWrap

Copy

printf("Danh sach ban dau: ");

print\_list(head);

* + **Giải thích**: In danh sách để kiểm tra: 10 20 30 40.
  + **Tại sao**:
    - Cần hiển thị danh sách để xác nhận nó được tạo đúng trước khi thực hiện tìm kiếm hoặc xóa.
    - Hàm print\_list duyệt và in từng giá trị, giúp người dùng thấy cấu trúc danh sách.

1. **Dòng 12-17: Tìm kiếm số 30**

c

CollapseWrap

Copy

struct Node\* found = search\_node(head, 30);

if (found != NULL) {

printf("Tim thay %d trong danh sach\n", found->data);

} else {

printf("Khong tim thay 30 trong danh sach\n");

}

* + **Giải thích**:
    - Gọi search\_node(head, 30) để tìm nút chứa 30.
    - Nếu found != NULL, in giá trị found->data (30). Nếu không, in thông báo không tìm thấy.
  + **Tại sao**:
    - Yêu cầu kiểm tra xem số 30 có trong danh sách không. Hàm search\_node trả về con trỏ hoặc NULL, giúp xác định kết quả.
    - In found->data để hiển thị giá trị tìm thấy, giúp người dùng hiểu rõ kết quả.
    - **Ví dụ**: search\_node tìm thấy nút 30, in Tim thay 30 trong danh sach.

1. **Dòng 19-20: Xóa số 20**

c

CollapseWrap

Copy

delete\_node(&head, 20);

printf("Danh sach sau khi xoa 20: ");

print\_list(head);

* + **Giải thích**:
    - Gọi delete\_node(&head, 20) để xóa nút chứa 20.
    - In danh sách sau khi xóa: 10 30 40.
  + **Tại sao**:
    - Yêu cầu xóa số 20 và hiển thị danh sách sau khi xóa để xác nhận kết quả.
    - Truyền &head để hàm có thể cập nhật head nếu cần (dù trong trường hợp này head không thay đổi vì 20 không phải nút đầu).

1. **Dòng 22-27: Giải phóng bộ nhớ**

c

CollapseWrap

Copy

struct Node\* current = head;

while (current != NULL) {

struct Node\* temp = current;

current = current->next;

free(temp);

}

* + **Giải thích**:
    - Duyệt danh sách, lưu current vào temp, di chuyển current đến nút tiếp theo, giải phóng temp.
    - Lặp lại đến khi current == NULL.
  + **Tại sao**:
    - Trong C, bộ nhớ được cấp phát động (bằng malloc trong create\_node) phải được giải phóng bằng free để tránh rò rỉ bộ nhớ.
    - Dùng temp để lưu current trước khi di chuyển, vì sau khi giải phóng current, truy cập current->next sẽ gây lỗi.
    - **Ví dụ**: Với [10 -> 30 -> 40]:
      * Lần 1: temp = 10, current = 30, giải phóng nút 10.
      * Lần 2: temp = 30, current = 40, giải phóng nút 30.
      * Lần 3: temp = 40, current = NULL, giải phóng nút 40.

**Kết quả chạy**

text

CollapseWrap

Copy

Danh sach ban dau: 10 20 30 40

Tim thay 30 trong danh sach

Danh sach sau khi xoa 20: 10 30 40

**Lời khuyên cho người mới**

* **Hiểu con trỏ**: Con trỏ giống như mũi tên chỉ đến một địa chỉ bộ nhớ. Node\*\* là mũi tên chỉ đến mũi tên khác, cho phép thay đổi head.
* **Vẽ sơ đồ**: Vẽ danh sách trên giấy, đánh dấu head, current, prev, next để hình dung sự thay đổi.
* **Kiểm tra trường hợp đặc biệt**:
  + Danh sách rỗng.
  + Xóa/tìm nút đầu, giữa, cuối.
  + Giá trị không tồn tại.
* **Tại sao cần giải phóng bộ nhớ?**: Trong C, nếu không dùng free, bộ nhớ cấp phát sẽ không được thu hồi, gây lãng phí tài nguyên.

**Tóm tắt với lý do**

* **delete\_node**:
  + Tìm và xóa nút đầu tiên chứa data, xử lý các trường hợp đặc biệt (rỗng, nút đầu, nút giữa/cuối) để đảm bảo an toàn và đúng yêu cầu.
  + Dùng Node\*\* để cập nhật head, dùng prev để liên kết lại danh sách.
* **search\_node**:
  + Duyệt tuyến tính, trả về con trỏ đến nút đầu tiên chứa data hoặc NULL, đơn giản vì không cần sửa danh sách.
  + Dùng Node\* vì chỉ đọc danh sách.
* **main**:
  + Tạo danh sách, tìm kiếm, xóa, và in kết quả đúng theo yêu cầu.
  + Giải phóng bộ nhớ để đảm bảo chương trình sạch.