**Cấu trúc Node**

typedef struct Node {

int data;

struct Node\* next;

} Node;

* **Giải thích**:
  + data: Lưu giá trị số nguyên của nút (ví dụ: 10, 20, 30).
  + next: Con trỏ trỏ tới nút tiếp theo trong danh sách, hoặc NULL nếu là nút cuối.
  + typedef: Đặt tên alias là Node để viết code ngắn gọn, không cần viết struct Node.
* **Tại sao**:
  + Danh sách liên kết đơn cần mỗi nút lưu trữ một giá trị và một liên kết tới nút tiếp theo để tạo thành chuỗi.
  + next cho phép duyệt danh sách để tìm kiếm hoặc xóa nút.
* **Ví dụ**:
  + Với danh sách [10 -> 20 -> 30]:
    - Nút đầu: data = 10, next trỏ tới nút có data = 20.
    - Nút cuối: data = 30, next = NULL.

**2. Hàm create\_node**

Node\* create\_node(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: Node\* new\_node = (Node\*)malloc(sizeof(Node));
     + Cấp phát bộ nhớ động cho một nút mới bằng malloc.
     + sizeof(Node): Lấy kích thước của cấu trúc Node.
     + (Node\*): Ép kiểu con trỏ từ malloc thành Node\*.
  2. **Dòng 2**: new\_node->data = data;
     + Gán giá trị data (tham số đầu vào) vào trường data của nút mới.
  3. **Dòng 3**: new\_node->next = NULL;
     + Đặt con trỏ next về NULL, vì nút mới chưa liên kết với nút nào.
  4. **Dòng 4**: return new\_node;
     + Trả về con trỏ tới nút mới vừa tạo.
* **Tại sao**:
  1. Hàm này tạo nút mới an toàn, đảm bảo data được gán và next là NULL.
  2. Sử dụng malloc để cấp phát bộ nhớ động trên heap, vì danh sách liên kết cần lưu trữ nút ngoài phạm vi hàm.
  3. Đặt next = NULL để tránh lỗi truy cập bộ nhớ ngẫu nhiên.
* **Ví dụ**:
  1. Gọi create\_node(10): Tạo nút với data = 10, next = NULL.

**3. Hàm insert\_at\_tail**

void insert\_at\_tail(Node\*\* head\_ref, int data) {

Node\* new\_node = create\_node(data);

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

Node\* current = \*head\_ref;

while (current->next != NULL) {

current = current->next;

}

current->next = new\_node;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void insert\_at\_tail(Node\*\* head\_ref, int data)
     + Nhận head\_ref (con trỏ đến con trỏ Node\*) để thay đổi head nếu danh sách rỗng.
     + data: Giá trị cần chèn.
  2. **Dòng 2**: Node\* new\_node = create\_node(data);
     + Tạo nút mới với giá trị data.
  3. **Dòng 3-5**: if (\*head\_ref == NULL) { \*head\_ref = new\_node; return; }
     + Nếu danh sách rỗng, đặt head trỏ tới nút mới và thoát.
  4. **Dòng 6**: Node\* current = \*head\_ref;
     + Tạo con trỏ current để duyệt tới nút cuối.
  5. **Dòng 7-8**: while (current->next != NULL) { current = current->next; }
     + Duyệt tới nút cuối (nút có next == NULL).
  6. **Dòng 9**: current->next = new\_node;
     + Liên kết nút cuối với nút mới.
* **Tại sao**:
  1. Chèn vào cuối giữ thứ tự nhập của người dùng (ví dụ: nhập 10, 20, 30 tạo [10 -> 20 -> 30]).
  2. Dùng Node\*\* để cập nhật head trong trường hợp danh sách rỗng.
  3. Phương pháp này phù hợp với yêu cầu nhập danh sách bất kỳ.
* **Ví dụ**:
  1. Ban đầu: head = NULL.
  2. Gọi insert\_at\_tail(&head, 10): [10].
  3. Gọi insert\_at\_tail(&head, 20): [10 -> 20].

**4. Hàm print\_list**

void print\_list(Node\* head) {

if (head == NULL) {

printf("Danh sach rong\n");

return;

}

Node\* current = head;

while (current != NULL) {

printf("%d", current->data);

if (current->next != NULL) {

printf(" -> ");

}

current = current->next;

}

printf("\n");

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void print\_list(Node\* head)
     + Nhận head trỏ tới nút đầu của danh sách.
  2. **Dòng 2-4**: if (head == NULL) { printf("Danh sach rong\n"); return; }
     + Kiểm tra nếu danh sách rỗng, in thông báo và thoát.
  3. **Dòng 5**: Node\* current = head;
     + Tạo con trỏ current để duyệt danh sách từ nút đầu.
  4. **Dòng 6-10**: while (current != NULL)
     + In giá trị data của nút hiện tại.
     + Nếu có nút tiếp theo, in " -> ".
     + Di chuyển current tới nút tiếp theo.
  5. **Dòng 11**: printf("\n");
     + In ký tự xuống dòng để định dạng.
* **Tại sao**:
  1. Hàm này in danh sách để người dùng thấy cấu trúc trước và sau khi xóa.
  2. Kiểm tra head == NULL để xử lý danh sách rỗng.
  3. Dấu " -> " giúp hình dung liên kết giữa các nút.
* **Ví dụ**:
  1. Với danh sách [10 -> 20 -> 30]: In 10 -> 20 -> 30.
  2. Với danh sách rỗng: In Danh sach rong.

**5. Hàm search\_node**

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

Node\* current = head;

while (current != NULL) {

if (current->data == data) {

return current;

}

current = current->next;

}

return NULL;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: Node\* search\_node(Node\* head, int data)
     + Nhận head trỏ tới nút đầu và data là giá trị cần tìm.
     + Trả về con trỏ Node\* tới nút đầu tiên chứa data, hoặc NULL nếu không tìm thấy.
  2. **Dòng 2**: Node\* current = head;
     + Khởi tạo current trỏ tới nút đầu để bắt đầu duyệt.
  3. **Dòng 3-6**: while (current != NULL)
     + Nếu current->data == data, trả về current (nút được tìm thấy).
     + Di chuyển current tới nút tiếp theo.
  4. **Dòng 7**: return NULL;
     + Nếu duyệt hết danh sách mà không tìm thấy, trả về NULL.
* **Tại sao**:
  1. Hàm này duyệt tuyến tính để tìm nút đầu tiên có giá trị data, phù hợp với danh sách liên kết đơn.
  2. Trả về con trỏ tới nút để cung cấp thông tin (ví dụ: địa chỉ nút) thay vì chỉ báo có tìm thấy hay không.
  3. Độ phức tạp O(n) thời gian, O(1) không gian.
* **Ví dụ**:
  1. Với danh sách [10 -> 20 -> 30], tìm data = 20:
     + current duyệt qua 10, 20, thấy current->data == 20, trả về con trỏ tới nút 20.
  2. Tìm data = 50: Duyệt hết danh sách, trả về NULL.

**6. Hàm delete\_node**

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

if (\*head\_ref == NULL) {

return;

}

Node\* current = \*head\_ref;

Node\* prev = NULL;

*// Trường hợp nút đầu cần xóa*

if (current->data == data) {

\*head\_ref = current->next;

free(current);

return;

}

*// Tìm nút cần xóa*

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

prev = current;

current = current->next;

}

*// Nếu không tìm thấy*

if (current == NULL) {

return;

}

*// Xóa nút*

prev->next = current->next;

free(current);

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void delete\_node(Node\*\* head\_ref, int data)
     + Nhận head\_ref để thay đổi head nếu cần xóa nút đầu.
     + data: Giá trị của nút cần xóa.
  2. **Dòng 2-3**: if (\*head\_ref == NULL) { return; }
     + Nếu danh sách rỗng, không làm gì và thoát.
  3. **Dòng 4-5**: Node\* current = \*head\_ref; Node\* prev = NULL;
     + current: Duyệt danh sách để tìm nút cần xóa.
     + prev: Theo dõi nút trước current để cập nhật liên kết khi xóa.
  4. **Dòng 6-9**: Trường hợp nút đầu cần xóa
     + Nếu current->data == data, cập nhật head để trỏ tới nút tiếp theo (current->next).
     + Giải phóng nút đầu (current) và thoát.
  5. **Dòng 10-13**: Tìm nút cần xóa
     + Duyệt current tới khi tìm thấy nút có data hoặc hết danh sách.
     + Cập nhật prev để trỏ tới current trước khi di chuyển current.
  6. **Dòng 14-15**: if (current == NULL) { return; }
     + Nếu không tìm thấy nút, thoát mà không làm gì.
  7. **Dòng 16-17**: Xóa nút
     + Cập nhật prev->next để trỏ tới current->next, bỏ qua nút current.
     + Giải phóng nút current.
* **Tại sao**:
  1. Dùng Node\*\* head\_ref để thay đổi head khi xóa nút đầu.
  2. Xử lý riêng trường hợp nút đầu để đơn giản hóa logic.
  3. Dùng prev để duy trì liên kết khi xóa nút ở giữa hoặc cuối.
  4. Độ phức tạp O(n) thời gian, O(1) không gian.
* **Ví dụ**:
  1. Với danh sách [10 -> 20 -> 30], xóa data = 20:
     + current duyệt qua 10, 20.
     + prev -> 10, current -> 20.
     + prev->next = current->next (10 trỏ tới 30).
     + Giải phóng nút 20.
     + Kết quả: [10 -> 30].
  2. Xóa data = 50: Không tìm thấy, không làm gì.

**7. Hàm free\_list**

void free\_list(Node\*\* head\_ref) {

Node\* current = \*head\_ref;

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp);

}

\*head\_ref = NULL;

}

* **Giải thích từng dòng**:
  1. **Dòng 1**: void free\_list(Node\*\* head\_ref)
     + Nhận head\_ref để thay đổi head trong hàm gọi.
  2. **Dòng 2**: Node\* current = \*head\_ref;
     + Khởi tạo current trỏ tới nút đầu.
  3. **Dòng 3-6**: while (current != NULL)
     + Lưu current vào temp để giải phóng.
     + Di chuyển current tới nút tiếp theo.
     + Giải phóng temp.
  4. **Dòng 7**: \*head\_ref = NULL;
     + Đặt head về NULL để đánh dấu danh sách rỗng.
* **Tại sao**:
  1. Giải phóng bộ nhớ của mỗi nút để tránh rò rỉ bộ nhớ.
  2. Dùng temp để tránh truy cập bộ nhớ đã giải phóng.
  3. Dùng Node\*\* để đặt head = NULL trong hàm gọi.
* **Ví dụ**:
  1. Với danh sách [10 -> 30]:
     + Lần 1: Giải phóng nút 10, current = 30.
     + Lần 2: Giải phóng nút 30, current = NULL.
     + \*head\_ref = NULL.

**8. Hàm main**

int main() {

Node\* head = NULL;

int n, value, search\_value, delete\_value;

*// Nhập số lượng phần tử*

printf("Nhap so luong phan tu cua danh sach: ");

scanf("%d", &n);

if (n < 0) {

printf("So luong phan tu khong hop le!\n");

return 1;

}

*// Nhập các phần tử*

printf("Nhap %d gia tri cho danh sach:\n", n);

for (int i = 0; i < n; i++) {

scanf("%d", &value);

insert\_at\_tail(&head, value);

}

*// In danh sách ban đầu*

printf("Danh sach vua nhap: ");

print\_list(head);

*// Tìm kiếm giá trị*

printf("Nhap gia tri can tim kiem: ");

scanf("%d", &search\_value);

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

if (found\_node != NULL) {

printf("Tim kiem gia tri %d: Tim thay tai dia chi %p\n", search\_value, found\_node);

} else {

printf("Tim kiem gia tri %d: Khong tim thay\n", search\_value);

}

*// Xóa giá trị*

printf("Nhap gia tri can xoa: ");

scanf("%d", &delete\_value);

delete\_node(&head, delete\_value);

printf("Danh sach sau khi xoa %d: ", delete\_value);

print\_list(head);

*// Giải phóng danh sách*

free\_list(&head);

printf("Danh sach sau khi giai phong: ");

print\_list(head);

return 0;

}

* **Giải thích từng phần**:
  1. **Dòng 2-3**: Khởi tạo biến
     + Node\* head = NULL: Khởi tạo danh sách rỗng.
     + int n, value, search\_value, delete\_value: Biến để lưu số lượng phần tử, giá trị nhập, giá trị tìm kiếm, và giá trị xóa.
     + **Tại sao**: Cần khởi tạo head = NULL để bắt đầu với danh sách rỗng. Các biến hỗ trợ nhập liệu và thao tác.
  2. **Dòng 5-10**: Nhập số lượng phần tử
     + Yêu cầu nhập n, kiểm tra n < 0 để đảm bảo hợp lệ.
     + **Tại sao**: Cho phép người dùng xác định kích thước danh sách, kiểm tra lỗi để tránh input không hợp lệ.
  3. **Dòng 12-16**: Nhập các phần tử
     + Yêu cầu nhập n giá trị, dùng insert\_at\_tail để thêm vào cuối danh sách.
     + **Tại sao**: Giữ thứ tự nhập của người dùng (ví dụ: nhập 10, 20, 30 tạo [10 -> 20 -> 30]).
  4. **Dòng 18-19**: In danh sách ban đầu
     + In danh sách vừa nhập.
     + **Tại sao**: Xác nhận danh sách được tạo đúng.
  5. **Dòng 21-27**: Tìm kiếm giá trị
     + Yêu cầu nhập search\_value, gọi search\_node.
     + In địa chỉ nút nếu tìm thấy, hoặc thông báo không tìm thấy.
     + **Tại sao**: Đáp ứng yêu cầu tìm kiếm, hiển thị kết quả rõ ràng.
  6. **Dòng 29-32**: Xóa giá trị
     + Yêu cầu nhập delete\_value, gọi delete\_node.
     + In danh sách sau khi xóa.
     + **Tại sao**: Đáp ứng yêu cầu xóa nút, kiểm tra kết quả sau khi xóa.
  7. **Dòng 34-35**: Giải phóng danh sách
     + Gọi free\_list, in danh sách rỗng để xác nhận.
     + **Tại sao**: Tránh rò rỉ bộ nhớ, xác nhận giải phóng thành công.
  8. **Dòng 37**: return 0;
     + Kết thúc chương trình thành công.
* **Tại sao cấu trúc này**:
  1. Đáp ứng yêu cầu nhập danh sách bất kỳ thay vì cố định [10, 20, 30, 40].
  2. Giao diện thân thiện, xử lý lỗi input cơ bản.
  3. Thực hiện tìm kiếm và xóa theo yêu cầu, hiển thị kết quả rõ ràng.

**Kết quả chạy**

**Trường hợp 1: Nhập [10, 20, 30, 40], tìm 30, xóa 20**

Nhap so luong phan tu cua danh sach: 4

Nhap 4 gia tri cho danh sach:

10 20 30 40

Danh sach vua nhap: 10 -> 20 -> 30 -> 40

Nhap gia tri can tim kiem: 30

Tim kiem gia tri 30: Tim thay tai dia chi 0x...

Nhap gia tri can xoa: 20

Danh sach sau khi xoa 20: 10 -> 30 -> 40

Danh sach sau khi giai phong: Danh sach rong

**Trường hợp 2: Nhập danh sách rỗng**

Nhap so luong phan tu cua danh sach: 0

Nhap 0 gia tri cho danh sach:

Danh sach vua nhap: Danh sach rong

Nhap gia tri can tim kiem: 10

Tim kiem gia tri 10: Khong tim thay

Nhap gia tri can xoa: 10

Danh sach sau khi xoa 10: Danh sach rong

Danh sach sau khi giai phong: Danh sach rong

**Trường hợp 3: Tìm/xóa giá trị không tồn tại**

Nhap so luong phan tu cua danh sach: 3

Nhap 3 gia tri cho danh sach:

10 20 30

Danh sach vua nhap: 10 -> 20 -> 30

Nhap gia tri can tim kiem: 50

Tim kiem gia tri 50: Khong tim thay

Nhap gia tri can xoa: 50

Danh sach sau khi xoa 50: 10 -> 20 -> 30

Danh sach sau khi giai phong: Danh sach rong

**Hình ảnh minh họa**

**Danh sách ban đầu (nhập 10, 20, 30, 40):**

head -> [10] -> [20] -> [30] -> [40] -> NULL

**Tìm kiếm giá trị 30:**

* search\_node(head, 30):
  + current duyệt qua 10, 20, 30, thấy data == 30.
  + Trả về con trỏ tới nút 30.

**Xóa giá trị 20:**

* delete\_node(&head, 20):
  + prev -> 10, current -> 20.
  + prev->next = current->next (10 trỏ tới 30).
  + Giải phóng nút 20.
* Danh sách sau xóa:

text

CollapseWrap

Copy

head -> [10] -> [30] -> [40] -> NULL

**Sau khi giải phóng:**

head -> NULL

**Lời khuyên cho người mới**

1. **Hiểu con trỏ**:
   * Node\*: Con trỏ tới một nút, chứa địa chỉ của Node.
   * Node\*\*: Con trỏ tới con trỏ Node\*, dùng để thay đổi head trong hàm gọi (ví dụ: delete\_node, insert\_at\_tail).
   * **Ví dụ**: head\_ref chứa địa chỉ của head, \*head\_ref là head.
2. **Hiểu tìm kiếm**:
   * search\_node duyệt tuyến tính từ đầu danh sách, so sánh data của từng nút.
   * Vẽ danh sách trên giấy, theo dõi current để thấy cách tìm kiếm.
3. **Hiểu xóa nút**:
   * Xóa nút đầu cần cập nhật head, dùng Node\*\*.
   * Xóa nút giữa/cuối cần prev để cập nhật liên kết (prev->next).
   * Vẽ danh sách và thử xóa nút đầu, giữa, cuối để hiểu rõ.
4. **Debug**:
   * Nếu gặp lỗi, thêm printf("%p, data: %d\n", node, node->data) để in địa chỉ và giá trị của nút.
   * Trong delete\_node, in prev và current để theo dõi quá trình duyệt.
5. **Tại sao giải phóng bộ nhớ?**:
   * Mỗi nút được tạo bằng malloc chiếm bộ nhớ heap. Không giải phóng sẽ gây rò rỉ bộ nhớ.
   * Đặt head = NULL sau khi giải phóng tránh lỗi truy cập bộ nhớ đã giải phóng.
6. **Tại sao dùng insert\_at\_tail?**:
   * Giữ thứ tự nhập của người dùng, phù hợp với yêu cầu nhập danh sách bất kỳ.
   * So với insert\_at\_head, insert\_at\_tail không đảo ngược thứ tự input.