**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI**

**PHÂN HIÊU TẠI TP.HỒ CHÍ MINH**

**BỘ MÔN CÔNG NGHỆ THÔNG TIN**

****

**BÁO CÁO BÀI TẬP LỚN**

**ĐỀ TÀI:QUẢN LÍ SÂN BÓNG**

|  |  |  |
| --- | --- | --- |
| Giảng viên hướng dẫn | **:** | TRẦN THỊ DUNG |
| Sinh viên thực hiện | **:** | LÊ NGUYỄN MINH HOÀNG |
|  | **:** | CAO VĂN NHỊP |
|  | **:** | NGUYỄN CÔNG PHƯƠNG |
| Lớp | **:** | Kĩ Thuật ĐTVT |
| Khóa | **:** | K64 |

Tp.Hồ Chí Minh, năm 2024

**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI**

**PHÂN HIÊU TẠI TP.HỒ CHÍ MINH**

**BỘ MÔN CÔNG NGHỆ THÔNG TIN**

****

**BÁO CÁO BÀI TẬP LỚN**

**ĐỀ TÀI:QUẢN LÍ SÂN BÓNG**

|  |  |  |
| --- | --- | --- |
| Giảng viên hướng dẫn | : | TRẦN THỊ DUNG |
| Sinh viên thực hiện | : | LÊ NGUYỄN MINH HOÀNG |
|  | : | CAO VĂN NHỊP |
|  | : | NGUYỄN CÔNG PHƯƠNG |
| Lớp | : | Kĩ Thuật ĐTVT |
| Khóa | : | K64 |

Tp.Hồ Chí Minh, năm 2024

**LỜI CẢM ƠN**

Trong quá trình thực hiện tiểu luận này, em xin chân thành cảm ơn cô giáo TRẦN THỊ DUNG đã tận tình hướng dẫn. Những kiến thức sâu sắc mà cô chia sẻ đã giúp em rất nhiều trong việc hoàn thiện bài viết này. Sự nhiệt huyết của cô với môn học đã cho em biết them những kiến thức,

Em cũng xin cảm ơn cô vì đã luôn tạo ra một môi trường học tập thân thiện, khuyến khích chúng học tập. Những buổi thảo luận và những lời khuyên chân thành của cô đã giúp em nhìn nhận vấn đề một cách sâu sắc hơn.

Cuối cùng, em xin gửi lời cảm ơn đến cô vì sự hỗ trợ không ngừng trong suốt quá trình học tập.

Xin chân thành cảm ơn cô!

**MỤC LỤC**

[**CHƯƠNG I: GIỚI THIỆU Ý TƯỞNG VÀ CƠ SỞ LÝ THUYẾT. 7**](#_Toc182146017)

[**1.1 Ý tưởng thiết kế . 7**](#_Toc182146018)

[**1.2 Cơ sở lý thuyết và cấu trúc danh sách liên kết đơn. 7**](#_Toc182146019)

[**1.2.1 khái niệm. 7**](#_Toc182146020)

[**1.2.2 Tính chất: 7**](#_Toc182146021)

[**1.2.3 Cấu trúc Node trong DSLK đơn. 8**](#_Toc182146022)

[**1.2.4 Thêm phần tử vào đầu danh sách ( addFirst ). 8**](#_Toc182146023)

[**1.2.5 Thêm phần tử vào cuối danh sách ( addLast ). 8**](#_Toc182146024)

[**1.2.6 Thêm vào sau Node bất kì ( InsertAfter ). 9**](#_Toc182146025)

[**1.2.7 Xoá phần tử khỏi danh sách (removeFirst ) . 9**](#_Toc182146026)

[**1.2.8 Xoá phần tử cuối khỏi danh sách ( removeLast ). 9**](#_Toc182146027)

[**1.2.9 Xoá ở sau Node bất kì . 10**](#_Toc182146028)

[**1.3 Cơ sở lý thuyết về sắp xếp. 11**](#_Toc182146031)

[**1.3.1 Khái niệm. 11**](#_Toc182146032)

[**1.3.2 Sắp xếp nổi bọt ( Bubble sort ). 13**](#_Toc182146039)

[**1.4 Cơ sở lý thuyết về tìm kiếm. 15**](#_Toc182146047)

[**1.4.1 khái niệm. 15**](#_Toc182146048)

[**1.4.2 Tìm kiếm tuần tự ( Linear Search ). 16**](#_Toc182146052)

[**Chương II: XÂY DỰNG CHƯƠNG TRÌNH. 17**](#_Toc182146056)

[**2.1 Giới thiệu thuật toán. 17**](#_Toc182146057)

[**2.1.1 Các thư viện trong chương trình. 17**](#_Toc182146058)

[**2.2 Xây dựng chương trình. 18**](#_Toc182146061)

[**2.2.1 Khai báo cấu trúc Node. 18**](#_Toc182146062)

[**2.2.2 Các hàm khai báo của cấu trúc LIST. 18**](#_Toc182146063)

[**2.2.3 Hàm khởi tạo . 18**](#_Toc182146064)

[**2.2.4 Hàm quản lý đặt sân . 20**](#_Toc182146065)

[**2.2.5 Hàm khởi tạo danh sách sân bóng. 23**](#_Toc182146066)

[**2.2.6 Hàm thêm sân bóng vào đầu danh sách. 24**](#_Toc182146067)

[**2.2.7 Hàm thêm sân bóng vào cuối danh sách . 24**](#_Toc182146068)

[**2.2.8 Hàm thêm sân bóng bất kì. 25**](#_Toc182146069)

[**2.2.9 Hàm xóa sân bóng ở đầu danh sách. 26**](#_Toc182146070)

[**2.2.10 Hàm xóa sân bóng ở cuối danh sách. 27**](#_Toc182146071)

[**2.2.11 Hàm xóa một sân bóng tại một vị trí cụ thể. 27**](#_Toc182146072)

[**2.2.12 Hàm kiểm tra sân bóng có tồn tại hay không. 28**](#_Toc182146073)

[**2.2.13 Hàm khởi tạo thông tin của khác hàng. 30**](#_Toc182146074)

[**2.2.14 Hàm xử lý menu và tương tác với người dung. 32**](#_Toc182146075)

[**CHƯƠNG III: KẾT QUẢ CHƯƠNG TRÌNH 33**](#_Toc182146076)

**PHỤ LỤC:................................................................................................................................................35**

**TÀI LIỆU THAM KHẢO:........................................................................................................................36**

|  |  |  |
| --- | --- | --- |
| **STT** | **Mô tả** | **Ý nghĩa** |
| **1** | **ĐTVT** | **Điện tử viễn thông** |
| **2** | **K** | **Khóa** |
| **3** | **DSLK** | **Danh sách liên kết** |
| **4** | **SDT** | **Số điện thoại** |
|  |  |  |
|  |  |  |
|  |  |  |

**DANH MỤC CHỮ VIẾT TẮT**

**BẢNG BIỂU, SƠ ĐỒ, HÌNH VẼ.**

Hình 1.1: Cấu trúc danh sách liên kết đơn.

Hình 1.2 Thuật toán sắp xếp nổi bọt -Bubble Sort.

Hình 1.3 Hàm sắp xếp nổi bọt Bubble Sort.

Hình 1.4 Hàm tìm kiếm tuần tự .

Hình 1.5 Thư viện trong chương trình.

Hình 1.6 Hàm khai báo cấu trúc Node.

Hình 1.7 Hàm khai báo của cấu trúc List.

Hình 1.8 Hàm khởi tạo sân bóng và danh sách sân bóng.

Hình 1.9 Hàm khởi tạo ngăn xếp đặt sân.

Hình 1.10 Hàm khởi tạo ngăn xếp stack.

Hình 1.11 Hàm khởi tạo khách hàng và danh sách khách hàng.

Hình 1.12 Hàm tính tiền thuê sân.

Hình 1.13 Hàm in thông tin lịch đặt sân.

Hình 1.14 Hàm xem lịch đặt sân .

Hình 1.15 Hàm tính chất cây heap.

Hình 1.16 Hàm sắp xếp mảng DatSan theo giờ đặt sân.

Hình 1.17 Hàm in ra tất cả các lịch đặt theo giờ bắt đầu tăng dần.

Hình 1.18 Hàm khởi tạo danh sách sân bóng.

Hình 1.19 hàm thêm sân bóng vào đầu danh sách .

Hình 1.20 Hàm thêm sân bóng vào cuối danh sách.

Hình 1.21 Hàm thêm sân bóng bất kì.

Hình 1.22 Hàm xóa sân bóng ở đầu danh sách.

Hình 1.23 Hàm xóa sân bóng ở cuối danh sách.

Hình 1.24 Hàm xóa một sân bóng tại một vị trí cụ thể.

Hình 1.25 Hàm tìm kiếm sân bóng trong danh sách.

Hình 1.26 Hàm sắp xếp danh sách sân bóng theo mã sân.

Hình 1.27 Hàm liệt kê danh sách sân bóng.

Hình 1.28 Hàm Khởi tạo danh sách khách hàng và thêm khách hàng vào danh sách.

Hình 1.29 Hàm tìm kiếm khách hàng theo số điện thoại và Cập nhật lịch sử đặt sân.

Hình 1.30 Hàm cập nhật thông tin khách hàng.

Hình 1.31 Hàm hiển thị thông tin đặt sân của khách hàng.

Hình 1.32 Hàm xử lí menu chính.

# CHƯƠNG I: GIỚI THIỆU Ý TƯỞNG VÀ CƠ SỞ LÝ THUYẾT.

**1.1 Ý tưởng thiết kế** .

1. Mô tả: Phát triển một công cụ để giúp lập trình viên đánh giá hiệu suất của các đoạn mã, phát hiện các điểm nghẽn và tối ưu hóa mã nguồn.

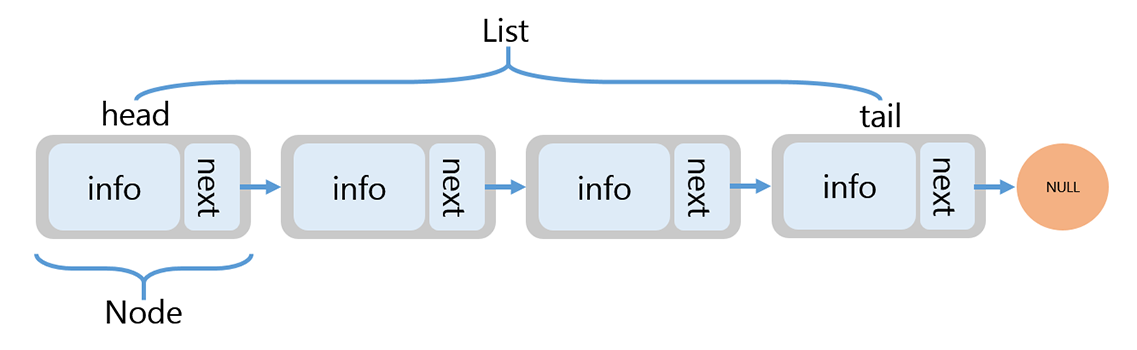
2. Tính năng chính:

* Đo lường thời gian thực thi của các đoạn mã trong môi trường thực tế.
* Phân tích độ phức tạp về mặt thời gian và không gian của mã nguồn (Big-O analysis).
* Cung cấp các gợi ý tối ưu hóa mã như việc sử dụng bộ nhớ hiệu quả hơn hoặc giảm thiểu số lần truy vấn cơ sở dữ liệu.
* Hỗ trợ nhiều ngôn ngữ lập trình như Python, JavaScript, Java, C++,…

**1.2 Cơ sở lý thuyết và cấu trúc danh sách liên kết đơn.**

**1.2.1 khái niệm.**

Danh sách liên kết đơn là một cấu trúc dữ liệu hữu ích, đặc biệt trong các tình huống cần thao tác linh hoạt với dữ liệu như thêm hoặc xóa phần tử mà không cần phải thay đổi kích thước bộ nhớ lớn như trong mảng. Tuy nhiên, nó yêu cầu phải duyệt qua các phần tử để truy cập một phần tử cụ thể, điều này làm giảm hiệu quả khi cần truy xuất dữ liệu nhanh chóng.



Hình 1.1 cấu trúc danh sách liên kết đơn

### 1.2.2 Tính chất:

1.Tuyến tính: Các phần tử được tổ chức theo trình tự, mỗi phần tử chỉ trỏ đến phần tử kế tiếp.

2. Dễ thay đổi kích thước: Kích thước linh hoạt, không cần cấp phát bộ nhớ cố định như mảng.

3. Không gian bộ nhớ: Tiết kiệm bộ nhớ, nhưng mỗi phần tử phải lưu trữ thêm một con trỏ.

4. Truy cập tuần tự: Không thể truy cập trực tiếp, phải duyệt qua các phần tử theo thứ tự.

5. Dễ dàng thêm/xóa: Thêm và xóa phần tử nhanh chóng ở đầu, nhưng xóa cuối có thể chậm.

6. Không hỗ trợ truy cập ngẫu nhiên: Không thể truy xuất phần tử theo chỉ số trực tiếp.

7. Chậm trong tìm kiếm**:** Tìm kiếm phải duyệt qua tất cả các phần tử, độ phức tạp O(n).

8. Quản lý bộ nhớ phân mảnh: Các phần tử không liên tiếp trong bộ nhớ.

### 1.2.3 Cấu trúc Node trong DSLK đơn.

Trong danh sách liên kết đơn, Node (hay Phần tử) là một đối tượng cơ bản chứa hai thành phần chính:

* Dữ liệu (Data): Lưu trữ thông tin của phần tử trong danh sách, có thể là bất kỳ kiểu dữ liệu nào (số nguyên, chuỗi, đối tượng, v.v.).
* Con trỏ (Next): Lưu trữ địa chỉ của phần tử tiếp theo trong danh sách. Đối với phần tử cuối cùng, con trỏ này trỏ đến null hoặc None (tùy vào ngôn ngữ lập trình).

### 1.2.4 Thêm phần tử vào đầu danh sách ( addFirst ).

1. Để thêm một phần tử vào đầu danh sách liên kết đơn, chúng ta cần thực hiện các bước sau:

* Tạo một node mới chứa giá trị cần thêm.
* Liên kết node mới với node đầu hiện tại (được trỏ bởi head).
* Cập nhật con trỏ head trỏ đến node mới, làm node mới trở thành phần tử đầu tiên trong danh sách.

2. Cách thức hoạt động:

* Tạo Node mới: Mỗi lần gọi hàm insertAtHead, một node mới được tạo ra với dữ liệu newData.
* Liên kết Node mới: Con trỏ next của node mới sẽ trỏ tới head hiện tại, tức là phần tử đầu tiên của danh sách.
* Cập nhật head: Cuối cùng, chúng ta cập nhật head để nó trỏ đến node mới, khiến node này trở thành phần tử đầu tiên của danh sách.
* Phương pháp thêm phần tử vào đầu danh sách liên kết đơn rất đơn giản và hiệu quả. Việc thay đổi con trỏ head giúp thao tác này có độ phức tạp thời gian là **O(1),** rất nhanh và không cần phải duyệt qua danh sách.

### 1.2.5 Thêm phần tử vào cuối danh sách ( addLast ).

1. Để thêm phần tử vào cuối danh sách ,bạn có thể làm điều này trong hai trường hợp: một là sử dụng std::list, và hai là tạo một **danh sách liên kết đơn (singly linked list)**

**2. Danh sách liên kết đơn (Singly Linked List):** Bạn phải tự quản lý bộ nhớ và thao tác với các con trỏ để thêm phần tử vào cuối danh sách.

**1.2.6 Thêm vào sau Node bất kì ( InsertAfter ).**

1. Giả sử bạn muốn thêm một phần tử vào sau một node đã biết trong danh sách liên kết đơn. Ta có.

* Duyệt qua danh sách để tìm node hiện tại.
* Tạo một node mới.
* Gán con trỏ next của node hiện tại trỏ đến node mới, và con trỏ next của node mới trỏ đến node sau node hiện tại.

**2. Ứng dụng:** Thao tác InsertAfter rất hữu ích khi bạn muốn chèn phần tử vào một vị trí cụ thể trong danh sách liên kết mà không cần phải duyệt lại toàn bộ danh sách.

### 1.2.7 Xoá phần tử khỏi danh sách (removeFirst ) .

1. Trong cấu trúc dữ liệu **danh sách liên kết,** việc **xóa phần tử** là một thao tác quan trọng. Một trong những thao tác cơ bản nhất là **xoá phần tử đầu tiên** của danh sách. Đây là thao tác thường gặp khi làm việc với danh sách liên kết, nơi phần tử đầu tiên cần được loại bỏ và bộ nhớ của nó được giải phóng.

2. Khi xoá phần tử đầu tiên trong danh sách liên kết đơn, cần:

* Kiểm tra danh sách rỗng: Nếu danh sách rỗng (head == nullptr), không thể xoá phần tử.
* Chỉnh sửa con trỏ head: Con trỏ head sẽ trỏ đến node tiếp theo, tức là phần tử thứ hai trong danh sách (nếu có).
* Giải phóng bộ nhớ: Node đầu tiên (cũ) sẽ bị xóa, giải phóng bộ nhớ để tránh rò rỉ bộ nhớ.

3. removeFirst():

* Hàm này xoá phần tử đầu tiên khỏi danh sách.
* Đầu tiên, nó kiểm tra nếu danh sách rỗng (head == nullptr), trong trường hợp này, không thể thực hiện thao tác xoá.
* Sau đó, nó lưu trữ con trỏ head vào một biến tạm (temp), di chuyển head đến node tiếp theo trong danh sách, và cuối cùng giải phóng bộ nhớ của node cũ (node ban đầu).

**4. Chức năng:** Xoá phần tử đầu tiên trong danh sách liên kết đơn.

**5. Ứng dụng:** Thao tác này rất hữu ích trong các tình huống cần loại bỏ phần tử đầu tiên của danh sách mà không ảnh hưởng đến các phần tử khác trong danh sách.

### 1.2.8 Xoá phần tử cuối khỏi danh sách ( removeLast ).

1. Trong cấu trúc dữ liệu **danh sách liên kết**, việc **xoá phần tử cuối cùng** là một thao tác quan trọng và thường gặp khi làm việc với danh sách liên kết. Thao tác này liên quan đến việc thay đổi các con trỏ trong danh sách để "ngắt" phần tử cuối cùng và giải phóng bộ nhớ của nó.

2. Để xoá phần tử cuối cùng trong danh sách liên kết đơn:

* Kiểm tra danh sách rỗng: Nếu danh sách rỗng (head == nullptr), không thể xoá phần tử.
* Nếu danh sách chỉ có một phần tử: Cập nhật head thành nullptr và giải phóng bộ nhớ.
* Nếu danh sách có nhiều phần tử: Duyệt qua danh sách để tìm phần tử cuối cùng và phần tử trước nó. Sau đó, thay đổi con trỏ next của phần tử trước cuối trỏ tới nullptr, và giải phóng bộ nhớ của phần tử cuối.

3. removeLast():

* Kiểm tra danh sách rỗng: Nếu head == nullptr, danh sách rỗng và không thể xoá phần tử.
* Danh sách chỉ có một phần tử: Nếu head->next == nullptr, danh sách chỉ có một phần tử. Trong trường hợp này, bạn chỉ cần xoá node duy nhất và đặt head thành nullptr.
* Danh sách có nhiều phần tử: Duyệt qua danh sách để tìm phần tử cuối cùng và phần tử trước nó. Sau đó, giải phóng bộ nhớ của phần tử cuối và thay đổi con trỏ next của phần tử trước cuối thành nullptr.

4. Chức năng: Xoá phần tử cuối cùng trong danh sách liên kết đơn.

5. Ứng dụng: Thao tác này rất hữu ích khi bạn muốn loại bỏ phần tử cuối cùng trong danh sách mà không ảnh hưởng đến các phần tử khác trong danh sách.

### 1.2.9 Xoá ở sau Node bất kì .

1. Trong cấu trúc dữ liệu **danh sách liên kết**, thao tác **xóa phần tử sau một node bất kỳ** (hay còn gọi là removeAfter) rất quan trọng khi bạn cần loại bỏ một phần tử sau một node đã biết trong danh sách. Việc này liên quan đến việc chỉnh sửa các con trỏ của các node sao cho phần tử bị xoá không còn tồn tại trong danh sách nữa.

2. Để thực hiện thao tác xoá phần tử sau một node bất kỳ (sau một node đã biết), bạn cần làm theo các bước sau:

* Tìm node trước node bị xóa: Node này sẽ là node mà bạn biết trước, và bạn muốn xóa phần tử ngay sau nó.
* Thay đổi con trỏ next của node trước: Cập nhật con trỏ next của node trước để nó trỏ đến node tiếp theo của node bị xoá (nếu có).
* Giải phóng bộ nhớ của node bị xoá: Sau khi thay đổi con trỏ, cần giải phóng bộ nhớ của node bị xoá.

### **3. Thực hiện thao tác** removeAfter**(Xoá sau một Node bất kỳ)**

### Cách thức hoạt động:

* **Kiểm tra nếu node trước không phải là nullptr**: Nếu node trước đó (prevNode) là nullptr hoặc không tồn tại, không thể thực hiện thao tác này.
* **Di chuyển con trỏ** next **của node trước**: Sau khi tìm được node cần xóa, thay đổi con trỏ next của node trước để nó trỏ đến node tiếp theo của node cần xóa (nếu có).
* **Giải phóng bộ nhớ của node bị xoá**.

4.removeAfter:

* Hàm này thực hiện thao tác xoá phần tử ngay sau prevNode.
* Đầu tiên, hàm kiểm tra nếu prevNode là nullptr hoặc nếu prevNode->next là nullptr (tức là không có phần tử nào sau prevNode để xoá).
* Nếu hợp lệ, nó lưu trữ con trỏ của phần tử cần xoá, cập nhật con trỏ next của node trước để trỏ đến node tiếp theo sau node bị xoá, và cuối cùng giải phóng bộ nhớ của node bị xoá.

5. Chức năng: Xoá phần tử ngay sau một node bất kỳ trong danh sách liên kết.

6. Ứng dụng: Thao tác này rất hữu ích khi bạn muốn loại bỏ một phần tử nào đó ngay sau một node đã biết mà không cần phải duyệt lại toàn bộ danh sách.

## 1.3 Cơ sở lý thuyết về sắp xếp.

### 1.3.1 Khái niệm.

1. Sắp xếp là quá trình thay đổi vị trí các phần tử trong một cấu trúc dữ liệu (ví dụ như mảng, danh sách liên kết) sao cho chúng tuân theo một thứ tự xác định. Thứ tự này có thể là:

* Tăng dần (Ascending Order): Các phần tử được sắp xếp sao cho phần tử nhỏ nhất ở đầu, phần tử lớn nhất ở cuối.
* Giảm dần (Descending Order): Các phần tử được sắp xếp sao cho phần tử lớn nhất ở đầu, phần tử nhỏ nhất ở cuối.

2. Các dạng sắp xếp phổ biến

Có rất nhiều thuật toán sắp xếp, nhưng một số thuật toán sắp xếp cơ bản và phổ biến nhất bao gồm:

* Sắp xếp nổi bọt (Bubble Sort):
  + Là một thuật toán đơn giản, liên tục so sánh và hoán đổi các phần tử liền kề cho đến khi mảng được sắp xếp.
  + Thời gian chạy: O(n²) trong trường hợp xấu nhất, tốt nhất là O(n) nếu mảng đã sắp xếp.
* Sắp xếp chọn (Selection Sort):
  + Thuật toán tìm phần tử nhỏ nhất (hoặc lớn nhất) trong mảng và đổi chỗ nó với phần tử ở đầu mảng.
  + Thời gian chạy: O(n²).
* Sắp xếp chèn (Insertion Sort):
  + Duyệt qua các phần tử từ trái sang phải, tại mỗi bước, phần tử đang xét sẽ được chèn vào đúng vị trí trong phần mảng đã được sắp xếp.
  + Thời gian chạy: O(n²) trong trường hợp xấu nhất, O(n) trong trường hợp tốt nhất khi mảng đã được sắp xếp.
* Sắp xếp hợp nhất (Merge Sort):
  + Sử dụng phương pháp chia và chinh phục (divide and conquer) để chia mảng thành các mảng con nhỏ hơn và sau đó hợp nhất chúng lại theo thứ tự.
  + Thời gian chạy: O(n log n), hiệu quả hơn so với các thuật toán sắp xếp O(n²).
* Sắp xếp nhanh (Quick Sort):
  + Cũng sử dụng phương pháp chia và chinh phục, nhưng thay vì chia mảng thành 2 phần đều như trong Merge Sort, Quick Sort chọn một "pivot" và chia mảng thành các phần nhỏ hơn dựa trên pivot này.
  + Thời gian chạy: O(n log n) trung bình, nhưng có thể O(n²) trong trường hợp xấu nhất.
* Sắp xếp đếm (Counting Sort):
  + Dành cho các dãy dữ liệu có giá trị nằm trong một phạm vi giới hạn, nó đếm số lần xuất hiện của mỗi phần tử và sau đó tái tạo lại dãy dữ liệu đã sắp xếp.
  + Thời gian chạy: O(n + k), với n là số lượng phần tử và k là phạm vi giá trị phần tử.
* Sắp xếp radix (Radix Sort):
  + Là một thuật toán sắp xếp không so sánh, sử dụng các chữ số (hoặc bit) của các phần tử để sắp xếp chúng theo từng đợt.
  + Thời gian chạy: O(nk), với n là số phần tử và k là số chữ số hoặc bit của mỗi phần tử.

3. Đặc điểm và phân loại các thuật toán sắp xếp

Các thuật toán sắp xếp có thể được phân loại dựa trên các đặc điểm sau:

* Thuật toán sắp xếp so sánh (Comparison-based Sorting): Các thuật toán này dựa trên việc so sánh các phần tử trong dãy. Ví dụ: Quick Sort, Merge Sort, Bubble Sort, Insertion Sort, Selection Sort.
  + Thời gian chạy trong trường hợp xấu nhất là O(n²) đối với các thuật toán đơn giản như Bubble Sort và Selection Sort, nhưng có thể đạt O(n log n) như Merge Sort và Quick Sort.
* Thuật toán sắp xếp không so sánh (Non-comparison-based Sorting): Các thuật toán này không dựa vào việc so sánh các phần tử mà thay vào đó sử dụng các đặc điểm khác của các phần tử (như giá trị của chúng) để xác định vị trí của chúng trong mảng.
  + Ví dụ: Counting Sort, Radix Sort, Bucket Sort.
* Sắp xếp ổn định (Stable Sorting): Thuật toán sắp xếp là ổn định nếu nó không thay đổi thứ tự của các phần tử có giá trị bằng nhau trong mảng. Ví dụ: Merge Sort, Insertion Sort.
* Sắp xếp không ổn định (Unstable Sorting): Thuật toán sắp xếp không ổn định có thể thay đổi thứ tự của các phần tử có giá trị bằng nhau. Ví dụ: Quick Sort, Selection Sort.

4. Độ phức tạp thời gian của các thuật toán sắp xếp

Độ phức tạp thời gian của thuật toán sắp xếp cho biết hiệu quả của thuật toán trong việc xử lý dãy dữ liệu. Đối với các thuật toán sắp xếp phổ biến, độ phức tạp thời gian có thể được phân loại như sau:

* O(n²): Thời gian chạy tỷ lệ với bình phương số phần tử. Các thuật toán như Bubble Sort, Selection Sort, Insertion Sort có độ phức tạp thời gian này trong trường hợp tồi tệ.
* O(n log n): Thời gian chạy tỷ lệ với n nhân với log n. Các thuật toán như Merge Sort, Quick Sort, Heap Sort thường có độ phức tạp này và rất hiệu quả trong thực tế.
* O(n): Thời gian chạy tỷ lệ với số lượng phần tử. Các thuật toán như Counting Sort, Bucket Sort có thể có độ phức tạp này nếu dữ liệu có phạm vi giá trị nhỏ và xác định.

5. Ứng dụng của các thuật toán sắp xếp

Sắp xếp đóng vai trò quan trọng trong rất nhiều ứng dụng trong khoa học máy tính, ví dụ như:

* Tìm kiếm nhị phân: Các thuật toán tìm kiếm như tìm kiếm nhị phân chỉ có thể áp dụng trên các mảng đã được sắp xếp.
* Quản lý dữ liệu: Sắp xếp là bước đầu tiên trong nhiều thuật toán và ứng dụng xử lý và phân tích dữ liệu, như phân loại danh sách người dùng, tài liệu, hoặc số liệu.
* Đo lường hiệu suất: Các thuật toán sắp xếp giúp đánh giá và tối ưu hóa các hệ thống, ví dụ như trong việc xử lý các danh sách lớn trong cơ sở dữ liệu.

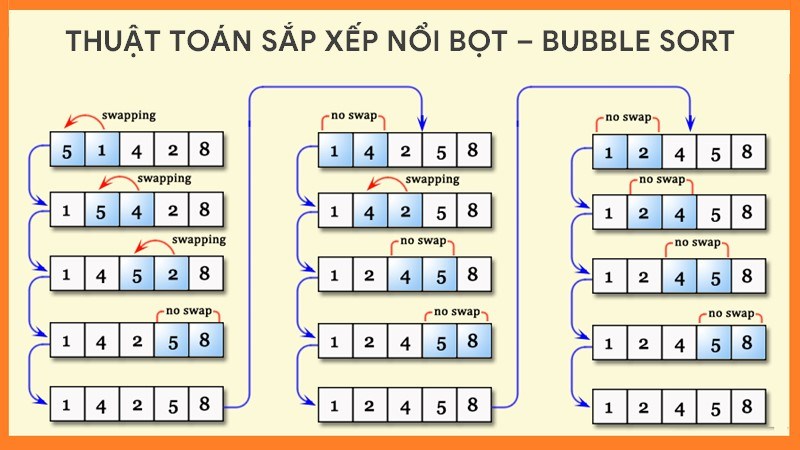
6. Các yếu tố ảnh hưởng đến việc chọn thuật toán sắp xếp

* Kích thước của dãy dữ liệu: Với dãy dữ liệu nhỏ, các thuật toán như Insertion Sort hoặc Bubble Sort có thể hiệu quả hơn, trong khi với dữ liệu lớn, các thuật toán như Merge Sort hoặc Quick Sort sẽ cho hiệu quả cao hơn.
* Tính ổn định của thuật toán: Nếu thứ tự của các phần tử có giá trị giống nhau là quan trọng, cần chọn thuật toán sắp xếp ổn định.
* Dữ liệu đã được sắp xếp một phần: Một số thuật toán như Insertion Sort có thể hoạt động rất nhanh trên dữ liệu gần như đã được sắp xếp.

### 1.3.2 Sắp xếp nổi bọt ( Bubble sort ).

### **1. Định nghĩa:**

* Sắp xếp nổi bọt là thuật toán sắp xếp mà tại mỗi bước, thuật toán so sánh các phần tử liền kề trong danh sách và hoán đổi chúng nếu chúng không theo thứ tự mong muốn (ví dụ: sắp xếp tăng dần). Quá trình này lặp đi lặp lại cho đến khi danh sách đã được sắp xếp hoàn toàn.



Hình 1.2 Thuật toán sắp xếp nổi bọt -Bubble Sort.

2. Cơ chế hoạt động của Bubble Sort:

* Thuật toán hoạt động theo cơ chế dần dần đưa phần tử lớn nhất (hoặc nhỏ nhất) lên cuối dãy sau mỗi vòng lặp qua danh sách. Chính vì vậy nó được gọi là "nổi bọt" (bubble), như những bong bóng khí nổi lên mặt nước.
* Mỗi vòng lặp qua danh sách sẽ so sánh từng cặp phần tử liền kề và hoán đổi chúng nếu chúng không thỏa mãn thứ tự mong muốn. Sau một vòng lặp, phần tử lớn nhất (hoặc nhỏ nhất) sẽ được đặt ở cuối danh sách, và quá trình lặp lại cho phần còn lại của danh sách chưa được sắp xếp.

3. Quy trình thuật toán Bubble Sort:

a.Bắt đầu từ đầu danh sách: So sánh phần tử hiện tại với phần tử kế tiếp.

b.Hoán đổi nếu cần thiết: Nếu phần tử hiện tại lớn hơn phần tử kế tiếp (trong trường hợp sắp xếp tăng dần), chúng sẽ được hoán đổi vị trí.

c.Lặp lại: Tiếp tục so sánh các cặp phần tử liên tiếp cho đến khi không còn phần tử nào cần hoán đổi.

d.Lặp lại với phần chưa sắp xếp: Sau mỗi vòng, phần tử lớn nhất sẽ được "nổi lên" cuối danh sách, do đó trong vòng lặp tiếp theo, ta sẽ giảm đi một phần tử (vì nó đã được sắp xếp).

e. Kết thúc khi không còn bất kỳ hoán đổi nào trong một vòng lặp, nghĩa là danh sách đã được sắp xếp hoàn toàn.

4. Độ phức tạp thời gian (Time Complexity)

* Trường hợp xấu nhất (Worst-case): O(n²), khi dãy dữ liệu hoàn toàn đảo ngược, cần phải thực hiện các phép hoán đổi cho tất cả các cặp phần tử.
* Trường hợp tốt nhất (Best-case): O(n), khi dãy dữ liệu đã được sắp xếp, thuật toán chỉ cần một lần kiểm tra và không có hoán đổi nào.
* Trường hợp trung bình (Average-case): O(n²), với các tình huống dữ liệu ngẫu nhiên, thuật toán phải so sánh và hoán đổi các phần tử một cách thường xuyên.

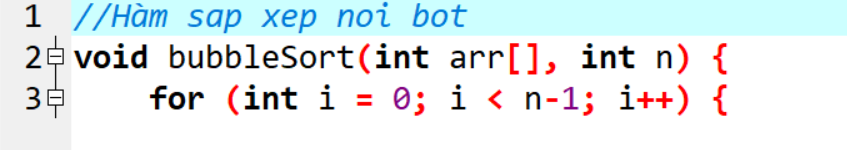
5. Ưu điểm và nhược điểm của Bubble Sort

Ưu điểm:

* Đơn giản và dễ hiểu: Thuật toán rất dễ cài đặt và dễ giải thích.
* Không yêu cầu bộ nhớ bổ sung: Đây là thuật toán sắp xếp tại chỗ, tức là không yêu cầu bộ nhớ bổ sung ngoài mảng cần sắp xếp (O(1) bộ nhớ bổ sung).

Nhược điểm:

* Hiệu suất kém: Độ phức tạp thời gian của Bubble Sort là O(n²) trong trường hợp xấu nhất, do đó thuật toán này trở nên không hiệu quả khi số lượng phần tử tăng lên.
* Chậm đối với dãy dữ liệu lớn: Với các dãy dữ liệu lớn, thuật toán sẽ rất chậm và không phù hợp để sử dụng trong các ứng dụng yêu cầu hiệu suất cao.



Hình 1.3 Hàm sắp xếp nổi bọt Bubble Sort.

6. Tổng hợp.

* Sắp xếp nổi bọt (Bubble Sort) là một thuật toán đơn giản và dễ hiểu, nhưng không hiệu quả cho các tập dữ liệu lớn do độ phức tạp thời gian O(n²).
* Tuy nhiên, Bubble Sort có một số cải tiến, như dừng sớm nếu danh sách đã được sắp xếp, giúp cải thiện hiệu suất trong trường hợp tốt nhất.
* Ứng dụng của thuật toán này chủ yếu dùng trong giáo dục để giải thích các thuật toán sắp xếp cơ bản. Trong thực tế, các thuật toán sắp xếp khác như Quick Sort, Merge Sort, và Heap Sort thường được sử dụng vì hiệu suất của chúng tốt hơn.

## 1.4 Cơ sở lý thuyết về tìm kiếm.

### 1.4.1 khái niệm.

* Tìm kiếm (Searching) là quá trình tìm kiếm một phần tử trong một tập hợp dữ liệu, như mảng, danh sách hoặc cây. Mục tiêu là xác định sự tồn tại của phần tử và trả về vị trí của nó nếu có.

1. Các phương pháp tìm kiếm chính:

* Tìm kiếm tuyến tính (Linear Search):
  + Hoạt động: Duyệt qua từng phần tử trong mảng và so sánh với giá trị cần tìm.
  + Ưu điểm: Dễ cài đặt, không yêu cầu dữ liệu sắp xếp.
  + Nhược điểm: Chậm với dữ liệu lớn, độ phức tạp O(n).
* Tìm kiếm nhị phân (Binary Search):
  + Hoạt động: Áp dụng cho mảng đã sắp xếp. Chia đôi mảng và so sánh với giá trị cần tìm.
  + Ưu điểm: Nhanh hơn tìm kiếm tuyến tính, độ phức tạp O(log n).
  + Nhược điểm: Chỉ áp dụng cho mảng đã sắp xếp.

2. Độ phức tạp thời gian:

* Tìm kiếm tuyến tính: O(n) trong trường hợp xấu nhất.
* Tìm kiếm nhị phân: O(log n), yêu cầu dữ liệu phải được sắp xếp.

3. Các phương pháp tìm kiếm nâng cao:

* Tìm kiếm trong bảng băm: Thời gian O(1) trung bình, rất nhanh nếu sử dụng băm.
* Tìm kiếm trong cây nhị phân (Binary Search Tree): Thời gian O(log n) trong trường hợp lý tưởng.

### 1.4.2 Tìm kiếm tuần tự ( Linear Search ).

1. Tìm kiếm tuần tự (Linear Search) là thuật toán tìm kiếm đơn giản và dễ hiểu, dùng để tìm một phần tử trong danh sách bằng cách duyệt qua từng phần tử của danh sách một cách lần lượt từ đầu đến cuối, và so sánh nó với giá trị cần tìm. Nếu phần tử cần tìm tồn tại trong danh sách, thuật toán sẽ trả về chỉ mục của phần tử đó; nếu không, trả về giá trị báo không tìm thấy (thường là -1).

2. Cách thức hoạt động của Tìm kiếm tuần tự:

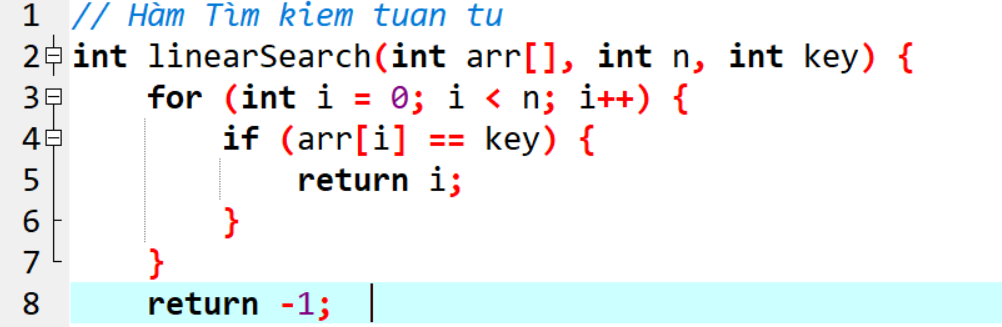
* + Bắt đầu từ phần tử đầu tiên trong danh sách hoặc mảng.
  + So sánh phần tử hiện tại với giá trị cần tìm.
  + Nếu phần tử hiện tại bằng giá trị cần tìm**,** trả về chỉ mục của phần tử đó.
  + Nếu không, tiếp tục duyệt qua các phần tử tiếp theo cho đến khi tìm thấy hoặc hết danh sách.
  + Nếu không tìm thấy phần tử, thuật toán trả về **-1** (hoặc một giá trị báo lỗi).

3. Độ phức tạp thời gian:

* + Trường hợp tốt nhất: O(1) khi phần tử cần tìm ở vị trí đầu tiên.
  + Trường hợp xấu nhất: O(n) khi phần tử cần tìm ở cuối danh sách hoặc không có trong danh sách.
  + Trường hợp trung bình: O(n), vì thuật toán cần kiểm tra trung bình n/2 phần tử.

4. Ưu điểm và Nhược điểm:

* **Ưu điểm**:
* Dễ cài đặt và đơn giản.
* Áp dụng được cho danh sách chưa sắp xếp.
* **Nhược điểm**:
* Chậm đối với các danh sách lớn vì phải duyệt qua từng phần tử.
* Độ phức tạp O(n), kém hiệu quả khi so với các thuật toán tìm kiếm khác như tìm kiếm nhị phân.



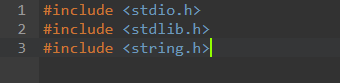
Hình 1.4 Hàm tìm kiếm tuần tự .

# Chương II: XÂY DỰNG CHƯƠNG TRÌNH.

## 2.1 Giới thiệu thuật toán.

### 2.1.1 Các thư viện trong chương trình.

* string: Cung cấp lớp string để làm việc với chuỗi ký tự
* stdlib.h là thư viện chuẩn trong C cung cấp các chức năng như cấp phát bộ nhớ, xử lý chuỗi, sinh số ngẫu nhiên, quản lý chương trình, v.v.
* stdio.h là thư viện chuẩn trong **C** và **C++** cung cấp các hàm để thao tác với I/O (nhập/xuất), bao gồm việc đọc và ghi từ bàn phím và tệp tin.

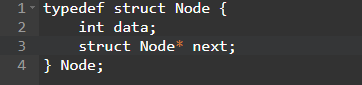
,

Hình 1.5 Thư viện trong chương trình.

## 2.2 Xây dựng chương trình.

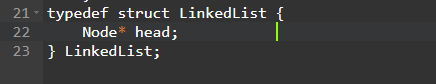
### 2.2.1 Khai báo cấu trúc Node.

1. Cấu trúc dữ liệu của một node trong danh sách liên kết.



Hình 1.6 Hàm khai báo cấu trúc Node.

### 2.2.2 Các hàm khai báo của cấu trúc LIST.



Hình 1.7 Hàm khai báo của cấu trúc List.

* 1. LinkedList: Đây là danh sách liên kết, có con trỏ head trỏ đến node đầu tiên.

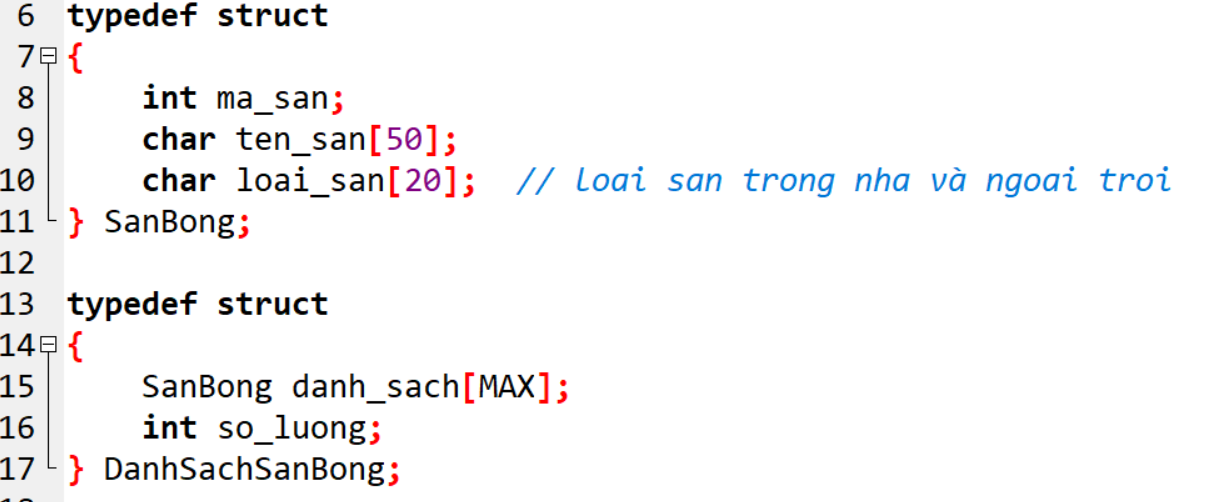
### 2.2.3 Hàm khởi tạo .

1. Cấu trúc sân bóng.

* Cấu trúc sân bóng chứa thông tin về một sân bóng, bao gồm mã sân, tên sân và loại sân (trong nhà hoặc ngoài trời).

2. Cấu trúc danh sách sân bóng .

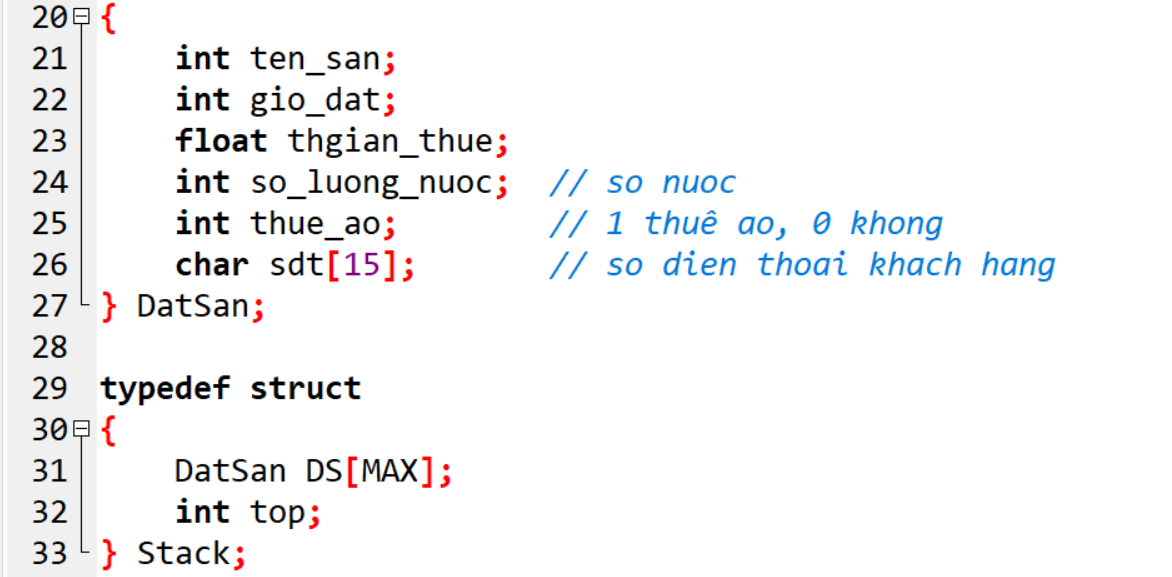
* Cấu trúc danh sách sân bóng lưu trữ danh sách các sân bóng, có thể chứa tối đa MAX sân bóng, và có một biến đếm số lượng để theo dõi số lượng sân bóng hiện tại trong danh sách.



Hình 1.8 Hàm khởi tạo sân bóng và danh sách sân bóng.

3. Cấu trúc đặt sân.

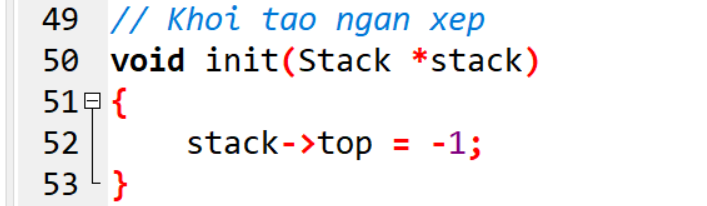
* Cấu trúc đặt sân chứa thông tin về một lần đặt sân, bao gồm tên sân, giờ đặt, thời gian thuê, số lượng nước, thuê áo, và số điện thoại khách hàng.



Hình 1.9 Hàm khởi tạo ngăn xếp đặt sân.

4. Cấu trúc Stack.

* Cấu trúc Stack là một ngăn xếp (stack) lưu trữ các yêu cầu đặt sân, trong đó danh sách là mảng các đối tượng đặt sân và top là biến theo dõi chỉ số của phần tử trên cùng trong ngăn xếp.



Hình 1.10 Hàm khởi tạo ngăn xếp stack.

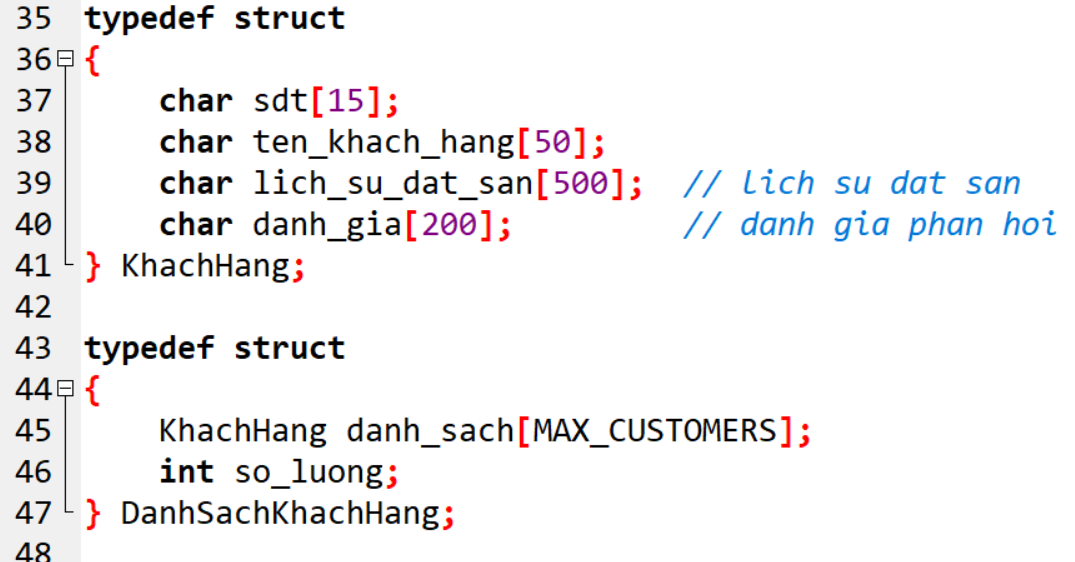
* is\_empty: Kiểm tra ngăn xếp có rỗng không.
* is\_full: Kiểm tra ngăn xếp có đầy không.
* push: Thêm phần tử vào ngăn xếp.
* pop: Lấy phần tử ra khỏi ngăn xếp.

5. Cấu trúc khách hàng.

* Cấu trúc khách hàng lưu trữ thông tin của một khách hàng, bao gồm số điện thoại, tên khách hàng, lịch sử đặt sân và đánh giá của khách hàng.

6. Cấu trúc danh sách khách hàng.

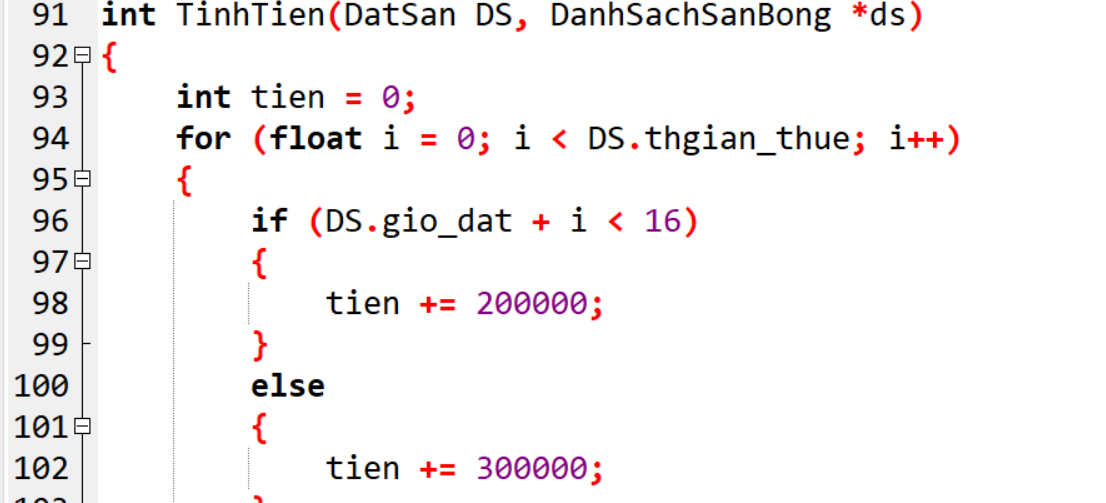
* Cấu trúc danh sách khách hàng lưu trữ danh sách các khách hàng, với mảng danh sách chứa các đối tượng đối khách hàng và số lượng là số lượng khách hàng hiện có trong danh sách.



Hình 1.11 Hàm khởi tạo khách hàng và danh sách khách hàng.

### 2.2.4 Hàm quản lý đặt sân .

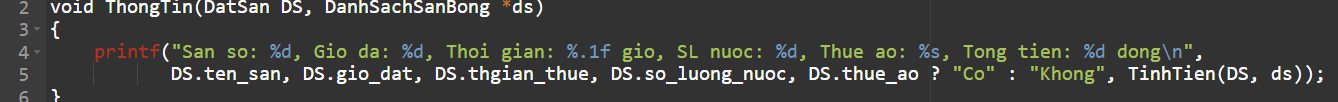
1. Tính tiền thuê sân



Hình 1.12 Hàm tính tiền thuê sân.

* Mục đích: Tính toán tiền thuê sân dựa trên thời gian thuê, loại sân và các chi phí dịch vụ.
* Cách hoạt động:
* Hàm tính tiền theo số giờ thuê sân. Nếu giờ thuê trước 16:00, mỗi giờ tính 200,000 đồng, sau 16:00 là 300,000 đồng.
* Tính thêm phí nếu sân là "trong nhà" (thêm 10%).
* Tính phí cho số lượng nước uống và nếu thuê áo.

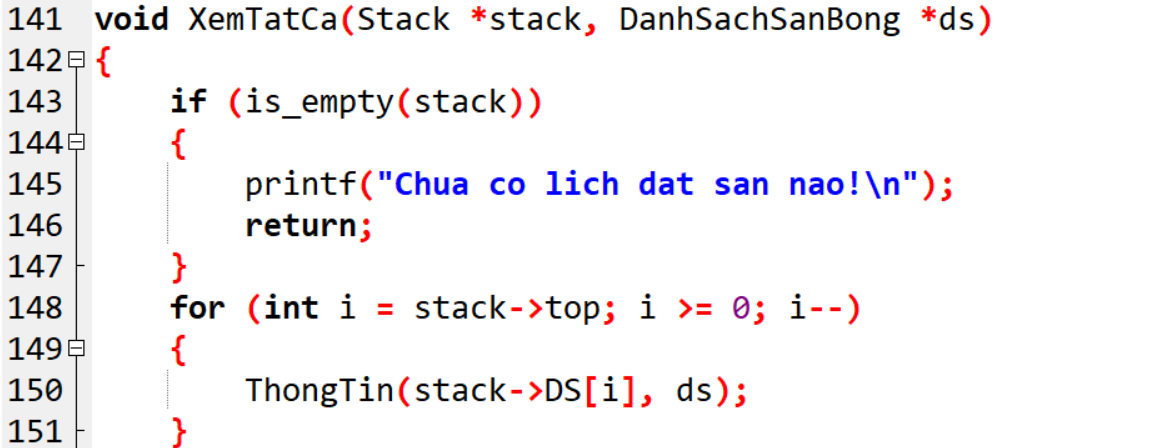
2. In thông tin chi tiết về lịch đặt sân.



Hình 1.13 Hàm in thông tin lịch đặt sân.

* Mục đích: In ra thông tin chi tiết của một lịch đặt sân, bao gồm mã sân, giờ đặt, thời gian thuê, số lượng nước, có thuê áo hay không, và tổng tiền (tính bằng hàm tính tiền).

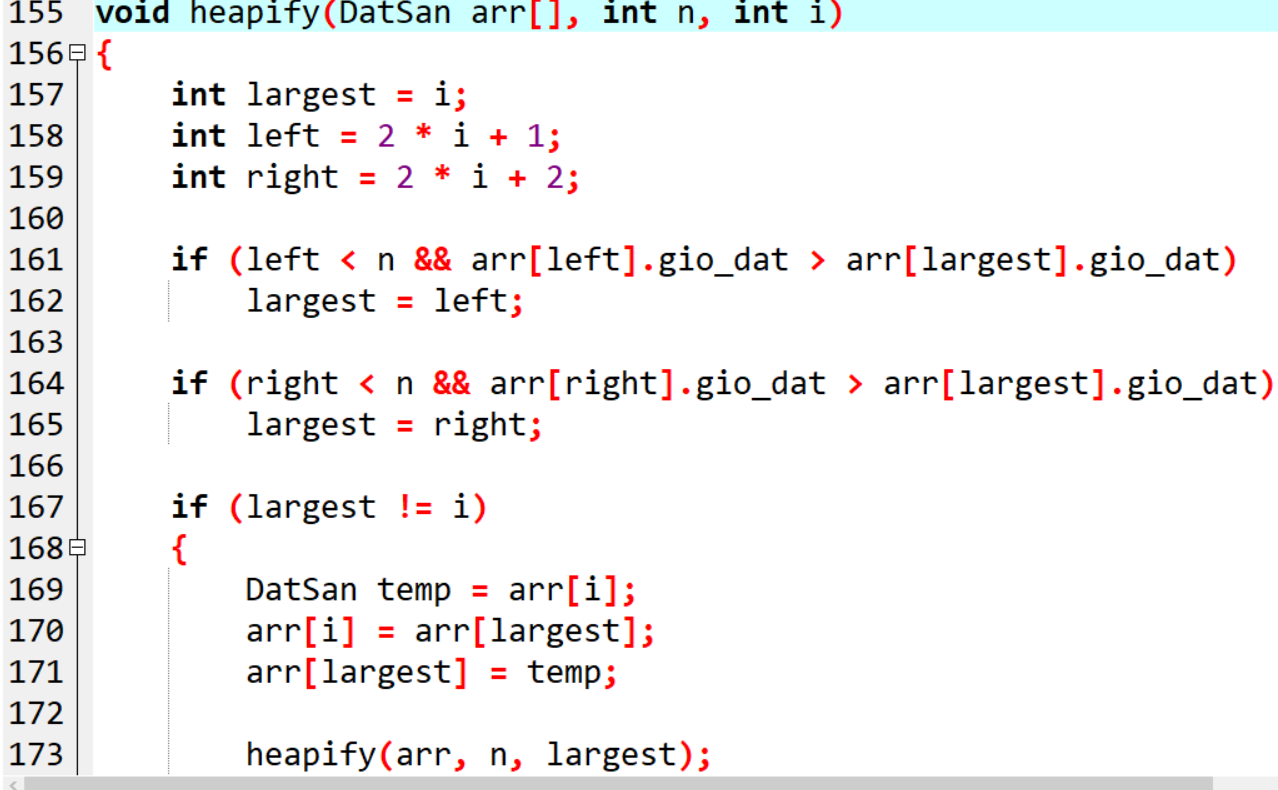
3. Xem tất cả các lịch đặt sân trong ngăn xếp.



Hình 1.14 Hàm xem lịch đặt sân .

* Mục đích: In ra tất cả các lịch đặt sân có trong ngăn xếp.
* Cách hoạt động: Hàm duyệt từ phần tử trên cùng (top) của ngăn xếp và gọi ThongTin để in thông tin từng lịch đặt. Nếu ngăn xếp trống, nó sẽ in thông báo "Chua co lich dat san nao!".

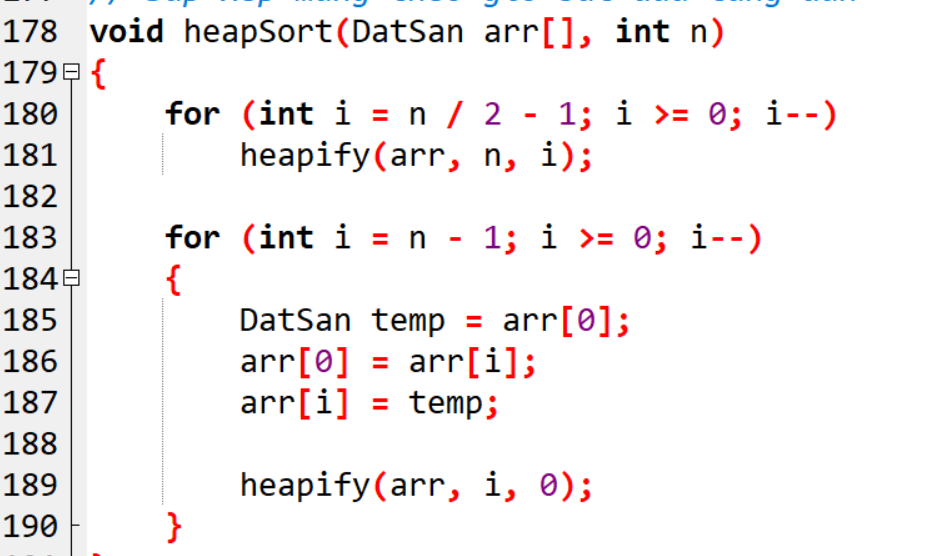
4. Tính chất của cây heap tại vị trí i.



Hình 1.15 Hàm tính chất cây heap.

* Mục đích: Đảm bảo tính chất của cây heap tại một nút.
* Cách hoạt động: Hàm heapify giúp sắp xếp lại cây heap sau khi có sự thay đổi (hoán đổi phần tử).

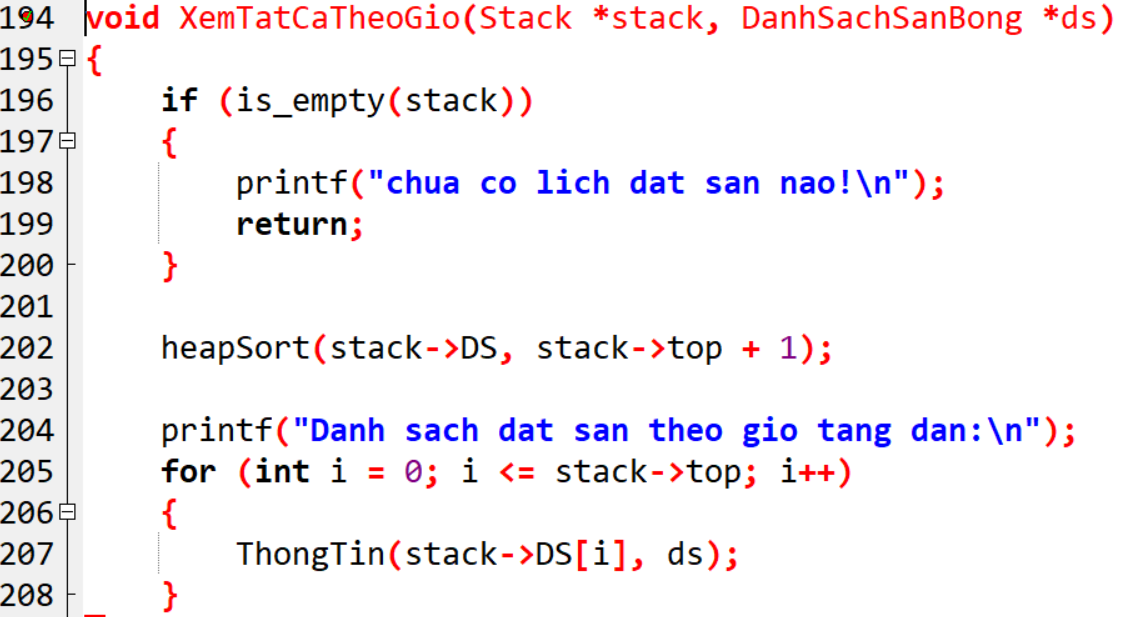
5. Sắp xếp mảng DatSan theo giờ đặt sân.



Hình 1.16 Hàm sắp xếp mảng DatSan theo giờ đặt sân.

* Mục đích: Sắp xếp các lịch đặt sân theo giờ bắt đầu (tăng dần) bằng cách sử dụng thuật toán Heap Sort.
* Cách hoạt động:Đầu tiên, xây dựng một cây heap.

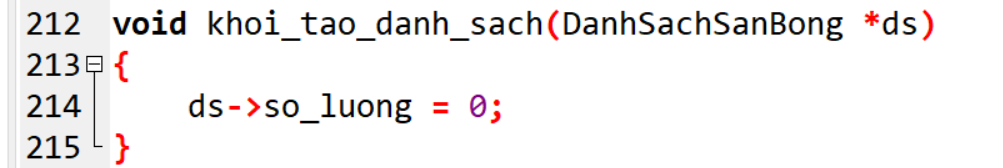
6. In ra tất cả các lịch đặt theo giờ bắt đầu tăng dần.



Hình 1.17 Hàm in ra tất cả các lịch đặt theo giờ bắt đầu tăng dần.

* Mục đích: Hàm này sẽ in ra tất cả các lịch đặt sân theo thứ tự giờ bắt đầu từ sớm đến muộn (tăng dần). Đây là một hàm được dùng để hiển thị danh sách các lịch đặt sân đã được thực hiện trong ngăn xếp (stack), sắp xếp lại các lịch đặt theo giờ và sau đó in ra thông tin chi tiết của từng lịch đặt.
* Cách hoạt động:
* Kiểm tra ngăn xếp rỗng: Trước tiên, hàm kiểm tra xem ngăn xếp có chứa lịch đặt hay không. Nếu ngăn xếp trống, chương trình sẽ in thông báo "Chua co lich dat san nao!" và kết thúc.
* Sắp xếp danh sách theo giờ: Nếu ngăn xếp không trống, hàm sẽ sắp xếp danh sách các lịch đặt sân trong ngăn xếp theo giờ bắt đầu (tăng dần) bằng cách sử dụng hàm heapSort.
* In danh sách: Sau khi sắp xếp, hàm sẽ duyệt qua các phần tử trong ngăn xếp từ trên xuống dưới (top đến 0), và in thông tin chi tiết của mỗi lịch đặt sân bằng cách gọi hàm ThongTin.

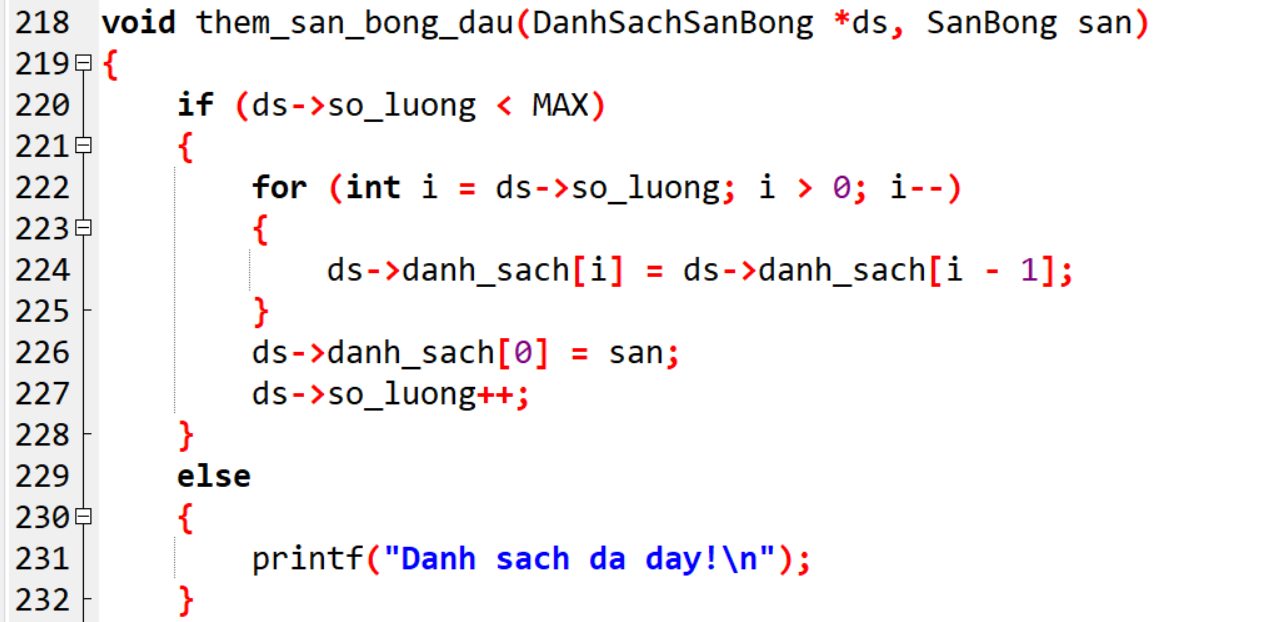
### 2.2.5 Hàm khởi tạo danh sách sân bóng.



Hình 1.18 Hàm khởi tạo danh sách sân bóng.

* Mục đích: Hàm này có tác dụng khởi tạo một danh sách sân bóng (DanhSachSanBong) bằng cách thiết lập số lượng sân ban đầu là 0. Đây là bước chuẩn bị cho việc thêm các sân bóng vào danh sách sau này.
* Cách hoạt động:
* Hàm nhận tham số là một con trỏ đến một đối tượng DanhSachSanBong.
* Sau khi gọi hàm, trường so\_luong của danh sách này được gán giá trị 0, có nghĩa là ban đầu danh sách chưa có sân nào.
* Hàm này chỉ đơn giản là khởi tạo trạng thái ban đầu của đối tượng danh sách sân bóng.

### 2.2.6 Hàm thêm sân bóng vào đầu danh sách.



Hình 1.19 hàm thêm sân bóng vào đầu danh sách .

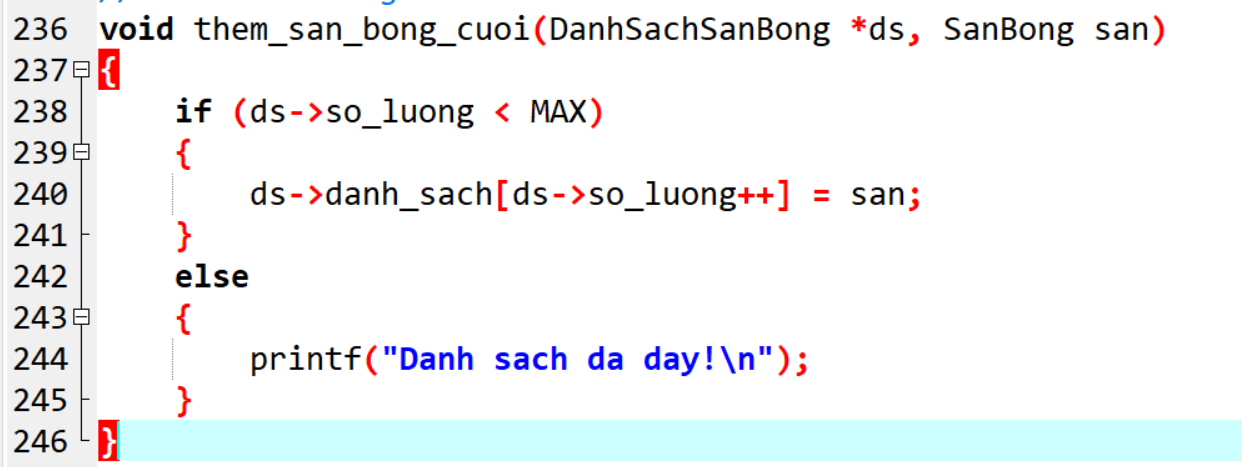
1. Điều kiện kiểm tra: Kiểm tra nếu số lượng sân bóng hiện tại trong danh sách chưa đạt giới hạn tối đa (so\_luong < MAX). Nếu có thể thêm, tiếp tục thực hiện các bước sau.

2. Dịch chuyển phần tử: Sử dụng vòng lặp for để di chuyển tất cả các phần tử hiện tại sang một vị trí phía sau, từ vị trí cuối cùng đến vị trí đầu tiên (i > 0).

3. Thêm sân bóng vào đầu: Sau khi dịch chuyển các phần tử, ta chèn sân bóng mới vào vị trí đầu tiên (ds->danh\_sach[0] = san).

4. Cập nhật số lượng: Tăng số lượng sân bóng (ds->so\_luong++).

### 2.2.7 Hàm thêm sân bóng vào cuối danh sách .



Hình 1.20 Hàm thêm sân bóng vào cuối danh sách.

1. Điều kiện kiểm tra: Kiểm tra nếu số lượng sân bóng hiện tại trong danh sách chưa đạt giới hạn tối đa (so\_luong < MAX).

2. Thêm vào cuối danh sách: Nếu danh sách chưa đầy, ta đơn giản thêm đối tượng sân bóng vào vị trí cuối cùng (ds->danh\_sach[ds->so\_luong++] = san).

3. Cập nhật số lượng: Sau khi thêm, ta tự động tăng số lượng sân bóng (ds->so\_luong++).

### 2.2.8 Hàm thêm sân bóng bất kì.

* Kiểm tra điều kiện đầu vào ds->so\_luong < MAX: Kiểm tra xem danh sách còn đủ chỗ để thêm sân bóng hay không.
* k >= 0 && k <= ds->so\_luong: Kiểm tra vị trí k có hợp lệ không, tức là phải nằm trong phạm vi từ 0 đến số lượng phần tử hiện tại trong danh sách (bao gồm cả việc chèn vào cuối danh sách).
* Nếu điều kiện này không thỏa mãn, hàm sẽ thông báo "vị trí không hợp lệ".

2. Dịch chuyển phần tử:

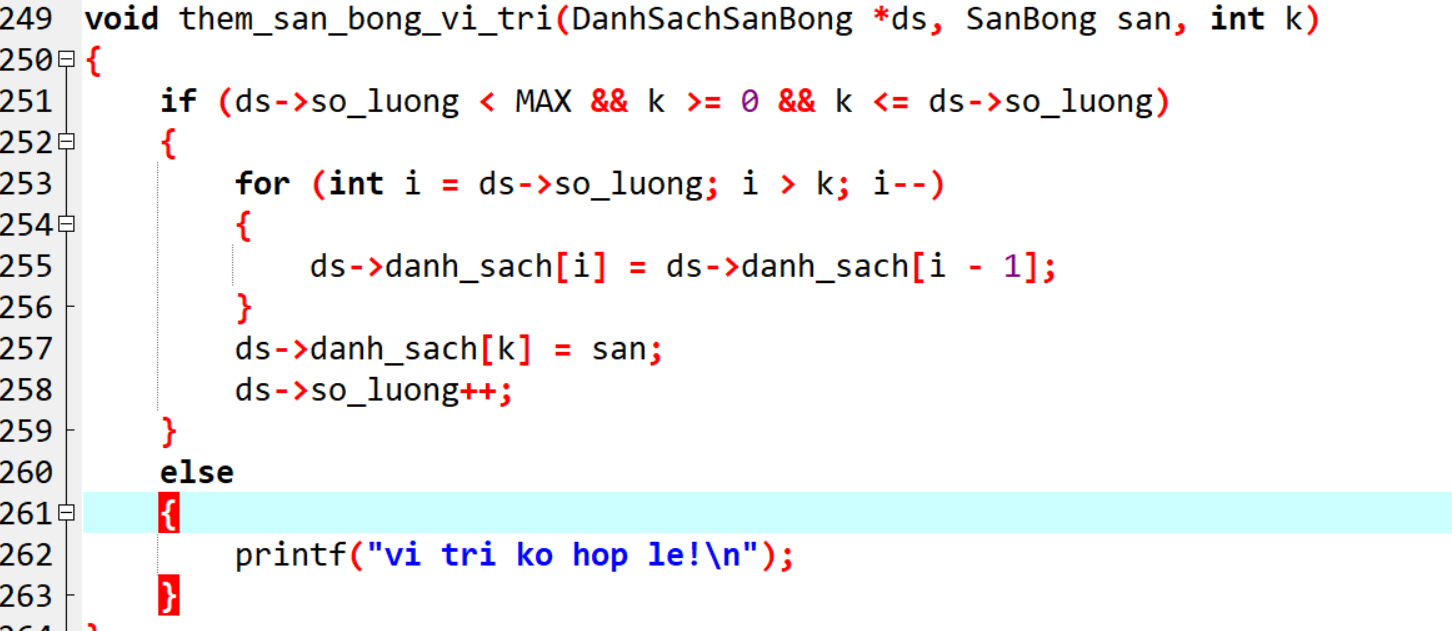
* Nếu các điều kiện trên thỏa mãn, hàm sẽ sử dụng vòng lặp for để dịch chuyển các phần tử từ vị trí k đến vị trí so\_luong - 1 sang một vị trí phía sau, tạo không gian cho phần tử mới.
* Cụ thể, các phần tử từ ds->danh\_sach[i] sẽ được gán cho ds->danh\_sach[i+1], bắt đầu từ phần tử cuối cùng và đi ngược lên vị trí k.

3. Chèn phần tử mới:

* Sau khi dịch chuyển các phần tử, phần tử san được gán vào vị trí k trong danh sách.

4. Cập nhật số lượng:

* Sau khi thêm phần tử vào danh sách, số lượng phần tử trong danh sách (ds->so\_luong) được tăng lên 1.



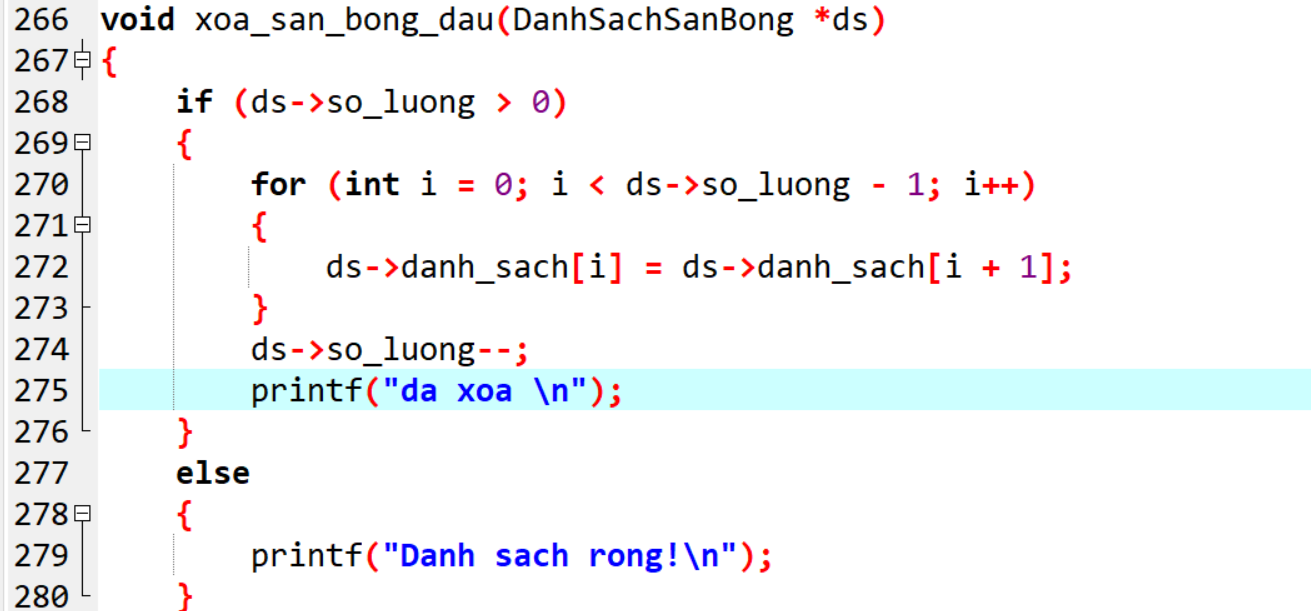
Hình 1.21 Hàm thêm sân bóng bất kì.

### 2.2.9 Hàm xóa sân bóng ở đầu danh sách.

1. Kiểm tra danh sách không rỗng: Hàm bắt đầu bằng việc kiểm tra số lượng sân bóng (so\_luong) trong danh sách. Nếu so\_luong lớn hơn 0, nghĩa là danh sách có ít nhất một sân bóng để xóa.

2. Di chuyển các phần tử còn lại: Sau đó, hàm sẽ thực hiện vòng lặp để di chuyển tất cả các phần tử từ vị trí thứ 2 trở đi lên một vị trí phía trước, nhằm "xóa" phần tử đầu tiên. Điều này đảm bảo rằng phần tử thứ nhất bị xóa đi, và các phần tử còn lại được giữ lại ở các vị trí cũ.

3. Giảm số lượng sân bóng: Cuối cùng, hàm giảm giá trị của so\_luong để phản ánh việc xóa một sân bóng.



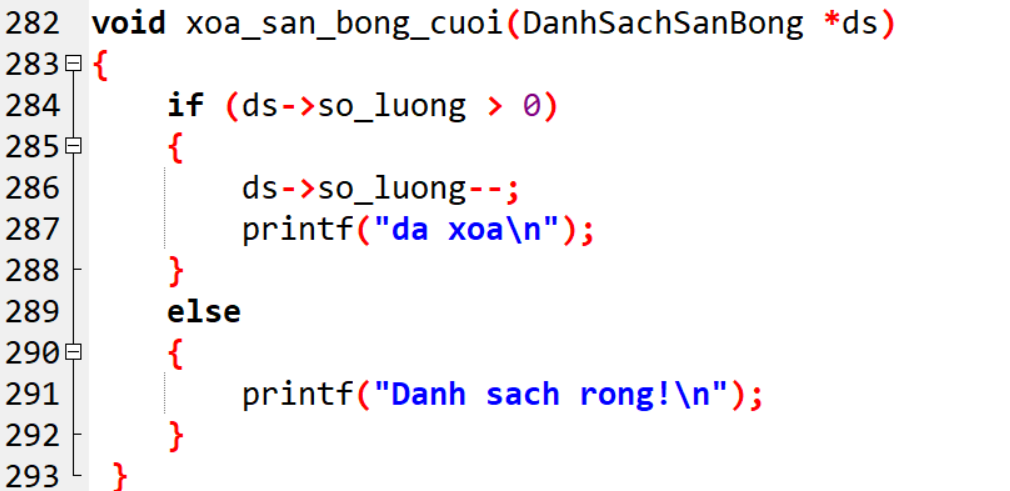
Hình 1.22 Hàm xóa sân bóng ở đầu danh sách.

### 2.2.10 Hàm xóa sân bóng ở cuối danh sách.

1. Kiểm tra danh sách không rỗng: Tương tự như hàm trên, hàm này kiểm tra xem danh sách có sân bóng hay không.

2. Giảm số lượng sân bóng: Nếu danh sách không rỗng, hàm chỉ cần giảm giá trị của so\_luong đi một đơn vị, vì phần tử cuối cùng đã được "xóa" khi giảm so\_luong.

3. Không cần phải di chuyển phần tử: Vì chỉ xóa phần tử cuối cùng, không cần phải di chuyển các phần tử khác trong danh sách.



Hình 1.23 Hàm xóa sân bóng ở cuối danh sách.

### 2.2.11 Hàm xóa một sân bóng tại một vị trí cụ thể.

1. Kiểm tra điều kiện hợp lệ:

* if (ds->so\_luong > 0 && k >= 0 && k < ds->so\_luong): Điều kiện này kiểm tra 3 yếu tố:
* ds->so\_luong > 0: Danh sách phải có ít nhất một sân bóng.
* k >= 0: Vị trí k phải là chỉ số hợp lệ (không âm).
* k < ds->so\_luong: Vị trí k không vượt quá số lượng sân bóng hiện tại trong danh sách.

2. Di chuyển các phần tử phía sau vị trí k lên một ô:

* for (int i = k; i < ds->so\_luong - 1; i++): Vòng lặp này bắt đầu từ chỉ số k và di chuyển từng phần tử từ vị trí i + 1 về vị trí i. Điều này sẽ "lấp đầy" khoảng trống mà sân bóng tại vị trí k đã để lại.

3. Giảm số lượng sân bóng trong danh sách:

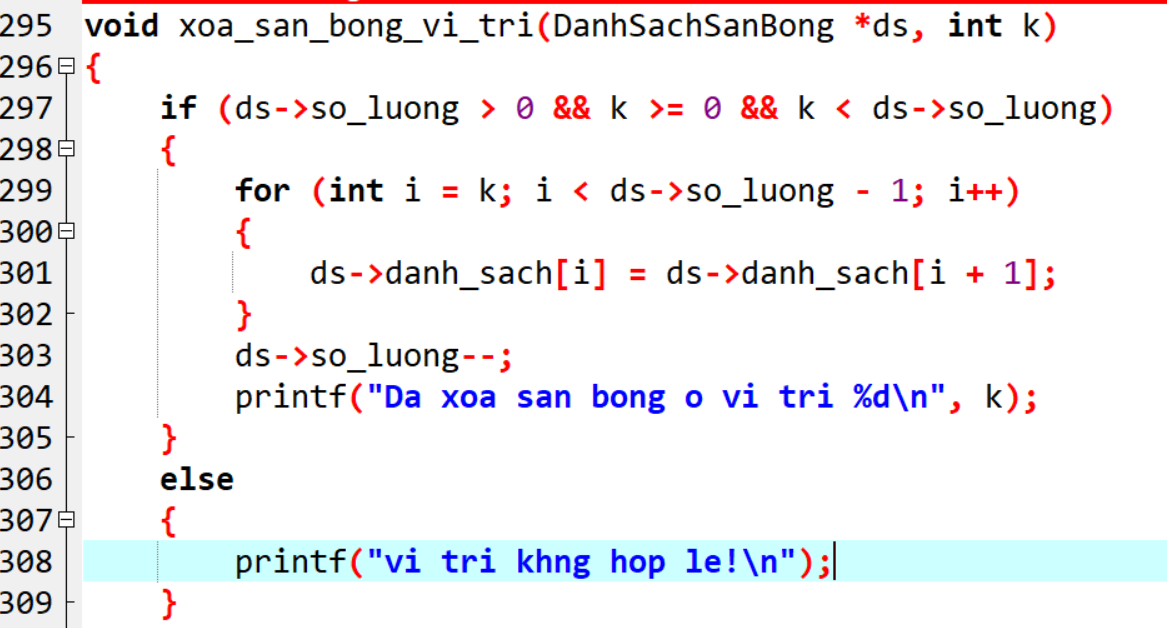
* ds->so\_luong--;: Sau khi đã di chuyển các phần tử, số lượng sân bóng trong danh sách giảm đi 1 để phản ánh việc đã xóa một phần tử.

4. Thông báo kết quả:

* printf("Da xoa san bong o vi tri %d\n", k);: In ra thông báo cho người dùng biết rằng sân bóng tại vị trí k đã được xóa.

5. Xử lý trường hợp không hợp lệ:

* Nếu vị trí k không hợp lệ (ví dụ: âm hoặc vượt quá số lượng sân bóng), hàm sẽ in thông báo vi trí không hợp lệ !.

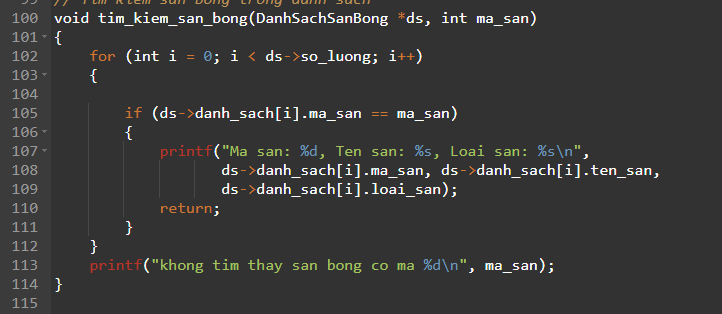


Hình 1.24 Hàm xóa một sân bóng tại một vị trí cụ thể.

### 2.2.12 Hàm kiểm tra sân bóng có tồn tại hay không.

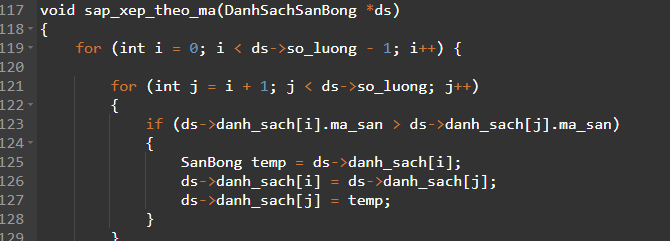
1. Tìm kiếm sân bóng trong danh sách.

* Hàm tìm kiếm sân bóng: Duyệt qua mảng sân bóng và tìm sân theo mã sân.Nếu tìm thấy, in thông tin của sân bóng đó, nếu không tìm thấy thì thông báo không có sân bóng nào với mã đó.



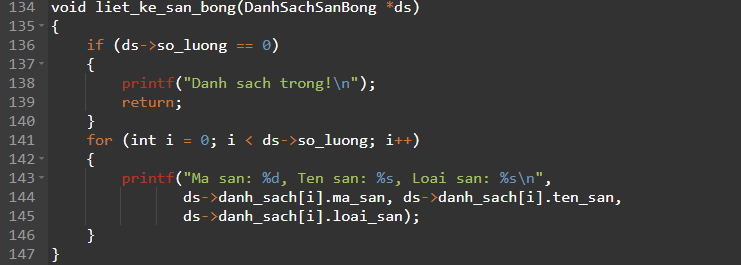
Hình 1.25 Hàm tìm kiếm sân bóng trong danh sách.

* 1. Sắp xếp danh sách sân bóng theo mã sân.
* Hàmtìm kiếm sân bóng: Duyệt qua mảng sân bóng và tìm sân theo mã sân. Nếu tìm thấy, in thông tin của sân bóng đó, nếu không tìm thấy thì thông báo không có sân bóng nào với mã đó.



Hình 1.26 Hàm sắp xếp danh sách sân bóng theo mã sân.

* 1. Liệt kê danh sách sân bóng.
* Hàmliệt kê sân bóng: Liệt kê tất cả sân bóng trong danh sách. Nếu danh sách trống, thông báo "Danh sách trống".



Hình 1.27 Hàm liệt kê danh sách sân bóng.

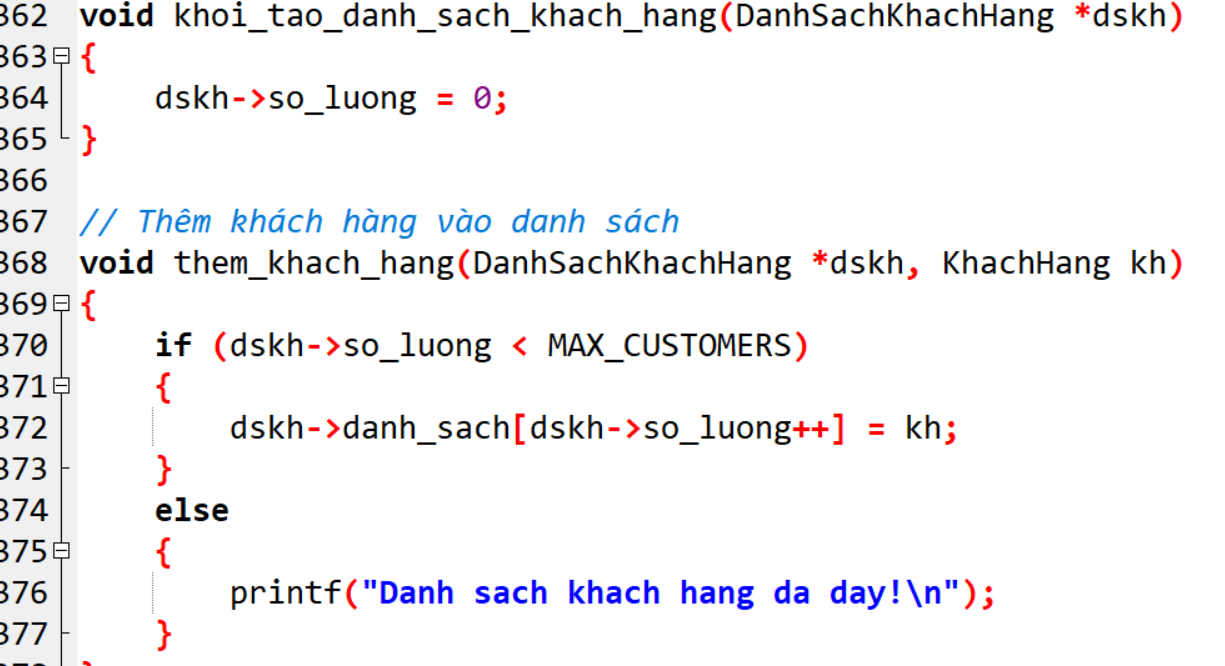
### 2.2.13 Hàm khởi tạo thông tin của khác hàng.

1. Khởi tạo danh sách khách hàng.

* Mục đích: Hàm này có tác dụng khởi tạo danh sách khách hàng (DanhSachKhachHang) bằng cách đặt số lượng khách hàng ban đầu là 0.
* Cách hoạt động: Hàm này chỉ đơn giản là khởi tạo danh sách khách hàng trong trạng thái ban đầu, với số lượng khách hàng hiện tại là 0.

2. Thêm khách hàng vào danh sách.

* Mục đích: Thêm một khách hàng vào danh sách khách hàng.
* Cách hoạt động:Nếu số lượng khách hàng hiện tại chưa đạt giới hạn (MAX\_CUSTOMERS), khách hàng sẽ được thêm vào cuối danh sách và số lượng khách hàng sẽ tăng lên.
* Nếu danh sách đã đầy, hàm sẽ thông báo rằng danh sách khách hàng đã đầy.



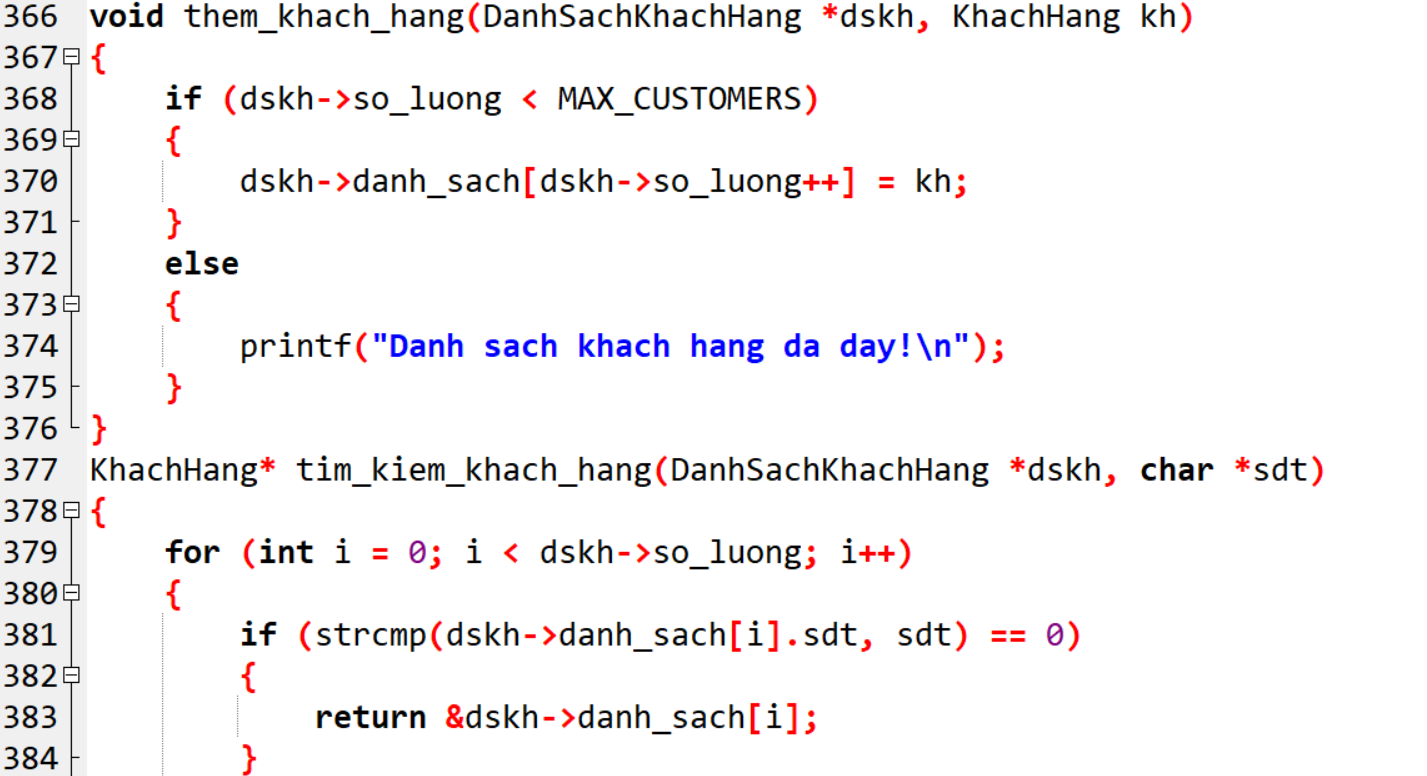
Hình 1.28 Hàm Khởi tạo danh sách khách hàng và thêm khách hàng vào danh sách.

3. Tìm kiếm khách hàng theo số điện thoại (SDT).

* Mục đích: Tìm kiếm một khách hàng trong danh sách khách hàng dựa trên số điện thoại (sdt).
* Cách hoạt động: Hàm duyệt qua tất cả khách hàng trong danh sách và so sánh số điện thoại của từng khách hàng với số điện thoại tìm kiếm. Nếu tìm thấy, trả về con trỏ đến khách hàng đó. Nếu không tìm thấy, trả về NULL.

4. Cập nhật lịch sử đặt sân của khách hàng.

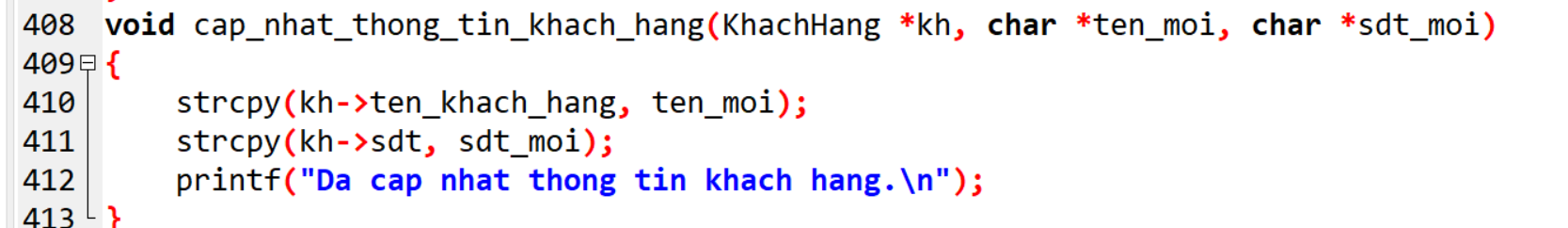
* Mục đích: Cập nhật lịch sử đặt sân của khách hàng.
* Cách hoạt động:
* Tạo một chuỗi mới mô tả thông tin về lịch đặt sân (số sân, giờ đặt và thời gian thuê).
* Thêm thông tin này vào chuỗi lịch sử đặt sân của khách hàng (lich\_su\_dat\_san).



Hình 1.29 Hàm tìm kiếm khách hàng theo số điện thoại và Cập nhật lịch sử đặt sân.

5. Cập nhật thông tin khách hàng.

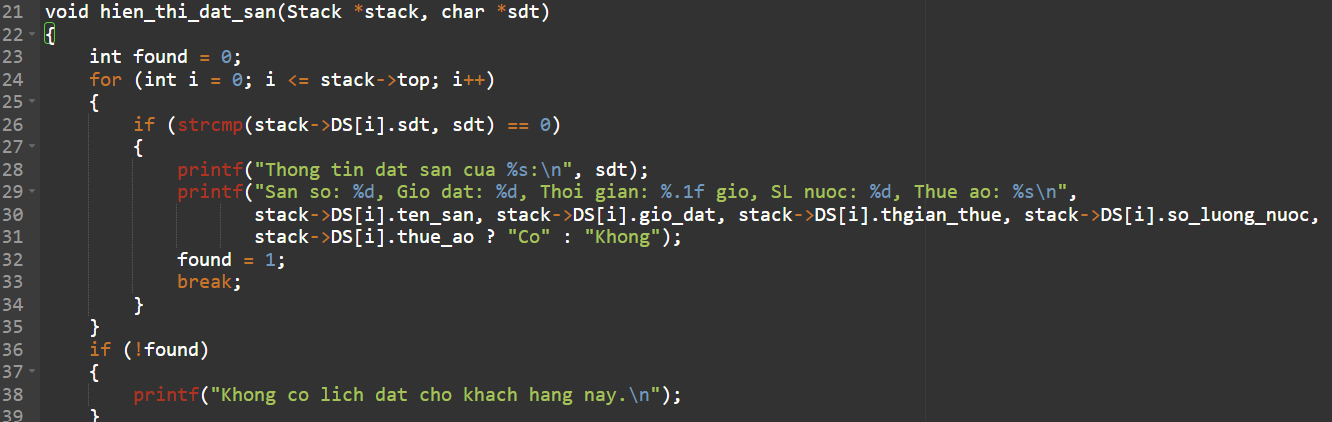
* Mục đích: Cập nhật thông tin của khách hàng, bao gồm tên và số điện thoại mới.
* Cách hoạt động: Hàm nhận tên mới và số điện thoại mới và cập nhật vào trường ten\_khach\_hang và sdt của khách hàng.



Hình 1.30 Hàm cập nhật thông tin khách hàng.

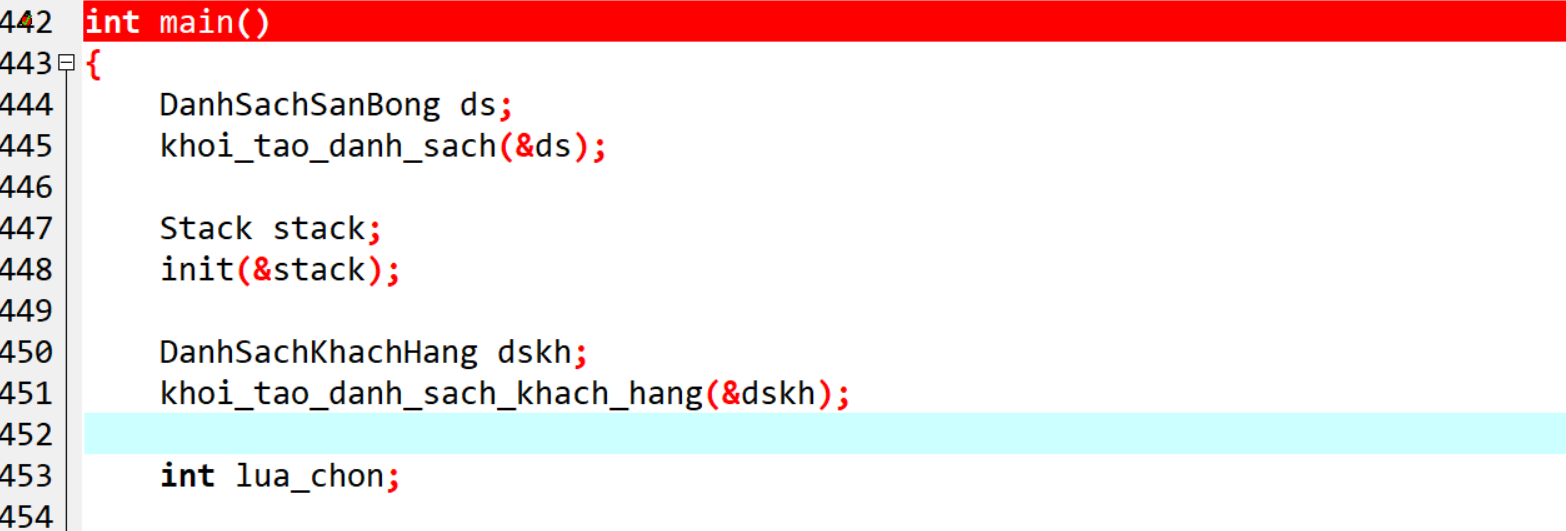
6. Hiển thị thông tin đặt sân của khách hàng.

* Mục đích: Hiển thị thông tin về các lịch đặt sân của khách hàng theo số điện thoại.
* Cách hoạt động:
* Hàm duyệt qua ngăn xếp (stack) để tìm các lịch đặt sân của khách hàng có số điện thoại khớp với sdt.
* Nếu tìm thấy, in thông tin chi tiết về lịch đặt sân. Nếu không tìm thấy, in thông báo rằng không có lịch đặt cho khách hang.



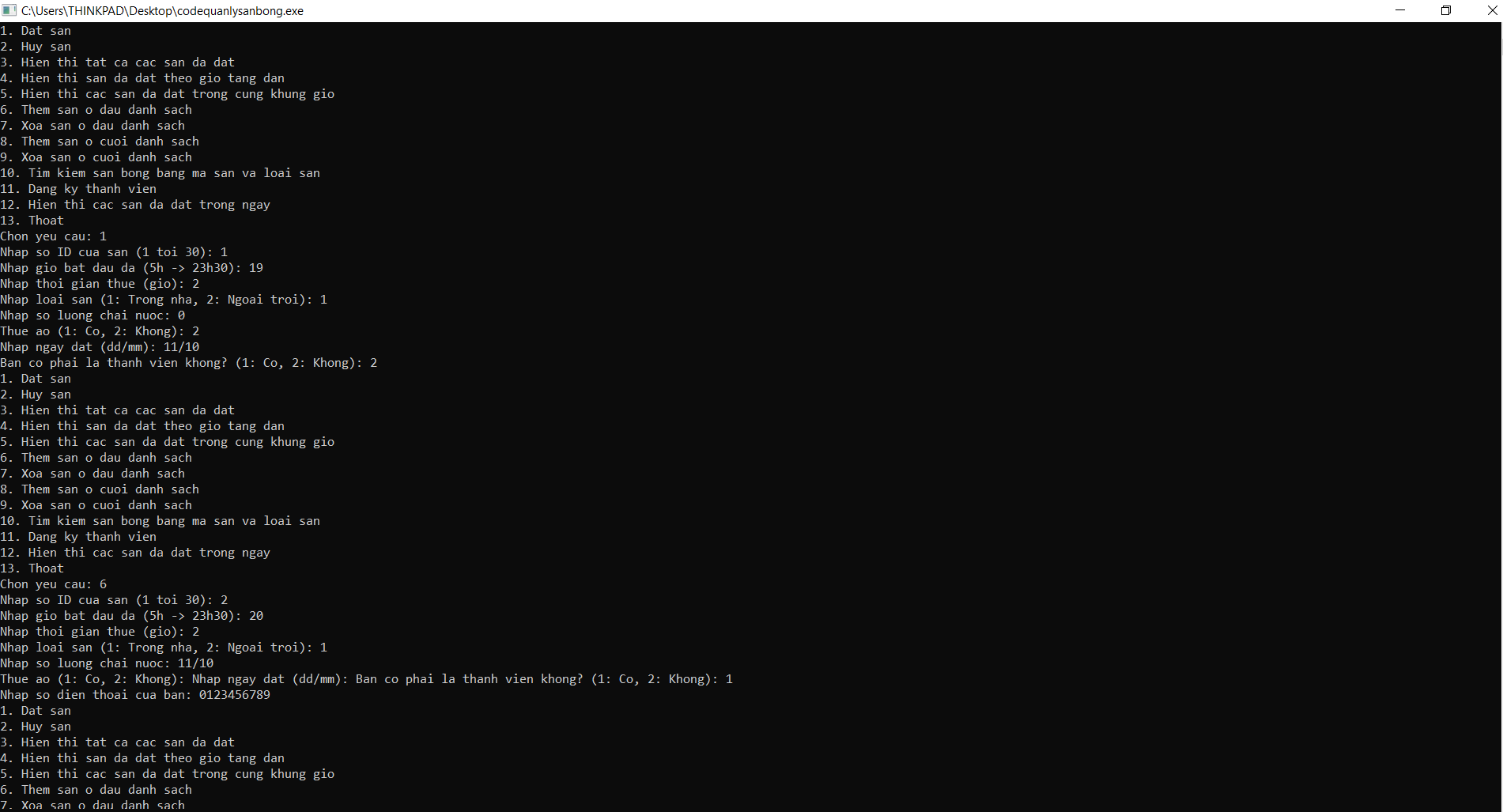
Hình 1.31 Hàm hiển thị thông tin đặt sân của khách hàng.

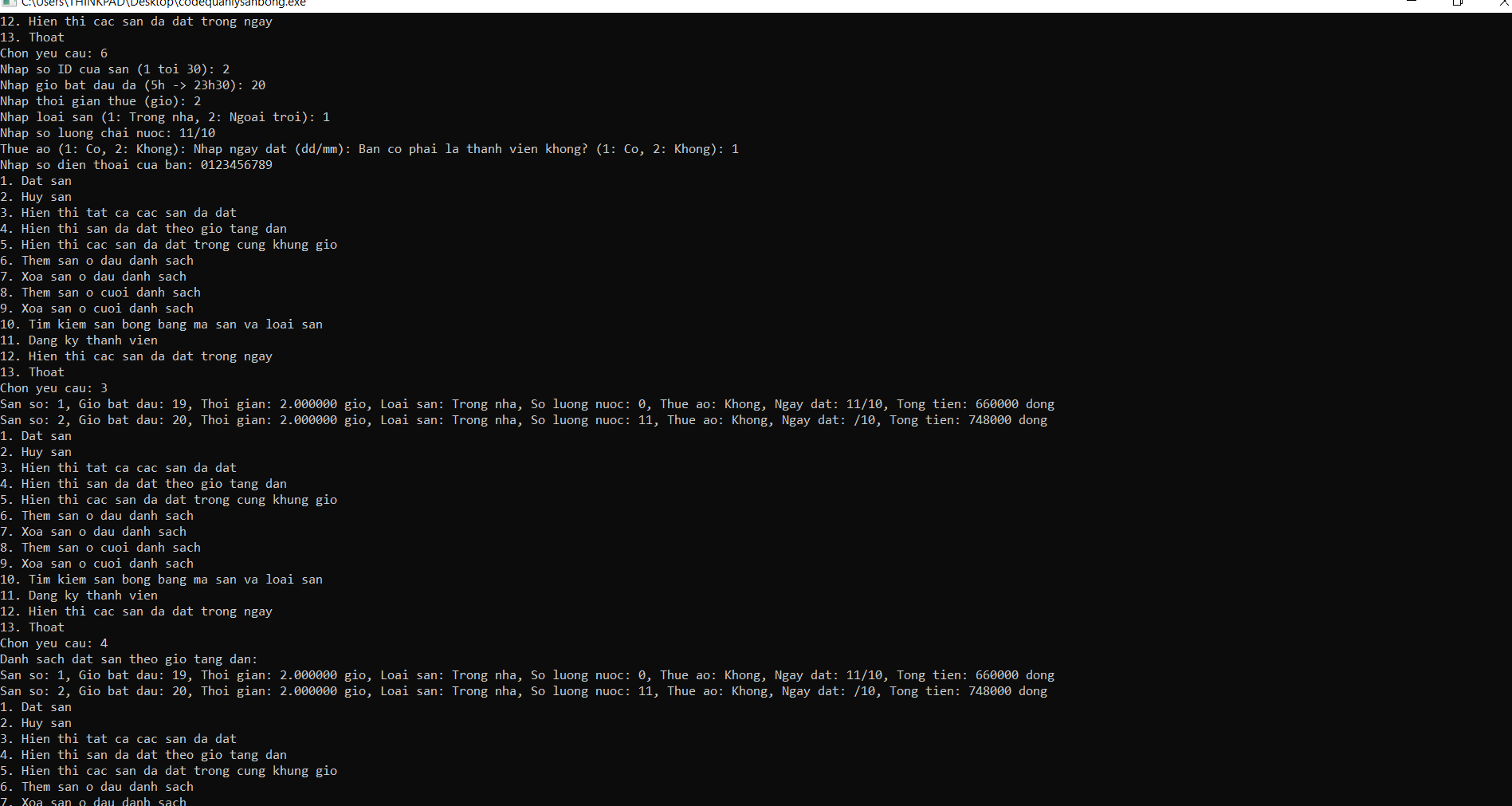
### 2.2.14 Hàm xử lý menu và tương tác với người dung.

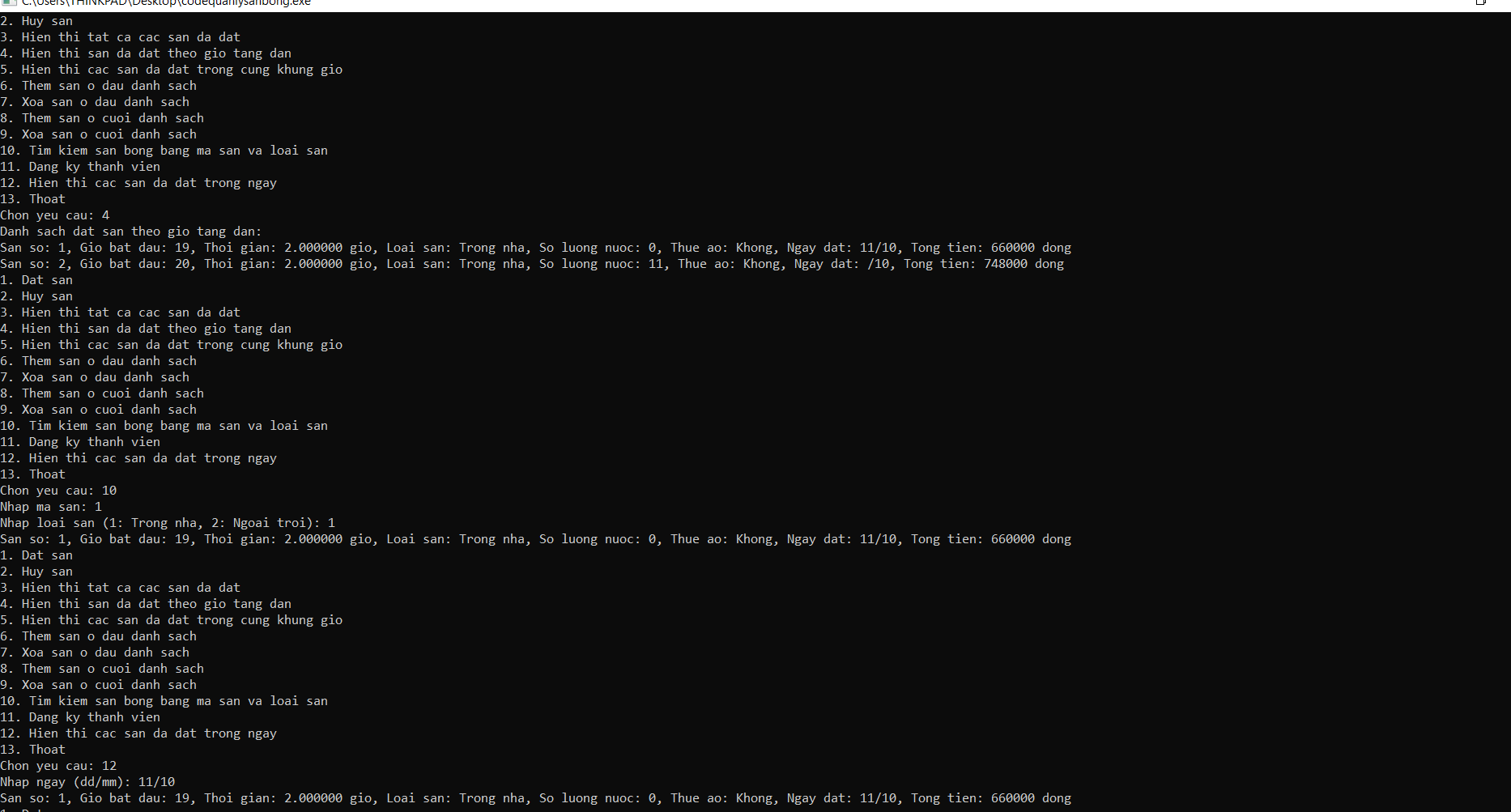


Hình 1.32 Hàm xử lí menu chính.

CHƯƠNG III: KẾT QUẢ CHƯƠNG TRÌNH.



****

****

**PHỤ LỤC**

Phụ lục 1: hướng dẫn cài đặt.

Phụ lục 2: hướng dẫn sử dụng.

# 

# TÀI LIỆU THAM KHẢO.

[1] "Cấu trúc dữ liệu và giải thuật" – Nguyễn Đức Hoàng.

[2] "C++ Data Structures and Algorithm Design" – Mehmet K. Dalkilic.

[3] "Cấu trúc dữ liệu và Giải thuật" – Nguyễn Tử Quảng.

[4] Nguyễn Trung Trực, Cấu trúc dữ liệu, NXB Thống

kê, 2002.

[5] Nguyễn Hồng Chương, Cấu trúc dữ liệu, cài đặt bằng

C, NXB TP.HCM, 2002.

[6] Phạm Văn Ất , Kỹ thuật lập trình C, NXB Thống kê,

2005.