****

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG TP.HCM**

**BỘ MÔN CẤU TRÚC DỮ LIỆU & GIẢI THUẬT**

****

**NHÓM 2**

**BÁO CÁO ĐỒ ÁN MÔN HỌC**

**ĐỀ** **TÀI : QUÁN LÝ DANH SÁCH SINH VIÊN**

**Môn học : Cấu trúc dữ liệu và giải thuật**

**Giáo viên hướng dẫn: ThS. Dương Thanh Thảo**

**Thực hiện bởi nhóm sinh viên, bao gồm:**

1. **Nguyễn Đông Din**  **N23DCCN146** **<Trưởng nhóm>**
2. **Nguyễn Lê Hoàng Học N23DCCN158** **<Thành viên>**
3. **Huỳnh Hoàng Khoa**  **N23DCCN166** **<Thành viên>**
4. **Nguyễn Hữu Duy**  **N23DCCN151** **<Thành viên>**

**TP.HCM 2025**

**Lời mở đầu**

Trong thời đại công nghệ thông tin phát triển mạnh mẽ, dữ liệu trở thành một yếu tố cốt lõi, ảnh hưởng đến hầu hết các lĩnh vực của đời sống xã hội. Việc tổ chức và quản lý dữ liệu hiệu quả là một yêu cầu thiết yếu, đặc biệt trong môi trường giáo dục – nơi khối lượng thông tin liên quan đến sinh viên ngày càng lớn. Quản lý thông tin sinh viên như mã số, họ tên, lớp học hay kết quả học tập một cách chính xác và khoa học sẽ góp phần nâng cao chất lượng đào tạo cũng như hiệu quả công tác quản lý.

Môn học *Cấu trúc dữ liệu và Giải thuật* đã trang bị cho chúng em nền tảng kiến thức quan trọng về cách lưu trữ, tổ chức và xử lý dữ liệu thông qua các cấu trúc như mảng, danh sách liên kết, cây nhị phân, hàng đợi, ngăn xếp và bảng băm,… Bên cạnh đó, môn học còn giúp hình thành tư duy thuật toán, khả năng phân tích bài toán và lựa chọn phương án xử lý tối ưu. Đây là những kiến thức cốt lõi trong việc xây dựng các hệ thống phần mềm có khả năng vận hành hiệu quả, linh hoạt và dễ mở rộng.

Từ những kiến thức đã học, nhóm chúng em thực hiện đề tài *“Quản lý sinh viên bằng các cấu trúc lưu trữ”* với mục tiêu ứng dụng các cấu trúc dữ liệu một cách phù hợp vào bài toán thực tế. Trong đề tài này, nhóm sẽ xây dựng một chương trình cho phép thực hiện các chức năng như thêm, xóa, sửa, tìm kiếm và sắp xếp thông tin sinh viên, đồng thời thử nghiệm nhiều cấu trúc lưu trữ khác nhau để đánh giá hiệu suất. Việc lựa chọn cấu trúc dữ liệu không chỉ dựa trên chức năng mà còn phụ thuộc vào mức độ tối ưu hóa cho từng thao tác xử lý.

Đây không chỉ là một bài tập lập trình thông thường mà còn là cơ hội để chúng em rèn luyện tư duy giải thuật, khả năng tổ chức hệ thống, và kỹ năng làm việc nhóm. Trong suốt quá trình thực hiện, chúng em đã phân chia công việc hợp lý, trao đổi thường xuyên và cùng nhau giải quyết các khó khăn phát sinh. Bài tiểu luận là kết quả của sự phối hợp, nỗ lực và tinh thần học hỏi của cả nhóm, đồng thời là bước khởi đầu để tiếp cận các dự án thực tế trong tương lai.

[Chương I: Bài tập 1. 3](#_Toc1558804241)

[1. Giới thiệu đề tài: 3](#_Toc1053668738)

[2. Cấu trúc lưu trữ: 4](#_Toc1363977272)

[a. Quản lí thông tin sinh viên: 4](#_Toc450965604)

[b. Mảng: 4](#_Toc1043967858)

[c. Danh sách liên kết đơn: 5](#_Toc1363549578)

[d. Danh sách liên kết vòng: 5](#_Toc745212099)

[e. Danh sách liên kết kép: 6](#_Toc1405925117)

[3. Thuật toán áp dụng: 7](#_Toc939895218)

[a. Sắp xếp danh sách sinh viên: 7](#_Toc235855586)

[i. Sắp xếp nổi bọt: 7](#_Toc1227578511)

[ii. Sắp xếp chọn: 8](#_Toc375147068)

[iii. Sắp xếp chèn: 9](#_Toc1107533340)

[iv. Sắp xếp vun đóng: 10](#_Toc2527722)

[v. Sắp xếp nhanh: 12](#_Toc1236809085)

[vi. Sắp xếp trộn: 13](#_Toc420399005)

[b. Tìm kiếm sinh viên: 15](#_Toc1066436650)

[i. Tìm kiếm vét cạn: 15](#_Toc1404830964)

[ii. Tìm kiếm nhị phân: 16](#_Toc2007882516)

[4. Kết quả kiểm thử: 17](#_Toc1219283363)

[Chương II: Bài tập 2. 20](#_Toc718475059)

[1. Giới thiệu đề tài: 20](#_Toc372576713)

[2. Cấu trúc lưu trữ: 21](#_Toc1560143031)

[3. Thuật toán áp dụng: 22](#_Toc1615692966)

[a. Thuật toán chèn node: 22](#_Toc424481214)

[b. Thuật toán xóa node: 23](#_Toc254944172)

[c. Duyệt cây bằng cách đệ quy : 25](#_Toc948548417)

[d. Duyệt cây bằng cách không đệ quy: 27](#_Toc1669976984)

[4. Kết quả kiểm thử: 29](#_Toc1324864357)

[Chương III: Một số yêu cầu cần cải thiện: 31](#_Toc1753033703)

# **Chương I: Bài tập 1.**

## **Giới thiệu đề tài:**

Việc quản lý thông tin sinh viên là một nhu cầu thiết yếu trong môi trường giáo dục. Tuy nhiên, khi số lượng sinh viên lớn, việc xử lý thủ công dễ gây sai sót và mất thời gian. Đề tài đồ án hướng đến xây dựng một chương trình quản lý danh sách sinh viên bằng ngôn ngữ C++, giúp người dùng thực hiện các thao tác như nhập dữ liệu, thêm, sửa, xóa, tìm kiếm, sắp xếp và thống kê thông tin một cách hiệu quả.

Mục tiêu của đề tài:

* Xây dựng chương trình C++ quản lý danh sách sinh viên với đầy đủ chức năng: nhập dữ liệu từ file, thêm/xóa/cập nhật sinh viên, tìm kiếm theo tiêu chí, sắp xếp theo thuật toán tùy chọn, và tính toán thống kê (điểm trung bình, sinh viên cao/thấp điểm nhất).
* Cho phép người dùng lựa chọn kiểu lưu trữ dữ liệu: mảng, danh sách liên kết đơn, liên kết kép, hoặc liên kết vòng.
* Hỗ trợ tìm kiếm tuyến tính và nhị phân, không phân biệt chữ hoa/thường, có chức năng đảo ngược họ tên khi xuất kết quả.
* Đo thời gian thực hiện các thuật toán sắp xếp và tìm kiếm để đánh giá hiệu suất.

## **Cấu trúc lưu trữ:**

### Quản lí thông tin sinh viên:

Cấu trúc lưu trữ Student được định nghĩa như sau:

* Là một lớp (class Student) chứa các thuộc tính riêng tư:
* string id: Mã định danh.
* string fname: Tên đệm.
* string lname: Họ.
* string \_class: Lớp học.
* float mark: Điểm số.
* Cung cấp các phương thức công khai:
* Constructor Student(string \_id, string \_fname, string \_lname, string \_class, float \_mark): Khởi tạo đối tượng với các giá trị đầu vào.
* Hàm operator<< để xuất thông tin sinh viên ra luồng đầu ra.
* Các getter: getId(), getFirstName(), getLastName(), getClass(), getMark() để truy xuất dữ liệu.
* Các setter: changeId(), changeFirstName(), changeLastName(), changeClass(), changeMark() để cập nhật dữ liệu.

Không sử dụng cấu trúc mảng hoặc vector trực tiếp trong lớp này, mà tập trung vào quản lý thông tin cá nhân của một sinh viên.

### Mảng:

Cấu trúc lưu trữ Array được định nghĩa như sau:

* Sử dụng một mảng tĩnh head[MAX] với kích thước cố định MAX = 1000 để lưu trữ các đối tượng Student.
* Bao gồm một biến index bắt đầu với giá trị bằng 0 để theo dõi số lượng phần tử hiện tại trong mảng.
* Cung cấp các phương thức quản lý như:
* insertStudent(Student x): Thêm một sinh viên vào mảng.
* deleteStudent(int x): Xóa một sinh viên khỏi mảng dựa trên chỉ số.
* Hỗ trợ lưu trữ động thông qua vector<Student> allStudents để quản lý danh sách sinh viên.
* Có các hàm sắp xếp (mergeSort, quickSort, bubbleSort, v.v.) .
* Tìm kiếm (search, bruteForceSearch, binarySearch) để xử lý dữ liệu trong mảng.

### Danh sách liên kết đơn:

Cấu trúc lưu trữ SinglyLinkedList được định nghĩa như sau:

* Sử dụng một cấu trúc node với các thành phần:
* Student data: Dữ liệu sinh viên.
* node\* next: Con trỏ đến nút tiếp theo trong danh sách liên kết đơn.
* Lớp SinglyLinkedList bao gồm:
* Con trỏ head để quản lý đầu danh sách.
* Phương thức makeMergeSort để sắp xếp danh sách bằng thuật toán merge sort.
* Phương thức merge để gộp hai danh sách đã sắp xếp.
* Phương thức getId để lấy mã định danh của sinh viên.
* Cung cấp các phương thức công khai:
* Constructor SinglyLinkedList(): Khởi tạo danh sách rỗng.
* insertStudent(Student data): Thêm một sinh viên vào danh sách.
* deleteStudent(string id): Xóa sinh viên dựa trên mã định danh.
* store(vector<Student>& Students): Lưu danh sách sinh viên vào vector.
* allStudents(): Trả về vector chứa tất cả sinh viên.
* search(string id): Tìm kiếm sinh viên theo mã định danh.
* bruteForceSearch và binarySearch: Tìm kiếm theo điều kiện với giá trị cụ thể.
* Các phương thức sắp xếp: bubbleSort, insertionSort, selectionSort, mergeSort, quickSort, heapSort.
* Sử dụng vector allStudents để lưu trữ toàn bộ danh sách sinh viên.

### Danh sách liên kết vòng:

Cấu trúc lưu trữ CircularLinkedList được định nghĩa như sau:

* Sử dụng một cấu trúc `node` với các thành phần:
* Student data: Dữ liệu sinh viên.
* node\* next: Con trỏ đến nút tiếp theo trong danh sách liên kết vòng.
* Lớp CircularLinkedList bao gồm:
* Con trỏ head để quản lý đầu danh sách.
* Phương thức makeMergeSort để sắp xếp danh sách bằng thuật toán merge sort.
* Phương thức merge để gộp hai danh sách đã sắp xếp.
* Phương thức getMid để lấy nút giữa danh sách.
* Cung cấp các phương thức công khai:
* Constructor CircularLinkedList(): Khởi tạo danh sách rỗng.
* makeNode(Student data): Tạo một nút mới.
* insertStudent(Student data): Thêm một sinh viên vào danh sách.
* deleteStudent(string id): Xóa sinh viên dựa trên mã định danh.
* store(vector<Student>& Students): Lưu danh sách sinh viên vào vector.
* allStudents(): Trả về vector chứa tất cả sinh viên.
* search(string id): Tìm kiếm sinh viên theo mã định danh.
* bruteForceSearch và binarySearch: Tìm kiếm theo điều kiện với giá trị cụ thể.
* Các phương thức sắp xếp: bubbleSort, insertionSort, selectionSort, mergeSort, quickSort, heapSort.

### Danh sách liên kết kép:

Cấu trúc lưu trữ DoublyLinkedList được định nghĩa như sau:

* Sử dụng một cấu trúc node với các thành phần:
* Student data: Dữ liệu sinh viên.
* node\* left: Con trỏ đến nút phía trước.
* node\* right: Con trỏ đến nút phía sau.
* Lớp DoublyLinkedList bao gồm:
* Con trỏ head để quản lý đầu danh sách.
* Phương thức length() để lấy độ dài danh sách.
* Phương thức makeMergeSort để sắp xếp danh sách bằng thuật toán merge sort.
* Phương thức merge để gộp hai danh sách đã sắp xếp.
* Phương thức getMid để lấy nút giữa danh sách.
* Phương thức getTail để lấy nút cuối danh sách.
* Phương thức partition và quickSortRecursive để thực hiện sắp xếp quick sort.
* Phương thức heapify để xây dựng heap.
* Phương thức swapData để hoán đổi dữ liệu giữa hai nút.
* Cung cấp các phương thức công khai:
* Constructor DoublyLinkedList(): Khởi tạo danh sách rỗng.
* makeNode(Student data): Tạo một nút mới.
* insertStudent(Student data): Thêm một sinh viên vào danh sách.
* deleteStudent(string id): Xóa sinh viên dựa trên mã định danh.
* store(vector<Student>& Students): Lưu danh sách sinh viên vào vector.
* allStudents(): Trả về vector chứa tất cả sinh viên.
* search(string id): Tìm kiếm sinh viên theo mã định danh.
* bruteForceSearch và binarySearch: Tìm kiếm theo điều kiện với giá trị cụ thể.
* Các phương thức sắp xếp: bubbleSort, insertionSort, selectionSort, mergeSort, quickSort, heapSort.

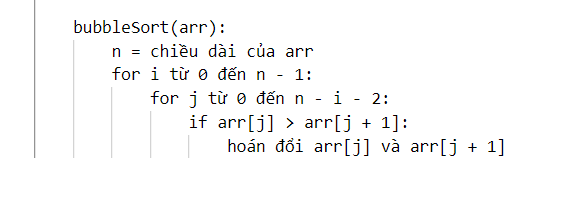
## **Thuật toán áp dụng:**

### Sắp xếp danh sách sinh viên:

#### Sắp xếp nổi bọt:

#### Ý tưởng:

* So sánh từng cặp phần tử liền kề, nếu chúng không đúng thứ tự, thì đổi chỗ.
* Sau mỗi vòng lặp (mỗi lần "bubble"), phần tử lớn nhất (hoặc nhỏ nhất) sẽ "nổi" dần về cuối mảng - giống như bong bóng nổi lên mặt nước, vì thế gọi là Bubble Sort.
* Lặp lại quá trình cho đến khi không còn cặp nào cần đổi chỗ.
* Mã giả:



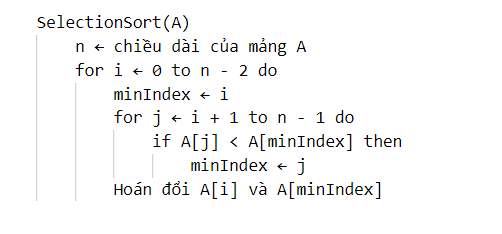
* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Tốt nhất | O( ) | Mảng đã sắp xếp. |
| Xấu nhất | O( ) | Mảng chưa sắp xếp. |

* Độ phức tạp không giản là O(n).
* Đặc điểm:
* Thuật toán đơn giản, dễ hiểu dễ cài đặt.
* Thuật toán không tối ưu trong trường hợp mảng có số lượng phần tử lớn.
* Bubble Sort thường được sử dụng để dạy học vì dễ hiểu, dễ minh họa, và giúp người học làm quen với khái niệm so sánh – hoán đổi – vòng lặp lồng nhau.
* Thường được sử dụng trong các phần mềm mô phỏng thuật toán để giúp sinh viên và học sinh hình dung rõ ràng quá trình sắp xếp.
* Dễ dàng minh họa bằng đồ họa, hoạt ảnh (animations) hoặc game học thuật.
* Khi kích thước dữ liệu nhỏ và hiệu suất không phải là vấn đề lớn, Bubble Sort có thể được sử dụng vì nó dễ cài đặt.

#### Sắp xếp chọn:

* Ý tưởng:
* Duyệt qua dãy, tại mỗi bước tìm phần tử nhỏ nhất (hoặc lớn nhất) trong phần chưa được sắp xếp.
* Hoán đổi phần tử nhỏ nhất này với phần tử ở vị trí đầu của phần chưa sắp xếp.
* Lặp lại cho đến khi dãy được sắp xếp hoàn toàn.
* Mả giả:



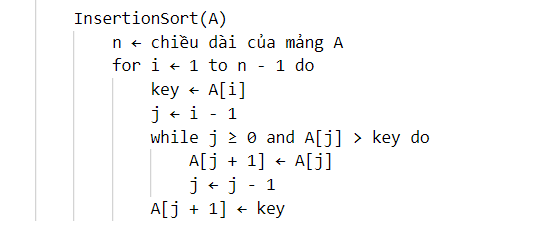
* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Tốt nhất | O(n²) | Dù đã sắp xếp, vẫn phải tìm phần tử nhỏ nhất mỗi vòng |
| Trung bình | O(n²) | Luôn có hai vòng lặp lồng nhau |
| Xấu nhất | O(n²) | Tương tự trường hợp trung bình |

* Độ phức tạp không gian: O(1), thuật toán sắp xếp tại chỗ.
* Đặc điểm:
* Dễ cài đặt và dễ hiểu.
* Thường dùng trong các bài tập lập trình, không yêu cầu hiệu suất cao.
* Không phù hợp với những bài toán có dữ liệu quá lớn.

#### Sắp xếp chèn:

* Ý tưởng:
* Xây dựng dãy đã sắp xếp bằng cách chèn từng phần tử vào đúng vị trí của nó.
* Giống như cách bạn sắp xếp bài trên tay khi chơi bài: từng lá được chèn vào đúng vị trí theo thứ tự đã có.
* Mã giả:



* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Tốt nhất | O(n) | Nếu mảng đã được sắp xếp |
| Trung bình | O(n²) |  |
| Xấu nhất | O(n²) | Nếu mảng sắp xếp ngược |

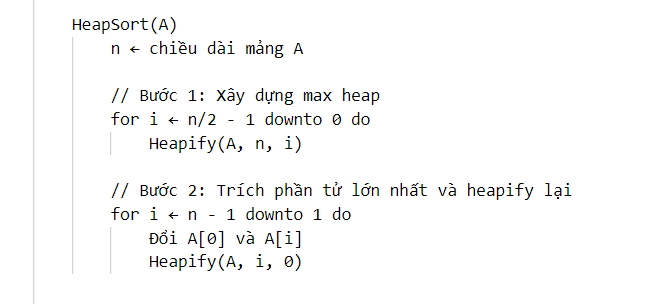
* Độ phức tạp không gian: O(1) - sắp xếp tại chỗ.
* Đặc điểm:
* Có tính ổn định vì không thay đổi các phần tử bằng nhau.
* Hiệu quả với dãy dữ liệu nhỏ hoặc gần như đã sắp xếp, không thích hợp với dữ liệu quá lớn.
* Dùng để sắp xếp một dãy số theo thứ tự tăng dần hoặc giảm dần.
* Thường dùng để minh họa tư duy thuật toán tuyến tính trong lập trình cơ bản.

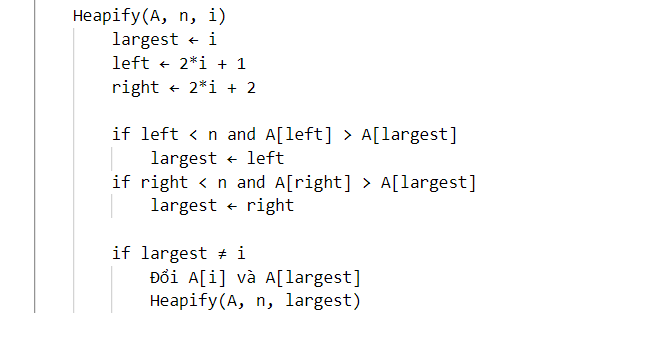
#### Sắp xếp vun đóng:

* Ý tưởng:
* Xây dựng một max-heap(cây nhị phân mà nút cha luôn lớn hơn hoặc bằng các nút con).
* Lặp lại:
* Đưa phần tử lớn nhất (gốc heap) về cuối mảng.
* Giảm kích thước heap đi 1.
* Heapify lại mảng để duy trì tính chất max-heap.
* Lặp đến khi toàn bộ mảng được sắp xếp.

Với min-heap có thể dùng để sắp xếp giảm dần.

* Mã giả:





* Độ phức tạp:
* Độ phức tạp thời gian: O(), trong mọi trường hợp.
* Độ phức tạp không gian: O(n).
* Đặc điểm:
* Không có tính ổn định có thể thay đổi vị trí của các phần tử bằng nhau.
* Hiệu quả cao với tập dữ liệu lớn, hiệu suất ổn định do không dùng đệ quy sâu và không yêu cầu bộ nhớ phụ, Heap Sort thích hợp để sắp xếp mảng lớn trong môi trường hạn chế tài nguyên.
* Không quan trọng với dữ liệu đầu vào đã sắp xếp, ngẫu nhhieen hay nghịch đảo, tốc độ vẫn gần như như nhau.
* Sắp xếp một dãy phần tử theo thứ tự tăng dần hoặc giảm dần.
* Sử dụng heap (đống) – một cấu trúc dữ liệu dạng cây nhị phân, giúp việc tìm phần tử lớn nhất hoặc nhỏ nhất rất nhanh.
* Thường dùng khi cần sắp xếp hiệu quả, ổn định về thời gian và không dùng thêm bộ nhớ.

#### Sắp xếp nhanh:

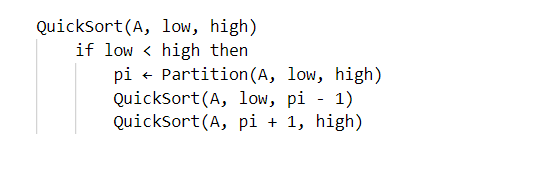
* Ý tưởng:

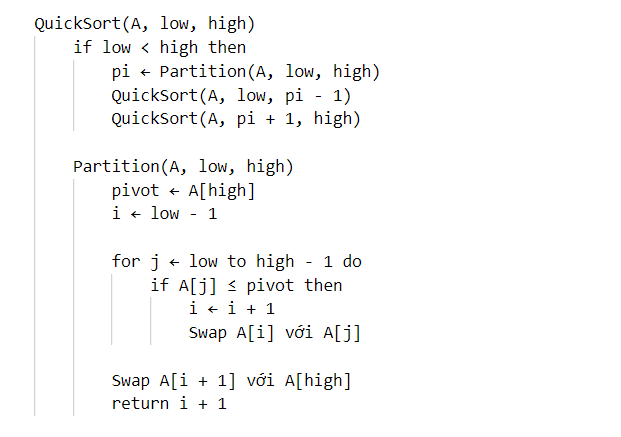
Thuật toán Quick Sort hoạt động theo nguyên lý divide and conquer (chia để trị). Ý tưởng cốt lõi được mô tả như sau:

* Chọn một phần tử gọi là "pivot" (phần tử chốt). Đây là phần tử làm chuẩn để chia mảng.
* Phân chia mảng thành hai phần:
* Phần bên trái chứa các phần tử nhỏ hơn hoặc bằng pivot.
* Phần bên phải chứa các phần tử lớn hơn pivot.
* Gọi đệ quy để tiếp tục sắp xếp hai phần con này.
* Sau khi đệ quy hoàn tất, các phần tử sẽ được sắp xếp đầy đủ theo thứ tự mong muốn.

Quá trình phân chia này được thực hiện tại chỗ, không cần tạo thêm mảng phụ (tiết kiệm bộ nhớ).

* Mã giả:





* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Tốt nhất | O(n log n) | Khi pivot chia mảng đều |
| Trung bình | O(n log n) | Phân chia tương đối cân bằng |
| Xấu nhất | O(n²) | Khi pivot là phần tử lớn nhất/nhỏ nhất |

* Độ phức tạp không gian: O(log n) do sử dụng đệ quy (ngăn xếp gọi hàm).
* Đặc điểm:
* Giải quyết bài toán sắp xếp bằng cách tối ưu hóa thời gian chạy, đặc biệt với các tập dữ liệu lớn.
* Là cơ sở cho nhiều hàm sắp xếp chuẩn trong thư viện ngôn ngữ lập trình (ví dụ: sort() trong Python dùng biến thể của Quick Sort).
* Tốc độ rất nhanh trong thực tế, thường nhanh hơn Merge Sort và Heap Sort.
* Không cần bộ nhớ phụ lớn.
* Dễ hiện thực bằng đệ quy hoặc không đệ quy.
* Không ổn định: nếu hai phần tử có giá trị bằng nhau, thứ tự của chúng có thể thay đổi.
* Nếu chọn pivot không tốt, có thể dẫn đến hiệu suất tệ (O(n²)).
* Dễ gây tràn ngăn xếp (stack overflow) nếu cài đặt đệ quy không đúng cách với mảng lớn.

#### Sắp xếp trộn:

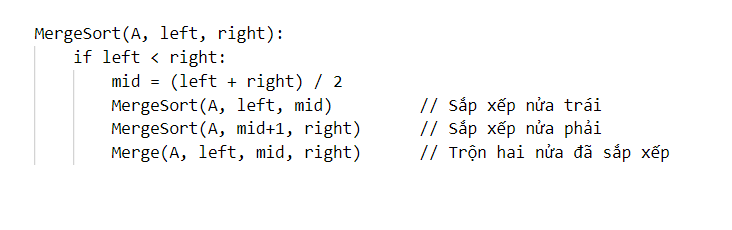
* Mục đích:

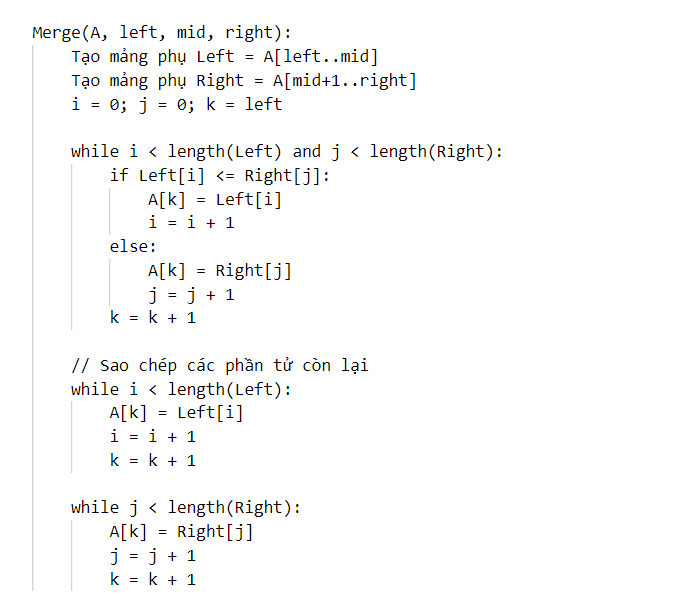
Thuật toán Merge Sort ra đời với mục tiêu:

* Sắp xếp dữ liệu một cách hiệu quả, đảm bảo hiệu suất ổn định và nhanh chóng ngay cả khi dữ liệu có kích thước rất lớn.
* Đặc biệt hữu ích trong các bài toán xử lý dữ liệu ngoài bộ nhớ, file lớn, hoặc danh sách liên kết – nơi các thuật toán như Quick Sort hay Insertion Sort không phù hợp.
* Là nền tảng cho các thuật toán lai (hybrid) như Timsort – được sử dụng trong ngôn ngữ lập trình Python, Java, Swift,...
* Được áp dụng nhiều trong các lĩnh vực như xử lý văn bản, thống kê, máy học, thuật toán chia để trị,...
* Ý tưởng:

Merge Sort là một ví dụ điển hình của thuật toán chia để trị (divide and conquer). Ý tưởng có thể tóm tắt như sau:

* Chia:
* Nếu mảng có nhiều hơn một phần tử, chia mảng thành hai nửa bằng nhau.
* Lặp lại việc chia cho đến khi mỗi mảng con chỉ còn 1 phần tử (tức là đã sắp xếp theo định nghĩa).
* Trị:
* Áp dụng đệ quy để sắp xếp riêng từng nửa.
* Trộn:
* Trộn hai nửa đã sắp xếp lại thành một mảng lớn hơn, cũng được sắp xếp.
* Việc trộn giữ nguyên thứ tự tương đối của các phần tử bằng nhau → Merge Sort là thuật toán ổn định.
* Mã giả





* Độ phức tạp:
* Độ phức tạp thời gian:

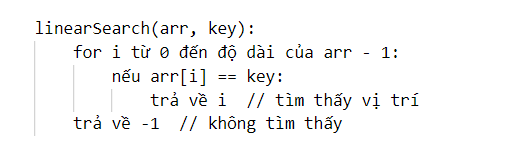
|  |  |
| --- | --- |
| Trường hợp | Độ phức tạp |
| Tốt nhất | O(n log n) |
| Trung bình | O(n log n) |
| Xấu nhất | O(n log n) |

* Độ phức tạp không gian:
* Không gian phụ: O(n).
* Merge Sort cần bộ nhớ tạm để lưu trữ các mảng con, vì vậy không phải là thuật toán sắp xếp tại chỗ (in-place).
* Nếu dùng trên danh sách liên kết (linked list), có thể giảm thiểu chi phí bộ nhớ phụ.
* Đặc điểm:
* Tính ổn định: Merge Sort giữ nguyên vị trí tương đối của các phần tử bằng nhau.
* Không sắp xếp tại chỗ: Không phải thuật toán tại chỗ (in-place), vì cần sử dụng thêm mảng phụ để thực hiện thao tác trộn.
* Không phụ thuộc vào trạng thái ban đầu của dữ liệu:
* Merge Sort không tối ưu hơn nếu dữ liệu đã sắp xếp.
* Khác với Insertion Sort (hiệu quả với mảng gần sắp xếp), Merge Sort luôn chạy với O(n log n).
* Sắp xếp dữ liệu một cách hiệu quả, đảm bảo hiệu suất ổn định và nhanh chóng ngay cả khi dữ liệu có kích thước rất lớn.
* Đặc biệt hữu ích trong các bài toán xử lý dữ liệu ngoài bộ nhớ, file lớn, hoặc danh sách liên kết – nơi các thuật toán như Quick Sort hay Insertion Sort không phù hợp.
* Là nền tảng cho các thuật toán lai (hybrid) như Timsort – được sử dụng trong ngôn ngữ lập trình Python, Java, Swift,...
* Được áp dụng nhiều trong các lĩnh vực như xử lý văn bản, thống kê, máy học, thuật toán chia để trị,...

### Tìm kiếm sinh viên:

#### Tìm kiếm vét cạn:

* Ý tưởng:
* Duyệt qua từng phần tử của mảng từ trái sang phải (hoặc đầu đến cuối).
* So sánh từng phần tử với giá trị cần tìm (key).
* Nếu tìm thấy, trả về vị trí của phần tử đó.
* Nếu duyệt hết mà không tìm thấy, trả về -1 (hoặc "không tồn tại").
* Mã giả:



* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Tốt nhất | O(1) | Phần tử nằm đầu mảng |
| Xấu nhất | O(n) |  |
| Trung bình | O(n/2) |  |

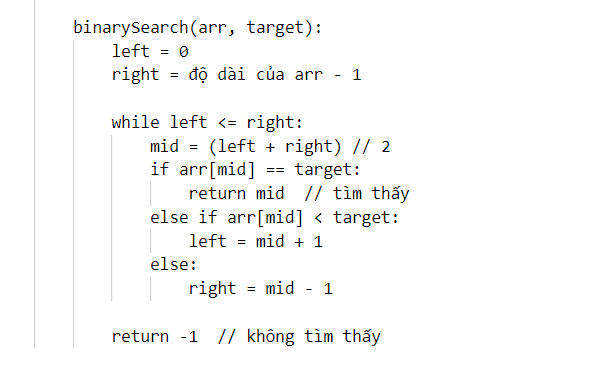
* Độ phức không gian: O(1).
* Đặc điểm:
* Không yêu cầu mảng sắp xếp.
* Dễ cài đặt, đơn giản, phù hợp cho người mới học lập trình, áp dụng linh hoạt.
* Không hiệu quả với dữ liệu lớn.
* Tìm kiếm vị trí của một phần tử trong một danh sách hoặc mảng.
* Áp dụng trong trường hợp:
* Dữ liệu chưa sắp xếp.
* Dữ liệu không quá lớn.
* Cần giải pháp đơn giản, nhanh chóng để triển khai.

#### Tìm kiếm nhị phân:

* Ý tưởng:
* So sánh phần tử cần tìm với phần tử ở giữa mảng.
* Nếu bằng => tìm thấy.
* Nếu nhỏ hơn => tìm ở nửa bên trái.
* Nếu lớn hơn => tìm ở nửa bên phải.
* Lặp lại quá trình này cho đến khi tìm được phần tử hoặc không còn gì để tìm.

Thuật toán này giống như việc tìm từ trong từ điển — bạn không tìm từ đầu đến cuối mà mở ở giữa và thu hẹp phạm vi.

* Mả giả:



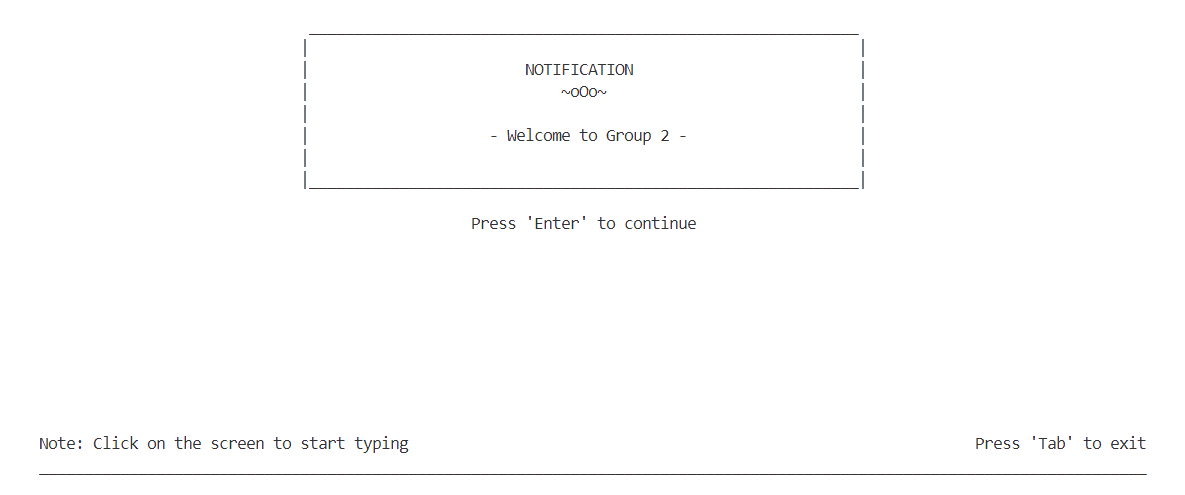
* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |
| --- | --- |
| Trường hợp | Độ phức tạp |
| Tốt nhất | O(n) |
| Xấu nhất | O(log n) |
| Trung bình | O(log n) |

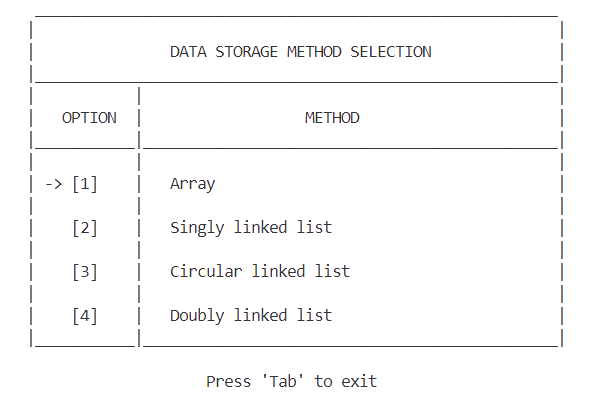
* Độ phức tạp không gian: O(1) vòng lặp hoặc O(n) dùng đệ quy.
* Đặc điểm:
* Chỉ áp dụng cho mảng đã được sắp xếp.
* Phổ biến và quan trọng.
* Tốc độ tìm kiếm rất nhanh,
* Không phải là một thuật toán ổn định.
* Tìm kiếm vị trí của một phần tử trong một mảng đã được sắp xếp (tăng hoặc giảm).
* Hiệu quả hơn nhiều so với tìm kiếm tuyến tính (Linear Search) khi làm việc với dữ liệu lớn đã sắp xếp.
* Độ phúc tạp thời gian trong trường hợp tốt nhất là O(1), trường hợp trung bình và xấu nhất là O(n).

## **Kết quả kiểm thử:**

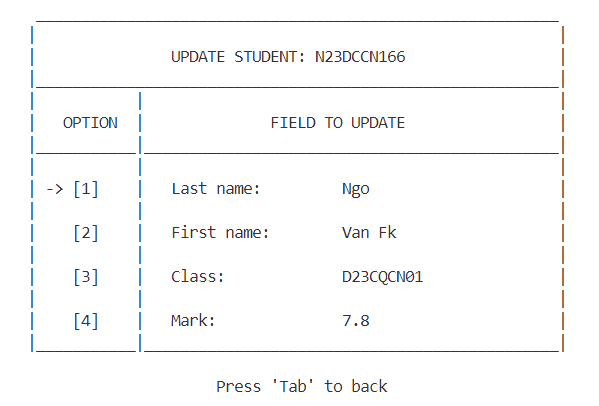
* Giao diện khởi động:



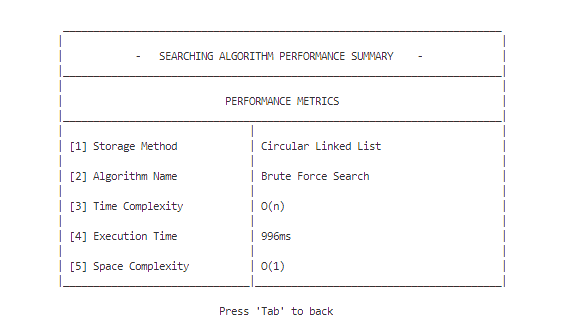
* Giao diện cấu trúc dữ liệu lưu trữ:



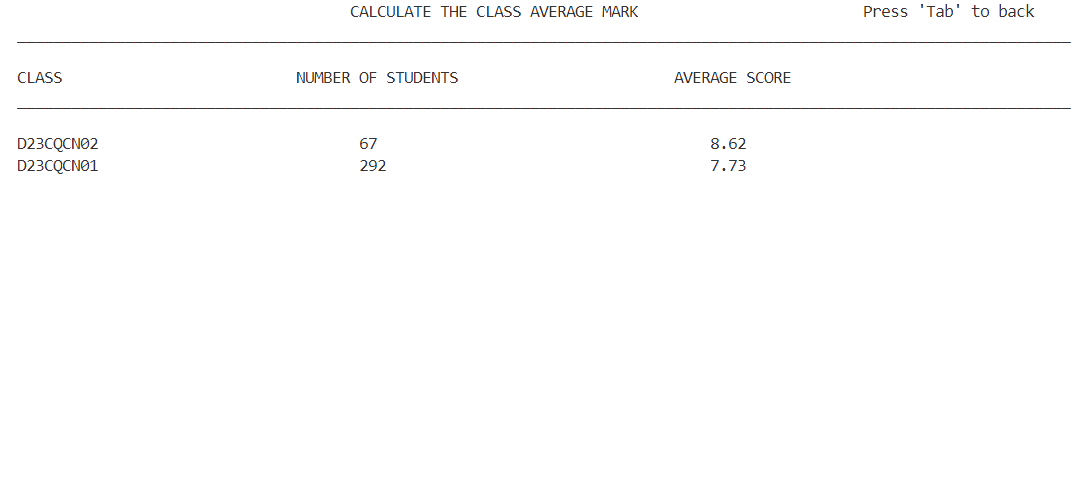
* Giao diện cập nhật sinh viên trong danh sách:



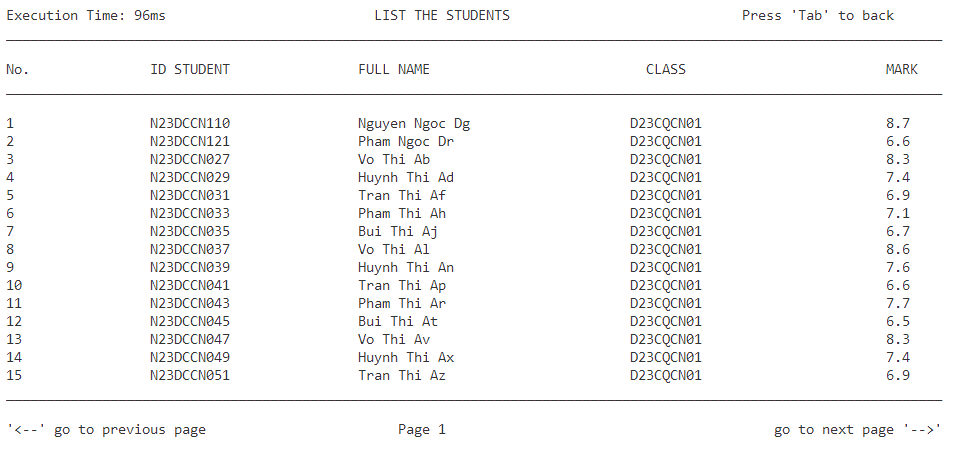
Bảng thống kê độ phức tạp tìm kiếm sinh viên theo kiểu vét cạn:



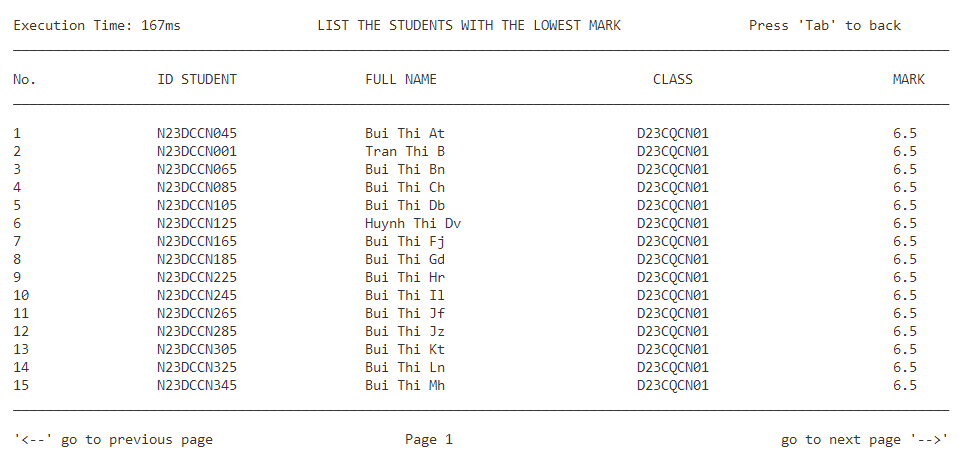
* Danh sách điểm trung bình của các lớp:



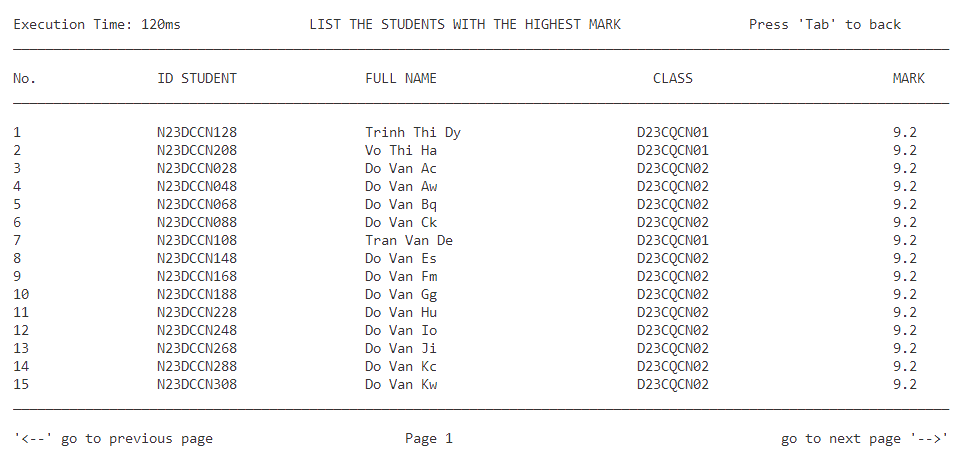
* Danh sách sinh viên ban đầu:



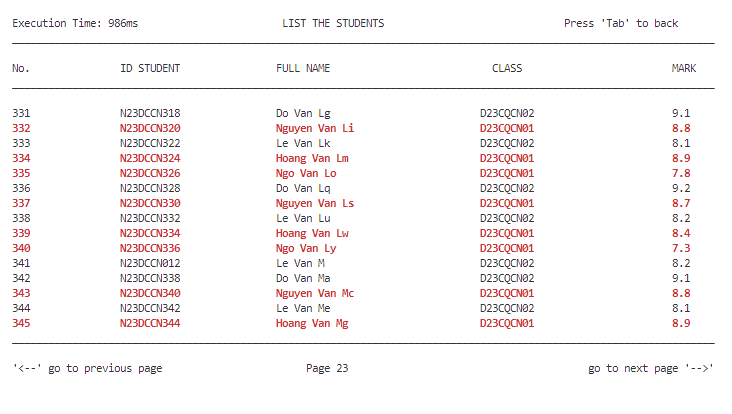
* Tìm sinh viên có điểm thấp nhất:



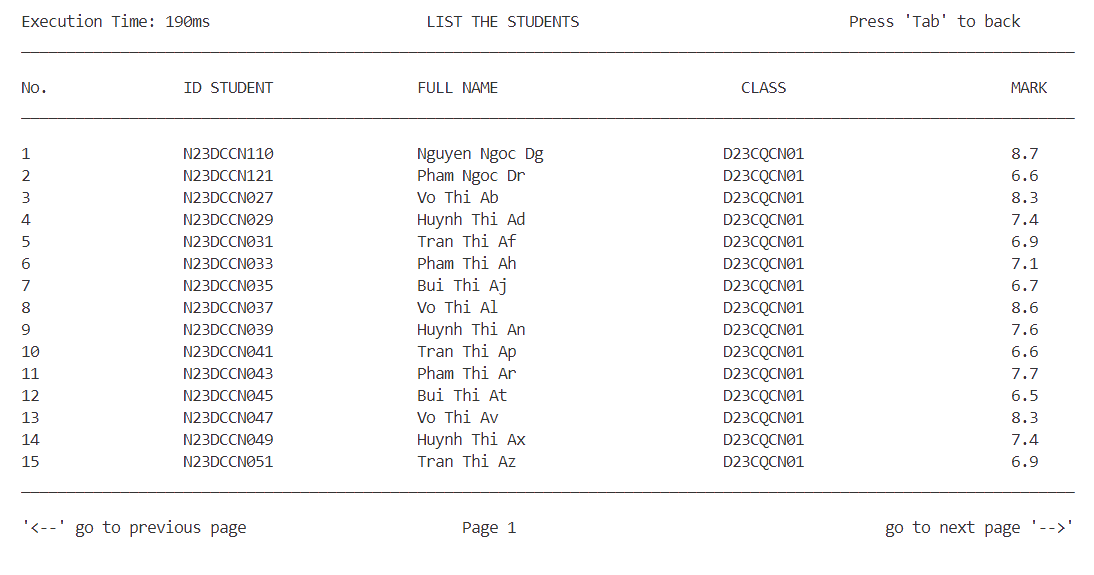
* Tìm sinh viên có điểm cao nhất:



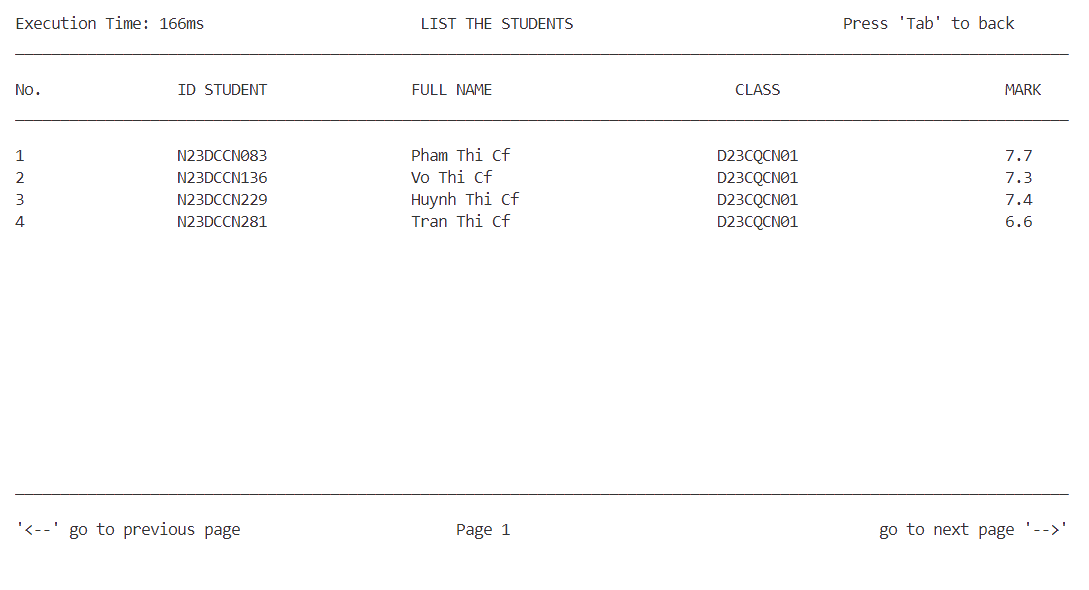
* Highlight sinh viên có lớp D23CQCN01:



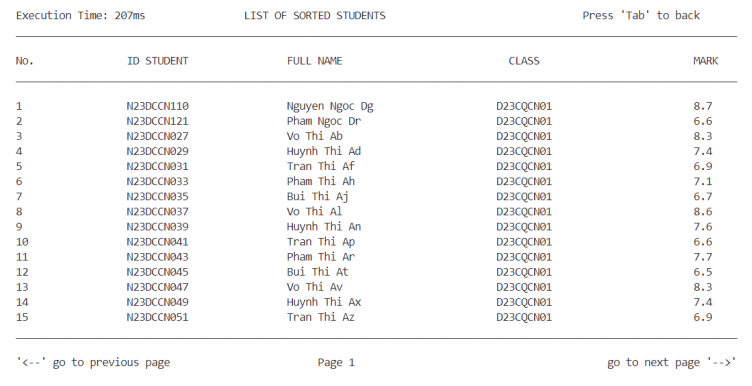
* Danh sách sinh viên tìm kiếm theo lớp theo thuật toán vét cạn:



* Danh sách sinh viên được tìm kiếm theo First name:



* Danh sách sinh viên được xếp theo lớp dùng thuật toán merge sort:



# **Chương II: Bài tập 2.**

## **Giới thiệu đề tài:**

Xây dựng chương trình quản lý danh sách sinh viên sử dụng cây nhị phân tìm kiếm BST. Dự án rèn luyện kỹ năng lập trình cấu trúc dữ liệu mà còn mô phỏng một hệ thống thực tế trong quản lý học tập. Qua đó, sinh viên nắm bắt được cách tổ chức, lưu trữ và xử lý dữ liệu hiệu quả một kỹ năng thiết yếu trong phát triển phần mềm quản lý, hệ thống thông tin giáo dục, và ứng dụng thực tiễn trong doanh nghiệp.

Mục đích của bài toán trong thực tế:

* Quản lý và tra cứu hiệu quả theo điểm số
* Trong các hệ thống quản lý sinh viên, việc lưu trữ và tra cứu danh sách sinh viên theo điểm số là một yêu cầu phổ biến (ví dụ: lọc những sinh viên đạt cùng một điểm để xét học bổng, học lại, hoặc thống kê).
* Cây nhị phân tìm kiếm giúp tìm kiếm nhanh theo điểm (O(log n) nếu cân bằng tốt), thay vì phải duyệt toàn bộ danh sách.
* Lưu trữ nhiều sinh viên có cùng điểm
* Mỗi node của cây không chỉ chứa một sinh viên, mà còn là danh sách sinh viên có cùng điểm, điều này phản ánh thực tế khi nhiều sinh viên đạt điểm giống nhau.
* Cập nhật động:
* Hệ thống cho phép người dùng thêm hoặc xóa sinh viên bất kỳ lúc nào. Cây BST giúp thực hiện các thao tác này hiệu quả hơn so với danh sách tuyến tính.
* Duyệt cây theo nhiều cách để phục vụ báo cáo/thống kê:
* Duyệt InOrder để hiển thị sinh viên theo thứ tự điểm tăng dần (phổ biến khi lập bảng điểm).
* Duyệt PreOrder hoặc PostOrder phục vụ cho mục đích đặc thù như sao lưu, tái cấu trúc cây, phân tích dữ liệu.
* Phân tích hiệu năng thuật toán:
* Việc đo thời gian thực thi giúp sinh viên hiểu và đánh giá được hiệu năng giữa các thuật toán duyệt cây đệ quy và không đệ quy, từ đó chọn giải pháp phù hợp cho các hệ thống thực tế.

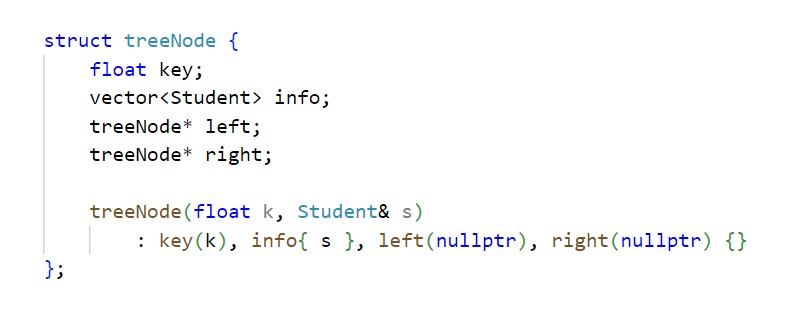
## **Cấu trúc lưu trữ:**

Trong bài toán này, xây dựng một cây nhị phân tìm kiếm (Binary Search Tree – BST) để quản lý danh sách sinh viên theo điểm số. Mỗi node trong cây đại diện cho một mức điểm cụ thể, và mỗi node sẽ lưu danh sách các sinh viên có cùng điểm đó.

Cấu trúc này giúp thực hiện hiệu quả các thao tác sau:

* Thêm sinh viên mới
* Xóa sinh viên hoặc cả node
* Duyệt cây theo thứ tự PreOrder, InOrder, PostOrder
* Tối ưu tìm kiếm sinh viên theo điểm

Mã code:



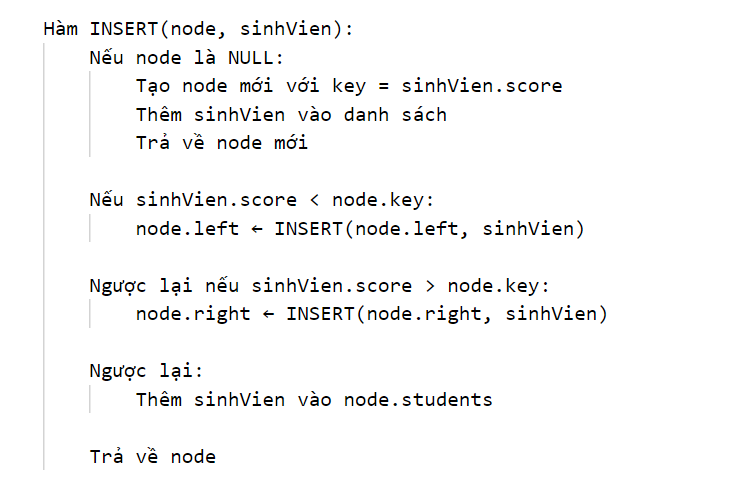
Ưu điểm khi chọn lưu trữ theo BST:

* BST có thời gian tìm kiếm, thêm, xóa là O(log n) nếu cây cân bằng.
* Lưu sinh viên theo điểm giúp tôi dễ dàng truy xuất, ví dụ:
* Tìm tất cả sinh viên đạt điểm 8.5.
* In danh sách sinh viên theo điểm tăng dần.
* Cấu trúc linh hoạt, dễ mở rộng, phù hợp với các hệ thống quản lý giáo dục thực tế.

## **Thuật toán áp dụng:**

### Thuật toán chèn node:

* Ý tưởng:
* So sánh điểm của sinh viên với key của các node.
* Di chuyển sang trái hoặc phải cho đến khi: Tìm thấy node có key = score → thêm sinh viên vào danh sách. Hoặc tìm thấy vị trí thích hợp để tạo node mới.
* Mã giả:



* Độ phức tạp:
* Độ phức tạp thời gian

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Xấu nhất | O(n) | Nếu cây bị lệch hoàn toàn |
| Trung bình | O(log n) | Nếu cây cân bằng |

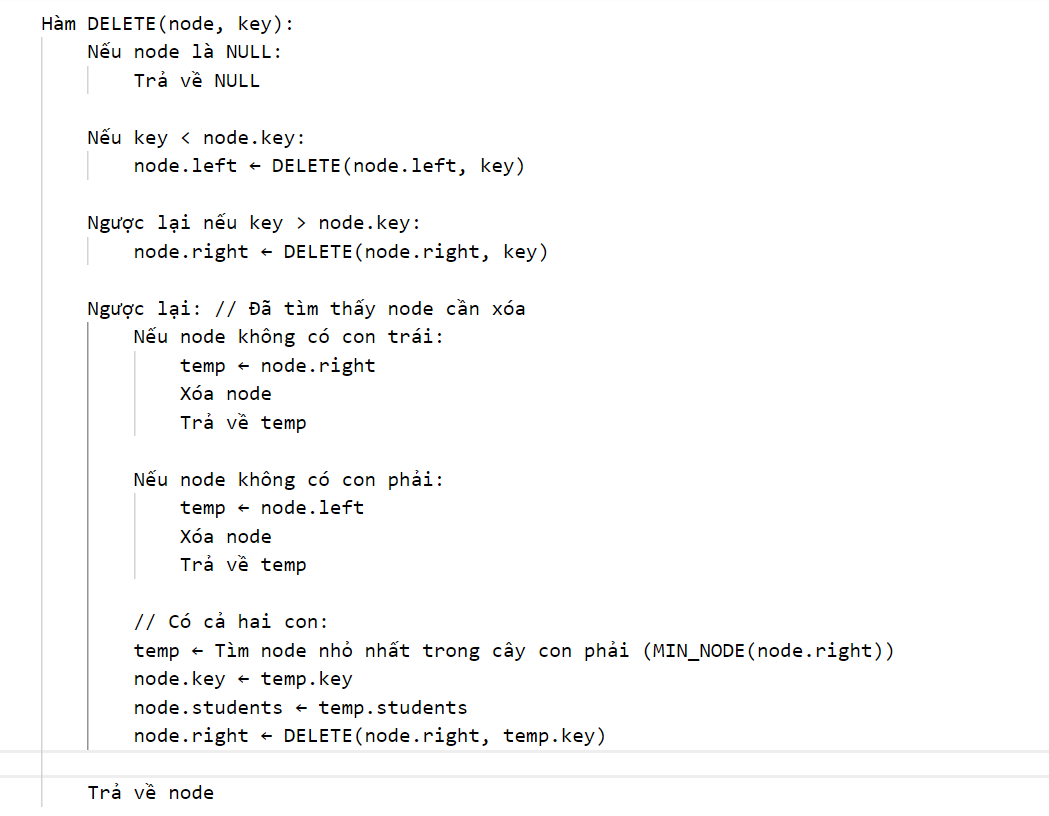
* Độ phức tạp không gian:
* O(1) cho mỗi lần insert *(chỉ cấp phát node)*
* Nếu dùng đệ quy, có thể cần thêm O(h) không gian ngăn xếp, với h là chiều cao của cây.
* Đặc điểm:
* Thuật toán chèn node giúp xây dựng và cập nhật cây BST một cách hiệu quả.
* Đảm bảo sinh viên được lưu trữ gọn gàng, dễ tra cứu theo điểm số.
* Kết hợp với các thao tác khác (xoá, duyệt), tạo thành hệ thống quản lý mạnh mẽ.
* Đảm bảo rằng sinh viên có cùng điểm nằm chung một node → tiết kiệm bộ nhớ và dễ quản lý.

### Thuật toán xóa node:

* Ý tưởng:

|  |  |  |
| --- | --- | --- |
| Trường hợp | Mô tả | Cách xử lý |
| Node là lá | Không có con trái và phải | Xóa trực tiếp |
| Node có một con | Có 1 con trái hoặc phải | Nối node cha với con của node bị xóa |
| Node có hai lá | Có cả hai con | Tìm node nhỏ nhất trong cây con phải (hoặc lớn nhất cây con trái), sao chép dữ liệu, sau đó xóa lại node thay thế |

* Mã giả:



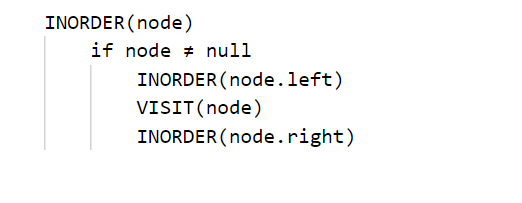
* Độ phức :
* Độ phức tạp thời gian

|  |  |  |
| --- | --- | --- |
| Trường hợp | Độ phức tạp | Ghi chú |
| Xấu nhất | O(n) | Nếu cây bị lệch hoàn toàn |
| Trung bình | O(log n) | Nếu cây cân bằng |

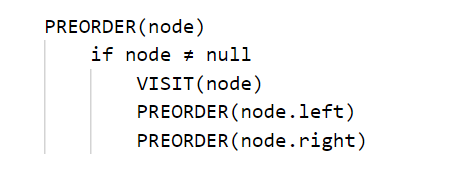
* Độ phức tạp không gian:
* O(1) cho mỗi lần insert *(chỉ cấp phát node)*
* Nếu dùng đệ quy, có thể cần thêm O(h) không gian ngăn xếp, với h là chiều cao của cây.
* Đặc điểm:
* Thuật toán xóa node trong BST là cần thiết để quản lý dữ liệu động: thêm/xóa sinh viên theo yêu cầu người dùng.
* Đảm bảo cây hoạt động ổn định, chính xác sau khi cập nhật dữ liệu.
* Xử lý linh hoạt 3 trường hợp giúp cây không bị hỏng cấu trúc.
* Kết hợp với thuật toán chèn và duyệt cây, tạo thành hệ thống BST hoàn chỉnh và hiệu quả.

### Duyệt cây bằng cách đệ quy :

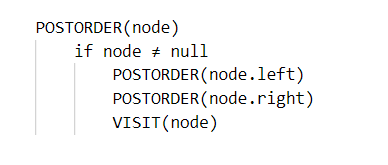
* Ý tưởng:
* Mỗi node trong cây được xử lý theo một thứ tự xác định (InOrder, PreOrder, PostOrder).
* Đệ quy được dùng để tự động lặp lại quá trình duyệt cho từng cây con (trái và phải) mà không cần viết vòng lặp tường minh.
* Mã giả:
* Inorder:



* PreOrder:



* PostOrder:



* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |
| --- | --- |
| Trường hợp | Độ phức tạp |
| In-Order | O(n) |
| Pre-Order | O(n) |
| Post-Order | O(n) |

* Độ phức tạp không gian: Tốt nhất là O(log n), tệ nhất là O(n).
* Đặc điểm:
* Inorder:
* Thứ tự: Cây con bên trái → Nút hiện tại → Cây con bên phải
* Trong cây nhị phân tìm kiếm (BST), duyệt theo cách này sẽ cho ra dãy giá trị tăng dần.
* Rất phổ biến khi làm việc với BST.
* Thường dùng khi muốn xử lý dữ liệu theo thứ tự sắp xếp.
* InOrder (LNR): In dữ liệu theo thứ tự tăng dần của điểm.
* PreOrder:
* Thứ tự: Nút hiện tại → Cây con bên trái → Cây con bên phải
* Xử lý nút gốc trước tiên.
* Phù hợp để sao chép cây, biểu diễn cây dưới dạng prefix (biểu thức tiền tố).
* Dễ dùng để xây lại cây nhị phân nếu kết hợp với In-order.
* PreOrder (NLR): Dùng để biểu diễn cấu trúc cây, hoặc sao lưu cây.
* PostOrder:
* Thứ tự: Cây con bên trái → Cây con bên phải → Nút hiện tại
* Xử lý các nút con trước, rồi mới đến nút gốc.
* Phù hợp khi muốn xóa cây, vì xóa từ lá đến gốc.
* Dùng trong biểu diễn cây dưới dạng hậu tố (postfix).
* PostOrder (LRN): Phục vụ xóa cây, tổng hợp dữ liệu từ dưới lên.
* Dễ cài đặt và hiểu:
* Cấu trúc cây phù hợp với cách viết đệ quy.
* Giúp sinh viên dễ học và hình dung luồng duyệt.
* Ứng dụng thực tế:
* Xuất danh sách sinh viên theo điểm.
* Tính tổng điểm, đếm số lượng sinh viên.
* Xử lý cây trong lưu trữ hoặc tính toán.

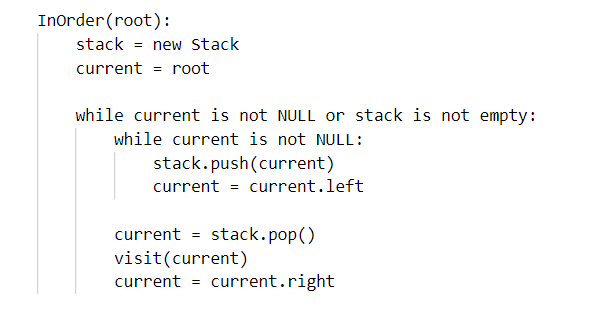
Lưu ý: Với cây lớn, nên cân nhắc duyệt không đệ quy để tránh tràn ngăn xếp.

### Duyệt cây bằng cách không đệ quy:

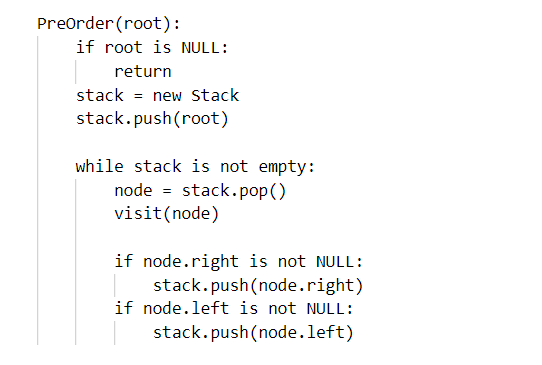
* Ý tưởng:

|  |  |  |
| --- | --- | --- |
| **Loại duyệt** | **Cấu trúc dùng thay đệ quy** | **Ý tưởng chính** |
| In-order | Stack | Đẩy trái đến cùng → xử lý nút → sang phải |
| Pre-order | Stack | Push gốc → xử lý → đẩy phải → đẩy trái |
| Post-order | 2 Stack hoặc Stack + đánh dấu | Đẩy trái và phải trước → xử lý sau cùng |

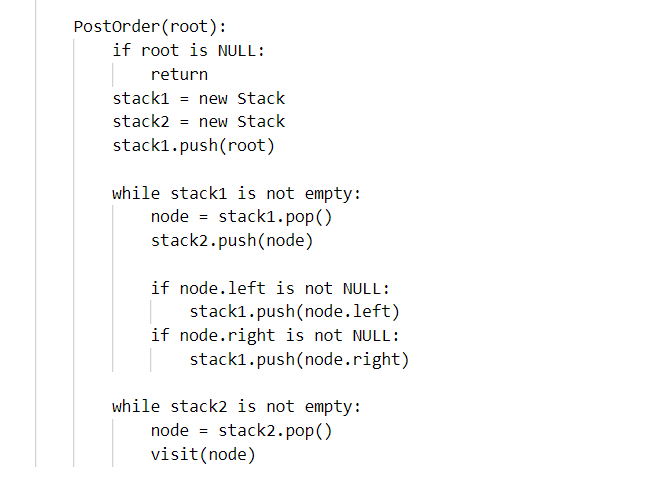
* Mã giả:
* InOrder:



* PreOrder:



* PostOrder:



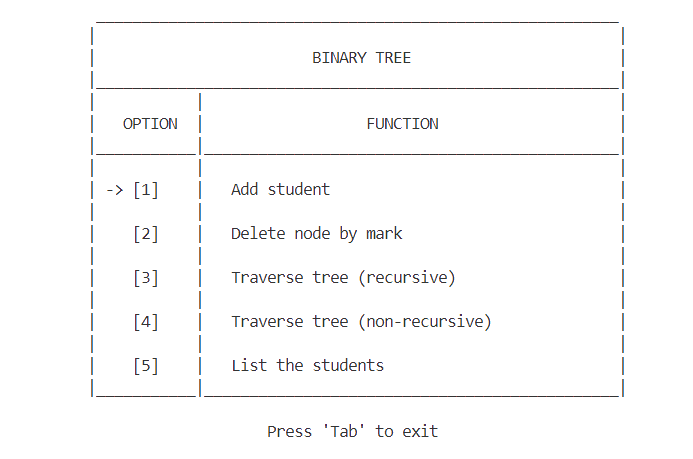
* Độ phức tạp:
* Độ phức tạp thời gian:

|  |  |
| --- | --- |
| Trường hợp | Độ phức tạp |
| In-Order | O(n) |
| Pre-Order | O(n) |
| Post-Order | O(n) |

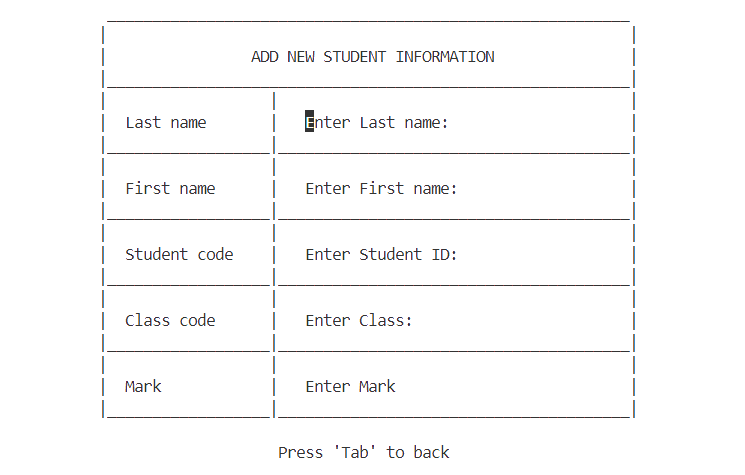
* Độ phức tạp không gian: Tốt nhất là O(log n), tệ nhất là O(n).
* Đặc điểm:
* InOrder:
* Thăm cây con bên trái trước, sau đó đến gốc và cuối cùng là cây con bên phải.
* Sử dụng ngăn xếp, đi sâu hết cây trái, sau đó lùi lại xử lý gốc, rồi đi sang phải.
* Trong cây nhị phân tìm kiếm (BST), duyệt in-order sẽ cho ra dãy giá trị tăng dần.
* Trích xuất các phần tử BST theo thứ tự sắp xếp, xử lý dữ liệu tuần tự, cần thiết khi xây lại cây từ các thứ tự duyệt.
* PreOrder:
* Thăm nút gốc đầu tiên, sau đó đến cây con trái và cuối cùng cây con phải.
* Sử dụng ngăn xếp để lưu các nút cần xử lý tiếp theo.
* Phải đẩy nút phải trước, trái sau vào ngăn xếp để đảm bảo duyệt đúng thứ tự.
* Dễ triển khai, ít thao tác hơn post-order.
* Sao chép cây, biểu diễn biểu thức tiền tố, ghi lại cấu trúc cây.
* PostOrder:
* Phức tạp nhất trong 3 loại duyệt vì nút gốc được xử lý sau cả hai cây con.
* Phải kiểm soát kỹ thứ tự xử lý để không duyệt gốc quá sớm.
* Xóa cây, Tính toán biểu thức hậu tố , duyệt cây theo chiều từ dưới lên.

## **Kết quả kiểm thử:**

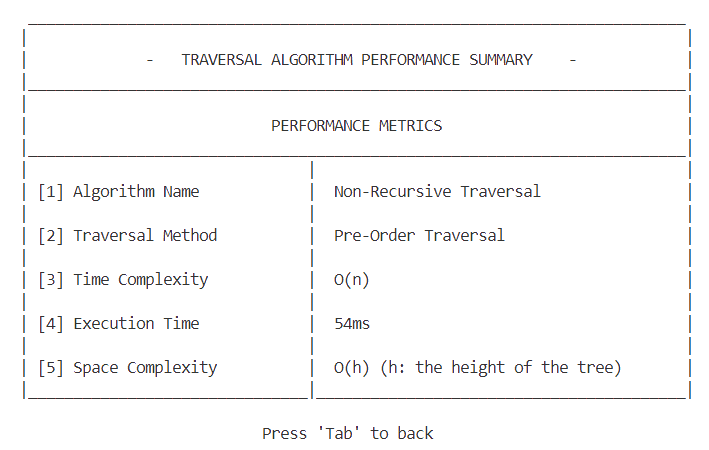
* Giao diện option:



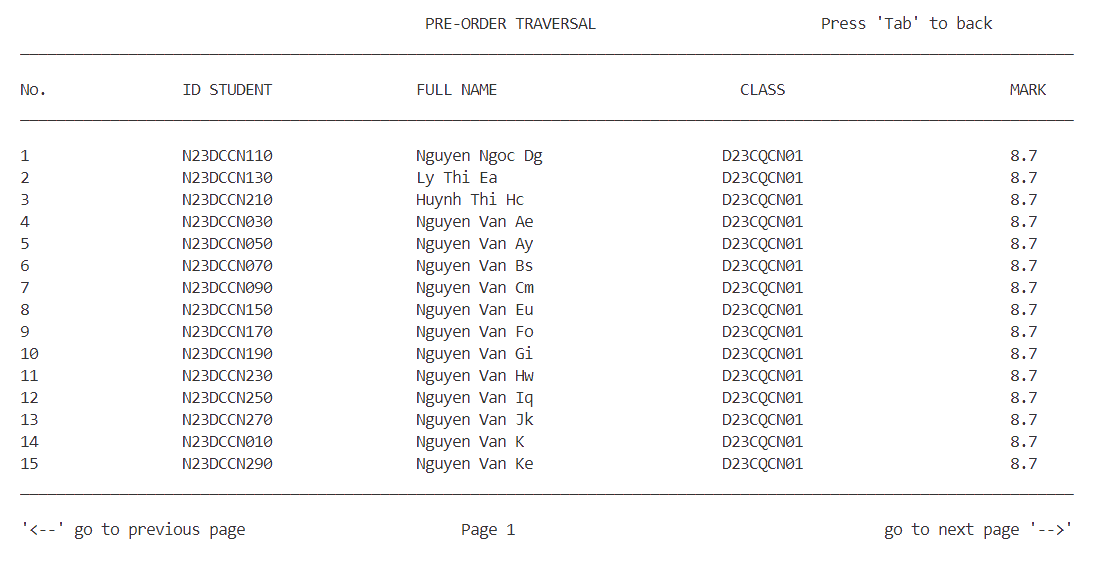
* Thêm sinh viên vào danh sách:



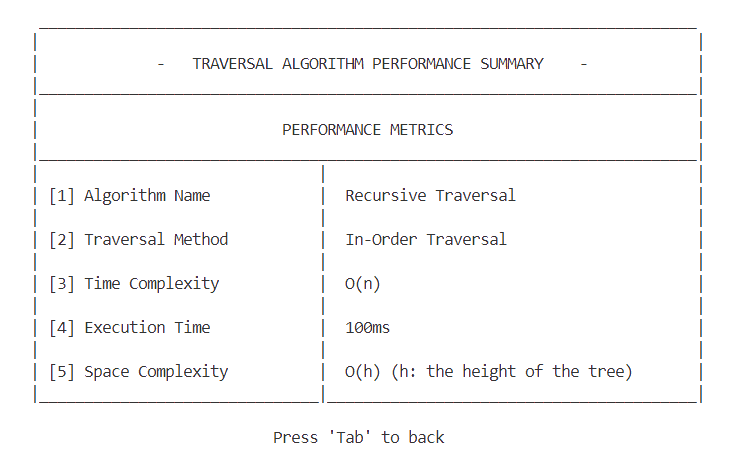
* Bảng thông kê thuật toán duyệt Pre-Order không đệ quy:



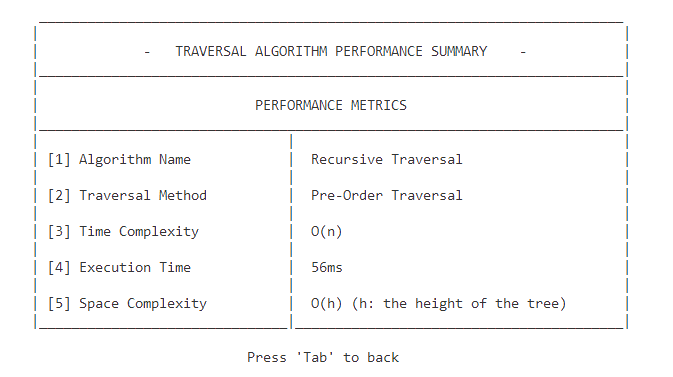
* Danh sách sinh viên được duyệt Pre-Order không đệ quy:



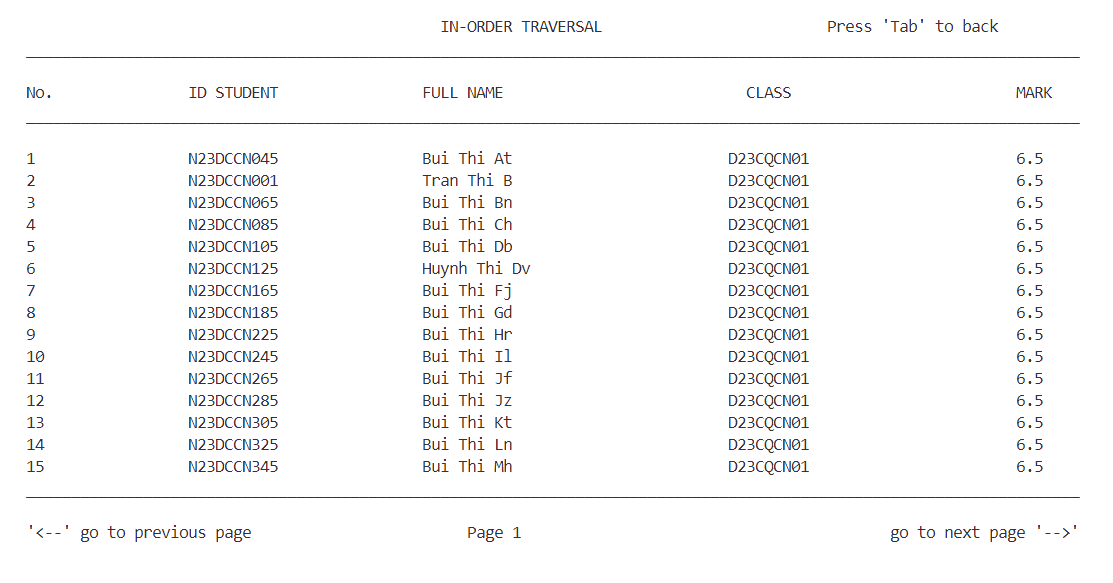
* Bảng thông kê thuật toán duyệt In-Order đệ quy:



* Bảng thông kê thuật toán duyệt Pre-Order đệ quy:



* Danh sách sinh viên được duyệt In-Order không đệ quy:



**Tài liệu tham khảo:**

1. John Bullinaria, Data Structures and Algorithms, School of Computer Science University of Birmingham, 2019.

2. Nguyễn Mạnh Sơn, Bài tập Cấu trúc dữ liệu và giải thuật, Học viện công nghệ bưu chính viễn thông, 2022.

3. Nguyễn Duy Phương và Nguyễn Mạnh Sơn, Cấu trúc dữ liệu và giải thuật, Học viện công nghệ bưu chính viễn thông, 2020.