### Cấu trúc dữ liệu mảng là gì ?

**Mảng (Array)** là một trong các cấu trúc dữ liệu cũ và quan trọng nhất. Mảng có thể lưu giữ một số phần tử cố định và các phần tử này nền có cùng kiểu. Hầu hết các cấu trúc dữ liệu đều sử dụng mảng để triển khai giải thuật. Dưới đây là các khái niệm quan trọng liên quan tới Mảng.

* **Phần tử**: Mỗi mục được lưu giữ trong một mảng được gọi là một phần tử.
* **Chỉ mục (Index)**: Mỗi vị trí của một phần tử trong một mảng có một chỉ mục số được sử dụng để nhận diện phần tử.

Mảng gồm các bản ghi có kiểu giống nhau, có kích thước cố định, mỗi phần tử được xác định bởi chỉ số.

Mảng là cấu trúc dữ liệu được cấp phát liên tục cơ bản.

Ưu điểm của mảng :

* Truy câp phần tử vơi thời gian hằng số **O(1).**
* Sử dụng bộ nhớ hiệu quả.
* Tính cục bộ về bộ nhớ.

Nhược điểm của mảng:

* Không thể thay đổi kích thước của mảng khi chương trình dang thực hiện.

**Mảng động:**

**Mảng động (dynamic aray)** : cấp phát bộ nhớ cho mảng một cách động trong quá trình chạy chương trình trong C là malloc và calloc, trong C++ là new.

Sử dụng mảng động ta bắt đầu với mảng có 1 phần tử, khi số lượng phần tử vượt qua khả năng của ảng thì ta gấp đôi kích thước mảng cũ và copy phần tử mảng cũ vào nửa đầu của mảng mới.

**Ưu điểm:**

* Tránh lãng phí bộ nhớ khi phải khai báo mảng có kích thước lớn ngay từ đầu.

**Nhược điểm:**

* Phải thực hiện them thao tác copy phần tử mỗi khi thay đổi kích thước.
* Một số thời gian thực hiện thao tác không còn là hằng số nữa.

### Biểu diễn Cấu trúc dữ liệu mảng.

Mảng có thể được khai báo theo nhiều cách đa dạng trong các ngôn ngữ lập trình. Để minh họa, chúng ta sử dụng phép khai báo mảng trong ngôn ngữ C:

**Diagram

Description automatically generated**

Hình minh họa phần tử và chỉ số:

**Table

Description automatically generated**

Dưới đây là một số điểm cần ghi nhớ về cấu trúc dữ liệu mảng:

* Chỉ mục bắt đầu với 0.
* Độ dài mảng là 5, nghĩa là mảng có thể lưu giữ 5 phần tử.
* Mỗi phần tử đều có thể được truy cập thông qua chỉ số của phần tử đó. Ví dụ, chúng ta có thể lấy giá trị của phần tử tại chỉ số 3 là -9.

Phép toán cơ bản được hỗ trợ bởi mảng:

Dưới đây là các hoạt động cơ bản được hỗ trợ bởi một mảng:

* **Duyệt**: In tất cả các phần tử mảng theo cách in từng phần tử một.
* **Chèn**: Thêm một phần tử vào mảng tại chỉ mục đã cho.
* **Xóa**: Xóa một phần tử từ mảng tại chỉ mục đã cho.
* **Tìm kiếm**: Tìm kiếm một phần tử bởi sử dụng chỉ mục hay bởi giá trị.
* **Cập nhật**: Cập nhật giá trị một phần tử tại chỉ mục nào đó.

**Giới thiệu về cấu trúc stack.**

**Stack** là một loại container adaptor, được thiết kế để hoạt động theo kiểu **LIFO** (**Last - in first - out**) (vào sau ra trước), tức là một kiểu danh sách mà việc bổ sung và loại bỏ một phần tử được thực hiển ở cuối danh sách. Vị trí cuối cùng của **stack** gọi là đỉnh (**top**) của ngăn xếp.

**Diagram

Description automatically generated**

**Stack** giống như việc giáo viên kiểm tra vở bài tập của học sinh vậy, ai nộp sau cùng thì vở bài tập của người đó sẽ được giáo viên kiểm tra đầu tiên, đương nhiên người nộp vợ đầu tiên sẽ được kiểm tra cuối cùng.

**Stack** có các hàm sau (ví dụ cho C++):

* **size** : trả về kích thước hiện tại của **stack**. ĐPT O(1).
* **empty** : true **stack** nếu rỗng, và ngược lại. ĐPT O(1).
* **push** : đẩy phần tử vào **stack**. ĐPT O(1).
* **pop** : loại bỏ phẩn tử ở đỉnh của **stack**. ĐPT O(1).
* **top** : truy cập tới phần tử ở đỉnh **stack**. ĐPT O(1).

**Giới thiệu về queue:**

**Queue(hàng đợi)** là một loại container, được thiết kế để hoạt động theo kiểu **FIFO** (**First- in first – out**) (vào trước ra trước), tức là một kiểu danh sách mà việc bổ sung được thực hiển ở cuối danh sách và loại bỏ ở đầu danh sách.

Trong **queue**, có hai vị trí quan trọng là vị trí đầu danh sách **(front)**, nơi phần tử được lấy ra, và vị trí cuối danh sách **(back)**, nơi phần tử cuối cùng được thêm vào.

Shape, rectangle

Description automatically generated

Khai báo (C++)

#include<queue>

Các phương thức (ngôn ngữ C++):

|  |  |
| --- | --- |
| **Capacity** | |
| size() | Trả về số lượng phần tử của queue |
| empty() | Trả về true(1) nếu queue rỗng, ngược lại là false (0) |
| **Element access:** |  |
| front() | Truy xuất phần tử ở đầu queue (phần tử đầu tiên được thêm vào) |
| back() | Truy xuất phần tử ở cuối queue (phần tử cuối cùng được thêm vào) |
| **Modifier:** | |
| push (const x) | Thêm phần tử có giá trị x vào cuối queue. Kích thước queue tăng thêm 1. |
| pop () | Loại bỏ phần tử ở đầu queue. Kích thước queue giảm đi 1. |

**Giới thiệu danh sách liên kết.**

**Danh sách liên kết đơn(Single linked list)** là ví dụ tốt nhất và đơn giản nhất về cấu trúc dữ liệu động sử dụng con trỏ để cài đặt. Do đó, kiến thức con trỏ là rất quan trọng để hiểu cách danh sách liên kết hoạt động, vì vậy nếu bạn chưa có kiến thức về con trỏ thì bạn nên học về con trỏ trước. Bạn cũng cần hiểu một chút về cấp phát bộ nhớ động. Để đơn giản và dễ hiểu, phần nội dung cài đặt danh sách liên kết của bài này sẽ chỉ trình bày về danh sách liên kết đơn.

Danh sách liên kết đơn là một tập hợp các Node được phân bố động, được sắp xếp theo cách sao cho mỗi Node chứa một giá trị (**Data**) và một con trỏ (**Next**)*.*Con trỏ sẽ trỏ đến phần tử kế tiếp của danh sách liên kết đó. Nếu con trỏ mà trỏ tới **NULL**, nghĩa là đó là phần tử cuối cùng của **linked list**.

Hình ảnh mình họa một **node**:

Graphical user interface, diagram

Description automatically generated

Hình ảnh mình họa một **listLinker**:

Diagram

Description automatically generated

So sánh mảng và danh sách liên kết:

|  |  |  |
| --- | --- | --- |
| **Nội dung** | **Mảng** | **Danh sách liên kết** |
| Kích thước | * Kích thước cố định * Cần chỉ rõ kích thước trong khi khai báo | * *Kích thước thay đổi trong quá trình thêm/ xóa phần tử* * *Kích thước tối đa phụ thuộc vào bộ nhớ* |
| Cấp phát bộ nhớ | * Tĩnh: Bộ nhớ được cấp phát trong quá trình biên dịch | * *Động: Bộ nhớ được cấp phát trong quá trình chạy* |
| Thứ tự & sắp xếp | * Được lưu trữ trên một dãy ô nhớ liên tục | * *Được lưu trữ trên các ô nhớ ngẫu nhiên* |
| Truy cập | * *Truy cập tới phần tử ngẫu nhiên trực tiếp bằng cách sử dụng chỉ số mảng: O(1)* | * Truy cập tới phần tử ngẫu nhiên cần phải duyệt từ đầu/cuối đến phần tử đó: O(n) |
| Tìm kiếm | * *Tìm kiếm tuyến tính hoặc tìm kiếm nhị phân* | * Chỉ có thể tìm kiếm tuyến tính |

Cách loại danh sách liên kết:

* Danh sách liên kết đơn- Linked List.
* Danh sách liên kết đôi - Doubly Linked List.
* Danh sách liên kết vòng - Circular Linked List.

Ví dụ cho C++:

Khai báo **node**:

struct node{

int data;

node \*next;

};

Tạo mới một **node**:

node \*createNode(int x){

node \*temp = new node; // tạo mới một node

temp->next = NULL; // node này chưa trỏ đến phần tử khác nên "next" nhận giá trị NULL

temp->data = x; // gán giá trị cho node

return temp;

}

Thêm một phần tử vào cuối **listLinker** khi biết con trỏ đang trỏ vào phần tử cuối:

node \*addElement(node\*p, int x){

node \*temp = createNode(x); // Tạo 1 node mới có giá trị là x.

p->next = temp; // Thêm node đó và cuối danh sách.

return temp; // trả về node temp, vì temp giờ đã là node cuối của list.

}

(sẽ nói rõ việc thêm phần tử vào vị trí bất kỳ ở bài sau).

Duyệt các phần tử trong danh sách liên kết l:

node \*p = l;

while (p != NULL){

// xử lý p

p = p->next;

}

Lưu ý là khi xử lý một danh sách liên kết đơn ta nên dùng một node khác lưu danh sách đó, tránh tình trạng bị mất node đầu của danh sách.

### Giới thiệu danh sách liên kết đôi.

**Danh sách liên kết đôi (Doubly Linked List)** là một tập hợp các **Node** được phân bố động, được sắp xếp theo cách sao cho mỗi **Node** chứa:

* Một giá trị (**Data**).
* Một con trỏ (Next)sẽ trỏ đến phần tử kế tiếp của danh sách liên kết đó, nếu con trỏ mà trỏ tới **NULL**, nghĩa là đó là phần tử cuối cùng của **douList**.
* Một con trỏ (Pre)sẽ trỏ đến phần tử trước của danh sách liên kết đó, nếu con trỏ mà trỏ tới **NULL**, nghĩa là đó là phần tử đầu tiên của **doulist**.

Hình ảnh mình họa cho một **node** trong **douList**.

Graphical user interface

Description automatically generated

Hình ảnh đầy đủ của một danh sách liên kết đôi.

Graphical user interface, diagram

Description automatically generated

**Khai báo dánh sách liên kết đôi (ngôn ngữ C++).**

struct node{

int data;

node \*next;

node \*pre;

};

struct douList{

node \*head;

node \*tail;

};

Mỗi **node** trong danh sách liên kết đôi sẽ có hai liên kết là phần tử đầu và cuối của node đó, cho nên để quản lý danh sách liên kết này chúng ta có thể sử dụng hai **node** là head và tail để lưu **node** đầu và **node** cuối của danh sách.

Tạo một **node** trong dánh sách liên kết đôi l với giá trị của **node** đầu là x:

douList \*createList(int x){

douList \*l = new douList;

l->head = new node;

l->head->data = x;

l->head->pre = NULL;

l->head->next = NULL;

l->tail = l->head;

return l;

}

**Thêm phần tử vào đầu douList.**

Việc thêm phần tử vào đầu của dánh sách liên kết đôi cũng tương tự như danh sách liên kết đơn với độ phức tạp là **O(1)**.

douList \*addHead(douList \*l, int x){

node \*temp = new node;

temp->data = x;

temp->pre = NULL;

temp->next = l->head;

l->head->pre = temp;

l->head = temp;

return l;

}

**Thêm phần tử vào cuối dánh sách liên kết đôi.**

Trong khi để thêm một phần tử và cuối của dánh sách liên kết đơn ta cần độ phức tạp **O(n)** vì cần phải xác định được **node** của cùng của danh sách liên kết đơn đó thì đối với danh sách đôi, **node** cuối cùng đã hoàn toàn đươc xác định (**node** tail) nên khi ta thêm phần tử vào cuối cùng của **douList** chỉ cần độ phức tạp là **O(1)**.

douList \*addTail(douList \*l, int x){

node \*temp = new node;

temp->data = x;

temp->next = NULL;

temp->pre = l->tail;

l->tail->next = temp;

l->tail = temp;

return l;

}

**Duyệt dánh sách liên kết đôi.**

Để duyệt danh sách liên kết đôi ta có thể dùng hai cách: duyệt từ phần tử đầu, hoặc duyệt từ phần tử cuối.

Ví dụ như để in ra một dánh sách liên kết đơn ta có hai cách sau:

Cách 1: Duyệt theo phần tử đầu:

void printList(douList \*l){

node \*p = l->head;

while (p != NULL){

cout << p->data << " ";

p = p->next;

}

}

Cách 2: Duyệt theo phần tử cuối:

void printList(douList \*l){

node \*p = l->tail;

while (p != NULL){

cout << p->data << " ";

p = p->pre;

}

}

**Giới thiệu về tìm kiếm tuyến tính (Linear Search).**

**Linear Search** là một giải thuật tìm kiếm rất cơ bản. Trong kiểu tìm kiếm này, một hoạt động tìm kiếm liên tiếp được diễn ra qua tất cả từng phần tử. Mỗi phần tử đều được kiểm tra và nếu tìm thấy bất kỳ kết nối nào thì phần tử cụ thể đó được trả về; nếu không tìm thấy thì quá trình tìm kiếm tiếp tục diễn ra cho tới khi tìm kiếm hết dữ liệu.

Ví dụ như muốn tìm chỉ số đầu tiên của phần tử có giá trị 10 trong dãy a = [3, 5, 10, 4] ta thực hiện các bước sau:

A picture containing text, clock

Description automatically generated

Kiểm tra phần tử thứ nhất a[0] = 3, không bằng 10 ta tiếp tục kiểm tra phần tử tiếp theo.

Diagram, table

Description automatically generated

Ta kiểm tra a[1] = 5, không bằng 10 ta tiếp tục kiểm tra phần tử tiếp theo.

Diagram, table

Description automatically generated

Ta kiểm tra a[2] = 10, lúc này ta đã tìm thấy phần tử đầu tiên bằng 10, đó là phần tử ở chỉ số thứ 2, trong trường hợp duyệt hết dãy mà vẫn không tìm thấy giá trị 10 thì dãy đó không tồn tại giá trị đó.

Code C++ mẫu:

int search(int a[], int n, int x){

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

if (a[i] == x){

return i;

}

}

return -1;

}

Việc đưa ra phần tử cuối cùng cũng tương tự, ta có duyệt dãy theo chiều ngược lại.

int search(int a[], int n, int x){

for (int i = n-1; i >= 0; i--){

if (a[i] == x){

return i;

}

}

return -1;

}

Hoặc làm như sau:

int search(int a[], int n, int x){

int k = -1;

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

if (a[i] == x){

k = i;

}

}

return k;

}

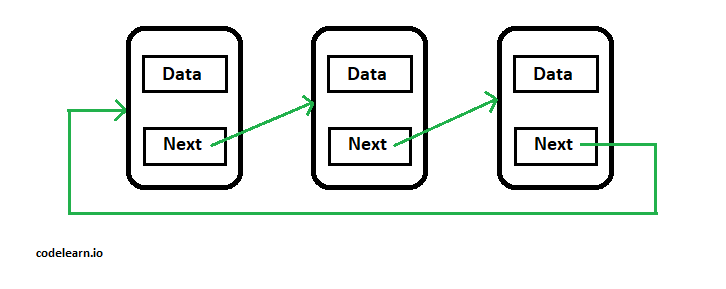
**Giới thiệu danh sách liên kết vòng.**

**Danh sách liên kết vòng (Circular Linked List)** là một biến thể của **Danh sách liên kết (Linked List)**, trong đó phần tử đầu tiên trỏ tới phần tử cuối cùng và phần tử cuối cùng trỏ tới phần tử đầu tiên.

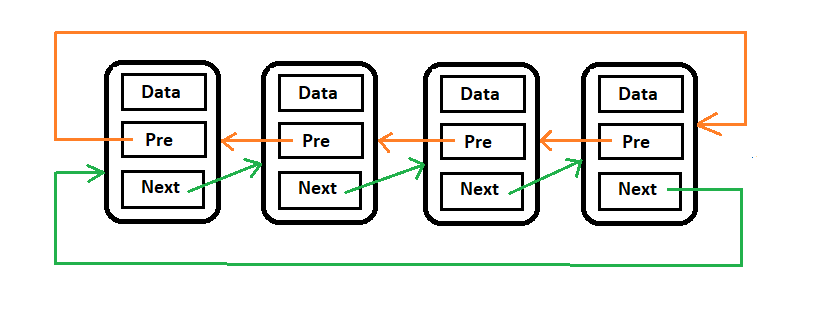
Cả hai loại **Danh sách liên kết đơn (Singly Linked List)** và **Danh sách liên kết đôi (Doubly Linked List)** đều có thể được tạo thành dạng Danh sách liên kết vòng.

Hình ảnh danh sách liên kết vòng:

Tạo tử danh sách liên kết đơn.



Tạo từ danh sách liên kết đồi:



Việc xử lý trên danh sách liên kết vòng cũng tương tự như danh sách liên kết, nhưng chú ý là không được làm mất tính liên kết vòng trong lúc xử lý.

**Giới thiệu giải thuật tìm kiếm nhị phân (Binary Search).**

**Binany Search (Tìm kiếm nhị phân)** là một giải thuật tìm kiếm nhanh với độ phức tạp thời gian chạy là **Ο(log n)**. Giải thuật tìm kiếm nhị phân làm việc dựa trên nguyên tắc chia để trị (**Divide and Conquer**). Để giải thuật này có thể làm việc một cách chính xác thì tập dữ liệu nên ở trong dạng **đã được sắp xếp**.

**Binary Search** tìm kiếm một phần tử cụ thể bằng cách so sánh phần tử tại **vị trí giữa nhất** của tập dữ liệu. Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về. Nếu phần tử cần tìm là **lớn hơn** giá trị phần tử giữa thì phần tử cần tìm được tìm trong mảng con nằm ở **bên phải** phần tử giữa, nếu không thì sẽ tìm ở trong mảng con nằm ở **bên trái** phần tử giữa. Tiến trình sẽ tiếp tục như vậy trên mảng con cho tới khi tìm hết mọi phần tử trên mảng con này.

Chúng ta xem ví dụ sau để hiểu rõ cách hoạt động của **Binary Search**:

Tìm vị trí đầu tiên của x = 10 trong dãy a = [1, 2, 5, 8, 10, 13, 17, 24, 30, 50, 55, 61].

Table

Description automatically generated with medium confidence

Ta dùng biến l để lưu chỉ số đầu của dãy cần xết, biến r để lưu chỉ số cuối của dãy cần xét, ban đầu l = 0 và r = n-1.

Với mỗi bước ta sẽ cần tìm vị trí mid = (l+r)/2. So sánh a[mid] với x.

A picture containing chart

Description automatically generated

Với mid = (l+r)/2 = (0+11)/2 = 5, a[5] = 13 ≥ 10. Ta dễ nhận thấy ràng với a[mid] ≥ x thì phần tử cần tìm chỉ nằm trong khoảng từ vị trí l đến mid.  
Lúc này ta có thể thay đổi r = mid.

Chart

Description automatically generated with medium confidence

Chart

Description automatically generated with medium confidence

Với mid = (l+r)/2 = (0+5)/2 = 2. Với a[2] = 5 < 10. Với a[mid] < x thì dãy cần xét sẽ là dãy từ vị trí mid+1 đến r.  
Cập nhật l = mid+1.

Chart

Description automatically generated with medium confidence

A picture containing histogram

Description automatically generated

Với mid = (l+r)/2 = (3+5)/2 = 4, với a[4] = 10, tuy đã tìm được phần tử bằng x những có thể đó chưa phải là phần tử đầu tiên nên ta tiếp tục xét đoạn l đến mid.

A picture containing histogram

Description automatically generated

Với mid = (l+r)/2 = (3+4)/2 = 3. Với a[3] = 8 < 10. Cập nhất l = mid+1. Lúc này l = r = 4. Lúc này ta cần kiểm tra a[l] nếu a[l] bằng x thì đưa ra l, ngược lại đưa ra -1.

Code C++ mẫu:

int BinSearch(int a[], int n, int x){

int l = 0, r = n-1;

while (l < r){

int mid = (l+r)/2;

if (a[mid] < x){

l = mid+1;

}

else{

r = mid;

}

}

if (a[l] == x){

return l;

}

return -1;

}

**Giới thiệu giải thuật Tìm kiếm nội suy (Interpolation Search).**

**Tìm kiếm nội suy (Interpolation Search)** là biến thể cải tiến của tìm kiếm nhị phân (**Binary Search**). Để giải thuật tìm kiếm này làm việc chính xác thì tập dữ liệu phải được sắp xếp.

**Binary Search** có lợi thế lớn về độ phức tạp thời gian khi so sánh với **Linear Search**. **Linear Search** có độ phức tạp trường hợp xấu nhất là **Ο(n)** trong khi **Binary Search** là **Ο(log n)**.

Có một số tình huống mà vị trí của dữ liệu cần tìm có thể đã được biết trước. Ví dụ, trong trường hợp danh bạ điện thoại, nếu chúng ta muốn tìm số điện thoại của **Hải**chẳng hạn. Trong trường hợp này, **Linear Search** và cả **Binary Search** có thể là chậm khi thực hiện tìm kiếm, khi mà chúng ta có thể trực tiếp nhảy tới phần không gian bộ nhớ có tên bắt đầu với **H** được lưu giữ.

**Xác định vị trí trong Interpolation Search.**

Trong **Binary Search**, nếu dữ liệu cần tìm không được tìm thấy thì phần còn lại của danh sách được phân chia thành hai phần: phần bên trái (chứa giá trị nhỏ hơn) và phần bên phải (chứa giá trị lớn hơn). Sau đó tiến trình tìm kiếm được thực hiện trên một trong hai phần này.

mid = (l+r)/2.

Tìm kiếm nội suy tìm kiếm một phần tử cụ thể bằng việc tính toán vị trí dò (**Probe Position**). Ban đầu thì vị trí dò là vị trí của phần tử nằm ở giữa nhất của tập dữ liệu.

Nếu tìm thấy phần tử đó thì chỉ mục của phần tử được trả về.  
Để chia danh sách thành hai phần, chúng ta sử dụng phương thức sau:

mid = l + (r-l)\*(x-a[l])/(a[r]-a[l])

Trong đó tỉ số (x-a[l])/(a[r]-a[l]) dùng để ước lượng ví trí của x ở trong đoạn từ l đén r.

Nếu phần tử cần tìm có giá trị lớn hơn phần tử ở giữa thì phần tử cần tìm sẽ ở mảng con bên phải phần tử ở giữa và chúng ta lại tiếp tục tính vị trí dò, nếu không phần tử cần tìm sẽ ở mảng con bên trái phần tử ở giữa. Tiến trình này tiến tụp diễn ra trên các mảng con cho tới khi kích cỡ của mảng con giảm về 0.

Độ phức tạp thời gian chạy của **Interpolation Search** là **Ο(log (log n))**, trong khi của **Binary Search** là **Ο(log n)**.

Xem ví dụ sau để biết cách hoạt động của **Interpolation Search:**

Tìm ví trí của x = 20 trong dãy a = [1, 3, 5, 7, 8, 9, 10, 15, 17, 20, 25, 30]

Ta dùng biến l để lưu chỉ số đầu của dãy cần xết, biến r để lưu chỉ số cuối của dãy cần xét, ban đầu l = 0 và r = n-1.

A picture containing timeline

Description automatically generated

Với mỗi bước ta sẽ cần tìm vị trí mid = l + (r-l)\*(x-a[l])/(a[r]-a[l]). So sánh a[mid] với x.

Chart

Description automatically generated with medium confidence

Với mid = l + (r-l)\*(x-a[l])/(a[r]-a[l]) = 7, với a[7] = 15 < x. Lúc này ta gán l = mid + 1 và tiếp tục thực hiện.

A picture containing histogram

Description automatically generated

Với mid = l + (r-l)\*(x-a[l])/(a[r]-a[l]) = 8, với a[8] = 17 < x. Lúc này ta gán l = mid + 1 và tiếp tục thực hiện.

Chart, scatter chart

Description automatically generated

Với mid = l + (r-l)\*(x-a[l])/(a[r]-a[l]) = 9, với a[9] = 20 = x. Trả về giá trị mid và kết thúc.

Nếu như khi a[l] = a[r] hoặc x > a[r] hoặc x < a[l] thì ta kiểm tra giá trị của a[l], nếu a[l] = x thì trả về giá trị l, ngược lại đưa ra -1.

Code mẫu C++:

int interpolationSearch(int a[], int n, int x){

int l = 0, r = n-1;

while (a[r] != a[l] && x >= a[l] && x <= a[r]){

int mid = l + (r-l)\*(x-a[l])/(a[r]-a[l]);

if (a[mid] < x){

l = mid + 1;

} else if (a[mid] > x) {

r = mid - 1;

} else{

if (mid > 0 && a[mid-1] == x){

r = mid - 1;

} else {

return mid;

}

}

}

if (a[l] == x){

return l;

}

return -1;

}

Ta thấy điều kiện vòng while của hàm tìm kiếm nội suy nó khác so với tìm kiếm nhị phân, lý do là chỉ số chọn làm **key** (mid) phải có giá trị nằm trong khoảng 0 đến n-1. và mid = l + (r-l)\*(x-a[l])/(**a[r]-a[l]**).

* Phải kiếm tra a[l] != a[r], phải kiểm tra phần này tránh trường hợp a[l] = a[r] dãn tới tình trạng chia cho số 0. Nếu a[l] = a[r] ta dừng vòng lặp và so sánh giá trị của đoạn cần tìm cho x.
* Để kiểm soát chỉ số mid nằm trong khoảng từ 0 đến n-1 ta phải kiểm tra x >= a[l] && x <= a[r] tránh tình trạng chỉ số mid < 0 hoặc mid > n-1.  
  Ví dụ như với dãy a = [1, 2, 3] và số cần tìm kiếm là x = 5.  
  Nếu ta áp dụng công thức tính mid thì ta đươc mid = 4, không nằm trong khoảng chỉ số của dãy là từ 0 đến n-1, rất cả khả năng dẫn tới lỗi trong quá trình thực hiện.

**Giới thiệu sắp xếp nổi bọt (Bubble Sort).**

**Sắp xếp nổi bọt** là một giải thuật sắp xếp đơn giản. Giải thuật sắp xếp này được tiến hành dựa trên việc so sánh cặp phần tử liền kề nhau và tráo đổi thứ tự nếu chúng không theo thứ tự.

Giải thuật này không thích hợp sử dụng với các tập dữ liệu lớn khi mà độ phức tạp trường hợp xấu nhất và trường hợp trung bình là **Ο(n2)** với n là số phần tử.

Giải thuật sắp xếp nổi bọt là giải thuật chậm nhất trong số các giải thuật sắp xếp cơ bản. Giải thuật này còn chậm hơn giải thuật đổi chỗ trực tiếp mặc dù số lần so sánh bằng nhau, nhưng do đổi chỗ hai phần tử kề nhau nên số lần đổi chỗ nhiều hơn.

**Cách Bubble Sort hoạt động:**

Thuật toán cho cách sắp xếp này sẽ lần lượt đưa các số lớn nhất về ví trí cuối dãy bằng cách so sánh các cặp số kề nhau.

Ví dụ sắp xếp dãy a = [4, 1, 3, 2] thành dãy tăng dần.

A picture containing table

Description automatically generated

Đầu tiên ta so sánh 2 phần tử kề nhau là a[0] và a[1].

A picture containing table

Description automatically generated

Nếu a[0] > a[1] thì ta tráo đổi vị trí của chúng.

Table

Description automatically generated with low confidence

Tiếp tục so sánh a[1] và a[2]:

A picture containing diagram

Description automatically generated

Nếu a[1] > a[2], thì ta tráo đổi vị trí giữa chúng.

A picture containing diagram

Description automatically generated

Tiếp tục so sánh a[2] và a[3].

Table

Description automatically generated

Nếu a[2] > a[3], thì ta tráo đổi vị trí giữa chúng:

Table

Description automatically generated

Lúc này ta nhận thấy rằng phần tử lớn nhất đã được đưa đến vị trí cuối cùng của dãy a, kết thúc một bước sắp xếp.

Như vậy ta thấy rằng: mỗi bước sắp xếp ta sẽ lần lượt đưa được phần tử lớn nhất về cuối dãy.

Với bước sắp xếp tiếp theo ta chỉ cần xét các cặp phần tử a[0] và a[1], a[1] và a[2], không xét cặp a[2] và a[3] nữa, vì a[3] đã ở đúng vị trí rồi.

Sau khi đổi cho a[1] và a[2] dãy sẽ thành:

Table

Description automatically generated

Ta thấy rằng phần tử a[2] đã được đưa về đúng vị trí. Tuy dãy đã được sắp xếp đúng yêu cầu, nhưng ta vẫn phải thực hiện bược sắp xếp tiếp theo là so sánh a[0] và a[1].

Code C++ mẫu:

void bubbleSort(int a[], int n){

for (int i = n-1; i >= 1; i--){

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

if (a[j] > a[j+1]){

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

}

Chúng ta thấy rằng giải thuật sắp xếp nổi bọt so sánh mỗi cặp phần tử trong mảng trừ khi cả toàn bộ mảng đó đã hoàn toàn được sắp xếp theo thứ tự tăng dần. Điều này có thể làm tăng độ phức tạp, tức là tăng các thao tác so sánh và tráo đổi không cần thiết nếu như mảng này không cần sự tráo đổi nào nữa khi tất cả các phần tử đã được sắp xếp theo thứ tự tăng dần rồi.

Để tránh việc này xảy ra, chúng ta có thể sử dụng một biến **swapped** chẳng hạn để giúp chúng ta biết có cần thực hiện thao tác tráo đổi thứ tự hay không. Nếu không cần thiết thì thoát khỏi vòng lặp.

Code C++ mẫu:

void bubbleSort(int a[], int n){

for (int i = n-1; i >= 1; i--){

bool swapped= true;

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

if (a[j] > a[j+1]){

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

swapped = false;

}

}

if (swapped){

break;

}

}

}

**Giới thiệu giải thuật sắp xếp chèn (Insertion Sort).**

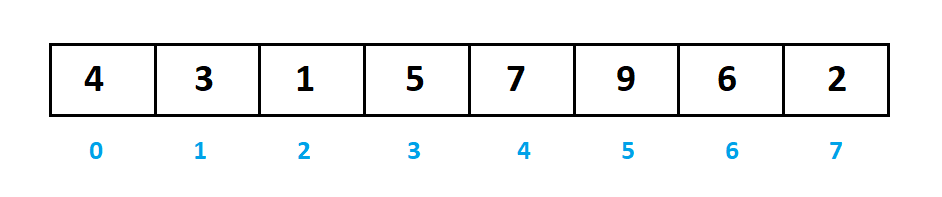
**Sắp xếp chèn** là một giải thuật sắp xếp dựa trên so sánh**in-place**. Ở đây, một danh sách con luôn luôn được duy trì dưới dạng đã qua sắp xếp. Sắp xếp chèn là **chèn** thêm một phần tử vào danh sách con **đã qua sắp xếp**. Phần tử được chèn vào vị trí thích hợp sao cho vẫn đảm bảo rằng danh sách con đó vẫn sắp theo thứ tự.

Với cấu trúc dữ liệu mảng, chúng ta tưởng tượng là: mảng gồm hai phần: một danh sách con đã được sắp xếp và phần khác là các phần tử không có thứ tự. Giải thuật sắp xếp chèn sẽ thực hiện việc tìm kiếm liên tiếp qua mảng đó, và các phần tử không có thứ tự sẽ được di chuyển và được chèn vào vị trí thích hợp trong danh sách con (của cùng mảng đó).

Giải thuật này không thích hợp sử dụng với các tập dữ liệu lớn khi độ phức tạp trường hợp xấu nhất và trường hợp trung bình là **Ο(n2)** với n là số phần tử.

**Cách hoạt động của giải thuật sắp xếp chèn.**

Ví dụ như bài toán sắp xếp dãy a thành dãy không giảm với:  
a = [4, 3, 1, 5, 7, 9, 6, 2]



Để minh họa chúng ta sẽ dùng những phần tử màu xanh để chỉ những phần tử trong dãy con. phần tử màu nâu đề chỉ phần tử đang xét vào.

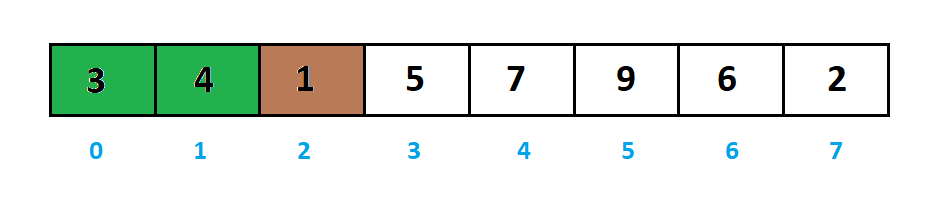
Ban đầu ta sẽ khởi tạo dãy con bằng a[0], đương nhiên với dãy con một phần tử thì nó đã được sắp xếp.

Ta bắt đầu xét từ phần tử có chỉ số 1 đến n-1.

Calendar

Description automatically generated with medium confidence

Sau khi chèn phần tử a[1] vào dãy con đã sắp xếp, dãy a sẽ thay đổi như sau. Ta tiếp tục xét phần tử a[2].

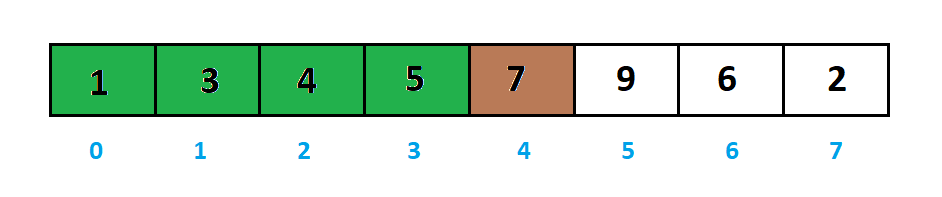


Sau khi chèn phần tử a[2] vào dãy con đã sắp xếp, dãy a sẽ thay đổi như sau. Ta tiếp tục xét phần tử a[3].

Chart, waterfall chart

Description automatically generated

Sau khi chèn phần tử a[3] vào dãy con đã sắp xếp, dãy a sẽ thay đổi như sau. Ta tiếp tục xét phần tử a[4].

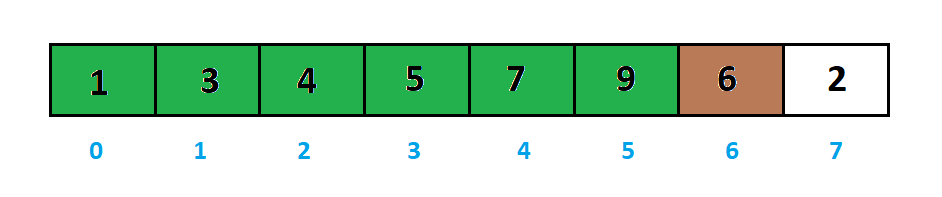


Sau khi chèn phần tử a[4] vào dãy con đã sắp xếp, dãy a sẽ thay đổi như sau. Ta tiếp tục xét phần tử a[5].

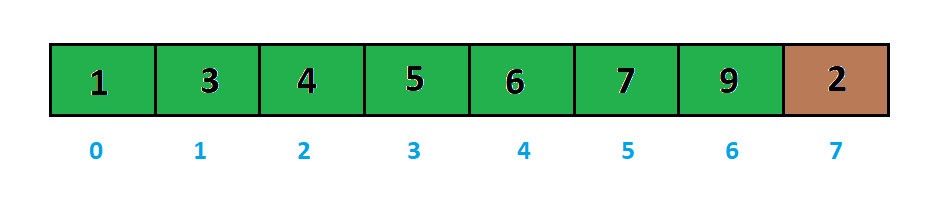
A picture containing chart

Description automatically generated

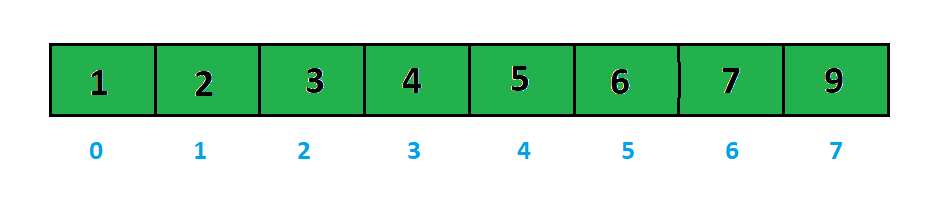
Sau khi chèn phần tử a[5] vào dãy con đã sắp xếp, dãy a sẽ thay đổi như sau. Ta tiếp tục xét phần tử a[6].



Sau khi chèn phần tử a[6] vào dãy con đã sắp xếp, dãy a sẽ thay đổi như sau. Ta tiếp tục xét phần tử a[7].



Chèn a[7] vào dãy a xong là ta đã hoàn thành quá trình sắp xếp.



Code mẫu C++:

void insertionSort(int a[], int n){

int index, value;

for (int i = 1; i < n; i++){

index = i;

value = a[i];

while (index > 0 && a[index - 1] > value){

a[index] = a[index - 1];

index --;

}

a[index] = value;

}

}

**Giới thiệu giải thuật sắp xếp chọn (Selection Sort).**

**Giải thuật sắp xếp chọn (Selection Sort)** là một giải thuật đơn giản. Giải thuật sắp xếp này là một giải thuật dựa trên việc so sánh **in-place**, trong đó danh sách được chia thành hai phần, phần được sắp xếp (**sorted list**) ở bên trái và phần chưa được sắp xếp (**unsorted list**) ở bên phải. Ban đầu, phần được sắp xếp là trống và phần chưa được sắp xếp là toàn bộ danh sách ban đầu.

Phần tử nhỏ nhất được lựa chọn từ mảng chưa được sắp xếp và được tráo đổi với phần bên trái nhất và phần tử đó trở thành phần tử của mảng được sắp xếp. Tiến trình này tiếp tục cho tới khi toàn bộ từng phần tử trong mảng chưa được sắp xếp đều được di chuyển sang mảng đã được sắp xếp.

Giải thuật này không phù hợp với tập dữ liệu lớn khi mà độ phức tạp trường hợp xấu nhất và trường hợp trung bình là **O(n2)** với n là số phần tử.

**Cách hoạt động của Selection Sort:**

Ví dụ như bài toán sắp xếp dãy a thành dãy không giảm với:  
a = [4, 3, 1, 5, 7, 9, 6, 2]

Calendar

Description automatically generated with medium confidence

Để minh họa chúng ta sẽ dùng những phần tử màu xanh để chỉ những phần tử đã được sắp xếp đúng vị trí. phần tử màu nâu đề chỉ phần tử đang nhỏ nhất trong đoạn còn lại.

Ban đầu ta sẽ khởi tạo dãy con bằng a[0], đương nhiên với dãy con một phần tử thì nó đã được sắp xếp.

Ta bắt đầu xét từ phần tử có chỉ số 0 đến n-1.

Đầu tiên ta tìm phần tử nhỏ nhất trong dãy trong khoảng từ 0 đến 7. Sau đó hoán đổi cho phần tử a[0].

Calendar

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with low confidence

Tiếp theo tìm phần tử nhỏ nhất trong dãy trong khoảng từ 1 đến 7. Sau đó hoán đổi cho phần tử a[1].

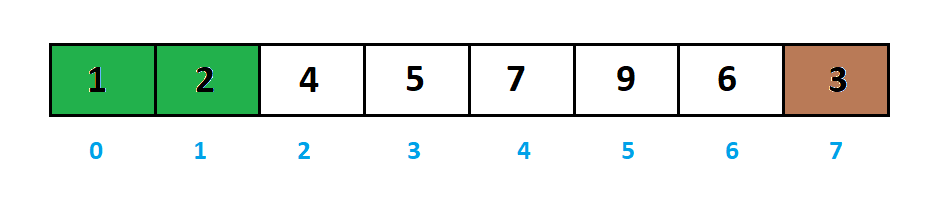
A screenshot of a computer

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with low confidence

Tiếp theo tìm phần tử nhỏ nhất trong dãy trong khoảng từ 2 đến 7. Sau đó hoán đổi cho phần tử a[2].



Calendar

Description automatically generated with medium confidence

Tiếp theo tìm phần tử nhỏ nhất trong dãy trong khoảng từ 3 đến 7. Sau đó hoán đổi cho phần tử a[3].

Calendar

Description automatically generated with medium confidence

Chart, waterfall chart

Description automatically generated

Tiếp theo tìm phần tử nhỏ nhất trong dãy trong khoảng từ 4 đến 7. Sau đó hoán đổi cho phần tử a[4].

Chart, waterfall chart

Description automatically generated

Chart

Description automatically generated with medium confidence

Tiếp theo tìm phần tử nhỏ nhất trong dãy trong khoảng từ 5 đến 7. Sau đó hoán đổi cho phần tử a[5].

Chart

Description automatically generated with medium confidence

Chart

Description automatically generated with low confidence

Tiếp theo tìm phần tử nhỏ nhất trong dãy trong khoảng từ 6 đến 7. Sau đó hoán đổi cho phần tử a[6].

Chart

Description automatically generated with low confidence

Chart

Description automatically generated with low confidence

Còn 1 phần tử cuối cùng chắc chắn đã ở vị trí đúng.

Code C++ mẫu:

void selectionSort(int a[], int n){

int indexMin;

for (int i = 0; i < n-1; i++){

indexMin = i;

for (int j = i+1; j < n; j++){

if (a[indexMin] > a[j]){

indexMin = j;

}

}

if (i != indexMin){

int temp = a[i];

a[i] = a[indexMin];

a[indexMin] = temp;

}

}

}

**Giới thiệu giải thuật sắp xếp trộn (Merge Sort).**

**Sắp xếp trộn (Merge Sort)** là một giải thuật sắp xếp dựa trên giải thuật **Chia để trị (Divide and Conquer)**. Với độ phức tạp thời gian trường hợp xấu nhất là **Ο(n log n)** thì đây là một trong các giải thuật đáng được quan tâm nhất.

Đầu tiên, giải thuật sắp xếp trộn chia mảng thành hai nửa và sau đó kết hợp chúng lại với nhau thành một mảng đã được sắp xếp.

**Ý tưởng thuật toán.**

mergeSort(a[], l, r)

if r > l

1. Tìm chỉ số nằm giữa mảng để chia mảng thành 2 nửa:

middle m = (l+r)/2

2. Gọi đệ quy hàm mergeSort cho nửa đầu tiên:

mergeSort(a, l, m)

3. Gọi đệ quy hàm mergeSort cho nửa thứ hai:

mergeSort(a, m+1, r)

4. Gộp 2 nửa mảng đã sắp xếp ở (2) và (3):

merge(a, l, m, r)

**Hình ảnh ví dụ:**

Diagram

Description automatically generated

Hình ảnh trên hiển thị cho bạn toàn bộ sơ đồ tiến trình của thuật toán merge sort cho mảng [5, 4, 6, 1, 2, 7, 3]. Nếu nhìn kỹ hơn vào sơ đồ này, chúng ta có thể thấy mảng ban đầu được lặp lại hành động chia cho tới khi kích thước các mảng sau chia là 1. Khi kích thước các mảng con là 1, tiến trình gộp sẽ bắt đầu thực hiện gộp lại các mảng này cho tới khi hoàn thành và chỉ còn một mảng đã sắp xếp.

**Giới thiệu giải thuật shell Sort.**

**Shell Sort** là một giải thuật sắp xếp mang lại hiệu quả cao dựa trên giải thuật **sắp xếp chèn (Insertion Sort)**. Giải thuật này tránh các trường hợp phải tráo đổi vị trí của hai phần tử xa nhau trong giải thuật sắp xếp chọn (nếu như phần tử nhỏ hơn ở vị trí bên phải khá xa so với phần tử lớn hơn bên trái).

Đầu tiên, giải thuật này sử dụng giải thuật sắp xếp chọn trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn. Khoảng cách này còn được gọi là **khoảng (interval).**

**interval**sẽ nhận giá trị lần lượt là n/2, n/4, n/8 cho đến khi interval = 1.

Giải thuật này khá hiệu quả với các tập dữ liệu có kích cỡ trung bình khi mà độ phức tạp trường hợp xấu nhất và trường hợp trung bình là **O(n)**, với n là số phần tử.

Cách **shell Sort**hoạt động:

Ví dụ sắp xết dãy a = [9, 1, 3, 7, 8, 4, 2, 6, 5] thành dãy không giảm.

Table, calendar

Description automatically generated

Với interval = 9/2 = 4, ta sẽ chia dãy thành các dãy con với các số cách nhau một khoảng là interval: [9, 8, 5], [1, 4], [3, 2] và [7, 6].

Sắp xếp những dãy con này theo cách **sắp xếp chèn (Insertion Sort).**

Chart

Description automatically generated

Sau khi sắp xếp các dãy con dãy sẽ thành.

Chart

Description automatically generated with medium confidence

Với interval = 9/4 = 2, ta sẽ chia dãy thành các dãy con với các số cách nhau một khoảng là interval: [5, 2, 8, 3, 9], [1, 6, 4, 7].

Chart

Description automatically generated

Sau khi sắp xếp các dãy con dãy sẽ thành.

Chart

Description automatically generated

Với interval = 9/8 = 1, lúc này interval = 1 ta áp dụng sắp xếp chèn với cả dãy a:

A picture containing chart

Description automatically generated

Dãy sau khi sắp xếp là:

Chart

Description automatically generated with low confidence

Code mẫu C++:

void shellSort(int a[], int n){

int interval, i, j, temp;

for(interval = n/2; interval > 0; interval /= 2){

for(i = interval; i < n; i++){

temp = a[i];

for(j = i; j >= interval && a[j - interval] > temp; j -= interval){

a[j] = a[j - interval];

}

a[j] = temp;

}

}

}

**Giới thiệu sắp xếp nhanh (Quick Sort).**

**Giải thuật sắp xếp nhanh (Quick Sort)** là một giải thuật hiệu quả cao và dựa trên việc chia mảng dữa liệu thành các mảng nhỏ hơn. Giải thuật sắp xếp nhanh chia mảng thành hai phần bằng cách so sánh từng phần tử của mảng với một phần tử được chọn gọi là **phần tử chốt (Pivot)**: một mảng bao gồm các phần tử nhỏ hơn hoặc bằng phần tử chốt và mảng còn lại bao gồm các phần tử lớn hơn hoặc bằng phần tử chốt.

Tiến trình chia này diễn ra tiếp tục cho tới khi độ dài của các mảng con đều bằng 1. Giải thuật sắp xếp nhanh tỏ ra khá hiệu quả với các tập dữ liệu lớn khi mà độ phức tạp trường hợp trung bình và trường hợp xấu nhất là **O(nlogn)** với n là số phần tử.

**Kỹ thuật chọn phần tử chốt trong giải thuật sắp xếp nhanh (Quick Sort)**

Kỹ thuật chọn phần tử chốt ảnh hưởng khá nhiều đến khả năng rơi vào các vòng lặp vô hạn đối với các trường hợp đặc biệt. Tốt nhất là chọn phần tử chốt (**pivot**) nằm ở trung vị của danh sách. Khi đó, sau **log2(n)** lần chia chúng ta sẽ đạt tới kích thước các mảng con bằng 1.

Dưới đây là các cách chọn phần tử chốt:

* Chọn phần tử đứng đầu hoặc đứng cuối làm phần tử chốt.
* Chọn phần tử đứng giữa danh sách làm phần tử chốt.
* Chọn phần tử trung vị trong ba phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt.
* Chọn phần tử ngẫu nhiên làm phần tử chốt. Tuy nhiên cách này rất dễ dẫn đến khả năng rơi vào các trường hợp đặc biệt.

Cách hoạt động của **Quick Sort**:

Ví dụ sắp xếp dãy a = [6, 3, 5, 2, 1, 4 , 8, 7] thành dãy không giảm:

Khởi tạo biến l và r là chỉ số đầu và cuối của đoạn cần sắp xếp, khởi tạo l = 0 và r = n-1.

Xác định phần tử chốt trong dãy p = a[(l+r)/2] = a[3] = 2.

Table

Description automatically generated with medium confidence

Sử dụng biến i và biến j để chia dãy thành 2 phần. Biến i sẽ chạy từ l đến r và biến j sẽ chạy từ r về l. Nếu phát hiện a[i] >= p và a[j] <= p thì dừng lại và tráo đổi ví trí của chúng.

while (i < j){

while (a[i] < p){

i++;

}

while (a[j] > p){

j--;

}

if (i <= j){

int temp = a[i];

a[i] = a[j];

a[j] = temp;

i++;

j--;

}

}

Hình ảnh quá trình thực hiện đến khi i >= j thì dừng:

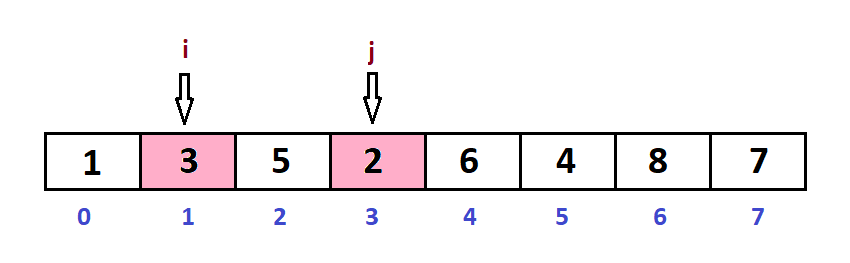
Chart

Description automatically generated

Đổi chỗ a[i] và a[j]:

Table

Description automatically generated with medium confidence



Đổi chỗ a[i] và a[j]:

Table

Description automatically generated

Ta thấy i >= j, ta đã chia dãy dc thành hai phần là:

* Dãy từ 0 đến j: Dãy có giá trị nhỏ hơn hoặc bằng phần tử chốt.
* Dãy từ i đến n-1: Dãy có giá trị lớn hơn hoặc bằng phần tử chốt.

Lúc này ta tiếp tục dùng đệ quy để sắp xếp hai dãy đó.

Code mẫu C++:

void quickSort(int a[], int l, int r){

int p = a[(l+r)/2];

int i = l, j = r;

while (i < j){

while (a[i] < p){

i++;

}

while (a[j] > p){

j--;

}

if (i <= j){

int temp = a[i];

a[i] = a[j];

a[j] = temp;

i++;

j--;

}

}

if (i < r){

quickSort(a, i, r);

}

if (l < j){

quickSort(a, l, j);

}

}