# Giải thuật cơ bản

## Tổng quan về giải thuật

### Tổng quan môn học

**Giới thiệu kiến thức cơ bản về Giải thuật**

1. **Giải thuật là gì**

Giải thuật (hay còn gọi là thuật toán - tiếng Anh là Algorithms) là một tập hợp hữu hạn các chỉ thị để được thực thi theo một thứ tự nào đó để thu được kết quả mong muốn. Nói chung thì giải thuật là độc lập với các ngôn ngữ lập trình, tức là một giải thuật có thể được triển khai trong nhiều ngôn ngữ lập trình khác nhau

Một số giải thuật quan trọng:

**Giải thuật Tìm kiếm:** Giải thuật để tìm kiếm một phần tử trong một cấu trúc dữ liệu

**Giải thuật Sắp xếp:** Giải thuật để sắp xếp các phần tử theo thứ tự nào đó

**Giải thuật Chèn:** Giải thuật để chèn phần từ vào trong một cấu trúc dữ liệu

**Giải thuật Cập nhật:** Giải thuật để cập nhật (hay update) một phần tử đã tồn tại trong một cấu trúc dữ liệu

**Giải thuật Xóa:** Giải thuật để xóa một phần tử đang tồn tại từ một cấu trúc dữ liệu

1. **Đặc điểm của giải thuật**: Một giải thuật nên có các đặc điểm sau

**Tính xác định**: Giải thuật nên rõ ràng và không mơ hồ. Mỗi một giai đoạn (hay mỗi bước) nên rõ ràng và chỉ mang một mục đích nhất định

**Dữ liệu đầu vào xác định**: Một giải thuật nên có 0 hoặc nhiều hơn dữ liệu đầu vào đã xác định

**Kết quả đầu ra**: Một giải thuật nên có một hoặc nhiều dữ liệu đầu ra đã xác định, và nên kết nối với kiểu kết quả bạn mong muốn

**Tính dừng**: Các giải thuật phải kết thúc sau một số hữu hạn các bước

**Tính hiệu quả**: Một giải thuật nên là có thể thi hành được với các nguồn có sẵn, tức là có khả năng giải quyết hiệu quả vấn đề trong điều kiện thời gian và tài nguyên cho phép

**Tính phổ biến**: Một giải thuật có tính phổ biến nếu giải thuật này có thể giải quyết được một lớp các vấn đề tương tự

**Độc lập**: Một giải thuật nên có các chỉ thị độc lập với bất kỳ phần code lập trình nào

1. **Phân tích giải thuật**: Hiệu quả của một giải thuật có thể được phân tích dựa trên 2 góc độ: trước khi triển khai và sau khi triển khai:

**Phân tích lý thuyết**: Có thể coi đây là phân tích chỉ dựa trên lý thuyết. Hiệu quả của giải thuật được đánh giá bằng việc giả sử rằng tất cả các yếu tố khác (ví dụ: tốc độ vi xử lý, …) là hằng số và không ảnh hưởng tới sự triển khai giải thuật

**Phân tích tiệm cận**: Việc phân tích giải thuật này được tiến hành sau khi đã tiến hành trên một ngôn ngữ lập trình nào đó. Sau khi chạy và kiểm tra đo lường các thông số liên quan thì hiệu quả của giải thuật dựa trên các thông số như thời gian chạy, thời gian thực thi, lượng bộ nhớ cần dùng, …

1. **Độ phức tạp của giải thuật**

Độ phức tạp giải thuật là một hàm ước lượng (có thể không chính xác) số phép tính mà giải thuật cần thực hiện (từ đó dễ dàng suy ra thời gian thực hiện của giải thuật) đối với bộ dữ liệu đầu vào (Input) có kích thước **n**. Trong đó, n có thể là số phần tử của mảng trong trường hợp bài toán sắp xếp hoặc tìm kiếm, hoặc có thể là độ lớn của số trong bài toán kiểm tra số nguyên tố, …

Giả sử X là một giải thuật và n là kích cỡ của dữ liệu đầu vào. Thời gian và lượng bộ nhớ được sử dụng bởi giải thuật X là hai nhân tố chính quyết định hiệu quả của giải thuật X:

**+ Nhân tố thời gian (Time complexity)**: Thời gian được đánh giá bằng việc tính số phép tính chính (chẳng hạn như các phép so sánh trong thuật toán sắp xếp)

**+ Nhân tố bộ nhớ (Space complexity)**: Lượng bộ nhớ được đánh giá bằng việc tính lượng bộ nhớ tối đa mà giải thuật cần sử dụng

**Giới thiệu kiến thức cơ bản về Cấu trúc dữ liệu**

1. **Cấu trúc dữ liệu (Data structure) là gì**

Cấu trúc dữ liệu là cách lưu trữ, tổ chức dữ liệu có thứ tự, có hệ thống để dữ liệu có thể được sử dụng một cách hiệu quả.

Hai khái niệm nền tảng hình thành nên một cấu trúc dữ liệu:

**Interface**: Mỗi cấu trúc dữ liệu có một Interface. Interface biểu diễn một tập hợp các phép tính mà một cấu trúc dữ liệu hỗ trợ. Một Interface chỉ cung cấp danh sách các phép tính được hỗ trợ, các loại tham số mà chúng có thể chấp nhận và kiểu trả về của các phép tính này

**Implementation** (có thể hiểu là sự triển khai): Cung cấp sự biểu diễn nội bộ của một cấu trúc dữ liệu. Implementation cũng cung cấp phần định nghĩa của giải thuật được sử dụng trong các phép tính của cấu trúc dữ liệu.

1. **Đặc điểm của Cấu trúc dữ liệu**

**Chính xác**: Sự triển khai của Cấu trúc dữ liệu nên triển khai Interface của nó một cách chính xác

**Độ phức tạp về thời gian (Time Complexity)**: Thời gian chạy hoặc thời gian thực thi của các phép tính của cấu trúc dữ liệu phải là nhỏ nhất có thể

**Độ phức tạp về bộ nhớ (Space Complexity)**: Sự sử dụng bộ nhớ của mỗi phép tính của cấu trúc dữ liệu nên là nhỏ nhất có thể

1. **Tại sao cấu trúc dữ liệu là cần thiết:** Các vấn đề cần lưu ý:

**Tìm kiếm dữ liệu**: Giả sử có 1 triệu hàng hóa được lưu giữ vào trong kho hàng hóa. Và giả sử có một ứng dụng cần để tìm kiếm một hàng hóa. Thì mỗi khi thực hiện tìm kiếm, ứng dụng này sẽ phải tìm kiếm 1 hàng hóa trong 1 triệu hàng hóa. Khi dữ liệu tăng lên thì việc tìm kiếm sẽ càng trở lên chậm và tốn kém hơn

**Tốc độ bộ vi xử lý**: Mặc dù bộ vi xử lý có tốc độ rất cao, tuy nhiên nó cũng có giới hạn và khi lượng dữ liệu lên tới hàng tỉ bản ghi thì tốc độ xử lý cũng sẽ không còn được nhanh nữa

**Đa yêu cầu**: Khi hàng nghìn người dùng cùng thực hiện một phép tính tìm kiếm trên một Web Server thì cho dù Web Server đó có nhanh đến mấy thì việc phải xử lý hàng nghìn phép tính cùng một lúc là thực sự rất khó

1. **Độ phức tạp thời gian thực thi trong cấu trúc dữ liệu và giải thuật:** Có 3 trường hợp để so sánh thời gian thực thi các cấu trúc dữ liệu khác nhau:

**Trường hợp xấu nhất (Worst Case)**: là tình huống mà một phép tính của cấu trúc dữ liệu nào đó tốn thời gian tối đa (thời gian dài nhất). Ví dụ với ba số 1, 2, 3 thì nếu sắp xếp theo thứ tự giảm dần thì thời gian thực thi sẽ là dài nhất (và đây là trường hợp xấu nhất); còn nếu sắp xếp theo thứ tự tăng dần thì thời gian thực thi sẽ là ngắn nhất (và đây là trường hợp tốt nhất)

**Trường hợp trung bình (Average Case)**: miêu tả thời gian thực thi trung bình một phép tính của một cấu trúc dữ liệu

**Trường hợp tốt nhất (Best Case)**: là tình huống mà thời gian thực thi một phép tính của một cấu trúc dữ liệu là ít nhất. Ví dụ như trên

1. **Thuật ngữ cơ bản trong Cấu trúc dữ liệu**

**Dữ liệu**: Dữ liệu là các giá trị hoặc là tập hợp các giá trị

**Phần tử dữ liệu**: Phần tử dữ liệu là một đơn vị đơn lẻ của giá trị

**Các phần tử nhóm**: Phần tử dữ liệu mà được chia thành các phần tử con thì được gọi là các phần tử nhóm

**Các phần tử cơ bản**: Phần tử dữ liệu mà không thể bị chia nhỏ thành các phần tử con thì gọi là các phần tử cơ bản

**Thuộc tính và Thực thể**: Một thực thể là cái mà chứa một vài thuộc tính nào đó, và các thuộc tính này có thể được gán các giá trị

**Tập hợp thực thể**: Các thực thể mà có các thuộc tính tương tự nhau thì cấu thành một tập hợp thực thể

**Trường**: Trường là một đơn vị thông tin cơ bản biểu diễn một thuộc tính của một thực thể

**Bản ghi**: Bản ghi là một tập hợp các giá trị trường của một thực thể đã cho

**File**: Là một tập hợp các bản ghi của các thực thể trong một tập hợp thực thể đã cho

### Một số bài toán về giải thuật cơ bản và độ phức tạp của thuật toán

* **Đệ quy trong Python**

Đệ quy trong Python hay còn gọi là **recursion python**. Nói về toán học thì đệ quy là thuật toán giải quyết bài toán bằng cách gọi lại chính thuật toán đó, thao tác này sẽ thực hiện liên tục cho đến khi gặp điều kiện dừng

VD: Ta có thể sử dụng đệ quy để xác định dãy Fibonacci, tìm ước số chung lớn nhất,…

* **Khái niệm BigO**

Khái niệm Big O hoạc với tên gọi khác trong tiếng Việt là “độ phức tạp của thuật toán” là thuật ngữ thường dùng để chỉ khoảng thời gian tiêu hao để chạy một thuật toán. Các lập trình viên thường sử dụng Big O như một phương tiện để so sánh mức độ hiệu quả của nhiều cách xử lý khác nhau cho cùng một vấn đề

## Giải thuật tham lam

**Giải thuật tham lam** ([tiếng Anh](https://vi.wikipedia.org/wiki/Ti%E1%BA%BFng_Anh): *Greedy algorithm*) là một [thuật toán](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n) giải quyết một bài toán theo kiểu [metaheuristic](https://en.wikipedia.org/wiki/Metaheuristic) để tìm kiếm lựa chọn tối ưu địa phương ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục.

VD: Ở mỗi bước hãy đi đến thành phố gần thành phố hiện tại nhất", bài toán phân nhóm trẻ, bài toán xếp balo

### 5 thành phần trong giải thuật tham lam

* Một tập hợp các ứng viên (candidate), để từ đó tạo ra lời giải
* Một hàm lựa chọn, để theo đó lựa chọn ứng viên tốt nhất để bổ sung vào lời giải
* Một hàm khả thi (*feasibility*), dùng để quyết định nếu một ứng viên có thể được dùng để xây dựng lời giải
* Một hàm mục tiêu, ấn định giá trị của lời giải hoặc một lời giải chưa hoàn chỉnh
* Một hàm đánh giá, chỉ ra khi nào ta tìm ra một lời giải hoàn chỉnh

### 2 thành phần quyết định nhất sử dụng thuật toán Tham Lam:

**Tính chất lựa chọn tham lam**

Thuật toán tiến triển theo kiểu thực hiện các chọn lựa theo một vòng lặp, cùng lúc đó thu nhỏ bài toán đã cho về một bài toán con nhỏ hơn. Đấy là khác biệt giữa thuật toán này và [giải thuật quy hoạch động](https://vi.wikipedia.org/wiki/Quy_ho%E1%BA%A1ch_%C4%91%E1%BB%99ng). Giải thuật quy hoạch động duyệt hết và luôn đảm bảo tìm thấy lời giải. Tại mỗi bước của thuật toán, quy hoạch động đưa ra quyết định dựa trên các quyết định của bước trước, và có thể xét lại đường đi của bước trước hướng tới lời giải. Giải thuật tham lam quyết định sớm và thay đổi đường đi thuật toán theo quyết định đó, và không bao giờ xét lại các quyết định cũ. Đối với một số bài toán, đây có thể là một thuật toán không chính xác.

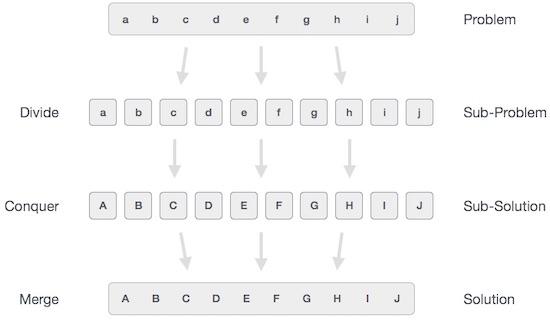
**Cấu trúc con tối ưu**

Một bài toán được gọi là "có cấu trúc tối ưu", nếu một lời giải tối ưu của bài toán con chứa lời giải tối ưu của bài toán lớn hơn.

## Giải thuật chia để trị

### Giới thiệu về giải thuật

Ý tưởng của phương pháp này khá đơn giản và rất dễ hiểu: Khi cần giải quyết một bài toán, ta sẽ tiến hành chia bài toán đó thành các bài toán con nhỏ hơn. Tiếp tục chia cho đến khi các bài toán nhỏ này không thể chia thêm nữa, khi đó ta sẽ giải quyết các bài toán nhỏ nhất này và cuối cùng kết hợp giải pháp của tất cả các bài toán nhỏ để tìm ra giải pháp của bài toán ban đầu



* Tiến trình 1: Chia nhỏ (Devide/Break)

Chia bài toán ban đầu thành các bài toán con. Mỗi bài toán con nên là một phần của bài toán ban đầu. Nói chung, bước này sử dụng phương pháp đệ qui để chia nhỏ các bài toán cho đến khi không thể chia thêm nữa. Khi đó, các bài toán con được gọi là "atomic – nguyên tử", nhưng chúng vẫn biểu diễn một phần nào đó của bài toán ban đầu

* Tiến trình 2: Giải bài toán con (Conquer/Solve)

Các bài toán con được giải

* Tiến trình 3: Kết hợp lời giải (Merge/Combine)

Kết hợp chúng một cách đệ qui để tìm ra giải pháp cho bài toán ban đầu

### Hạn chế

**Tồn tại 2 hạn chế:**

+ Làm thế nào để chia tách bài toán một cách hợp lý thành các bài toán con, bởi vì nếu các bài toán con được giải quyết bằng các thuật toán khác nhau thì sẽ rất phức tạp

+ Việc kết hợp lời giải các bài toán con được thực hiện như thế nào

**Một số giải thuật được xây dựng dựa phương pháp chia để trị:**

+ Giải thuật sắp xếp trộn (Merge Sort)

+ Giải thuật sắp xếp nhanh (Quick Sort)

+ Giải thuật tìm kiếm nhị phân (Binary Search)

…

### 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

**Các bước cho giải thuật tìm kiếm tuyến tính:**

Bước 1: Thiết lập i thành 1

Bước 2: Nếu i > n thì chuyển tới bước 7

Bước 3: Nếu A[i] = x thì chuyển tới bước 6

Bước 4: Thiết lập i thành i + 1

Bước 5: Tới bước 2

Bước 6: In phần tử x được tìm thấy tại chỉ mục i và tới bước 8

Bước 7: In phần tử không được tìm thấy

Bước 8: Thoát

### Tìm kiếm nhị phân (Binary search)

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

Bài toán ví dụ: Trò chơi 2 ô màu khác nhau, Phép nhân đa thức, Giải thuật định lý thợ

## Giải thuật quy hoạch động

**Qui hoạch động (Dynamic Programming)** giống như giải thuật **chia để trị (Divide and Conquer)** trong việc chia nhỏ bài toán thành các bài toán con nhỏ hơn và sau đó thành các bài toán con nhỏ hơn nữa có thể. Nhưng không giống chia để trị, các bài toán con này không được giải một cách độc lập. Thay vào đó, kết quả của các bài toán con này được lưu lại và được sử dụng cho các bài toán con tương tự hoặc các **bài toán con gối nhau (Overlapping Sub-problems)**

Chúng ta sử dụng Qui hoạch động (Dynamic Programming) khi chúng ta có các bài toán mà có thể được chia thành các bài toán con tương tự nhau, để mà các kết quả của chúng có thể được tái sử dụng. Thường thì các giải thuật này được sử dụng cho tối ưu hóa. Trước khi giải bài toán con, giải thuật Qui hoạch động sẽ cố gắng kiểm tra kết quả của các bài toán con đã được giải trước đó. Các lời giải của các bài toán con sẽ được kết hợp lại để thu được lời giải tối ưu.

### So sánh Giải thuật tham lam và Quy hoạch động:

Giải thuật tham lam (Greedy Algorithms) là giải thuật tìm kiếm, lựa chọn giải pháp tối ưu địa phương ở mỗi bước với hi vọng tìm được giải pháp tối ưu toàn cục

Giải thuật Qui hoạch động tối ưu hóa các bài toán con gối nhau

### So sánh Giải thuật chia để trị và Quy hoạch động:

Giải thuật chia để trị (Divide and Conquer) là kết hợp lời giải của các bài toán con để tìm ra lời giải của bài toán ban đầu

Giải thuật Qui hoạch động sử dụng kết quả của bài toán con và sau đó cố gắng tối ưu bài toán lớn hơn. Giải thuật Qui hoạch động sử dụng phương pháp lưu trữ (**Memoization**) để ghi nhớ kết quả của các bài toán con đã được giải

Ví dụ áp dụng giải thuật: Dãy Fibonacci, Bài toán tháp Hà Nội, Bài toán Balo, Bài toán đổi tiền, So sánh 2 string, Bài toán cái túi,…

## Các giải thuật sắp xếp

### Giải thuật sắp xếp trong cấu trúc dữ liệu & giải thuật

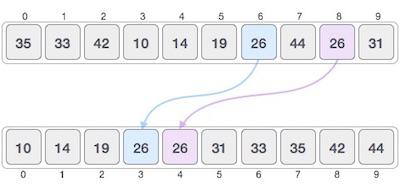
**Giải thuật sắp xếp In-Place và Not-in-Place**

Những giải thuật mà không yêu cầu thêm bất kỳ bộ nhớ phụ và việc sắp xếp được tiến hành trong chính phần bộ nhớ đã khai báo trước đó (ví dụ trong một mảng chẳng hạn) thì được gọi là **in-place sorting**. Ví dụ cho loại giải thuật sắp xếp này là giải thuật sắp xếp nổi bọt (**bubble sorting**)

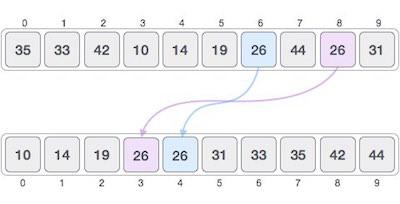
Nhưng trong một số giải thuật sắp xếp, chương trình cần thêm lượng bộ nhớ mà có thể lớn hơn hoặc bằng với số phần tử đang được sắp xếp. Các giải thuật này được gọi là **not-in-place sorting**. Ví dụ cho loại giải thuật này là sắp xếp trộn (**merge sort**)

**Giải thuật sắp xếp cố định và sắp xếp so sánh**

Một giải thuật sắp xếp được gọi là sắp xếp cố định nếu sau khi tiến hành sắp xếp thì vị trí tương đối giữa các phần tử bằng nhau không bị thay đổi



Một giải thuật được gọi là sắp xếp so sánh nếu trong quá trình thực hiện giải thuật chúng ta tiến hành so sánh các khóa và đổi chỗ các phần tử cho nhau. Tức là khi đó vị trí tương đối của các phần tử bằng nhau bị thay đổi



**Giải thuật sắp xếp Adaptive và Non-Adaptive**

Một giải thuật được xem như là adaptive, nếu nó tận dụng các phần tử đã được sắp xếp trong danh sách mà đã được sắp xếp. Đó là, trong khi sắp xếp nếu danh sách ban đầu có một số phần tử đã được sắp xếp, thì giải thuật dạng adaptive sẽ ghi nhận các phần tử này và sẽ cố gắng không thay đổi thứ tự của chúng.

Trái ngược với loại giải thuật trên, giải thuật dạng non-adaptive sẽ không ghi nhận các phần tử đã được sắp xếp trước đó. Giải thuật loại này sẽ vấn cố gắng sắp xếp lại từng phần tử trong danh sách ban đầu.

### Selection Sort (Sắp xếp chọn)

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.

**Bước thực hiện giải thuật:**

Bước 1: Thiết lập MIN về vị trí 0

Bước 2: Tìm kiếm phần tử nhỏ nhất trong danh sách

Bước 3: Tráo đổi với giá trị tại vị trí MIN

Bước 4: Tăng MIN để trỏ tới phần tử tiếp theo

Bước 5: Lặp lại cho tới khi toàn bộ danh sách đã được sắp xếp

### Merge Sort (Sắp xếp trộn)

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.

**Bước thực hiện giải thuật:**

Bước 1: Nếu chỉ có một phần tử trong list thì list này được xem như là đã được sắp xếp. Trả về list hay giá trị nào đó.

Bước 2: Chia list một cách đệ qui thành hai nửa cho tới khi không thể chia được nữa.

Bước 3: Kết hợp các list nhỏ hơn (đã qua sắp xếp) thành list mới (cũng đã được sắp xếp).

### Sắp xếp không dùng phép so sánh (Tham khảo)

<https://programmersought.com/article/44427356811/>

### Quick Sort (Sắp xếp nhanh)

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

Tham khảo:

<https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm>

# Cấu trúc dữ liệu tuyến tính

## Mảng và danh sách liên kết

### Mảng

**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 với 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. Sử dụng mảng động ta bắt đầu với mảng có 1 phần tử, khi số lượng phần tử vượt qua khả năng của mảng 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.

**Các phép toán có trong 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.

Tham khảo: https://www.tutorialspoint.com/python\_data\_structure/python\_arrays.htm

### Các loại danh sách liên kết

**Danh sách liên kết đơn (Simple Linked List)**: chỉ duyệt các phần tử theo chiều về trước.

**Danh sách liên kết đôi (Doubly Linked List)**: các phần tử có thể được duyệt theo chiều về trước hoặc về sau

**Danh sách liên kết vòng (Circular Linked List)**: phần tử cuối cùng chứa link của phần tử đầu tiên như là next và phần tử đầu tiên có link tới phần tử cuối cùng như là prev

### Danh sách liên kết đơn

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.

### 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.

### Hoạt độ**n**g cơ bản trên Danh sách liên kết

**Hoạt động chèn**: thêm một phần tử vào đầu danh sách liên kết

**Hoạt động xóa (phần tử đầu)**: xóa một phần tử tại đầu danh sách liên kết

**Hiển thị**: hiển thị toàn bộ danh sách

**Hoạt động tìm kiếm**: tìm kiếm phần tử bởi sử dụng khóa (key) đã cung cấp

**Hoạt động xóa (bởi sử dụng khóa)**: xóa một phần tử bởi sử dụng khóa (key) đã cung cấp

Tham khảo:

<https://www.tutorialspoint.com/python_data_structure/python_advanced_linked_list.htm>

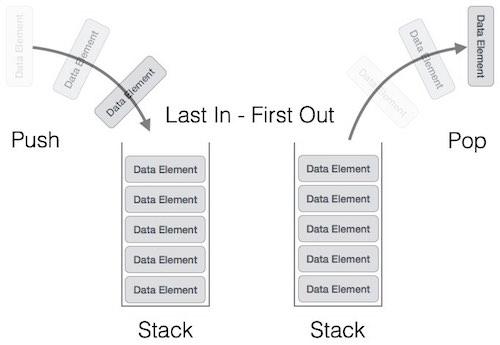
## Ngăn xếp và hàng đợi

### Stack (Ngăn xếp)

Stack là một cấu trúc dữ liệu trừu tượng hoạt động theo nguyên lý "vào sau ra trước" (Last In First Out - LIFO). Một ngăn xếp là một cấu trúc dữ liệu dạng thùng chứa (container) của các phần tử (thường gọi là các nút (node)) và có hai phép toán cơ bản: push and pop. Push bổ sung một phần tử vào đỉnh (top) của ngăn xếp, nghĩa là bổ sung vào sau các phần tử đã có trong ngăn xếp. Pop là giải phóng và trả về phần tử đang đứng ở đỉnh của ngăn xếp . Trong stack, các đối tượng có thể được bổ sung vào stack bất kỳ lúc nào nhưng chỉ được phép lấy ra phần tử phía đỉnh của stack

Một trong các ứng dụng quan trọng của các cấu trúc dữ liệu là chúng có thể được sử dụng để triển khai các cấu trúc dữ liệu khác. Để cài đặt Stack chúng ta có thể sử dụng mảng để cài đặt

**Biểu diễn cấu trúc ngăn xếp**



Một ngăn xếp có thể được triển khai theo phương thức của Mảng (Array), Cấu trúc (Struct), Con trỏ (Pointer) và Danh sách liên kết (Linked List). Ngăn xếp có thể là ở dạng kích cỡ cố định hoặc ngăn xếp có thể thay đổi kích cỡ. Phần dưới chúng ta sẽ triển khai ngăn xếp bởi sử dụng các mảng với việc triển khai các ngăn xếp cố định.

### Các hoạt động cơ bản trên cấu trúc dữ liệu ngăn xếp

Các hoạt động cơ bản trên ngăn xếp có thể liên quan tới việc khởi tạo ngăn xếp, sử dụng nó và sau đó xóa nó. Ngoài các hoạt động cơ bản này, một ngăn xếp có hai hoạt động nguyên sơ liên quan tới khái niệm, đó là:

**Hoạt động push()**: lưu giữ một phần tử trên ngăn xếp

**Hoạt động pop()**: xóa một phần tử từ ngăn xếp

Khi dữ liệu đã được PUSH lên trên ngăn xếp:

Để sử dụng ngăn xếp một cách hiệu quả, chúng ta cũng cần kiểm tra trạng thái của ngăn xếp. Để phục vụ cho mục đích này, dưới đây là một số tính năng hỗ trợ khác của ngăn xếp:

**Hoạt động peek()**: lấy phần tử dữ liệu ở trên cùng của ngăn xếp, mà không xóa phần tử này

**Hoạt động isFull()**: kiểm tra xem ngăn xếp đã đầy hay chưa

**Hoạt động isEmpty()**: kiểm tra xem ngăn xếp là trống hay không

**Hoạt động Push**

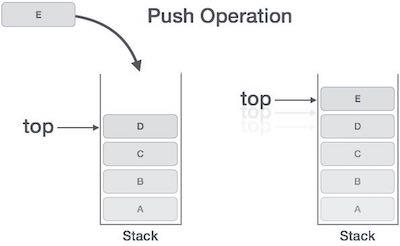
**Bước 1**: kiểm tra xem ngăn xếp đã đầy hay chưa

**Bước 2**: nếu ngăn xếp là đầy, tiến trình bị lỗi và thoát ra

**Bước 3**: nếu ngăn xếp chưa đầy, tăng top để trỏ tới phần bộ nhớ trống tiếp theo

**Bước 4**: thêm phần tử dữ liệu vào vị trí nơi mà top đang trỏ đến trên ngăn xếp

**Bước 5**: trả về success



**Hoạt động Pop**

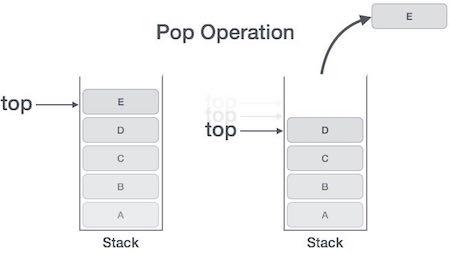
**Bước 1**: kiểm tra xem ngăn xếp là trống hay không.

**Bước 2**: nếu ngăn xếp là trống, tiến trình bị lỗi và thoát ra.

**Bước 3**: nếu ngăn xếp là không trống, truy cập phần tử dữ liệu tại **top** đang trỏ tới.

**Bước 4**: giảm giá trị của top đi 1.

**Bước 5**: trả về success.



# Cấu trúc dữ liệu phi tuyến tính

## Cây – Các thao tác cơ bản

### Tổng quan về Cây

**Các khái niệm cơ bản về cây nhị phân**

**Đường**: là một dãy các nút cùng với các cạnh của một cây.

**Nút gốc (Root)**: nút trên cùng của cây được gọi là nút gốc. Một cây sẽ chỉ có một nút gốc và một đường xuất phát từ nút gốc tới bất kỳ nút nào khác. Nút gốc là nút duy nhất không có bất kỳ nút cha nào.

**Nút cha**: bất kỳ nút nào ngoại trừ nút gốc mà có một cạnh hướng lên một nút khác thì được gọi là nút cha.

**Nút con**: nút ở dưới một nút đã cho được kết nối bởi cạnh dưới của nó được gọi là nút con của nút đó.

**Nút lá**: nút mà không có bất kỳ nút con nào thì được gọi là nút lá.

**Cây con**: cây con biểu diễn các con của một nút.

**Truy cập**: kiểm tra giá trị của một nút khi điều khiển là đang trên một nút đó.

**Duyệt**: duyệt qua các nút theo một thứ tự nào đó.

**Bậc**: bậc của một nút biểu diễn số con của một nút. Nếu nút gốc có bậc là 0, thì nút con tiếp theo sẽ có bậc là 1, và nút cháu của nó sẽ có bậc là 2,…

**Khóa (Key)**: biểu diễn một giá trị của một nút dựa trên những gì mà một thao tác tìm kiếm thực hiện trên nút.

### Hoạt động cơ bản trên cây tìm kiếm nhị phân

**Chèn**: chèn một phần tử vào trong một cây/ tạo một cây.

**Tìm kiếm**: tìm kiếm một phần tử trong một cây.

**Duyệt tiền thứ tự**: duyệt một cây theo cách thức duyệt tiền thứ tự (tham khảo chương sau).

**Duyệt trung thứ tự**: duyệt một cây theo cách thức duyệt trung thứ tự (tham khảo chương sau).

**Duyệt hậu thứ tự**: duyệt một cây theo cách thức duyệt hậu thứ tự (tham khảo chương sau).

### Duyệt cây

**Duyệt PreOrder**

Quy trình duyệt PreOrder sẽ thực hiện theo thứ tự Node -> Left -> Right, cụ thể như sau:

1. Ghé thăm Node root
2. Gọi đệ quy duyệt qua cây con bên trái
3. Gọi đệ quy duyệt qua cây con bên phải

**Duyệt InOrder**

Quy trình duyệt PreOrder sẽ thực hiện theo thứ tự Left-> Node -> Right, cụ thể như sau:

1. Gọi đệ quy duyệt qua cây con bên trái
2. Ghé thăm Node root
3. Gọi đệ quy duyệt qua cây con bên phải

**Duyệt PostOrder**

Quy trình duyệt PreOrder sẽ thực hiện theo thứ tự Left -> Right -> Node, cụ thể như sau:

1. Gọi đệ quy duyệt qua cây con bên trái
2. Gọi đệ quy duyệt qua cây con bên phải
3. Ghé thăm Node root

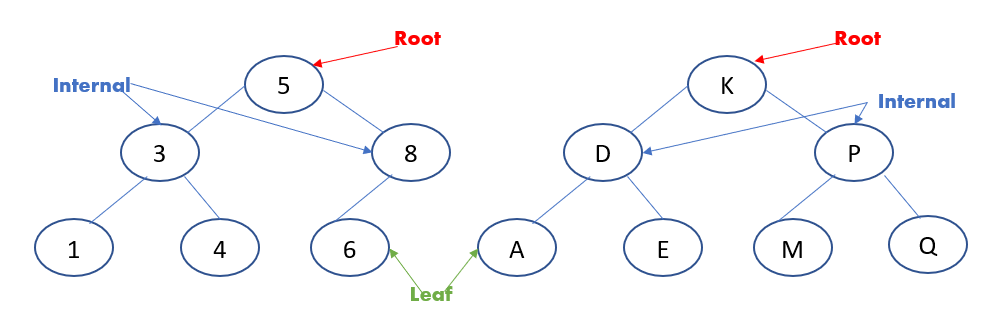
## Cây tìm kiếm nhị phân: BST

### Cây tìm kiếm nhị phân là gì ?

Cây tìm kiếm nhị phân có tên tiếng anh là Binary Search Tree (BST), là một trong những cấu trúc dữ liệu cơ bản bên cạnh queue, stack, linked-list, array. Cây tìm kiếm nhị phân là 1 dạng đồ thị nhưng các nút (node) của cây phải có những tính chất sau:

* Mỗi node chỉ có thể có tối đa 2 node con
* Giá trị của node con bên trái phải nhỏ hơn node cha của nó
* Giá trị của node con bên phải lớn hơn node cha của nó

Tính chất cũng phải đúng cới các node con của 2 node con trên, nói cách khác giá trị tất cả các con bên trái của 1 node phải nhỏ hơn giá trị của node đó và giá tị tất cả các con bên phải của node đó phải lớn hơn giá tị của nó. Sự so sánh giá trị ở trên có thể là so sánh toán học, so sánh chuỗi kí tự,…

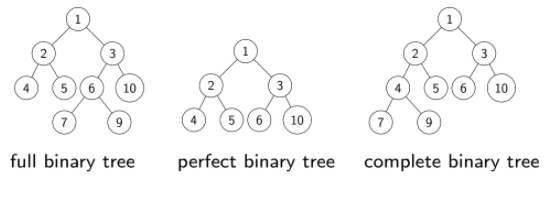


Đặc biệt trong 1 cây tìm kiếm nhị phân không cho phép 2 giá trị trùng nhau. Chính quy luật và cách sắp xếp như trên cấu trúc BST đã giúp sắp xếp dữ liệu theo một cách có trật tự, từ đó giúp người sử dụng dễ dàng hơn trong việc tổ chức dữ liệu cũng như việc tìm kiếm

### Một vài khái niệm trong cây tìm kiếm nhị phân

* Root node (nút gốc) : node đầu tiên của cây.
* Leaf node (nút lá): node không có con trái và con phải.
* Internal node : những node không phải nút gốc cũng không phải nút lá.
* Level (tầng): như hình minh họa trên chúng ta có 2 cây với 3 tầng.

### Các loại cây nhị phân cơ bản



**Full binary tree:** Những node không phải nút lá đều có 2 con trái và phải

**Complete binary tree:** Tất cả các tầng đều chứa đầy nodes ngoại trừ tầng cuối có thể đầy hoặc không nhưng các node tầng cuối phải đc xếp lần lượt từ trái đến phải

**Perfect binary tree:** Tất cả nodes đều có 2 con và các nút lá ở cùng một level

Tham khảo:

<https://www.tutorialspoint.com/python_data_structure/python_binary_search_tree.htm>

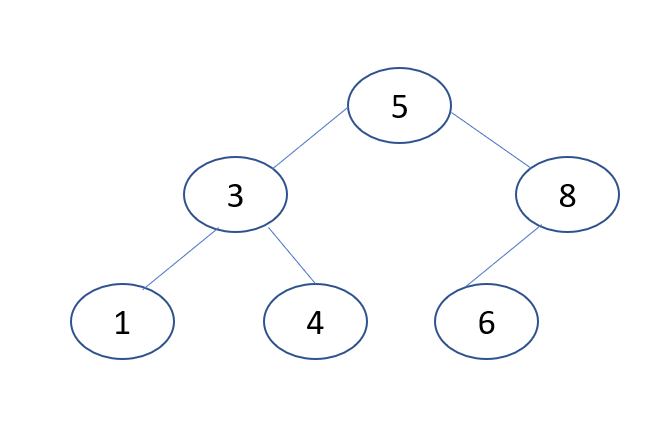
### Các thao tác cơ bản trên cây nhị phân

**Search**

Để bắt đầu hoạt động tìm kiếm một giá trị x cho trước, chúng ta bắt đầu từ nút gốc (root). Nếu giá trị x nhỏ hơn giá trị của root thì chuyển đến so sánh với node con trái của root vì như đã nói ở trên mọi node bên trái root sẽ nhỏ hơn root nên nếu giá trị x có tồn tại thì chỉ có thể ở bên trái root, còn với x lớn hơn giá trị của root thì chuyển đến so sánh với node con phải của root.

Tiếp tục quá trình xét như trên với các node tiếp theo đến khi tìm được, còn nếu đến nút lá mà so sánh x không bằng giá trị nút lá thì xác nhận không tìm thấy.

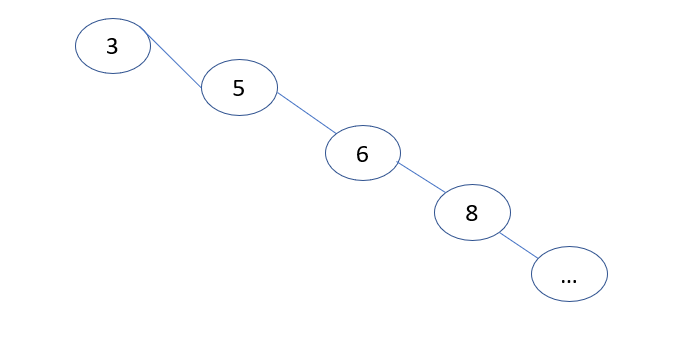
Ví dụ:



Trường hợp thứ nhất, ta muốn xác nhận 4 có nằm trong cây trên hay không thì các node ta sẽ đi qua là 5,3. Ban đầu bắt đầu từ 5, vì 4<5 nên so sánh tiếp với con trái của 5 là 3. Thấy 4>3 nên so sánh tiếp với con phải của 3 là 4, ở đây 4 = 4 nên kết quả là có tìm thấy.

Trường hợp thứ 2, ta muốn xác nhận 7 có nằm trong cây trên không? Các node đi qua sẽ là 5,8,6 (không tìm thấy). Ban đầu chúng ta cũng bắt đầu từ root node là 5 sau đó lần lượt đến 8 và 6, cuối cùng xác nhận không tìm thấy.

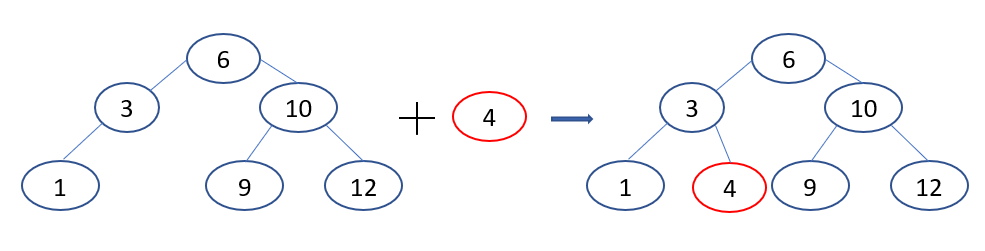
Đây chính là ưu điểm của cấu trúc cây tìm kiếm nhị phân, độ phức tạp của thuật toán tìm kiếm này là O(logn), với trường hợp xấu nhất khi tất cả cá node chỉ có con phải hoặc chỉ có con trái thì độ phức tạp là O(n), ta có thể thấy rằng với trường hợp xấu nhất này cây nhị phân sẽ giống với cấu trúc mảng.



Chúng ta nên tránh trường hợp này khi xây dựng cấu trúc dữ liệu với cây nhị phân tìm kiếm.

**Insert**

Khi muốn thêm một giá trị x vào cây nhị phân, ta bắt đầu tìm kiếm từ nút gốc (root), nếu giá trị x nhỏ hơn giá trị nút gốc thì tìm vị trí trống của cây con bên trái nút gốc, nếu x lớn hơn giá trj nút gốc ta tìm vị trí trống của cây con bên phải nút gốc. Trường hợp tìm được giá trị của 1 node trong cây bằng với x thì xác nhận x đã tồn tại trong cây.



Với ví dụ trong ảnh trên, ta cần thêm 4 vào trong cây nhị phân cho trước. Bắt đầu từ nút gốc là 6, vì 4 < 6 nên tìm vị trí trống phía cây con bên trái của 6, tiếp theo 4>3 và có một ví trí trống phía bên phải của nút 3 vây nên đó là vị trí phù hợp để them 4 vào trong cây.

**Remove**

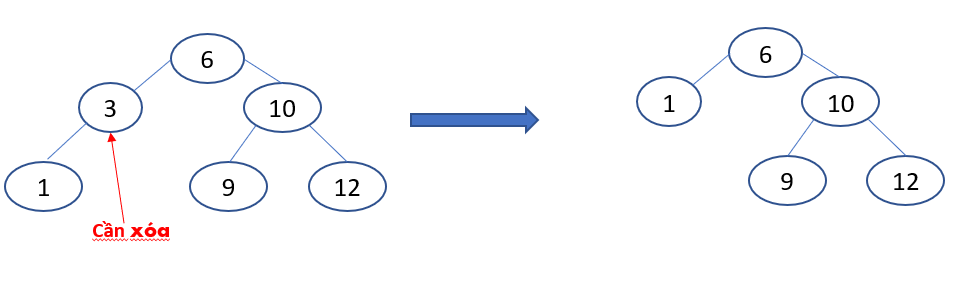
Trong hoạt động xóa 1 node của cây nhị phân chúng ta sẽ gặp phải 3 trường hợp sau:

+ Node cần xóa chỉ có 1 node con (trái hoặc phải)

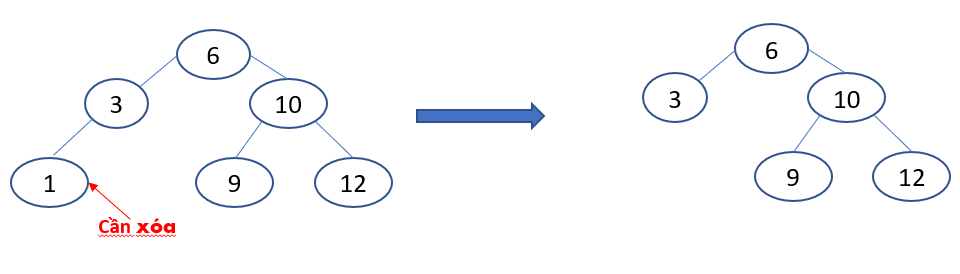
+ Node cần xóa không có node con

+ Node cần xóa có cả 2 node

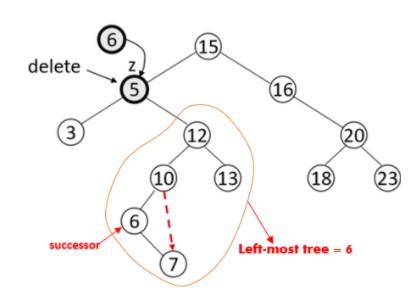
Với trường hợp đầu tiên node cần xóa có 1 node con, ta chỉ cần thay vị trí của node con đó với node cần xóa.



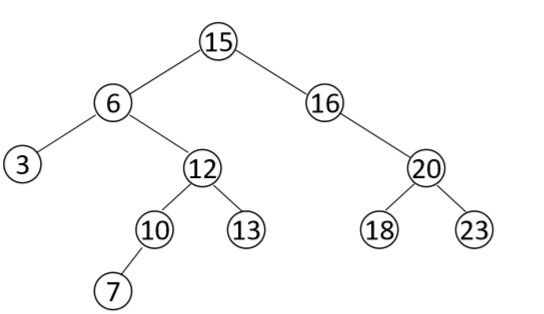
Với trường hợp node cần xóa không có node con thì chúng ta đơn giản chỉ cần xóa vị trí node đó khỏi cây.



Trường hợp cuối cùng node cần xóa có cả 2 node con. Với trường hợp này việc của ta cần làm là tìm được 1 node thế (successor) để lắp vào vị trí của node cần xóa, nói cách khác node thế phải có tính chất bé hơn tất cả node bên trái của node cần xóa và lớn hơn tất cả các node bên phải của node cần xóa.



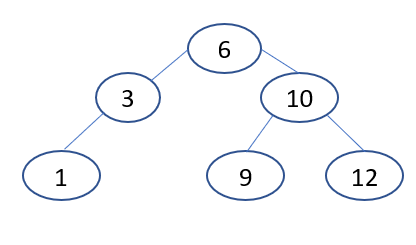
Với cây nhị phân trong hình, khi ta muốn xóa node 5 thì node 6 chính là node thế cho node 5, node 6 còn được gọi là left-most tree, tức node trái cùng của 1 cây. Sau khi thay node 5 bằng node 6 ta cần xóa node 6 ở vị trí cũ đi, khi này ta quay trở lại trường hợp xóa 1 node có 1 node con tại vì node 6 cũ có 1 node còn là 7. Đây là kết quả sau khi xóa node 5:



Node thế (successor) có thể là left-most của cây con bên phải hoặc right-most tree của cây con bên trái. Với left-most tree được định nghĩa là con trái cùng hay giá trị nhỏ nhất trong cây nhị phân, right-most là conphair cùng, hay giá trị lớn nhất trong cây nhị phân

**Pre – order traversal**

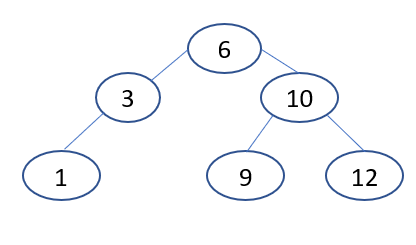
Với cách duyệt này ta sẽ đi qua node cha trước sau đến node con trái rồi mới đến node con phải.



Với cây trên thứ tự các code sau khi duyệt là: 6,3,1,10,9,12.

* **In – order traversal**

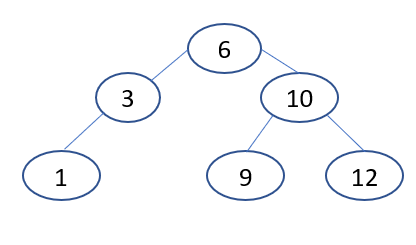
Trong cách duyệt in-order, duyệt lần lượt node con trái sau đến node cha rồi đến node con phải.



Duyệt cây nhị phân trên với inorder traversal ta được một dãy tăng dần: 1,3,6,9,10,12.

**Post – order traversal**

Ta duyệt lần lượt node con trái, node con phải rồi đến node cha



Với cây trên thứ tự các code sau khi duyệt là1,3,9,12,10,6.

## Đống

### Hàng đợi ưu tiên

Priority Queue (Hàng đợi ưu tiên) là một phần mở rộng của Queue.

Nó có các đặc điểm sau:

* Mọi phần tử trong hàng đợi đều có một mức độ ưu tiên gắn với nó
* Một phần tử có độ ưu tiên cao hơn sẽ được xử lý (dequeued) trước một phần tử có độ ưu tiên thấp
* Nếu hai phần tử có cùng độ ưu tiên, chúng sẽ được xử lý lần lượt theo thứ tự của chúng trong hàng đợi

### Heap (Đống) – Binary heap (Đống nhị phân)

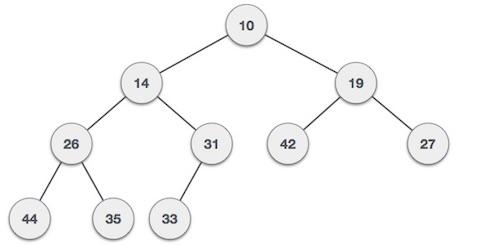
Heap là loại cấu trúc dữ liệu dạng cây, và tất cả các node trong cây đó được sắp xếp theo một thứ tự nhất định, có thể là theo chiều tăng dần hoặc giảm dần. Giả sử ta có A là node cha của B, tuân theo định nghĩa về heap, nếu giá trị của node A lớn hơn giá trị của node B thì quan hệ này cũng được áp dụng cho toàn bộ cây. Có nghĩa là giá trị của node B sẽ lớn hơn giá trị của node con của nó và cứ như vậy, thứ tự này được áp dụng cho toàn bộ cây.

Số con tối đa của một node trong heap phụ thuộc vào từng loại heap. Trong thực tế ta có nhiều loại heap nhưng loại được dùng phổ biến nhất là heap nhị phân. Nếu heap nhị phân là một cây nhị phân hoàn thiện với N node. Thì chiều cao của nó sẽ là nhỏ nhất trong các cây nhị phân có N node và ta có chiều cao của nó là:

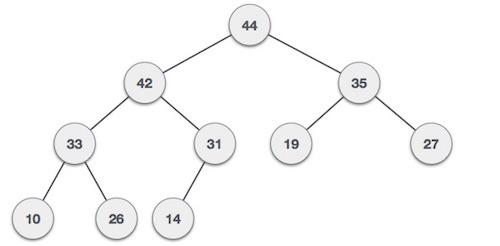
**H=log2(N+1)−1**

**Một Heap có thể là một trong hai kiểu sau:**

**Min-Heap**: ở đây giá trị của nút gốc là nhỏ hơn hoặc bằng các giá trị của các nút con



**Max-Heap**: ở đây giá trị của nút gốc là lớn hơn hoặc bằng giá trị của các nút con.



### Xây dựng MaxHeap (MinHeap tương tự)

**Bước 1**: Tạo một nút mới tại vị trí cuối cùng của Heap.

**Bước 2**: Gán giá trị mới cho nút này.

**Bước 3**: So sánh giá trị của nút con với giá trị cha.

**Bước 4**: Nếu giá trị của cha là nhỏ hơn con thì tráo đổi chúng.

**Bước 5**: Lặp lại bước 3 và 4 cho tới khi vẫn duy trì thuộc tính của Heap.

### Xóa trong MaxHeap

**Bước 1**: Xóa nút gốc.

**Bước 2**: Di chuyển phần tử cuối cùng có bậc thấp nhất lên nút gốc.

**Bước 3**: So sánh giá trị của nút con này với giá trị của cha.

**Bước 4**: Nếu giá trị của cha là nhỏ hơn của con thì **tráo đổi** chúng.

**Bước 5**: Lặp lại bước 3 và 4 cho tới khi vẫn duy trì thuộc tính của Heap.

### Heap sort (Sắp xếp vun đống)

Giải thuật Heapsort còn được gọi là giải thuật vun đống, có thể được xem như bản cải tiến của Selection Sort khi chia các phần tử thành 2 mảng con

* 1  mảng các phần tử đã được sắp xếp
* 1 mảng các phần tử chưa được sắp xếp

Trong mảng chưa được sắp xếp, các phần tử lớn nhất sẽ được tách ra và đưa vào mảng đã được sắp xếp. Điều cải tiến ở Heapsort so với Selection Sort ở việc sử dụng cấu trúc dữ liệu heap thay vì tìm kiếm tuyến tính (linear-time search) như Selection sort để tìm ra phần tử lớn nhất

Heapsort là thuật toán in-place, nghĩa là không cần thêm bất cứ cấu trúc dữ liệu phụ trợ trong quá trình chạy thuật toán

Tham khảo: <https://www.tutorialspoint.com/python_data_structure/python_heaps.htm>

## Hảm băm và bảng băm

### Hashing

Hashing là một kỹ thuật dùng để xác định duy nhất một đối tượng cụ thể từ một nhóm các đối tượng tương tự nó. Một vài ví dụ về hashing trong cuộc sống thực tế:

Mỗi sinh viên trong một trường đại học được giao cho một số ID, từ ID này ta có thể truy cập để tìm kiếm các thông tin liên quan tới sinh viên. Mỗi cuốn sách trong thư viện cũng được gán cho một ID duy nhất, từ ID này ta có thể biết được vị trí chính xác của nó trong thư viện hoặc các người dùng đã mượn nó. Trong các ví dụ trên, sách và sinh viên đã được mã hóa với một ID duy nhất, kỹ thuật đó gọi là hashing.

Giả sử rằng bạn có một đối tượng và bạn một gán cho nó một ID để dễ quản lý. Khi gán cho đối tượng một ID thì một cặp key/value được hình thành, với key là ID mà bạn gán và value chính là đối tượng được gán cho ID đó. Để thực hiện việc trên, đơn giản là bạn có thể dùng mảng, lúc này key chính là chỉ số của mảng, nơi lưu trữ đối tượng (value). Tuy nhiên, trong trường hợp khi key quá lớn thì việc dùng key trực tiếp làm chỉ số của mảng không còn hiệu quả nữa và bạn nên dùng hashing.

**Hashing có thể được thực hiện trong hai bước:**

* Chuyển đổi khóa (key) thành số nguyên (khóa nhỏ hơn) dùng hàm băm. Khóa sau khi chuyển đổi có thể được dùng như chỉ số để chứa phần tử ban đầu nằm trong bảng băm
* Phần tử được chứa trong bảng băm có thể được truy cập nhanh chóng qua khóa đã được chuyển đổi

### Bảng băm

Sau khi hashing, cặp key/value (key đã dược chuyển đổi) được chứa trong một cấu trúc dữ liệu là bảng băm (hash table). Bảng băm (Hash Table) là một cấu trúc dữ liệu lưu giữ dữ liệu theo cách thức liên hợp, lưu trữ theo dạng cặp (key, value). Nó sử dụng hàm băm để tính toán chỉ số (chỉ mục) thông qua key, chỉ số này được dùng cho việc chèn hoặc tìm kiếm dữ liệu được dễ dàng hơn. Với một bảng băm có một hàm băm được thực hiện tốt thì trong trường hợp lý tưởng, việc tìm kiếm các phần tử trong bảng có thời gian là O(1).

Tham khảo:

<https://www.tutorialspoint.com/python_data_structure/python_hash_table.htm>

## Đồ thị - Các thao tác cơ bản

### Giới thiệu về đồ thị

Trong toán học và tin học, **đồ thị** là đối tượng nghiên cứu cơ bản của lý thuyết đồ thị. Một cách không chính thức, đồ thị là một tập các đối tượng gọi là **đỉnh** nối với nhau bởi các **cạnh**. Thông thường, đồ thị được vẽ dưới dạng một tập các điểm (đỉnh, nút) nối với nhau bởi các đoạn thẳng (cạnh). Tùy theo ứng dụng mà một số cạnh có thể có hướng.

Một cách không chính thức, đồ thị là một tập các đối tượng được gọi là các đỉnh nối với nhau bởi các cạnh.

Một đồ thị kí hiệu là: **G = (V, E)**

Trong đó:

+ V là tập các đỉnh của đồ thị. Đỉnh biểu diễn các đối tượng trong đồ thị, thường được đánh dấu bằng các số hoặc kí hiệu bằng các chữ cái in thường u,v,…

+ E là tập các cạnh của đồ thị. Cạnh nối đỉnh *x* với đỉnh *y* là một tập gồm hai phần tử x, y thường được vẽ dưới dạng một *đoạn thẳng* nối hai đỉnh

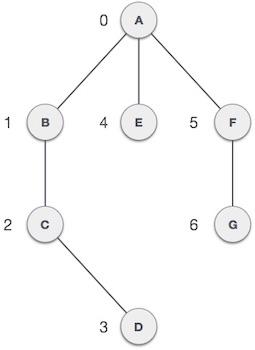
### Cấu trúc dữ liệu đồ thị (Graph)

**Đỉnh (Vertex)**: Mỗi nút của hình được biểu diễn như là một đỉnh. Trong ví dụ dưới đây, các hình tròn biểu diễn các đỉnh. Do đó, các điểm từ A tới G là các đỉnh. Chúng ta có thể biểu diễn các đỉnh này bởi sử dụng một mảng, trong đó đỉnh A có thể được nhận diện bởi chỉ mục 0, điểm B là chỉ mục 1, …

**Cạnh (Edge)**: Cạnh biểu diễn một đường nối hai đỉnh. Trong hình dưới, các đường nối A và B, B và C, … là các cạnh. Chúng ta có thể sử dụng một mảng hai chiều để biểu diễn các cạnh này. Trong ví dụ dưới, AB có thể được biểu diễn như là 1 tại hàng 0; BC là 1 tại hàng 1, cột 2, …

**Kề nhau**: Hai đỉnh là kề nhau nếu chúng được kết nối với nhau thông qua một cạnh. Trong hình dưới, B là kề với A; C là kề với B, …

**Đường**: Đường biểu diễn một dãy các cạnh giữa hai đỉnh. Trong hình dưới, ABCD biểu diễn một đường từ A tới D



### Các thao tác cơ bản trên cấu trúc dữ liệu đồ thị

**Thêm đỉnh**: Thêm một đỉnh vào trong đồ thị

**Thêm cạnh**: Thêm một cạnh vào giữa hai đỉnh của một đồ thị

**Hiển thị đỉnh**: Hiển thị một đỉnh của một đồ thị

### Biểu diễn và kết nối trong đồ thị

Có nhiều cách để biểu diễn đồ thị trên máy tính, tùy thuộc vào tính chất của đồ thị hoặc thuật toán áp dụng với đồ thị… Ta cũng có thể lưu kèm theo các thông tin như trọng số, giá trị phù hợp với từng cạnh.

**Ma trận kề:**

Tạo một ma trận A kích thước n\*n trong đó n là số đỉnh của đồ thị. Ta gán a[u][v] = 0 nếu có cạnh cạnh nối hai đỉnh u, v

Nếu đồ thị là đa đồ thị, ta có thể gán a[u][v] = số cạnh nối u và v

Định nghĩa và gán tùy theo lập trình viên hiểu là vô hướng hay có hướng, đơn đồ thị hay đa đồ thị

+ Ưu điểm: Để kiểm tra hai đỉnh u, v có kề nhau không, ta chỉ cần kiểm tra trong độ phức tạp O(1)

+ Nhược điểm: Dù đồ thị có nhiều cạnh hay ít cạnh thì cũng phải mất n\*n ô nhớ để lưu. Để duyệt tất cả các đỉnh kề với u, ta phải duyệt tất cả các đỉnh v ∊ V cho dù đỉnh u kề với ít hoặc không kề với đỉnh nào khác

* + Biểu diễn bằng ma trận kề thường được dùng khi đồ thị có ít đỉnh, hoặc đồ thị dày, nhiều cạnh, hoặc thuật toán để thao tác trên đồ thị yêu cầu

**Danh sách cạnh:**

Với đồ thị G=(V,E)G=(V,E) có n đỉnh, m cạnh, ta có thể liệt kê tất cả các cạnh của đồ thị bằng một danh sách tương ứng, mỗi phần tử của mảng tương ứng là một cặp (u,v) là một cạnh thuộc E, tùy theo người lập trình định nghĩa là có hướng hay vô hướng

+ Ưu điểm: Với đồ thị thưa, ta chỉ cần mất m (số lượng cạnh) ô nhớ để lưu đồ thị

+ Nhược điểm: Khi cần kiểm tra hai đỉnh u,v có kề nhau hay không, ta không thể kiểm tra nhanh trong //( O(1) //) như cách lưu bằng ma trận kề, mặc dù tùy theo cách lưu danh sách cạnh mà ta có thể kiểm tra trong //( O(logn) //) hoặc ít hơn

**Danh sách kề:**

Với mỗi đỉnh của đồ thị, ta lưu một danh sách các đỉnh kề với đỉnh đó.

+ Ưu điểm: Với phương pháp này, việc duyệt tất cả các đỉnh kề với đỉnh u vô cùng dễ dàng

+ Nhược điểm: Khi cần kiểm tra hai đỉnh u,v có kề nhau hay không, ta không thể kiểm tra nhanh trong O(1)O(1) như cách lưu bằng ma trận kề, mặc dù tùy theo cách lưu danh sách cạnh mà ta có thể kiểm tra trong O(logn)O(logn) hoặc ít hơn

### Đồ thị vô hướng và đồ thị có hướng

**Đồ thị vô hướng** là một đồ thị mà các cạnh của nó không có hướng. Mỗi cạnh luôn là một mối quan hệ hai chiều, và mỗi cạnh có thể được duyệt qua theo hai hướng. **Đồ thị có hướng** là trường hợp ngược lại của đồ thị vô hướng, với các cạnh có hướng, xuất phát từ hoặc kết thúc tại một đỉnh, thông thường ký hiệu bằng dấu mũi tên.

**Đồ thị vô hướng**

**Đồ thị vô hướng** hoặc **đồ thị** *G* là một [cặp không có thứ tự](https://vi.wikipedia.org/w/index.php?title=C%E1%BA%B7p_kh%C3%B4ng_c%C3%B3_th%E1%BB%A9_t%E1%BB%B1&action=edit&redlink=1) *(unordered pair)* *G*:=(*V*, *E*), trong đó

+ *V*, [tập](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_h%E1%BB%A3p) các đỉnh hoặc nút

+ *E*, tập các cặp không thứ tự chứa các đỉnh phân biệt, được gọi là **cạnh**. Hai đỉnh thuộc một cạnh được gọi là các đỉnh đầu cuối của cạnh đó

**Đồ thị có hướng**

Đồ thị có hướng *G* là một cặp có thứ tự *G* = (*V*, *A*), trong đó

+ *V*, [tập](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_h%E1%BB%A3p) các đỉnh hoặc nút

+ *A*, tập các cặp có thứ tự chứa các đỉnh, được gọi là các cạnh có hướng hoặc cung. Một cạnh *e* = (*x*, *y*) được coi là có hướng từ *x* tới *y*; *x* được gọi là điểm đầu/gốc và *y* được gọi là điểm cuối/ngọn của cạnh

**Đồ thị liên thông**

Một đồ thị được gọi là liên thông (connected) nếu có đường đi giữa mọi cặp đỉnh phân biệt của đồ thị. Ngược lại, đồ thị này được gọi là không liên thông (disconnected).

Tham khảo: <https://www.tutorialspoint.com/python_data_structure/python_graphs.htm>

# Thuật toán kinh điển

## Bài toán tuyến đường ít chặng nhất

### Giải thuật tìm kiếm theo chiều rộng (BFS)

Giải thuật tìm kiếm theo chiều rộng (Breadth First Search – viết tắt là BFS) duyệt qua một đồ thị theo chiều rộng và sử dụng hàng đợi (queue) để ghi nhớ đỉnh liền kề để bắt đầu việc tìm kiếm khi không gặp được đỉnh liền kề trong bất kỳ vòng lặp nào

**Qui tắc 1**: Duyệt tiếp tới đỉnh liền kề mà chưa được duyệt. Đánh dấu đỉnh mà đã được duyệt. Hiển thị đỉnh đó và đẩy vào trong một hàng đợi (queue)…

**Qui tắc 2**: Nếu không tìm thấy đỉnh liền kề, thì xóa đỉnh đầu tiên trong hàng đợi

**Qui tắc 3**: Lặp lại Qui tắc 1 và 2 cho tới khi hàng đợi là trống

**Ví dụ về duyệt BFS:**

| Bước | Duyệt | Miêu tả |
| --- | --- | --- |
| 1 | Giải thuật tìm kiếm theo chiều rộng | Khởi tạo hàng đợi (queue) |
| 2 | Giải thuật tìm kiếm theo chiều rộng | Chúng ta bắt đầu duyệt đỉnh **S** (đỉnh bắt đầu) và đánh dấu đỉnh này là **đã duyệt** |
| 3 | Giải thuật tìm kiếm theo chiều rộng | Sau đó chúng ta tìm đỉnh liền kề với **S**mà chưa được duyệt. Trong ví dụ này chúng ta có 3 đỉnh, và theo thứ tự chữ cái chúng ta chọn đỉnh **A** đánh dấu là đã duyệt và xếp **A** vào hàng đợi |
| 4 | Giải thuật tìm kiếm theo chiều rộng | Tiếp tục duyệt đỉnh liền kề với **S** là **B**. Đánh dấu là đã duyệt và xếp đỉnh này vào hàng đợi |
| 5 | Giải thuật tìm kiếm theo chiều rộng | Tiếp tục duyệt đỉnh liền kề với **S** là **C**. Đánh dấu là đã duyệt và xếp đỉnh này vào hàng đợi |
| 6 | Giải thuật tìm kiếm theo chiều rộng | Bây giờ đỉnh **S** không còn đỉnh nào liền kề mà chưa được duyệt. Bây giờ chúng ta rút **A** từ hàng đợi |
| 7 | Giải thuật tìm kiếm theo chiều rộng | Từ đỉnh **A** chúng ta có đỉnh liền kề là **D** và là đỉnh chưa được duyệt. Đánh dấu đỉnh **D** là đã duyệt và xếp vào hàng đợi |

### Giải thuật tìm kiếm theo chiều sâu (DFS)

Giải thuật tìm kiếm theo chiều sâu (Depth First Search – viết tắt là DFS), còn được gọi là giải thuật tìm kiếm ưu tiên chiều sâu, là giải thuật duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị và sử dụng stack (ngăn xếp) để ghi nhớ đỉnh liền kề để bắt đầu việc tìm kiếm khi không gặp được đỉnh liền kề trong bất kỳ vòng lặp nào. Giải thuật tiếp tục cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉ nh vừa mới tìm kiếm ở bước trước

**Qui tắc 1**: Duyệt tiếp tới đỉnh liền kề mà chưa được duyệt. Đánh dấu đỉnh mà đã được duyệt. Hiển thị đỉnh đó và đẩy vào trong một ngăn xếp (stack)

**Qui tắc 2**: Nếu không tìm thấy đỉnh liền kề, thì lấy một đỉnh từ trong ngăn xếp (thao tác pop up). (Giải thuật sẽ lấy tất cả các đỉnh từ trong ngăn xếp mà không có các đỉnh liền kề nào)

**Qui tắc 3**: Lặp lại các qui tắc 1 và qui tắc 2 cho tới khi ngăn xếp là trống

**Ví dụ về duyệt BFS:**

| Bước | Duyệt | Miêu tả |
| --- | --- | --- |
| 1 | Giải thuật tìm kiếm theo chiều sâu | Khởi tạo ngăn xếp (stack) |
| 2 | Giải thuật tìm kiếm theo chiều sâu | Đánh dấu đỉnh **S** là **đã duyệt** và đặt đỉnh này vào trong ngăn xếp. Tìm kiếm bất kỳ đỉnh liền kề nào mà chưa được duyệt từ đỉnh **S**. Chúng ta có 3 đỉnh và chúng ta có thể lấy bất kỳ đỉnh nào trong số chúng. Ví dụ, chúng ta lấy đỉnh **A** theo thứ tự chữ cái |
| 3 | Giải thuật tìm kiếm theo chiều sâu | Đánh dấu đỉnh **A** là đã duyệt và đặt vào trong ngăn xếp. Tìm kiếm bất kỳ đỉnh liền kề nào với đỉnh **A**. Cả **S** và **D** đều là hai đỉnh liền kề **A** nhưng chúng ta chỉ quan tâm về đỉnh chưa được duyệt |
| 4 | Giải thuật tìm kiếm theo chiều sâu | Duyệt đỉnh **D**, đánh dấu đỉnh này là đã duyệt và đặt vào trong ngăn xếp. Ở đây, chúng ta có **B** và **C** là hai đỉnh kề với **D** và cả hai đều là chưa được duyệt. Chúng ta sẽ chọn theo thứ tự chữ cái một lần nữa |
| 5 | Giải thuật tìm kiếm theo chiều sâu | Chọn **B**, đánh dấu là đã duyệt và đặt vào trong ngăn xếp. Ở đây **B** không có bất kỳ đỉnh liền kề nào mà chưa được duyệt. Vì thế chúng ta lấy **B** ra khỏi ngăn xếp |
| 6 | Giải thuật tìm kiếm theo chiều sâu | Kiểm tra phần tử trên cùng của ngăn xếp để trở về nút đã duyệt trước đó và kiểm tra xem đỉnh này có đỉnh nào liền kề mà chưa được duyệt hay không. Ở đây, chúng ta tìm thấy đỉnh **D** nằm ở trên cùng của ngăn xếp |
| 7 | Giải thuật tìm kiếm theo chiều sâu | Chỉ có một đỉnh liền kề với **D** mà chưa được duyệt, đó là đỉnh **C**. Chúng ta duyệt **C**, đánh dấu là đã duyệt và đặt vào trong ngăn xếp |

### So sánh BFS và DFS

BFS tốn bộ nhớ hơn vì nó lưu lại tất cả các con ở mỗi cấp. Tuy nhiên , tùy vào bài toán cụ thể mà BFS hay DFS sẽ “chạy” nhanh hơn

Ví dụ: Ta có 1 cây gia phả qua nhiều thế hệ của một gia tộc lớn

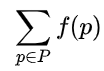
Nếu 1 người nào đó trong gia phả muốn tìm kiếm 1 người nào đó vẫn còn sống thì ta thấy DFS  thường hoạt động nhanh hơn.Tuy nhiên, nếu người đó muốn tìm một người nào đó đã mất cách đây rất lâu thì BFS thường nhanh hơn DFS

* + Cái nào tốt hơn cái nào thì còn tùy thuộc vào dữ liệu và những gì bạn đang tìm kiếm

## Bài toán tuyến đường ngắn nhất

### Tổng quan về bài toán tìm đường ngắn nhất

Trong lý thuyết đồ thị, bài toán đường đi ngắn nhất nguồn đơn là bài toán tìm một đường đi giữa hai đỉnh sao cho tổng các trọng số của các cạnh tạo nên đường đi đó là nhỏ nhất. Định nghĩa một cách hình thức, cho trước một đồ thị có trọng số (nghĩa là một tập đỉnh V, một tập cạnh E, và một hàm trong số có giá trị thực f : E → R), cho trước một đỉnh v thuộc V, tìm một đường đi P từ v tới mỗi đỉnh v' thuộc V sao cho



là nhỏ nhất trong tất cả các đường nối từ v tới v' . Bài toán đường đi ngắn nhất giữa mọi cặp đỉnh là một bài toán tương tự, trong đó ta phải tìm các đường đi ngắn nhất cho mọi cặp đỉnh v và v'

### Thuật toán Dijkstra

Thuật toán Dijkstra, là một thuật toán giải quyết bài toán đường đi ngắn nhất nguồn đơn trong một đồ thị có hướng không có cạnh mang trọng số không âm. Thuật toán thường được sử dụng trong định tuyến với một chương trình con trong các thuật toán đồ thị hay trong công nghệ Hệ thống định vị toàn cầu (GPS)

**Cách hoạt động của Thuật toán Dijkstra**

**Bước 1**: Chọn S = {} là tập các soure\_node bao gồm current\_node và passed\_node . Với current\_node là node đang được xét đến, passed\_node là các node đã được xét. current\_node đầu tiên sẽ là node đích của bài toán tìm đường đi ngắn nhất

**Bước 2**: Khởi tạo giải thuật với current\_node là node đích và cost(N) là giá trị của đường đi ngắn nhất từ N đến node đích

**Bước 3**: Xét các node kề N với current\_node . Gọi d(current\_node,N) là khoảng cách giữa node kề N và current\_node . Với p = d(current\_node,N) + cost (current\_node). Nếu p < cost(N) thì cost(N) = p . Nếu không thì cost(N) giữ nguyên giá trị

**Bước 4**: Sau khi xét hết các node kề N, đánh dấu current\_node thành passed\_node

**Bước 5**: Tìm current\_node mới với 2 điều kiện: không phải passed\_node và cost(current\_node) là nhỏ nhất

**Bước 6**: Nếu tập S = {} chứa đủ các node của đồ thị thì dừng thuật toán. Nếu không thì quay trở lại **Bước 3**

### Thuật toán Bellman – Ford

Thuật toán Bellman-Ford là một thuật toán tính các đường đi ngắn nhất nguồn đơn trong một đồ thị có hướng có trọng số (trong đó một số cung có thể có trọng số âm). Thuật toán Dijkstra giải cùng bài toán này tuy nhiên Dijkstra có thời gian chạy nhanh hơn, đơn giản là đòi hỏi trọng số của các cung phải có giá trị không âm

**Bước 1:** Bắt đầu với đồ thị có trọng số

**Bước 2:** Chọn đỉnh bắt đầu và nhập trọng số là inf cho mọi cạnh trong đồ thị

**Bước 3:** Thăm các cạnh và bỏ qua các cạnh nếu nó không chính xác

**Bước 4:** Ta phải lặp lại V lần (số đỉnh) trong trường hợp xấu nhất, một đường đi của đỉnh có thể cần điều chỉnh lại V lần

**Bước 5:** Để ý đường đi qua mỗi đỉnh đã được điều chỉnh độ dài như thế nào

**Bước 6:** Sauk hi mọi cạnh đã có số đo, ta kiểm tra nếu có chu kỳ âm

### So sánh Bellman Ford với Dijkstra

Dijkstra chỉ nhìn vào các điểm hàng xóm trực tiếp của đỉnh được xét, còn Bellman đi qua từng cạnh qua mỗi lần lặp lại

## Thuật toán KMP (Knuth – Morris – Pratt Algorithm)

### Rabin – Karp

Tư tưởng chính của phương pháp này là sử dụng phương pháp băm (hashing). Tức là mỗi một xâu sẽ được gán với một giá trị của hàm băm (hash function), ví dụ xâu “hello” được gán với giá trị 5 chẳng hạn, và hai xâu được gọi là bằng nhau nếu giá trị băm của nó bằng nhau. Như vậy thay vì việc phải đối sánh các xâu con của y với mẫu x, ta chỉ cần so sánh giá trị hàm băm của chúng và đưa ra kết luận.

### Knuth – Morris – Pratt

Thuật toán KMP được xây dựng dựa vào quan sát rằng một xâu con chung của S và T sẽ đưa ra được thông tin S có khớp với các vị trí sau của T hay không. Bởi vì xâu con chung này đồng nghĩa với một phần của T đã khớp với một phần của S, nên bằng việc khởi tạo trước một số thông tin với xâu S, ta sẽ thu được những kết luận về T (nhờ xâu con chung) mà không cần quay ngược và so sánh lại những ký tự đã khớp.Cụ thể hơn, ta muốn tính toán trước cách xâu S tự khớp với chính nó. Nhờ vậy thuật toán sẽ không quay nhìn lại và chỉ duyệt qua T một lần duy nhất

Cùng với quá trình tiền xử lí tuyến tính O(|T|) (với |T| là độ dài xâu T), thuật toán có thời gian chạy tuyến tính

### So sánh Robin Karp và KMP

Sự khác biệt quan trọng nhất là độ tin cậy giữa 2 thuật toán. KMP đảm bảo độ tin cậy 100% với kết quả đưa ra, trong khi Rabin Karp không đảm bảo điều này bởi vì có thể xảy ra xung đột trong quá trình tra cứu bảng băm. Ngày nay, có rất nhiều hàm băm rất tốt và Rabin Karp cũng có thể đem đến một độ tin cậy tiệm cận 100%. Về mặt triển khai, Rabin Karp dễ thực thi hơn và tốn ít bộ nhớ hơn một chút