ĐẠI HỌC HUẾ



**KHOA KỸ THUẬT VÀ CÔNG NGHỆ**

-----🙞🙜🕮🙞🙜-----



**BÁO CÁO ĐỒ ÁN**

**Học kỳ II, năm học 2022-2023**

**Học phần:**

**Cấu trúc dữ liệu và giải thuật**

**Số phách**

(*do hội đồng chấm thi ghi*)

***Thừa Thiên Huế,ngày … tháng … năm 2023***

ĐẠI HỌC HUẾ



**KHOA KỸ THUẬT VÀ CÔNG NGHỆ**

-----🙞🙜🕮🙞🙜-----

**(Bìa phụ 2)**

**BÁO CÁO ĐỒ ÁN**

**Học kỳ II, năm học 2022-2023**

**Học phần:**

**Cấu trúc dữ liệu và giải thuật**

**Lớp**: AIK3

**Giáo viên hướng dẫn:** Nguyễn Thanh Nam

**Sinh viên thực hiện: Văn Khiêm Chương 22E1020002**

**Số phách**

(*do hội đồng chấm thi ghi*)

***Thừa Thiên Huế,ngày … tháng … năm 2023***

# LỜI CẢM ƠN

Bài báo cáo đồ án cuối học kì II của môn chuyên ngành cấu trúc dữ liệu và giải thuật là kết quả nổ lực không ngừng nghỉ của bản thân và được sự giúp đở thầy bộ môn . Qua bài bài báo cáo này em xin gửi lời cảm ơn tới thầy Nguyền Thanh Nam vì đã giúp đở em trong quá trình hoàn thiện đồ án và giúp em hiểu rỏ về chuyên ngành mình đang theo đuổi, thầy còn giúp em hoàn thiện những kĩ năng còn thiếu của bản thân trong giờ học của thầy.

Em xin chân thành cảm ơn đến lãnh đạo và thầy cô bộ môn của khoa kĩ thuật và công nghệ đã tạo điều kiền cho chúng em có một môi trường học tập đầy mới mẻ và hiện tại.

# DANH MỤC HÌNH ẢNH

# DANH MỤC BẢNG BIỂU

# MỤC LỤC

[LỜI CẢM ƠN i](#_Toc135040502)

[DANH MỤC HÌNH ẢNH ii](#_Toc135040503)

[DANH MỤC BẢNG BIỂU iii](#_Toc135040504)

[MỤC LỤC iv](#_Toc135040505)

[I. GIẢI THUẬT ĐỆ QUY 1](#_Toc135040506)

[BÀI 1: BÀI TOÁN THÁP HÀ NỘI 1](#_Toc135040507)

[1. Mô tả bài toán tháp hà nội 1](#_Toc135040508)

[2. Mã nguồn giải bài toán tháp Hà Nội bằng ngôn ngữ lập trình python 2](#_Toc135040509)

[BÀI 2. ƯỚC SỐ CHUNG LỚN NHẤT 3](#_Toc135040510)

[1. Mô tả bài toán ước chung lớn nhất 3](#_Toc135040511)

[2. Mã nguồn giải bài toán ước chung lớn nhất bằng ngôn ngữ python 3](#_Toc135040512)

[BÀI 3 : TÍNH GIAI THỪA CỦA 1 SỐ 4](#_Toc135040513)

[1. Mô tả bài toán giai thừa của 1 số 4](#_Toc135040514)

[2. Mã nguồn bài toán tính giai thừa của một số bằng ngôn ngữ python 4](#_Toc135040515)

[BÀI 4 :BÀI TOÁN MÃ ĐI TUẦN 5](#_Toc135040516)

[1. Mô tả bài toán mã đi tuần 5](#_Toc135040517)

[2. Mã Nguồn giải bài toán mã đi tuần bằng ngôn ngư python 5](#_Toc135040518)

[BÀI 5 : TÁM QUÂN HẬU 8](#_Toc135040519)

[1. Mô tả bài toán tám quân hậu 8](#_Toc135040520)

[2. Mã nguồn giải bài toán tám quân hậu bằng ngôn ngữ python 8](#_Toc135040521)

[II. DANH SÁCH LIÊN KẾT 10](#_Toc135040522)

[BÀI 1: DANH SÁCH LIÊN KẾT ĐƠN 10](#_Toc135040523)

[1. Nội dung 10](#_Toc135040524)

[2. Ý nghĩa 10](#_Toc135040525)

[3. Cách thức hoạt động của giải thuật 10](#_Toc135040526)

[4. Mã nguồn danh sách liên kết 11](#_Toc135040527)

[BÀI 2: CÀI ĐẶT LIÊN KẾT ĐÔI 14](#_Toc135040528)

[1. Nội dung 14](#_Toc135040529)

[2. Ý nghĩa 14](#_Toc135040530)

[3. Cách thức hoạt động của giải thuật. 14](#_Toc135040531)

[4. Mã nguồn danh sách liên kết đôi 15](#_Toc135040532)

[BÀI 3: CÀI ĐẶT NGĂN XẾP - STACK 18](#_Toc135040533)

[1. Nội dung 18](#_Toc135040534)

[2. Ý nghĩa 18](#_Toc135040535)

[3. Cách thức hoạt động của giải thuật 19](#_Toc135040536)

[4. Mã nguồn cài đặt ngăn xếp - stack 19](#_Toc135040537)

[BÀI 4: Cài đặt hàng đợi – queue 23](#_Toc135040538)

[1. Nội dung 23](#_Toc135040539)

[2. Ý nghĩa 23](#_Toc135040540)

[3. Cách thức hoạt động của giải thuật 24](#_Toc135040541)

[4. Mã nguồn cài đặt hàng đợi-queue 24](#_Toc135040542)

[III. CÂY 26](#_Toc135040543)

[BÀI 1: CÀI ĐẶT CÂY 26](#_Toc135040544)

[1. Nội dung 26](#_Toc135040545)

[2. Ý nghĩa 26](#_Toc135040546)

[3. Cách thức hoạt động 27](#_Toc135040547)

[4. Mã nguồn cài đặt cây trong ngôn ngữ python 27](#_Toc135040548)

[BÀI 2: CÀI ĐẶT CÂY- DUYỆT CÂY THEO THỨ TỰ TRƯỚC 28](#_Toc135040549)

[1. Nội dung 28](#_Toc135040550)

[2. Ý nghĩa 28](#_Toc135040551)

[3. Cách thức hoạt động 29](#_Toc135040552)

[4. Mã nguồn cài đặt cây-duyệt theo thứ tự trước 29](#_Toc135040553)

[BÀI 3: CÀI ĐẶT CÂY- DUYỆT CÂY THEO THỨ TỰ SAU 30](#_Toc135040554)

[1. Nội dung 30](#_Toc135040555)

[2. Ý nghĩa 30](#_Toc135040556)

[3. Cách thức hoạt động 31](#_Toc135040557)

[4. Mã nguồn cài đặt cây-duyệt cây theo thứ tự sau 31](#_Toc135040558)

[IV. ĐỒ THỊ 31](#_Toc135040559)

[BÀI 1: ĐỒ THỊ VÔ HƯỚNG 31](#_Toc135040560)

[1. Nội dung 31](#_Toc135040561)

[2. Ý nghĩa 32](#_Toc135040562)

[3. Mã nguồn cài đặt đồ thị vô hướng trong python: 32](#_Toc135040563)

[BÀI 2: CÀI ĐẶT ĐỒ THỊ CÓ HƯỚNG 32](#_Toc135040564)

[1. Nội dung 32](#_Toc135040565)

[2. Ý nghĩa 33](#_Toc135040566)

[3. Mã nguồn cài đặt đồ thị có hướng 33](#_Toc135040567)

[CHƯƠNG 5: SẮP XẾP 34](#_Toc135040568)

[BÀI 1: SẮP XẾP CHỌN 34](#_Toc135040569)

[1. Nội dung 34](#_Toc135040570)

[2. Ý nghĩa 34](#_Toc135040571)

[3. Mã nguồn sắp xếp chọn 35](#_Toc135040572)

[BÀI 2: SẮP XẾP CHÈN 35](#_Toc135040573)

[1. Nội dung 35](#_Toc135040574)

[2. Ý Nghĩa 36](#_Toc135040575)

[5. Mã nguồn sắp xếp chèn 36](#_Toc135040576)

[BÀI 3: SẮP XẾP NỔI BỌT 37](#_Toc135040577)

[1. Nội dung 37](#_Toc135040578)

[2. Ý Nghĩa 37](#_Toc135040579)

[3. Mã nguồn sắp xếp nổi bọt 37](#_Toc135040580)

[BÀI 4: THUẬT TOÁN SẮP XẾP NHANH-QUICK SORT 38](#_Toc135040581)

[1. Nội dung 38](#_Toc135040582)

[2. Ý nghĩa 39](#_Toc135040583)

[3. Mã Nguồn thuật toán sắp xếp nhanh - quick sort 40](#_Toc135040584)

[BÀI 5: THUẬT TOÁN HEAP SORT 40](#_Toc135040585)

[1.Nội dung 40](#_Toc135040586)

[2. Ý Nghĩa 41](#_Toc135040587)

[3. mã nguồn thuật toán heap sort 41](#_Toc135040588)

[BÀI 6: THUẬT TOÁN SẮP XẾP TRỘN-MERGE SORT 42](#_Toc135040589)

[1. Nội dung 42](#_Toc135040590)

[2. Ý nghĩa 43](#_Toc135040591)

[3. Mã nguồn thuật toán sắp xếp trộn-merge sort 43](#_Toc135040592)

[TÀI LIỆU THAM KHẢO 45](#_Toc135040593)

[KẾT QUẢ KIỂM TRA ĐẠO VĂN 46](#_Toc135040594)

# GIẢI THUẬT ĐỆ QUY

## BÀI 1: BÀI TOÁN THÁP HÀ NỘI

### Mô tả bài toán tháp hà nội

Bài toán Tháp Hà Nội là một bài toán truyền thống trong lĩnh vực toán học và giải đố. Bài toán này bao gồm một cấu trúc bao gồm ba cột, và trên cột thứ nhất được xếp đống đĩa lớn dần từ dưới lên, sao cho đĩa nhỏ nhất ở trên cùng. Nhiệm vụ của bài toán là chuyển tất cả các đĩa từ cột thứ nhất sang cột thứ ba, theo quy tắc mỗi lần chỉ được di chuyển một đĩa và không được đặt một đĩa lớn hơn lên đĩa nhỏ hơn.

Để giải bài toán này, ta có thể sử dụng thuật toán đệ quy. Thuật toán này sẽ được triển khai như sau:

Bước 1: Đặt tất cả các đĩa lên cột thứ nhất theo thứ tự từ lớn đến nhỏ.

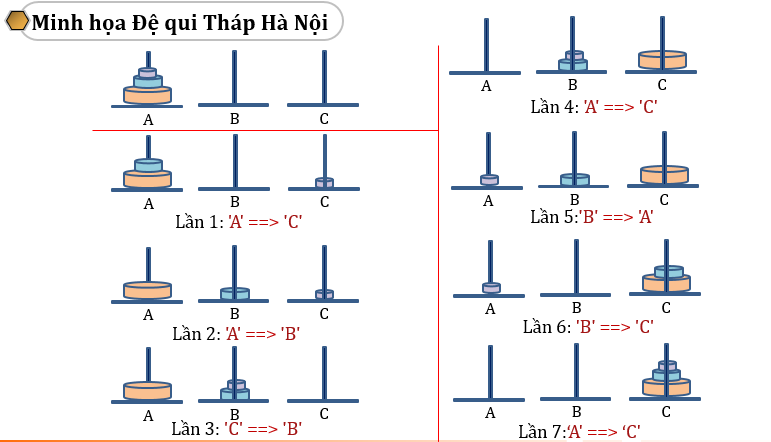
Bước 2: Di chuyển n-1 đĩa từ cột thứ nhất sang cột thứ hai, sử dụng cột thứ ba làm cột trung gian.

Bước 3: Di chuyển đĩa còn lại từ cột thứ nhất sang cột thứ ba.

Bước 4: Di chuyển n-1 đĩa từ cột thứ hai sang cột thứ ba, sử dụng cột thứ nhất làm cột trung gian.

Khi hoàn thành thuật toán này, tất cả các đĩa sẽ được chuyển từ cột thứ nhất sang cột thứ ba theo đúng thứ tự. Bài toán Tháp Hà Nội không chỉ là một bài toán thú vị, mà còn được áp dụng trong nhiều lĩnh vực như lập trình, trí tuệ nhân tạo và tối ưu hóa.

Hình ảnh minh họa của bài toán tháp Hà Nội được biểu diễn ở hình 1.



hình

### Mã nguồn giải bài toán tháp Hà Nội bằng ngôn ngữ lập trình python

def ThapHanoi(n , nguon, dich, phu):

    if n==1:

        print ("di chuyển 1 đĩa từ nguồn",nguon,"đến đích",dich)

        return

    ThapHanoi(n-1, nguon, phu, dich)

    print ("di chuyển đĩa",n,"từ nguồn",nguon,"đến đích",dich)

    ThapHanoi(n-1, phu, dich, nguon)

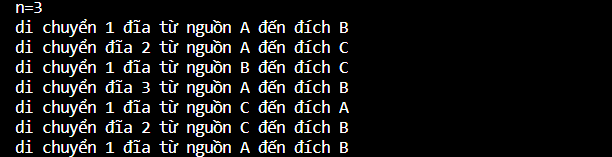
n =int(input('n='))

ThapHanoi(n,'A','B','C')

**Giải thích :**

* ThapHanoi là một hàm đệ quy để giải bài toán Tháp Hà Nội với bốn đối số đầu vào là “ n”,”nguon”,”dich” và “phu”.
* Dòng thứ hai kiểm tra nếu số lượng đĩa cần di chuyển là 1, thì hàm sẽ in ra bước dị chuyển cần thực hiện để di chuyển đĩa đơn từ cột đích và kết thúc hàm bằng câu lệnh “return”.
* Dòng thức ba và thứ tư là đệ quy của hàm “ThapHanoi” với đối số đầu tiên là “n-1”, giảm số lượng đĩa cần di chuyển xuống một và hai đối số tiếp theo lần lượt là cột nguồi, cột phụ và cột đích.
* Dòng thứ 6 yêu cầu người dùng thiết lập số lượng địa cần di chuyển và lưu giá trị đó vào biến “n” bằng hàm “input”.
* Dòng thứ bảy gọi hàm ThapHanoi” với các đối số là số lượng đĩa cần di chuyển, tên cột nguồn, tên cột đích và tên cột phụ.

**Kết quả:**

****

## BÀI 2. ƯỚC SỐ CHUNG LỚN NHẤT

### Mô tả bài toán ước chung lớn nhất

Bài toán ước chung lớn nhất là bài toán tìm số lớn nhất là ước chung của hai hay nhiều số nguyên dương.

Có nhiều phương pháp để tìm ước chung lớn nhất, trong đó có hai phương pháp thông dụng nhất là:

* Phương pháp sử dụng thuật toán Euclid. Đây là phương pháp đơn giản và hiệu quả để tìm ước chung lớn nhất của hai số nguyên dương.Thuật toán Euclid cho phép tìm ước chung lớn nhất bằng cách lập lại việc chia số lớn nhất cho số bé nhất đến khi bằng 0, và kết quả ước chung lớn nhất chính là số lớn nhất còn lại.
* Phương pháp sử dụng phân tích thừa số nguyên tố: đây là phương pháp tìm ước chung lớn nhất bằng cách phân tích hai số thành tích các thừa số nguyên tố và tính ước chung lớn nhất bằng cách lấy tích của các thừa số nguyên tố chung với số mũ nhỏ nhất.

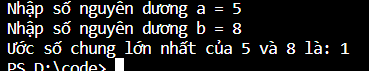
### Mã nguồn giải bài toán ước chung lớn nhất bằng ngôn ngữ python

1. def uscln(a, b):
2. if (b == 0):
3. return a;
4. return uscln(b, a % b);
5. a = int(input("Nhập số nguyên dương a = "));
6. b = int(input("Nhập số nguyên dương b = "));
7. print("Ước số chung lớn nhất của", a, "và", b, "là:", uscln(a, b));

**Giải thích:**

* Hàm uscln(a,b) nhận vào hai số a và b là hai số nguyên dương cần tìm ước chung lớn nhất. Đầu tiên, nó kiểm tra xem giá trị của biến b có bằng 0 hay không.Nếu có, nó trả về giá trị của biến a vì ước chung của a và 0 chính là a.Ngược lại, nó thực hiện lời gọi đế quy với tham số là b và a%b ( phần dư của a khi chia cho b),đến khi b bằng 0 thì kết quả là a.
* Hàng 7,8 là hàm nhạp hai số nguyên dương a và b vào từ bàn phím.
* Hàng 9 gọi hàm uscln(a,b) để tính và in kết quả ước chung lớn nhất của 2 số đó ra màn hình.

**Kết quả:**

****

## BÀI 3 : TÍNH GIAI THỪA CỦA 1 SỐ

### Mô tả bài toán giai thừa của 1 số

Bài toán tính giai thừa của một số là bài toán tìm tích của tất cả các số nguyên dương từ 1 đến số đó. Ký hiệu của giao thừa là dấu chấm than(!).

Để tính giai thừa của một số, có thể sử dụng hai cách:

* Sử dụng vòng lặp:Cách này dùng để tính giai thừa của một số bằng cách sử dụng vòng lặp for để nhân từng số từ 1 đến số đó với nhau.
* Sử dụng đế quy: cách này dùng để tính giao thừa của một số bằng cách sử dụng phép tính đế quy, trong đó ta sẽ giảm số cần tính giao thừa đi 1 và nhân với kết quả của số đô đã được tính giai thừa.

### Mã nguồn bài toán tính giai thừa của một số bằng ngôn ngữ python

* Sử dụng vòng lặp:

def tinh\_giai\_thua(n):

    giai\_thua = 1

    for i in range(1, n + 1):

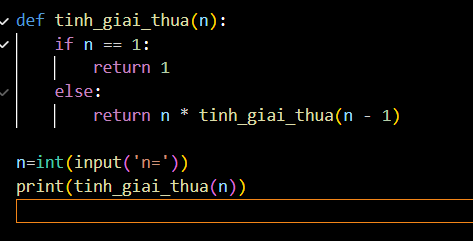
        giai\_thua \*= i

    return giai\_thua

# Ví dụ tính giai thừa của số 5

print(tinh\_giai\_thua(5)) # Kết quả là 120

* Sử dụng đệ quy:



**Kết quả:**

****

## BÀI 4 :BÀI TOÁN MÃ ĐI TUẦN

### Mô tả bài toán mã đi tuần

Bài toán Mã đi tuần (Knight's Tour problem) là bài toán tìm đường đi của quân mã trên bàn cờ vua có kích thước nxn sao cho quân mã đi qua tất cả các ô trên bàn cờ đúng một lần mà không đi qua ô nào hai lần. Bài toán này có nhiều ứng dụng trong lý thuyết đồ thị, tối ưu hóa và trò chơi máy tính.

Cách giải bài toán Mã đi tuần thường sử dụng thuật toán quay lui (backtracking algorithm), tìm kiếm đệ quy, và có thể áp dụng thêm các kỹ thuật tối ưu hóa như cắt tỉa (pruning) để giảm số lần duyệt các ô trên bàn cờ.

Thuật toán quay lui được sử dụng để duyệt từng ô trên bàn cờ và đặt quân mã vào mỗi ô. Sau đó, thuật toán kiểm tra xem ô đó đã được đi qua chưa. Nếu chưa, nó tiếp tục đặt quân mã vào ô đó và tiếp tục đi tiếp theo. Nếu tất cả các ô đều đã được đi qua, thuật toán sẽ trả về một đường đi hợp lệ của quân mã trên bàn cờ. Nếu không, thuật toán sẽ quay lui, trở lại ô trước đó và thử các vị trí khác cho quân mã

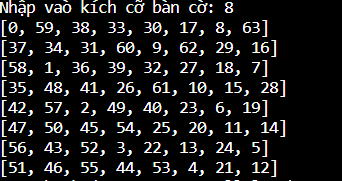
Các kỹ thuật tối ưu hóa được áp dụng để giảm số lần duyệt các ô trên bàn cờ và tăng tốc độ thực thi của thuật toán, bao gồm:

* Sắp xếp các ô tiếp theo theo thứ tự tăng dần số ô mà nó có thể đi tiếp để thuật toán có thể tối ưu hóa quyết định đi tới ô nào tiếp theo.
* Dùng quay lui ngược lại sẽ tiết kiệm thời gian thực hiện so với duyệt theo chiều thuận.
* Áp dụng cắt tỉa, chẳng hạn như cắt tỉa đi các nút trong cây tìm kiếm đệ quy mà không thể dẫn đến giải pháp.

### Mã Nguồn giải bài toán mã đi tuần bằng ngôn ngư python

1. def is\_valid\_move(x, y, sol, N):
2. """
3. Kiểm tra nước đi có hợp lệ hay không
4. """
5. return x >= 0 and y >= 0 and x < N and y < N and sol[x][y] == -1
6. def print\_solution(sol):
7. """
8. In ra ma trận giải pháp
9. """
10. for row in sol:
11. print(row)
12. def solve\_knight\_tour(N):
13. """
14. Tìm giải pháp cho bài toán mã đi tuần bằng backtracking
15. """
16. # Khởi tạo ma trận giải pháp
17. sol = [[-1 for \_ in range(N)] for \_ in range(N)]
18. # Khởi tạo vị trí bắt đầu là (0, 0)
19. sol[0][0] = 0
20. # Khởi tạo 2 mảng lưu tọa độ các bước đi của mã
21. x\_move = [2, 1, -1, -2, -2, -1, 1, 2]
22. y\_move = [1, 2, 2, 1, -1, -2, -2, -1]
23. # Bắt đầu tìm kiếm giải pháp
24. if not solve\_knight\_tour\_util(0, 0, 1, sol, x\_move, y\_move, N):
25. print("Không tìm thấy giải pháp")
26. else:
27. print\_solution(sol)
28. def solve\_knight\_tour\_util(x, y, move\_num, sol, x\_move, y\_move, N):
29. """
30. Hàm đệ quy để tìm kiếm giải pháp
31. """
32. # Kiểm tra xem đã đi hết các ô trên bàn cờ chưa
33. if move\_num == N \* N:
34. return True
35. # Duyệt qua các bước đi của mã
36. for i in range(8):
37. next\_x = x + x\_move[i]
38. next\_y = y + y\_move[i]
39. # Kiểm tra nước đi có hợp lệ hay không
40. if is\_valid\_move(next\_x, next\_y, sol, N):
41. # Đánh dấu ô hiện tại là đã đi qua
42. sol[next\_x][next\_y] = move\_num
43. # Gọi đệ quy để tìm giải pháp tiếp theo
44. if solve\_knight\_tour\_util(next\_x, next\_y, move\_num + 1, sol, x\_move, y\_move, N):
45. return True
46. # Nếu không tìm thấy giải pháp, trả lại giá trị ban đầu cho ô hiện tại
47. sol[next\_x][next\_y] = -1
48. # Nếu không có bước đi nào hợp lệ, trả về False
49. return False
50. n=int(input("Nhập vaò kích cỡ bàn cờ: "))
51. solve\_knight\_tour(n)
52. def main():
53. is\_valid\_move()
54. print\_solution()
55. solve\_knight\_tour()
56. solve\_knight\_tour\_util()
57. if \_\_name\_\_=='\_\_main\_\_':
58. main()

**Kết quả:**

****

## BÀI 5 : TÁM QUÂN HẬU

### Mô tả bài toán tám quân hậu

Bài toán tám quân hậu là một bài toán cổ điển trong lý thuyết trò chơi và lập trình máy tính. Bài toán yêu cầu đặt quân hậu trên một bàn cờ vui kích thước 8\*8 sao cho không có hai quân hậu nào có thể ăn được nhau theo quy tắc cờ vui.

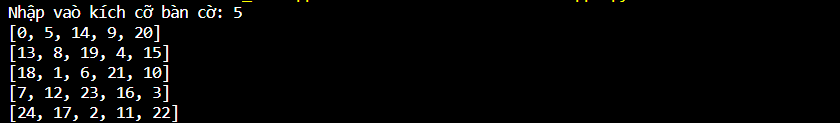
Về mặt hình thức, bài toán tám quân hậu là bài toán tìm kiếm giải pháp trong không gian tọa độ, trong đó mỗi quân hậu được đặt một hàng và một cột duy nhất. Để giải quyết bài toán, ta cần tìm cách xắp xếp các quân hậu sao cho không có hai quân hậu nào cùng nằm trên một hàng, một cột hoặc một đường chéo.

Bài toán tám quân hậu có tính thực tiễn cao và được ứng dụng rộng rãi trong lĩnh vực như máy tính, trí tuệ nhân tạo, kỹ thuật chế tạo và cả trong việc giải quyết các vấn đề thực tiễn. Ngoài ra, bài toán cũng là một bài toán thách thức đối với các nhà khoa học và những người yêu thích trò chơi logic.

### Mã nguồn giải bài toán tám quân hậu bằng ngôn ngữ python

1. def is\_valid\_move(x, y, sol, N):
2. """
3. Kiểm tra nước đi có hợp lệ hay không
4. """
5. return x >= 0 and y >= 0 and x < N and y < N and sol[x][y] == -1
6. def print\_solution(sol):
7. """
8. In ra ma trận giải pháp
9. """
10. for row in sol:
11. print(row)
12. def solve\_knight\_tour(N):
13. """
14. Tìm giải pháp cho bài toán mã đi tuần bằng backtracking
15. """
16. # Khởi tạo ma trận giải pháp
17. sol = [[-1 for \_ in range(N)] for \_ in range(N)]
18. # Khởi tạo vị trí bắt đầu là (0, 0)
19. sol[0][0] = 0
20. # Khởi tạo 2 mảng lưu tọa độ các bước đi của mã
21. x\_move = [2, 1, -1, -2, -2, -1, 1, 2]
22. y\_move = [1, 2, 2, 1, -1, -2, -2, -1]
23. # Bắt đầu tìm kiếm giải pháp
24. if not solve\_knight\_tour\_util(0, 0, 1, sol, x\_move, y\_move, N):
25. print("Không tìm thấy giải pháp")
26. else:
27. print\_solution(sol)
28. def solve\_knight\_tour\_util(x, y, move\_num, sol, x\_move, y\_move, N):
29. """
30. Hàm đệ quy để tìm kiếm giải pháp
31. """
32. # Kiểm tra xem đã đi hết các ô trên bàn cờ chưa
33. if move\_num == N \* N:
34. return True
35. # Duyệt qua các bước đi của mã
36. for i in range(8):
37. next\_x = x + x\_move[i]
38. next\_y = y + y\_move[i]
39. # Kiểm tra nước đi có hợp lệ hay không
40. if is\_valid\_move(next\_x, next\_y, sol, N):
41. # Đánh dấu ô hiện tại là đã đi qua
42. sol[next\_x][next\_y] = move\_num
43. # Gọi đệ quy để tìm giải pháp tiếp theo
44. if solve\_knight\_tour\_util(next\_x, next\_y, move\_num + 1, sol, x\_move, y\_move, N):
45. return True
46. # Nếu không tìm thấy giải pháp, trả lại giá trị ban đầu cho ô hiện tại
47. sol[next\_x][next\_y] = -1
48. # Nếu không có bước đi nào hợp lệ, trả về False
49. return False
50. n=int(input("Nhập vaò kích cỡ bàn cờ: "))
51. solve\_knight\_tour(n)
52. def main():
53. is\_valid\_move()
54. print\_solution()
55. solve\_knight\_tour()
56. solve\_knight\_tour\_util()
57. if \_\_name\_\_=='\_\_main\_\_':
58. main()

**Kết quả:**

****

# DANH SÁCH LIÊN KẾT

## BÀI 1: DANH SÁCH LIÊN KẾT ĐƠN

### Nội dung

Cài đặt danh sách liên kết đơn là một trong những cấu trúc dữ liệu cơ bản trong lập trình. Các thành phần chính của danh sách liên kết đơn bao gồm các nút (node) và các con trỏ (pointer)

Mỗi nút chứa một giá trị dữ liệu và một con trỏ trỏ tới nút tiếp theo trong danh sách. Nút cuối cùng của danh sách sẽ có con trỏ trỏ đến giá trị null.

### Ý nghĩa

Cài đặt danh sách liên kết đơn là một phương pháp để lưu trữ và quản lý các dữ liệu trong lập trình. Với cách thức hoạt động của nó, danh sách liên kết đơn có thể được sử dụng để giải quyết nhiều bài toán thực tế, từ các bài toán đơn giản như tìm kiếm và thêm/xóa phần tử đến các bài toán phức tạp hơn như tìm kiếm đường đi trong đồ thị hoặc sử dụng trong các thuật toán như thuật toán sắp xếp và tìm kiếm. Cài đặt danh sách liên kết đơn cũng cung cấp khả năng linh hoạt hơn so với các cấu trúc dữ liệu khác, bởi vì nó có thể được sử dụng để lưu trữ dữ liệu có kích thước khác nhau và có thể được mở rộng một cách dễ dàng. Do đó, cài đặt danh sách liên kết đơn là một phương tiện mạnh mẽ để xây dựng các chương trình có tính tái sử dụng cao và có khả năng mở rộng.

### Cách thức hoạt động của giải thuật

Cách hoạt động của giải thuật

Giải thuật danh sách liên kết đơn hoạt động dựa trên các phương thức để thêm, xóa và truy xuất các phần tử trong danh sách.

1.Thêm phần tử vào đầu danh sách: Khi muốn thêm một phần tử vào đầu danh sách, giải thuật sẽ tạo một nút mới, gán giá trị cho nút đó và thay đổi con trỏ trỏ đến nút tiếp theo của nút mới này bằng con trỏ của nút đầu tiên hiện tại. Sau đó, con trỏ của nút đầu tiên sẽ được thay đổi để trỏ đến nút mới. Điều này sẽ làm thay đổi kết nối giữa các nút trong danh sách.

2.Thêm phần tử vào cuối danh sách: Khi muốn thêm một phần tử vào cuối danh sách, giải thuật sẽ tạo một nút mới, gán giá trị cho nút đó và tìm nút cuối cùng trong danh sách. Sau đó, con trỏ của nút cuối cùng sẽ được thay đổi để trỏ đến nút mới. Nút mới này sẽ là nút cuối cùng của danh sách.

3.Xóa phần tử khỏi danh sách: Khi muốn xóa một phần tử khỏi danh sách, giải thuật sẽ tìm kiếm phần tử đó trong danh sách và thay đổi kết nối giữa các nút để bỏ qua phần tử đó.

4.Truy xuất phần tử trong danh sách: Khi muốn truy xuất một phần tử trong danh sách, giải thuật sẽ bắt đầu từ nút đầu tiên của danh sách và duyệt từng nút cho đến khi tìm thấy phần tử cần truy xuất hoặc cho đến khi duyệt hết danh sách.

Giải thuật danh sách liên kết đơn có độ phức tạp thời gian O(n) để truy xuất, thêm hoặc xóa phần tử trong danh sách. Tuy nhiên, giải thuật này có thể được tối ưu hơn bằng cách sử dụng các biến để lưu trữ tham chiếu đến nút cuối cùng hoặc nút đầu tiên của danh sách, để giảm độ phức tạp của các thao tác truy xuất, thêm hoặc xóa phần tử.

### Mã nguồn danh sách liên kết

# lớp node

class Node:

    # chức năng khởi tao đối tượng node

    def \_\_init\_\_(self, data):

        self.data = data  # gán dữ liệu

        self.next = None  # khởi tạo

                          # tiếp theo là null null

# lớp danh sách liên kết

class danhsachlienket:

    # chức năng khởi tạo liên kết

    # danh sách đối tượng

    def \_\_init\_\_(self):

        self.head = None

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def add\_to\_head(self, data):

        new\_node = Node(data)

        new\_node.next = self.head

        self.head = new\_node

    def add\_to\_tail(self, data):

        new\_node = Node(data)

        if self.head is None:

            self.head = new\_node

        else:

            current = self.head

            while current.next is not None:

                current = current.next

            current.next = new\_node

    def remove(self, data):

        if self.head is None:

            return

        if self.head.data == data:

            self.head = self.head.next

            return

        current = self.head

        while current.next is not None:

            if current.next.data == data:

                current.next = current.next.next

                return

            current = current.next

    def search(self, data):

        current = self.head

        while current is not None:

            if current.data == data:

                return True

            current = current.next

        return False

# Tạo danh sách liên kết đơn mới

llist = LinkedList()

# Thêm phần tử vào đầu danh sách

llist.add\_to\_head(input("Phần tử đầu danh sách bạn chọn là: "))

llist.add\_to\_head(input("Phần tử đầu danh sách bạn chọn là: "))

llist.add\_to\_head(input("Phần tử đầu danh sách bạn chọn là: "))

# Thêm phần tử vào cuối danh sách

llist.add\_to\_tail(input("Phần tử cuối danh sách bạn chọn là: "))

llist.add\_to\_tail(input("Phần tử cuối danh sách bạn chọn là: "))

llist.add\_to\_tail(input("Phần tử cuối danh sách bạn chọn là: "))

# Xóa phần tử khỏi danh sách

llist.remove(input("Xóa phần tử: "))

# In kết quả

print("Kết quả sau khi thực hiện thao tác được trả về như sau:")

current = llist.head

while current is not None:

    print(current.data)

    current = current.next

# Kiểm tra phần tử có tồn tại trong danh sách hay không

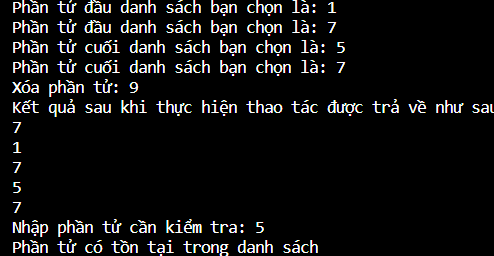
if llist.search(input("Nhập phần tử cần kiểm tra: ")):

    print(f"Phần tử có tồn tại trong danh sách")

else:

    print("Phần tử không tồn tại trong danh sách")

Kết quả:



## BÀI 2: CÀI ĐẶT LIÊN KẾT ĐÔI

### 1. Nội dung

Cài đặt liên kết đôi (double linked list) là một cấu trúc dữ liệu phổ biến trong lập trình. Nó được sử dụng để lưu trữ một danh sách các phần tử dữ liệu, trong đó mỗi phần tử bao gồm dữ liệu và hai con trỏ, một trỏ trỏ tới phần tử phía trước và một trỏ trỏ tới phần tử phía sau. Điều này cho phép ta dễ dàng thêm, xóa và duyệt các phần tử trong danh sách mà không cần phải tạo lại cấu trúc của toàn bộ danh sách.

Với cài đặt liên kết đôi, ta có thể thêm hoặc xóa một phần tử bất kỳ trong danh sách chỉ trong thời gian O(1), còn thời gian truy cập phần tử bất kỳ trong danh sách là O(n). Điều này giúp cải thiện hiệu suất của các thuật toán sử dụng danh sách, đặc biệt là trong các trường hợp cần thực hiện nhiều thao tác thêm/xóa phần tử.

Ví dụ về ứng dụng của cài đặt liên kết đôi là trong việc lưu trữ các bản ghi dữ liệu của một ứng dụng, ví dụ như lịch sử duyệt web. Bằng cách sử dụng cấu trúc liên kết kép, ta có thể dễ dàng thêm và xóa các bản ghi mới/ cũ mà không cần phải duyệt qua toàn bộ lịch sử từ đầu đến cuối.

### 2. Ý nghĩa

Ý nghĩa của cài đặt liên kết đôi trong lập trình là cung cấp một cấu trúc dữ liệu linh hoạt và hiệu quả cho việc lưu trữ và quản lý danh sách các phần tử dữ liệu.

Các phần tử trong danh sách được liên kết với nhau thông qua các con trỏ, giúp ta dễ dàng thêm, xóa và duyệt các phần tử trong danh sách mà không cần phải thay đổi cấu trúc của toàn bộ danh sách. Với cấu trúc này, ta có thể thực hiện các thao tác thêm/xóa phần tử một cách nhanh chóng và hiệu quả hơn so với các cấu trúc dữ liệu khác.

Cài đặt liên kết đôi được sử dụng rộng rãi trong các ứng dụng lưu trữ và quản lý dữ liệu như cơ sở dữ liệu, bộ nhớ đệm (cache), các ứng dụng xử lý ảnh và video, và các trình duyệt web.

### 3. Cách thức hoạt động của giải thuật.

Liên kết đôi là một cấu trúc dữ liệu trong đó mỗi nút (node) trong danh sách chứa dữ liệu và liên kết với hai nút khác, một phía trước và một phía sau. Giải thuật trong liên kết đôi thường được sử dụng để thực hiện các thao tác trên danh sách như thêm, xóa, tìm kiếm và duyệt danh sách.

Một số giải thuật phổ biến trong liên kết đôi bao gồm:

* Thêm nút mới vào đầu danh sách: để thêm một nút mới vào đầu danh sách, ta tạo một nút mới và gán giá trị cho nó, sau đó gán nút phía trước của nó là nút hiện tại đầu danh sách. Tiếp theo, gán nút phía trước của nút đầu danh sách hiện tại là nút mới. Cuối cùng, gán nút đầu danh sách hiện tại là nút mới.
* Xóa nút khỏi danh sách: để xóa một nút khỏi danh sách, ta chỉ cần điều chỉnh các liên kết của nút trước và sau nút cần xóa để bỏ qua nút đó. Sau đó, ta có thể giải phóng bộ nhớ được cấp phát cho nút bị xóa.
* Tìm kiếm nút trong danh sách: để tìm kiếm một nút trong danh sách, ta bắt đầu từ nút đầu danh sách và di chuyển qua các nút kế tiếp cho đến khi tìm thấy nút có giá trị mong muốn hoặc đến khi đến cuối danh sách. Nếu nút được tìm thấy, ta có thể trả về giá trị của nó hoặc chỉ định vị trí của nó trong danh sách.
* Duyệt danh sách: để duyệt danh sách, ta bắt đầu từ nút đầu danh sách và di chuyển qua các nút kế tiếp cho đến khi đến cuối danh sách. Khi duyệt qua từng nút, ta có thể thực hiện một số thao tác trên giá trị của nút đó, như in ra giá trị hoặc thực hiện một tính toán nào đó.

### Mã nguồn danh sách liên kết đôi

#Tạo 1 lớp để tạo 1 nút trong danh sách được liên kết

class Node:

    def \_\_init\_\_ (self, data):

        self.item = data    #Biến item: lưu phần tử thực tế của Node

        self.next = None    #Biến next: lưu trữ địa chỉ đến nút tiếp theo

        self.prev = None    #Biến prev: lưu trữ địa chỉ đến nút trước đó trong danh sách được liên kết kép

#Tạo một lớp doublyLinkedList

#Chứa các hàm để chèn, xóa và hiển thị các phần tử của danh sách được liên kết kép

class doublyLinkedList:

    def \_\_init\_\_(self):

        self.start\_node = None

#Chúng ta thêm các hàm sau vào lớp doublyLinkedList

    #1. Chèn các mục vào danh sách trống

    def InsertToEmptyList (self, data):

        if self.start\_node is None:

            new\_node = Node(data)

            self.start\_node = new\_node

        else:

            print("The list is empty")

    #2. Chèn các mục ở cuối

    def InsertToEnd (self, data):

        #Kiểm tra nếu danh sách trống

        if self.start\_node is None:

            new\_node = Node(data)

            self.start\_node = new\_node

            return

        n = self.start\_node

        #Lặp lại cho đến khi gặp NULL tiếp theo

        while n.next is not None:

            n = n.next

        new\_node = Node(data)

        n.next = new\_node

        new\_node.prev = n

    #3. Xóa các phần tử khỏi màn hình bắt đầu

    def DeleteAtStart (self):

        if self.start\_node is None:

            print("The Linked list is empty, no element to delete")

            return

        if self.start\_node.next is None:

            self.start\_node = None

            return

        self.start\_node = self.start\_node.next

        self.start\_prev = None;

    #4. Xóa các yếu tố từ cuối

    def delete\_at\_end (self):

        # Check if the List is empty

        if self.start\_node is None:

            print("The Linked list is empty, no element to delete")

            return

        if self.start\_node.next is None:

            self.start\_node = None

            return

        n = self.start\_node

        while n.next is not None:

            n = n.next

        n.prev.next = None

    #5. Đi qua danh sách được liên kết

    def Display (self):

        if self.start\_node is None:

            print("The list is empty")

            return

        else:

            n = self.start\_node

            while n is not None:

                print("Element is: ", n.item)

                n = n.next

        print("\n")

#Tạo 1 danh sách liên kết đôi

NewDoublyLinkedList = doublyLinkedList()

#Chèn phần tử vào danh sách rỗng

NewDoublyLinkedList.InsertToEmptyList(10)

#Chèn phần tử vào cuối

NewDoublyLinkedList.InsertToEnd(20)

NewDoublyLinkedList.InsertToEnd(30)

NewDoublyLinkedList.InsertToEnd(40)

NewDoublyLinkedList.InsertToEnd(50)

NewDoublyLinkedList.InsertToEnd(60)

#Hiển thị dữ liệu

NewDoublyLinkedList.Display()

#Xóa các phần tử từ đầu

NewDoublyLinkedList.DeleteAtStart()

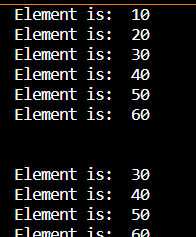
#Xóa các phần tử từ cuối

NewDoublyLinkedList.DeleteAtStart()

#Hiển thị dữ liệu

NewDoublyLinkedList.Display()

**Kết quả:**



## BÀI 3: CÀI ĐẶT NGĂN XẾP - STACK

### 1. Nội dung

Ngăn xếp (Stack) là một cấu trúc dữ liệu tập hợp các phần tử giống như một danh sách, tuy nhiên nó có tính chất hoạt động đặc biệt. Các phần tử được thêm vào và lấy ra từ ngăn xếp theo quy tắc "LIFO" (Last In First Out) - phần tử được thêm vào cuối cùng sẽ được lấy ra đầu tiên.

Về mặt cấu trúc, ngăn xếp có hai thao tác chính:

1. Push: Thêm một phần tử vào đầu ngăn xếp. Quy trình này còn được gọi là đẩy (pushing) phần tử vào ngăn xếp.
2. Pop: Lấy ra phần tử trên đầu ngăn xếp. Quy trình này còn được gọi là rút (popping) phần tử ra khỏi ngăn xếp.

Một số thao tác khác cũng có thể được thực hiện trên ngăn xếp, bao gồm:

1. Peek: Truy cập phần tử đầu tiên trong ngăn xếp mà không xóa nó khỏi ngăn xếp.
2. IsEmpty: Kiểm tra xem ngăn xếp có rỗng hay không.
3. Size: Trả về số lượng phần tử hiện có trong ngăn xếp.

Ngăn xếp thường được sử dụng trong các thuật toán đệ quy, giải thuật phân tích cú pháp, tìm kiếm theo độ sâu, v.v.

Top of Form

### 2. Ý nghĩa

Ngăn xếp (Stack) có rất nhiều ứng dụng trong lập trình và các lĩnh vực khác. Dưới đây là một số ví dụ về ý nghĩa của ngăn xếp:

1. Lập trình đệ quy: Trong đệ quy, một hàm có thể gọi chính nó. Khi điều này xảy ra, các biến cục bộ và trạng thái của hàm cần được lưu trữ tạm thời. Ngăn xếp được sử dụng để lưu trữ các giá trị này trong quá trình thực hiện đệ quy.
2. Phân tích cú pháp: Khi phân tích cú pháp một đoạn mã, ngăn xếp được sử dụng để lưu trữ các thành phần phù hợp của đoạn mã. Điều này giúp cho việc phân tích cú pháp trở nên đơn giản hơn.
3. Lịch sử hoạt động của trình duyệt: Khi bạn truy cập vào một trang web, trình duyệt của bạn thực hiện nhiều thao tác theo trình tự nhất định. Các thao tác này có thể được lưu trữ trong ngăn xếp để bạn có thể quay lại trang trước đó bằng cách sử dụng nút "Quay lại".
4. Giải thuật tìm kiếm theo độ sâu: Trong giải thuật tìm kiếm theo độ sâu (Depth-First Search), các nút trên đường đi đến đích được lưu trữ trong ngăn xếp. Khi một đường không còn thể đi được nữa, ngăn xếp được sử dụng để lấy lại nút trước đó để tìm kiếm các đường khác.
5. Các ứng dụng khác: Ngăn xếp còn được sử dụng trong rất nhiều ứng dụng khác như các hệ thống Undo/Redo, xử lý biểu thức toán học, v.v.

### 3. Cách thức hoạt động của giải thuật

Ngăn xếp (stack) là một cấu trúc dữ liệu được sử dụng để lưu trữ và quản lý dữ liệu theo cơ chế "First-In, Last-Out" (FILO), có nghĩa là phần tử cuối cùng được đưa vào ngăn xếp sẽ được lấy ra đầu tiên. Một cách để thực hiện ngăn xếp là sử dụng giải thuật đệ quy.

Giải thuật đệ quy sử dụng ngăn xếp để thực hiện việc gọi hàm đệ quy. Khi một hàm được gọi, các biến cục bộ và các giá trị đối số của hàm sẽ được đưa vào ngăn xếp. Sau đó, hàm sẽ thực hiện một số lệnh và gọi lại chính nó để xử lý các giá trị đầu vào khác. Khi hàm gọi lại chính nó, các giá trị đầu vào mới cũng sẽ được đưa vào ngăn xếp. Quá trình này sẽ tiếp tục cho đến khi điều kiện dừng của hàm được đáp ứng.

Khi điều kiện dừng được đáp ứng, giá trị kết quả của hàm được trả về và ngăn xếp được giải phóng theo cơ chế LIFO. Khi đó, hàm gọi trước đó sẽ lấy các giá trị đầu ra từ hàm trên ngăn xếp và sử dụng chúng để thực hiện các phép toán khác.

### Mã nguồn cài đặt ngăn xếp - stack

# định nghĩa 1 lớp (Class) Node để biểu diễn 1 nút(Node) trong danh sách liên kết đơn

class Node:

    # hàm khởi tạo init của lớp Node nhận vào tham số data

    def \_\_init\_\_(self, data):

        # lưu trữ giá trị của data vào biến self.data của nút(Node) hiện tại

        self.data = data

        # ban đầu gía trị của self.next sẽ được gán bằng None nghĩa là ở nút hiện tại cũng là nút cuối cùng của danh sách

        # sau khi có 1 nút mới được thêm vào danh sách thì nút trước nó sẽ thay đổi để trỏ đến nút mới và tham chiếu "next" của nút mới cũng được thiết lập thành None

        # các nút trong danh sách được kết nối với nhau thông qua các tham chiếu "next" của chúng

        self.next = None

# định nghĩa 1 lớp (Stack) với 2 phương thức là khởi tạo và 1 thuộc tính là "top"

class Stack:

    # hàm khởi tạo init của lớp Stack

    def \_\_init\_\_(self):

        # thuộc tính top được khởi tạo bằng giá trị None đại diện cho trạng thái ban đầu của Stack là rỗng

        self.top = None

    # hàm kiểm tra xem ngăn xếp có trống hay không

    def is\_empty(self):

        # self.top là phần tử đầu tiên của stack, nếu nó không tồn tại thì ngăn xếp đang rỗng

        # nếu self.top là "None" tức không có phần tử nào trong Stack thì hàm trả giá trị True , ngược lại False

        if self.top is None:

            return True

        return False

    # hàm thêm phần tử mới vào đỉnh của ngăn xếp, trong đó x là giá trị của phần tử mới

    def push(self, x):

        new\_node = Node(x)

    # kiểm tra xem nút mới có được khởi tạo thành công hay không (không bị None)

    #  nếu nút mới không bị None(khởi tạo thành công) thì thực hiện các bước tiếp theo để thêm nó vào danh sách

        if new\_node is not None:

            # kiểm tra xem danh sách có bị rỗng hay không ("top" = "None")

            # nếu rỗng thì gán "new \_node" cho "top" để khởi tạo danh sách

            if self.is\_empty():

                self.top = new\_node

            # nếu không rỗng thì đặt"new\_node.next" bằng"top" để nối new\_node với phần tử đầu tiên ở hiện tại

            # rồi gán "top" cho "new\_node" để cập nhật phần tử đầu tiên của danh sách

            else:

                new\_node.next = self.top

                self.top = new\_node

    # hàm dùng để loại bỏ phần tử đầu tiên khỏi Stack và trả về giá trị của phần tử đó

    def pop(self):

        # kiểm tra xem danh sách có trống không, nếu không thì tiếp tục, ngược lại thì nó sẽ trả giá trị None

        if not self.is\_empty():

            # gán giá trị biến p cho phần tử trên cùng của Stack, nghĩa là phần tử sẽ bị loại bỏ

            p = self.top

            # đoạn code có nhiệm vụ thay đổi từ node đầu tiên sang node tiếp theo trong danh sách liên kết

            self.top = p.next

            # lưu trữ giá trị của Node hiện tại ( nút đầu tiên của Stack) vào biến x

            x = p.data

            # xóa nút đầu tiên ra khỏi danh sách

            del p

            # trả về giá trị của nút vừa bị xóa

            return x

    # hàm nhập vào số n từ bàn phím và thêm vào đầu của danh sách

    def input(self, n):

        # sử dụng vòng lặp để lặp qua từng giá trị "i" trong khoảng từ 0 đến n - 1

        for i in range(n):

            # yêu cầu người dùng nhập giá trị số từ bàn phím

            x = int(input("NHAP SO THU " + str(i) + ": "))

            # sau khi đã nhập xong, sử dụng phương thức "push" để thêm giá trị vừa nhập đó vào đầu của ngăn xếp

            self.push(x)

    # hàm dùng để in ra các phần tử trong ngăn, bắt đầu từ phần tử đầu tiên đến phần tử cuối cùng

    def output(self):

        # tạo biến "p" để duyệt qua các phần tử đã có. lúc đầu "p" được gán bằng giá trị của phần tử đầu tiên

        p = self.top

        # vòng lặp sẽ tiếp tục chạy cho đến khi biến "p" trỏ đến "None",tức là đã duyệt hết các phần tử trong ngăn

        while p is not None:

            # in ra giá trị của phần tử hiện tại mà biến "p" đang trỏ tới

            print(p.data)

            # di chuyển biến "p" đến phần tử tiếp theo trong ngăn

            p = p.next

# khởi tạo đối tượng Stack mới và gán vào biến "s"

s = Stack()

# nhập vào số phần tử cần thêm vào

n = int(input("nhập n "))

# gọi phương thức input của "s" để thêm "n" phần tử vào

s.input(n)

# in ra màn hình

print("danh sách các phần tử vừa được nhập vào STACK")

# gọi phương thức output của "s" để in ra các phần tử trong Stack

s.output()

# yêu cầu người dùng nhập vào phần tử họ cần

x = int(input("nhập phần tử muốn thêm vào"))

# thêm phần tử vừa mới được nhập vào vào Stack

s.push(x)

# in ra màn hình

print("STACK sau khi được thêm phần tử", x)

# câu lệnh dùng để in ra các phần tử trong Stack bắt đầu từ phần tử đầu tiên và kết thúc với phần tử cuối cùng

s.output()

# câu lệnh dùng để lấy ra phần tử trên đỉnh của Stack và đồng thời cũng xóa phần tử đó khỏi Stack "s" gán cho "p"

p = s.pop()

# in ra màn hình

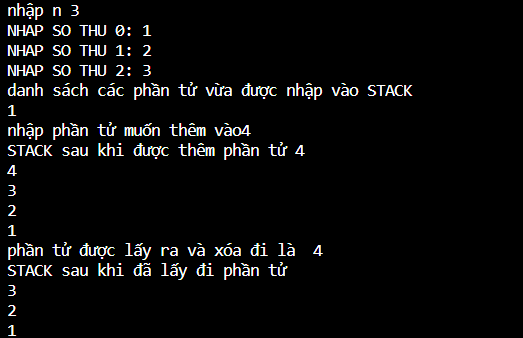
print("phần tử được lấy ra và xóa đi là ", p)

print("STACK sau khi đã lấy đi phần tử")

# câu lệnh dùng để in ra các phần tử trong Stack bắt đầu từ phần tử đầu tiên và kết thúc với phần tử cuối cùng

s.output()

**Kết quả:**

****

## BÀI 4: Cài đặt hàng đợi – queue

### 1. Nội dung

Cài đặt hàng đợi (queue) là một cấu trúc dữ liệu được sử dụng để lưu trữ các phần tử theo cơ chế "First-In, First-Out" (FIFO), có nghĩa là phần tử đầu tiên được đưa vào hàng đợi sẽ được lấy ra đầu tiên.

Các phương thức thường được sử dụng trong cài đặt hàng đợi bao gồm:

1. enqueue(item): Thêm một phần tử vào cuối của hàng đợi.
2. dequeue(): Lấy ra phần tử đầu tiên của hàng đợi và xóa nó khỏi hàng đợi.
3. front(): Trả về phần tử đầu tiên của hàng đợi mà không xóa nó khỏi hàng đợi.
4. isEmpty(): Kiểm tra xem hàng đợi có rỗng hay không.
5. size(): Trả về số lượng phần tử trong hàng đợi.

Cài đặt hàng đợi có thể được thực hiện bằng nhiều cách khác nhau, tùy thuộc vào mục đích sử dụng và yêu cầu hiệu suất của ứng dụng. Một số cách phổ biến để cài đặt hàng đợi bao gồm:

1. Sử dụng mảng: Sử dụng một mảng để lưu trữ các phần tử của hàng đợi. Một con trỏ front sẽ chỉ đến phần tử đầu tiên của hàng đợi và một con trỏ rear sẽ chỉ đến phần tử cuối cùng của hàng đợi. Khi một phần tử được thêm vào hàng đợi, rear sẽ được tăng lên một đơn vị. Khi một phần tử được lấy ra khỏi hàng đợi, front sẽ được tăng lên một đơn vị. Tuy nhiên, khi mảng đầy, ta cần phải di chuyển tất cả các phần tử lên một vị trí để tạo ra vị trí trống đầu tiên, dẫn đến hiệu suất thấp.
2. Sử dụng danh sách liên kết: Sử dụng một danh sách liên kết để lưu trữ các phần tử của hàng đợi. Mỗi nút của danh sách sẽ chứa một phần tử và một con trỏ đến nút tiếp theo trong danh sách. Đầu danh sách (head) sẽ chỉ đến phần tử đầu tiên của hàng đợi và cuối danh sách (tail) sẽ chỉ đến phần tử cuối cùng của hàng đợi. Khi một phần tử được thêm vào hàng đợi, một nút mới sẽ được tạo ra và được thêm vào danh sách liên kết. Khi một phần tử được lấy ra

### 2. Ý nghĩa

Trong lĩnh vực khoa học máy tính, hàng đợi (queue) là một cấu trúc dữ liệu được sử dụng để lưu trữ và quản lý các phần tử theo thứ tự đến trước, đến sau. Nó là một danh sách các phần tử, trong đó phần tử mới được chèn vào cuối danh sách và phần tử đầu tiên được lấy ra từ đầu danh sách, theo cơ chế FIFO (First In First Out) – đầu vào trước đầu ra trước.

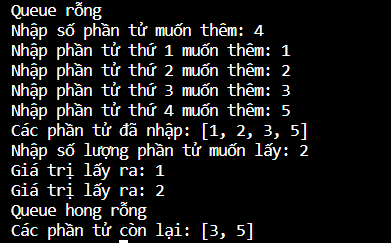
Các phần tử trong hàng đợi có thể được thêm vào (enqueue) hoặc lấy ra (dequeue). Việc cài đặt hàng đợi giúp cho việc xử lý các tác vụ theo thứ tự đến trước, đến sau, tránh được sự xáo trộn và đảm bảo tính nhất quán trong việc xử lý các phần tử. Hàng đợi được sử dụng rộng rãi trong các ứng dụng của khoa học máy tính như lập trình đa luồng, giải thuật tìm kiếm và sắp xếp, xử lý tín hiệu và thông tin, đồ họa máy tính, hệ thống truyền thông, v.v.

### Cách thức hoạt động của giải thuật

### Mã nguồn cài đặt hàng đợi-queue

1. import queue as que #Sử dụng thư viện Queue
2. q = que.Queue() #tạo Queue rỗng
3. #Tạo hàm kiểm tra giá trị Queue có rỗng không, nếu không rỗng in ra giá trị còn lại
4. def kiem\_tra\_rong(q):
5. if q.empty():
6. print("Queue rỗng")
7. else:
8. print("Queue hong rỗng")
9. q\_list = list(q.queue)
10. print("Các phần tử còn lại:", q\_list)
12. #Tạo hàm kiểm tra giá trị Queue có đầy không (Giới hạn queue) (Trường hợp đặc biệt)
13. q1 = que.Queue(maxsize=3)
14. q1.put(1)
15. q1.put(2)
16. q1.put(3)
17. def kiem\_tra\_day(q):
18. if q.full():
19. print("Queue đã đầy")
20. else:
21. print("Queue chưa đầy")
23. #tạo hàm thêm nhiều giá trị vào Queue
24. def them\_phan\_tu(q):
25. n = int(input("Nhập số phần tử muốn thêm: "))
26. for i in range (1,n+1):
27. x = int(input("Nhập phần tử thứ " + str(i) + " muốn thêm: "))
28. q.put(x) #Thêm 1 giá trị vào Queue
29. #Tạo hàm lấy phần tử đồng thời xóa nó khỏi Queue
30. def lay\_phan\_tu(q):
31. #Đầu tiên, ta xác định nếu hàm Queue ta đã tạo có đủ số lượng phần tử cần lấy ra không (lấy ra quá nhiều sẽ lỗi)
32. n = int(input("Nhập số lượng phần tử muốn lấy: "))
33. if q.qsize() < n:
34. print("Không đủ phần tử để lấy ra")
35. return
36. for i in range (1,n+1):
37. print("Giá trị lấy ra:",q.get()) #Lấy phần tử trong Queue, đồng thời xóa phần tử đó khỏi hàng đợi
38. return q
39. #Tạo hàm hiển thị Queue bằng cách chuyển Queue sang giá trị list
40. def hien\_thi\_queue(q):
41. q\_list = list(q.queue)
42. print("Các phần tử đã nhập:", q\_list)
43. #Chương trình hoàn chỉnh
44. #Tạo giá trị Queue 4 rỗng
45. q4 = que.Queue()
46. #Kiểm tra rỗng
47. kiem\_tra\_rong(q4)
48. #Nhập giá trị
49. them\_phan\_tu(q4)
50. #Hiển thị
51. hien\_thi\_queue(q4)
52. #Lấy và xóa bỏ phần tử
53. lay\_phan\_tu(q4)
54. #Kiểm tra rỗng lại lần nữa
55. kiem\_tra\_rong(q4)

**Kết quả:**



# CÂY

## BÀI 1: CÀI ĐẶT CÂY

### Nội dung

Cây lập trình (Programming Tree) là một cấu trúc dữ liệu mô tả cách mà các khối mã (code blocks) được tổ chức trong một chương trình hoặc hàm. Các khối mã được đặt trong các nút của cây và quan hệ giữa chúng được mô tả bởi các cạnh của cây. Cây lập trình thường được sử dụng để biểu diễn cấu trúc chương trình hoặc hàm một cách rõ ràng, giúp cho việc đọc và hiểu mã nguồn trở nên dễ dàng hơn.

Các loại cây lập trình phổ biến bao gồm cây cú pháp (Syntax Tree), cây thực thi (Execution Tree) và cây AST (Abstract Syntax Tree). Cây cú pháp biểu diễn cấu trúc cú pháp của chương trình, cây thực thi mô tả các bước thực thi của chương trình và cây AST biểu diễn cấu trúc trừu tượng của chương trình một cách rõ ràng và đơn giản hơn.

### Ý nghĩa

Việc sử dụng cây lập trình mang lại nhiều lợi ích trong việc lập trình và phát triển phần mềm. Dưới đây là một số ý nghĩa của việc sử dụng cây lập trình:

1. Biểu diễn cấu trúc chương trình một cách rõ ràng: Cây lập trình giúp biểu diễn cấu trúc của chương trình một cách rõ ràng và dễ hiểu hơn. Các khối mã được tổ chức theo một thứ tự nhất định, giúp cho việc đọc và hiểu mã nguồn trở nên dễ dàng hơn.
2. Dễ dàng tìm lỗi: Cây lập trình cũng giúp cho việc tìm lỗi trong chương trình trở nên dễ dàng hơn. Với cây lập trình, các lỗi có thể được xác định và giải quyết nhanh chóng, giúp tiết kiệm thời gian và công sức trong việc sửa lỗi.
3. Phát triển phần mềm dễ dàng hơn: Sử dụng cây lập trình giúp cho việc phát triển phần mềm trở nên dễ dàng hơn. Việc biểu diễn cấu trúc của chương trình một cách rõ ràng giúp cho các nhà phát triển dễ dàng hơn trong việc tạo, thay đổi và cải tiến chương trình.
4. Dễ dàng đọc và hiểu mã nguồn: Cây lập trình giúp cho mã nguồn trở nên dễ đọc và hiểu hơn. Với cấu trúc rõ ràng, các nhà phát triển có thể dễ dàng hình dung được cấu trúc của chương trình mà không cần phải đọc toàn bộ mã nguồn.
5. Tăng tính cấu trúc và tái sử dụng của chương trình: Sử dụng cây lập trình giúp tăng tính cấu trúc và tái sử dụng của chương trình. Các khối mã được tổ chức một cách rõ ràng, giúp cho việc sử dụng lại mã nguồn trở nên dễ dàng hơn, từ đó giúp tiết kiệm thời gian và công sức trong quá trình phát triển phần mềm.

### Cách thức hoạt động

Cây lập trình hoạt động bằng cách sử dụng các nút và liên kết giữa chúng để biểu diễn cấu trúc của một chương trình máy tính. Mỗi nút trong cây lập trình đại diện cho một thực thể trong chương trình, ví dụ như một toán hạng, một biến, một phép tính, hoặc một lệnh điều khiển. Các liên kết giữa các nút thể hiện mối quan hệ logic giữa chúng.

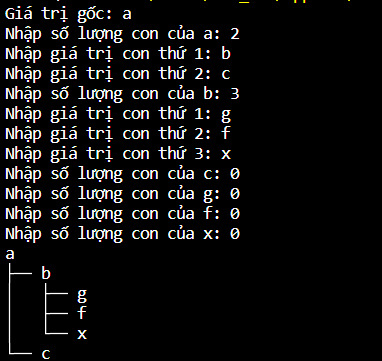
Cây lập trình thường được xây dựng dựa trên ngôn ngữ lập trình và có thể được tạo ra bằng tay hoặc bằng các công cụ tự động. Cây lập trình được sử dụng trong nhiều môi trường lập trình khác nhau, bao gồm các trình biên dịch, trình thông dịch, trình gỡ lỗi, và các công cụ phân tích chương trình.

Khi một chương trình được thực thi, cây lập trình được duyệt theo từ trên xuống dưới bằng cách sử dụng các thuật toán duyệt cây như duyệt theo chiều sâu (depth-first) hoặc duyệt theo chiều rộng (breadth-first). Khi duyệt cây, các nút sẽ được thực thi tuần tự theo thứ tự được quy định bởi cấu trúc của cây lập trình.

### Mã nguồn cài đặt cây trong ngôn ngữ python

1. import anytree as cay
2. from anytree import Node, RenderTree, NodeMixin
3. class Node:
4. def \_\_init\_\_(self, ten):
5. self.ten = ten
6. self.children = []
7. def them\_cay(self, node):
8. self.children.append(node)
9. def tao\_cay():
10. root\_value = input("Giá trị gốc: ")
11. root = Node(root\_value)
12. node\_queue = [root]
13. while len(node\_queue) > 0:
14. current\_node = node\_queue.pop(0)
15. num\_children = int(input(f"Nhập số lượng con của {current\_node.ten}: "))
16. for i in range(num\_children):
17. child\_value = input(f"Nhập giá trị con thứ {i+1}: ")
18. child\_node = Node(child\_value)
19. current\_node.them\_cay(child\_node)
20. node\_queue.append(child\_node)
21. return root
22. goc = tao\_cay()
23. for pre, \_, node in RenderTree(goc):
24. print("%s%s" % (pre, node.ten))

**kết quả:**

****

## BÀI 2: CÀI ĐẶT CÂY- DUYỆT CÂY THEO THỨ TỰ TRƯỚC

### Nội dung

Duyệt cây theo thứ tự trước (pre-order traversal) là một cách duyệt qua tất cả các nút trong cây theo thứ tự nhất định. Khi duyệt cây theo thứ tự trước, trước tiên ta sẽ duyệt qua nút gốc, sau đó duyệt qua tất cả các nút con bên trái của nút gốc, và cuối cùng là tất cả các nút con bên phải của nút gốc.

### Ý nghĩa

Việc sử dụng cài đặt cây và duyệt cây theo thứ tự trước là rất hữu ích trong việc giải quyết nhiều bài toán lập trình. Các ứng dụng của cây và duyệt cây theo thứ tự trước có thể bao gồm:

* Tìm kiếm: Cây được sử dụng để lưu trữ và tìm kiếm các phần tử dựa trên giá trị. Khi duyệt cây theo thứ tự trước, ta có thể duyệt qua tất cả các nút trong cây và kiểm tra giá trị của chúng để tìm kiếm phần tử cần tìm.
* Xây dựng cây tìm kiếm nhị phân: Cây tìm kiếm nhị phân được sử dụng để tìm kiếm phần tử trong danh sách. Khi xây dựng cây tìm kiếm nhị phân, ta có thể duyệt qua danh sách để thêm các phần tử vào cây theo thứ tự trước.
* Tính toán giá trị biểu thức: Cây biểu thức được sử dụng để tính toán giá trị của biểu thức số học. Khi duyệt cây biểu thức theo thứ tự trước, ta có thể tính toán giá trị của biểu thức bằng cách duyệt qua các nút theo thứ tự tương ứng.
* Tạo đồ thị: Cây cũng được sử dụng để tạo đồ thị. Khi tạo đồ thị, ta có thể duyệt qua danh sách các cạnh và tạo các nút và liên kết chúng với nhau theo thứ tự trước.

Vì vậy, việc sử dụng cài đặt cây và duyệt cây theo thứ tự trước là rất hữu ích trong nhiều bài toán lập trình khác nhau.

### Cách thức hoạt động

Cây được cấu tạo bởi các nút (node) và các cạnh (edge). Mỗi nút trong cây đại diện cho một giá trị (value) và có thể có một hoặc nhiều con (children) để tạo thành cấu trúc của cây.

Duyệt cây theo thứ tự trước (preorder traversal) bắt đầu từ gốc của cây, sau đó duyệt qua từng nút theo thứ tự: trước, trái, phải. Nghĩa là, ta truy cập vào giá trị của nút hiện tại trước, sau đó đi theo con bên trái của nó, và cuối cùng là con bên phải của nó.

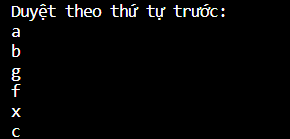
Các bước để duyệt cây theo thứ tự trước như sau:

1. Kiểm tra xem nút hiện tại có tồn tại hay không. Nếu không, quay lại bước trước đó hoặc kết thúc quá trình duyệt.
2. Truy cập giá trị của nút hiện tại.

### Mã nguồn cài đặt cây-duyệt theo thứ tự trước

1. #Duyệt theo thứ tự trước
2. def duyet\_theo\_thu\_tu\_truoc(node):
3. if node is not None:
4. print(node.ten)
5. for children in node.children:
6. duyet\_theo\_thu\_tu\_truoc(children)
7. print("Duyệt theo thứ tự trước: ")
8. duyet\_theo\_thu\_tu\_truoc(goc)

**kết quả:**

****

## BÀI 3: CÀI ĐẶT CÂY- DUYỆT CÂY THEO THỨ TỰ SAU

### Nội dung

Duyệt cây theo thứ tự sau (postorder traversal) là một phương pháp duyệt cây nhị phân bằng cách truy cập trước các nút con trái và phải của một nút, sau đó truy cập nút đó. Cụ thể, thứ tự duyệt cây theo thứ tự sau sẽ là:

1. Duyệt cây con trái của nút hiện tại theo thứ tự sau.
2. Duyệt cây con phải của nút hiện tại theo thứ tự sau.
3. Truy cập nút hiện tại.

Với cách duyệt này, các nút con của một nút sẽ được truy cập trước khi truy cập nút đó, và các nút ở mức thấp hơn trong cây sẽ được truy cập trước các nút ở mức cao hơn.

### Ý nghĩa

Duyệt cây theo thứ tự sau (postorder traversal) giúp chúng ta truy cập vào các nút con trước đó rồi mới đến nút hiện tại. Cách duyệt này thường được sử dụng khi ta cần thực hiện các thao tác trên cây theo đúng thứ tự kết quả của các nút con trước đó.

Một số ứng dụng của duyệt cây theo thứ tự sau như:

* Tính toán giá trị của các biểu thức toán học phức tạp. Trong trường hợp biểu thức toán học được biểu diễn dưới dạng cây, ta cần duyệt cây theo thứ tự sau để tính toán giá trị của các phép tính.
* Xóa cây hoặc giải phóng các tài nguyên được sử dụng để lưu trữ cây. Trong trường hợp ta muốn giải phóng bộ nhớ được sử dụng để lưu trữ các nút trong cây, ta cần duyệt cây theo thứ tự sau để xóa từng nút trong cây một cách đúng đắn.
* Tìm kiếm các đường đi trong cây. Trong một số trường hợp, ta cần tìm kiếm các đường đi trong cây sao cho các nút con của một nút được xét trước đến trước khi duyệt đến nút hiện tại. Việc duyệt cây theo thứ tự sau sẽ giúp chúng ta xác định các đường đi trong cây theo đúng thứ tự của các nút con.

### Cách thức hoạt động

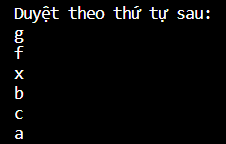
Để duyệt cây theo thứ tự sau (postorder traversal), ta sẽ truy cập vào các nút con trước đó rồi mới đến nút hiện tại. Các bước để thực hiện duyệt cây theo thứ tự sau như sau:

* Kiểm tra xem nút hiện tại có tồn tại hay không. Nếu không, quay lại bước trước đó hoặc kết thúc quá trình duyệt.
* Duyệt các nút con bên trái của nút hiện tại.
* Duyệt các nút con bên phải của nút hiện tại.
* Truy cập giá trị của nút hiện tại.

### Mã nguồn cài đặt cây-duyệt cây theo thứ tự sau

1. def duyet\_theo\_thu\_tu\_sau(node):
2. if node is not None:
3. for children in node.children:
4. duyet\_theo\_thu\_tu\_sau(children)
5. print(node.ten)
6. print("Duyệt theo thứ tự sau: ")
7. duyet\_theo\_thu\_tu\_sau(goc)

**kết quả:**

****

# ĐỒ THỊ

## BÀI 1: ĐỒ THỊ VÔ HƯỚNG

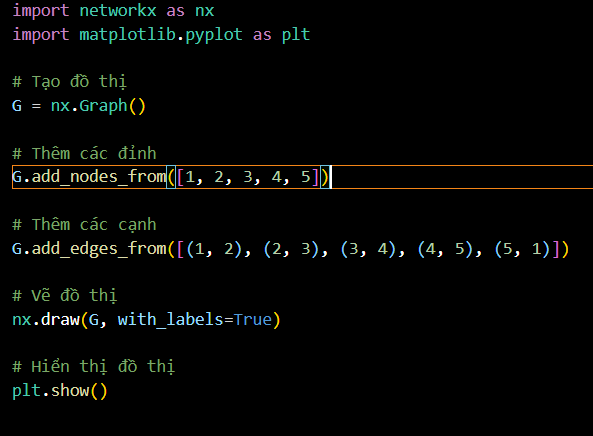
### Nội dung

Đồ thị vô hướng là một loại đồ thị trong đó các cạnh không có hướng đi riêng biệt, tức là các cạnh không có phân biệt giữa đỉnh đầu và đỉnh cuối. Ví dụ, nếu có một cạnh kết nối hai đỉnh A và B trong đồ thị vô hướng, thì ta có thể đi từ A đến B hoặc từ B đến A bằng cùng một cạnh. Các đỉnh của đồ thị vô hướng cũng không có hướng đi riêng biệt, có nghĩa là nếu hai đỉnh nối với nhau bằng một cạnh thì chúng được coi là kề nhau.

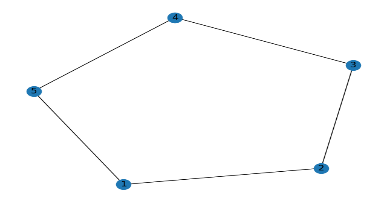
### Ý nghĩa

Đồ thị vô hướng thường được sử dụng để mô hình hóa mối quan hệ giữa các đối tượng hoặc sự kiện trong thực tế, ví dụ như mạng xã hội, đường phố, hệ thống giao thông và các mối liên kết trong phân tích mạng lưới.

### Mã nguồn cài đặt đồ thị vô hướng trong python:



**Kết quả:**

****

## BÀI 2: CÀI ĐẶT ĐỒ THỊ CÓ HƯỚNG

### Nội dung

Đồ thị có hướng (directed graph) là một loại đồ thị mà các cạnh có hướng đi từ một đỉnh (đỉnh gốc) đến đỉnh khác (đỉnh đích). Các cạnh trong đồ thị có hướng được gọi là cạnh đi và cạnh đến, và thường được biểu diễn bằng mũi tên.

### Ý nghĩa

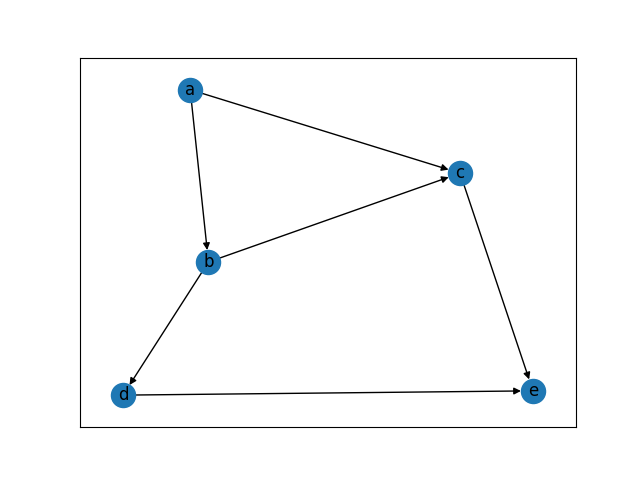
Đồ thị có hướng (directed graph) được sử dụng để mô hình hóa các quan hệ có hướng, tức là các quan hệ mà có sự tương tác giữa các đối tượng theo một hướng nhất định. Ví dụ như mô hình hệ thống định tuyến mạng, các mô hình quan hệ và liên kết giữa các trang web, hoặc các mô hình thực hiện truyền tin trên mạng xã hội.

Trong các ứng dụng thực tế, đồ thị có hướng được sử dụng để giải quyết các bài toán phức tạp như định tuyến mạng, phân tích quan hệ giữa các đối tượng, tìm kiếm đường đi ngắn nhất và phân tích mạng xã hội. Ngoài ra, đồ thị có hướng cũng được sử dụng trong các lĩnh vực như lý thuyết điều khiển, xử lý ngôn ngữ tự nhiên và sinh học tính toán.

### Mã nguồn cài đặt đồ thị có hướng

1. import networkx as nx
2. import matplotlib.pyplot as plt
3. # Tạo đồ thị
4. G = nx.DiGraph()
5. # Thêm các nút vào đồ thị
6. G.add\_nodes\_from(['a' , 'b', 'c', 'd', 'e'])
7. # Thêm các cạnh vào đồ thị
8. G.add\_edges\_from([('a', 'b'), ('a', 'c'), ('b', 'c'), ('b', 'd'), ('c', 'e'), ('d', 'e'), ('d', 'e')])
9. # Vẽ đồ thị
10. nx.draw\_networkx(G)
11. # Hiển thị đồ thị
12. plt.show()

**Kết quả:**



# CHƯƠNG 5: SẮP XẾP

## BÀI 1: SẮP XẾP CHỌN

### Nội dung

Sắp xếp chọn (Selection Sort) là một thuật toán sắp xếp đơn giản và dễ hiểu. Thuật toán này hoạt động bằng cách tìm kiếm phần tử nhỏ nhất (hoặc lớn nhất) trong mảng và đặt nó vào vị trí đầu tiên. Sau đó, tìm kiếm phần tử nhỏ nhất (lớn nhất) tiếp theo trong mảng chưa được sắp xếp và đặt nó vào vị trí thích hợp tiếp theo. Quá trình này được lặp lại cho đến khi tất cả các phần tử trong mảng được sắp xếp.

Cách thức hoạt động của thuật toán sắp xếp chọn:

1. Chọn phần tử nhỏ nhất (hoặc lớn nhất) trong mảng chưa được sắp xếp.
2. Hoán đổi phần tử này với phần tử ở vị trí đầu tiên của mảng.
3. Tiếp tục quá trình trên cho đến khi tất cả các phần tử được sắp xếp.

Thuật toán sắp xếp chọn có độ phức tạp thời gian là O(n^2), nghĩa là thời gian thực hiện tăng theo bình phương số lượng phần tử trong mảng. Do đó, đây là một thuật toán không hiệu quả cho các tập dữ liệu lớn. Tuy nhiên, vì tính đơn giản và dễ cài đặt, thuật toán sắp xếp chọn vẫn được sử dụng trong một số trường hợp đơn giản hoặc khi số lượng phần tử nhỏ.

### Ý nghĩa

Việc sắp xếp chọn là tạo ra một danh sách hoặc mảng được sắp xếp theo thứ tự tăng dần hoặc giảm dần. Qua đó, nó cung cấp một số lợi ích quan trọng:

1. Sắp xếp dữ liệu: Thuật toán sắp xếp chọn giúp sắp xếp dữ liệu theo một thứ tự cụ thể, tạo ra một danh sách có tổ chức hơn. Điều này giúp dễ dàng tìm kiếm, truy cập và xử lý dữ liệu.
2. Tìm kiếm dữ liệu: Khi dữ liệu đã được sắp xếp, việc tìm kiếm dữ liệu cụ thể trở nên hiệu quả hơn. Đặc biệt là với sắp xếp tăng dần, ta có thể sử dụng thuật toán tìm kiếm nhị phân để tìm kiếm nhanh chóng một phần tử trong danh sách.
3. Hiểu thuật toán: Sắp xếp chọn là một thuật toán đơn giản và dễ hiểu, giúp người học hiểu rõ cách hoạt động của thuật toán sắp xếp và các khái niệm cơ bản liên quan đến sắp xếp.
4. Tiền đề cho thuật toán phức tạp hơn: Sắp xếp chọn là một trong những thuật toán sắp xếp cơ bản và làm cơ sở cho nhiều thuật toán sắp xếp phức tạp hơn. Hiểu và nắm vững thuật toán sắp xếp chọn sẽ giúp người học tiếp cận và hiểu các thuật toán sắp xếp phức tạp hơn một cách dễ dàng hơn.

Tuy thuật toán sắp xếp chọn không hiệu quả với các tập dữ liệu lớn, nhưng nó có ý nghĩa giáo dục và là một cơ sở quan trọng để tìm hiểu về các thuật toán sắp xếp khác và tối ưu hóa hiệu suất sắp xếp.

### Mã nguồn sắp xếp chọn

Dưới đây là mã nguồn sắp xếp chọn ( tăng dần) như sau:

lst=[1,4,2,5,3,7,9,6,8,10]

def sap\_xep\_chon(lst):

    lst\_1=lst.copy()

    for i in range(len(lst)-1):

        min=i

        for j in range(i+1,len(lst\_1)):

            if lst\_1[j]<lst\_1[min]:

                min=j

        lst\_1[min],lst\_1[i]=lst\_1[i],lst\_1[min]

    return lst\_1

print(sap\_xep\_chon(lst))

**giải thích:**

- Dựng hàm sap\_xep\_chon với tham số truyền vào là 1 list số thực.

- Tạo một bản sao của list đó để có thể tái sử dụng.

- Sử dụng vòng lặp lồng vòng lặp, gán biến min=i ở vị trí đầu tiền,sau đó sử dụng vòng lặp bắt đầu từ vị trí thứ i+1 đến phần tử cuối cùng, nếu gặp phần tử bé hơn vị trí min thì gán lại min bằng vị trí đó.

- Sau khi ra khỏi vòng lặp thì hoán đổi vị trí min ban đầu với vị trí min vừa tìm thấy ( lúc này trị trí có giá trị min sẽ lên đầu danh sách)

**Kết quả:**



## BÀI 2: SẮP XẾP CHÈN

## Nội dung

Sắp xếp chèn là một thuật toán sắp xếp cơ bản trong lập trình. Ý nghĩa của việc sắp xếp chèn là tạo ra một danh sách hoặc mảng được sắp xếp theo thứ tự tăng dần hoặc giảm dần.

### Ý Nghĩa

1. Sắp xếp dữ liệu: Thuật toán sắp xếp chèn giúp sắp xếp dữ liệu theo một thứ tự cụ thể. Nó đi qua từng phần tử của danh sách và chèn nó vào vị trí thích hợp trong danh sách đã sắp xếp. Khi hoàn thành, ta có một danh sách được sắp xếp.
2. Hiệu quả với dữ liệu nhỏ: Sắp xếp chèn thường hiệu quả khi xử lý các danh sách có kích thước nhỏ hoặc đã được sắp xếp một phần. Với dữ liệu nhỏ, hiệu suất của thuật toán không bị ảnh hưởng quá nhiều.
3. Thích hợp cho dữ liệu liên tục: Thuật toán sắp xếp chèn làm việc tốt với dữ liệu đến từng phần tử một, tức là dữ liệu liên tục. Với dữ liệu đầu vào được nhập từ người dùng hoặc trực tiếp từng phần tử một, sắp xếp chèn là một lựa chọn tốt.
4. Dễ hiểu và triển khai: Thuật toán sắp xếp chèn là một trong những thuật toán đơn giản và dễ hiểu. Nó có cấu trúc đơn giản và ít yêu cầu kiến thức toán học phức tạp. Điều này làm cho nó dễ dàng triển khai và debug.

Tuy nhiên, sắp xếp chèn không phải lúc nào cũng hiệu quả với các tập dữ liệu lớn vì độ phức tạp thời gian của nó là O(n^2). Nhưng nó vẫn có ý nghĩa giáo dục và là cơ sở cho việc hiểu và nắm vững các khái niệm cơ bản về thuật toán sắp xếp.

### Mã nguồn sắp xếp chèn

Dưới đây là mã nguồn sắp xếp chèn ( tăng dần) như sau.

lst=[1,2,7,9,45,6,3,9]

def sap\_xep\_chen(lst):

    lst\_1=lst.copy()

    for i in range(len(lst)):

        index\_min=i

        min=lst\_1[i]

        while( index\_min>0 and(lst\_1[index\_min-1]>min)):

            lst\_1[index\_min]=lst\_1[index\_min-1]

            index\_min-=1

        lst\_1[index\_min]=min

    return lst\_1

print(sap\_xep\_chen(lst))

**Giải thích :**

- Dựng hàm sap\_xep\_chen với tham số lst là 1 list số thực.

- Gán vị trí có giá trị nhỏ nhất là phần tử đầu tiên.

- Nếu index\_min >0 và giá trị nằm trước phần tử nhỏ nhất lớn hơn min thì gán lại phần tử có giá trị nhỏ nhất bằng phần tử trước đó, sau khi thoát khỏi vòng lặp giá trị sẽ được cập nhật lại. Nếu không qua vòng lặp thì sẽ vẫn giữ nguyên.

**Kết quả:**

****

## BÀI 3: SẮP XẾP NỔI BỌT

### Nội dung

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.

### Ý Nghĩa

Sắp xếp dữ liệu: Thuật toán sắp xếp nổi bọt giúp sắp xếp dữ liệu theo một thứ tự cụ thể. Nó so sánh các cặp phần tử liền kề và hoán đổi chúng nếu chúng không theo thứ tự mong muốn. Quá trình này được lặp lại cho đến khi không còn cặp phần tử nào cần hoán đổi, tức là danh sách đã được sắp xếp.

Đơn giản và dễ hiểu: Sắp xếp nổi bọt là một trong những thuật toán sắp xếp đơn giản và dễ hiểu nhất. Nó chỉ yêu cầu việc so sánh và hoán đổi các phần tử liền kề, không có các khái niệm phức tạp hay công thức toán học.

Dễ triển khai: Với cấu trúc đơn giản, sắp xếp nổi bọt dễ dàng triển khai và debug. Việc triển khai thuật toán này không đòi hỏi nhiều kiến thức toán học hay lập trình phức tạp.

### Mã nguồn sắp xếp nổi bọt

# Hàm khai báo sắp xếp nổi bọt với 2 tham số đầu vào là mảng a và kích thước của mảng n

def bubbleSort(a, n):

    # biến sorted được gán bằng giá trị False để kiểm tra xem mảng đã sắp xếp xong hay chưa

    sorted = False

    # vòng lặp for lặp qua các phần tử của mảng a từ phần tử thứ 2 đến phần tử cuối cùng của mảng

    for i in range(1, n):

        # nếu mảng chưa được sắp xếp xong thì thực hiện các câu lệnh bên dưới

        if not sorted:

            # Gán giá trị True cho biến sorted để kiểm tra mảng

            sorted = True

            # vòng lặp for lồng vào vòng lặp của bubblesort để thực hiện việc so sánh và hoán đổi vị trí các phần tử trong mảng

            # bắt đầu từ phần tử đầu tiên đến n-i-1 (giảm số phép so sánh vì sau mỗi lần lặp phần tử lớn nhất đã nằm ở vị trí cuối cùng của mảng)

            for j in range(0, n-i):

                # so sánh phần tử a[j] và a[j+1] nghĩa là 1 phần tử và phần tử tiếp theo trong mảng

                # nếu giá trị a[j] lớn hơn a[j+1] thì ta thực hiện hoán đổi vị trí 2 phần tử này

                if a[j] > a[j+1]:

                    a[j], a[j+1] = a[j+1], a[j]

                    # trong quá trình lặp ta thực hiện ít nhất 1 lần hoán đổi(mảng chưa được sắp xếp xong) vì thế cần gán False cho sorted để đảm bảo rằng vòng lặp tiếp tục được thực hiện để kiểm tra mảng

                    sorted = False

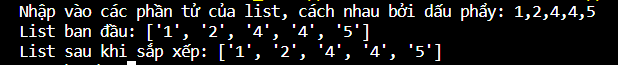
my\_list = input("Nhập vào các phần tử của list, cách nhau bởi dấu phẩy: ").split(",")

print("List ban đầu:", my\_list)

bubbleSort(my\_list, len(my\_list))

print("List sau khi sắp xếp:", my\_list)

**Kết quả :**



## BÀI 4: THUẬT TOÁN SẮP XẾP NHANH-QUICK SORT

### Nội dung

Thuật toán sắp xếp nhanh (Quick Sort) là một thuật toán sắp xếp phổ biến trong lập trình. Nó dựa trên phương pháp chia để trị (divide and conquer) để sắp xếp các phần tử trong danh sách.

Ý tưởng cơ bản của Quick Sort là chọn một phần tử chốt (pivot) từ danh sách và xếp các phần tử nhỏ hơn pivot vào một phía, các phần tử lớn hơn pivot vào phía còn lại. Sau đó, quá trình sắp xếp được áp dụng đệ quy cho cả hai phần đã chia, cho đến khi danh sách đã được sắp xếp hoàn toàn.

Cách hoạt động của Quick Sort như sau:

1. Chọn một phần tử chốt (pivot) từ danh sách.
2. Tạo hai con trỏ, một ở đầu danh sách (con trỏ trái) và một ở cuối danh sách (con trỏ phải).
3. Di chuyển con trỏ trái từ đầu danh sách và con trỏ phải từ cuối danh sách. Nếu con trỏ trái trỏ vào một phần tử lớn hơn pivot và con trỏ phải trỏ vào một phần tử nhỏ hơn pivot, hoán đổi hai phần tử này.
4. Tiếp tục di chuyển con trỏ trái và con trỏ phải cho đến khi chúng gặp nhau. Khi đó, hoán đổi phần tử pivot với phần tử mà con trỏ trái đang trỏ tới.
5. Tiếp tục quá trình này đệ quy cho hai phần danh sách được tạo ra từ phân chia (phần bên trái và phần bên phải của pivot), cho đến khi danh sách đã sắp xếp hoàn toàn.

Quick Sort có độ phức tạp trung bình là O(n log n), nơi n là số lượng phần tử trong danh sách. Tuy nhiên, trong trường hợp xấu nhất khi danh sách đã sắp xếp hoặc gần sắp xếp, độ phức tạp có thể lên đến O(n^2). Quick Sort cũng là một thuật toán không ổn định, nghĩa là thứ tự tương đối của các phần tử bằng nhau có thể bị thay đổi sau quá trình sắp xếp.

Tuy Quick Sort có một số hạn chế, nhưng với hiệu suất cao và tính linh hoạt trong triển khai, nó vẫn được sử dụng rộng rãi trong các ứng dụng thực tế.

### Ý nghĩa

1. Thuật toán sắp xếp nhanh (Quick Sort) là một thuật toán sắp xếp phổ biến và hiệu quả trong lập trình. Ý nghĩa của việc sắp xếp nhanh là tạo ra một danh sách hoặc mảng được sắp xếp theo thứ tự tăng dần hoặc giảm dần. Dưới đây là ý nghĩa cụ thể của sắp xếp nhanh:
2. Hiệu suất cao: Quick Sort là một trong những thuật toán sắp xếp hiệu suất cao nhất trong thực tế. Với độ phức tạp trung bình là O(n log n), Quick Sort thường cho kết quả nhanh chóng và hiệu quả đối với các tập dữ liệu lớn.
3. Phân chia đối tượng: Quick Sort sử dụng phương pháp phân chia (partitioning) để chia danh sách ban đầu thành các phân đoạn nhỏ hơn. Các phần tử trong danh sách sẽ được so sánh với một phần tử chốt (pivot), và sau quá trình phân chia, các phần tử nhỏ hơn pivot sẽ nằm bên trái pivot, các phần tử lớn hơn pivot sẽ nằm bên phải pivot. Quá trình này sẽ được thực hiện đệ quy cho từng phân đoạn, cho đến khi danh sách được sắp xếp hoàn toàn.
4. Dễ triển khai: Mặc dù Quick Sort có một số bước phức tạp, nhưng nó vẫn dễ triển khai trong lập trình. Thuật toán Quick Sort có thể được triển khai bằng đệ quy hoặc vòng lặp, và không yêu cầu nhiều bộ nhớ phụ hay cấu trúc dữ liệu phức tạp.
5. Khả năng tùy chỉnh: Quick Sort cho phép tuỳ chỉnh các phương pháp chọn pivot, ví dụ như chọn pivot ngẫu nhiên hoặc chọn pivot ở vị trí trung tâm. Điều này giúp tối ưu hóa thuật toán và tránh trường hợp xấu nhất.

### Mã Nguồn thuật toán sắp xếp nhanh - quick sort

1. def quick\_sort(arr):
2. if len(arr) <= 1:
3. return arr
4. #nếu danh sách chỉ có 1 phần tử hoặc rỗng,thì sẽ được trả về ngay lập tức
5. else:
6. pivot = arr[0]#gán phần tử đầu tiên vào pivot
7. smaller = [x for x in arr[1:] if x <= pivot]
8. #tạo danh sách chứa các phần tử nhỏ hơn hoặc bằng pivot
9. greater = [x for x in arr[1:] if x > pivot]
10. #tạo danh sách chứa các phần tử lớn hơn pivot
11. return quick\_sort(smaller) + [pivot] + quick\_sort(greater)
12. #sử dụng đệ quy để sắp xếp danh sách smaller và greater
13. # Sử dụng hàm quick\_sort để sắp xếp một danh sách
14. my\_list = [6, 2, 8, 1, 5, 9, 3]
15. sorted\_list = quick\_sort(my\_list)
16. print(sorted\_list)

**Kết quả:**

****

## BÀI 5: THUẬT TOÁN HEAP SORT

### 1.Nội dung

Thuật toán Heap Sort là một thuật toán sắp xếp dựa trên cấu trúc dữ liệu Heap. Thuật toán này hoạt động bằng cách sử dụng một cây nhị phân đầy đủ (cây Heap) để sắp xếp các phần tử trong danh sách.

Các bước thực hiện thuật toán Heap Sort như sau:

1. Xây dựng một cây Heap từ danh sách ban đầu. Cây Heap có thể là Max Heap (phần tử cha lớn hơn hoặc bằng các phần tử con) hoặc Min Heap (phần tử cha nhỏ hơn hoặc bằng các phần tử con).
2. Hoán đổi phần tử gốc của cây Heap (phần tử lớn nhất trong trường hợp Max Heap hoặc phần tử nhỏ nhất trong trường hợp Min Heap) với phần tử cuối cùng trong danh sách.
3. Giảm kích thước của cây Heap đi 1.
4. Bảo toàn tính chất Heap bằng cách thực hiện phép sift-down (đẩy phần tử xuống) từ gốc của cây Heap.
5. Lặp lại bước 2-4 cho đến khi cây Heap chỉ còn một phần tử.
6. Kết quả là danh sách được sắp xếp tăng dần (hoặc giảm dần) từ phần tử nhỏ nhất đến lớn nhất (hoặc từ lớn nhất đến nhỏ nhất).

### 2. Ý Nghĩa

Việc sử dụng thuật toán Heap Sort đem lại một số ý nghĩa quan trọng như sau:

1. Hiệu suất tốt: Thuật toán Heap Sort có độ phức tạp trung bình là O(n log n), nhanh chóng và hiệu quả trong việc sắp xếp danh sách có kích thước lớn. Với hiệu suất tốt, Heap Sort thường được sử dụng trong các ứng dụng yêu cầu sắp xếp nhanh và đáng tin cậy.
2. Ổn định: Heap Sort là một thuật toán ổn định, nghĩa là nó bảo toàn thứ tự tương đối của các phần tử có cùng giá trị. Điều này có ý nghĩa quan trọng trong việc xử lý dữ liệu có nhiều trường hợp có cùng giá trị và cần duy trì thứ tự ban đầu của chúng.
3. Khả năng sắp xếp theo yêu cầu: Heap Sort có thể được sử dụng để sắp xếp danh sách theo thứ tự tăng dần hoặc giảm dần tùy thuộc vào việc xây dựng cây Heap là Max Heap hay Min Heap. Điều này làm cho Heap Sort linh hoạt và phù hợp với nhiều loại ứng dụng.
4. Khả năng sắp xếp dữ liệu phức tạp: Heap Sort không chỉ áp dụng cho các dạng dữ liệu đơn giản mà còn hoạt động hiệu quả trên các dạng dữ liệu phức tạp như cấu trúc dữ liệu có liên kết, dữ liệu đối tượng, và nhiều trường hợp khác.
5. Không gian bộ nhớ tốn ít: Heap Sort sử dụng không gian bộ nhớ hạn chế và chỉ cần một phần tử bổ sung để hoán đổi trong quá trình sắp xếp. Điều này làm cho Heap Sort phù hợp với việc sắp xếp trong môi trường có bộ nhớ hạn chế.

### 3. mã nguồn thuật toán heap sort

def heapify(arr, n, i):

    largest = i

    left = 2 \* i + 1

    right = 2 \* i + 2

    if left < n and arr[left] > arr[largest]:

        largest = left

    if right < n and arr[right] > arr[largest]:

        largest = right

# so sánh nút trái và nút phải với nút hiện tại nếu trong nút trái có giá trị lớn hơn sẽ được cập nhật tương ứng

    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)

# đổi chổ nút hiện tại và nút con lớn nhất

def heap\_sort(arr):

    n = len(arr)

    # xây dựng một heap tối đa

    for i in range(n // 2 - 1, -1, -1):

        heapify(arr, n, i)

    # Trích xuất các phần tử một cách tuần tự

    for i in range(n - 1, 0, -1):

        arr[i], arr[0] = arr[0], arr[i]  # hoán đôi

        heapify(arr, i, 0)

# sử dụng hàm heap sort để sắp xếp danh sách

my\_list = [6, 11, 8, 1, 5, 9, 10]

heap\_sort(my\_list)

print(my\_list)

**Kết quả:**



## BÀI 6: THUẬT TOÁN SẮP XẾP TRỘN-MERGE SORT

### Nội dung

Thuật toán sắp xếp trộn (Merge Sort) là một thuật toán sắp xếp đệ quy dựa trên nguyên tắc chia để trị. Nó chia mảng thành các nửa đơn giản hơn, sau đó kết hợp các nửa này lại một cách đệ quy để tạo ra một mảng đã được sắp xếp.

Ý tưởng chính của thuật toán Merge Sort là:

1. Chia mảng ban đầu thành hai nửa bằng cách tìm chỉ số giữa.
2. Đệ quy chia tiếp các nửa con đã tạo ra cho đến khi mỗi nửa chỉ còn một phần tử.
3. Kết hợp hai nửa đã chia để tạo ra một mảng đã được sắp xếp.

### Ý nghĩa

Ý nghĩa của việc sử dụng thuật toán sắp xếp trộn - Merge Sort là nó mang lại các lợi ích sau:

1. Độ phức tạp thời gian ổn định: Merge Sort có độ phức tạp thời gian trung bình và tối đa O(n log n), với n là số phần tử trong mảng. Điều này đảm bảo rằng thuật toán hoạt động hiệu quả ngay cả với các tập dữ liệu lớn.
2. Ổn định và tin cậy: Merge Sort duy trì tính ổn định của các phần tử có cùng giá trị. Điều này có ý nghĩa là các phần tử có cùng giá trị trong mảng gốc sẽ không thay đổi thứ tự sau khi được sắp xếp.
3. Dễ hiểu và dễ cài đặt: Thuật toán Merge Sort có cấu trúc đơn giản và dễ hiểu. Nó không đòi hỏi kiến thức toán học phức tạp và có thể được cài đặt một cách dễ dàng trong nhiều ngôn ngữ lập trình.
4. Phù hợp với các loại dữ liệu đa dạng: Merge Sort hoạt động tốt với các loại dữ liệu khác nhau, bao gồm số nguyên, số thực, chuỗi và các kiểu dữ liệu phức tạp khác. Điều này làm cho nó trở thành một thuật toán đa năng cho việc sắp xếp.
5. Tính ổn định: Merge Sort là một thuật toán ổn định, tức là nếu có hai phần tử bằng nhau, thì thứ tự tương đối của chúng trong mảng gốc sẽ được duy trì trong mảng đã sắp xếp. Điều này là quan trọng trong nhiều tình huống, như sắp xếp dữ liệu theo nhiều tiêu chí khác nhau cùng lúc.
6. Tính khả thi và linh hoạt: Merge Sort có thể được sử dụng cho cả mảng động và mảng tĩnh. Nó cũng có thể được tùy chỉnh để sắp xếp theo nhiều tiêu chí khác nhau, bằng cách thay đổi phần logic so sánh trong hàm merge.

### Mã nguồn thuật toán sắp xếp trộn-merge sort

1. def merge\_sort(arr):
2. if len(arr) <= 1:
3. return arr
4. # Chia mảng thành hai nửa
5. mid = len(arr) // 2
6. left\_half = arr[:mid]
7. right\_half = arr[mid:]
8. # Đệ quy sắp xếp các nửa con
9. left\_half = merge\_sort(left\_half)
10. right\_half = merge\_sort(right\_half)
11. # Kết hợp hai mảng đã sắp xếp
12. return merge(left\_half, right\_half)
13. def merge(left, right):
14. result = []
15. i = j = 0
16. # So sánh các phần tử và thêm vào mảng kết quả theo thứ tự tăng dần
17. while i < len(left) and j < len(right):
18. if left[i] <= right[j]:
19. result.append(left[i])
20. i += 1
21. else:
22. result.append(right[j])
23. j += 1
24. # Thêm các phần tử còn lại của nửa chưa được thêm vào kết quả
25. while i < len(left):
26. result.append(left[i])
27. i += 1
28. while j < len(right):
29. result.append(right[j])
30. j += 1
31. return result
32. # Sử dụng hàm merge\_sort để sắp xếp một danh sách
33. my\_list = [16, 12, 14, 10, 50, 49, 33]
34. sorted\_list = merge\_sort(my\_list)
35. print(sorted\_list)

**kết quả:**



# TÀI LIỆU THAM KHẢO

# KẾT QUẢ KIỂM TRA ĐẠO VĂN