**SỔ TAY DSA**

DATA STRUCTURES AND ALGORITHMS

# Topic 1: ABSTRACT DATA TYPES

**▪ Abstract data types (ADT)** bao gồm: Values và Operation

**Đặc điểm của ADT**: - Trừu tượng

- Độc lập với ngôn ngữ lập trình

- Hướng đối tượng (Object Oriented Programming - OOP)

**Một số ADT phổ biến**: string, array list, tuple, set, dictionary (map), linked list, stack, queue, priority queue (thường là heap), hash table, hash map, tree, graph,…

**▪ Bài toán chuyển đổi biểu thức trung tố (infix) sang hậu tố (postfix)** là một bài toán quan trọng trong Computer Science, thường được sử dụng trong việc xử lí biểu thức số học trong máy tính và trình biên dịch.

**VD:** A + B \* C → A B C \* + (A + B) \* C → A B + C \*

**Quy tắc chuyển đổi:** Duyệt qua chuỗi biểu thức:

- Toán hạng (số hoặc biến): đưa trực tiếp vào kết quả.

- Dấu mở ngoặc ‘(‘: đưa vào stack.

- Dấu đóng ngoặc ‘)’: pop các toán tử khỏi stack cho đến khi gặp ‘(‘.

- Toán tử (+, -, \*, /, ^,…):

+ Nếu stack rỗng, push vào stack.

+ Nếu stack không rỗng:

∙ So sánh độ ưu tiên với toán tử trên đỉnh stack.

Độ ưu tiên toán tử: (+, -) < (\*, /) < (^)

∙ Nếu toán tử mới có độ ưu tiên cao hơn, push vào stack.

∙ Nếu không, pop toán tử từ stack ra kết quả rồi mới push toán tử mới vào stack.

- Sau khi duyệt hết biểu thức, pop hết toán tử trong stack ra kết quả.

**Code Python:**

def infix\_to\_postfix(expression):

precedence = {‘+’: 1, ‘-‘: 1, ‘\*’: 2, ‘/’: 2, ‘^’: 3}

stack = []

queue = [] # kết quả

for char in expression:

if char.isalnum(): queue.append(char)

elif char == ‘(‘: stack.append(char)

elif char == ‘)’:

while stack and stack[-1] != ‘(‘:

queue.append(stack.pop())

stack.pop() # bỏ dấu ‘(‘ khỏi stack

else:

while stack and stack[-1] != ‘(‘ and

precedence.get(stack[-1], 0) >= precedence.get(char, 0):

queue.append(stack.pop())

stack.append(char)

while stack: queue.append(stack.pop())

return ‘ ‘.join(queue)

# Hàm tính biểu thức postfix:

def calculate\_postfix(expression):

stack = []

for char in expression:

if char.isdigit(): stack.append(int(char)) # đẩy số vào stack

else: # nếu là toán tử thì lấy 2 toán hạng từ stack

b = stack.pop()

a = stack.pop()

if char == ‘+’: stack.append(a + b)

elif char == ‘-‘: stack.append(a – b)

elif char == ‘\*’: stack.append(a \* b)

elif char == ‘/’: stack.append(a / b)

elif char == ‘^’: stack.append(a \*\* b)

return stack[0] # kết quả cuối cùng còn lại trong stack

**▪ Code Python định nghĩa Linked List, Stack, Queue:**

◦ **Singly Linked List:**

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def traverse(self):

cur = self.head

while cur:

print(cur.data, end = ‘ -> ‘)

cur = cur.next

print(‘None’)

def count\_nodes(self):

count = 0

cur = self.head

while cur:

count += 1

cur = cur.next

return count

def add\_head(self, data):

new = Node(data)

new.next = self.head

self.head = new

def add\_tail(self, data):

new = Node(data)

if not self.head:

self.head = new

return

cur = self.head

while cur.next: cur = cur.next

cur.next = new

def remove\_head(self):

if not self.head: return

self.head = self.head.next

def remove\_tail(self):

if not self.head: return

if not self.head.next:

self.head = None

return

cur = self.head

while cur.next and cur.next.next: cur = cur.next

cur.next = None

◦ **Stack (định nghĩa bằng Linked List):**

import LinkedList

class Stack:

def \_\_init\_\_(self):

self.linked\_list = LinkedList()

def push(self, data):

self.linked\_list.add\_head(data)

def pop(self):

if not self.linked\_list.head:

print(“Stack is empty!”)

return None

top\_data = self.linked\_list.head.data

self.linked\_list.remove\_head()

return top\_data

def top(self):

if not self.linked\_list.head:

print(“Stack is empty!”)

return None

return self.linked\_list.head.data

◦ **Queue (định nghĩa bằng Linked List):**

import LinkedList

class Queue:

def \_\_init\_\_(self):

self.linked\_list = LinkedList()

def enqueue(self, data):

self.linked\_list.add\_tail(data)

def dequeue(self):

if not self.linked\_list.head:

print(“Queue is empty!”)

return None

front\_data = self.linked\_list.head.data

self.linked\_list.remove\_head()

return front\_data

def front(self):

if not self.linked\_list.head:

print(“Queue is empty!”)

return None

return self.linked\_list.head.data

# Topic 2: RECURSION

**▪ Recursion (Đệ quy):** hàm gọi lại hàm chính nó.

**Các loại đệ quy:**

* Đệ quy tuyến tính (Linear Recursion)
* Đệ quy đuôi (Tail Recursion)
* Đệ quy tương hỗ (Mutual Recursion)
* Đệ quy nhị phân (Binary Recursion)
* Đệ quy lồng nhau (Nested Recursion)

❶ **Đệ quy tuyến tính:** hàm gọi lại chính nó 1 lần trong quá trình thực thi. VD: Tính tổng các phần tử trong danh sách:

def sum\_list(lst):

if not lst: return 0

return lst[0] + sum\_list(lst[1:])

❷  **Đệ quy đuôi:** lời gọi đệ quy là thao tác cuối cùng của hàm, giúp tối ưu bộ nhớ. VD: Tính giai thừa:

def factorial(n, acc = 1):

if n == 0: return acc

return factorial(n – 1, n \* acc)

❸ **Đệ quy tương hỗ:**  2 hoặc nhiều hàm gọi lẫn nhau một cách đệ quy. VD: Xác định số chẵn lẻ:

def is\_even(n):

if n == 0: return True

return is\_odd(n – 1)

def is\_odd(n):

if n == 0: return False

return is\_even(n – 1)

❹ **Đệ quy nhị phân:**  mỗi lời gọi đệ quy tạo ra 2 lời gọi con. VD: Tính Fibonacci:

def fibonacci(n):

if n <= 1: return n

return fibonacci(n – 1) + fibonacci(n – 2)

❺ **Đệ quy lồng nhau:**  hàm gọi chính nó ngay bên trong đối số của lời gọi đệ quy. VD:

def nested\_recursion(n):

if n > 100: return n – 10

return nested\_recursion(nested\_recursion(n + 11))

**Tổng kết:**

- Đệ quy tuyến tính, đệ quy đuôi → Dễ hiểu, tối ưu hơn.

- Đệ quy tương hỗ, đệ quy nhị phân → Xuất hiện trong lý thuyết số và thuật toán.

- Đệ quy lồng nhau → Hiếm gặp, dùng trong bài toán phức tạp.

▪ **Bài toán tháp Hà Nội:** bài toán đệ quy kinh điển, trong đó có n đĩa đặt trên 1 trong 3 cọc. Mục tiêu là di chuyển toàn bộ đĩa từ cọc nguồn A sang cọc đích C, tuân theo 2 quy tắc:

- Chỉ được di chuyển 1 đĩa mỗi lần.

- Không được đặt đĩa lớn hơn lên trên đĩa nhỏ hơn.

**Cách giải bài toán:**

- Chia vấn đề lớn thành các bài toán con nhỏ hơn.

- Gọi đệ quy để di chuyển n – 1 đĩa sang cọc trung gian.

- Chuyển đĩa lớn nhất sang cọc đích.

- Gọi đệ quy để di chuyển n – 1 đĩa từ cọc trung gian sang cọc đích.

**Code Python:**

def hanoi(n, source, auxiliary, target):

if n == 1:

print(f”Di chuyển đĩa 1 từ {source} đến {target}”)

return

hanoi(n – 1, source, target, auxiliary)

print(f”Di chuyển đĩa {n} từ {source} đến {target}”)

hanoi(n – 1, auxiliary, source, target)

hanoi(3, ‘A’, ‘B’, ‘C’)

Kết quả với 3 đĩa:

Di chuyển đĩa 1 từ A đến C

Di chuyển đĩa 2 từ A đến B

Di chuyển đĩa 1 từ C đến B

Di chuyển đĩa 3 từ A đến C

Di chuyển đĩa 1 từ B đến A

Di chuyển đĩa 2 từ B đến C

Di chuyển đĩa 1 từ A đến C

**Phân tích độ phức tạp thuật toán:**

- Gọi T(n) là số bước cần giải bài toán với n đĩa.

- Công thức truy hồi: T(n) = 2T(n – 1) + 1

- Giải ra được T(n) = 2^n – 1, nghĩa là số lần di chuyển tăng theo cấp số nhân.

**▪ Các kĩ thuật thiết kế giải thuật nâng cao:**

❶ **Chia để trị (Divide and Conquer)**

Chia bài toán lớn thành các bài toán con nhỏ hơn giống nhau, giải từng cái rồi kết hợp kết quả. VD: Merge Sort chia mảng thành 2 nửa, sắp xếp từng nửa rồi trộn lại.

❷ **Quay lui (Backtracking)**

Thử từng lựa chọn một cách tuần tự, nếu thấy không hợp thì quay lui để thử hướng khác. Có 2 loại:

- Vét cạn (Brute-force): thử tất cả mọi khả năng. VD: Tìm tất cả hoán vị của một chuỗi.

- Nhánh cận (Branch and Bound): giống quay lui nhưng có cắt nhánh sớm nếu thấy hướng đó không tối ưu. VD: Giải bài toán cái túi (knapsack) bằng cách loại bỏ nhánh không khả thi.

❸ **Tham lam/Háu ăn (Greedy)**

Luôn chọn lựa chọn tốt nhất ở hiện tại với hi vọng nó dẫn đến kết quả tối ưu toàn cục. VD: Bài toán tiền lẻ chọn mệnh giá lớn nhất trước để đạt tổng cần đổi.

❹ **Quy hoạch động (Dynamic Programming)**

Chia nhỏ bài toán, lưu kết quả của các bài toán con để tránh tính lại. VD: Fibonacci thay vì đệ quy lặp lại, ta lưu f(i – 1) và f(i – 2) để tính f(i).

# Topic 3: ANALYSIS OF ALGORITHMS

**▪ Độ phức tạp của thuật toán:**

**∙** T(n) là một hàm số để mô tả thời gian thực thi của một thuật toán, dựa trên kích thước đầu vào của nó.

**∙** Big-O notation là kí hiệu đại số để biểu diễn độ phức tạp thời gian của một thuật toán, biểu thị tốc độ tăng của hàm thời gian T(n) khi n → ∞.

**∙** Thứ tự độ phức tạp từ tốt nhất đến kém nhất (theo tốc độ tăng khi n lớn):

* Constant time: O(1)
* Logarithmic time: O(logn)
* Linear time: O(n)
* Linearithmic time: O(nlogn)
* Quadratic time: O(n^2)
* Cubic time: O(n^3)
* Exponential time: O(2^n)
* Factorial time: O(n!)

**VD1:**

for i in range(1, n + 1):

m = m + 2

T(n) = C.n = O(n) với C là hằng số (tuỳ theo quy ước số bước)

**VD2:**

for i in range(1, n + 1):

for j in range(1, n + 1):

k = k + 1

T(n) = C.n.n = C.n^2 = O(n^2)

**VD3:**

x = x + 1

for i in range(1, n + 1):

m = m + 2

for i in range(1, n + 1):

for j in range(1, n + 1):

k = k + 1

T(n) = C0 + C1.n + C2.n^2 = O(n^2)

**VD4:**

if len(stack1) != len(stack2):

return False

else:

for n in range(len(stack1)):

if stack1[n] != stack2[n]: return False

return True

if condition: → T0(n)

statement1 → T1(n)

else: statement2 → T2(n)

T(n) = T0(n) + max(T1(n), T2(n))

T(n) = C0 + C1.n + (C2 + C3).n = O(n)

**VD5:**

i = 1

while i <= n:

i = i \* 2

Giả sử sau k lần lặp, i trở thành 2^k, khi đó: 2^k > n

→ k > log2n, do k là số nguyên nên số lần lặp tối đa là: k = log2n + 1

T(n) = log2n + 1 = O(logn)

**VD6:**

for i in range(1, n + 1):

for j in range(i + 1, n + 1):

k = k + j + i

T(n) = C.(n – 1).n = O(n^2)

**VD7:**

for i in range(4, n):

sum = a[i – 4]

for j in range(i – 3, i + 1):

sum += a[j]

T(n) = (n – 4).(C0 + 4C1) = O(n)

**VD8:**

def binary\_search(arr, key):

lo, hi = 0, len(arr) – 1

while lo <= hi:

mid = (lo + hi) // 2

if key < arr[mid]: hi = mid – 1

elif arr[mid] < key: lo = mid + 1

else: return mid

return -1

T(n) = C0 + C1.log2n = O(logn)

**VD9:**

s = 0

for i in range(n + 1):

p = 1

for j in range(1, n + 1):

p = p \* x / j

s = s + p

T(n) = C0 + (n + 1).(C1 + n.C2) = O(n^2)

**VD10:**

for i in range(1, n + 1):

for j in range(1, m + 1):

x += 2

T(n) = n.m.C = O(n.m)

Trong trường hợp xấu nhất nếu cả n và m đều lớn, số phép toán sẽ tăng bậc 2

**VD11:**

def fibonacci(n):

if n <= 1: return n

return fibonacci(n – 1) + fibonacci(n – 2)

Số lần gọi hàm tăng theo dạng nhị phân vì mỗi lời gọi sinh ra 2 lời gọi con

T(n) = T(n – 1) + T(n – 2) + O(1)

Nghiệm truy hồi: T(n) = O(2^n)

# Topic 4: SORT ALGORITHMS

**▪ Thuật toán sắp xếp so sánh cơ bản:**

**1. Selection Sort (Sắp xếp chọn):** ít dùng thực tế, chủ yếu để giảng dạy

▫ **Ý tưởng:**

- Tìm phần tử nhỏ nhất trong danh sách.

- Đổi chỗ phần tử nhỏ nhất với phần tử ở vị trí đầu tiên trong danh sách.

- Tìm phần tử nhỏ nhất trong phần còn lại của danh sách (trừ phần tử ở vị trí đầu tiên).

- Đổi chỗ phần tử nhỏ nhất với phần tử ở vị trí thứ 2 trong danh sách.

- Tiếp tục quá trình tìm kiếm và đổi chỗ phần tử nhỏ nhất cho đến khi danh sách đã được sắp xếp.

▫ **Code Python:**

def selection\_sort(arr):

for i in range(len(arr) - 1):

min\_index = i

for j in range(i + 1, len(arr)):

if arr[j] < arr[min\_index]: min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

return arr

▫ **Độ phức tạp thuật toán:** xấu nhất = tốt nhất = trung bình = O(n^2)

**2. Interchange Sort (Sắp xếp đổi chỗ):** gần như chỉ dùng để giảng dạy, hiệu suất kém

▫ **Ý tưởng:**

- Duyệt qua từng phần tử trong danh sách.

- Với mỗi phần tử, so sánh nó với tất cả các phần tử đứng sau.

- Nếu phần tử hiện tại lớn hơn phần tử sau nó, hoán đổi 2 phần tử.

- Lặp lại quá trình cho đến khi toàn bộ danh sách đã được sắp xếp.

▫ **Code Python:**

def interchange\_sort(arr):

for i in range(len(arr) – 1):

for j in range(i + 1, len(arr)):

if arr[i] > arr[j]:

arr[i], arr[j] = arr[j], arr[i]

return arr

▫ **Độ phức tạp thuật toán:** xấu nhất = tốt nhất = trung bình = O(n^2)

**3. Insertion Sort (Sắp xếp chèn):** dùng cho mảng nhỏ hoặc gần sắp xếp, hay kết hợp trong thuật toán lớn

▫ **Ý tưởng:**

- Giả sử phần tử đầu tiên đã được sắp xếp.

- Duyệt từ phần tử thứ 2 đến cuối danh sách.

- Chèn phần tử hiện tại vào đúng vị trí trong phần danh sách đã sắp xếp (di chuyển các phần tử lớn hơn phần tử cần chèn một bước về phía sau).

- Lặp lại cho đến khi toàn bộ danh sách đã được sắp xếp.

▫ **Code Python:**

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i] # phần tử cần chèn vào

j = i – 1

# Di chuyển các phần tử lớn hơn key sang phải

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key # chèn key vào đúng vị trí

return arr

▫ **Độ phức tạp thuật toán:** tốt nhất = O(n), xấu nhất = trung bình = O(n^2)

**4. Binary Insertion Sort (Sắp xếp chèn nhị phân):** cải tiến của Insertion Sort, hiếm dùng

▫ **Ý tưởng:**

- Giả sử phần tử đầu tiên đã được sắp xếp.

- Duyệt từ phần tử thứ 2 đến cuối danh sách.

- Dùng tìm kiếm nhị phân để xác định vị trí chèn của phần tử hiện tại trong phần đã sắp xếp.

- Dịch chuyển các phần tử lớn hơn về phía sau để tạo chỗ trống.

- Chèn phần tử vào vị trí đã tìm được.

- Lặp lại quá trình trên cho đến khi danh sách đã được sắp xếp hoàn toàn.

▫ **Code Python:**

def binary\_search(arr, left, right, key):

while left < right:

mid = (left + right) // 2

if arr[mid] < key: left = mid + 1

else: right = mid

return left # trả về vị trí chèn phù hợp

def binary\_insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

# Tìm vị trí chèn bằng tìm kiếm nhị phân trong phần đã sắp xếp

pos = binary\_search(arr, 0, i, key)

# Dịch chuyển các phần tử để chèn phần tử key vào đúng vị trí

j = i

while j > pos:

arr[j] = arr[j - 1]

j -= 1

arr[pos] = key # chèn phần tử vào vị trí đã tìm được

return arr

▫ **Độ phức tạp thuật toán:** tốt nhất = O(nlogn), xấu nhất = trung bình = O(n^2)

**5. Bubble Sort (Sắp xếp nổi bọt):** rất phổ biến trong giảng dạy, hiếm dùng thực tế

▫ **Ý tưởng:**

- Duyệt qua danh sách từ đầu đến cuối và so sánh từng cặp phần tử liền kề, nếu phần tử đứng trước lớn hơn phần tử đứng sau, hoán đổi chúng.

- Sau mỗi lần duyệt, phần tử lớn nhất sẽ được đẩy về cuối danh sách, giống như bong bóng nổi lên trên mặt nước.

- Lặp lại quá trình trên cho các phần tử còn lại.

- Tiếp tục duyệt danh sách cho đến khi không còn sự hoán đổi nào, nghĩa là danh sách đã được sắp xếp hoàn toàn.

▫ **Code Python:**

def bubble\_sort(arr):

for i in range(len(arr) – 1):

swapped = False

for j in range(len(arr) – 1 – i):

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

swapped = True

if not swapped: break

▫ **Độ phức tạp thuật toán:** tốt nhất = O(n), xấu nhất = trung bình = O(n^2)

**6. Shaker Sort (Sắp xếp cocktail):** cải tiến của Bubble Sort, cũng hiếm dùng thực tế

▫ **Ý tưởng:**

- Thay vì chỉ duyệt theo 1 chiều, thuật toán này duyệt theo cả 2 chiều luân phiên, giúp giảm bớt số lần hoán đổi không cần thiết.

- Lượt đầu tiên, nó di chuyển từ trái sang phải, đẩy phần tử lớn nhất về cuối danh sách.

- Lượt thứ hai, nó di chuyển từ phải sang trái, đẩy phần tử nhỏ nhất về đầu danh sách.

- Quá trình này lặp lại cho đến khi danh sách được sắp xếp hoàn toàn.

▫ **Code Python:**

def shaker\_sort(arr):

left = 0

right = len(arr) – 1

while left < right:

for i in range(left, right):

if arr[i] > arr[i + 1]:

arr[i], arr[i + 1] = arr[i + 1], arr[i]

right -= 1

for i in range(right, left, -1):

if arr[i] < arr[i – 1]:

arr[i], arr[i – 1] = arr[i – 1], arr[i]

left += 1

return arr

▫ **Độ phức tạp thuật toán:** tốt nhất = O(n), xấu nhất = trung bình = O(n^2)

**▪ Thuật toán sắp xếp chia để trị:**

**7. Quick Sort (Sắp xếp nhanh):** rất phổ biến, hiệu suất tốt, dùng trong thư viện chuẩn

▫ **Ý tưởng:**

- Chọn một phần tử gọi là pivot (chốt) (có thể là phần tử đầu, cuối, giữa hoặc ngẫu nhiên).

- Chia danh sách thành 2 phần: nhóm nhỏ hơn pivot và nhóm lớn hơn pivot.

- Đệ quy tiếp tục sắp xếp 2 nhóm con.

- Ghép các nhóm lại để tạo danh sách đã sắp xếp.

- Dừng khi danh sách chỉ còn 1 phần tử hoặc rỗng.

▫ **Code Python:**

def quick\_sort(arr, left, right):

if left >= right: return

i, j = left, right

pivot = arr[(left + right) // 2] # chọn pivot là phần tử giữa danh sách

while i <= j:

while arr[i] < pivot: i += 1

while arr[j] > pivot: j -= 1

if i <= j:

arr[i], arr[j] = arr[j], arr[i]

i += 1

j -= 1

if left < j: quick\_sort(arr, left, j)

if i < right: quick\_sort(arr, i, right)

▫ **Độ phức tạp thuật toán:** xấu nhất = O(n^2), tốt nhất = trung bình = O(nlogn)

**8. Merge Sort (Sắp xếp trộn):** phổ biến với dữ liệu lớn và cần độ ổn định

▫ **Ý tưởng:**

- Chia: chia danh sách thành 2 nửa bằng nhau (hoặc gần bằng nhau).

- Trị: đệ quy sắp xếp từng nửa.

- Kết hợp: gộp 2 danh sách đã sắp xếp thành 1 danh sách hoàn chỉnh.

▫ **Code Python:**

def merge(left, right):

sorted\_lst = []

i = j = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

sorted\_lst.append(left[i])

i += 1

else:

sorted\_lst.append(right[j])

j += 1

sorted\_lst.extend(left[i:])

sorted\_lst.extend(right[j:])

return sorted\_lst

def merge\_sort(arr):

if len(arr) <= 1: return arr

mid = len(arr) // 2

left\_half = merge\_sort(arr[:mid])

right\_half = merge\_sort(arr[mid:])

return merge(left\_half, right\_half)

▫ **Độ phức tạp thuật toán:** xấu nhất = tốt nhất = trung bình = O(nlogn)

**▪ Thuật toán sắp xếp dựa trên cấu trúc dữ liệu:**

**9. Heap Sort (Sắp xếp vun đống):** dùng trong vài trường hợp đặc biệt, không phổ biến lắm

▫ **Ý tưởng:**

- Dựa trên cấu trúc dữ liệu Heap (một dạng Binary Tree đặc biệt, thường là Max Heap, ngược lại là Min Heap)

- Xây dựng Max Heap từ mảng đầu vào (tức là cây nhị phân thoả mãn tính chất node cha lớn hơn node con) (hàm heapify).

- Phần tử lớn nhất luôn nằm ở gốc Heap (arr[0]).

- Hoán đổi phần tử lớn nhất với phần tử cuối mảng, sau đó giảm kích thước Heap đi 1.

- Gọi heapify để tái lập tính chất Max Heap sau mỗi lần hoán đổi.

- Lặp lại quá trình trên cho đến khi toàn bộ mảng đã được sắp xếp.

A diagram of numbers and circles

AI-generated content may be incorrect.

▫ **Code Python:**

def heapify(arr, n, i):

largest = i # giả sử node hiện tại là lớn nhất

left = 2 \* i + 1 # chỉ số con trái

right = 2 \* i + 2 # chỉ số con phải

# Nếu con trái lớn hơn node hiện tại (bố nó)

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

# Nếu con phải lớn hơn node hiện tại (bố nó)

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

# Nếu largest không phải node hiện tại, hoán đổi và tiếp tục heapify

if largest != i:

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

heapify(arr, n, largest)

def heap\_sort(arr):

# Xây dựng Max Heap (chạy heapify từ các node không phải lá)

for i in range(len(arr) // 2 – 1, -1, -1): # giữa mảng về đầu mảng

heapify(arr, len(arr), i)

# Trích xuất từng phần tử từ Heap

for i in range(len(arr) – 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i] # đưa max về cuối mảng

heapify(arr, i, 0)

▫ **Độ phức tạp thuật toán:** xấu nhất = tốt nhất = trung bình = O(nlogn)

**▪ Thuật toán sắp xếp không dựa trên so sánh:**

**10. Radix Sort (Sắp xếp cơ số):** chỉ dùng với số nguyên, ứng dụng đặc thù

▫ **Ý tưởng:**

- Khởi tạo 10 danh sách (buckets) ứng với các chữ số từ 0-9.

- Lặp qua m chữ số từ hàng đơn vị đến hàng lớn nhất của số lớn nhất trong dãy.

- Với mỗi chữ số t, đưa từng phần tử vào bucket tương ứng với chữ số t của nó rồi lần lượt gom các phần tử từ bucket vào mảng ban đầu.

- Tiếp tục cho đến khi tất cả chữ số được xử lí.

▫ **Code Python:**

def get\_digit(num, digit\_index):

return (num // (10 \*\* digit\_index)) % 10

def radix\_sort(arr):

max\_val = max(arr)

max\_digits = len(str(max\_val))

for digit\_index in range(max\_digits):

buckets = [[] for \_ in range(10)]

for num in arr:

digit = get\_digit(num, digit\_index)

buckets[digit].append(num)

arr.clear()

for bucket in buckets: arr.extend(bucket)

▫ **Độ phức tạp thuật toán:** xấu nhất = tốt nhất = trung bình =O(nk) với k số chữ số lớn nhất (max\_digits)

# Topic 5: SEARCH ALGORITHMS & HASING

**▪ Các thuật toán tìm kiếm hay dùng:**

- **Linear/Sequential Search (1)**: đơn giản, dễ cài đặt, dùng cho tập dữ liệu nhỏ hoặc chưa sắp xếp, không phổ biến với dữ liệu lớn.

- **Binary Search (2)**: nhanh, hiệu quả, rất phổ biến, dùng rộng rãi cho mảng đã sắp xếp hoặc trong các thuật toán chia để trị.

- **Hashing (3)**: cực kì phổ biến trong thực tế, dùng Hash Table (Bảng Băm), tra cứu nhanh.

- **Search Tree (tìm hiểu sau)**: sẽ phổ biến nếu triển khai dạng cân bằng như AVL, Red-Black Tree, BST,… dùng trong cơ sở dữ liệu, compiler,…

❶ **Linear/Sequential Search (Tìm kiếm tuyến tính)**

▫ **Ý tưởng:**

- Duyệt tuần tự từng phần tử trong mảng.

- So sánh từng phần tử với giá trị cần tìm.

- Nếu tìm thấy thì trả về vị trí (index).

- Nếu duyệt hết mà không thấy thì trả về -1.

▫ **Code Python:**

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target: return i

return -1

▫ **Độ phức tạp thuật toán:** tốt nhất = O(1), xấu nhất = trung bình = O(n)

❷ **Binary Search (Tìm kiếm nhị phân)**

▫ **Ý tưởng:**

- Áp dụng cho mảng đã được sắp xếp, duyệt theo chiến lược chia để trị.

- So sánh phần tử ở giữa với giá trị cần tìm.

- Nếu bằng thì trả về vị trí ở giữa luôn.

- Nếu nhỏ hơn thì tiếp tục tìm ở nửa bên phải.

- Nếu lớn hơn thì tiếp tục tìm ở nửa bên trái.

- Lặp lại đến khi tìm thấy hoặc không còn phần tử nào.

▫ **Code Python:**

def binary\_search(arr, target):

left, right = 0, len(arr) – 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target: return mid

elif arr[mid] < target: left = mid + 1

else: right = mid – 1

return -1

# Phiên bản sử dụng đệ quy

def binary\_search\_recursive(arr, target, left, right):

if left > right: return -1

mid = (left + right) // 2

if arr[mid] == target: return mid

elif arr[mid] < target:

return binary\_search\_recursive(arr, target, mid + 1, right)

else:

return binary\_search\_recursive(arr, target, left, mid – 1)

▫ **Độ phức tạp thuật toán:** tốt nhất = O(1), xấu nhất = trung bình = O(logn)

❸ **Hashing (Băm) và Hash Table (Bảng Băm)**

▫ **Các khái niệm:**

- **Cơ sở dữ liệu**: bảng băm được sử dụng trong các hệ quản trị cơ sở dữ liệu để tìm kiếm nhanh dữ liệu.

- **Tìm kiếm và tra cứu**: bảng băm được sử dụng để tìm kiếm nhanh các phần tử trong tập dữ liệu lớn. Ví dụ, trong các công cụ tìm kiếm web, bảng băm được sử dụng để lưu trữ và tìm kiếm các từ khoá.

- **Xác thực và bảo mật**: mật khẩu người dùng thường được lưu trữ dưới dạng giá trị băm trong cơ sở dữ liệu để đảm bảo tính an toàn và riêng tư.

- **Mã hoá**: bảng băm được sử dụng trong các thuật toán mã hoá để tạo chữ kí số và băm các thông tin nhạy cảm. Ví dụ, thuật toán SHA-256 sử dụng bảng băm để tạo giá trị băm 256-bit cho dữ liệu đầu vào.

- **Tối ưu hoá và tìm kiếm trong đồ thị**: các thuật toán như Dijkstra và A\* sử dụng bảng băm để lưu trữ và truy xuất các đỉnh và cạnh trong đồ thị.

- **Key (khoá)**: là giá trị duy nhất được sử dụng để xác định vị trí lưu trữ và tìm kiếm trong bảng băm.

- **Hash table (bảng băm)**: thường có dạng mảng có kích thước cố định (N phần tử), trong đó mỗi phần tử của mảng là một bucket hoặc một singly linked list. Hash table cho phép lưu trữ, tìm kiếm và xoá phần tử với độ phức tạp trung bình là O(1) nếu hàm băm hoạt động tốt và ít xảy ra va chạm.

- **Hash map (bản đồ băm)**: là cấu trúc dữ liệu ánh xạ khoá đến giá trị thông qua hàm băm, mỗi phần tử trong hash map là một cặp (key, value) (dictionary). Hash map cho phép truy xuất, chèn và xoá dữ liệu hiệu quả với độ phức tạp trung bình O(1).

- **Hash function (hàm băm)**: là một hàm được sử dụng để chuyển đổi khoá thành một chỉ số trong mảng lưu trữ của bảng băm. Hàm băm cần phải có tính chất phân tán tốt để tránh xung đột khoá và giúp phân tán dữ liệu đều trong bảng.

- **Collision (đụng độ)**: xảy ra khi 2 khoá được ánh xạ đến cùng 1 vị trí trong mảng. Điều này có thể xảy ra do hàm băm không phân tán tốt hoặc do kích thước mảng không đủ lớn để chứa tất cả các khoá.

- **Collision resolution (giải quyết đụng độ)**: là quá trình giải quyết xung đột khi 2 khoá ánh xạ đến cùng 1 vị trí. Có nhiều phương pháp để giải quyết xung đột, bao gồm nối kết và băm lại.

- **Load (tải)**: là tỉ lệ giữa số lượng phần tử hiện có trong bảng băm và kích thước của mảng. Tải cao có thể dẫn đến hiện tượng xung đột tăng và làm giảm hiệu suất của bảng băm.

▫ **Hash function:**

Hàm băm hay gặp nhất là băm kiểu modulo như sau: **H(k) = k mod N**

∙ **k** là khoá (số nguyên muốn băm).

∙ **N** là kích thước của bảng băm, thường là một số nguyên tố để giảm va chạm.

∙ **H(k)** là index trong mảng (bảng băm) mà khoá được ánh xạ tới.

Code Python:

def hash\_modulo(k, N):

return k % N

**VD1:** k = 256, N = 17, H(256) = 256 mod 17 = 1

Đưa 256 vào vị trí có index = 1

**VD2:** k = 26, N = 17, H(26) = 26 mod 17 = 9

Đưa 26 vào vị trí có index = 9

Tuy nhiên với k = 60, N = 17 thì H(60) = 60 mod 17 = 9

Ta thấy 2 khoá cùng 1 địa chỉ, điều này xảy ra va chạm giữa các khoá với nhau. Vậy làm cách nào để xử lý chúng?

▫ **Collision resolution:**

Trong thực tế, người ta giải quyết đụng độ theo 2 phương pháp: **phương pháp nối kết** và **phương pháp băm lại**

**1. Phương pháp nối kết (Chaining)**

**1.1. Phương pháp nối kết trực tiếp (Separate Chaining)**

- Mỗi ô trong bảng băm là con trỏ đến danh sách liên kết. Ban đầu, các con trỏ này được gán là NULL, vì chưa có phần tử nào.

- Thêm khoá vào vị trí index tương ứng với **H(k) = k mod N**, nếu các khoá bị va chạm thì thêm vào cuối danh sách tại chỉ số đó.

- Khi tìm kiếm, tính chỉ sốH(k), rồi duyệt danh sách tại vị trí đó để tìm khoá.

- Thời gian tìm kiếm phụ thuộc vào độ dài danh sách (O(1) nếu băm đều, O(N) nếu không đều). Tốn thêm bộ nhớ để cấp phát danh sách liên kết.

- VD: Cho bảng băm có N = 5, hàm băm H(k) = k % N, xử lý đụng độ bằng phương pháp nối kết trực tiếp. Lần lượt thêm các khoá 25, 7, 2, 10, 40, 12, 27, 9, 28, 3 vào bảng. Ta sẽ được kết quả:

A diagram of a flowchart

AI-generated content may be incorrect.

**1.2. Phương pháp nối kết hợp nhất (Coalesced Chaining)**

- Bảng băm được cài bằng mảng gồm N node, mỗi node gồm 2 trường là key (lưu khoá) và next (chỉ số node tiếp theo nếu có đụng độ). Ban đầu, tất cả key được gán NULL, next là -1.

- Khi thêm một khoá k, hàm băm **H(k) = k mod N** xác định vị trí index. Nếu ô index đó trống thì chèn tại đó. Còn nếu ô đó đã có dữ liệu (đụng độ) thì tìm ô trống ở cuối mảng, chèn vào đó và cập nhật next của ô index để nối sang ô mới.

- Khi tìm kiếm, bắt đầu từ vị trí H(k) và lần theo các trường next cho đến khi gặp khoá cần tìm hoặc hết danh sách.

- Thời gian tìm kiếm nhanh (O(1)) nếu phân bố băm đều, nhưng có thể chậm (O(N)) nếu xảy ra nhiều va chạm. Cần thêm mảng phụ (next) để lưu chỉ số liên kết, làm tăng độ phức tạp và sử dụng bộ nhớ.

- VD: Cho bảng băm có N = 6, lần lượt thêm các khoá 12, 18, 24, 1, 49 vào bảng. Sử dụng hàm băm H(k) = k % N, giải quyết đụng độ bằng phương pháp nối kết hợp nhất. Ta sẽ được kết quả:

A blue arrow pointing to a blue rectangle

AI-generated content may be incorrect.

**2. Phương pháp băm lại (Open Addressing)**

**2.1. Phương pháp dò tuyến tính (Linear Probing)**

- Bảng băm gồm N ô, mỗi ô có thể chứa một khoá hoặc là NULL nếu trống. Khi khởi tạo, tất cả ô đều được gán NULL.

- Tính chỉ số ban đầu **H(k) = k mod N**, nếu ô tại chỉ số đó trống thì chèn khoá vào.

- Nếu ô đã có khoá thì dò tuyến tính ô tiếp theo theo công thức: **h(k, i) = (H(k) + i) mod N** với i = 1, 2, 3,… Tiếp tục đến khi gặp ô trống để chèn khoá vào. Nếu dò đến cuối bảng thì quay lại đầu bảng (vị trí 0).

- Khi tìm kiếm, bắt đầu từ H(k), nếu không phải thì dò tuyến tính theo h(k, i) cho đến khi thấy khoá hoặc gặp ô trống (kết luận không tồn tại).

- Nếu khoá được phân bố đều thì truy xuất rất nhanh (O(1)). Nếu xảy ra nhiều va chạm thì có thể phải duyệt gần như toàn bộ bảng (O(N)).

- VD: Cho bảng băm ở trạng thái vừa khởi tạo có N = 5 node, lần lượt thêm các khoá 27, 32, 46, 51, 14 vào bảng. Hàm băm H(k) = k % N, xử lý đụng độ bằng phương pháp dò tuyến tính. Ta có:

**Thêm khoá 27:** H(27) = 27 % 5 = 2 → Vị trí trống, điền vô

A white rectangular box with black text

AI-generated content may be incorrect.

**Thêm khoá 32:** H(32) = 32 % 5 = 2 → Xảy ra đụng độ, băm lại

h(32, 1) = (2 + 1) % 5 = 3 → Vị trí trống, điền vô

**Thêm khoá 46:** H(46) = 46 % 5 = 1 → Vị trí trống, điền vô

A white rectangular box with black text

AI-generated content may be incorrect.

**Thêm khoá 51:** H(51) = 51 % 5 = 1 → Xảy ra đụng độ, băm lại

h(51, 1) = (1 + 1) % 5 = 2 → Xảy ra đụng độ, băm lại lần 2

h(51, 2) = (1 + 2) % 5 = 3 → Xảy ra đụng độ, băm lại lần 3

h(51, 3) = (1 + 3) % 5 = 4 → Vị trí trống, điền vô

A screenshot of a table

AI-generated content may be incorrect.

**Thêm khoá 14:** H(14) = 14 % 5 = 4 → Xảy ra đụng độ, băm lại

h(14, 1) = (4 + 1) % 5 = 0 → Vị trí trống, điền vô

A screenshot of a graph

AI-generated content may be incorrect.

**2.2. Phương pháp dò bậc hai (Quadratic Probing)**

- Bảng băm gồm N ô, mỗi ô có thể chứa một khoá hoặc là NULL nếu trống. Khi khởi tạo, tất cả các ô đều được gán NULL.

- Tính chỉ số ban đầu **H(k) = k mod N**. Nếu ô tại chỉ số đó trống thì chèn khoá vào.

- Nếu ô đã có khoá thì dò bậc hai các ô tiếp theo theo công thức **h(k, i) = (H(k) + i²) mod N**, với i = 1, 2, 3,… Tiếp tục tăng i và tính theo công thức cho đến khi gặp ô trống để chèn khoá vào. Nếu vượt quá cuối bảng thì sẽ quay vòng lại đầu bảng nhờ phép toán mod.

- Khi tìm kiếm, bắt đầu từ H(k), nếu không phải thì tiếp tục dò bậc hai theo h(k, i) cho đến khi gặp khoá hoặc gặp ô NULL (kết luận không tồn tại).

- Phương pháp dò bậc hai giúp giảm hiện tượng tích tụ va chạm (primary clustering), nên hiệu quả cao hơn dò tuyến tính trong nhiều trường hợp. Tuy nhiên, cần đảm bảo kích thước bảng đủ lớn và chọn hệ số phù hợp để tránh rơi vào vòng lặp vô tận.

- Truy xuất tốt trong điều kiện các khoá phân bố đều. Trong trường hợp xấu nhất, độ phức tạp vẫn có thể lên đến O(N) nếu bảng gần đầy.

- VD: Cho bảng băm ở trạng thái vừa khởi tạo có N = 11 node, lần lượt thêm các khoá 31, 19, 2, 13, 25, 24, 21, 9 vào bảng. Hàm băm H(k) = k % N, xử lý đụng độ bằng phương pháp dò bậc hai. Ta có:

**Thêm khoá 31:** H(31) = 31 % 11 = 9 → Vị trí trống, điền vô

**Thêm khoá 19:** H(19) = 19 % 11 = 8 → Vị trí trống, điền vô

**Thêm khoá 2:** H(2) = 2 % 11 = 2 → Vị trí trống, điền vô

A table of numbers with black text

AI-generated content may be incorrect.

**Thêm khoá 13:** H(13) = 13 % 11 = 2 → Xảy ra đụng độ, băm lại

H(13, 1) = (2 + 1^2) % 11 = 3 → Vị trí trống, điền vô

A table of numbers with black text

AI-generated content may be incorrect.

**Thêm khoá 25:** H(25) = 25 % 11 = 3 → Xảy ra đụng độ, băm lại

H(25, 1) = (3 + 1^2) % 11 = 4 → Vị trí trống, điền vô

A table of numbers with black text

AI-generated content may be incorrect.

**Thêm khoá 24:** H(24) = 24 % 11 = 2 → Xảy ra đụng độ, băm lại

H(24, 1) = (2 + 1^2) % 11 = 3 → Xảy ra đụng độ, băm lại lần 2

H(24, 2) = (2 + 2^2) % 11 = 6 → Vị trí trống, điền vô

**Thêm khoá 21:** H(21) = 21 % 11 = 10 → Vị trí trống, điền vô

A table with numbers and letters

AI-generated content may be incorrect.

**Thêm khoá 9:** H(9) = 9 % 11 = 9 → Xảy ra đụng độ, băm lại

H(9, 1) = (9 + 1^2) % 11 = 10 → Xảy ra đụng độ, băm lại lần 2

H(9, 2) = (9 + 2^2) % 11 = 2 → Xảy ra đụng độ, băm lại lần 3

H(9, 3) = (9 + 3^2) % 11 = 7 → Vị trí trống, điền vô

A table of numbers with numbers

AI-generated content may be incorrect.

**2.3. Phương pháp băm kép (Double Hashing)**

- Bảng băm gồm N ô, mỗi ô có thể chứa một khoá hoặc là NULL nếu trống. Khi khởi tạo, tất cả các ô đều được gán NULL.

- Tính chỉ số ban đầu **H₁(k) = k mod N**. Nếu ô tại chỉ số đó trống thì chèn khoá vào.

- Nếu ô đã có khoá, sử dụng một hàm băm thứ hai (bất kì) để tính khoảng cách bước nhảy như **H2(k) = (N – 2) – k % (N – 2)** hay **H₂(k) = 1 + (k mod (N - 2))**…

- Dò tiếp các vị trí theo công thức **h(k, i) = (H₁(k) + i \* H₂(k)) mod N**, với i = 1, 2, 3,… Tiếp tục tính và kiểm tra các chỉ số mới cho đến khi gặp ô trống để chèn khoá vào.

- Khi tìm kiếm, bắt đầu từ H₁(k), nếu không đúng thì tiếp tục dò theo h(k, i) cho đến khi gặp khoá cần tìm hoặc gặp ô NULL (kết luận không tồn tại).

- Phương pháp băm kép giúp giảm đáng kể hiện tượng tích tụ va chạm vì các bước nhảy khác nhau cho mỗi khoá, làm cho các chuỗi dò ít khi trùng nhau.

- Độ phức tạp truy xuất trung bình O(1) nếu bảng chưa đầy. Trong trường hợp xấu nhất (bảng đầy hoặc gần đầy), độ phức tạp có thể lên đến O(N).

- Để đảm bảo hoạt động hiệu quả, nên chọn N là số nguyên tố và H₂(k) luôn khác 0 với mọi khoá.

- VD: Cho bảng băm có N = 5, hàm băm H1(k) = k % N, H2(k) = (N – 2) – k % (N – 2), xử lý đụng độ bằng phương pháp băm kép. Lần lượt thêm các khoá 27, 32, 46, 51, 14 vào bảng. Ta có:

**Thêm khoá 27:** H1(27) = 27 % 5 = 2 → Vị trí trống, điền vô

A blue rectangular object with black text

AI-generated content may be incorrect.

**Thêm khoá 32:** H1(32) = 32 % 5 = 2 → Xảy ra đụng độ, băm lại

H2(32) = (5 – 2) – 32 % (5 – 2) = 1

h(32, 1) = (2 + 1 \* 1) % 5 = 3 → Vị trí trống, điền vô

**Thêm khoá 46:** H1(46) = 46 % 5 = 1 → Vị trí trống, điền vô

A screenshot of a computer

AI-generated content may be incorrect.

**Thêm khoá 51:** H1(51) = 51 % 5 = 1 → Xảy ra đụng độ, băm lại

H2(51) = (5 – 2) – 51 % (5 – 2) = 3

h(51, 1) = (1 + 1 \* 3) % 5 = 4 → Vị trí trống, điền vô

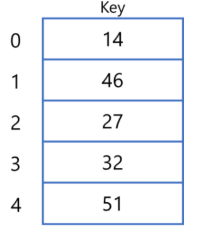
A screenshot of a table

AI-generated content may be incorrect.

**Thêm khoá 14:** H1(14) = 14 % 5 = 4 → Xảy ra đụng độ, băm lại

H2(14) = (5 – 2) – 14 % (5 – 2) = 1

h(51, 1) = (4 + 1 \* 1) % 5 = 0 → Vị trí trống, điền vô



# Topic 6: TREE

**▪ Tree (Cây):**

**a) Khái niệm cây:** Cây là một tập hợp các phần tử hay còn gọi là các node, gồm:

∙ Node gốc (root)

∙ Các node còn lại chia thành các tập con T1, T2, …, Tn (Ti là các cây con)

A diagram of a network

AI-generated content may be incorrect.

**b) Tính chất cây:**

- **Bậc (Degree) của node**: Là số cây con của node đó. Ở VD trên, bậc của A là 3, của B là 2, của C là 0.

- **Bậc (Degree) của cây**: Là bậc của node có bậc lớn nhất trên cây, cây có bậc n gọi là cây n-phân. Ở VD trên, bậc của cây là 3, gọi là cây tam phân.

- **Node lá**: Là node có bậc bằng 0 (không có cây con nào). Ở VD trên, có G, K là node lá.

- **Node nhánh**: Là node có bậc khác 0 và không phải node gốc (node trung gian). Ở VD trên, có B, D, H là node nhánh.

- **Mức (Level) của node**: Xét mức của gốc T0 = 0, xét T1, T2, …, Tn  là các cây con của T0, khi đó: Mức (T1) = Mức (T2) = … = Mức (Tn) = Mức (T0) + 1. Ở VD trên, mức của node A là 0, mức của B, C, D là 1, mức của G, K là 3.

- **Chiều cao (Height)**: Tổng số node lớn nhất từ node gốc đến các node lá (mức lớn nhất của các node lá cộng thêm 1). Ở VD trên, cây có chiều cao là 4.

- **Đường đi (Path) từ node u đến node v**: dãy liên tiếp các cạnh hoặc các node bắt đầu từ u, kết thúc tại v, trong đó mỗi cạnh nối 2 node liên tiếp. Ở VD trên, đường đi từ node A đến node F là A -> B -> F.

**c) Phân loại cây:** Các loại cây thường gặp bao gồm:

Nhị phân: Binary Tree --> BST --> Balanced BST --> AVL (tập trung dòng này)

--> AA Tree

--> Red-Black Tree ⇄ 2-3-4 Tree --> 2-3 Tree

Đa phân: M-ary Tree --> M-way Search Tree --> B-tree

Đặc biệt: Special Tree --> Heap

❶ **Binary Tree (Cây nhị phân)**

**1. Khái niệm cây nhị phân:** Là cây mà mỗi node có tối đa 2 cây con (hoặc là cây rỗng).

A diagram of a network

AI-generated content may be incorrect.

**2. Các thao tác trên cây nhị phân:**

**a. Biểu diễn cây nhị phân:**

class Node: # định nghĩa node trong cây

def \_\_init\_\_(self, data):

self.data = data # giá trị của node

self.left = None # con trái

self.right = None # con phải

**3. Duyệt cây nhị phân:**

- Có 3 phép duyệt chính:

+ Duyệt trước (Pre-order): **Node – Left – Right**

+ Duyệt giữa (In-order): **Left – Node – Right**

+ Duyệt sau (Post-order): **Left – Right – Node**

+ Ngoài ra còn có Duyệt theo mức (Level-order): theo từng mức từ trên xuống (BFS)

- Các phép duyệt cây cho độ phức tạp O(log2h) với h là chiều cao của cây.

- VD:

A diagram of numbers and circles

AI-generated content may be incorrect.

+ NLR: **27, 18, 48, 13, 25, 1, 9**

+ LNR: **48, 18, 13, 27, 1, 25, 9**

+ LRN: **48, 13, 18, 1, 9, 25, 27**

- Code Python:

def NLR(node):

if node is not None:

print(node.data, end=' ')

NLR(node.left)

NLR(node.right)

def LNR(node):

if node is not None:

LNR(node.left)

print(node.data, end=' ')

LNR(node.right)

def LRN(node):

if node is not None:

LRN(node.left)

LRN(node.right)

print(node.data, end=' ')

from collections import deque

def BFS(root):

if root is None: return

queue = deque()

queue.append(root)

while queue:

node = queue.popleft()

print(node.data, end=' ')

if node.left: queue.append(node.left)

if node.right: queue.append(node.right)

❷ **Binary Search Tree (BST – Cây nhị phân tìm kiếm)**

**1. Khái niệm BST:** Là cây nhị phân, đảm bảo nguyên tắc bố trí khoá tại mỗi node sao cho:

- Các node trong cây con bên trái nhỏ hơn node hiện hành.

- Các node trong cây con bên phải lớn hơn node hiện hành.

A diagram of numbers and circles

AI-generated content may be incorrect.

**2. Các thao tác trên BST:**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BST:

    def \_\_init\_\_(self):

        self.root = None

    def is\_empty(self):

        return self.root is None

    def find(self, key):

        return self.\_find(self.root, key)

    def \_find(self, node, key):

        if node is None: return False

        if key == node.key: return True

        elif key < node.key: return self.\_find(node.left, key)

        else: return self.\_find(node.right, key)

    def insert(self, key):

        self.root = self.\_insert(self.root, key)

    def \_insert(self, node, key):

        if node is None: return Node(key)

        if key < node.key: node.left = self.\_insert(node.left, key)

        elif key > node.key: node.right = self.\_insert(node.right, key)

        return node

    def delete(self, key):

        self.root = self.\_delete(self.root, key)

    def \_delete(self, node, key):

        if node is None: return None

        if key < node.key: node.left = self.\_delete(node.left, key)

        elif key > node.key: node.right = self.\_delete(node.right, key)

        else:

            # Trường hợp node có 1 con hoặc không có con

            if node.left is None: return node.right

            elif node.right is None: return node.left

            # Trường hợp node có 2 con

            # Tìm phần tử nhỏ nhất ở cây con bên phải để thay thế

            min\_larger\_node = self.\_find\_min(node.right)

            node.key = min\_larger\_node.key

            node.right = self.\_delete(node.right, min\_larger\_node.key)

        return node

    def \_find\_min(self, node):

        current = node

        while current.left is not None:

            current = current.left

        return current

    # Duyệt cây theo thứ tự in-order để in ra các phần tử

    def inorder\_traversal(self):

        result = []

        self.\_inorder(self.root, result)

        return result

    def \_inorder(self, node, result):

        if node:

            self.\_inorder(node.left, result)

            result.append(node.key)

            self.\_inorder(node.right, result)

❸ **Balanced BST (Cây nhị phân tìm kiếm cân bằng)**

**1. Khái niệm Balanced BST:**

Một cây BST được gọi là “balanced” nếu:

- Độ cao của 2 cây con trái và phải của bất kì node nào không chênh lệch quá 1.

- Tổng thể cây có chiều cao khoảng log2N.

Một số biến thể nổi bật của Balanced BST:

- **AVL**:cân bằng rất nghiêm ngặt (hiệu độ cao <= 1), cân bằng sau mỗi phép chèn/xoá, thích hợp cho các hệ thống yêu cầu nhiều thao tác tìm kiếm nhanh và ít cập nhật như hệ thống từ điển, bảng kí hiệu.

- **2-3 Tree**: mỗi node chứa 1 hoặc 2 key, có 2 hoặc 3 con, luôn cân bằng tuyệt đối, thường dùng trong lý thuyết và làm nền tảng để triển khai các cấu trúc như Red-Black Tree.

- **2-3-4 Tree**: mở rộng của 2-3 Tree, mỗi node có thể chứa 1-3 key, tương ứng có 2-4 con, đảm bảo cân bằng tuyệt đối, dùng trong hệ thống tệp và cơ sở dữ liệu nhỏ cần cây tìm kiếm cân bằng chặt.

- **Red-Black Tree**: là biểu diễn nhị phân của 2-3-4 Tree, cân bằng lỏng lẻo, dễ cài đặt hơn AVL, dùng trong STL (Standard Template Library – Thư viện chuẩn trong C++).

**2. Các thao tác trên Balanced BST:**

**a. Tìm kiếm (Search):**

- Giống BST thường: so sánh giá trị và đi trái/phải.

- Thời gian trung bình: O(logN) (nhờ cây được cân bằng).

**b. Thêm node (Insert):**

- Chèn giống BST: đi đến vị trí phù hợp và chèn node mới.

- Sau khi chèn xong, nếu cây bị mất cân bằng, ta phải thực hiện các phép xoay (rotation) để phục hồi. Khi chênh lệch chiều cao giữa 2 nhánh của 1 node > 1, thì sẽ rơi vào 1 trong 4 trường hợp sau:

+ Left – Left (LL) thì Right Rotation.

+ Right – Right (RR) thì Left Rotation.

+ Left – Right (LR) thì Left → Right Rotation.

+ Right – Left (RL) thì Right → Left Rotation.

**c. Xoá node (Delete):**

- Xoá tương tự BST:

+ Nếu node là lá → xóa thẳng.

+ Nếu có 1 con → thay thế bằng con đó.

+ Nếu có 2 con → tìm node kế tiếp (in-order successor) và thay thế.

- Sau khi xoá, cây có thể mất cân bằng, và ta cũng cần rotation để cân bằng lại giống như lúc chèn.

**d. Duyệt cây (Traversal):**

- Duyệt theo kiểu in-order sẽ trả về các phần tử theo thứ tự tăng dần.

- Các kiểu duyệt khác: pre-order, post-order, level-order (BFS).

❹ **AVL (Cây AVL)**

**1. Khái niệm AVL:** Là một cây nhị phân tìm kiếm tự cân bằng, nghĩa là nó đảm bảo rằng độ cao của cây không bị lệnh quá nhiều về một phía. Với mỗi node trong cây, hiệu chiều cao của 2 cây con trái và phải luôn <= 1. Nếu điều kiện này bị vi phạm sau khi thêm hoặc xoá node, ta phải thực hiện quay (rotation) để cân bằng lại cây.

**2. Các thao tác trên AVL:**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1 # chiều cao (ban đầu là 1)

class AVL:

    def get\_height(self, root):

        if not root: return 0

        return root.height

    def get\_balance(self, root): # tính hiệu chiều cao con trái và con phải

        if not root: return 0

        return self.get\_height(root.left) - self.get\_height(root.right)

    def right\_rotate(self, z): # quay phải

        y = z.left

        T3 = y.right

        # Thực hiện quay

        y.right = z

        z.left = T3

        # Cập nhật chiều cao

        z.height = 1 + max(self.get\_height(z.left), self.get\_height(z.right))

        y.height = 1 + max(self.get\_height(y.left), self.get\_height(y.right))

        return y

    def left\_rotate(self, z): # quay trái

        y = z.right

        T2 = y.left

        # Thực hiện quay

        y.left = z

        z.right = T2

        # Cập nhật chiều cao

        z.height = 1 + max(self.get\_height(z.left), self.get\_height(z.right))

        y.height = 1 + max(self.get\_height(y.left), self.get\_height(y.right))

        return y

    def insert(self, root, key):

        # Chèn như BST

        if not root: return Node(key)

        elif key < root.key: root.left = self.insert(root.left, key)

        else: root.right = self.insert(root.right, key)

        # Cập nhật chiều cao

        root.height = 1 + max(self.get\_height(root.left), self.get\_height(root.right))

        # Kiểm tra cân bằng

        balance = self.get\_balance(root)

        # Left Left

        if balance > 1 and key < root.left.key:

            return self.right\_rotate(root)

        # Right Right

        if balance < -1 and key > root.right.key:

            return self.left\_rotate(root)

        # Left Right

        if balance > 1 and key > root.left.key:

            root.left = self.left\_rotate(root.left)

            return self.right\_rotate(root)

        # Right Left

        if balance < -1 and key < root.right.key:

            root.right = self.right\_rotate(root.right)

            return self.left\_rotate(root)

        return root

    def get\_min\_value\_node(self, root):

        current = root

        while current.left:

            current = current.left

        return current

    def delete(self, root, key):

        if not root: return root

        if key < root.key: root.left = self.delete(root.left, key)

        elif key > root.key: root.right = self.delete(root.right, key)

        else:

            # Node có 1 con hoặc không có con

            if not root.left: return root.right

            elif not root.right: return root.left

            # Node có 2 con

            temp = self.get\_min\_value\_node(root.right)

            root.key = temp.key

            root.right = self.delete(root.right, temp.key)

        # Cập nhật chiều cao

        root.height = 1 + max(self.get\_height(root.left), self.get\_height(root.right))

        # Kiểm tra cân bằng

        balance = self.get\_balance(root)

        # Left Left

        if balance > 1 and self.get\_balance(root.left) >= 0:

            return self.right\_rotate(root)

        # Left Right

        if balance > 1 and self.get\_balance(root.left) < 0:

            root.left = self.left\_rotate(root.left)

            return self.right\_rotate(root)

        # Right Right

        if balance < -1 and self.get\_balance(root.right) <= 0:

            return self.left\_rotate(root)

        # Right Left

        if balance < -1 and self.get\_balance(root.right) > 0:

            root.right = self.right\_rotate(root.right)

            return self.left\_rotate(root)

        return root

    # Duyệt cây theo thứ tự in-order

    def in\_order(self, root):

        if not root: return

        self.in\_order(root.left)

        print(root.key, end=' ')

        self.in\_order(root.right)

# Topic 7: GRAPH

▪ **Đồ thị và các khái niệm liên quan:**

**1. Đồ thị (Graph) là gì?**

- **Đồ thị (graph)** trong lĩnh vực cấu trúc dữ liệu là một cấu trúc dữ liệu phi tuyến tính (non-linear) bao gồm tập các đỉnh (vertex) và tập các cạnh (edge). Kí hiệu: **G(V, E)**.

- **Đỉnh (vertex ~ node, point)**: là đơn vị cơ bản của đồ thị, mỗi node có thể được dán nhãn hoặc không.

- **Cạnh (edge ~ link, line)**: cạnh trong đồ thị thể hiện sự kết nối giữa 2 đỉnh, mỗi cạnh có thể được dán nhãn hoặc không.

A diagram of a network

AI-generated content may be incorrect.

**2. Đồ thị vô hướng (Undirected Graph):**

Đồ thị G = (V, E) là một đồ thị vô hướng gồm tập đỉnh V và tập cạnh E. Mỗi cạnh e ∈ E được liên kết với một cặp đỉnh {i, j} ∈ V2 không có sự phân biệt giữa đỉnh đầu và đỉnh cuối của mỗi cạnh.

A diagram of a hexagon with circles and lines

AI-generated content may be incorrect.

**3. Đồ thị có hướng (Directed Graph hoặc Digraph):**

Đồ thị G = (V, U) là một đồ thị có hướng gồm tập đỉnh V và tập cạnh U. Mỗi cạnh u ∈ E được liên kết với một cặp đỉnh {i, j} ∈ V2 có sự phân biệt giữa các đỉnh đầu và đỉnh cuối của mỗi cạnh. Mỗi cạnh chỉ cho phép di chuyển từ đỉnh đầu đến đỉnh cuối của cạnh.

A diagram of a diagram

AI-generated content may be incorrect.

**4. Đồ thị có trọng số (Weighted Graph):**

- Đồ thị có trọng số (weighted graph) là một loại đồ thị trong đó mỗi cạnh được gán một giá trị số gọi là trọng số.

- Trọng số đại diện cho một thuộc tính, độ quan trọng hoặc khoảng cách giữa 2 đỉnh trong đồ thị. Trọng số của mỗi cạnh có thể được biểu diễn bằng một số thực dương hoặc số nguyên.

- Đồ thị có trọng số cho phép mô hình hoá và phân tích các vấn đề phức tạp hơn trong thực tế như tìm kiếm đường đi ngắn nhất dựa trên trọng số, tối ưu hoá đường đi dựa trên chi phí hoặc tìm kiếm cấu trúc mạng phù hợp dựa trên sự tương tự,…

A diagram of a diagram

AI-generated content may be incorrect.

**5. Đồ thị con (Subgraph):**

- Đồ thị con (subgraph) là một đồ thị nhỏ hơn được tạo thành bằng cách chọn một tập hợp con của các đỉnh và cạnh từ một đồ thị gốc. Đồ thị con giữ lại một phần của đồ thị gốc bằng cách loại bỏ một số đỉnh và/hoặc cạnh.

- Xét 2 đồ thị G = (X, U), G’ = (X’, U’). G’ là con của G. Kí hiệu: G’ ∈ G thoả mãn:

+ X’ ∈ X, U’ ∈ U

+ u = (i, j) ∈ U của G, nếu u ∈ U’ thì i, j ∈ X’

A diagram of a network

AI-generated content may be incorrect.

**6. Các dạng đồ thị (Types of Graph):**

- **Đồ thị rỗng (empty graph ~ null graph)**: là một đồ thị không chứa bất kì đỉnh nào. Nó không có đỉnh và không có cạnh. Trong đồ thị rỗng, không có mối quan hệ hoặc liên kết nào giữa các đỉnh vì không có đỉnh nào tồn tại để tạo ra các mối quan hệ đó.

- **Đơn đồ thị (simple graph)**: là đồ thị không có các cạnh song song (đa cạnh) giữa 2 đỉnh và không có cạnh nối một đỉnh với chính nó (khuyên).

- **Đa đồ thị (multigraph)**: là đồ thị có các cạnh song song và tuỳ theo định nghĩa mà nó có khuyên hay không.

A diagram of a hexagon with circles and lines

AI-generated content may be incorrect.

- **Đồ thị đầy đủ (complete graph)**: là đồ thị trong đó mỗi cặp đỉnh bất kì đều được nối bằng đúng một cạnh. Kí hiệu: Kn với n là số đỉnh của đồ thị, số cạnh = n(n – 1)/2.

A diagram of a network

AI-generated content may be incorrect.

**7. Đường đi (Path) và Chu trình (Cycle):**

- **Đường đi**: Cho đồ thị G = (X, U):

+ Một đường đi trong G là một dãy luân phiên các đỉnh và cạnh:

x1 u1, x2 u2, …, xm-1 um-1, xm (xi là đỉnh và ui là cạnh)

+ Trong đồ thị thoả mãn điều kiện ui liên kết với cặp đỉnh (xi, xi+1), nghĩa là:

ui = (xi, xi+1) ≠ (xi+1, xi) nếu đồ thị có hướng

ui = {xi, xi+1} nếu đồ thị vô hướng

Khi đó ta gọi x1 là đỉnh đầu và xm là đỉnh cuối của đường đi.

A diagram of a network

AI-generated content may be incorrect.

E B A C là một đường đi

- **Chu trình**: Một chu trình trong đồ thị là một đường đi có đỉnh đầu trùng đỉnh cuối có dạng: x1 u1, x2 u2, …, xm-1 um-1, xm um, x1 sao cho các cạnh u1, u2, …, um đôi một khác nhau.

A diagram of a network

AI-generated content may be incorrect.

C D F C là một chu trình

- **Tính chất đơn và sơ cấp**:

+ Tính chất đơn của chu trình hay đường đi: không có cạnh nào xuất hiện quá 1 lần trong chu trình hay đường đi đó.

+ Tính chất sơ cấp: không có đỉnh nào xuất hiện quá 1 lần (trừ đỉnh đầu và đỉnh cuối).

**8. Đỉnh (Vertex):**

- **Đỉnh kề (adjacency vertex)**: 2 đỉnh được coi là kề nhau nếu chúng được nối với nhau bởi 1 cạnh.

- **Khuyên (loop)**: là 1 cạnh nối từ 1 đỉnh về chính nó.

- **Đỉnh treo (hanging vertex)**: là 1 đỉnh trong đồ thị chỉ có 1 cạnh kết nối nó với các đỉnh khác.

- **Đỉnh cô lập (isolated vertex)**: là 1 đỉnh trong đồ thị không có cạnh kết nối nó với bất kì đỉnh khác.

- **Bậc của đỉnh (degree)**:

+ **Đồ thị vô hướng**: bậc của 1 đỉnh trong đồ thị vô hướng là số lượng cạnh kết nối với đỉnh đó. Kí hiệu bằng **deg(v)** hoặc **d(v)**.

∙ Đỉnh treo có bậc là 1.

∙ Đỉnh cô lập có bậc là 0.

+ **Đồ thị có hướng**: bậc của 1 đỉnh trong đồ thị có hướng là tổng số cạnh đi ra từ đỉnh đó và số cạnh đi vào đỉnh đó.

∙ Bậc ra (outdegree): số lượng cạnh đi ra từ 1 đỉnh, kí hiệu: **deg+(v)**.

∙ Bậc vào (indegree): số lượng cạnh đi vào 1 đỉnh, kí hiệu: **deg-(v)**.

Ta có bậc của 1 đỉnh v: **deg(v) = deg+(v) + deg-(v)**.

- **Mối liên hệ giữa bậc - số cạnh**: trong mọi đồ thị G = (V, E), tổng số bậc của các đỉnh của G bằng 2 lần số cạnh của nó:

A mathematical equation with numbers and symbols

AI-generated content may be incorrect.

A diagram of a network

AI-generated content may be incorrect.

H, C là đỉnh treo (deg(H) = deg(C) = 1)

I là đỉnh cô lập (deg(I) = 0)

**9. Liên thông (Connection):**

**a. Đồ thị vô hướng:**

- **Đồ thị liên thông**: Đồ thị vô hướng G = (V, E) được gọi là liên thông nếu luôn tìm được đường đi giữa 2 đỉnh bất kì của nó.

A diagram of a molecule

AI-generated content may be incorrect.

G là đồ thị liên thông, H không phải là đồ thị liên thông

- **Thành phần liên thông**: Trong trường hợp đồ thị là không liên thông, nó sẽ rã ra thành một số đồ thị con liên thông không có đỉnh chung. Những đồ thị con liên thông như vậy ta sẽ gọi là các thành phần liên thông của đồ thị. Ở trên, đồ thị H gồm 3 thành phần liên thông là H1, H2, H3.

- **Cầu**: Cạnh e được gọi là cầu nếu việc loại bỏ nó khỏi đồ thị làm tăng số thành phần liên thông của đồ thị. Ở trên, cạnh AB là cầu vì sau khi bỏ AB ta được 2 thành phần liên thông, cạnh BC không phải là cầu vì sau khi bỏ BC đồ thị vẫn liên thông.

A diagram of a network

AI-generated content may be incorrect.

**b. Đồ thị có hướng:**

- **Liên thông mạnh**: Đồ thị có hướng G = (V, A) được gọi là liên thông mạnh nếu luôn tìm được đường đi giữa 2 đỉnh bất kì của nó.

- **Liên thông yếu**: Đồ thị có hướng G = (V, A) được gọi là liên thông yếu nếu đồ thị vô hướng tương ứng với nó là vô hướng liên thông.

- Đồ thị liên thông mạnh thì cũng liên thông yếu.

A diagram of a diagram of a diagram

AI-generated content may be incorrect.

▪ **Các thao tác trên đồ thị:**

**1. Biểu diễn (Define) đồ thị:** Có 2 cách hay dùng nhất để biểu diễn đồ thị:

- Biểu diễn bằng **Ma trận kề (Adjacency Matrix)**

- Biểu diễn bằng **Danh sách kề (Adjacency List)**

**2. Duyệt (Traverse) đồ thị:** Có 2 kinh điển để duyệt đồ thị:

- **DFS (Depth-First Search)**: duyệt theo chiều sâu

+ Đi sâu vào nhánh hiện tại trước khi quay lại.

+ Dùng đệ quy hoặc stack để nhớ đường đi.

+ Dễ cài đặt và thích hợp để tìm thành phần liên thông, tìm chu trình, cây khung nhỏ nhất.

- **BFS (Breadth-First Search)**: duyệt theo chiều rộng

+ Duyệt theo từng mức, bắt đầu từ đỉnh xuất phát, sau đó mở rộng đến các đỉnh kề.

+ Dùng queue để lưu trữ các đỉnh chờ duyệt.

+ Tốt cho việc tìm đường đi ngắn nhất.

**3. Code Python minh hoạ:**

◦ **Sử dụng ma trận kề:**

from collections import deque

class GraphMatrix:

    def \_\_init\_\_(*self*, *size*): # khởi tạo ma trận kề với size x size phần tử 0

*self*.size = *size*

*self*.matrix = [[0] \* *size* for \_ in range(*size*)]

    def add\_edge(*self*, *u*, *v*): # thêm cạnh (vô hướng)

*self*.matrix[*u*][*v*] = 1

*self*.matrix[*v*][*u*] = 1

    def remove\_edge(*self*, *u*, *v*): # xóa cạnh (vô hướng)

*self*.matrix[*u*][*v*] = 0

*self*.matrix[*v*][*u*] = 0

    def is\_edge(*self*, *u*, *v*): # kiểm tra xem có cạnh không

        return *self*.matrix[*u*][*v*] == 1

    def num\_vertices(*self*): # đếm số đỉnh

        return *self*.size

    def num\_edges(*self*): # đếm số cạnh (chia 2 vì mỗi cạnh đếm 2 lần)

        return sum(sum(row) for row in *self*.matrix) // 2

    def \_\_str\_\_(*self*):

        return "\n".join(str(row) for row in *self*.matrix)

    def dfs(*self*, *start*, *visited*=None):

        if *visited* is None: *visited* = set()

*visited*.add(*start*)

        print(*start*, *end*=' ')

        for v in range(*self*.size):

            if *self*.matrix[*start*][v] == 1 and v not in *visited*:

*self*.dfs(v, *visited*)

    def bfs(*self*, *start*):

        visited = set([*start*])

        queue = deque([*start*])

        while queue:

            u = queue.popleft()

            print(u, *end*=' ')

            for v in range(*self*.size):

                if *self*.matrix[u][v] == 1 and v not in visited:

                    visited.add(v)

                    queue.append(v)

    def connected\_components(*self*): # tìm các thành phần liên thông

        visited = set()

        components = []

        for v in range(*self*.size):

            if v not in visited:

                comp = []

*self*.\_dfs\_collect(v, visited, comp)

                components.append(comp)

        return components

    def \_dfs\_collect(*self*, *v*, *visited*, *comp*):

*visited*.add(*v*)

*comp*.append(*v*)

        for u in range(*self*.size):

            if *self*.matrix[*v*][u] == 1 and u not in *visited*:

*self*.\_dfs\_collect(u, *visited*, *comp*)

◦ **Sử dụng danh sách kề:**

class GraphList:

    def \_\_init\_\_(*self*): # khởi tạo danh sách kề (dictionary)

*self*.adj\_list = {}

    def add\_edge(*self*, *u*, *v*): # thêm cạnh

*self*.adj\_list.setdefault(*u*, []).append(*v*)

*self*.adj\_list.setdefault(*v*, []).append(*u*)

    def remove\_edge(*self*, *u*, *v*): # xóa cạnh

        if *u* in *self*.adj\_list and *v* in *self*.adj\_list[*u*]:

*self*.adj\_list[*u*].remove(*v*)

        if *v* in *self*.adj\_list and *u* in *self*.adj\_list[*v*]:

*self*.adj\_list[*v*].remove(*u*)

    def is\_edge(*self*, *u*, *v*): # kiểm tra cạnh

        return *v* in *self*.adj\_list.get(*u*, [])

    def num\_vertices(*self*): # đếm số đỉnh

        return len(*self*.adj\_list)

    def num\_edges(*self*): # đếm số cạnh

        return sum(len(neigh) for neigh in *self*.adj\_list.values()) // 2

    def \_\_str\_\_(*self*):

        return "\n".join(f"{k}: {v}" for k, v in *self*.adj\_list.items())

    def dfs(*self*, *start*, *visited*=None):

        if *visited* is None: *visited* = set()

*visited*.add(*start*)

        print(*start*, *end*=' ')

        for neighbor in *self*.adj\_list.get(*start*, []):

            if neighbor not in *visited*:

*self*.dfs(neighbor, *visited*)

    def bfs(*self*, *start*):

        visited = set([*start*])

        queue = deque([*start*])

        while queue:

            u = queue.popleft()

            print(u, *end*=' ')

            for neighbor in *self*.adj\_list.get(u, []):

                if neighbor not in visited:

                    visited.add(neighbor)

                    queue.append(neighbor)

    def connected\_components(*self*): # tìm các thành phần liên thông

        visited = set()

        components = []

        for v in *self*.adj\_list.keys():

            if v not in visited:

                comp = []

*self*.\_dfs\_collect(v, visited, comp)

                components.append(comp)

        return components

    def \_dfs\_collect(*self*, *v*, *visited*, *comp*):

*visited*.add(*v*)

*comp*.append(*v*)

        for u in *self*.adj\_list.get(*v*, []):

            if u not in *visited*:

*self*.\_dfs\_collect(u, *visited*, *comp*)

◦ **Chuyển đổi giữa 2 dạng biểu diễn đồ thị:**

# Chuyển từ ma trận kề sang danh sách kề

def matrix\_to\_list(*matrix*):

    g\_list = GraphList()

    size = len(*matrix*)

    for u in range(size):

        for v in range(u, size):

            if *matrix*[u][v] == 1:

                g\_list.add\_edge(u, v)

    return g\_list

# Chuyển từ danh sách kề sang ma trận kề

def list\_to\_matrix(*adj\_list*):

    size = max(*adj\_list*.adj\_list.keys()) + 1

    g\_matrix = GraphMatrix(size)

    for u, neighbors in *adj\_list*.adj\_list.items():

        for v in neighbors:

            g\_matrix.add\_edge(u, v)

    return g\_matrix

▪ **Các bài toán liên quan đến đồ thị:**

❶ **Sắp xếp topo (Topological Sort):**

**Mục đích:**

- Xếp các đỉnh của đồ thị có hướng không chứa chu trình theo thứ tự mà nếu có cạnh A -> B thì A luôn đứng trước B.

- Ứng dụng: xác định thứ tự thực hiện công việc, tính toán phụ thuộc,…

**Ý tưởng (Kahn’s Algorithm):**

- Đếm số bậc vào (in-degree) của mỗi đỉnh.

- Bắt đầu từ các đỉnh có bậc vào bằng 0 (không phụ thuộc ai).

- Xoá cạnh của nó và cập nhật bậc vào của các đỉnh kề.

- Lặp lại cho đến khi không còn đỉnh nào.

**Code Python:**

from collections import deque

# Sắp xếp topo bằng Kahn's Algorithm (BFS)

def topo\_sort\_kahn(graph, n):

indegree = [0] \* n

for u in range(n):

for v in graph[u]:

indegree[v] += 1

queue = deque([u for u in range(n) if indegree[u] == 0])

result = []

while queue:

u = queue.popleft()

result.append(u)

for v in graph[u]:

indegree[v] -= 1

if indegree[v] == 0: queue.append(v)

if len(result) != n: print("Có chu trình! (Kahn)")

else: print("Thứ tự topo (Kahn):", result)

# Giải thuật khác: Sắp xếp topo bằng DFS

def topo\_sort\_dfs(graph, n):

visited = [False] \* n

result = []

def dfs(u):

visited[u] = True

for v in graph[u]:

if not visited[v]: dfs(v)

result.append(u)

for u in range(n):

if not visited[u]: dfs(u)

result.reverse()

print("Thứ tự topo (DFS):", result)

❷  **Cây khung tối tiểu (Minimum Spanning Tree - MST):**

**Mục đích:**

- Cây khung là tập hợp các cạnh của một đồ thị liên thông vô hướng sao cho nó kết nối tất cả các đỉnh của đồ thị mà không có chu trình. Cây khung tối tiểu là cây khung có tổng trọng số các cạnh là nhỏ nhất.

- Ứng dụng: kết nối mạng, thiết kế mạch điện, xây dựng hệ thống đường giao thông,…

**Ý tưởng (Prim’s Algorithm):**

- Bắt đầu từ bất kì đỉnh nào, đưa đỉnh vào cây MST.

- Chọn cạnh có trọng số nhỏ nhất nối giữa đỉnh trong cây và đỉnh chưa trong cây.

- Thêm cạnh đó và đỉnh mới vào cây.

- Lặp lại cho đến khi tất cả đỉnh đã được thăm.

**Code Python:**

import heapq

def prim(graph, n):

visited = [False] \* n

min\_heap = [(0, 0)] # (trọng số, đỉnh)

total\_weight = 0

mst\_edges = []

while min\_heap:

weight, u = heapq.heappop(min\_heap)

if visited[u]: continue

visited[u] = True

total\_weight += weight

mst\_edges.append((u, weight))

for v, w in graph.get(u, []):

if not visited[v]: heapq.heappush(min\_heap, (w, v))

print("Cây khung tối tiểu có tổng trọng số:", total\_weight)

print("Cạnh trong MST:", mst\_edges)

❸ **Tìm đường đi ngắn nhất (Find Shortest Path):**

**Mục đích:**

- Tìm đường đi có tổng trọng số nhỏ nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trong đồ thị có hướng.

- Ứng dụng: tìm đường đi ngắn nhất trên bản đồ, tìm đường nhanh nhất trong hệ thống giao thông, tối ưu hoá chi phí trong mạng máy tính,…

**Ý tưởng (Dijkstra’s Algorithm):**

- Khởi tạo khoảng cách từ nguồn đến chính nó là 0, đến các đỉnh khác là vô hạn (∞).

- Chọn đỉnh chưa thăm có khoảng cách nhỏ nhất hiện tại để duyệt.

- Cập nhật khoảng cách của các đỉnh kề, nếu đi qua đỉnh này ngắn hơn thì cập nhật khoảng cách mới.

- Lặp lại cho đến khi tất cả các đỉnh đã thăm xong.

**Code Python:**

import heapq

def dijkstra(graph, n, start):

dist = [float('inf')] \* n

dist[start] = 0

visited = [False] \* n

heap = [(0, start)] # (khoảng cách, đỉnh)

while heap:

d, u = heapq.heappop(heap)

if visited[u]: continue

visited[u] = True

for v, w in graph.get(u, []):

if not visited[v] and dist[v] > d + w:

dist[v] = d + w

heapq.heappush(heap, (dist[v], v))

print(f"Khoảng cách ngắn nhất từ đỉnh {start}:")

for i in range(n):

print(f"Đỉnh {i}: {dist[i]}")

return dist