Handout Buổi 13: Cây — Phần 2: Duyệt cây nhị phân

Mục tiêu bài học

- Phân loại được các dạng cây nhị phân.
- Vận dụng các thuật toán để duyệt cây nhị phân.
- Ứng dụng các phương pháp duyệt vào các bài toán lập trình như tìm kiếm, biểu diễn biểu thức, đánh giá cấu trúc dữ liệu,…

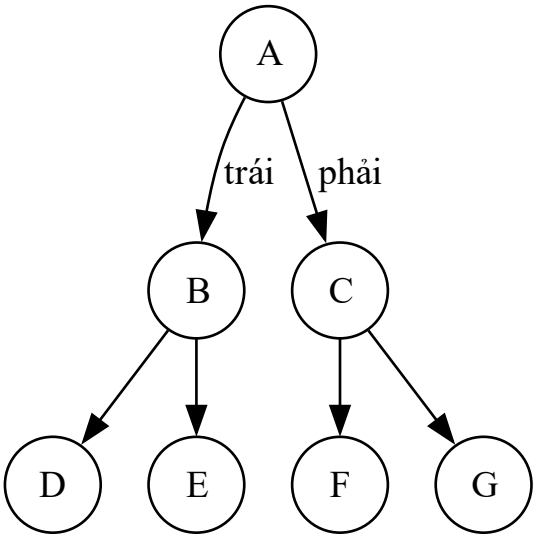
PHẦN 1: LÝ THUYẾT

6.3 Duyệt cây nhị phân

6.3.1 Định nghĩa

1. Khái niệm cây nhị phân

Một **cây nhị phân (binary tree)** là một cấu trúc dữ liệu phân cấp (hierarchical data structure), trong đó mỗi **nút (node)** có nhiều nhất **hai con**, gọi là **nút con trái (left child)** và **nút con phải (right child)**.
Biểu diễn tổng quát cây nhị phân như sau:

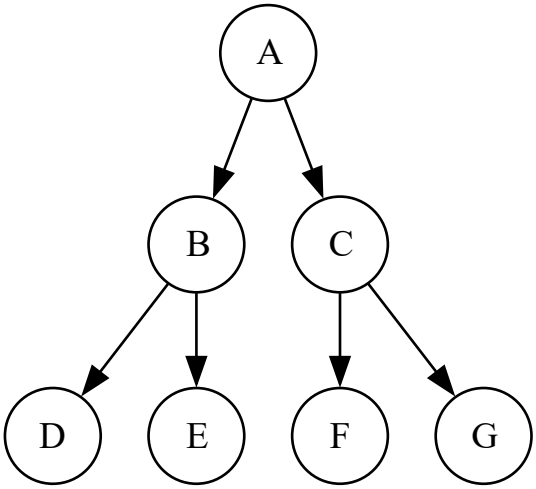


2. Phân loại các loại cây nhị phân

Trong thực tế và lý thuyết, có nhiều loại cây nhị phân khác nhau, mỗi loại có tính chất và ứng dụng riêng:

2.1. Cây nhị phân đầy đủ (Full Binary Tree)

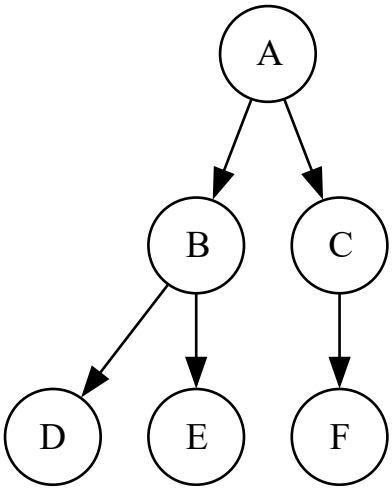
Là cây trong đó **mỗi nút hoặc có 0 hoặc có đúng 2 con**.
Ví dụ:



2.2. Cây nhị phân hoàn chỉnh (Complete Binary Tree)

Là cây trong đó **mọi mức (trừ mức cuối) đều đầy đủ**, và các nút ở mức cuối được **xếp từ trái sang phải**.

Ví dụ:

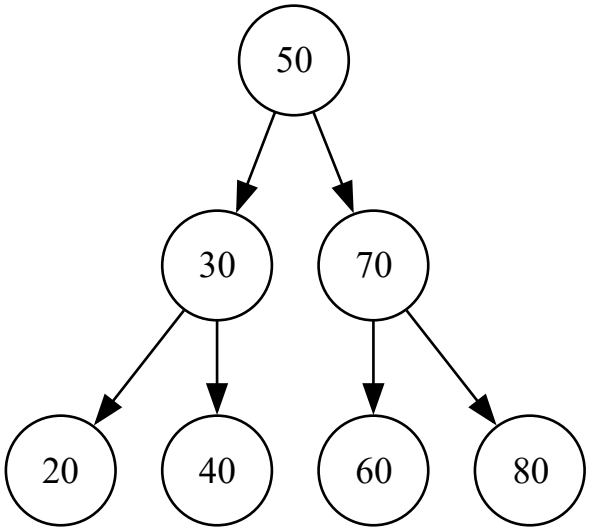


2.3. Cây nhị phân tìm kiếm (Binary Search Tree - BST)

Là cây nhị phân mà **với mỗi nút x**, tất cả các giá trị trong cây con trái của x **nhỏ hơn x**, và tất cả các giá trị trong cây con phải **lớn hơn x**.

Ứng dụng: tìm kiếm, chèn, xóa nhanh với độ phức tạp trung bình O(log n) nếu cân bằng.

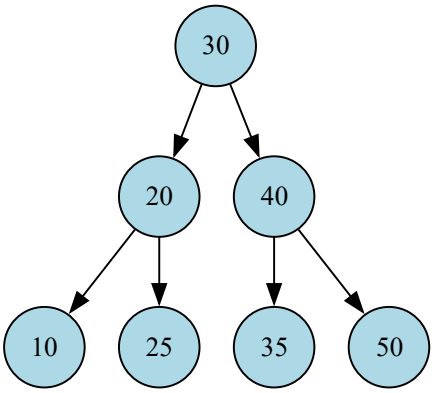
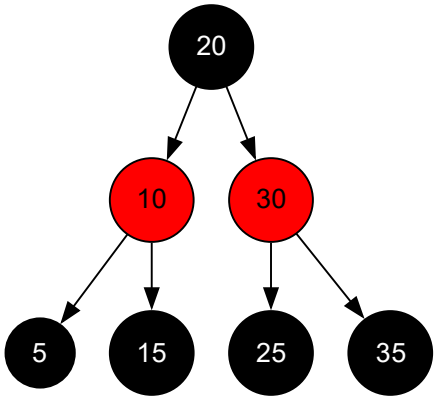
Ví dụ:



2.4. Cây nhị phân cân bằng (Balanced BST)

Là **cây nhị phân tìm kiếm mà độ cao (height) của hai cây con bất kỳ tại mỗi nút không chênh lệch quá 1**.

Ví dụ: AVL Tree, Red-Black Tree (được dùng trong STL của C++, TreeMap trong Java, v.v.)

Cây AVL (AVL Tree)	**Cây Đỏ-Đen (Red-Black Tree)**
 • Đây là cây AVL đã được cân bằng sau khi chèn các nút: 10, 20, 25, 30, 35, 40, 50. • Tại mọi nút, độ cao hai cây con chênh lệch không quá 1.	 • Cây tuân thủ quy tắc đỏ-đen: gốc đen, nút đỏ không có con đỏ, và mọi đường đi từ một nút đến lá có cùng số lượng nút đen.

Ý nghĩa thực tiễn: Giữ cho các phép toán trên cây có độ phức tạp ổn định trong các ứng dụng quan trọng như cơ sở dữ liệu, compiler, và các cấu trúc lưu trữ từ điển.

---

3. Các thuật ngữ cơ bản trong cây nhị phân

Thuật ngữ	Mô tả
Gốc (Root)	Nút bắt đầu của cây
Nút lá (Leaf)	Nút không có con
Nút trong (Internal Node)	Nút có ít nhất một con
Chiều cao (Height)	Số cạnh dài nhất từ gốc đến nút lá
Độ sâu (Depth)	Số cạnh từ gốc đến nút hiện tại
Mức (Level)	Tập hợp các nút có cùng độ sâu

4. Tính đệ quy của cây nhị phân

Tự nhiên, cây nhị phân có cấu trúc **đệ quy**: cây con trái và cây con phải của mỗi nút **cũng là cây nhị phân**. Điều này làm cho cây nhị phân rất thích hợp để cài đặt bằng **đệ quy** – một kỹ thuật lập trình mạnh trong Toán rời rạc và CNTT.

6.3.2. Các thuật toán duyệt cây nhị phân

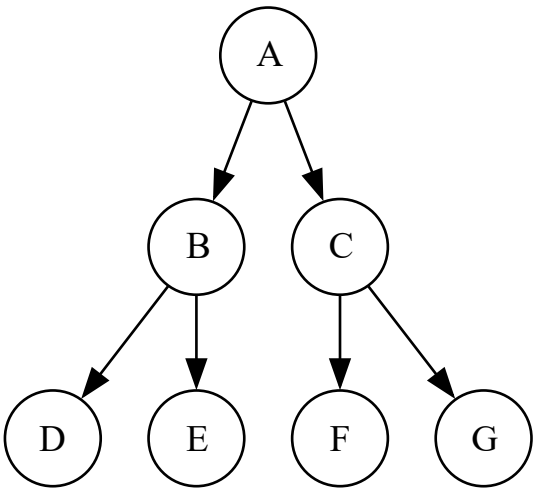
Trong cây nhị phân, **duyệt cây** là quá trình thăm các nút của cây theo một trình tự cụ thể. Mỗi phương pháp duyệt phù hợp với các ứng dụng khác nhau trong tin học như: xử lý biểu thức, cấu trúc dữ liệu, xây dựng trình biên dịch, v.v.

Các thuật toán duyệt cây nhị phân được chia làm hai nhóm chính:

- **Duyệt theo chiều sâu (Depth-First Traversal)**: Inorder, Preorder, Postorder
- **Duyệt theo chiều rộng (Breadth-First Traversal)**: Sẽ được trình bày ở phần sau.

A. Duyệt theo thứ tự (Depth-First Traversal)

Giả sử chúng ta có cây nhị phân sau:



1. Duyệt Inorder (LNR – Left, Node, Right)

Nguyên lý hoạt động:

- Duyệt cây con trái
- Thăm nút gốc
- Duyệt cây con phải

Mã giả (Pseudocode):

```
def inorder(node):
    if node is not None:
        inorder(node.left)
        print(node.value)
        inorder(node.right)
```

Kết quả khi áp dụng lên cây trên:

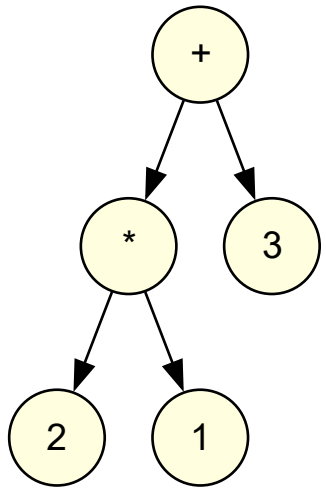
D, B, E, A, F, C, G

Ứng dụng trong CNTT:

- Duyệt cây nhị phân tìm kiếm (BST) để **in ra các phần tử theo thứ tự tăng dần**.
- Phân tích cấu trúc biểu thức toán học trung tố (infix expression).

Ví dụ thực tế:

```
# Biểu thức toán học biểu diễn bằng cây:
```




```
# Inorder sẽ in: 2 * 1 + 3
```

2. Duyệt Preorder (NLR – Node, Left, Right)

Nguyên lý hoạt động:

- Thăm nút gốc
- Duyệt cây con trái
- Duyệt cây con phải

Mã giả (Pseudocode):

```
def preorder(node):
```

```
    if node is not None:
        print(node.value)
        preorder(node.left)
        preorder(node.right)
```

Kết quả khi áp dụng lên cây trên:

A, B, D, E, C, F, G

Ứng dụng trong CNTT:

- Duyệt cây để **sao chép cấu trúc dữ liệu**.
- Biểu diễn biểu thức toán học ở dạng **tiền tố (prefix)**: dùng trong trình biên dịch để phân tích cú pháp biểu thức.

Ví dụ thực tế:

```
# Biểu thức tiền tố: + * 2 1 3
# Tức là: (2 * 1) + 3
```

3. Duyệt Postorder (LRN – Left, Right, Node)

Nguyên lý hoạt động:

- Duyệt cây con trái
- Duyệt cây con phải
- Thăm nút gốc

Mã giả (Pseudocode):

```
def postorder(node):
    if node is not None:
        postorder(node.left)
        postorder(node.right)
        print(node.value)
```

Kết quả khi áp dụng lên cây trên:

D, E, B, F, G, C, A

Ứng dụng trong CNTT:

- Giải biểu thức số học khi biểu diễn cây biểu thức ở **hậu tố (postfix)** (RPN - Reverse Polish Notation).
- Giải phóng bộ nhớ, xóa toàn bộ cây (cần duyệt con trước, gốc sau).

Ví dụ thực tế:

```
# Biểu thức hậu tố: 2 1 * 3 +
# Máy tính sẽ dễ dàng tính toán theo stack: push 2, push 1, pop * => push 2*1, push 3, pop + => 2*1 + 3
```

So sánh các phương pháp duyệt

Phương pháp	Thứ tự thăm	Ứng dụng chính
Inorder	Trái → Gốc → Phải	In thứ tự tăng dần trong BST

Phương pháp	Thứ tự thăm	Ứng dụng chính
Preorder	Gốc → Trái → Phải	Biểu diễn cây, prefix notation
Postorder	Trái → Phải → Gốc	Giải biểu thức hậu tố, hủy cây

Độ phức tạp của các thuật toán duyệt

- **Thời gian**: Tất cả các thuật toán đều có độ phức tạp **O(n)**, với n là số lượng nút trong cây.
- **Không gian**:
  - Đệ quy: phụ thuộc vào chiều cao của cây (stack gọi hàm): O(h)
  - Phi đệ quy: cần sử dụng stack tường minh, độ phức tạp vẫn là O(n) trong trường hợp xấu nhất.

Kết luận phần A:
Các phương pháp duyệt cây nhị phân không chỉ là lý thuyết đơn thuần, mà là **nền tảng cho hàng loạt ứng dụng lập trình và thuật toán hiện đại**, như:

- Phân tích biểu thức trong trình biên dịch
- Hệ quản trị cơ sở dữ liệu (cây B+, cây AVL,…)
- Lập trình giải thuật (backtracking, game tree,…)
- Thiết kế hệ thống (cây quyết định, hệ chuyên gia,…)

B. Duyệt theo mức (Level-order Traversal)

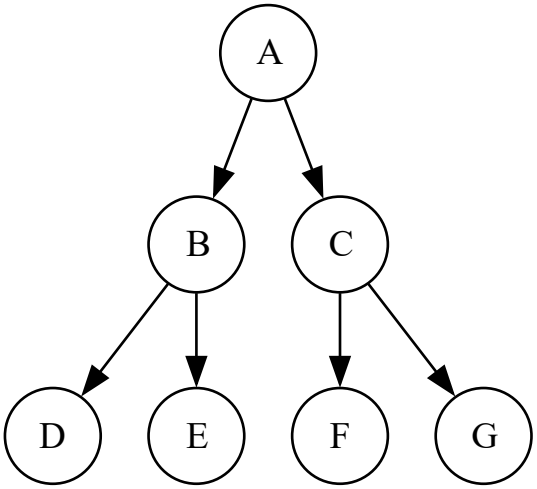
Trong phần này, chúng ta sẽ tìm hiểu phương pháp duyệt cây nhị phân theo **mức**, còn gọi là **duyệt theo chiều rộng** (Breadth-First Traversal). Đây là một kỹ thuật thiết yếu, có nhiều ứng dụng thực tiễn trong lập trình và giải thuật đồ thị.

1. Khái niệm duyệt theo mức

Duyệt theo mức là quá trình thăm các nút trong cây nhị phân **từ trên xuống dưới**, và **từ trái sang phải** ở mỗi mức.
Duyệt theo mức thực hiện tương tự như thuật toán **Breadth-First Search (BFS)** trong đồ thị — ta sử dụng một **hàng đợi (queue)** để lưu trữ các nút chờ duyệt.

2. Nguyên lý hoạt động

Giả sử ta có cây nhị phân sau:



Trình tự duyệt theo mức của cây trên là:
A, B, C, D, E, F, G

3. Mã giả (Pseudocode)

```
from collections import deque

def level_order(root):
    if root is None:
        return
    queue = deque()
    queue.append(root)
    while queue:
        node = queue.popleft()
        print(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

4. Giải thích thuật toán

- Khởi tạo một hàng đợi rỗng.
- Thêm gốc cây vào hàng đợi.
- Lặp cho đến khi hàng đợi rỗng:
    - Lấy phần tử đầu hàng ra (nút hiện tại).
    - Xử lý nút hiện tại (in ra, lưu trữ, v.v.).
    - Nếu có con trái, đưa vào hàng đợi.
    - Nếu có con phải, đưa vào hàng đợi.

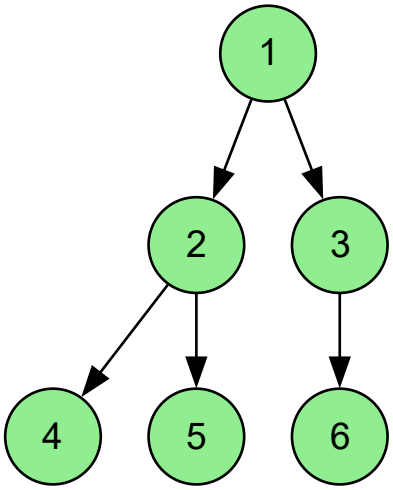
Thuật toán này đảm bảo **các nút được duyệt đúng thứ tự mức**, với độ phức tạp thời gian tuyến tính O(n) và bộ nhớ phụ thuộc vào chiều rộng lớn nhất của cây.

5. Ứng dụng trong CNTT

5.1. Dựng cây từ mảng

- Khi biểu diễn cây nhị phân dưới dạng mảng (như trong cây heap, cây hoàn chỉnh), duyệt theo mức cho phép dựng lại cây:

```
arr = [1, 2, 3, 4, 5, 6]
# Cây hoàn chỉnh tương ứng:
```



- Dễ dàng xây dựng cây theo cách: cha ở chỉ số i → con trái ở 2i+1, con phải ở 2i+2.

5.2. Tìm kiếm theo mức (BFS trong đồ thị)

- Duyệt theo mức chính là **BFS trên cây** — một dạng đồ thị không có chu trình và liên thông.
- Được dùng trong:
  - Tìm đường đi ngắn nhất trên đồ thị không trọng số.
  - Phân tích cấu trúc mạng (network routing).
  - Hệ thống trò chơi AI (game trees).
  - Truy vấn cây lớn trong cơ sở dữ liệu.

5.3. Cấu trúc dữ liệu ưu tiên theo mức

- Dùng trong thuật toán A*, Dijkstra khi muốn mở rộng theo thứ tự ưu tiên.
- Trong serialization/deserialization của cây nhị phân (ví dụ: LeetCode, các bài toán lưu trữ cây).

6. Độ phức tạp

Yếu tố	Giá trị
Thời gian	**O(n)** với n là số nút
Không gian	**O(w)** với w là chiều rộng lớn nhất của cây (số nút tối đa ở một mức)

7. Mở rộng: Duyệt theo mức nâng cao

- **Duyệt theo mức có phân nhóm**:
  - Trả về danh sách các mức (list of levels) → dùng trong cây phân loại, cây quyết định.

```
# Output: [[A], [B, C], [D, E, F, G]]
```

- **Duyệt theo mức có xen kẽ trái/phải (zigzag level order)**:
  - Ứng dụng trong xử lý cây cân bằng, tối ưu hóa các phép toán cây trong AI.

8. Kết luận phần B

Duyệt theo mức là kỹ thuật thiết yếu không chỉ trong xử lý cây nhị phân, mà còn là **cơ sở cho các thuật toán đồ thị tổng quát**. Kỹ thuật này đặc biệt quan trọng trong:

- **Xây dựng cấu trúc cây từ dữ liệu tuyến tính**
- **Tìm kiếm theo chiều rộng trong hệ thống phân cấp**
- **Tối ưu hoá luồng xử lý song song (task scheduling)**

🔎 **So sánh nhanh giữa duyệt theo chiều sâu và duyệt theo mức**

Tiêu chí	DFS (Inorder, Pre, Post)	BFS (Level-order)
Cấu trúc dùng	Stack (hoặc đệ quy)	Queue
Thứ tự duyệt	Trái/Phải theo gốc	Theo từng mức
Ứng dụng chính	Phân tích biểu thức, xây cây	Tìm kiếm, xây dựng cây
Bộ nhớ	O(h)	O(w)

C. Triển khai bằng đệ quy và không đệ quy

Trong lập trình và thuật toán, việc triển khai duyệt cây có thể được thực hiện bằng **đệ quy** hoặc **không đệ quy** (thường sử dụng **stack tường minh** để mô phỏng lại cơ chế ngăn xếp của lời gọi hàm đệ quy).
Hiểu rõ sự khác biệt giữa hai cách triển khai này là **cần thiết để sinh viên CNTT xử lý các bài toán có kích thước lớn**, yêu cầu tối ưu bộ nhớ, hoặc cần kiểm soát luồng thực thi cụ thể.

1. Triển khai bằng đệ quy

Nguyên lý:

Đệ quy tận dụng **ngăn xếp lời gọi hàm (call stack)** để thực hiện việc duyệt cây. Do cấu trúc đệ quy tự nhiên của cây nhị phân, phương pháp này thường dễ viết và ngắn gọn.

Ví dụ: Duyệt Inorder (LNR)

```
def inorder(node):
    if node is not None:
        inorder(node.left)
        print(node.value)
        inorder(node.right)
```

Ưu điểm:

- Mã ngắn gọn, dễ hiểu
- Trực quan khi xử lý cây

Nhược điểm:

- **Tiêu tốn bộ nhớ stack**, dễ gây lỗi **tràn ngăn xếp (stack overflow)** với cây có chiều cao lớn (deep trees)
- **Khó kiểm soát luồng thực thi** trong các ứng dụng yêu cầu tính linh hoạt cao (như tạm dừng giữa chừng, backtracking phức tạp,...)

2. Triển khai không đệ quy (dùng Stack)

Nguyên lý:

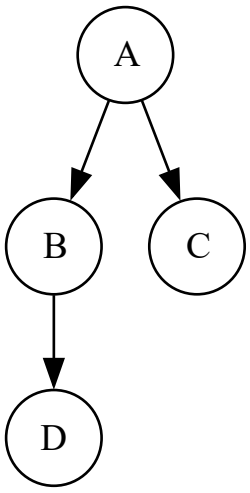
Sử dụng một **ngăn xếp tường minh (explicit stack)** để mô phỏng lại quá trình duyệt giống như call stack. Điều này cho phép lập trình viên **chủ động điều khiển việc thăm và xử lý nút**, thích hợp trong các trường hợp cần hiệu năng cao hoặc xử lý dữ liệu lớn.

Ví dụ: Inorder không đệ quy

```
def inorder_iterative(root):
    stack = []
    current = root
    while current is not None or stack:
        while current:
            stack.append(current)
            current = current.left
        current = stack.pop()
        print(current.value)
        current = current.right
```

Mô phỏng quá trình:

Giả sử ta có cây:



- Stack ban đầu: []
- Duyệt trái đến hết: stack = [A, B, D]
- Bắt đầu pop và xử lý theo thứ tự: D → B → A → C

Ưu điểm:

- **Không phụ thuộc vào call stack của hệ thống**, tránh lỗi stack overflow
- Có thể **tạm dừng, lưu trạng thái**, dễ dùng trong các ứng dụng tương tác, đa luồng

Nhược điểm:

- Mã dài hơn, **khó đọc và bảo trì hơn đệ quy**
- Dễ sai sót nếu không hiểu rõ luồng thực thi

3. So sánh tổng quát

Tiêu chí	**Đệ quy**	**Không đệ quy (Stack)**
Cấu trúc mã	Ngắn gọn, dễ hiểu	Dài hơn, tường minh hơn
Bộ nhớ sử dụng	Dùng call stack hệ thống	Dùng stack tự quản lý
Khả năng kiểm soát	Hạn chế (khó tạm dừng, tiếp tục)	Linh hoạt hơn
Rủi ro	Stack overflow nếu cây sâu	Khó bảo trì nếu xử lý nhiều logic phức
Ứng dụng	Thích hợp cho bài toán nhỏ, trung bình	Cần thiết cho xử lý cây lớn, hiệu năng cao

4. Mở rộng: Áp dụng stack cho các loại duyệt khác

Preorder không đệ quy

```
def preorder_iterative(root):
    if root is None:
```

```
        return
    stack = [root]
    while stack:
        node = stack.pop()
        print(node.value)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```

Postorder không đệ quy

```
def postorder_iterative(root):
    if root is None:
        return
    stack1 = [root]
    stack2 = []
    while stack1:
        node = stack1.pop()
        stack2.append(node)
        if node.left:
            stack1.append(node.left)
        if node.right:
            stack1.append(node.right)
    while stack2:
        print(stack2.pop().value)
```

5. Ứng dụng thực tiễn

Ứng dụng	**Phương pháp triển khai đề xuất**
Biểu thức toán học / compiler	Đệ quy
Cây lớn, dữ liệu từ file/bộ nhớ ngoài	Không đệ quy
Tìm kiếm trạng thái trong game AI	Stack thủ công để dễ kiểm soát
Duyệt theo lệnh tương tác (interpreter)	Không đệ quy (dễ dừng/tiếp tục)

6. Kết luận phần C

- Việc lựa chọn giữa **đệ quy** và **không đệ quy** phụ thuộc vào **yêu cầu của bài toán và đặc điểm hệ thống**.
- Trong các **bài toán kỹ thuật và thực tiễn lớn**, lập trình viên cần **làm chủ cả hai cách triển khai** để ứng dụng linh hoạt, tối ưu hóa hiệu năng và bộ nhớ.

- Việc hiểu sâu cấu trúc stack trong đệ quy và cách mô phỏng lại là **một bước quan trọng trong quá trình trưởng thành tư duy thuật toán của sinh viên ngành CNTT.**

D. Phân tích độ phức tạp của các thuật toán duyệt cây nhị phân

1. Đặt vấn đề

Duyệt cây là một **bài toán cơ bản và phổ biến** trong các ứng dụng xử lý dữ liệu phân cấp (cây thư mục, cây biểu thức, cây tìm kiếm,...). Hiệu quả của việc duyệt cây được đo bằng độ phức tạp **thời gian** và **không gian** — yếu tố then chốt trong mọi thuật toán.

Chúng ta sẽ phân tích độ phức tạp theo hai tiêu chí:

- **Độ phức tạp thời gian (Time Complexity)**
- **Độ phức tạp không gian (Space Complexity)**

2. Độ phức tạp thời gian

✅ **Mệnh đề chính:**

Tất cả các thuật toán duyệt cây nhị phân (Inorder, Preorder, Postorder, Level-order) đều có độ phức tạp thời gian là: O(n)

Trong đó:

- n: số lượng nút trong cây

📌 **Giải thích:**

- Mỗi nút được **truy cập đúng một lần** và có thể được **đưa vào/ra khỏi stack hoặc queue đúng một lần**.
- Không có thao tác lặp thừa hoặc truy cập lại nút.
- Ngay cả trong các cây mất cân bằng (cây lệch trái/phải), tổng số thao tác duyệt vẫn tỷ lệ tuyến tính với số nút.

🧠 **Tư duy thuật toán:**

Dù trình tự thăm nút khác nhau giữa các phương pháp (Inorder, Preorder,...), **số lượng bước không đổi** vì mục tiêu chung là thăm toàn bộ cây — điều này tạo nên tính **ổn định của độ phức tạp**.

3. Độ phức tạp không gian

✅ **Phụ thuộc vào cấu trúc cây:**

- Với **duyệt đệ quy**: không gian phụ thuộc vào chiều cao cây hhh.
- Với **duyệt không đệ quy**: không gian phụ thuộc vào số phần tử có thể có trong stack hoặc queue tại một thời điểm.

📊 **Tổng hợp so sánh không gian bộ nhớ:**

Thuật toán	Cấu trúc hỗ trợ	Độ phức tạp không gian
Inorder (đệ quy)	Call stack	O(h)
Inorder (stack)	Stack tường minh	O(h)
Preorder	Stack	O(h)
Postorder	Stack1 + Stack2	O(h) (gấp đôi)
Level-order	Queue	O(w), với w là chiều rộng lớn nhất

📌 **Chú thích:**

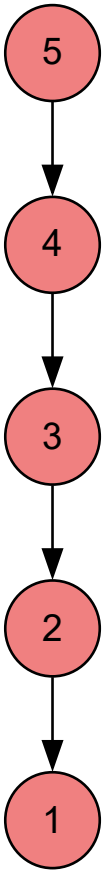
- h: chiều cao của cây ($\log_2 n$ với cây cân bằng, gần n với cây lệch)
- w: số nút lớn nhất trong một mức (đặc biệt với cây hoàn chỉnh, $w \approx n/2$)

4. Các tình huống biên (Edge Cases)

Trường hợp đặc biệt	Phân tích
Cây trống	Thời gian và bộ nhớ là O(1)
Cây một nhánh (cây lệch)	Thời gian vẫn là O(n), nhưng call stack hoặc stack có thể lên đến O(n)
Cây hoàn chỉnh/cân bằng	Hiệu quả nhất: $h=\log_2 n$, $w=n/2$

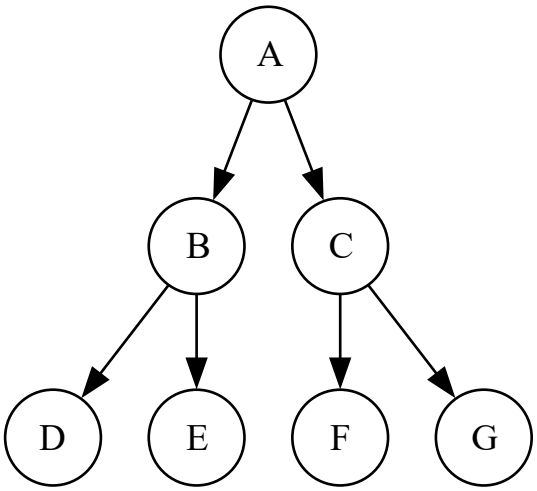
5. Ví dụ minh họa

⏱ **Trường hợp cây lệch trái:**



- Gọi đệ quy đến độ sâu = n → stack depth = O(n)
- Bài toán này đòi hỏi chuyển sang stack tường minh để tránh lỗi stack overflow

⏱ **Trường hợp cây đầy đủ:**



- Chiều cao h = $\log_2 7 \approx 3$
- Tối ưu về cả thời gian và bộ nhớ stack

6. Tổng kết

Yếu tố	Giá trị chung cho tất cả phương pháp
Thời gian	O(n) – vì mỗi nút thăm đúng 1 lần
Không gian (stack)	O(h) – nếu dùng đệ quy hoặc stack
Không gian (queue)	O(w) – với duyệt theo mức
Ảnh hưởng của cấu trúc cây	Cây càng lệch → bộ nhớ càng tăng

7. Gợi ý chiến lược cho sinh viên IT

- Trong các hệ thống lớn, hạn chế đệ quy nếu không kiểm soát được chiều cao cây → dùng **phi đệ quy để an toàn bộ nhớ**.
- Khi xử lý cây từ dữ liệu tuyến tính (array, JSON,...), dùng duyệt **Level-order** để tái cấu trúc cây hiệu quả.
- Khi cần **tìm kiếm nhanh**, chọn cây cân bằng và dùng **Inorder** để xử lý tuần tự.

III. Ứng dụng của các thuật toán duyệt cây nhị phân

1. Tìm kiếm, chèn, xóa trên cây nhị phân tìm kiếm (BST)

1.1. Ôn tập khái niệm BST

Cây nhị phân tìm kiếm (Binary Search Tree – BST) là một cấu trúc cây nhị phân đặc biệt, trong đó **mỗi nút đều thỏa mãn điều kiện:**

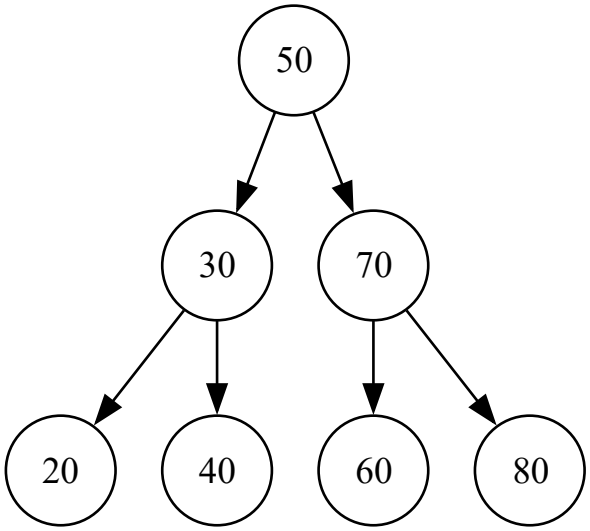
Giá trị bên trái < Giá trị gốc < Giá trị bên phải

Điều này cho phép ta khai thác tính chất **sắp xếp** của BST để thực hiện nhanh các thao tác tìm kiếm, chèn, xóa — tất cả với độ phức tạp **trung bình O(log n)** nếu cây cân bằng.

1.2. Ứng dụng Inorder Traversal trong BST

Trong BST, duyệt **Inorder (LNR: Left → Node → Right)** sẽ **trả về các phần tử theo thứ tự tăng dần**.

📌 **Ví dụ minh họa:**



➞ **Duyệt Inorder** sẽ trả về: 20 30 40 50 60 70 80

➞ Đây là thứ tự **sắp xếp tăng dần** — cực kỳ hữu ích cho xuất dữ liệu hoặc kiểm tra BST hợp lệ.

1.3. Tìm kiếm trong BST

✅ **Thuật toán:**

```
def search_bst(root, key):
    if root is None or root.value == key:
        return root
    if key < root.value:
        return search_bst(root.left, key)
    else:
        return search_bst(root.right, key)
```

📌 **Phân tích:**

- **Trung bình**: O(logn) nếu cây cân bằng
- **Tệ nhất**: O(n) nếu cây lệch (giống danh sách liên kết)

1.4. Chèn phần tử vào BST

✅ **Thuật toán:**

```
def insert_bst(root, key):
    if root is None:
        return Node(key)
    if key < root.value:
        root.left = insert_bst(root.left, key)
    else:
        root.right = insert_bst(root.right, key)
```

```
    return root
```

- Duyệt từ gốc theo quy tắc trái/ phải cho đến khi gặp vị trí phù hợp
- **BST không cho phép phần tử trùng (theo định nghĩa chuẩn)** — cần xử lý riêng nếu muốn hỗ trợ

1.5. Xóa phần tử khỏi BST

✅ **Có 3 trường hợp:**

1. **Nút là lá** → xóa trực tiếp
2. **Nút có một con** → nối con vào cha
3. **Nút có hai con** → tìm **nút nhỏ nhất bên phải (inorder successor)** hoặc lớn nhất bên trái (inorder predecessor), thay thế giá trị rồi xóa đệ quy

✅ **Thuật toán mẫu:**

```
def delete_bst(root, key):
    if root is None:
        return None
    if key < root.value:
        root.left = delete_bst(root.left, key)
    elif key > root.value:
        root.right = delete_bst(root.right, key)
    else:
        # Nút có một hoặc không có con
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        # Nút có hai con: tìm inorder successor
        temp = find_min(root.right)
        root.value = temp.value
        root.right = delete_bst(root.right, temp.value)
    return root

def find_min(node):
    while node.left:
        node = node.left
    return node
```

1.6. Tại sao Inorder lại quan trọng trong BST?

Ứng dụng	**Vai trò của Inorder**
Kiểm tra cây có là BST hợp lệ	Kiểm tra xem kết quả Inorder có tăng dần
Xuất dữ liệu đã sắp xếp	Inorder trả về danh sách đã sắp
Cắt cây (split), ghép cây (merge)	Duyệt Inorder giúp phân tích cấu trúc
Tạo danh sách từ BST	Inorder xuất danh sách dạng mảng dễ dùng

1.7. Bài toán thực tiễn sử dụng BST + Inorder

Bài toán	**Cách áp dụng**
Tự động hoàn thành từ khóa (autocomplete)	Sắp xếp từ điển, tìm kiếm nhanh với BST
Biên dịch ngôn ngữ (compiler syntax tree)	Biểu diễn biểu thức → Inorder = biểu thức trung tố
Cấu trúc dữ liệu cho bộ tìm kiếm (search engine)	Lưu trữ và tìm kiếm từ khóa nhanh chóng
Kiểm tra cây đã sắp xếp hay chưa	Inorder phải là mảng tăng

1.8. Độ phức tạp các thao tác trên BST

Thao tác	**Cây cân bằng**	**Cây lệch (xấu nhất)**
Tìm kiếm	O(logn)	O(n)
Chèn	O(logn)	O(n)
Xóa	O(logn)	O(n)
Inorder traversal	O(n)	O(n)

✅ **Kết luận phần 1:**

- Inorder traversal không chỉ là một phương pháp duyệt, mà còn là **xương sống logic** của cây BST.
- Việc hiểu rõ và sử dụng Inorder giúp:
  - **Duy trì tính đúng đắn của BST**
  - **Xuất và kiểm tra dữ liệu một cách đáng tin cậy**
  - **Tạo cầu nối giữa cấu trúc dữ liệu và các bài toán thực tiễn như tìm kiếm, biên dịch, AI,...**

2. Biểu diễn và tính giá trị biểu thức số học

2.1. Cây biểu thức là gì?

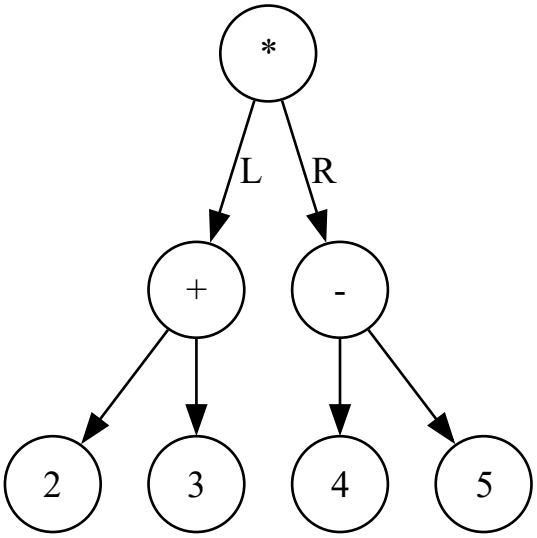
Cây biểu thức (expression tree) là một loại **cây nhị phân đặc biệt**, trong đó:

- **Lá (leaf nodes)** chứa **toán hạng (operand)**: số hoặc biến.
- **Nút trong (internal nodes)** chứa **toán tử (operator)**: +, -, *, /, ^,…

Cây biểu thức được xây dựng để **biểu diễn cấu trúc toán học của biểu thức số học** theo cú pháp ưu tiên đúng.

2.2. Ví dụ cây biểu thức

Biểu thức trung tố (infix):
(2+3)∗(4−5)(2 + 3) * (4 - 5)(2+3)∗(4−5)
➞ Cây biểu thức tương ứng:



2.3. Các dạng biểu diễn thông qua duyệt cây

Kiểu duyệt	**Biểu diễn**	**Kết quả từ cây trên**
Inorder (LNR)	Trung tố (infix)	2 + 3 * 4 - 5 *(không đủ ngoặc, dễ nhầm)*
Preorder (NLR)	Tiền tố (prefix)	* + 2 3 - 4 5
Postorder (LRN)	Hậu tố (postfix)	2 3 + 4 5 - *

2.4. Tính giá trị biểu thức bằng Postorder

Với **biểu thức hậu tố (postfix)**, ta có thể dễ dàng tính toán giá trị bằng **stack**, do quy tắc thực hiện từ trái sang phải và toán tử đứng sau toán hạng.

✅ **Thuật toán:**

1. Duyệt cây **theo Postorder** để lấy biểu thức hậu tố.
2. Duyệt biểu thức hậu tố:
   - Gặp **toán hạng** → đẩy vào stack.
   - Gặp **toán tử** → lấy 2 toán hạng ra khỏi stack, thực hiện phép tính, đẩy kết quả vào lại stack.
3. Kết quả cuối cùng là phần tử còn lại trên stack.

📌 **Cài đặt mẫu:**

```
def eval_postfix(expr):
```

```
    stack = []
    for token in expr:
        if token.isdigit():
            stack.append(int(token))
        else:
            b = stack.pop()
            a = stack.pop()
            if token == '+': stack.append(a + b)
            elif token == '-': stack.append(a - b)
            elif token == '*': stack.append(a * b)
            elif token == '/': stack.append(a / b)
    return stack[0]
```

Ví dụ:

```
expr = ['2', '3', '+', '4', '5', '-', '*']  # (2 + 3) * (4 - 5)
result = eval_postfix(expr)  # ➞ -5
```

2.5. Biểu diễn cây bằng Preorder

Preorder được dùng để **ghi nhớ, truyền tải, hoặc serialize** biểu thức toán học một cách rõ ràng và không mơ hồ, vì tiền tố không cần ngoặc để phân biệt độ ưu tiên.

➞ **Tiền tố của biểu thức**:

```
* + 2 3 - 4 5
```

➞ Ưu điểm:

- Dễ dàng ghi vào file, lưu trữ, hoặc phân tích bởi máy.
- Phù hợp với **trình biên dịch (compiler)** khi xây dựng cây cú pháp (syntax tree).

2.6. Tổng kết so sánh hai dạng duyệt

Tiêu chí	**Preorder (Prefix)**	**Postorder (Postfix)**
Biểu diễn	Dùng cho lưu trữ, biểu diễn	Dùng để tính toán giá trị
Tính giá trị	Không trực tiếp, cần cây	Dễ dàng qua stack
Dễ hiểu cho con người	Ít trực quan hơn Inorder	Thường dùng cho máy tính
Dùng trong thực tế	Compiler, biên dịch	Máy tính, trình tính toán, máy ảo

2.7. Ứng dụng thực tiễn

Ứng dụng	Dạng duyệt và vai trò
Trình biên dịch (compiler)	Xây dựng cây cú pháp → Preorder lưu cấu trúc
Máy tính biểu thức (calculator)	Postorder để dễ đánh giá bằng stack
Máy ảo (virtual machine)	Biểu thức hậu tố → nạp lệnh vào runtime stack
Serialization cây biểu thức	Preorder để lưu/ghi dữ liệu có cấu trúc
AI và hệ chuyên gia (rule trees)	Tiền tố để phân tích quyết định

✅ **Kết luận phần 2**

- Cây biểu thức là một ví dụ điển hình của **ứng dụng toán rời rạc vào lập trình**, trong đó cây và duyệt cây đóng vai trò trung tâm.
- Hiểu và triển khai được các **dạng biểu diễn biểu thức** là bước quan trọng trong việc xây dựng:
  - Trình biên dịch
  - Hệ thống AI xử lý quy tắc
  - Công cụ toán học (CAS – Computer Algebra Systems)

3. Kiểm tra cấu trúc và so sánh hai cây

3.1. Vấn đề đặt ra

Trong nhiều ứng dụng, ta cần xác định xem **hai cây nhị phân có giống nhau hay không**, có thể theo nhiều tiêu chí:

- **Cùng cấu trúc và cùng giá trị** → cây "giống hệt" (identical)
- **Cùng cấu trúc nhưng giá trị khác** → cấu trúc tương đương (isomorphic)
- **Giống về mặt "hình học"** nếu cho phép hoán đổi trái-phải (mirror symmetry)
- **So sánh một phần (subtree)**

🧠 **Bài toán cơ bản**: Kiểm tra hai cây có **giống hệt nhau** hay không (gồm cả cấu trúc và giá trị từng nút)

3.2. Duyệt kết hợp là gì?

Ta sẽ **duyệt cả hai cây đồng thời**, theo cùng một thứ tự (Preorder, Inorder, hoặc Postorder), để so sánh từng cặp nút tương ứng.

➞ Điều này được gọi là **duyệt kết hợp (simultaneous traversal)**.

3.3. Thuật toán đệ quy

✅ **Mã giả (Python):**

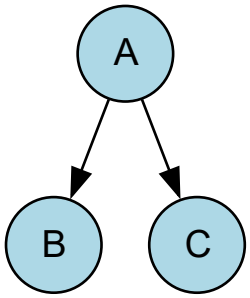
```
def is_same_tree(t1, t2):
    if not t1 and not t2:
        return True  # cả hai cùng rỗng
    if not t1 or not t2:
        return False  # chỉ một trong hai là rỗng
    if t1.value != t2.value:
        return False  # giá trị khác nhau
    return (
        is_same_tree(t1.left, t2.left) and
        is_same_tree(t1.right, t2.right)
    )
```

📌 **Duyệt theo Preorder (NLR): Gốc → Trái → Phải**

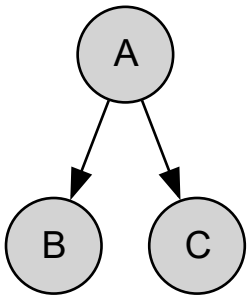
- So sánh giá trị hiện tại → đệ quy trái → đệ quy phải

3.4. Ví dụ minh họa

✅ **Cây 1:**

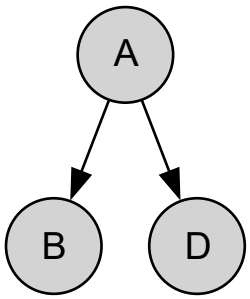


✅ **Cây 2:**



➞ Kết quả: **True**

❌ **Cây 3:**



➞ Kết quả so sánh với Cây 1: **False** (khác giá trị tại nút phải)

3.5. So sánh không đệ quy (sử dụng Stack)

Trong môi trường giới hạn bộ nhớ hoặc khi không muốn dùng call stack, ta có thể dùng duyệt không đệ quy:

```
def is_same_tree_iterative(t1, t2):
    stack = [(t1, t2)]
    while stack:
        n1, n2 = stack.pop()
        if not n1 and not n2:
            continue
        if not n1 or not n2:
            return False
        if n1.value != n2.value:
            return False
        stack.append((n1.right, n2.right))
        stack.append((n1.left, n2.left))
    return True
```

3.6. Mở rộng: So sánh cấu trúc nhưng không cần giống giá trị

✅ **Kiểm tra đẳng cấu (structure equality):**

```
def is_same_structure(t1, t2):
    if not t1 and not t2:
        return True
    if not t1 or not t2:
        return False
    return (
        is_same_structure(t1.left, t2.left) and
        is_same_structure(t1.right, t2.right)
    )
```

➞ Ứng dụng trong kiểm thử hệ thống, khi chỉ cần khớp cấu trúc (ví dụ, kiểm thử dữ liệu JSON, XML, cây DOM,…)

3.7. Ứng dụng thực tiễn

Bài toán / hệ thống	**Mục tiêu kiểm tra**
Kiểm thử phần mềm (test case tree)	So sánh cấu trúc cây đầu ra
Công cụ so sánh cây DOM HTML/XML	So khớp cấu trúc & nội dung thẻ
Compiler (AST – abstract syntax tree)	Kiểm tra biểu thức có tương đương cú pháp
Hệ thống đồng bộ (sync)	So cây thư mục hoặc cấu trúc dữ liệu cây

3.8. Độ phức tạp

Thuật toán kiểm tra	**Thời gian**	**Không gian**
So sánh đệ quy	O(n)	O(h)
So sánh không đệ quy	O(n)	O(h) stack
So sánh cấu trúc	O(n)	O(h)

Trong đó:

- nnn: số nút
- hhh: chiều cao cây

✅ **Kết luận phần 3**

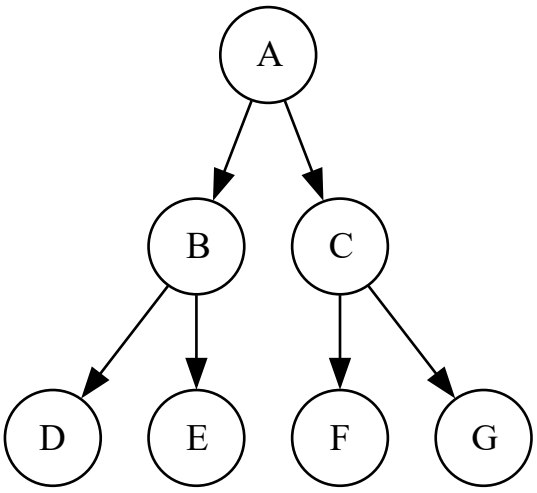
- Duyệt kết hợp là **kỹ thuật nền tảng** để kiểm tra hai cây có giống nhau hay không.
- Việc so sánh cây giúp đánh giá:
  - Tính đúng đắn (cấu trúc, giá trị)
  - Tính tương đương cú pháp (syntax)
  - Tính đối xứng, đẳng cấu
- Trong thực tế, kỹ thuật này có thể mở rộng để **so sánh cây lớn, cây con, hoặc so khớp mẫu (pattern matching)** trong AI và xử lý ngôn ngữ.

IV. Minh họa và ví dụ

1. Cây nhị phân đơn giản gồm 7 nút

Chúng ta sử dụng một cây nhị phân đầy đủ (full binary tree) với 7 nút để minh họa cho tất cả các thuật toán duyệt đã học.

✅ **Cấu trúc cây:**



2. Kết quả duyệt theo từng thuật toán

Phương pháp	Trình tự kết quả
Inorder (LNR)	D, B, E, A, F, C, G
Preorder (NLR)	A, B, D, E, C, F, G
Postorder (LRN)	D, E, B, F, G, C, A
Level-order	A, B, C, D, E, F, G

3. Cài đặt bằng Python

Chúng ta xây dựng lớp Node cơ bản và triển khai các phương pháp duyệt khác nhau.

✅ 3.1. Cấu trúc cây:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# Tạo cây
root = Node('A')
root.left = Node('B')
root.right = Node('C')
root.left.left = Node('D')
root.left.right = Node('E')
root.right.left = Node('F')
root.right.right = Node('G')
```

✅ 3.2. Duyệt cây bằng đệ quy

```
def inorder(node):
    if node:
        inorder(node.left)
        print(node.value, end=' ')

def preorder(node):
    if node:
        print(node.value, end=' ')
        preorder(node.left)
        preorder(node.right)

def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.value, end=' ')
```

✅ 3.3. Duyệt cây theo mức (Level-order) dùng queue

```
from collections import deque

def level_order(root):
    if root is None:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

4. Kết quả khi chạy từng phương pháp

```
print("Inorder:")
inorder(root)  # → D B E A F C G

print("\nPreorder:")
```

```
preorder(root)  # → A B D E C F G

print("\nPostorder:")
postorder(root)  # → D E B F G C A

print("\nLevel-order:")
level_order(root)  # → A B C D E F G
```

5. Liên hệ thực tiễn

Phương pháp	Ứng dụng thực tế
Inorder	Duyệt BST theo thứ tự tăng dần
Preorder	Biểu diễn cấu trúc cây, serialization
Postorder	Giải biểu thức hậu tố, hủy cây
Level-order	Duyệt rộng (BFS), cấu trúc cây hoàn chỉnh, đồng bộ

✅ **Tổng kết phần minh họa**

- Cây 7 nút đơn giản cung cấp một mô hình trực quan, phù hợp để luyện tập và kiểm thử các thuật toán duyệt.
- Việc thực hành bằng Python giúp sinh viên **kết nối lý thuyết với thực hành**, từ đó hiểu rõ bản chất của từng phương pháp duyệt.
- Với nền tảng này, sinh viên có thể áp dụng vào các bài toán nâng cao: xây dựng cây biểu thức, cây quyết định, hệ thống AI dạng cây,…

V. Tổng kết kiến thức

1. Các dạng cây nhị phân phổ biến và đặc điểm

Loại cây nhị phân	Đặc điểm chính	Ứng dụng thực tiễn
Cây nhị phân thường	Mỗi nút có tối đa 2 con (trái, phải)	Cấu trúc tổng quát, cơ sở của các cây khác
Cây nhị phân đầy đủ	Mọi nút đều có 0 hoặc 2 con	Dùng trong các mô hình phân chia nhị phân, cây quyết định
Cây nhị phân hoàn chỉnh	Mọi mức đầy đủ, trừ mức cuối điền từ trái sang phải	Triển khai Heap, cây lưu bằng mảng
Cây nhị phân tìm kiếm (BST)	Trái < Gốc < Phải	Tìm kiếm, chèn, xóa hiệu quả trong cấu trúc dữ liệu

Loại cây nhị phân	Đặc điểm chính	Ứng dụng thực tiễn
Cây nhị phân cân bằng (AVL/Red-Black)	Đảm bảo chiều cao cân đối để tránh thoái hóa hiệu năng	STL (C++), TreeMap (Java), cơ sở dữ liệu nội bộ
Cây biểu thức (Expression Tree)	Lá là toán hạng, nút trong là toán tử	Xây dựng và tính toán biểu thức, trình biên dịch

2. 4 phương pháp duyệt cơ bản

Tên thuật toán	Trình tự thăm	Kết quả	Mô hình logic	Ứng dụng điển hình
Inorder (LNR)	Trái → Gốc → Phải	Trung tố	Tăng dần trong BST	Kiểm tra sắp xếp, xử lý biểu thức toán học
Preorder (NLR)	Gốc → Trái → Phải	Tiền tố	Duyệt gốc trước, dựng cấu trúc	Ghi cây ra file, serialization, AST trong compiler
Postorder (LRN)	Trái → Phải → Gốc	Hậu tố	Tính toán từ dưới lên	Giải biểu thức hậu tố, xóa cây, máy tính RPN
Level-order	Từng mức trái → phải	Theo tầng	Duyệt rộng (BFS)	Xây dựng cây từ mảng, đồng bộ cấu trúc, tìm kiếm tầng

3. Ứng dụng thực tế trong CNTT và lập trình thuật toán

Lĩnh vực / Hệ thống	Ứng dụng cụ thể	Loại cây / duyệt dùng
Trình biên dịch (compiler)	Xây dựng cây cú pháp, xử lý biểu thức	Cây biểu thức, Pre/Postorder
Cơ sở dữ liệu	Chỉ mục tìm kiếm (B+, AVL, RB Tree)	BST, duyệt Inorder
Hệ thống tập tin, đồng bộ dữ liệu	So sánh cây thư mục, cấu trúc lưu trữ	Duyệt kết hợp, Level-order
Game AI, AI phân loại	Cây quyết định, cây hành động	Preorder / Level-order
Tìm kiếm và sắp xếp	Tìm kiếm nhanh, sắp thứ tự	BST + Inorder
Truyền và lưu trữ dữ liệu cây	Serialization/Deserialization cấu trúc JSON/XML	Preorder, Level-order
Trình tính toán biểu thức	Tính biểu thức trung tố, tiền tố, hậu tố	Cây biểu thức, Postorder

Lĩnh vực / Hệ thống	Ứng dụng cụ thể	Loại cây / duyệt dùng
Thuật toán đồ thị	Duyệt BFS dạng cây, dựng cây bao phủ	Level-order

✅ Lời khuyên cho sinh viên ngành CNTT

- **Không chỉ học cách duyệt**, hãy học **khi nào nên dùng cách duyệt nào**. Mỗi bài toán yêu cầu một cách duyệt tối ưu riêng.
- **Luyện tập với cây nhỏ** (5–10 nút) bằng tay và code để hiểu sâu, sau đó **nâng cấp lên cây lớn**, cây ngẫu nhiên hoặc mất cân bằng.
- **Kết hợp duyệt với thuật toán khác** (Tìm kiếm, dynamic programming, đồ thị) để giải bài toán phức tạp hơn.

✅ Ghi nhớ nhanh

Câu hỏi	Câu trả lời
Duyệt nào in BST theo thứ tự?	Inorder
Duyệt nào để đánh giá biểu thức?	Postorder
Duyệt nào để dựng lại cây?	Preorder
Duyệt nào dùng hàng đợi?	Level-order
Duyệt nào dùng đệ quy hiệu quả?	Inorder, Postorder

✅ Tổng kết

Duyệt cây nhị phân không chỉ là phần kiến thức cơ bản của Toán rời rạc, mà còn là **nền tảng quan trọng để sinh viên IT xây dựng kỹ năng giải thuật**, tư duy phân rã, và xử lý dữ liệu phân cấp — những kỹ năng then chốt trong lập trình hiện đại.

PHẦN 2: BÀI TẬP CÓ LỜI GIẢI

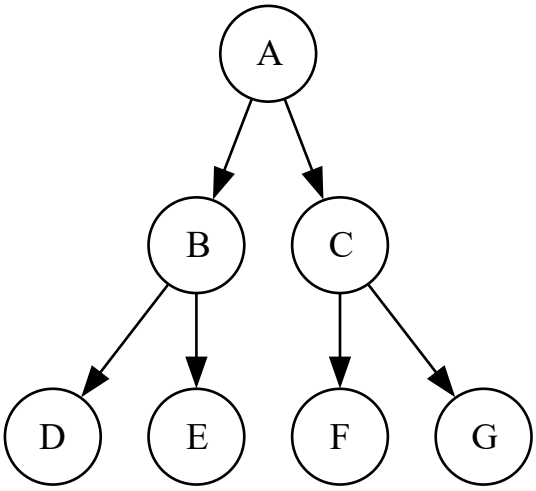
📝 Bài 1. Phân loại và nhận diện các loại cây nhị phân

Cho 5 cây nhị phân được biểu diễn bằng sơ đồ sau (dưới dạng Graphviz). Với mỗi cây, hãy xác định nó thuộc loại nào trong số sau:

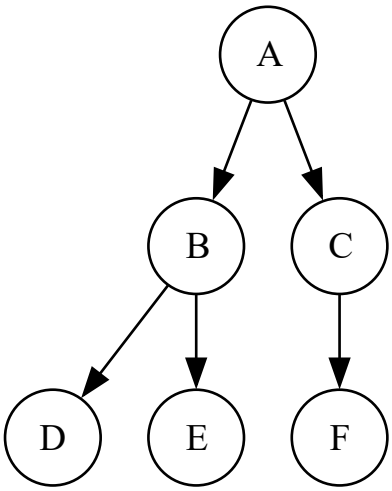
- Cây nhị phân thường
- Cây nhị phân đầy đủ (Full Binary Tree)
- Cây nhị phân hoàn chỉnh (Complete Binary Tree)
- Cây nhị phân tìm kiếm (BST)
- Cây nhị phân cân bằng (Balanced BST)

Yêu cầu: Với mỗi cây, chỉ ra **tất cả các loại** mà nó thuộc (một cây có thể thuộc nhiều loại).

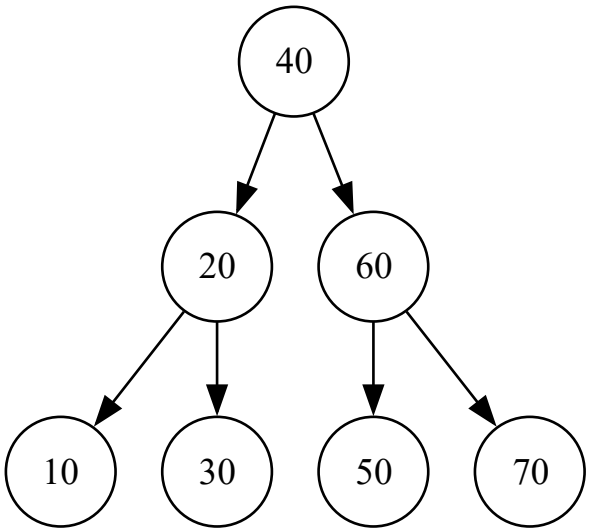
Cây A – Cây nhị phân đầy đủ và cân bằng



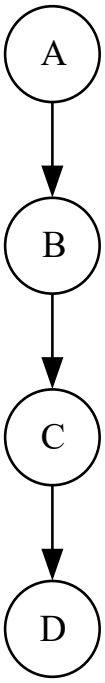
Cây B – Cây hoàn chỉnh nhưng không đầy đủ



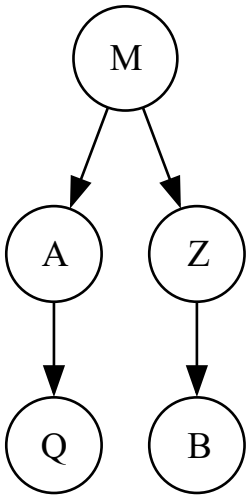
Cây C – Cây nhị phân tìm kiếm và cân bằng



Cây D – Cây lệch trái



Cây E – Cây nhị phân thường (không là BST, không cân bằng)



🔍 **Phân tích và lời giải chi tiết**

Cây	Phân loại thuộc	Lý do
A	✅ Cây nhị phân ✅ Full Binary Tree ✅ Balanced BST *(về hình học)*	Mọi nút đều có 0 hoặc 2 con và chiều cao cân bằng
B	✅ Cây nhị phân ✅ Complete Binary Tree	Mọi mức đầy đủ, trừ mức cuối điền từ trái → phải. Nút C chỉ có 1 con trái → không phải Full
C	✅ Cây nhị phân ✅ Full Binary Tree ✅ Balanced BST ✅ BST	Thỏa mãn BST (Trái < Gốc < Phải), độ cao cân bằng, đầy đủ
D	✅ Cây nhị phân 🚫 Không hoàn chỉnh, không đầy đủ 🚫 Không cân bằng 🚫 Không BST	Cây lệch hoàn toàn về trái, độ sâu cao, không sắp xếp BST
E	✅ Cây nhị phân	Không phải BST (Q nằm sai vị trí), không cân bằng, không đầy đủ

✅ **Tóm tắt đáp án**

Cây	Nhị phân	Full	Complete	BST	Cân bằng
A	✅	✅	❌	❌	✅
B	✅	❌	✅	❌	❌
C	✅	✅	✅	✅	✅
D	✅	❌	❌	❌	❌
E	✅	❌	❌	❌	❌

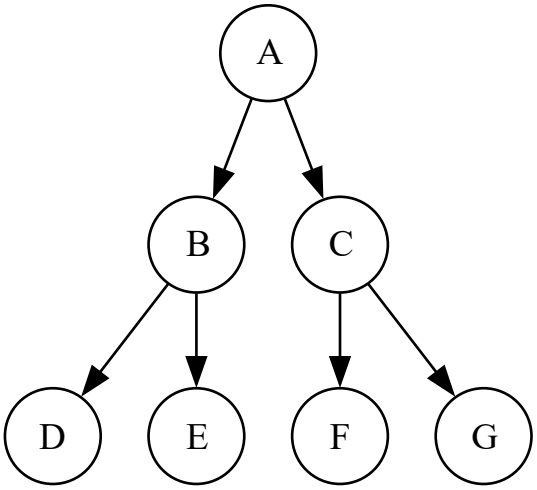
🧠 **Gợi ý mở rộng**

Bạn có thể yêu cầu sinh viên:

- Chuyển cây B thành cây đầy đủ bằng cách thêm nút
- Kiểm tra chiều cao và số lá để xác định tính cân bằng
- Chuyển cây D thành BST hợp lệ bằng hoán đổi giá trị

📝 Bài 2. Thực hiện các phương pháp duyệt trên cây nhị phân

Cho cây nhị phân gồm 7 nút như sau:



Yêu cầu

1. Viết ra kết quả **duyệt cây** theo 4 phương pháp:
    - Inorder (LNR)
    - Preorder (NLR)
    - Postorder (LRN)
    - Level-order (duyệt theo mức)
2. **(Tùy chọn)**: Cài đặt các thuật toán bằng Python để kiểm tra kết quả.

🎯 Mục tiêu

- Ôn tập thứ tự duyệt và thao tác từng bước trên cây nhị phân
- So sánh sự khác biệt giữa 4 cách duyệt
- Chuẩn bị kiến thức cho phần ứng dụng cây trong tìm kiếm, biểu thức, serialization,...

✅ Phân tích và lời giải

1. Inorder Traversal (LNR – Left, Node, Right)

- Duyệt trái → gốc → phải
- Trình tự:

D → B → E → A → F → C → G

2. Preorder Traversal (NLR – Node, Left, Right)

- Gốc → trái → phải
- Trình tự:

A → B → D → E → C → F → G

3. Postorder Traversal (LRN – Left, Right, Node)

- Trái → phải → gốc
- Trình tự:

D → E → B → F → G → C → A

4. Level-order Traversal (Duyệt theo mức)

- Từ trên xuống dưới, từ trái sang phải mỗi mức
- Trình tự:

A → B → C → D → E → F → G

💡 Mã Python kiểm tra kết quả

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# Tạo cây
root = Node('A')
root.left = Node('B')
root.right = Node('C')
root.left.left = Node('D')
```

```
root.left.right = Node('E')
root.right.left = Node('F')
root.right.right = Node('G')

def inorder(node):
    if node:
        inorder(node.left)
        print(node.value, end=' ')

def preorder(node):
    if node:
        print(node.value, end=' ')
        preorder(node.left)
        preorder(node.right)

def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.value, end=' ')

from collections import deque
def level_order(root):
    if root is None:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

✅ **Kết quả in ra từ chương trình**

```
Inorder:     D B E A F C G
Preorder:    A B D E C F G
Postorder:   D E B F G C A
Level-order: A B C D E F G
```

✅ **Tổng kết kiến thức**

Duyệt	Trình tự (7 nút A-G)	Ứng dụng điển hình
Inorder	D B E A F C G	Duyệt BST theo thứ tự tăng
Preorder	A B D E C F G	Ghi lại cấu trúc cây, prefix notation
Postorder	D E B F G C A	Giải biểu thức, hủy cây
Level	A B C D E F G	Serialization, dựng cây hoàn chỉnh từ mảng

📝 Bài 3. Cài đặt và so sánh các phương pháp duyệt đệ quy và không đệ quy

Viết chương trình Python để:

1. Cài đặt **duyệt Inorder** bằng **đệ quy** và **không đệ quy** (dùng stack).
2. Dựng cây nhị phân **lệch trái** gồm **1000 nút** (mỗi nút chỉ có con trái).
3. **So sánh thời gian chạy và độ sâu stack (gọi hàm)** của hai cách duyệt.

🎯 Mục tiêu

- Thấy rõ hiệu năng giữa hai cách duyệt cây nhị phân.
- Nhận biết giới hạn của **đệ quy** trong trường hợp **cây sâu**, dễ gây **tràn ngăn xếp (stack overflow)**.
- Làm quen với việc đo hiệu năng trong thực nghiệm thuật toán.

1. Mô tả cấu trúc cây lệch trái 1000 nút

```
   1
  /
 2
/
3
.
.
1000
```

➞ Cây có **chiều cao h = 1000**, **rất mất cân bằng**.

2. Cài đặt cây lệch trái

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

```
def build_left_skewed_tree(n):
    root = Node(1)
    current = root
    for i in range(2, n + 1):
        current.left = Node(i)
        current = current.left
    return root
```

3. Duyệt Inorder: Đệ quy

```
def inorder_recursive(node):
    if node:
        inorder_recursive(node.left)
        # Giả lập xử lý (in giá trị)
        _ = node.value
```

4. Duyệt Inorder: Không đệ quy (dùng stack)

```
def inorder_iterative(root):
    stack = []
    current = root
    while current or stack:
        while current:
            stack.append(current)
            current = current.left
        current = stack.pop()
        # Giả lập xử lý (in giá trị)
        _ = current.value
        current = current.right
```

5. So sánh thời gian thực thi

```
import time

tree = build_left_skewed_tree(1000)

# Đệ quy
start = time.perf_counter()
inorder_recursive(tree)
end = time.perf_counter()
print(f"Recursive Inorder: {end - start:.6f} seconds")
```

```
# Không đệ quy
start = time.perf_counter()
inorder_iterative(tree)
end = time.perf_counter()
print(f"Iterative Inorder: {end - start:.6f} seconds")
```

✅ **Kết quả dự kiến**

Tiêu chí	**Đệ quy**	**Không đệ quy**
Thời gian (trung bình)	~0.001–0.005 giây	~0.001–0.005 giây
Chiều sâu stack hệ thống	**1000 mức gọi hàm**	Không sử dụng call stack
Rủi ro tràn ngăn xếp	**Có thể xảy ra** (tuỳ hệ điều hành và giới hạn hệ thống)	Không xảy ra
Khả năng kiểm soát luồng	Khó	Linh hoạt hơn (pause, resume)
Độ dài code	Ngắn hơn	Dài hơn

⚠️ **Chú ý kỹ thuật**

- Một số môi trường (như Python mặc định) có giới hạn **đệ quy là 1000 mức**. Có thể gây lỗi:

```
RecursionError: maximum recursion depth exceeded
```

- Để kiểm tra giới hạn hoặc tăng giới hạn:

```
import sys
print(sys.getrecursionlimit())
sys.setrecursionlimit(2000)  # (cẩn thận)
```

⚠️ **Không khuyến khích tăng recursion limit tùy tiện**, vì dễ gây lỗi hệ thống.

✅ **Kết luận**

- **Với cây sâu**, phương pháp **không đệ quy an toàn hơn và có thể kiểm soát tốt hơn** trong thực tế.
- **Đệ quy thích hợp cho cây cân bằng hoặc nhỏ**, nhờ code ngắn và dễ hiểu.
- Khi xử lý dữ liệu lớn (ví dụ: cây phân tích cú pháp, cây dữ liệu máy học...), **nên ưu tiên dùng stack tường minh** hoặc thư viện chuyên dụng.

📝 Bài 4. Biểu diễn và tính giá trị biểu thức số học bằng cây

Cho biểu thức số học trung tố:
(5+3)∗(10−2)(5 + 3) * (10 - 2)(5+3)∗(10−2)
Thực hiện các yêu cầu sau:

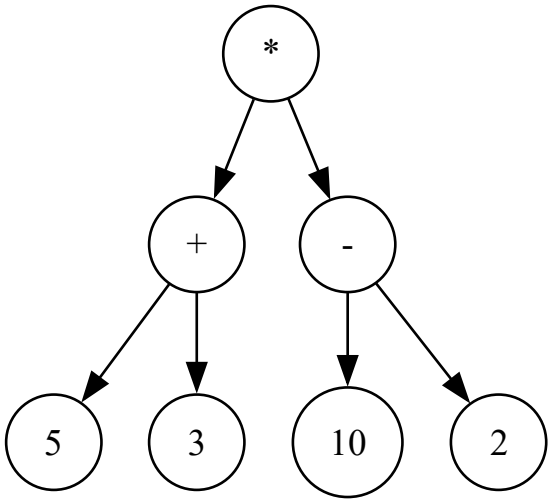
1. Dựng cây biểu thức tương ứng.
2. Viết lại biểu thức theo dạng:
    - **Tiền tố (prefix)**
    - **Hậu tố (postfix)**
3. Tính giá trị của biểu thức bằng cách **duyệt hậu tố (postorder traversal)**.

🎯 Mục tiêu

- Biết cách chuyển đổi biểu thức toán học thành cây nhị phân.
- Hiểu mối liên hệ giữa các phương pháp duyệt cây và các dạng biểu diễn biểu thức (prefix/postfix).
- Ứng dụng duyệt cây vào việc **đánh giá biểu thức** như một máy tính đơn giản (mini calculator).

✅ 1. Dựng cây biểu thức

Biểu thức gốc:
(5+3)∗(10−2)(5 + 3) * (10 - 2)(5+3)∗(10−2)
➞ Cây biểu thức nhị phân tương ứng:



Nút **gốc**: phép nhân *

- Hai cây con:
    - Trái: (5 + 3)
    - Phải: (10 - 2)

✅ 2. Biểu diễn theo các dạng duyệt

a. Tiền tố (prefix) – Duyệt Preorder (NLR)

```
* + 5 3 - 10 2
```

b. Hậu tố (postfix) – Duyệt Postorder (LRN)

```
5 3 + 10 2 - *
```

✅ 3. Tính giá trị biểu thức bằng duyệt hậu tố

🧠 **Ý tưởng:**

- Duyệt hậu tố tạo ra chuỗi **postfix**
- Dùng **stack** để tính biểu thức hậu tố (Reverse Polish Notation):
  - Gặp số: push vào stack
  - Gặp toán tử: pop 2 số, thực hiện phép toán, push kết quả

✅ Cài đặt bằng Python

```
def eval_postfix(expr):
    stack = []
    for token in expr:
        if token.isdigit():
            stack.append(int(token))
        else:
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                stack.append(a / b)
    return stack[0]

# Biểu thức hậu tố từ cây: 5 3 + 10 2 - *
postfix_expr = ['5', '3', '+', '10', '2', '-', '*']
result = eval_postfix(postfix_expr)
print("Giá trị biểu thức:", result)
```

✅ Kết quả:

```
Giá trị biểu thức: 64
```

🧮 Phân tích tính toán

Bước 1: 5 3 + → push 5 → push 3 → pop → 5 + 3 = 8 → push 8
Bước 2: 10 2 - → push 10 → push 2 → pop → 10 - 2 = 8 → push 8
Bước 3: * → pop 8, 8 → 8 * 8 = 64

✅ Tổng kết bài học

Thành phần	Nội dung
Biểu thức trung tố	(5 + 3) * (10 - 2)
Cây biểu thức	Nhánh trái là +, nhánh phải là -
Tiền tố (prefix)	* + 5 3 - 10 2
Hậu tố (postfix)	5 3 + 10 2 - *
Giá trị tính được	64

🔎 Ứng dụng mở rộng

Ứng dụng	Vai trò cây biểu thức
Trình biên dịch (compiler)	Biểu diễn và phân tích biểu thức cú pháp
Máy tính (calculator)	Thực hiện tính toán hậu tố
Serialization biểu thức	Duyệt Preorder
Phân tích dữ liệu ngôn ngữ	Abstract Syntax Tree (AST)

📝 Bài 5. Kiểm tra hai cây có giống nhau hay không

Cho hai cây nhị phân được biểu diễn dưới dạng sơ đồ hoặc danh sách. Hãy viết chương trình kiểm tra:

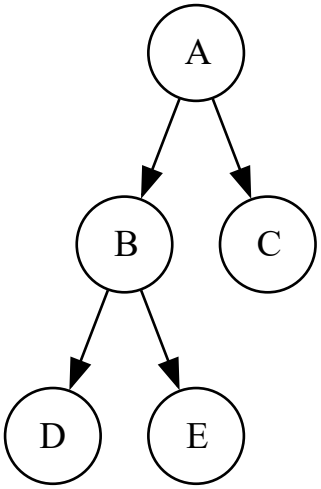
1. Hai cây **giống hệt nhau** về cả cấu trúc lẫn giá trị.
2. Hai cây có **cùng cấu trúc** nhưng giá trị các nút **khác nhau**.
3. Hai cây có **đối xứng nhau** (mirror symmetry).

🎯 Mục tiêu

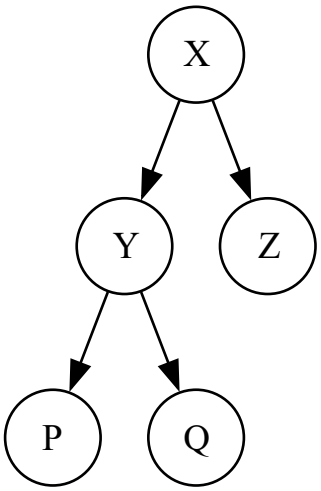
- Ôn tập khái niệm **cấu trúc cây**, **đệ quy**, và **duyệt kết hợp**.
- Làm quen với kiểm thử **tính đẳng cấu**, **tính tương đương cú pháp** — những kỹ năng thường dùng trong xử lý cây cú pháp (AST), DOM XML, hoặc các hệ thống so khớp cây (diff tool, kiểm thử phần mềm).

✅ 1. Hai cây minh họa

Cây 1 (T1)

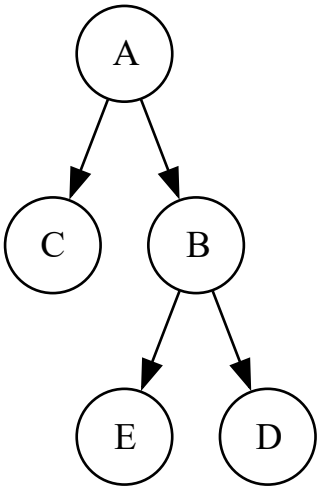


Cây 2 (T2)



- **Cây T1 và T2** có cùng cấu trúc, nhưng giá trị các nút khác nhau.

Cây 3 (T3) – đối xứng với T1



- Cây này là **ảnh gương** của T1 (mirror tree).

✅ 2. Cài đặt các hàm kiểm tra

2.1. Cây nhị phân cơ bản

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

2.2. Hàm kiểm tra "giống hệt nhau"

```
def is_identical(t1, t2):
    if not t1 and not t2:
        return True
    if not t1 or not t2:
        return False
    if t1.value != t2.value:
        return False
    return (
        is_identical(t1.left, t2.left) and
        is_identical(t1.right, t2.right)
    )
```

2.3. Hàm kiểm tra "cùng cấu trúc"

```
def is_same_structure(t1, t2):
    if not t1 and not t2:
        return True
    if not t1 or not t2:
        return False
    return (
        is_same_structure(t1.left, t2.left) and
        is_same_structure(t1.right, t2.right)
    )
```

2.4. Hàm kiểm tra "đối xứng (mirror)"

```
def is_mirror(t1, t2):
    if not t1 and not t2:
        return True
    if not t1 or not t2:
        return False
    return (
```

```
        t1.value == t2.value and
        is_mirror(t1.left, t2.right) and
        is_mirror(t1.right, t2.left)
    )
```

✅ 3. Dữ liệu kiểm tra

```
# T1
t1 = Node('A')
t1.left = Node('B')
t1.right = Node('C')
t1.left.left = Node('D')
t1.left.right = Node('E')

# T2 – cùng cấu trúc, khác giá trị
t2 = Node('X')
t2.left = Node('Y')
t2.right = Node('Z')
t2.left.left = Node('P')
t2.left.right = Node('Q')

# T3 – đối xứng với T1
t3 = Node('A')
t3.left = Node('C')
t3.right = Node('B')
t3.right.left = Node('E')
t3.right.right = Node('D')
```

✅ 4. Kiểm tra và kết quả

```
print("T1 vs T2 – giống hệt:", is_identical(t1, t2))        # False
print("T1 vs T2 – cùng cấu trúc:", is_same_structure(t1, t2))  # True
print("T1 vs T3 – đối xứng:", is_mirror(t1, t3))            # True
```

✅ Kết quả mong đợi

```
T1 vs T2 – giống hệt: False
T1 vs T2 – cùng cấu trúc: True
T1 vs T3 – đối xứng: True
```

📌 Phân biệt 3 tiêu chí so sánh

Tiêu chí	So sánh gì	Ứng dụng thực tế
Giống hệt	Cấu trúc + giá trị	Kiểm tra đồng bộ dữ liệu, so sánh snapshot
Cùng cấu trúc	Chỉ cấu trúc	So khớp mẫu, AST tương đương cú pháp
Đối xứng (mirror)	Phản chiếu trái-phải	Cây gương, bài toán đối xứng trong AI

✅ Tổng kết bài học

- Việc kiểm tra cấu trúc và giá trị cây là một kỹ thuật then chốt trong:
  - **Kiểm thử hệ thống** (test tree structure)
  - **Xác minh đồng bộ dữ liệu**
  - **Biên dịch và phân tích mã nguồn**
  - **So sánh cấu trúc DOM/XML**
- Duyệt kết hợp (simultaneous traversal) là nền tảng để xây dựng các thuật toán kiểm tra này một cách tối ưu.

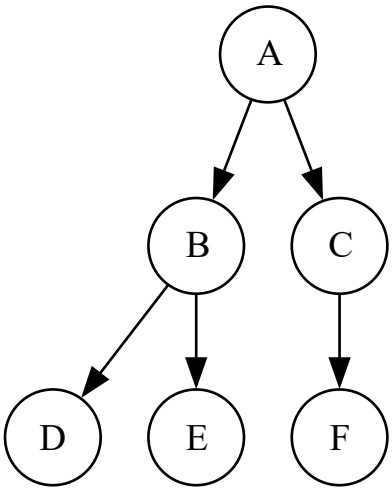
PHẦN 3: TRẮC NGHIỆM

📘 Phần A – Nhận biết khái niệm cơ bản (Câu 1–5)

❓ Câu 1: Trong cây nhị phân, mỗi nút có tối đa bao nhiêu nút con?

A. 1
B. 2
C. 3
D. Không giới hạn

❓ Câu 2: Cây nào sau đây là cây nhị phân đầy đủ?



A. Đúng
B. Sai
C. Chỉ đúng nếu thêm nút con phải cho C
D. Không thể xác định

❓ Câu 3: Duyệt cây Inorder có thứ tự thăm nút là:

A. Node → Left → Right
B. Left → Node → Right
C. Left → Right → Node
D. Right → Left → Node

❓ Câu 4

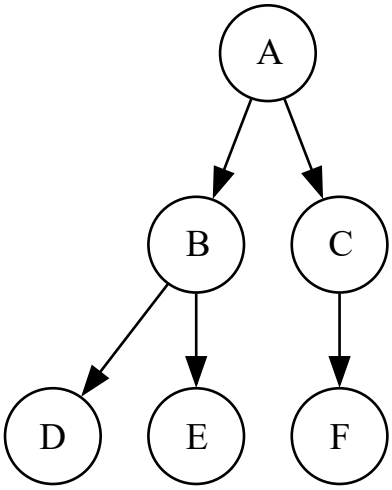
Câu hỏi: Cây nhị phân tìm kiếm (BST) có tính chất nào sau đây?
A. Các nút ở mức thấp luôn có giá trị nhỏ hơn gốc
B. Tất cả giá trị cây con trái nhỏ hơn gốc, cây con phải lớn hơn gốc
C. Giá trị các nút tăng dần theo chiều cao
D. Không có quy tắc sắp xếp cụ thể

❓ Câu 5: Duyệt cây nào trả về danh sách theo thứ tự tăng dần với cây BST?

A. Preorder
B. Inorder
C. Postorder
D. Level-order

📗 Phần B – Duyệt cây (Câu 6–12)

❓ Câu 6: Với cây sau, kết quả duyệt Preorder là gì?



A. D B E A F C
B. A B D E C F
C. D E B F C A
D. A D B E F C

❓ Câu 7: Với cây ở Câu 6, kết quả duyệt Inorder là:

A. D B E A F C
B. A B D E C F
C. D E B F C A
D. A D E B F C

❓ Câu 8: Duyệt Level-order thực hiện theo:

A. Đệ quy
B. Stack
C. Queue
D. TreeMap

❓ Câu 9: Với cây lệch trái có 1000 nút, dùng duyệt đệ quy có thể gây ra lỗi nào?

A. SyntaxError
B. MemoryError
C. RecursionError
D. TypeError

❓ Câu 10
Câu hỏi: Trong duyệt Postorder, thứ tự xử lý nút là:
A. Gốc → Trái → Phải
B. Trái → Phải → Gốc
C. Phải → Gốc → Trái
D. Trái → Gốc → Phải

❓ Câu 11: Cách nào dưới đây mô phỏng duyệt đệ quy?
A. Queue
B. Dictionary
C. Stack
D. Set

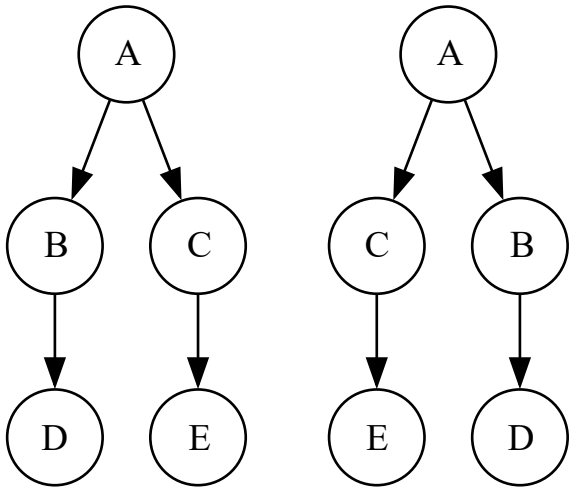
❓ Câu 12: Duyệt cây hậu tố dùng để:
A. Tính giá trị biểu thức
B. Xóa toàn bộ cây
C. Biểu diễn dạng infix
D. A và B đúng

📙 Phần C – Ứng dụng nâng cao (Câu 13–20)

❓ Câu 13: Với biểu thức trung tố (2 + 4) * (3 - 1), hậu tố tương ứng là:
A. 2 + 4 * 3 - 1
B. * + 2 4 - 3 1
C. 2 4 + 3 1 - *
D. 2 3 1 - 4 + *

❓ Câu 14: Hàm nào sau đây kiểm tra hai cây giống hệt?
A. So sánh Inorder duyệt của hai cây
B. So sánh Preorder duyệt
C. So sánh cấu trúc và từng giá trị
D. So sánh chiều cao hai cây

❓ Câu 15: Cặp cây nào sau đây là đối xứng?



A. Cây giống hệt
B. Cây đối xứng
C. Cùng cấu trúc nhưng giá trị khác
D. Không liên quan

❓ Câu 16: Khi so sánh hai cây có cùng số nút và chiều cao, ta cần gì để khẳng định chúng giống hệt?

A. Cùng Inorder
B. Cùng Level-order
C. Cùng cấu trúc và từng giá trị tương ứng
D. Cùng số lá

❓ Câu 17: Độ phức tạp thời gian duyệt cây nhị phân có bao nhiêu?

A. $O(n^2)$
B. O(log n)
C. O(n)
D. O(n log n)

❓ Câu 18: Trong Level-order traversal, độ phức tạp bộ nhớ là bao nhiêu?

A. O(n)
B. O(h)
C. O(1)
D. O(w) với w là chiều rộng lớn nhất

❓ Câu 19: Khi biểu diễn cây bằng mảng (heap), con trái và phải của nút i lần lượt nằm ở vị trí nào?

A. 2i và 2i + 1
B. i + 1 và i + 2
C. 2i + 1 và 2i + 2
D. i / 2 và i / 2 + 1

❓ Câu 20: Kỹ thuật nào phù hợp để lưu trữ cây biểu thức và khôi phục lại sau này?

A. Inorder traversal
B. Serialization bằng Preorder
C. Sorting tất cả giá trị
D. BFS sorting

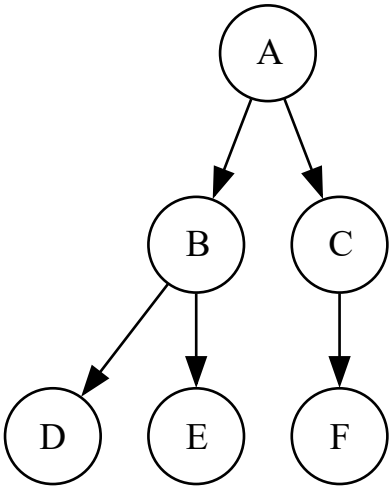
PHẦN 4: BÀI TẬP LUYỆN TẬP

▨ Phần A – Nhận biết và phân loại cây (Bài 1–2)

◈ Bài 1: Phân loại cây nhị phân

Cho các cây sau, hãy xác định cây nào là:

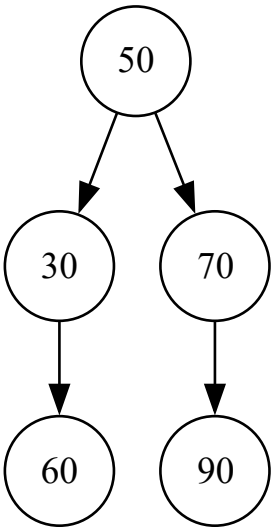
- Cây nhị phân đầy đủ (Full)
- Cây nhị phân hoàn chỉnh (Complete)
- Cây nhị phân tìm kiếm (BST)
- Cây cân bằng



Gợi ý: Kiểm tra số lượng con của mỗi nút, cấu trúc và tính chất sắp xếp.

◈ Bài 2: Kiểm tra BST

Cho cây sau, xác định cây có phải là **BST hợp lệ** hay không:

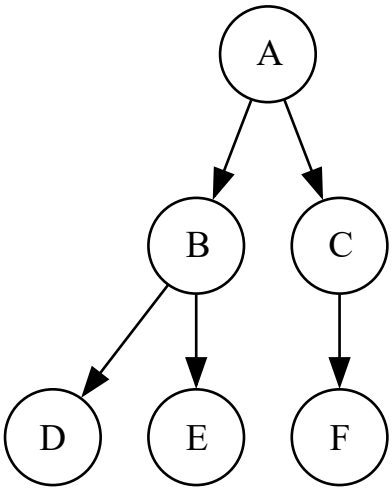


Gợi ý: Duyệt Inorder, kiểm tra tính chất "Trái < Gốc < Phải".

▨ Phần B – Duyệt cây (Bài 3–5)

◈ Bài 3: Viết kết quả duyệt

Cho cây:



Hãy viết kết quả duyệt theo thứ tự:

- Inorder
- Preorder
- Postorder
- Level-order

Gợi ý: Làm từng bước theo thứ tự gốc/trái/phải, dùng bảng đánh dấu.

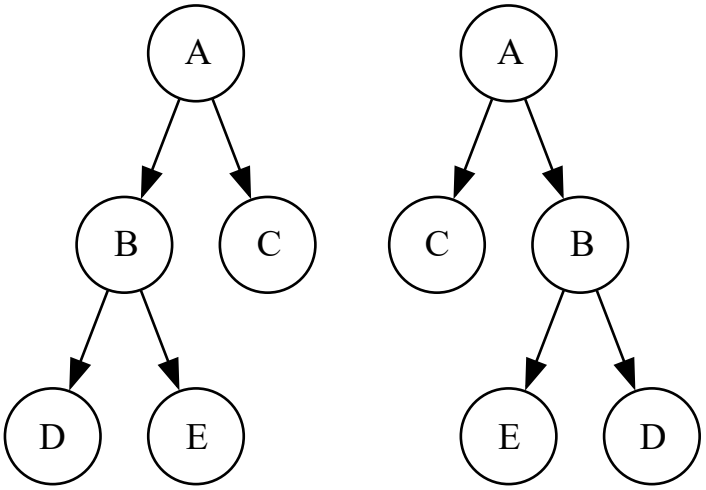
◈ Bài 4: Cài đặt duyệt không đệ quy

Viết hàm Python để duyệt cây theo **Inorder**, nhưng **không sử dụng đệ quy**.
Gợi ý: Sử dụng **stack tường minh**, mô phỏng việc đệ quy đi về trái.

◈ Bài 5: So sánh kết quả duyệt

Cho hai cây, hãy duyệt theo Inorder và cho biết:

- Có cùng kết quả duyệt Inorder không?
- Nếu có, cây có chắc là giống hệt không?



Gợi ý: Hai cây có thể có cùng Inorder nhưng cấu trúc khác nhau.

▩ Phần C – Ứng dụng thực tế (Bài 6–8)

◈ Bài 6: Cây biểu thức hậu tố

Cho biểu thức hậu tố sau:

```
2 3 + 5 1 - *
```

Hãy:

1. Dựng cây biểu thức
2. Viết lại biểu thức theo:
    - Trung tố (infix)
    - Tiền tố (prefix)

Gợi ý: Duyệt Postorder để xây dựng cây từ stack ngược.

◈ Bài 7: Đánh giá biểu thức

Từ cây biểu thức trong Bài 6, viết chương trình Python để **tính giá trị biểu thức** bằng cách **duyệt hậu tố**.
Gợi ý: Stack để thực hiện toán tử, lấy 2 toán hạng ra, push kết quả vào lại.

🔷 Bài 8: So sánh hai cây

Viết hàm Python kiểm tra:

- Hai cây có giống hệt nhau không?
- Hai cây có cùng cấu trúc không?
- Hai cây có đối xứng nhau không?

Gợi ý: Viết 3 hàm dùng duyệt kết hợp (simultaneous traversal).

🟦 Phần D – Phân tích và mở rộng (Bài 9–10)

🔷 Bài 9: Phân tích độ sâu cây lệch trái

Viết hàm dựng cây nhị phân **lệch trái gồm n nút**. Duyệt cây bằng đệ quy và kiểm tra xem:

- Với n = 1000, có bị lỗi tràn ngăn xếp không?
- Cài đặt lại bằng stack tường minh và đo thời gian.

Gợi ý: Dùng sys.setrecursionlimit() để kiểm chứng giới hạn đệ quy.

🔷 Bài 10: Duyệt theo mức và xây cây từ mảng

Cho mảng biểu diễn cây nhị phân hoàn chỉnh:

```
arr = [1, 2, 3, 4, 5, 6, 7]
```

1. Xây cây từ mảng theo thứ tự Level-order.
2. Viết hàm duyệt cây theo Inorder để kiểm tra kết quả.

Gợi ý: Dùng công thức:

- i → trái: 2i + 1, phải: 2i + 2

PHẦN 5: BÀI TẬP DỰ ÁN

🏵️ Dự án 1: Trình biên dịch biểu thức số học đơn giản

⭐⭐ **Mức độ: Trung bình – Khá**

📌 **Chủ đề: Cây biểu thức, duyệt cây, hậu tố, tiền tố**

🎯 Mục tiêu

Xây dựng một chương trình có thể:

1. Nhận đầu vào là biểu thức trung tố (ví dụ: ((2 + 3) * (7 - 4)))
2. Phân tích và dựng **cây biểu thức nhị phân**
3. In ra biểu thức theo dạng:
    - Tiền tố (prefix)
    - Hậu tố (postfix)
4. **Tính giá trị biểu thức**

📄 Yêu cầu chức năng

- **Đầu vào**: Biểu thức trung tố, có ngoặc và 4 toán tử (+, -, *, /)
- **Đầu ra**:
    - Biểu thức tiền tố và hậu tố
    - Kết quả tính giá trị cuối cùng

Ví dụ:

Input: ((2 + 3) * (7 - 4)) Output: Prefix: * + 2 3 - 7 4 Postfix: 2 3 + 7 4 - * Giá trị: 15

🧠 Phân tích nghiệp vụ

- Biểu thức trung tố → cần **chuyển sang cây nhị phân** → có thể dùng **shunting yard algorithm** hoặc **chuyển sang hậu tố trước → dựng cây từ postfix**.
- Duyệt cây:
    - **Preorder → prefix**
    - **Postorder → postfix**
- Tính giá trị → **duyệt hậu tố và dùng stack**

💡 Hướng dẫn kỹ thuật

1. **Chuyển trung tố sang hậu tố** (dùng stack, ưu tiên toán tử)
2. **Dựng cây biểu thức từ hậu tố**:
    - Gặp toán hạng → tạo nút lá
    - Gặp toán tử → pop 2 nút, ghép thành cây con
3. Duyệt cây:
    - Preorder → in prefix
    - Postorder → in postfix
4. Tính toán hậu tố → dùng stack như máy tính RPN

✅ Kết quả kỳ vọng

Sinh viên có thể:

- Làm việc với cây nhị phân theo cách thực tiễn
- Hiểu quy trình **biên dịch và tính toán biểu thức**
- Áp dụng các dạng duyệt để **thao tác dữ liệu theo yêu cầu ngữ nghĩa**

🏵️ Dự án 2: Trình quản lý thư mục phân cấp và tìm kiếm theo mức

⭐⭐⭐ **Mức độ: Khá – Nâng cao**

📌 **Chủ đề: Level-order, duyệt cây, tìm kiếm theo mức, serialization**

🎯 Mục tiêu

Xây dựng một trình quản lý thư mục mô phỏng **cấu trúc cây phân cấp** (giống cây thư mục Windows), với các tính năng:

1. Nhập dữ liệu cây từ mảng hoặc JSON
2. Duyệt thư mục theo mức (level-order)
3. Tìm kiếm tên thư mục theo chiều rộng
4. Ghi lại cấu trúc cây bằng serialization
5. Hiển thị cây bằng sơ đồ text hoặc ASCII tree

📄 Yêu cầu chức năng

- **Đầu vào**: Dữ liệu cây dạng danh sách cha–con hoặc JSON

Ví dụ:

```
{
  "root": {
    "Documents": {
      "Work": {},
      "Personal": {}
    },
    "Downloads": {
      "Images": {},
      "Videos": {}
    }
  }
}
```

- **Chức năng**:
  - Duyệt toàn bộ thư mục theo từng mức
  - Tìm kiếm thư mục "Personal" → trả về đường dẫn: /root/Documents/Personal
  - Ghi cấu trúc cây ra file (serialization)
  - Vẽ cây ra dạng text như:

```
root
├── Documents
│   ├── Work
│   └── Personal
└── Downloads
```

```
├── Images
└── Videos
```

🧠 Phân tích nghiệp vụ

- Cấu trúc dữ liệu nên là **Node(name, children[])**
- Duyệt theo mức → dùng **queue (Level-order traversal)**
- Tìm kiếm → BFS đến khi gặp tên cần
- Serialization → duyệt Preorder hoặc Level-order, in ra theo format

💡 Hướng dẫn kỹ thuật

1. Xây dựng class DirectoryNode có:
    - name: str
    - children: List[DirectoryNode]
2. Hàm duyệt cây theo mức:
    - Queue, in từng mức, cách đều bằng khoảng trắng
3. Hàm tìm kiếm:
    - BFS trả về đường dẫn từ root
4. Ghi cấu trúc:
    - Duyệt Preorder, dùng indent để mô phỏng cấp bậc
5. (Tùy chọn): Xuất cây ra dạng .dot file để vẽ bằng Graphviz

✅ Kết quả kỳ vọng

Sinh viên có thể:

- Dùng cây để mô hình hóa cấu trúc thực tế (filesystem, XML, tổ chức công ty,...)
- Duyệt và thao tác cây bằng các thuật toán đã học
- Rèn luyện kỹ năng xử lý dữ liệu phân cấp và serialization