Tuần 3 - Tổng hợp kiến thức Buổi học số 4

Time-Series Team

Ngày 24 tháng 6 năm 2025

Buổi học số 3 (Thứ 5, 19/06/2025) bao gồm 6 nội dung chính:

- ***Phần 1: Giới thiệu về cấu trúc dữ liệu Cây***
- ***Phần 2: Các thuật toán trên cây***
- ***Phần 3: Stack***
- ***Phần 4: Queue***
- ***Phần 5: Cây nhị phân***
- ***Phần 6: Cây tìm kiếm nhị phân (BST)***
- ***Phần 7: Mở rộng: Giới thiệu về KD-Tree***

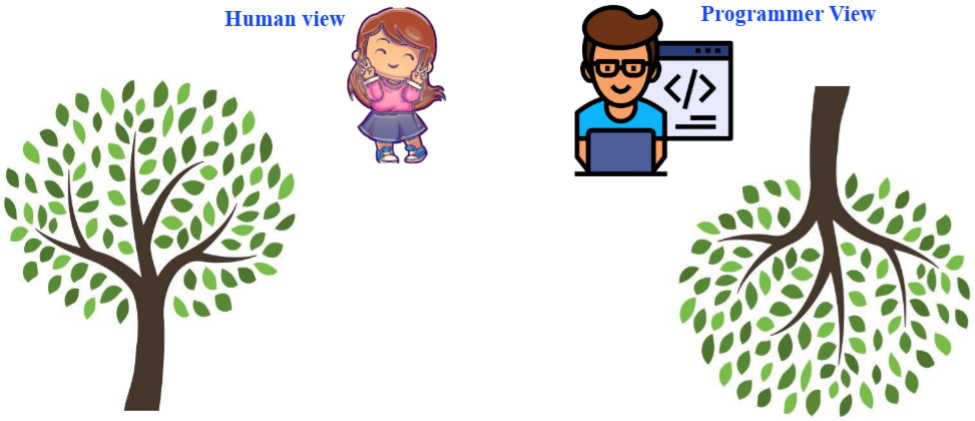
Phần 1: Giới thiệu về cấu trúc dữ liệu Cây

1. Cây là gì?

- Cây (Tree) là một **cấu trúc dữ liệu phi tuyến tính** (non-linear data structure), dùng để **lưu trữ dữ liệu theo kiểu phân cấp (hierarchical)**, không giống như mảng hay danh sách liên kết vốn lưu dữ liệu theo thứ tự tuần tự.
- Mỗi cây gồm nhiều **nút (node)** được nối với nhau bằng các **cạnh (edge)**, tạo thành các tầng lớp (mức) như sơ đồ phả hệ hoặc thư mục máy tính.

Góc nhìn trực quan:

- Với con người, cây thường mọc từ **gốc dưới lên**.
- Trong lập trình, cây được vẽ từ **gốc ở trên**, các nhánh **đi xuống**



2. Vì sao cần dùng cây?

Các cấu trúc dữ liệu tuyến tính như **mảng (array)** hoặc **danh sách liên kết (linked list)** thường mất **O(n)** thời gian để tìm kiếm phần tử, vì phải duyệt qua từng phần tử một.
Ví dụ:

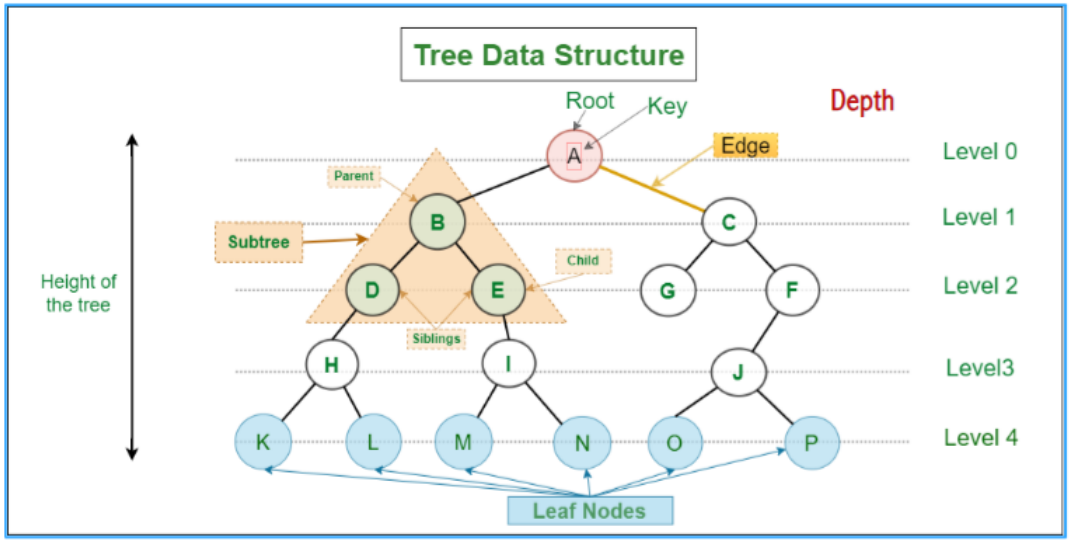
```
List: [5, 4, 7, 1, 3, 6, 8]
Searching for 8 → must traverse almost the entire list
```

Trong khi đó, nếu tổ chức dữ liệu theo **cây nhị phân tìm kiếm (BST)**:

```
        5
       / \
      4   7
     /   / \
    1   6   8
```

→ Việc tìm kiếm `8` sẽ nhanh hơn rất nhiều (**O(log n)** nếu cây cân bằng), vì ta loại bỏ được nửa không gian tìm kiếm sau mỗi bước.

3. Thuật ngữ cơ bản trong cây *(Basic Terminologies)*



Thuật ngữ	Giải thích đơn giản
Root Node	Nút gốc – nơi bắt đầu của cây
Parent Node	Nút cha – nút có nút con
Child Node	Nút con – được kết nối từ một nút cha
Leaf Node	Nút lá – không có nút con nào
Sibling (Anh em)	Các node cùng một cha
Ancestor	Toàn bộ tổ tiên của một node, tính từ gốc
Level / Depth	Số cạnh từ gốc đến node (gốc có depth = 0)
Subtree	Một node cùng toàn bộ cây con bên dưới nó
Height	Số cạnh dài nhất từ node đó đến lá

Ví dụ minh họa:

```
        A
      /   \
    B       C
   / \       \
  D   E       F
```

Phân tích	**Kết quả**
Root	A
Leaf	D, E, F
Siblings	D và E
Ancestors of E	B, A
Depth of E	2
Height of B	1

4. Cách cài đặt cây trong Python

Trong Python, không có cấu trúc cây sẵn như `list`, ta phải tự định nghĩa lớp `TreeNode` để tạo node.

Cây tổng quát (mỗi node có nhiều con):

```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.children = []

# Vi du:
a = TreeNode('A')
b = TreeNode('B')
c = TreeNode('C')
a.children = [b, c]
```

Cây nhị phân (Binary Tree – mỗi node có tối đa 2 con):

```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

# Vi du:
a = TreeNode('A')
b = TreeNode('B')
c = TreeNode('C')
a.left = b
a.right = c
```

Nếu muốn tuân thủ nguyên lý lập trình hướng đối tượng (OOP) đầy đủ, bạn có thể tách ra thêm class `Tree` để quản lý root, hỗ trợ thêm các thao tác như `insert()`, `delete()`, `traverse()`...

5. Ứng dụng

Cây là một trong những cấu trúc dữ liệu **phổ biến và mạnh mẽ nhất** trong lập trình, với rất nhiều ứng dụng thực tế:

Lĩnh vực	Ứng dụng
File Explorer	Cấu trúc thư mục: thư mục cha – thư mục con – tệp
Cơ sở dữ liệu (Database)	Cây B-Tree / B+ Tree giúp truy vấn hiệu quả
HTML DOM (Web)	HTML được tổ chức dưới dạng cây DOM (cha – con)
Machine Learning	Cây quyết định (Decision Tree) để phân loại hoặc dự đoán
AI & Robotics	Cây K-D (K-Dimensional Tree) để tìm kiếm gần nhất trong không gian nhiều chiều

6. Tổng kết

- **Cây là một cấu trúc dữ liệu phân cấp**, gồm các node có thể có con cháu.
- **Khác với list/array**, cây cho phép truy xuất hiệu quả hơn trong các trường hợp tìm kiếm hoặc tổ chức dữ liệu dạng phân nhánh.
- Cây có mặt **khắp nơi** trong khoa học máy tính: từ tệp hệ thống, CSDL, web, đến AI.

Phần 2: Các thuật toán trên cây

1. Duyệt cây là gì?

- Duyệt cây (traversal) là quá trình **thăm từng nút trong cây đúng một lần** theo một thứ tự xác định. Đây là kỹ thuật nền tảng trong mọi bài toán liên quan đến cây.
- Có hai nhóm thuật toán duyệt cây chính:
  - **DFS – Depth First Search** (Duyệt theo chiều sâu)
  - **BFS – Breadth First Search** (Duyệt theo chiều rộng)

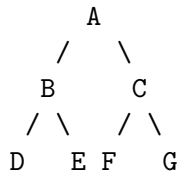
2. DFS – Duyệt theo chiều sâu

Ý tưởng chính

DFS đi **sâu vào từng nhánh** của cây trước, đến khi không thể đi tiếp thì **quay lui (backtrack)** và chuyển sang nhánh khác.

Ví dụ minh họa:

Cho cây:



- **Preorder (Gốc → Trái → Phải)**: A → B → D → E → C → F → G
- **Inorder (Trái → Gốc → Phải)**: D → B → E → A → F → C → G
- **Postorder (Trái → Phải → Gốc)**: D → E → B → F → G → C → A

Ghi chú

DFS thường được dùng trong bài toán đệ quy, backtracking, biểu thức toán học dạng cây, phân tích cây thư mục,...

—

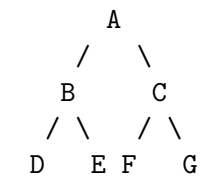
3. BFS – Duyệt theo chiều rộng

Ý tưởng chính

BFS duyệt cây **từng tầng**, từ gốc đến tất cả các node con ở tầng kế tiếp, sau đó mới xuống các tầng sâu hơn.

Ví dụ minh họa:

Với cùng cây trên:



BFS sẽ duyệt theo thứ tự: **A → B → C → D → E → F → G**

Ví dụ thực tế

Tưởng tượng một công ty:

- A là giám đốc,
- B, C là trưởng phòng,
- D, E, F, G là nhân viên cấp dưới.
  BFS sẽ đi qua từng cấp quản lý trước khi xuống cấp nhân viên.

—

4. So sánh DFS và BFS

Tiêu chí	**DFS**	**BFS**
Cách duyệt	Đi sâu từng nhánh	Duyệt theo tầng
Cấu trúc dùng	Stack / Đệ quy	Queue
Ưu điểm	Phân tích cấu trúc sâu, tiết kiệm bộ nhớ	Tìm đường ngắn nhất, xử lý theo mức
Nhược điểm	Có thể rơi vào vòng lặp, backtrack phức tạp	Dễ tốn bộ nhớ nếu cây rộng
Ứng dụng	Duyệt biểu thức, phân tích cây thư mục	Cây phân cấp, tìm kiếm theo khoảng cách

—

5. Tổng kết

- DFS và BFS là hai kỹ thuật duyệt cây phổ biến và quan trọng nhất.
- DFS đi theo nhánh – phù hợp cho các bài toán cần kiểm tra toàn bộ cây con.
- BFS đi theo tầng – cực kỳ hữu ích trong các bài toán tìm đường đi ngắn nhất hoặc phân tích theo mức.

Phần 3: Ngăn xếp (Stack)

1. Stack là gì?

Stack (ngăn xếp) là một cấu trúc dữ liệu tuyến tính tuân theo nguyên lý:

LIFO – Last In, First Out (Vào sau → ra trước)

Ví dụ hình dung

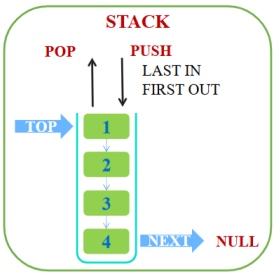
Stack giống như một **chồng đĩa**: bạn luôn lấy ra cái **trên cùng trước**, không thể lấy ở giữa hay đáy.

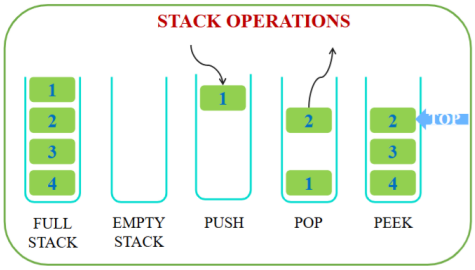
2. Các thao tác cơ bản trên Stack

Thao tác	Mô tả
`push(x)`	Đẩy phần tử `x` lên đỉnh stack
`pop()`	Lấy phần tử trên cùng ra khỏi stack
`peek()`	Xem phần tử đỉnh mà không loại bỏ
`isEmpty()`	Kiểm tra xem stack có rỗng không

Ví dụ quá trình thực hiện Stack:

```
Bắt đầu:   []
push(12):  [12]
push(8):   [12, 8]
push(21):  [12, 8, 21]
pop():     [12, 8]
```





3. Cài đặt Stack bằng List trong Python

```python
stack = []
stack.append(4)    # push
stack.pop()        # pop
```

Ghi chú

Python dùng `append()` để push và `pop()` để lấy phần tử cuối – tương đương ngăn xếp LIFO.

4. Ứng dụng Stack trong DFS (Depth-First Search)

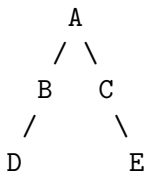
DFS là thuật toán duyệt cây hoặc đồ thị theo **chiều sâu**. Stack là cấu trúc lý tưởng để cài DFS.

Tư duy

- Bắt đầu từ node gốc, push vào stack.
- Trong khi stack chưa rỗng:
  - Pop đỉnh stack → đánh dấu đã thăm.
  - Push các node kề (theo thứ tự ngược nếu muốn duyệt trái trước).

5. Ví dụ

Cho cây:



Quá trình thực hiện:

Bước	Stack hiện tại	Đã thăm
1	[A]	
2	[C, B]	A
3	[C, D]	B
4	[C]	D
5	[]	C
6	[E]	
7	[]	E

Phân tích từng bước

- Bắt đầu từ `A`, push vào stack → stack = [A]
- Pop A, thăm A → push C, B (C sau để B được duyệt trước) → stack = [C, B]
- Pop B, thăm B → push D → stack = [C, D]
- Pop D, thăm D → không còn con → stack = [C]
- Pop C, thăm C → push E → stack = [E]
- Pop E, thăm E → hoàn tất

Thứ tự: A → B → D → C → E

6. Tổng kết

- **Stack** là cấu trúc dữ liệu hoạt động theo LIFO.
- `push`, `pop`, `peek` là thao tác cơ bản.
- Stack dùng để **cài đặt DFS** khi không dùng đệ quy.
- Ứng dụng trong toán học, duyệt cây, và hệ thống phần mềm.

Phần 4: Hàng đợi (Queue)

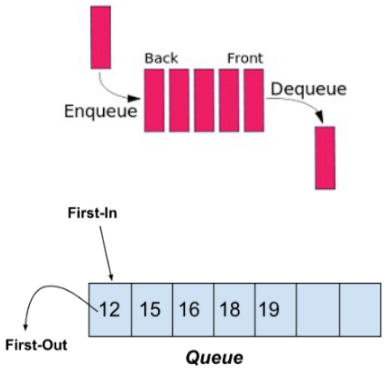
1. Queue là gì?

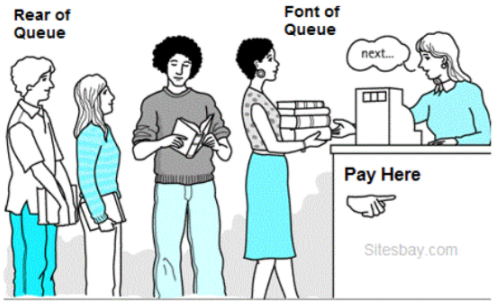
Queue (hàng đợi) là một cấu trúc dữ liệu tuyến tính, nơi các phần tử được xử lý theo nguyên tắc:

FIFO – First In, First Out (Vào trước → ra trước)

Ví dụ hình dung

Queue giống như một hàng xếp vé xem phim: người đến trước sẽ được phục vụ trước. Bạn không thể chen ngang vào giữa.





Real Life Example of Queue : Library Counter

2. Các thao tác cơ bản trên Queue

Thao tác	Mô tả
`enqueue(x)`	Thêm phần tử `x` vào cuối hàng đợi
`dequeue()`	Loại bỏ phần tử ở đầu hàng đợi
`peek()`	Xem phần tử đầu tiên mà không loại bỏ
`isEmpty()`	Kiểm tra xem hàng đợi có rỗng không

Ví dụ quá trình thực hiện Queue:

```
Bắt đầu:   []
enqueue(A): [A]
enqueue(B): [A, B]
enqueue(C): [A, B, C]
dequeue():  [B, C]
```

3. Cài đặt Queue bằng List trong Python

```python
queue = []
queue.append("A")  # enqueue
queue.pop(0)       # dequeue
```

Ghi chú

Có thể dùng `append()` để enqueue (thêm cuối), và `pop(0)` để dequeue (loại bỏ đầu).

4. Ứng dụng Queue trong BFS (Breadth-First Search)

BFS là thuật toán duyệt cây hoặc đồ thị theo **chiều rộng**. Queue rất phù hợp để triển khai BFS vì:

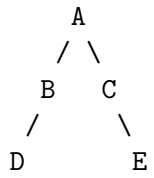
- BFS khám phá các node gần gốc trước, sau đó mới đến các node sâu hơn.
- Queue đảm bảo các node được thăm theo đúng thứ tự "gần trước, xa sau".

Tư duy BFS bằng Queue

- Bắt đầu từ node gốc, enqueue vào queue.
- Trong khi queue chưa rỗng:
  - Dequeue node đầu → đánh dấu đã thăm.
  - Enqueue các node kề (con của node đó).

5. Ví dụ minh họa BFS bằng Queue

Cho cây:



Quá trình thực hiện:

Bước	Queue hiện tại	Đã thăm
1	[A]	
2	[B, C]	A
3	[C, D]	B
4	[D, E]	C
5	[E]	D
6	[]	E

Phân tích từng bước

- Bắt đầu từ `A`, enqueue → queue = [A]
- Dequeue A → thăm A → enqueue B, C → queue = [B, C]
- Dequeue B → thăm B → enqueue D → queue = [C, D]
- Dequeue C → thăm C → enqueue E → queue = [D, E]
- Dequeue D → thăm D → không còn con → queue = [E]
- Dequeue E → thăm E → kết thúc

Thứ tự BFS: A → B → C → D → E

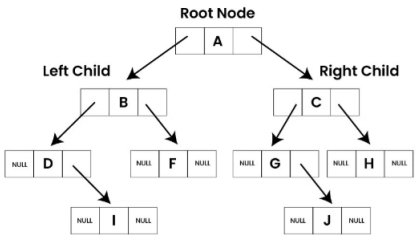
6. Tổng kết

- **Queue** là cấu trúc dữ liệu hoạt động theo **FIFO**.
- `enqueue`, `dequeue`, `peek` là các thao tác cơ bản.
- **BFS** sử dụng queue để duyệt cây/đồ thị theo chiều rộng.
- Dễ áp dụng vào bài toán ngắn nhất, tìm kiếm theo cấp độ, crawling, AI,...

Phần 5: Cây nhị phân (Binary Tree)

PHẦN 5: CÂY NHỊ PHÂN LÀ GÌ?

1. Định nghĩa Cây nhị phân



Cây nhị phân (Binary Tree) là một cấu trúc dữ liệu dạng cây, trong đó **mỗi nút có tối đa 2 nút con**, gọi là:

- **Nút con trái (left child)**
- **Nút con phải (right child)**

Ví dụ minh họa

Cấu trúc cây nhị phân cho phép tổ chức dữ liệu phân cấp theo 2 nhánh. Ví dụ:

```
    A
   / \
  B   C
```

Nút A có 2 con: B (trái), C (phải).

2. Các dạng đặc biệt của cây nhị phân

- **Left-skewed Tree:** Chỉ có nhánh trái, ví dụ:

```
    A
   /
  B
 /
C
```

- **Right-skewed Tree:** Chỉ có nhánh phải, ví dụ:

```
A
 \
  B
   \
    C
```

- **Full Binary Tree:** Mỗi nút có 0 hoặc 2 con.
- **Balanced Binary Tree:** Chênh lệch chiều cao giữa cây con trái và phải tại mọi nút không vượt quá 1.

3. Chiều cao và cân bằng cây nhị phân

Chiều cao (height) của cây

Chiều cao của một cây là độ dài đường đi dài nhất từ nút đó đến lá.

$$\text{Cây chỉ có 1 nút (gốc)} \Rightarrow \text{Height} = 0$$

Cây nhị phân cân bằng (Balanced Tree)

Một cây nhị phân được xem là cân bằng khi:

- Độ chênh lệch chiều cao giữa cây con trái và phải không vượt quá 1.
- Cây con trái và phải đều là cây cân bằng.

Tại sao cần cân bằng?

Nếu cây không cân bằng (quá lệch trái/phải), các thao tác như tìm kiếm, chèn, xóa sẽ chậm giống như danh sách liên kết.
Cây cân bằng đảm bảo hiệu suất tốt hơn: độ sâu nhỏ hơn → ít bước duyệt hơn.

4. Các thao tác cơ bản trên cây nhị phân

Chèn nút (Insert a node)

- Duyệt theo thứ tự **Level-Order** (BFS) để tìm nút đầu tiên thiếu con trái hoặc phải.
- Thêm nút mới vào vị trí đó.

Thuật toán tổng quát

1. Tạo nút mới.
2. Nếu cây rỗng → đặt nút mới là gốc.
3. Sử dụng `queue` duyệt từ trên xuống:
   - Nếu gặp nút thiếu trái → gán trái = node mới.
   - Nếu thiếu phải → gán phải = node mới.

Xóa nút (Delete a node)

- Nếu là lá → xóa luôn.
- Nếu không → thay thế bằng **nút sâu nhất bên phải**, rồi xóa nút đó.

Thuật toán:

1. Tìm nút cần xóa.
2. Tìm nút sâu nhất bên phải.
3. Gán giá trị nút sâu nhất cho nút cần xóa.
4. Xóa nút sâu nhất bên phải.

5. Hạn chế của cây nhị phân cơ bản

Vấn đề khi tìm kiếm

Cây nhị phân thường không đảm bảo hướng đi (trái/phải) khi tìm kiếm → mất thời gian duyệt toàn bộ cây.
Ví dụ: tìm số 9 trong cây không sắp xếp, ta không biết đi trái hay phải trước.

Điều này dẫn đến sự ra đời của các cây tối ưu hơn như:

- **Binary Search Tree (BST)**
- **AVL Tree, Red-Black Tree**

6. Tổng kết

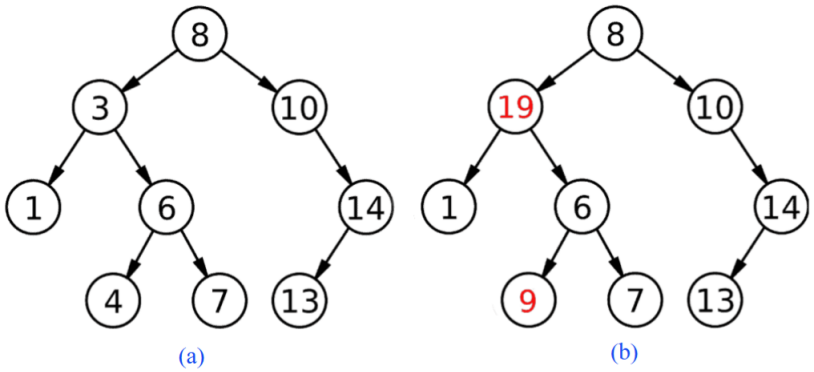
- Cây nhị phân là cây mà mỗi nút có tối đa 2 con.
- Có nhiều loại: lệch trái, lệch phải, đầy đủ, cân bằng.
- Các thao tác chính: chèn, xóa, tìm kiếm.
- Cây cần cân bằng để đảm bảo hiệu suất.
- Nhược điểm: khó định hướng tìm kiếm nếu không sắp xếp.

Phần 6: Cây tìm kiếm nhị phân (BST)

Trong phần trước, ta đã tìm hiểu về **Binary Tree** — một cấu trúc cây mà mỗi node có tối đa 2 con. Tuy nhiên, Binary Tree không áp đặt quy luật nào cho việc sắp xếp node trái/phải. Điều này gây khó khăn trong việc tìm kiếm dữ liệu.

Vấn đề đặt ra

Giả sử bạn cần tìm số 9 trong cây nhị phân thường. Bạn không biết nên đi trái hay phải vì cây không theo thứ tự nào cả.



Đây chính là lúc ta cần đến một cấu trúc có tổ chức hơn: **Binary Search Tree**.

2. Binary Search Tree (BST) là gì?

Binary Search Tree (Cây nhị phân tìm kiếm) là một dạng đặc biệt của Binary Tree, tuân thủ nguyên tắc:

Quy tắc BST

Với mỗi node:

- Tất cả giá trị ở cây con bên trái **nhỏ hơn** node hiện tại.
- Tất cả giá trị ở cây con bên phải **lớn hơn** node hiện tại.

Quy tắc này cho phép ta tìm kiếm, chèn, xoá... rất hiệu quả — tương tự như tìm trong danh bạ đã sắp xếp.

3. Ví dụ

Cây BST sau chứa các số: 5, 3, 7, 1, 4, 6, 8

```
      5
     / \
    3   7
   / \ / \
  1  4 6  8
```

- Nhánh trái (3, 1, 4) đều nhỏ hơn 5.
- Nhánh phải (7, 6, 8) đều lớn hơn 5.
- Mỗi node con cũng là một BST thu nhỏ.

—

4. Ứng dụng và ưu điểm của BST

- **Tìm kiếm (Search)** nhanh: mỗi lần đi xuống 1 tầng, không gian tìm kiếm giảm một nửa.
- **Chèn node (Insert)** nhanh, nhờ quy tắc so sánh trái/phải.
- **Xoá node (Delete)** linh hoạt, nhưng cần cẩn thận để duy trì quy tắc BST.
- **Duyệt theo thứ tự tăng dần** rất dễ: chỉ cần **In-order Traversal**.

Thời gian trung bình (Average Case)

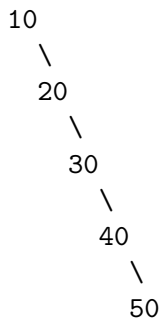
- Tìm kiếm / Chèn / Xoá: $O(\log n)$

5. Lưu ý quan trọng: Cây BST có thể bị mất cân bằng

BST được thiết kế để tìm kiếm nhanh — trung bình chỉ mất $O(\log n)$. Tuy nhiên, hiệu suất này **chỉ đúng khi cây cân bằng tương đối**. Nếu cây bị lệch nghiêm trọng về một phía, thời gian truy cập có thể tăng lên $O(n)$.

5.1. Ví dụ minh hoạ — Trường hợp lệch phải

Giả sử bạn chèn lần lượt dãy số `10, 20, 30, 40, 50` vào BST:



Đây là cây BST hợp lệ (trái $<$ node $<$ phải), nhưng do toàn bộ phần tử đều tăng dần nên cây bị **lệch phải hoàn toàn**, giống như danh sách liên kết. Khi đó:

- Muốn tìm phần tử cuối (50), ta phải duyệt qua tất cả node từ gốc $\rightarrow$ tốn $O(n)$.
- Mất lợi thế chia đôi không gian tìm kiếm như BST lý tưởng.

5.2. Cây càng mất cân bằng, hiệu suất càng giảm

Ngoài lệch phải, cây cũng có thể bị lệch trái nếu dãy số chèn vào giảm dần:

```
    50
   /
  40
 /
30
...
```

Kết quả vẫn tương tự: hiệu suất tìm kiếm, chèn, xoá đều bị ảnh hưởng nặng nề.

BST không tự động cân bằng. Nếu chèn dữ liệu theo thứ tự tăng hoặc giảm liên tục, hiệu suất sẽ tệ như danh sách liên kết — từ $O(\log n)$ xuống còn $O(n)$.

5.3. Giải pháp: Dùng các BST tự cân bằng

Để khắc phục điểm yếu này, người ta phát triển các cấu trúc cây tự cân bằng:

- **AVL Tree** — luôn giữ cân bằng tuyệt đối tại mọi node.
- **Red-Black Tree** — cho phép lệch nhẹ nhưng đảm bảo chiều cao giới hạn.

BST cơ bản dễ hiểu, nhưng để đảm bảo hiệu suất trong thực tế, ta thường dùng AVL hoặc Red-Black Tree.

6. Tổng kết

- **Binary Search Tree (BST)** là một dạng đặc biệt của cây nhị phân, nơi:

$$\text{Trái} < \text{node} < \text{Phải}$$

- Nhờ quy luật sắp xếp, BST hỗ trợ hiệu quả các thao tác:
  - **Tìm kiếm**, **chèn**, **xóa**: trong trường hợp lý tưởng chỉ mất $O(\log n)$
  - **Duyệt theo thứ tự tăng dần**: dùng **In-order Traversal**
- Tuy nhiên, BST không tự cân bằng. Nếu dữ liệu vào không ngẫu nhiên (ví dụ tăng dần), cây có thể bị lệch như danh sách liên kết → hiệu suất giảm còn $O(n)$.
- Để duy trì hiệu năng cao, người ta thường sử dụng các biến thể **BST tự cân bằng** như:
  - AVL Tree
  - Red-Black Tree

Phần 7: Mở rộng: Giới thiệu về KD-Tree

1. Đặt vấn đề

Trong phần trước, ta đã tìm hiểu về **Binary Search Tree (BST)** — một cấu trúc cây nhị phân dùng để lưu trữ dữ liệu 1 chiều (1D), như các số nguyên. Mỗi node chia không gian 1 chiều thành 2 phần: nhỏ hơn sang trái, lớn hơn sang phải.
Nhưng nếu dữ liệu có nhiều chiều hơn — ví dụ: mỗi điểm có tọa độ (x, y) hoặc (x, y, z)? Ta không thể áp dụng trực tiếp BST truyền thống. Lúc này, ta cần một cấu trúc tổng quát hơn: **k-d Tree (k-dimensional Tree)**.

Ví dụ

Giả sử bạn có danh sách toạ độ các điểm trong không gian 2D, ví dụ:

$$(7, 2),\ (5, 4),\ (9, 6),\ (2, 3),\ (4, 7),\ (8, 1)$$

Bạn muốn tìm kiếm nhanh điểm gần nhất, hoặc tìm tất cả các điểm nằm trong 1 hình chữ nhật. Duyệt tuyến tính sẽ rất chậm.

2. KD-Tree là gì?

k-d Tree là một cây nhị phân tổng quát của BST, dùng để lưu trữ và truy vấn hiệu quả trong không gian $\mathbb{R}^k$ (với k là số chiều).

Nguyên tắc hoạt động của KD-Tree (2D)

- Mỗi node đại diện cho một điểm trong không gian.
- Mỗi mức của cây phân chia theo một chiều **luân phiên**.
  - Mức 0: chia theo trục x
  - Mức 1: chia theo trục y
  - Mức 2: lại chia theo trục x, ...
- Điểm nhỏ hơn → nhánh trái, điểm lớn hơn → nhánh phải (dựa theo chiều đang xét).

3. Xây dựng KD-Tree từng bước (2D)

Giả sử ta có tập điểm:

$$\{(7, 2), (5, 4), (9, 6), (2, 3), (4, 7), (8, 1)\}$$

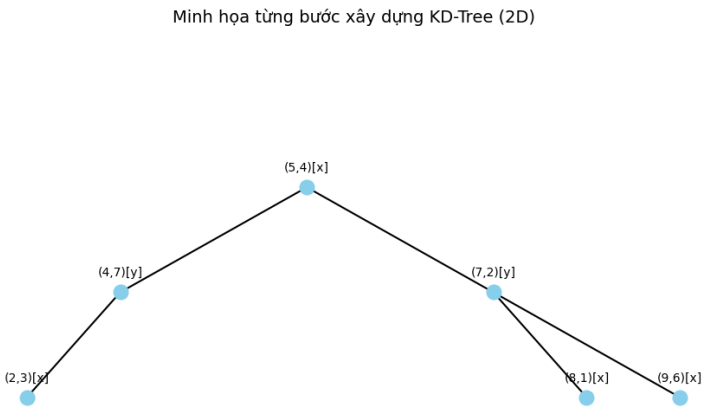
Các bước xây:

1. Bước 1 (Mức 0 - chia theo x):
   - Sắp xếp các điểm theo x: $(2, 3), (4, 7), (5, 4), (7, 2), (8, 1), (9, 6)$
   - Chọn trung vị: $(5, 4)$ làm root

2. Bước 2 (Mức 1 - chia theo y):
   - Trái: $(2,3),(4,7) \rightarrow$ sắp theo y: $(2,3),(4,7)$, chọn trung vị: $(4,7)$
   - Phải: $(7,2),(8,1),(9,6) \rightarrow$ sắp theo y: $(8,1),(7,2),(9,6)$, chọn trung vị: $(7,2)$
3. Bước 3 (Mức 2 - chia lại theo x):
   - Trái của (4,7): $(2,3)$
   - Trái của (7,2): $(8,1)$; phải của (7,2): $(9,6)$

Sơ đồ KD-Tree:

```
          (5,4) [x]
         /         \
    (4,7)[y]       (7,2)[y]
     /               /     \
(2,3)[x]       (8,1)[x]   (9,6)[x]
```



Hình 1: Minh họa từng bước xây dựng KD-Tree (2D)

4. Ứng dụng của KD-Tree

- **Nearest Neighbor Search (Tìm điểm gần nhất)**: rất phổ biến trong Machine Learning (ví dụ: KNN).
- **Range Search (Tìm các điểm trong vùng)**: xác định tất cả điểm trong hình chữ nhật / hình cầu.
- **Computer Graphics và Robotics**: dùng để phát hiện va chạm, ánh xạ không gian.
- **Image Retrieval**: tìm ảnh tương tự trong không gian đặc trưng.

5. Hiệu suất và Phân tích

- Mỗi lần đi xuống 1 mức, không gian bị chia đôi → hiệu quả tương tự **BST cân bằng**.
- Với dữ liệu tốt: **Tìm kiếm / Chèn / Gần nhất**: $O(\log n)$
- Trường hợp xấu (cây lệch): hiệu suất $O(n)$

So sánh BST vs KD-Tree

Tiêu chí	**BST**	**KD-Tree (2D)**
Số chiều dữ liệu	1	k $\geq$ 2
Phân chia không gian	Theo giá trị	Theo tọa độ luân phiên
Ứng dụng	Số học, danh sách	Không gian, ML, Robotics

7. Tìm kiếm điểm gần nhất (Nearest Neighbor Search)

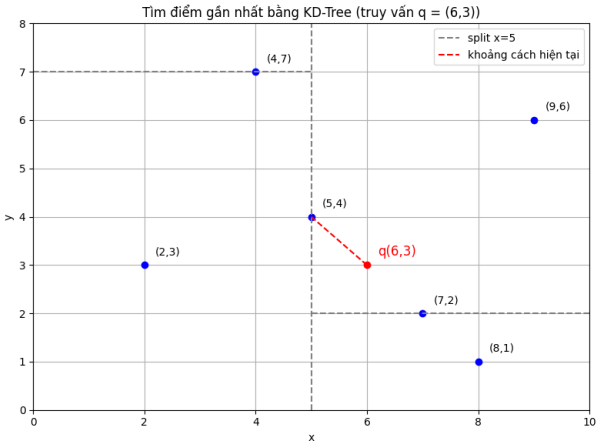
Một trong những ứng dụng quan trọng nhất của KD-Tree là **tìm điểm gần nhất** với một điểm cho trước trong không gian $\mathbb{R}^k$. Thuật toán này được tối ưu nhờ khả năng **cắt bỏ nhánh không cần thiết**.

Ý tưởng chính

- Bắt đầu từ root, ta duyệt cây như BST: so sánh theo chiều luân phiên.
- Tại mỗi node:
  - Cập nhật khoảng cách nhỏ nhất.
  - Nếu hình chiếu theo chiều phân chia gần hơn khoảng cách hiện tại → cần xét nhánh còn lại.
  - Nếu không → cắt nhánh đó (prune).
- Duyệt theo chiều sâu, nhưng có thể quay lại nhánh không đi qua nếu có khả năng có điểm gần hơn.

Ví dụ:

- Truy vấn $q = (6, 3)$
- Duyệt theo KD-Tree, tìm thấy ứng viên gần nhất hiện tại: $(5, 4)$
- Kiểm tra các node khác (như $(7, 2), (8, 1)$) chỉ khi hình chiếu qua mặt chia gần hơn khoảng cách hiện tại.
- Các node như $(2, 3), (4, 7)$ bị cắt vì không thể chứa điểm gần hơn.



Hình 2: Tìm điểm gần nhất cho điểm truy vấn $q = (6, 3)$

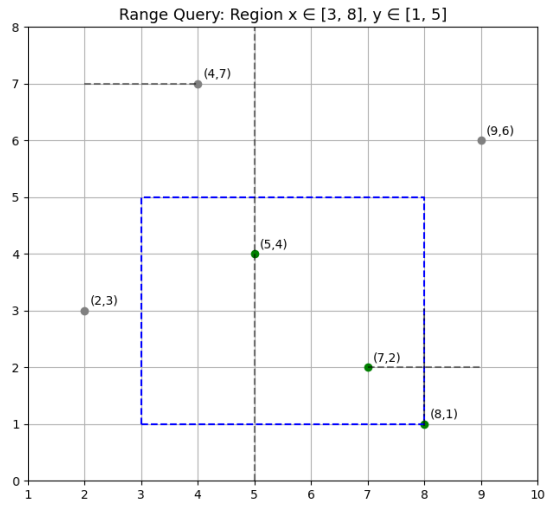
8. Duyệt theo vùng (Range Query)

Một ứng dụng khác là tìm tất cả các điểm nằm trong một **vùng cụ thể** (thường là hình chữ nhật hoặc hình cầu).

Ý tưởng

- Bắt đầu từ root, duyệt từng node và kiểm tra:
  - Nếu node nằm trong vùng → thêm vào kết quả.
  - Nếu vùng cắt qua mặt phân chia → xét cả hai nhánh.
  - Nếu không → chỉ xét nhánh bên trong vùng.

Ví dụ: Tìm các điểm trong hình chữ nhật $x \in [3, 8], y \in [1, 5]$



Hình 3: Range Query: hình chữ nhật màu xanh cắt không gian

Các điểm phù hợp: $(5, 4), (7, 2), (8, 1)$
Tối ưu: KD-Tree giúp tránh duyệt các node như $(2, 3), (9, 6)$ — nằm ngoài vùng quan tâm.

9. Mở rộng lên không gian nhiều chiều (k > 2)

KD-Tree tổng quát cho k-chiều, không giới hạn 2D.

Chiến lược chia chiều

Chia theo chiều luân phiên: tại mức d, chia theo chiều $d \bmod k$.

Ví dụ: Với 3D:

- Mức 0: chia theo x
- Mức 1: chia theo y
- Mức 2: chia theo z
- Mức 3: chia lại theo x, v.v.

Khó khăn:

- Càng nhiều chiều → cây càng khó cân bằng.
- Hiện tượng **Curse of Dimensionality**: dữ liệu thưa dần trong không gian cao → hiệu năng KD-Tree giảm rõ rệt.
- Trong thực tế: KD-Tree hiệu quả nhất với $k \leq 10$

Giải pháp thay thế:

- Dữ liệu cao chiều → dùng Ball Tree, VP Tree hoặc Approximate Nearest Neighbor.

10. Tổng kết

- **KD-Tree** là sự mở rộng tự nhiên của BST cho dữ liệu nhiều chiều ($k \geq 2$), đặc biệt mạnh trong bài toán truy vấn không gian.
- Mỗi tầng của cây luân phiên chia không gian theo trục khác nhau (x, y, z,...), giúp tổ chức dữ liệu hiệu quả trong không gian $\mathbb{R}^k$.
- **Tìm điểm gần nhất (Nearest Neighbor)**: KD-Tree loại trừ nhánh thông minh bằng cách kiểm tra khoảng cách đến mặt phân chia — giảm mạnh số lượng node cần xét.
- **Duyệt theo vùng (Range Query)**: Truy vấn tìm tất cả điểm trong một vùng hình chữ nhật/hình cầu trở nên cực kỳ nhanh nhờ khả năng cắt bỏ không gian ngoài vùng.
- **Cân bằng cây là điều cốt lõi**: Nếu không cân bằng, hiệu năng có thể rơi về $O(n)$. Do đó, cần chọn trung vị khi xây dựng để đảm bảo chiều sâu tối ưu.
- **Khả năng mở rộng chiều cao hơn (k lớn)** gặp hạn chế do "Lời nguyền chiều cao" (Curse of Dimensionality) — làm giảm hiệu quả tách không gian.
- KD-Tree rất phù hợp với các bài toán trong **Machine Learning, Robotics, Graphics, Vision**, nhưng với dữ liệu cao chiều hoặc phân bố phức tạp, nên cân nhắc các cấu trúc như *Ball Tree, Cover Tree, R-Tree.*