

Ứng dụng của các Cấu trúc Dữ liệu

Singly Linked List

a. Ứng dụng thực tế:

- Một số ví dụ:
- **Danh sách phát nhạc (Music Playlist)**
- **Quản lý tiến trình trong Hệ điều hành (Task Scheduler)**

b. Giải thích:

1. **Danh sách phát nhạc:** Cho phép thêm/xóa bài hát linh hoạt và phát theo thứ tự. Đây là nền tảng cho các ứng dụng nghe nhạc.
2. **Task Scheduler:** Giúp xử lý các tiến trình theo vòng (round-robin), đảm bảo công bằng và hiệu quả.

c. Cách dùng:

Trong ứng dụng danh sách phát nhạc, cấu trúc dữ liệu **Singly Linked List** được sử dụng để quản lý danh sách các bài hát. Mỗi bài hát được lưu trữ trong một nút (**SongNode**), và các nút được liên kết với nhau thông qua con trỏ **next**. Điều này cho phép:

- **Thêm bài hát mới:** Khi người dùng thêm bài hát, nút mới được thêm vào cuối danh sách. Điều này đảm bảo thứ tự phát nhạc được duy trì.
- **Xóa bài hát:** Dễ dàng loại bỏ một bài hát bằng cách cập nhật con trỏ **next** của nút trước đó.
- **Duyệt danh sách:** Có thể duyệt qua danh sách để phát từng bài hát theo thứ tự. Cách tiếp cận này phù hợp với các ứng dụng nghe nhạc vì nó tối ưu hóa việc thêm/xóa bài hát mà không cần phải dịch chuyển toàn bộ danh sách như trong mảng.

```
class SongNode:  
    def __init__(self, song):  
        self.song = song  
        self.next = None
```

```

class Playlist:
    def __init__(self):
        self.head = None

    def add_song(self, song):
        new_node = SongNode(song)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

```

Doubly Linked List

a. Ứng dụng thực tế:

1. Lịch sử trình duyệt (Browser History)
2. LRU Cache

b. Giải thích:

1. **Browser History:** Duyệt qua lại các trang đã truy cập, cải thiện trải nghiệm người dùng.
2. **LRU Cache:** Tối ưu hiệu suất bằng cách lưu trữ các phần tử được truy cập gần nhất.

c. Cách áp dụng:

Trong ứng dụng lịch sử trình duyệt, Doubly Linked List được sử dụng để lưu trữ các trang web đã truy cập. Mỗi nút (HistoryNode) chứa URL của một trang và hai con trỏ prev và next để liên kết với các nút trước và sau. Điều này mang lại các lợi ích:

- **Duyệt qua lại:** Người dùng có thể dễ dàng quay lại trang trước (prev) hoặc tiến tới trang tiếp theo (next).
- **Thêm trang mới:** Khi người dùng truy cập một trang mới, nút mới được thêm vào sau nút hiện tại, và các liên kết được cập nhật.
- **Xóa lịch sử:** Có thể xóa một phần hoặc toàn bộ lịch sử bằng cách điều chỉnh các con trỏ.

Cấu trúc này rất hữu ích trong trình duyệt vì nó cung cấp khả năng điều hướng linh hoạt giữa các trang đã truy cập.

```
class HistoryNode:
    def __init__(self, url):
        self.url = url
        self.prev = None
        self.next = None

class BrowserHistory:
    def __init__(self):
        self.current = None

    def visit(self, url):
        new_node = HistoryNode(url)
        if self.current:
            self.current.next = new_node
            new_node.prev = self.current
        self.current = new_node
```

Stack

a. Ứng dụng thực tế:

1. **Chức năng Undo/Redo**
2. **Call Stack trong thực thi chương trình**

b. Giải thích:

1. **Undo/Redo:** Cho phép người dùng sửa lỗi, rất quan trọng trong các trình soạn thảo.
2. **Call Stack:** Quản lý các lệnh gọi hàm, là cốt lõi của mọi chương trình.

c. Cách áp dụng

Trong ứng dụng chức năng Undo/Redo, Stack được sử dụng để lưu trữ các hành động của người dùng. Cấu trúc này hoạt động theo nguyên tắc LIFO (Last In, First Out), giúp:

- **Thực hiện hành động:** Mỗi hành động được thêm vào đỉnh của ngăn xếp (execute).

- **Hoàn tác hành động:** Khi người dùng chọn Undo, hành động gần nhất (ở đỉnh ngăn xếp) được lấy ra và hoàn tác (undo).

Điều này rất quan trọng trong các trình soạn thảo văn bản hoặc phần mềm chỉnh sửa, nơi người dùng cần sửa lỗi nhanh chóng mà không làm ảnh hưởng đến các thay đổi trước đó.

```
class UndoStack:
    def __init__(self):
        self.stack = []

    def execute(self, action):
        self.stack.append(action)

    def undo(self):
        if self.stack:
            return self.stack.pop()
```

Queue

a. Ứng dụng thực tế

1. Hàng đợi in ấn (Printer Queue)
2. Duyệt theo chiều rộng (BFS)

b. Giải thích giá trị

1. Printer Queue: Xử lý các tác vụ in theo thứ tự FIFO, tránh xung đột.
2. BFS: Duyệt đồ thị theo tầng, ứng dụng trong tìm đường ngắn nhất.

c. Minh họa code

Trong ứng dụng hàng đợi in ấn, Queue được sử dụng để quản lý các tác vụ in theo thứ tự FIFO (First In, First Out). Điều này đảm bảo:

- **Thêm tác vụ in:** Khi người dùng gửi một lệnh in, tác vụ được thêm vào cuối hàng đợi (add_job).
- **Xử lý tác vụ in:** Máy in lấy tác vụ đầu tiên trong hàng đợi để xử lý (print_next), đảm bảo thứ tự công bằng.

Cách tiếp cận này giúp tránh xung đột giữa các tác vụ in và đảm bảo rằng các tài liệu được in theo đúng thứ tự mà chúng được gửi.

```
from collections import deque

class Printer:
    def __init__(self):
        self.queue = deque()

    def add_job(self, job):
        self.queue.append(job)

    def print_next(self):
        if self.queue:
            return self.queue.popleft()
```