ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Cấu trúc dữ liệu Skew Heap

Đề tài: Skew Heap - Một cấu trúc dữ liệu dạng Heap tự tái cấu trúc và các vấn đề liên quan

---

Môn học: Cấu trúc dữ liệu và giải thuật

Sinh viên thực hiện:

Nguyễn Đình Lê Vũ (24120016)

Nguyễn Phước Khang (24120189)

Phạm Minh Đạt (24120174)

Giảng viên hướng dẫn:

Thầy Nguyễn Thanh Phương

Thầy Nguyễn Thanh Tĩnh

Ngày 25 tháng 5 năm 2025

Mục lục

1 Thông tin chung **1**

1.1 Giới thiệu thành viên . . . 1

1.2 Phân Công Công Việc và Đánh Giá Đóng Góp . . . 1

1.3 Sản phẩm của nhóm . . . 1

2 Giới thiệu tổng quan **2**

2.1 Khái niệm Skew Heap . . . 2

2.2 Lịch sử phát triển . . . 2

2.3 Ứng dụng . . . 2

2.4 Một số cấu trúc dữ liệu liên quan . . . 3

2.4.1 Leftist Heap . . . 3

2.4.2 Bottom-Up Skew Heap . . . 4

3 Các thao tác cơ bản **5**

3.1 Trộn . . . 5

3.1.1 Mô tả . . . 5

3.1.2 Ví dụ . . . 6

3.1.3 Mở rộng . . . 10

3.2 Chèn một phần tử . . . 17

3.2.1 Mô tả . . . 17

3.2.2 Thực hành . . . 18

3.3 Trích ra phần tử nhỏ nhất . . . 18

3.3.1 Mô tả . . . 18

3.3.2 Thực hành . . . 18

4 Cài đặt và độ phức tạp **19**

4.1 Thao tác trộn (`merge`) . . . 19

4.1.1 Mã giả (Pseudo-code) . . . 19

4.1.2 Chi phí thực tế . . . 20

4.2 Độ phức tạp trong trường hợp tốt nhất, xấu nhất . . . 21

4.2.1 Trường hợp tốt nhất (Best-case) . . . 21

4.2.2 Trường hợp xấu nhất (Worst-case) . . . 22
4.3 Chi phí Amortized . . . 22
4.3.1 Các định nghĩa . . . 23
4.3.2 Bổ đề . . . 23
4.3.3 Đánh giá chi phí amortized . . . 24
4.4 Độ phức tạp của các phép chèn và trích phần tử nhỏ nhất . . . 26
4.4.1 Thuật toán `insert` . . . 26
4.4.2 Thuật toán `extractMin` . . . 26
4.5 Xây dựng Skew Heap từ dãy ban đầu . . . 27

5 So sánh với một số cấu trúc dữ liệu Heap khác **29**
5.1 Giới thiệu chung . . . 29
5.2 Mô tả từng loại Heap . . . 30
5.2.1 Binary Heap . . . 30
5.2.2 Leftist Heap . . . 30
5.2.3 Skew Heap . . . 31
5.3 So sánh thao tác . . . 31
5.4 Các đặc điểm khác biệt . . . 31

6 Tổ chức dự án và Ghi chú lập trình **33**
6.1 Tổ chức dự án . . . 33
6.2 Ghi chú lập trình . . . 33
6.2.1 Ghi nhận kết quả thực nghiệm . . . 33
6.2.2 Biên dịch chương trình . . . 34

Tài liệu **35**

A Phụ lục **35**

Danh sách hình vẽ

1 Leftist Heap với các giá trị NPL được đặt cạnh các nút khóa 4
2 Hai skew heap cần trộn 6
3 Mức đệ quy đầu tiên 7
4 Mức đệ quy thứ 2 7
5 Mức đệ quy thứ 3 8
6 Trả đệ quy về mức 3 8
7 Trả đệ quy về mức 2 9
8 Trả đệ quy về mức 1 9
9 Kết quả của phép trộn skew heap H_1, H_2 10
10 Khởi tạo ngăn xếp và gán con trỏ H_1, H_2 11
11 Các bước 1.1, 1.2, 1.3 được thực hiện 12
12 Kết quả sau bước 1.3 và 1.4 12
13 Các bước 2.1, 2.2, 2.3 được thực hiện 13
14 Kết quả sau bước 2.3 và 2.4 13
15 Các bước 3.1, 3.2, 3.3 được thực hiện 14
16 Kết quả sau bước 3.3 và 3.4 14
17 Chuẩn bị cho vòng lặp liên kết 15
18 Kết quả của vòng lặp liên kết 1 15
19 Kết quả của vòng lặp liên kết 2 16
20 Kết quả của vòng lặp liên kết 3 17

1 Thông tin chung

1.1 Giới thiệu thành viên

Nhóm gồm 3 thành viên đến từ lớp 24CNTN, Khoa Công nghệ thông tin, Trường Đại học Khoa học tự nhiên, ĐHQG - HCM. Bảng dưới đây là tên và mã số sinh viên của các thành viên của nhóm.

Họ và Tên	**Mã số sinh viên**
Nguyễn Đình Lê Vũ	24120016
Nguyễn Phước Khang	24120189
Phạm Minh Đạt	24120174

Bảng 1: Bảng giới thiệu thành viên nhóm.

1.2 Phân Công Công Việc và Đánh Giá Đóng Góp

Bảng sau đây tóm tắt các nhiệm vụ được giao cho từng thành viên trong nhóm cùng với tỷ lệ đóng góp của mỗi bạn. Tỷ lệ đóng góp phản ánh mức độ hoàn thành các nhiệm vụ được giao.

Họ và Tên	**Các Nhiệm Vụ Được Giao**	**Tỷ Lệ Đóng Góp**
Nguyễn Đình Lê Vũ	Báo cáo: Giới thiệu tổng quan, Các thao tác cơ bản, Độ phức tạp thao tác trộn (merge); Chỉnh sửa slide và report	100%
Nguyễn Phước Khang	Báo cáo: Cài đặt và độ phức tạp, Tổ chức dự án và ghi chú lập trình; Viết source code (Phần: Merge, Insert, ExtractMin, Build Skew heap from given Array); Chỉnh sửa report	100%
Phạm Minh Đạt	Báo cáo: So sánh với cấu trúc dữ liệu Heap khác; Viết source code (Phần: CheckEmpty, getNumber of Items, Return items at the root, Remove all items from heap); Chỉnh sửa report	100%

Bảng 2: Bảng phân công công việc và tỷ lệ đóng góp của các thành viên nhóm.

1.3 Sản phẩm của nhóm

- Bài thuyết trình của nhóm được đăng không công khai tại YouTube.

2 Giới thiệu tổng quan

2.1 Khái niệm Skew Heap

Skew Heap [1] là một cấu trúc dữ liệu dạng heap tự điều chỉnh (self-adjusting heap), được triển khai dưới dạng cây nhị phân, tập trung vào việc tối ưu hóa thao tác trộn (merge). Skew Heap không có ràng buộc cấu trúc nào ngoài tính chất heap cơ bản: khóa của mỗi nút thì nhỏ hơn hay bằng (đối với min-heap) hoặc lớn hơn hoặc bằng (đối với max-heap) các khóa của nút con. Để cho tiện, xuyên suốt bài báo cáo này, ta sẽ làm việc với min-heap.

Một cách hình thức, một Skew Heap là một cây nhị phân được định nghĩa đệ quy như sau:

i. Cây rỗng là Skew heap.

ii. Một cây nhị phân có nút gốc mang khóa k và có hai cây con phải là L, R, thì cây ấy là Skew heap nếu như k bé hơn khóa của mọi nút trong cây L hoặc R (nếu có).

2.2 Lịch sử phát triển

Skew Heap là một biến thể của *Leftist Heap*, được thiết kế để đơn giản hóa quá trình trộn. Cấu trúc dữ liệu này được Daniel D. Sleator và Robert E. Tarjan giới thiệu lần đầu trong bài báo năm 1986, *Self-adjusting heaps*, đăng trên Tạp chí SIAM về Tính toán. Điểm sáng tạo của Skew Heap là thao tác trộn (merge), được sử dụng cho tất cả các thao tác chính (chèn, xóa, trộn) và đạt độ phức tạp *trung bình* $O(\log n)$ bằng cách hoán đổi vô điều kiện các nút con trái/phải trên đường trộn để duy trì tính cân bằng của heap [2].

2.3 Ứng dụng

Thiết kế của Skew Heap xuất phát từ nhu cầu về *hàng đợi ưu tiên có thể trộn*, nơi việc trộn hai heap là thao tác thường xuyên. Skew Heap được sử dụng trong nhiều ứng dụng nhờ thao tác trộn hiệu quả và tính đơn giản. Các ứng dụng chính bao gồm:

1. *Hàng chờ có độ ưu tiên*: Skew Heap là một cách triển khai cho hàng chờ có độ ưu tiên, nơi các phần tử được truy xuất dựa trên mức độ ưu tiên. Khả năng trộn nhanh chóng khiến Skew heap phù hợp cho các hệ thống cần kết hợp hàng chờ có độ ưu tiên, chẳng hạn như trong thuật toán lập lịch hoặc mô phỏng sự kiện.

2. *Triển khai hàng chờ có độ ưu tiên:* Skew Heap được sử dụng trong hệ thống quản lý hàng chờ công việc trên các cụm tính toán HPC. Ví dụ, một dự án tại Đại học Maryland (CMSC 341) triển khai hàng đợi công việc dựa trên max-Skew Heap để ưu tiên các công việc tính toán dựa trên số lượng bộ xử lý hoặc thời gian chạy dự kiến. Skew Heap đảm bảo công việc có ưu tiên cao nhất luôn ở gốc, và thao tác hợp nhất giúp cập nhật hàng đợi động. [3]

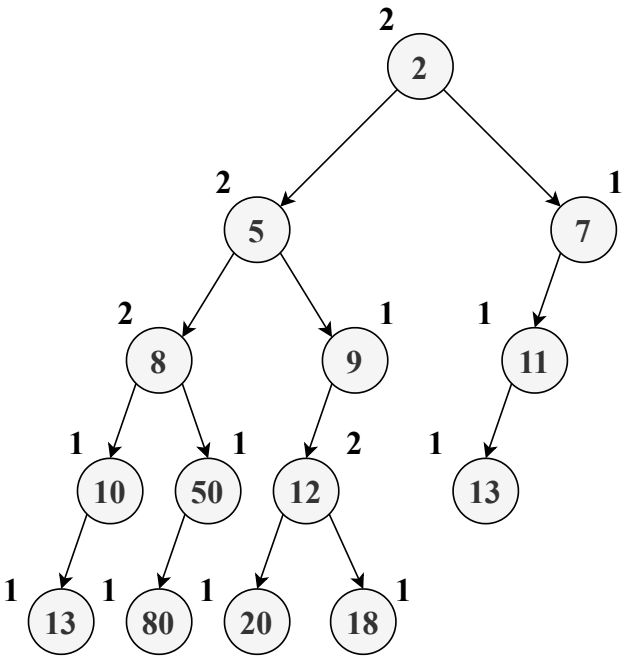
3. *Ứng dụng trong lập trình hàm:* Trong các ngôn ngữ lập trình hàm như Elixir, Skew Heap được sử dụng để triển khai hàng đợi ưu tiên nhờ sự đơn giản và hiệu quả trong thao tác hợp nhất. Thư viện `Prioqueue` của Elixir [4] xây dựng hàng đợi ưu tiên dựa trên Skew Heap, sử dụng một hàm "combine" để thực hiện tất cả các thao tác.

4. *Các thuật toán đồ thị*: Mặc dù ít phổ biến hơn Binary Heap hoặc Fibonacci Heap, Skew Heap vẫn có thể được sử dụng làm hàng đợi ưu tiên trong các thuật toán đồ thị như Dijkstra hoặc Prim, đặc biệt trong các tình huống xử lý song song hoặc phân tán, nơi thao tác hợp nhất được sử dụng thường xuyên. Tuy nhiên, Binary Heap thường được ưu tiên hơn do tính đơn giản và hiệu quả bộ nhớ đệm.

2.4 Một số cấu trúc dữ liệu liên quan

2.4.1 Leftist Heap

Leftist Heap là một cây nhị phân duy trì tính chất heap (khóa của mỗi nút nhỏ hơn hoặc bằng khóa của các nút con, nếu có) và một ràng buộc cấu trúc gọi là *độ dài đường dẫn null (NPL)*. NPL của một nút là độ dài của đường dẫn ngắn nhất đến một nút null, và Leftist Heap đảm bảo NPL của nút con trái lớn hơn hoặc bằng nút con phải (xem hình 1). Tính chất này giữ cây thiên về bên trái, đảm bảo chiều dài của cột phải tối đa là $O(\log n)$, dẫn đến thời gian $O(\log n)$ cho các thao tác trộn, chèn và xóa [5].



Hình 1: Leftist Heap với các giá trị NPL được đặt cạnh các nút khóa

Thực tế, Skew Heap là một biến thể trực tiếp của Leftist Heap, Skew Heap được xem như một giải pháp đơn giản hơn. Trong khi Leftist Heap sử dụng NPL để duy trì cân bằng trong quá trình trộn, Skew Heap loại bỏ yêu cầu này, dựa vào việc hoán đổi nút con vô điều kiện, giúp đơn giản hóa triển khai nhưng hy sinh sự đảm bảo về cân bằng [2].

2.4.2 Bottom-Up Skew Heap

Bottom-Up Skew Heap là một biến thể của Skew Heap, trong đó thao tác trộn được thực hiện theo cách từ dưới lên. Không giống như Skew Heap bình thường (từ trên xuống), vốn so sánh trực tiếp các gốc, trộn các cây con nhỏ hơn và hoán đổi nút con trong quá trình đi, chúng trộn dọc theo cột phải và hoán đổi nút con sau đó [2].

Bottom-Up Skew Heap giống Skew heap ở chỗ cả 2 cấu trúc dữ liệu đều không có ràng buộc cấu trúc và dựa vào hoán đổi nút con. Chúng khác nhau là ở hướng của thuật toán trộn, hiệu suất phụ thuộc tùy vào bài toán cụ thể do hành vi bộ nhớ đệm hoặc mẫu đệ quy khác nhau.

Theo Sleator và Tarjan [2], chi phí amortized của thao tác trộn (merge) và chèn (insert) trong bottom-up skew heap là $O(1)$ so với $O(\log n)$ của Top-down skew heap thông thường. Tuy nhiên

thao tác trích xuất phần tử nhỏ nhất (extract-min) có chi phí amortized là $O(\log n)$, giống với Top-down Skew heap.

3 Các thao tác cơ bản

3.1 Trộn

3.1.1 Mô tả

Thao tác *trộn* trong Skew Heap kết hợp hai Skew Heap thành một heap duy nhất, giữ nguyên tính chất heap (đối với min-heap, khóa của mỗi nút nhỏ hơn hoặc bằng khóa của các nút con). Không giống Leftist Heap, Skew Heap không duy trì ràng buộc cấu trúc như độ dài đường dẫn null (NPL), mà dựa vào việc hoán đổi nút con vô điều kiện để đảm bảo hiệu quả [2].

Có nhiều cách để định nghĩa thao tác trộn, tùy thuộc vào cách gọi đệ quy: Tail Recursion hay Non-Tail Recursion. Để đơn giản, phần ví dụ này sẽ được trình bày theo hình thức đệ quy đuôi (hay Non-Tail recursion). Các bước của thao tác trộn bao gồm:

1. *Trường hợp cơ bản*: Nếu một heap rỗng, trả về heap còn lại. Điều này cho thấy nếu cả hai heap đều rỗng, heap trả về là rỗng.

2. *So sánh khóa của hai nút gốc*: Chọn heap có khóa tại nút gốc nhỏ hơn (giả sử Heap A) làm gốc của heap sau khi trộn. Heap còn lại (Heap B) sẽ được trộn một cách đệ quy với cây con phải ban đầu của Heap A.

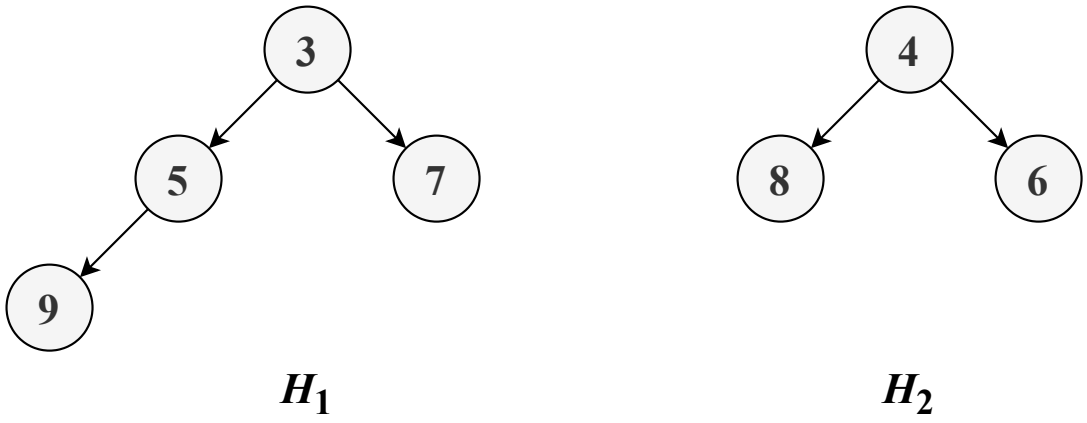
3. *Hoán đổi hai heap con trái, phải của nút gốc*: Tiến hành hoán đổi cây con trái và phải của gốc Heap A (heap có khóa tại gốc nhỏ hơn, tại mỗi bước đệ quy). [6].

4. *Trộn*: Lấy cây con trái của Heap A (sau khi hoán đổi) và trộn với Heap B (toàn bộ heap thứ hai, bao gồm gốc). Kết quả của trộn đệ quy trở thành cây con trái mới của gốc Heap A. Cây con phải của Heap A sau khi trộn giữ nguyên.

5. *Trả về heap được trộn*: Heap cuối cùng có gốc của Heap A làm gốc, với cây con trái (cây con phải ban đầu, đã được trộn đệ quy) và cây con phải (cây con trái ban đầu).

3.1.2 Ví dụ

Ta thử xét một ví dụ bất kì, 2 cây cây skew heap (H_1, H_2) như sau:



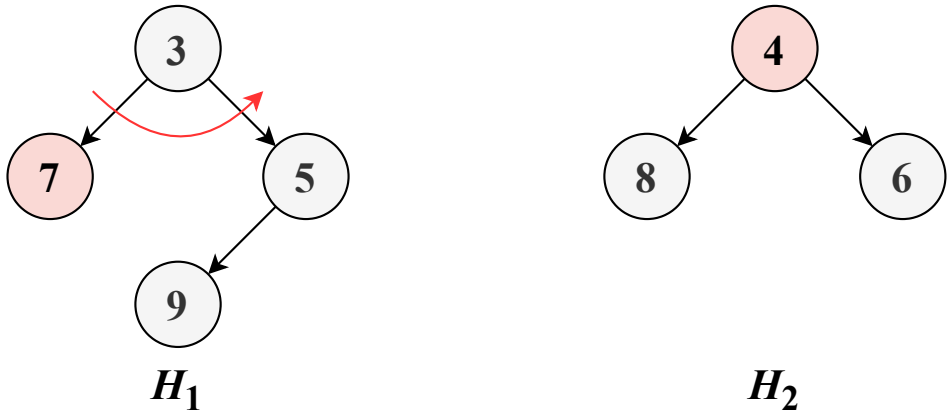
Hình 2: Hai skew heap cần trộn

Để dễ hiểu, ta xem như H_1 là cây bên trái, còn H_2 là cây bên phải, và ở mỗi mức đệ quy ta vẫn sẽ giữ nguyên tất cả các nút có từ mức trước đó trong khi tô màu các nút gốc của hai cây cần trộn ở mức hiện tại.

Ở *mức đệ quy thứ 1*: Ta cần trộn hai cây có gốc trỏ đến các nút 3, 4.

- Vì $3 < 4$, chọn nút trỏ đến (3) là nút gốc của A (tức cây heap có nút gốc nhỏ hơn trong hai heap cần trộn), và nút trỏ đến (4) là nút gốc của B (tức cây heap có nút gốc lớn hơn trong hai heap cần trộn).
- *Hoán đổi hai nút con của A*: Hoán đổi nút con trái (5) và phải (7) của nút gốc (3) trong A. Ghi nhớ nút gốc của cây con trái của A, lúc này là 7 để trộn với B.

Lúc này, các cây cần trộn có gốc trỏ đến các nút 7, 4 (các nút màu đỏ) như hình 3.

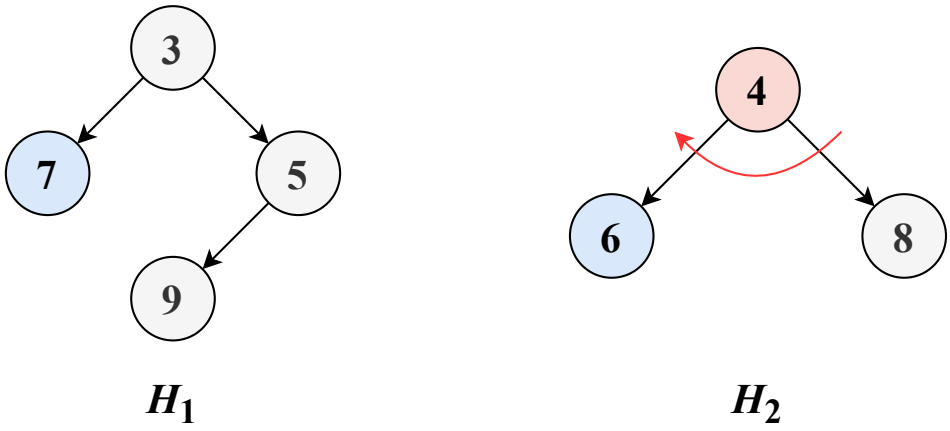


Hình 3: Mức đệ quy đầu tiên

Ở mức đệ quy thứ 2: Ta cần trộn hai cây có gốc trỏ đến các nút $7, 4$.

- Vì $4 < 7$, chọn nút trỏ đến (4) làm nút gốc của A, nút trỏ đến (7) là nút gốc của B.
- *Hoán đổi hai nút con của A*: Hoán đổi nút con trái (8) và phải (6) của nút gốc (4) trong A. Ghi nhớ nút gốc của cây con trái của A, lúc này là 6 để trộn với B.

Lúc này, các cây cần trộn có gốc trỏ đến các nút 7, 6 (các nút màu xanh) như hình 4.



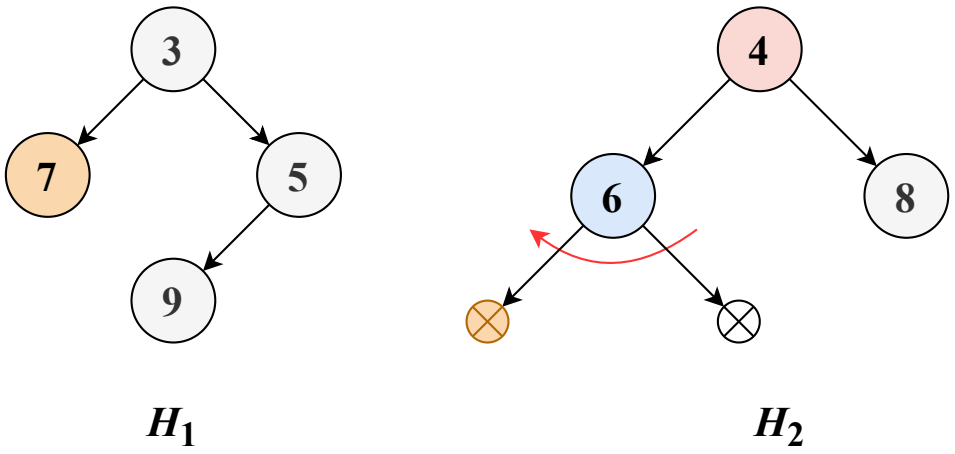
Hình 4: Mức đệ quy thứ 2

Ở mức đệ quy thứ 3: Ta cần trộn hai cây có gốc trỏ đến các nút 7, 6.

- Vì $6 < 7$, chọn nút trỏ đến (6) làm nút gốc của A, nút trỏ đến (7) là nút gốc của B.

- *Hoán đổi hai nút con của A*: Hoán đổi nút con trái (`NULL`) và phải (`NULL`) của nút gốc (6) trong A. Ghi nhớ nút gốc của cây con trái của A, lúc này là `NULL` để trộn với B.

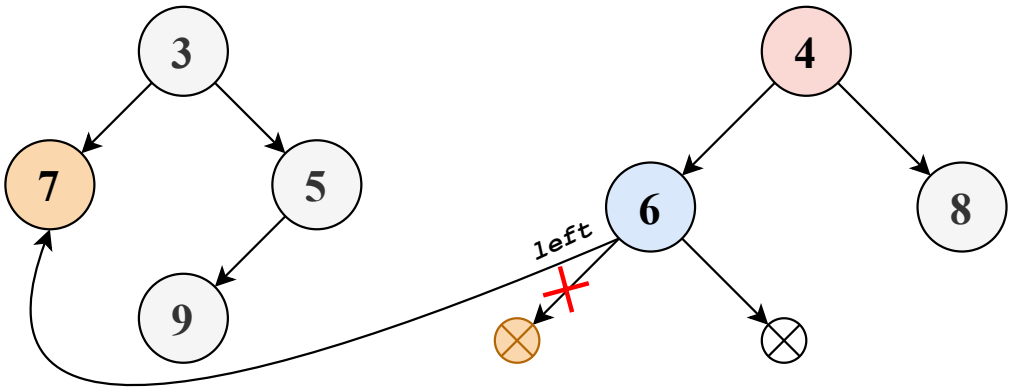
Lúc này, các các cây cần trộn có gốc trỏ đến các nút 7, `NULL` (các nút màu vàng) như hình 5.



Hình 5: Mức đệ quy thứ 3

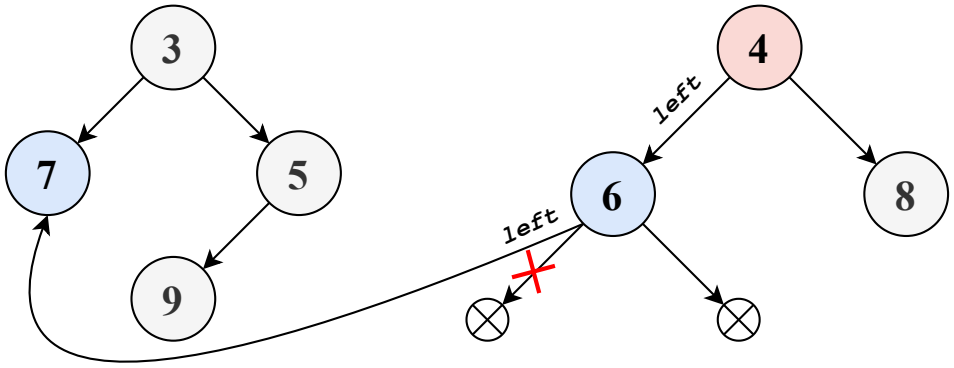
Ở mức đệ quy thứ 4: Ta cần trộn hai cây có gốc trỏ đến các nút 7, `NULL`. Gốc của một trong hai cây trỏ đến `NULL`, giải thuật trả về gốc còn lại, tức là nút 7.

Bây giờ ta sẽ minh họa quá trình trả đệ quy:
Quyền điều khiển trả đệ quy về tại mức 3 ở hình 6, cây con trái của nút gốc cần trộn (là 6) trỏ đến nút gốc của skew heap đã được trộn xong (là 7). Đối chiếu với hình 5.



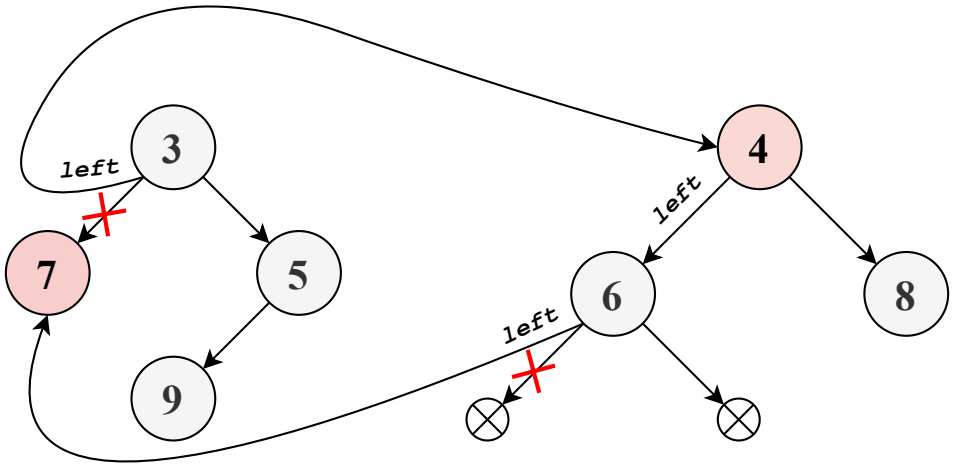
Hình 6: Trả đệ quy về mức 3

Quyền điều khiển trả đệ quy về tại mức 2 ở hình 7, cây con trái của nút gốc để trộn (là 4) trỏ đến nút gốc của skew heap đã được trộn xong (là 6). Thật ra ở bước này, vì nút 6 đã sẵn là cây con trái của nút 4, nên thực ra không có sự thay đổi. Đối chiếu với hình 4.



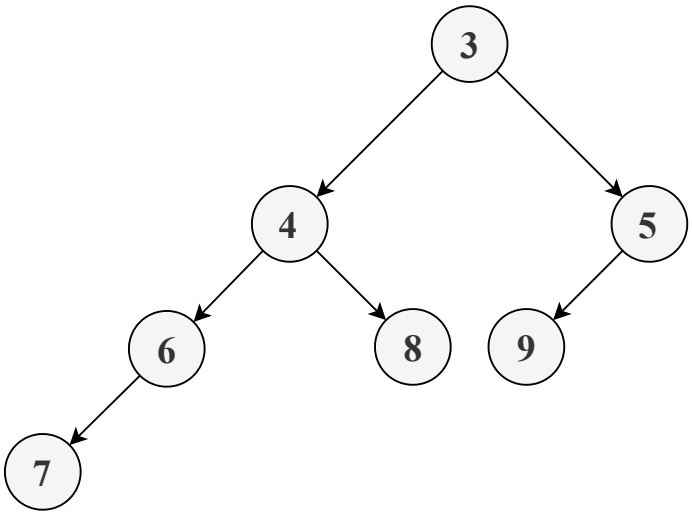
Hình 7: Trả đệ quy về mức 2

Cuối cùng, khi quyền điều khiển trả đệ quy về tại mức đầu tiên, cây con trái của nút gốc cần trộn (là 3) trỏ đến nút gốc của skew heap đã được trộn xong (chính là 4). Đối chiếu với hình 3.



Hình 8: Trả đệ quy về mức 1

Như vậy, sau khi trộn 2 skew heap H_1 và H_2, ta được cây sau:



Hình 9: Kết quả của phép trộn skew heap H_1, H_2

Đây đã là một skew heap vì nó thỏa mãn tính chất của min-heap.

3.1.3 Mở rộng

Ngoài cách cài đặt bằng đệ quy nói trên, còn có các cách không sử dụng đệ quy. Phần này giới thiệu sơ lược về cài đặt thao tác trộn bằng kỹ thuật khử đệ quy sử dụng cấu trúc dữ liệu Ngăn xếp (Stack).

Bây giờ ta mô tả thuật toán khử đệ quy. Các bước cơ bản bao gồm:

1. *Trường hợp cơ bản*: Nếu một heap rỗng, trả về heap còn lại. Nếu cả hai heap rỗng, trả về heap rỗng.
2. *Khởi tạo ngăn xếp*: Tạo một ngăn xếp rỗng để lưu trữ các cặp nút.
3. *Vòng lặp chính*: Trong khi H_1 và H_2 còn khác rỗng:
    - *So sánh các nút gốc cần trộn*: So sánh khóa của gốc của hai heap (H_1 và H_2). Nếu khóa gốc của H_1 lớn hơn H_2, hoán đổi H_1 và H_2 để đảm bảo H_1 có gốc nhỏ hơn.
    - *Đẩy vào ngăn xếp*: Đẩy nút gốc của H_1 vào ngăn xếp.
    - *Hoán đổi nút con*: Hoán đổi cây con trái và phải của gốc H_1 để duy trì tính tự điều chỉnh.

- *Cập nhật cây con*: Đặt cây con trái của gốc H_1 là kết quả của việc hợp nhất cây con phải hiện tại với H_2; nói cách khác, gán H_1 bởi cây con trái của nó.

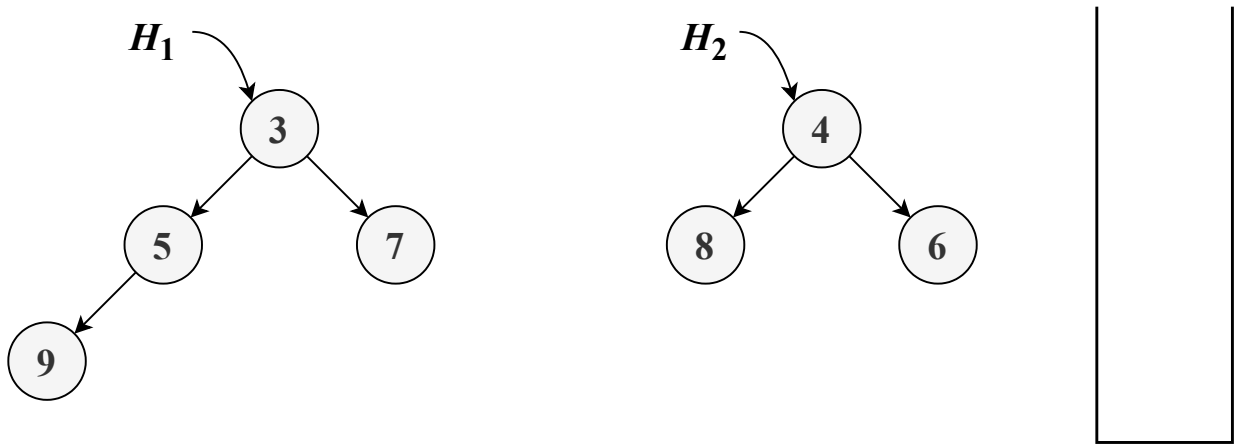
4. *Vòng lặp liên kết*: Ta biết rằng có đúng một trong hai heap H_1, H_2 khác rỗng, gọi nút gốc của heap ấy là *nút gốc của heap đã trộn xong hiện tại*. Trong khi ngăn xếp còn khác rỗng:

   - *Trích ra ngoài ngăn xếp phần tử đầu tiên*: Cây con trái của nút vừa trích sẽ trỏ đến *nút gốc của heap trộn xong hiện tại*.
   - *Nút gốc của heap trộn xong hiện tại*, lúc này, gán bởi nút vừa trích.

   Ta có thể nhận ra rằng vòng lặp chính lặp bao nhiêu lần thì vòng lặp liên kết lặp lại bấy nhiêu lần, tương ứng với các lần trả để quy ở thuật toán đệ quy thông thường.

5. Trả về kết quả là *nút gốc của heap trộn xong hiện tại*.

Bây giờ, chúng ta xét lại ví dụ ở mục trước để hiểu rõ hơn thuật toán khử đệ quy. Ta cần trộn hai heap ở hình 2. Vì đây không phải là trường hợp cơ bản, ta cần khởi tạo một ngăn xếp và gán các con trỏ H_1, H_2, minh họa bởi hình 10.



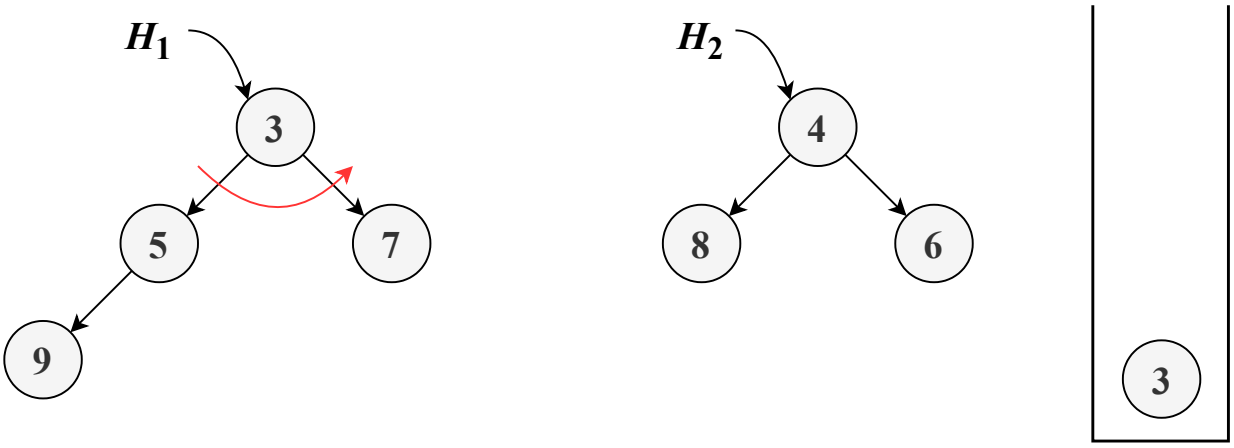
Hình 10: Khởi tạo ngăn xếp và gán con trỏ H_1, H_2

Ở vòng lặp chính 1:

1.1 *So sánh các nút gốc của* H_1, H_2: Các nút H_1, H_2 có khóa là 3, 4. Vì $3 < 4$, ta giữ nguyên H_1, H_2.

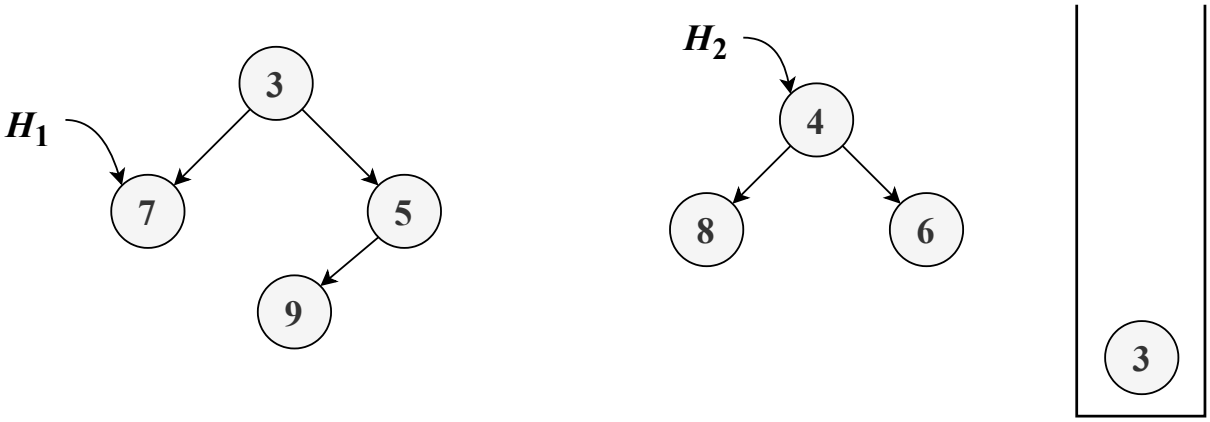
1.2 *Đẩy vào ngăn xếp*: Đẩy nút gốc của H_1 (3) vào ngăn xếp.

1.3 *Hoán đổi nút con*: Hoán đổi cây con trái (5) và phải (7) của nút gốc H_1.



Hình 11: Các bước 1.1, 1.2, 1.3 được thực hiện

1.4 *Cập nhật cây con*: Gán H_1 bởi cây con trái của nó, bây giờ H_1 trỏ đến nút (7).



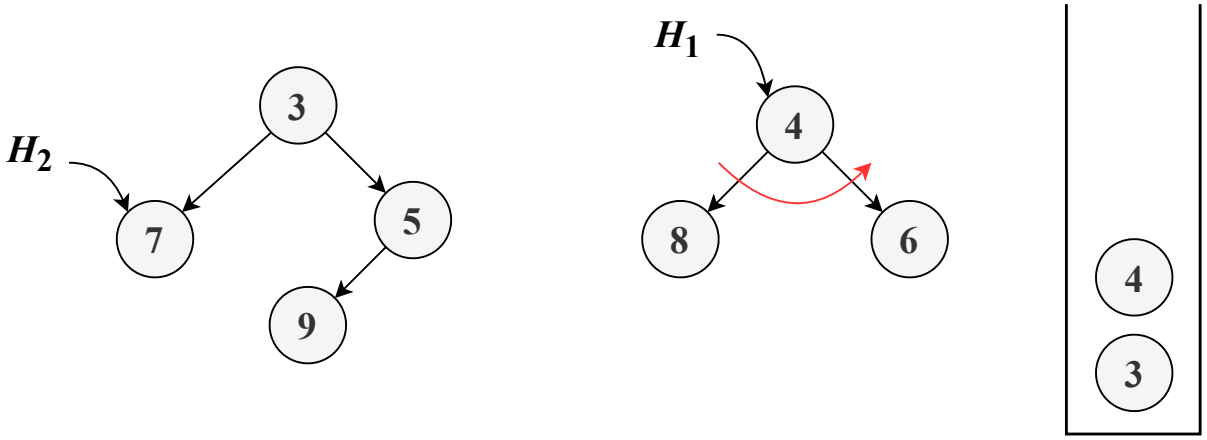
Hình 12: Kết quả sau bước 1.3 và 1.4

Ở vòng lặp chính 2:

2.1 *So sánh các nút gốc của* H_1, H_2: Các nút H_1, H_2 có khóa là 7, 4. Vì $7 > 4$, ta hoán chuyển H_1, H_2.

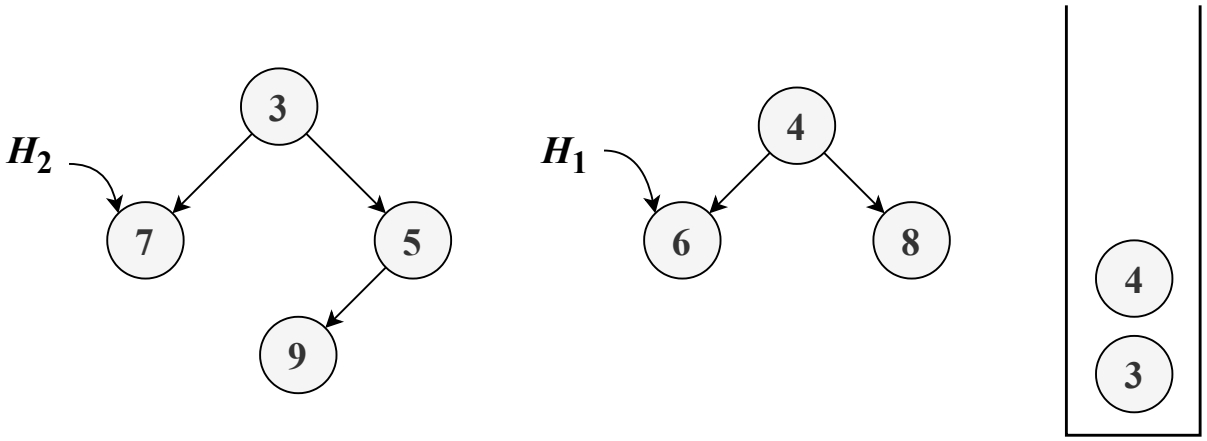
2.2 *Đẩy vào ngăn xếp*: Đẩy nút gốc của H_1 (4) vào ngăn xếp.

2.3 *Hoán đổi nút con*: Hoán đổi cây con trái (8) và phải (6) của nút gốc H_1.



Hình 13: Các bước 2.1, 2.2, 2.3 được thực hiện

2.4 *Cập nhật cây con*: Gán H_1 bởi cây con trái của nó, bây giờ H_1 trỏ đến nút (6).



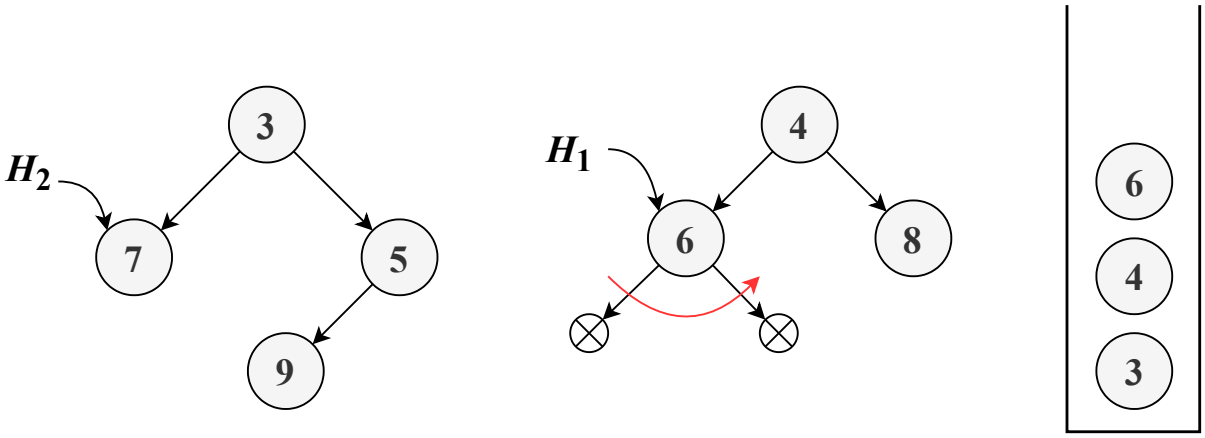
Hình 14: Kết quả sau bước 2.3 và 2.4

Ở vòng lặp chính 3:

3.1 *So sánh các nút gốc của* H_1, H_2: Các nút H_1, H_2 có khóa là 6, 7. Vì $6 > 7$, ta giữ nguyên H_1, H_2.

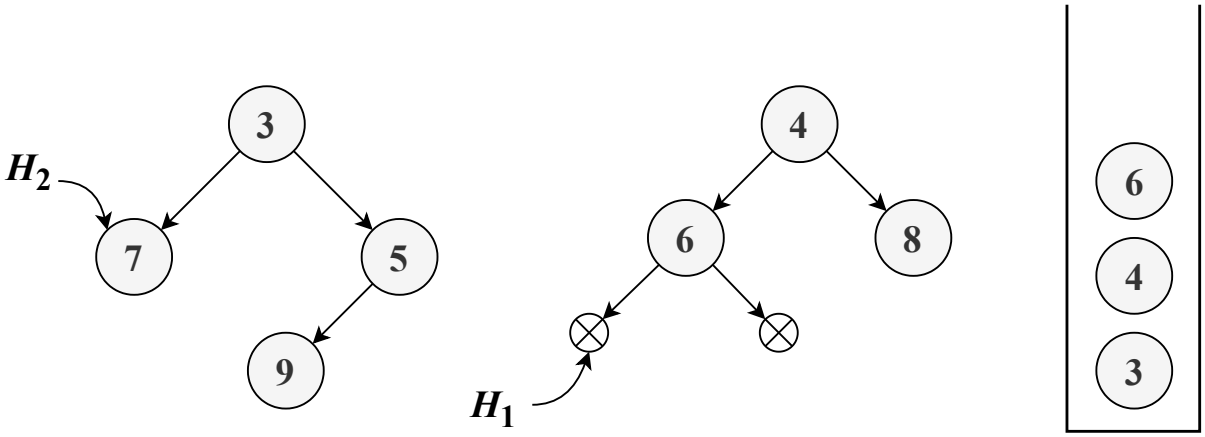
3.2 *Đẩy vào ngăn xếp*: Đẩy nút gốc của H_1 (6) vào ngăn xếp.

3.3 *Hoán đổi nút con*: Hoán đổi cây con trái (`NULL`) và phải (`NULL`) của nút gốc H_1.



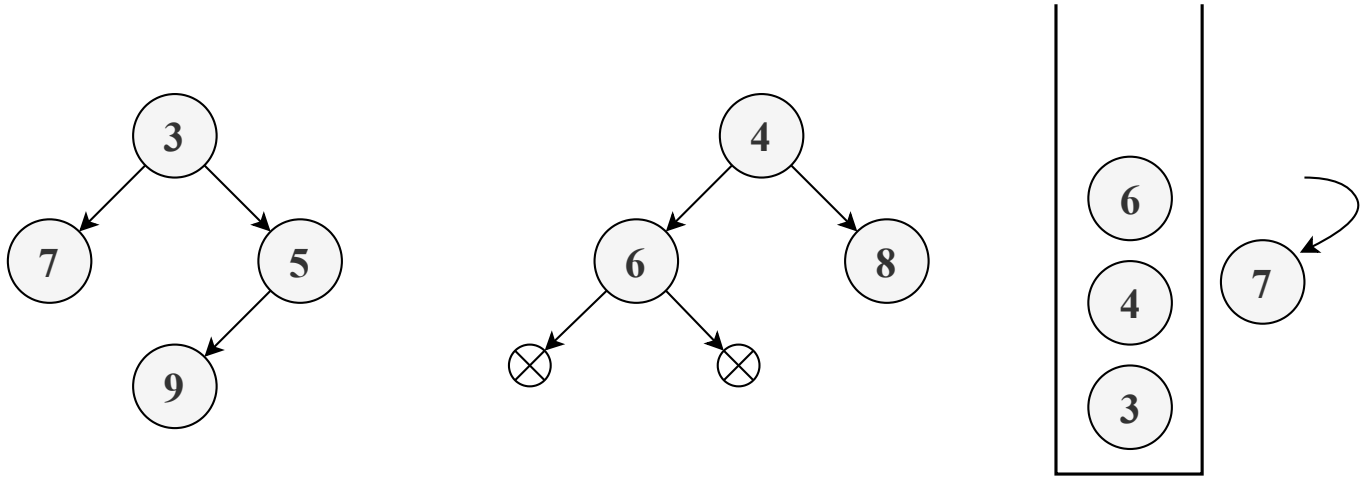
Hình 15: Các bước 3.1, 3.2, 3.3 được thực hiện

3.4 *Cập nhật cây con*: Gán H_1 bởi cây con trái của nó, bây giờ H_1 trỏ đến `NULL`.



Hình 16: Kết quả sau bước 3.3 và 3.4

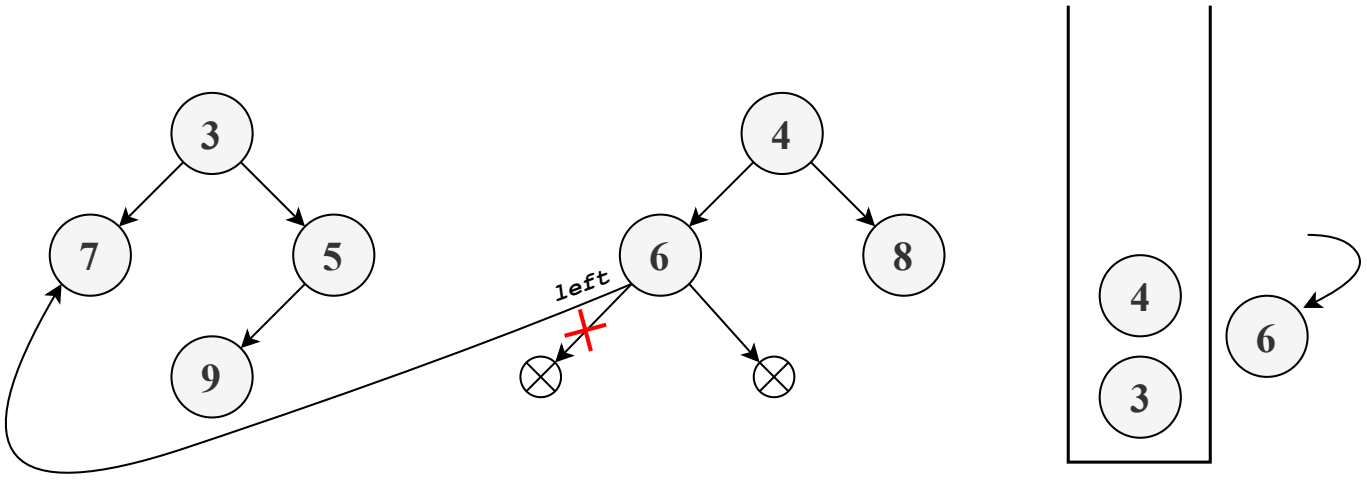
Sau khi thoát khỏi vòng lặp chính, ta biết rằng trong hai nút gốc H_1, H_2 khác rỗng. Trong trường hợp này là H_2 (7), vì thế gán *nút gốc của heap đã trộn xong hiện tại* đến nút trỏ đến (7), minh họa bởi hình 17.



Hình 17: Chuẩn bị cho vòng lặp liên kết

Vòng lặp liên kết 1:

- Trích ra ngoài ngăn xếp phần tử đầu tiên (6), cây con trái của (6) sẽ trỏ đến (7), tức *nút gốc của heap trộn xong hiện tại.*
- *Nút gốc của heap trộn xong hiện tại*, gán bởi nút (6), tức nút vừa trích khỏi ngăn xếp.



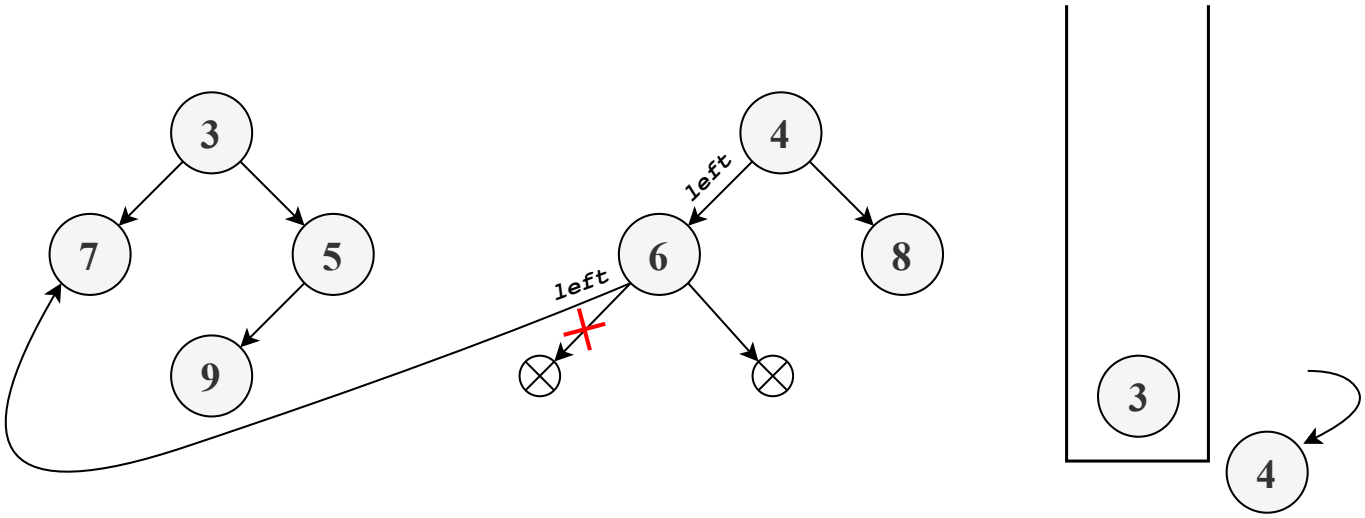
Hình 18: Kết quả của vòng lặp liên kết 1

Vòng lặp liên kết 2:

- Trích ra ngoài ngăn xếp phần tử đầu tiên (4), cây con trái của (4) sẽ trỏ đến (6), tức *nút gốc*

của heap trộn xong hiện tại. Có thể thấy rằng cây con trái của nút (4) đã sẵn có gốc là (6), nên về cơ bản không có sự thay đổi.

- *Nút gốc của heap trộn xong hiện tại*, gán bởi nút (4), tức nút vừa trích khỏi ngăn xếp.

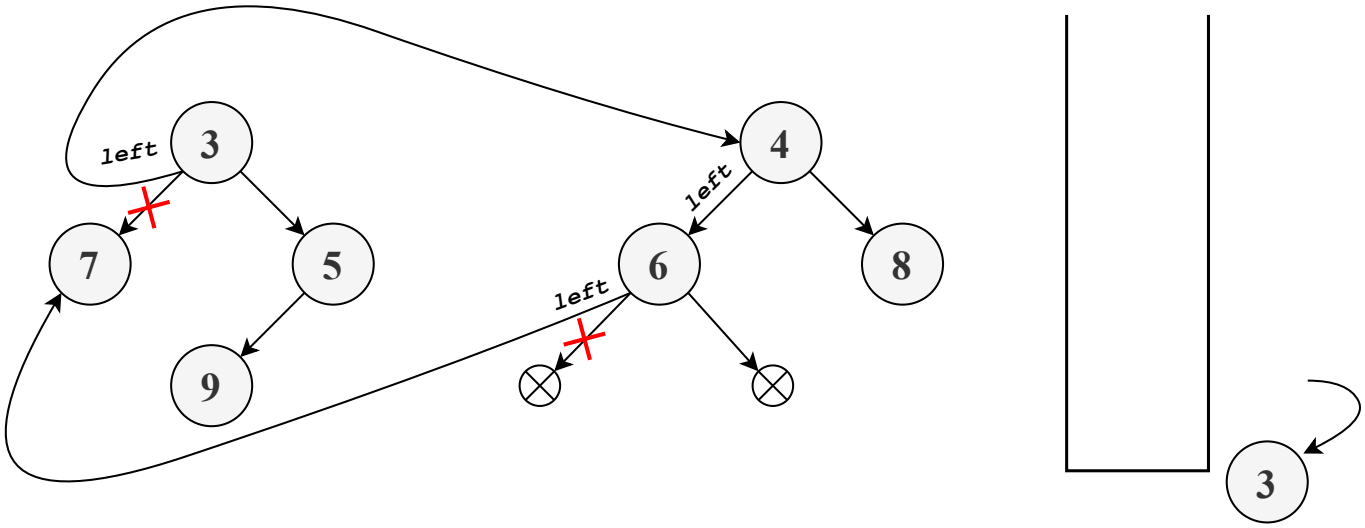


Hình 19: Kết quả của vòng lặp liên kết 2

Vòng lặp liên kết 3:

- Trích ra ngoài ngăn xếp phần tử đầu tiên (3), cây con trái của (3) sẽ trỏ đến (4), tức *nút gốc của heap trộn xong hiện tại*.

- *Nút gốc của heap trộn xong hiện tại*, gán bởi nút (3), tức nút vừa trích khỏi ngăn xếp.



Hình 20: Kết quả của vòng lặp liên kết 3

Đến lúc này ngăn xếp đã là rỗng, nên vòng lặp liên kết dừng lại (phù hợp với nhận xét đã nêu ở trên, số vòng lặp chính bằng số vòng lặp liên kết, và đều bằng 3 trong trường hợp này). Bây giờ, trả về kết quả là *nút gốc của heap trộn xong hiện tại*, tức là (3). Ta được cây heap hoàn toàn giống với hình 9 của phép trộn sử dụng đệ quy.

3.2 Chèn một phần tử

Thao tác *chèn một phần tử* vào Skew Heap được thực hiện bằng cách tạo một Skew Heap mới chứa chỉ phần tử đó, sau đó hợp nhất (merge) heap này với Skew Heap hiện có. Đây là cách tiếp cận đơn giản, tận dụng thao tác trộn – thao tác cốt lõi của Skew Heap – để đảm bảo tính chất min-heap (khóa của mỗi nút nhỏ hơn hoặc bằng khóa của các nút con).

3.2.1 Mô tả

Các bước chèn (cho min-heap):

1. *Tạo heap mới:* Tạo một Skew Heap đơn gồm một nút chứa phần tử cần chèn (không có nút con).

2. *Trộn*: Gọi thao tác trộn giữa heap hiện có và heap đơn vừa tạo, sử dụng thuật toán trộn (đệ quy hoặc không đệ quy) của Skew Heap.

3. *Trả về kết quả:* Heap sau hợp nhất là kết quả, với phần tử mới được tích hợp vào cấu trúc, duy trì tính chất heap.

3.2.2 Thực hành

Vì tính đơn giản, ta sẽ lược qua phần ví dụ của thao tác này. Chúng ta có thể thực hành thao tác chèn một phần tử vào skew heap bất kì một cách trực quan hóa thông qua trang web [7].

3.3 Trích ra phần tử nhỏ nhất

Thao tác *trích xuất phần tử nhỏ nhất* (extract-min) trong Skew Heap loại bỏ và trả về nút có khóa nhỏ nhất (gốc của min-heap), sau đó tái cấu trúc heap để duy trì tính chất min-heap (khóa của mỗi nút nhỏ hơn hoặc bằng khóa của các nút con). Trong Skew Heap, thao tác này được thực hiện bằng cách lấy gốc (phần tử nhỏ nhất), sau đó trộn hai cây con của gốc (cây con trái và cây con phải) để tạo ra heap mới.

3.3.1 Mô tả

1. *Kiểm tra rỗng:* Nếu heap rỗng, trả về lỗi hoặc giá trị đặc biệt (tùy triển khai).

2. *Lấy gốc:* Lấy nút gốc (chứa khóa nhỏ nhất) và lưu giá trị của nó để trả về.

3. *Trộn hai cây con:* Gọi thao tác hợp nhất (đệ quy hoặc không đệ quy) giữa cây con trái và cây con phải của gốc để tạo heap mới.

4. *Trả về:* Trả về giá trị của gốc và heap mới sau khi trộn.

3.3.2 Thực hành

Tương tự thao tác *chèn*, vì tính đơn giản, ta sẽ lược qua phần ví dụ của thao tác này. Chúng ta có thể thực hành thao tác trích ra phần tử nhỏ nhất của một skew heap bất kì một cách trực quan hóa thông qua trang web [7].

4 Cài đặt và độ phức tạp

4.1 Thao tác trộn (`merge`)

4.1.1 Mã giả (Pseudo-code)

Ta trình bày mã giả cho hai phiên bản của thao tác trộn các cây skew heap. Thuật toán trộn phiên bản đệ quy được trình bày như trong bảng 1, phù hợp với miêu tả và ví dụ minh họa đã được trình bày ở phần trước.

Algorithm 1 Thuật toán trộn hai skew heap sử dụng đệ quy

```
1:  function MERGE(H_1, H_2)
2:      if H_1 = null then
3:          return H_2
4:      else if H_2 = null then
5:          return H_1
6:      else if H_1.key > H_2.key then
7:          H_1 ⇌ H_2
8:      end if
9:      H_1.left ⇌ H_1.right
10:     H_1.left ← MERGE(H_1.left, H_2)
11:     return H_1
12: end function
```

Còn thuật toán trộn phiên bản khử đệ quy được trình bày như trong bảng 2, phù hợp với miêu tả và ví dụ minh họa đã được trình bày ở phần trước.

Algorithm 2 Thuật toán trộn hai skew heap sử dụng kỹ thuật khử đệ quy

```
1: function nonRecursionMerge(a, b)
2:     if a = NULL then
3:         a ← b
4:         b ← NULL
5:         return
6:     end if
7:     if b = NULL then
8:         b ← a
9:         a ← NULL
10:        return
11:    end if
12:    stack S ← rỗng
13:    p ← a, q ← b
14:    prev, Cur, temp ← NULL
15:    while p ≠ NULL and q ≠ NULL do
16:        if p.key > q.key then
17:            p ⇌ q
18:        end if
19:        S.push(p)
20:        p.left ⇌ p.right
21:        p ← p.left
22:    end while
23:    if p ≠ NULL then
24:        prev ← p
25:    else
26:        prev ← q
27:    end if
28:    while not S.empty() do
29:        Cur ← S.top()
30:        S.pop()
31:        Cur.left ← prev
32:        prev ← Cur
33:    end while
34:    return prev
35: end function
```

4.1.2 Chi phí thực tế

Chi phí thời gian phụ thuộc vào số nút trên đường đi phải (*right path*) của hai heap đầu vào. Ký hiệu r_1, r_2 lần lượt là số nút trên đường đi phải của H_1, H_2. (Ở đây, độ dài đường đi phải được tính theo cạnh, và một đường đi phải là đường đi xuất phát từ một nút, đến nút *tận cùng bên phải* của nút ấy).

Thực tế là tiến trình đệ quy chỉ diễn ra dọc theo các nút thuộc đường đi phải của hai nút gốc ban đầu. Để dễ tưởng tượng, ta bỏ qua việc hoán đổi, và xem như hoán đổi lúc trả đệ quy. Thực tế là đây là phiên bản non-tail recursion và cho ra kết quả tương tự, được trình bày trong các tài liệu như [1]. Ta tưởng tượng rằng ở mỗi bước, hai nút được trộn sẽ lần lượt thay đổi theo cách: một nút giữ nguyên (tức nút có khóa lớn hơn), và nút còn lại gán bởi nút con phải của nó. Quá trình trên thực hiện đến khi một trong hai nút là rỗng. Do đó số lần thực thi thao tác cơ sở không vượt quá tổng độ dài hai đường đi phải của hai cây gốc. Vậy chi phí gian thực tế là:

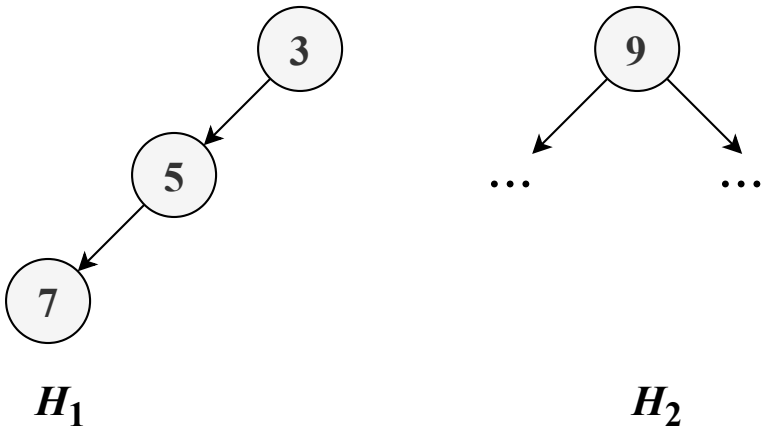
$$T_{\text{actual}} = O(r_1 + r_2)$$

4.2 Độ phức tạp trong trường hợp tốt nhất, xấu nhất

4.2.1 Trường hợp tốt nhất (Best-case)

Thời gian thực hiện trong trường hợp tốt nhất là $\Theta(1)$.

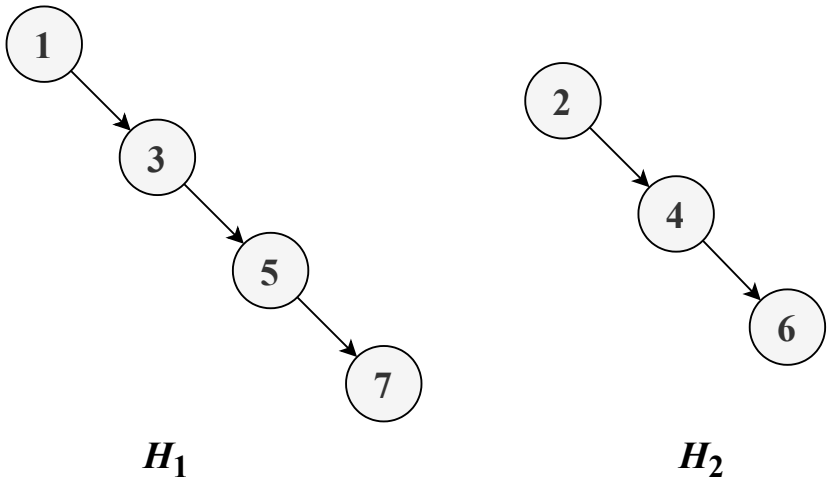
Thật vậy, xét khi một trong hai heap là rỗng hoặc heap có nút gốc có giá trị nhỏ hơn có đường đi phải độ dài 1, phép `merge` thực hiện không quá vài phép gán con trỏ. Chẳng hạn như việc trộn hai cây sau:



Rõ ràng các cây con của (9) không ảnh hưởng đến quá trình trộn. Chi phí chỉ là $\Theta(1)$ thao tác đệ quy và hoán đổi.

4.2.2 Trường hợp xấu nhất (Worst-case)

Thời gian thực hiện trong trường hợp xấu nhất là $\Theta(n)$.
Thật vậy, xét khi cả hai heap đều thỏa mãn tất cả các nút nằm trên đường đi phải (cây lệch phải hoàn toàn), đồng thời số nút trên hai heap chênh lệch nhau không đáng kể và các giá trị của nút trên hai cây là tăng dần xen kẽ, chẳng hạn như cây sau:



Ở mỗi mức đệ quy, ta thấy rằng nút nhỏ hơn trong hai nút cần trộn sẽ thay đổi luân phiên giữa H_1 và H_2 do *tính xen kẽ* đã nêu trên. Vì vậy, tất cả các nút trên đường đi phải của cả hai cây sẽ xuất hiện trong mỗi mức đệ quy. Vì cây lệch phải hoàn toàn nên đây là tất cả các nút của cây. Do đó độ dài đường đi phải của cây thuộc $\Theta(r_1 + r_2) \equiv \Theta(n)$.

4.3 Chi phí Amortized

Độ phức tạp trung bình cho thao tác *merge* nói riêng và các thao tác *insert*, *extract-min* của Skew heap vẫn chưa được xác định rõ ràng. Thậy vậy, một số nguồn tài liệu chỉ ra rằng việc xác định độ dài đường dẫn bên phải kỳ vọng (expected right path length) trong Skew Heap là một vấn đề mở, và khó hơn so với Leftist Heap. Điều này ngụ ý rằng phân tích trường hợp trung bình, liên quan đến thời gian kỳ vọng, vẫn chưa được giải quyết đầy đủ.

Vì vậy, bài báo cáo này chỉ đề cập đến việc phân tích chi phí amortized. Đây là một phương pháp phân tích hiệu suất của các cấu trúc dữ liệu, tập trung vào thời gian trung bình của một thao tác

trong một chuỗi các thao tác. Không giống như phân tích trường hợp xấu nhất, vốn chỉ xem xét chi phí tối đa của một thao tác, phân tích amortized xem xét tổng chi phí của một chuỗi thao tác và chia đều để có được chi phí trung bình trên mỗi thao tác. Điều này đặc biệt hữu ích khi một số thao tác có chi phí cao nhưng được bù đắp bởi nhiều thao tác rẻ hơn, dẫn đến hiệu suất tổng thể tốt hơn.

Đối với Skew Heap, các tài liệu như [2] sử dụng phương pháp thế năng (tiềm năng), định nghĩa một hàm thế năng Φ, ánh xạ trạng thái của cấu trúc dữ liệu thành một số không âm. Chi phí amortized được tính bằng:

$$a_i = c_i + (\Phi_i - \Phi_{i-1}),$$

trong đó c_i là chi phí thực tế, Φ_i là tiềm năng sau thao tác, và Φ_{i-1} là tiềm năng trước thao tác.

4.3.1 Các định nghĩa

Một nút p được xác định là:

- *Nút nặng*: Khi tổng số nút trên cây con bên phải của p lớn hơn tổng số nút trên cây con bên trái, tức là $\text{size}(p.\text{right}) > \text{size}(p.\text{left})$.
- *Nút nhẹ*: Trường hợp ngược lại, khi $\text{size}(p.\text{right}) \leq \text{size}(p.\text{left})$.
- *Nút bên phải*: Là nút con bên phải của nút cha.

Ta cũng định nghĩa *hàm thế năng* $\Phi(H)$ của một heap H là số *nút nặng bên phải* của H. Ý nghĩa của hàm thế năng là mức độ "khó khăn" tiềm tàng trong tương lai: khi trộn, nếu có nhiều nút nặng trên đường đi phải, thì có thể mất nhiều chi phí xử lý.

4.3.2 Bổ đề

Trên đường đi phải của một heap có n nút, số nút nhẹ không vượt quá $\log_2 n$. [2]
Thật vậy, xét một skew heap H có n nút. Trên đường đi phải của H, giả sử có k nút nhẹ. Ta chứng minh rằng $k \leq \log_2 n$. Vì mỗi nút nhẹ p có tính chất $\text{size}(\text{right}(p)) \leq \text{size}(\text{left}(p))$. Vậy nên $\text{size}(p) \geq 2 \cdot \text{size}(\text{right}(p))$.
Gọi n_i là số nút trong cây con gốc tại nút nhẹ thứ i trên đường đi phải, bắt đầu từ nút gốc là $n_0 = n$. Vì tại mỗi nút nhẹ p_i, kích thước cây con phải giảm ít nhất một nửa so với cây con trước đó, ta có: $n_0 \geq 2n_1 \geq 4n_2 \geq \cdots \geq 2^k n_k$. Suy ra: $n = n_0 \geq 2^k n_k \geq 2^k$. Vậy $k \leq \log_2 n$

4.3.3 Đánh giá chi phí amortized

Chứng minh này dựa trên [2]. Dựa vào công thức tính chi phí tại mỗi bước $a_i = c_i + (\Phi_i - \Phi_{i-1})$ ở trên, tổng chi phí amortized là:

$$T_{\text{amortized}} = \sum a_i = \sum [c_i + (\Phi_i - \Phi_{i-1})] = \sum c_i + \Delta\Phi = T_{\text{actual}} + \Delta\Phi$$

Ở đây, $T_{\text{actual}} = \sum a_i$ là tổng chi phí thực tế, $\Delta\Phi = \Phi_{\text{end}} - \Phi_0$ là độ biến thiên thế năng của skew heap sau khi trộn xong so với ban đầu.

Ta tiến hành đánh giá thao tác trộn hai skew heap. Xét hai skew heap H_1 và H_2 cần được trộn.

- Gọi l_1 và l_2 lần lượt là số nút nhẹ **trên đường trộn** của H_1, H_2.
- Gọi h_1 và h_2 lần lượt là số nút nặng **trên đường trộn** của H_1, H_2

Chúng ta sẽ tính một đơn vị cho mỗi nút trên đường trộn. Do đó, thời gian amortized của thao tác trộn là *số nút trên đường trộn* cộng với *sự thay đổi thế năng*.

1. Số nút thực tế trên đường trộn:

Như đã phân tích ở 4.1.2, số nút trên đường trộn của H_1, H_2 là

$$T_{\text{actual}} = (l_1 + h_1) + (l_2 + h_2)$$

Mặt khác, theo bổ đề 4.3.2, tổng số nút nhẹ trên đường đi phải của H_1 và H_2 là $\log_2 n_1$ và $\log_2 n_2$. Vậy:

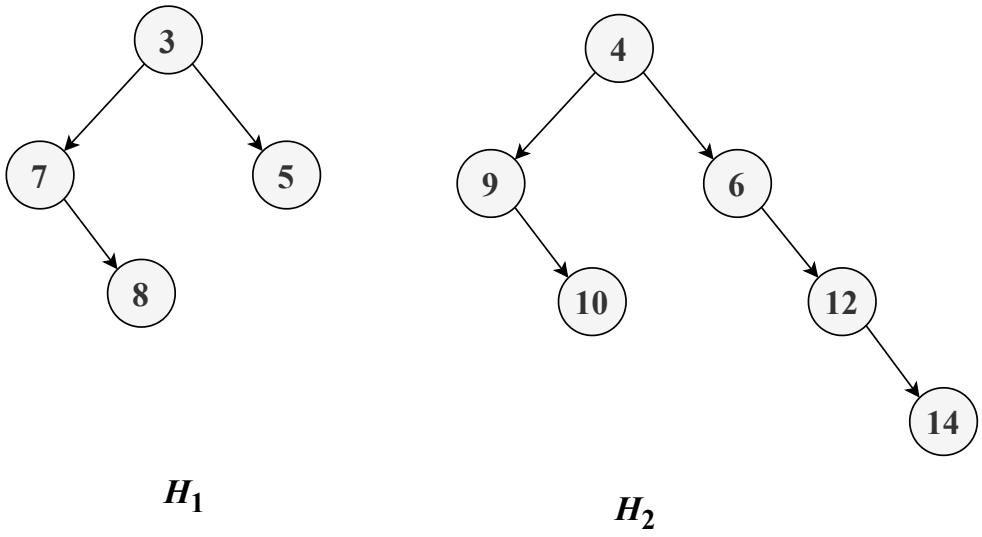
$$T_{\text{actual}} \leq (\log_2 n_1 + h_1) + (\log_2 n_2 + h_2) \leq 2\log_2 n + h_1 + h_2$$

3. Phân tích sự thay đổi thế năng:

Gọi n_1, n_2 là tổng số nút của H_1, H_2. Gọi k là số lượng những nút trở thành nút con phải của các nút trên đường trộn, và biến nút cha trên đường trộn thành nút nặng. Số lượng những nút này chính bằng lượng tăng thêm chênh lệch $\Delta\Phi$, vì vậy làm gia tăng chi phí amortized, nên định nghĩa này phù hợp.

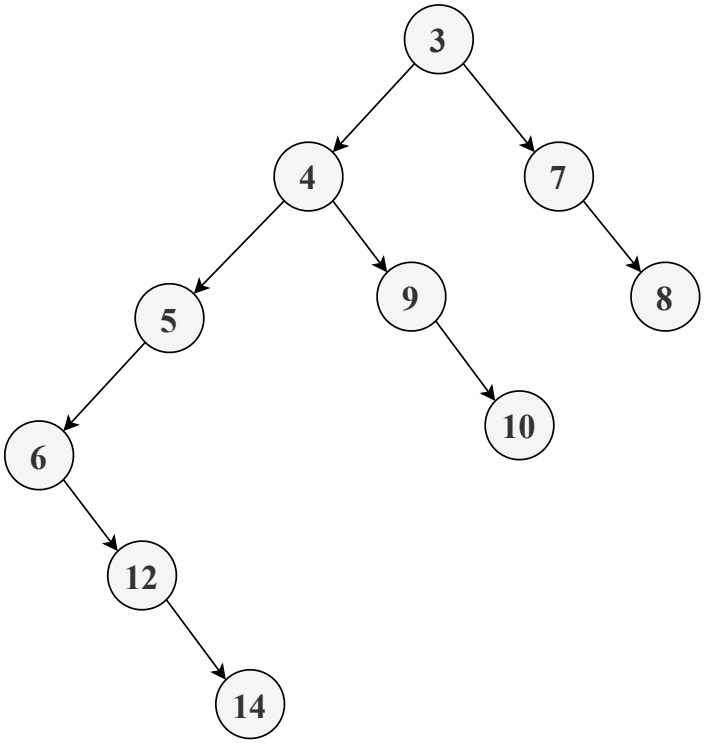
Rõ ràng một nút được tính bởi k tương ứng với một nút nhẹ nào đó trên đường trộn, vì vậy tổng số nút $k \leq \log_2 n$ theo bổ đề 4.3.2.

Độ chênh lệch $\Delta\Phi$ được tính bởi $k - h_1 - h_2$. Chẳng hạn, xét hai cây sau:



H_1 H_2

Ta thấy rằng các nút con nặng bên phải ban đầu là $4, 6, 12, 14$. Tuy nhiên, nếu thực hành quá trình trộn thì các nút nặng trên đường trộn ban đầu chỉ có $4, 6$. Sau khi trộn, ta được heap sau:



Các nút $4, 6$ lúc này vẫn là nút nặng nhưng không còn là nút bên phải nữa. Vì vậy $\Delta\Phi$ giảm đi $h_1 + h_2$. Đồng thời, các nút nặng $7, 9$ là con của các nút nhẹ ban đầu là $3, 4$, sau khi trộn thì chúng

trở thành các nút nặng bên phải, vì vậy $\Delta\Phi$ tăng lên k.

Tiếp theo, vì $k \leq \log_2 n$, nên ta có đánh giá:

$$T_{\text{amortized}} = T_{\text{actual}} + \Delta\Phi \leq (2\log_2 n + h_1 + h_2) + (k - h_1 - h_2) = 2\log_2 n + k \leq 3\log_2 n$$

Vậy chi phí amortized là $3\log_2 n \in O(\log n)$.

4.4 Độ phức tạp của các phép chèn và trích phần tử nhỏ nhất

4.4.1 Thuật toán `insert`

Phép chèn một phần tử vào skew heap có thể được thực hiện bằng cách tạo một heap mới chỉ chứa phần tử đó, rồi trộn với heap ban đầu bằng hàm `merge`.

Algorithm 3 `insert`(H, key)

```
1: function INSERT(H, key)
2:     H_new ← một nút mới với khóa key
3:     return MERGE(H, H_new)
4: end function
```

Vì thao tác tạo một nút mới tốn thời gian hằng số, độ phức tạp của phép chèn chính là độ phức tạp của thuật toán chèn (`merge`), vậy ta có:

`insert` : $O(1)$: trường hợp tốt nhất, $O(\log n)$: amortized, $O(n)$: trường hợp xấu nhất

4.4.2 Thuật toán `extractMin`

Phép loại bỏ nút gốc (nút có khóa nhỏ nhất trong skew heap) được thực hiện bằng cách trộn hai cây con trái và phải của nút gốc lại với nhau.

Algorithm 4 `extractMin`(H, k)

```
1: function EXTRACTMIN((H, k))
2:     k = H.key
3:     return MERGE(H.left, H.right)
4: end function
```

Tương tự như trên, vì thao tác duy nhất là gọi lại hàm `merge` trên hai cây con, nên độ phức tạp

của `extractMin` cũng bằng độ phức tạp của `merge`:

`extractMin` : $O(1)$: trường hợp tốt nhất, $O(\log n)$: amortized, $O(n)$: trường hợp xấu nhất

4.5 Xây dựng Skew Heap từ dãy ban đầu

Trong phần này, ta trình bày hai cách xây dựng Skew Heap từ một dãy số ban đầu, cùng với phân tích độ phức tạp chi tiết:

- *Hướng 1:* Chèn tuần tự từng phần tử bằng `insert`
- *Hướng 2:* Ghép từng cặp heap nhỏ theo phương pháp chia để trị

Cách 1: Chèn tuần tự bằng `insert`

Algorithm 5 `buildSkewHeapInsert`(items)

```
1: function BUILDSKEWHEAPINSERT(items)
2:     H ← null
3:     for all x trong items do
4:         node ← nút mới chứa x
5:         H ← merge(H, node)
6:     end for
7:     return H
8: end function
```

Phân tích chi phí amortized:

Gọi n là số phần tử cần chèn. Ta sẽ tính độ chi phí amortized cho thao tác này. Độ phức tạp amortized của thao tác trộn phần tử thứ i vào heap là $O(\log i)$. Vì vậy, độ phức tạp tổng cộng là

$$T_{\text{amortized}}(n) = \sum_{i=1}^{n} O(\log i) = O\left(\sum_{i=1}^{n} \log i\right)$$

Vì $i \leq n, \forall i$ nên:

$$\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$$

Vì vậy, $\sum_{i=1}^{n} \log i \in O(n \log n)$

Cách 2: Ghép đôi từng cặp

Algorithm 6 `buildSkewHeapPairwise`(items)

```
1: function buildSkewHeapPairwise(items)
2:     Q ← hàng đợi rỗng
3:     for i ← 0 đến items.size() − 1 bước nhảy 2 do
4:         p ← nút mới chứa items[i]
5:         if i + 1 < items.size() then
6:             q ← nút mới chứa items[i + 1]
7:             p ← merge(p, q)
8:         end if
9:         Q.enqueue(p)
10:    end for
11:    while Q.size() > 1 do
12:        sz ← Q.size()
13:        for j ← 1 đến ⌊sz/2⌋ do
14:            p ← Q.dequeue()
15:            q ← Q.dequeue()
16:            Q.enqueue(merge(p, q))
17:        end for
18:        if sz lẻ then
19:            Q.enqueue(Q.dequeue())
20:        end if
21:    end while
22:    return Q.dequeue()
23: end function
```

Phân tích chi phí amortized:

Đây là thuật toán trộn đệ quy từng heap, ý tưởng giống như merge sort. Điểm giống là có $\lfloor n/2 \rfloor$ tầng trộn. Mỗi tầng có $\lfloor \frac{n}{2^{k+1}} \rfloor$ cặp heap cần trộn lại với nhau, và chi phí amortized để trộn hai heap lại là $O(\log 2^k) = O(k)$ (chú ý rằng kích thước của mỗi heap sau khi trộn là không vượt quá 2^k).

Vì vậy, chi phí cho mỗi tầng là $O\left(\frac{n}{2^{k+1}} \cdot k\right)$. Chi phí tổng hợp là:

$$T_{\text{amortized}}(n) = \sum_{k=0}^{\lceil \log_2 n \rceil - 1} O\left(\frac{nk}{2^k}\right)$$

Xét tổng vô hạn $S = \sum_{k=0}^{\infty} \frac{k}{2^k}$. Với $|x| < 1$, ta có khai triển Taylor

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Đạo hàm hai vế ta được:

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

Nhân hai vế với x để có:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Thay $x = \frac{1}{2}$, ta được $S = 2$. Do đó, S tăng dần và hội tụ về 2. Suy ra:

$$T_{\text{amortized}}(n) \leq \sum_{k=0}^{\lceil \log_2 n \rceil - 1} \frac{nk}{2^k} = n \sum_{k=0}^{\lceil \log_2 n \rceil - 1} \frac{k}{2^k} \leq nS \leq 2n$$

Vậy $T_{\text{amortized}}(n) \in O(n)$.

5 So sánh với một số cấu trúc dữ liệu Heap khác

5.1 Giới thiệu chung

Heap là một cấu trúc dữ liệu dạng cây được sử dụng rộng rãi trong khoa học máy tính để quản lý các phần tử theo thứ tự ưu tiên. Heap đảm bảo rằng phần tử có giá trị nhỏ nhất (trong Min-Heap) hoặc lớn nhất (trong Max-Heap) luôn nằm ở gốc, giúp tối ưu hóa các thao tác như tìm kiếm, chèn, và xóa phần tử ưu tiên. Heap là nền tảng cho nhiều thuật toán quan trọng như thuật toán Dijkstra, thuật toán sắp xếp HeapSort, và quản lý hàng đợi ưu tiên.

Sự đa dạng của các loại heap xuất phát từ nhu cầu tối ưu hóa các thao tác cụ thể trong những kịch bản khác nhau. Một số ứng dụng ưu tiên tốc độ chèn và xóa, trong khi những ứng dụng khác cần trộn hai heap một cách hiệu quả. Trong bài viết này, chúng ta sẽ so sánh ba loại heap: **Binary Heap**, **Leftist Heap**, và **Skew Heap**, với trọng tâm là Skew Heap – một loại heap nổi bật nhờ sự đơn giản trong thao tác trộn và tính linh hoạt trong cấu trúc.

5.2 Mô tả từng loại Heap

5.2.1 Binary Heap

Binary Heap là một cây nhị phân hoàn chỉnh, trong đó mỗi nút cha có giá trị nhỏ hơn hoặc bằng các nút con (Min-Heap). Cấu trúc này được lưu trữ dưới dạng mảng, tận dụng tính chất của cây nhị phân hoàn chỉnh để truy cập nhanh: nút cha tại chỉ số i có con tại $2i + 1$ và $2i + 2$, và nút con tại i có cha tại $\lfloor (i - 1)/2 \rfloor$. Binary Heap không yêu cầu thuộc tính phụ.
Nguyên lý hoạt động:

- *Chèn (Insert):* Thêm phần tử vào cuối mảng, sau đó "sift up" bằng cách so sánh và hoán đổi với nút cha nếu vi phạm tính chất heap. Độ phức tạp: $O(\log n)$.
- *Trích xuất phần tử nhỏ nhất (extract Min):* Xóa gốc, thay bằng phần tử cuối mảng, rồi "sift down" để khôi phục tính chất heap. Độ phức tạp: $O(\log n)$.
- *Trộn (Merge):* Không hiệu quả, yêu cầu xây dựng lại heap từ tất cả phần tử, với độ phức tạp $O(n)$.

5.2.2 Leftist Heap

Leftist Heap là một cây nhị phân không hoàn chỉnh, được tối ưu cho thao tác trộn. Mỗi nút có thuộc tính *null path length* (NPL), biểu thị độ dài đường đi ngắn nhất đến một nút lá không có đủ hai con. Thuộc tính này đảm bảo NPL của nút con bên trái luôn lớn hơn hoặc bằng NPL của nút con bên phải, giúp duy trì cân bằng tương đối. Leftist Heap sử dụng con trỏ để lưu trữ.
Nguyên lý hoạt động:

- *Chèn (Insert):* Tạo heap đơn với phần tử mới và trộn với heap hiện tại. Độ phức tạp: $O(\log n)$.
- *Trích xuất phần tử nhỏ nhất (extract Min):* Xóa gốc, trộn hai cây con, và cập nhật NPL. Độ phức tạp: $O(\log n)$.
- *Trộn (Merge):* So sánh hai gốc, chọn gốc nhỏ hơn, trộn cây con bên phải với heap còn lại, và cập nhật NPL. Độ phức tạp: $O(\log n)$.

5.2.3 Skew Heap

Skew Heap là một cây nhị phân tự điều chỉnh, không yêu cầu thuộc tính phụ như NPL, khiến nó đơn giản hơn Leftist Heap. Skew Heap sử dụng con trỏ để lưu trữ, cho phép cấu trúc linh hoạt nhưng có thể mất cân bằng trong một số trường hợp.
Nguyên lý hoạt động:

- *Chèn (Insert):* Tạo heap đơn và trộn với heap hiện tại. Độ phức tạp amortized: $O(\log n)$, xấu nhất: $O(n)$.
- *Trích xuất phần tử nhỏ nhất (extract Min):* Xóa gốc, trộn hai cây con, với độ phức tạp amortized $O(\log n)$, xấu nhất $O(n)$.
- *Trộn (Merge):* Chọn gốc nhỏ hơn, hoán đổi cây con trái/phải của gốc mới, rồi trộn cây con bên phải với heap còn lại. Độ phức tạp amortized: $O(\log n)$, xấu nhất $O(n)$.

5.3 So sánh thao tác

Phần này phân tích chi tiết độ phức tạp thời gian và các đặc điểm của ba thao tác chính: chèn, xóa min, và trộn. Bảng dưới đây cung cấp cái nhìn tổng quan, sau đó là phân tích từng thao tác.

Bảng 3: So sánh độ phức tạp thời gian của các thao tác

Thao tác	**Binary Heap**	**Leftist Heap**	**Skew Heap**
Chèn (Insert)	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
Xóa Min (Delete)	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
trộn (Merge)	$O(n)$	$O(\log n)$	$O(\log n)^*$
Tìm Min (Find Min)	$O(1)$	$O(1)$	$O(1)$

*Trường hợp xấu nhất của Skew Heap có thể là $O(n)$, nhưng amortized là $O(\log n)$.

5.4 Các đặc điểm khác biệt

Binary Heap là lựa chọn tối ưu khi cần các thao tác chèn và xóa phần tử nhỏ nhất một cách nhanh chóng, nhờ sử dụng cấu trúc mảng đơn giản và hiệu quả. Nó đặc biệt phù hợp trong các ứng dụng phổ biến như thuật toán Dijkstra, A*, hay HeapSort – nơi không cần thao tác trộn các heap.
Leftist Heap và Skew Heap được thiết kế để hỗ trợ thao tác *merge* nhanh chóng giữa hai heap. Trong khi Leftist Heap đảm bảo hiệu suất ổn định nhờ có thuộc tính *null path length* (NPL), thì

Skew Heap đơn giản hóa việc cài đặt bằng cách không cần bất kỳ thuộc tính phụ nào, chấp nhận đánh đổi về hiệu suất trong trường hợp xấu nhất.

Dưới đây là một số so sánh cụ thể:

Về Binary Heap:

- Phù hợp với các thuật toán cổ điển như Dijkstra, Prim– nơi chỉ cần thao tác `insert`, `extract-min` nhưng không cần `merge`.
- Dễ triển khai và kiểm soát, thích hợp cho ứng dụng thời gian thực hoặc lập trình thi đấu.
- Cấu trúc mảng giúp truy cập và quản lý bộ nhớ hiệu quả.

Về Leftist Heap:

- Thích hợp trong các ứng dụng nơi thao tác `merge` giữa nhiều heap xảy ra thường xuyên, ví dụ: trình quản lý tiến trình, lập lịch xử lý ưu tiên hoặc thuật toán Huffman.
- Cung cấp hiệu suất ổn định hơn Skew Heap, nhờ đảm bảo cây không bị mất cân bằng nghiêm trọng.
- Yêu cầu thêm công sức để theo dõi thuộc tính *null path length*, nên cài đặt phức tạp hơn.

Về Skew Heap:

- Phù hợp trong môi trường học thuật hoặc các tình huống thực nghiệm, nơi cần thao tác `merge` nhưng không yêu cầu hiệu suất ổn định tuyệt đối.
- Ưu điểm là rất dễ cài đặt, không cần duy trì thông tin phụ như Leftist Heap.
- Tuy nhiên, không phù hợp nếu cần bảo đảm hiệu suất ổn định hoặc nếu dữ liệu có thể dẫn đến cây mất cân bằng nặng.

Tùy thuộc vào nhu cầu cụ thể – ví dụ như hiệu năng ổn định, thao tác trộn nhiều, hay đơn giản trong cài đặt – người dùng có thể lựa chọn cấu trúc heap phù hợp. Nếu chỉ làm việc với một heap duy nhất và thao tác đơn lẻ, Binary Heap là lựa chọn tốt nhất. Nếu cần merge thường xuyên và quan trọng tính ổn định, hãy chọn Leftist Heap. Ngược lại, nếu bạn muốn đơn giản, thử nghiệm hoặc chấp nhận đánh đổi hiệu suất trong một số tình huống, Skew Heap là một ứng viên sáng giá.

6 Tổ chức dự án và Ghi chú lập trình

6.1 Tổ chức dự án

Dự án được tổ chức đơn giản gồm các thành phần sau:

- **Tệp nguồn**
  - `SkewHeap.cpp`: Chứa các hàm cơ bản của Skew Heap (không đếm phép so sánh)
  - `RecordSkewHeap.cpp`: Mở rộng các hàm với biến đếm phép so sánh
  - `Main.cpp`: Điểm vào chương trình, chứa các test case
- **Tệp header**
  - `RecordSkewHeap.h`: Khai báo cấu trúc dữ liệu, prototype hàm và thư viện cần thiết
- **Tệp thực thi**
  - `Main.exe`: Tệp thực thi sau khi biên dịch

6.2 Ghi chú lập trình

6.2.1 Ghi nhận kết quả thực nghiệm

Để đơn giản hóa, chúng tôi định nghĩa struct và các hàm liên quan để lưu thời gian chạy và số phép so sánh:

- *Struct Record*, định nghĩa trong `RecordSkewHeap.h`, là cấu trúc lưu trữ các chỉ số hiệu suất:
  - `comparison`: Biến kiểu `long long` lưu số phép so sánh
  - `time`: Biến kiểu `long long` đo thời gian thực thi (ms)

```
struct Record {
    long long comparison;
    long long time;
};

```

- *Hàm getRecord*, triển khai trong `RecordSkewHeap.cpp`, được thiết kế để đánh giá hiệu suất:
  - Nguyên mẫu hàm:

```
Record getRecord(
    void (*operation)(SkewHeap&, long long&),
    SkewHeap& s
);

```

  - Cách hoạt động: Hàm thực thi thao tác trên Skew Heap và ghi lại số phép so sánh cùng thời gian chạy. Cần truyền vào con trỏ hàm của thao tác cần đo (có kèm biến đếm phép so sánh).

6.2.2 Biên dịch chương trình

Để tránh tràn stack (đặc biệt với thao tác merge đệ quy), sử dụng lệnh biên dịch sau:

```
cd $dir && g++ -std=c++11 --stack=16777216 *.cpp -o skew_heap
```

với `$dir` là đường dẫn tới thư mục dự án.

Tài liệu

[1] Skew heap. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Skew_heap

[2] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps," *SIAM Journal on Computing*, vol. 15, no. 1, 1986. [Online]. Available: https://www.cs.cmu.edu/~sleator/papers/adjusting-heaps.pdf

[3] University of Maryland, "CMSC 341: Data Structures - Project on Skew Heap," https://www.cs.umd.edu/class/fall2023/cmsc341, 2023, accessed: May 20, 2025.

[4] Elixir Community, "Prioqueue: Priority Queue Implementation in Elixir," https://hexdocs.pm/prioqueue, 2022, accessed: May 20, 2025.

[5] Leftist tree. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Leftist_tree

[6] (2023) Skew heap. GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/skew-heap/

[7] D. Galles. (2023) Skew heap visualization. University of San Francisco. [Online]. Available: https://www.cs.usfca.edu/~galles/visualization/SkewHeap.html

A Phụ lục

- Mẫu LaTeX này được cung cấp miễn phí bởi Quan, Tran Hoang tại GitHub và Overleaf với một số chỉnh sửa bởi các tác giả của đồ án này theo Giấy phép GNU General Public License v3.0.
- Template này **không phải** là template chính thức của Khoa Công nghệ thông tin - Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM.