

Skip Lists

Skip Lists

Tác giả: Vũ chipchip Phúc Hoàng

A: "Mày AC bài Z kiểu gì thế? Tao dùng set không được."

B: "Để mà mày, tao code Splay thôi. 400 dòng 20 phút ez gg."

A: "-_-"

Câu chuyện thật tưởng như đùa trên cũng không phải là hiếm gặp. Splay Tree (hay nói rộng hơn, Balanced Binary Search Tree) là một cấu trúc dữ liệu toàn năng có thể giải quyết rất nhiều bài toán, tuy nhiên nó cũng là một thứ đáng khiếp sợ đối với dân competitive programmers vì độ khó code của nó. Trong một contest với áp lực thời gian căng thẳng, chẳng mấy ai dám code Balanced Binary Search Tree và còn đảm bảo code không bị bug nữa. Tuy nhiên, bạn không thể hoàn toàn tránh được nó, vẫn có những bài mà những thư viện có sẵn như `std::set` hay những cấu trúc đơn giản như Segment Tree, Fenwick Tree không thể giải được, và bạn vẫn phải nhờ cậy đến Splay Tree trong nỗi tuyệt vọng.

May mắn thay, dân competitive programmers đã tìm ra cách sử dụng Skip Lists như một sự thay thế cho Balanced Binary Search Tree. Skip Lists với ý tưởng hết sức đơn giản dễ nhớ, cộng thêm với khả năng tùy biến tuyệt vời đã phần nào làm xua tan đi nỗi sợ code khó bug nhiều (trừ khi bạn có trình độ Da xua thượng thừa như nhân vật B trong đoạn hội thoại trên; trong trường hợp đó, bạn có thể bỏ qua bài viết này). Bài viết này sẽ giới thiệu cho các bạn những ý tưởng và cách sử dụng Skip Lists cơ bản nhất.

Bài toán cơ bản

Hãy lập trình một cấu trúc dữ liệu S có thể thực hiện các thao tác sau:

- Chèn một phần tử x vào S .
- Xóa một phần tử x khỏi S .
- Cho một phần tử x , tìm một phần tử gần x nhất trong S .

Các hướng tiếp cận

Ta sẽ xét một số cấu trúc dữ liệu (chưa đề cập đến Skip Lists) sử dụng để giải bài toán cơ bản trên:

- Sorted Array: Biểu diễn S là một mảng các phần tử. Các phần tử trong mảng được sắp xếp theo thứ tự tăng dần.
 - Chèn: Dịch tất cả các phần tử bên phải vị trí cần chèn sang phải một chỉ số, rồi chèn phần tử cần chèn vào vị trí đó.

- ▶ Xóa: Dịch tất cả các phần tử bên phải phần tử bị xóa sang trái một chỉ số.
- ▶ Tìm: Tìm kiếm nhị phân phần tử.
- ▶ Độ phức tạp: $O(N * \log(N))$ với thao tác khởi tạo, $O(N)$ với thao tác chèn/xóa, $O(\log(N))$ với thao tác tìm kiếm.
- ▶ Sorted Linked List: Biểu diễn S là một danh sách liên kết chứa các phần tử. Các phần tử trong danh sách liên kết được sắp xếp theo thứ tự tăng dần.
 - ▶ Chèn: Thay đổi liên kết giữa phần tử cần chèn, phần tử đứng trước, và phần tử đứng sau.
 - ▶ Xóa: Thay đổi liên kết giữa phần tử cần chèn, phần tử đứng trước, và phần tử đứng sau.
 - ▶ Tìm: Duyệt S từ đầu đến cuối.
 - ▶ Độ phức tạp: $O(N)$ với các thao tác chèn/xóa/tìm. $O(N * \log(N))$ với thao tác khởi tạo.
- ▶ Binary Search Tree: Biểu diễn S là một cây nhị phân tìm kiếm chứa các phần tử. Cây này có tính chất tất cả các nút thuộc cây con bên trái đều nhỏ hơn nút cha, và tất cả các nút thuộc cây con bên phải đều lớn hơn nút cha.
 - ▶ Chèn (Nhiều cách làm): Đi từ trên gốc xuống, so sánh phần tử được chèn với hai nút con, nếu có thì đi vào nút con, còn không thì thêm một nút vào cây chứa phần tử cần chèn.
 - ▶ Xóa (Nhiều cách làm): Thay nút bị xóa bằng nút lớn nhất của cây con bên trái (và xóa nút đó đi), nếu không có cây con bên trái thì thay bằng nút nhỏ nhất của cây con bên phải, nếu không có con thì đơn giản là xóa đi.
 - ▶ Tìm: Đi từ trên gốc xuống, so sánh phần tử cần tìm với hai nút con, đi vào cây con chứa nút cần tìm, cứ thế đến khi tìm được.
 - ▶ Độ phức tạp: Với trường hợp trung bình, các thao tác này có độ phức tạp $O(\log(N))$. Còn với trường hợp xấu nhất là $O(N)$.
- ▶ Balanced Binary Search Tree: Giống Binary Search Tree, nhưng cây có thêm cơ chế tự cân bằng để tránh việc cây bị suy biến, luôn giữ chiều cao cây ở mức ổn định (thường là $O(\log(N))$).

Lưu ý: Thao tác chèn và xóa đều phải đi qua thao tác tìm.

Ta nhận thấy mỗi cấu trúc dữ liệu kể trên đều có nhược điểm của nó. Sorted Array có thao tác chèn xóa chậm, Sorted Linked List có thao tác tìm chậm, Binary Search Tree có thể bị suy biến về chiều cao làm cho cả 3 thao tác đều chậm, Balanced Binary Search Tree hoàn hảo nhưng lại quá phức tạp để cài đặt trong giới hạn thời gian của competitive programming.

Từ đó, Skip Lists, một phiên bản nâng cấp của Sorted Linked List, được sử dụng trong competitive programming như một sự thay thế cho Balanced Binary Search Tree. Về tốc độ và bộ nhớ, Skip Lists không thua gì Balanced Binary Search Tree, tuy nhiên lại dễ cài đặt hơn rất nhiều.

Ý tưởng Skip Lists

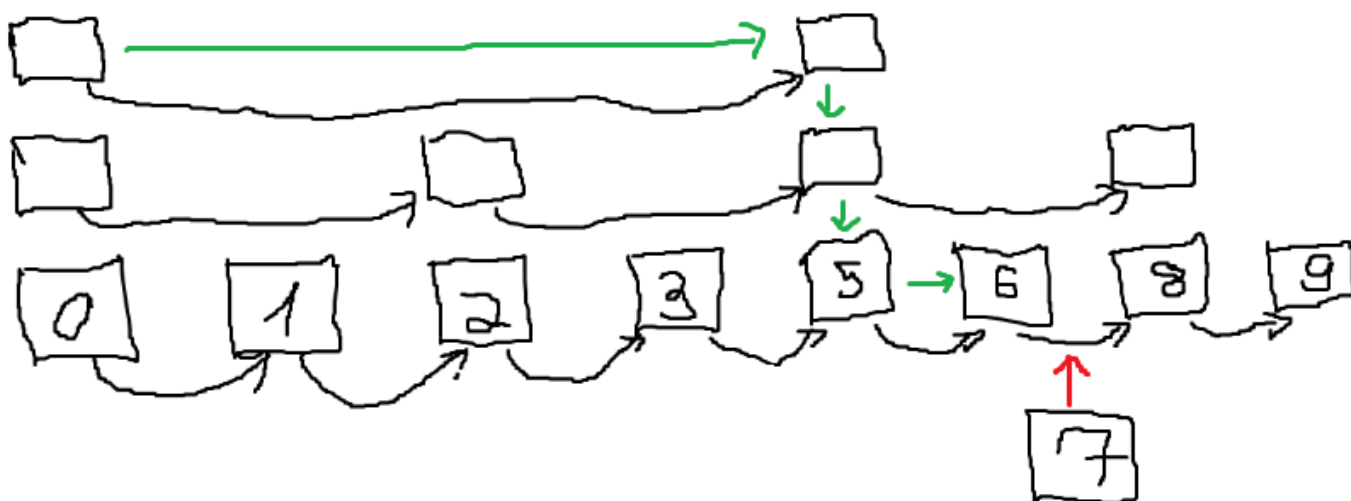
Skip Lists là một phiên bản nâng cấp của Sorted Linked Lists. Ta hãy bắt đầu với một ví dụ về Sorted Linked List chứa 8 số và nghĩ cách cải thiện vấn đề của nó.



Sorted Linked List có ưu điểm lớn khi thao tác chèn xóa chỉ mất $O(1)$ (ta chỉ việc chỉnh sửa liên kết giữa phần tử được chèn/xóa và các phần tử đằng trước/sau). Tuy nhiên thao tác tìm kiếm lại mất $O(N)$ do phải duyệt từ đầu đến cuối.

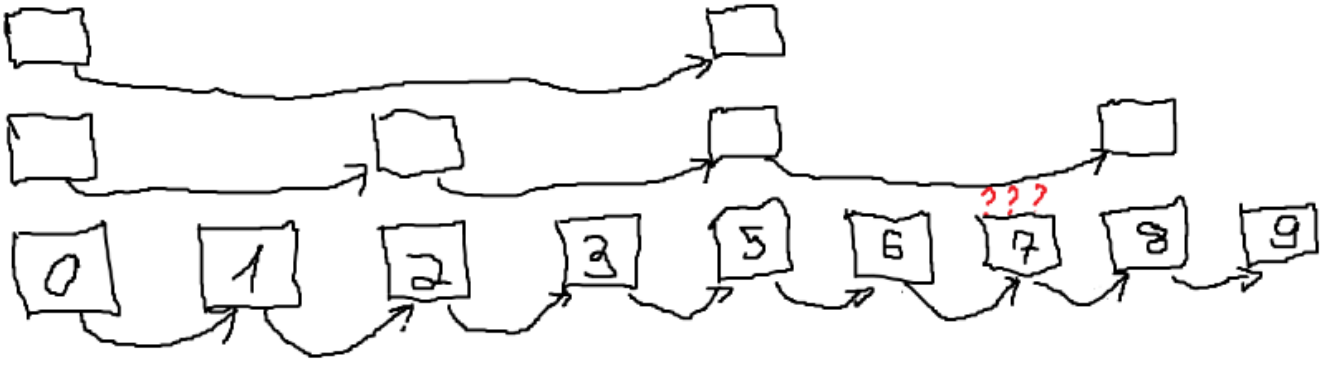


Một ý tưởng để cân bằng điều này là ta thêm nhiều tầng liên kết, cứ lên một tầng số liên kết lại giảm còn một nửa. Khi tìm phần tử, ta sẽ duyệt từ trái sang phải nhưng sẽ nhảy xa hơn nhờ những liên kết trên các tầng cao, khi nào không nhảy được mới xuống tầng thấp hơn. Ý tưởng này khá giống với phương pháp nhảy lên tổ tiên thứ 2^K khi tìm Lowest Common Ancestor (LCA).



Trong hình trên, để tìm số 7, ta sẽ nhảy thẳng từ 0 đến 5 bằng liên kết trên tầng thứ ba, sau đó nhảy từ 5 đến 6 bằng liên kết trên tầng thứ nhất. Ta tìm được 6 là số gần nhất với 7.

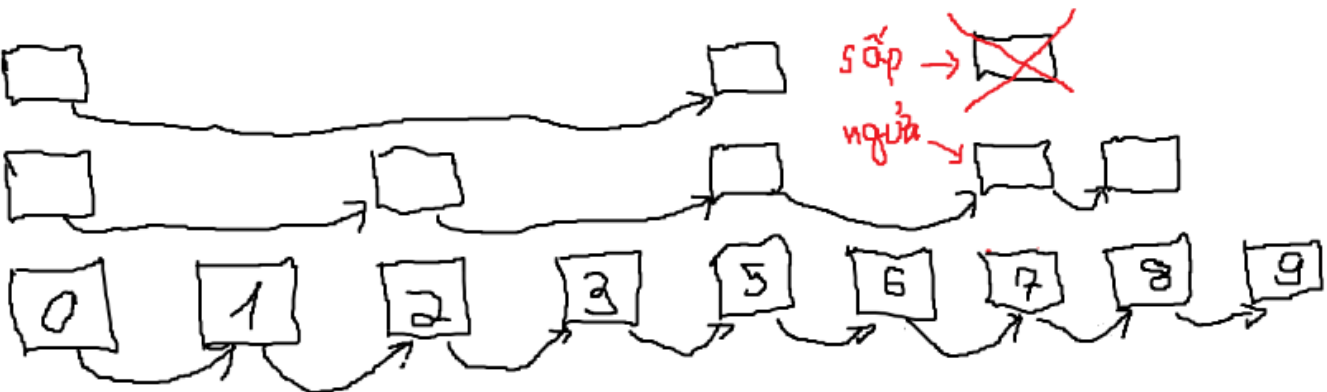
Với cấu trúc này, ta có thể thực hiện thao tác tìm trong $O(\log(N))$. Tuy nhiên việc chèn và xóa một phần tử vào sẽ làm thay đổi cấu trúc này. Chẳng hạn nếu ta chèn số 7:



Như hình trên, cấu trúc của ta không còn "chuẩn", có nghĩa là chính xác tầng thứ nhất liên kết cách 2^0 , tầng thứ hai liên kết cách 2^1 , tầng thứ ba liên kết cách 2^2 , ... Tuy nhiên, với cấu trúc như hình trên vẫn chạy tốt - chỉ có điều ở mỗi tầng ta có thể phải nhảy nhiều hơn một lần (chẳng hạn, muốn tìm số 7, ở tầng thứ nhất ta phải nhảy đến hai lần $5 \leadsto 6 \leadsto 7$).

Từ đó ta có nhận xét sau: Các liên kết trên mỗi tầng không nhất thiết phải chuẩn, tuy nhiên, nếu các độ dài giữa các liên kết xấp xỉ nhau và số liên kết ở tầng trên xấp xỉ bằng nửa số liên kết ở tầng dưới, thuật toán tìm kiếm vẫn chạy tốt và không mất quá nhiều lần nhảy ở mỗi tầng. Ta sẽ duy trì cấu trúc này bằng kĩ thuật tung đồng xu ngẫu nhiên:

Mỗi lần chèn một nút vào, đầu tiên ta xây dựng liên kết ở tầng thứ nhất cho nó. Sau đó ta tung đồng xu, nếu ngửa thì ta xây dựng liên kết ở tầng trên và tiếp tục tung đồng xu, còn nếu sấp ta dừng việc xây dựng liên kết lại.



Đây chính là Skip Lists - một cấu trúc dữ liệu được xây dựng bằng nhiều tầng Sorted Linked List được xây dựng một cách ngẫu nhiên, trong đó tầng cao chứa những bước nhảy dài hơn và tầng thấp chứa những bước nhảy ngắn hơn. Skip Lists cho phép ta thực hiện thao tác tìm kiếm với độ phức tạp xấp xỉ $O(\log(N))$.

So sánh các cấu trúc dữ liệu

	Sorted Array	Sorted Linked List	Binary Search Tree	Red-Black Tree (a Balanced BST)	Skip Lists
Bộ nhớ	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Chèn	$O(N)$	$O(1)$	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
Xóa	$O(N)$	$O(1)$	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
Tìm	$O(\log(N))$	$O(N)$	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
So sánh	* Cực dễ code * Chèn xóa chậm * ĐPT ổn định	* Cực dễ code * Tìm chậm * ĐPT ổn định	* Khó code * Nhanh * ĐPT ngẫu nhiên (sinh test chết được)	* Cực khó code * Nhanh * ĐPT ổn định (tùy loại)	* Dễ code * Nhanh * ĐPT ngẫu nhiên (chưa có cách sinh test chết)

Hướng dẫn chi tiết

Học phải đi đôi với hành. Cách hiểu lý thuyết nhanh nhất là đập ngay vào bài tập. Ta sẽ đi chi tiết vào cách sử dụng Skip Lists để giải bài [CPPSET](#) [☑](#). Bạn hãy đọc đề và ngâm nghĩ một lúc trước khi đọc tiếp bài viết này. Bài giải ở dưới được code bằng ngôn ngữ C++98.

CPPSET, đúng như tên gọi của nó, bạn có thể AC trong một nốt nhạc nếu sử dụng `std::set` của C++, một container đã được code sẵn bằng Red-Black Tree (một loại Balanced Binary Search Tree) để thực hiện bài toán cơ bản nêu ở đầu. Để luyện tập, ta sẽ tự code một cái set "dỏm" bằng Skip Lists.

Trước tiên ta cần xây dựng các struct biểu diễn Skip Lists. Ta sẽ có 3 struct: `SkipLists`, `Column`, `Cell`. `SkipLists` là một danh sách các `Column` liên kết với nhau. `Column` là một cột gồm các `Cell`, biểu diễn cho cột liên kết của một phần tử trong set của ta với các phần tử đứng trước và đứng sau. `Cell` là một liên kết cơ bản nhất trên một tầng của `Column`, chứa hai liên kết đến `Column` đứng trước và đứng sau. Để cho dễ hiểu, bạn hãy xem hình dưới.



```
struct SkipLists {
    static const int MAX_LEVEL = 20; // Giới hạn số tầng, nên chọn một số khoảng
```

```

3      Column *head, *tail; // thêm 2 cột không có giá trị vào đầu và cuối để dễ :
4  };
5
6  struct Column {
7      int value;
8      vector<Cell> cells;
9  };
10
11  struct Cell {
12      Column *previous_column, *next_column; // Mỗi Cell có hai liên kết đến Col
13  };

```

Sau khi đã biết cách biểu diễn dữ liệu, ta sẽ code các hàm cho [SkipLists](#). Set "dòm" của chúng ta sẽ gồm

```

1  struct SkipLists {
2      static const int MAX_LEVEL = 20;
3      Column *head, *tail;
4
5      SkipLists(); // Khởi tạo
6      bool empty(); // Kiểm tra SkipLists có rỗng không
7      Column *lower_bound(int); // Tìm vị trí Column chứa giá trị nhỏ nhất không
8      Column *upper_bound(int); // Tìm vị trí Column chứa giá trị nhỏ nhất lớn h
9      void insert(int); // Chèn một phần tử mang giá trị cho trước vào SkipLists
10     void erase(int); // Xóa một phần tử mang giá trị cho trước khỏi SkipLists
11 };

```

Ta sẽ bắt đầu với constructor [SkipLists\(\)](#). Để khởi tạo [SkipLists](#), ta sẽ tạo ra hai cột [head](#) và [tail](#) có chiều cao là số tầng tối đa, và tại liên kết giữa [head](#) và [tail](#) trên tất cả các [Cell](#).

```

1  SkipLists::SkipLists() {
2      head = new Column;
3      tail = new Column;
4      head->value = 0;
5      tail->value = 0;
6      for(int i = 0; i < MAX_LEVEL; i++) {
7          head->cells.push_back((Cell){NULL, tail});
8          tail->cells.push_back((Cell){head, NULL});
9      }
10 }

```

Với hàm `empty()`, ta chỉ đơn giản kiểm tra liên kết cấp 0 (liên kết trực tiếp) của `head` có nối với `tail` không.

```
1 | bool SkipLists::empty() {
2 |     return head->cells[0].next_column == tail;
3 | }
```

Với hàm `lower_bound()`, ta sẽ đi từ tầng cao nhất đến tầng thấp nhất, chừng nào nhảy về phía trước vẫn vào một phần tử có giá trị nhỏ hơn giá trị cần tìm thì ta cứ nhảy. Sau khi duyệt, ta sẽ đứng ở phần tử lớn nhất có giá trị nhỏ hơn giá trị cần tìm. Ta nhảy trên liên kết cấp 0 một lần nữa để lấy được `lower_bound()`.

```
1 | Column *SkipLists::lower_bound(int value) {
2 |     Column *iter = head;
3 |     for(int level = MAX_LEVEL - 1; level >= 0; level--) {
4 |         while(iter->cells[level].next_column != tail && iter->cells[level].next_column->value < value)
5 |             iter = iter->cells[level].next_column;
6 |     }
7 | }
8 | return iter->cells[0].next_column;
9 | }
```

Hàm `upper_bound()` không khác gì `lower_bound()`, ngoại trừ việc thay dấu `<` thành `<=` lúc so sánh với `value`.

```
1 | Column *SkipLists::upper_bound(int value) {
2 |     Column *iter = head;
3 |     for(int level = MAX_LEVEL - 1; level >= 0; level--) {
4 |         while(iter->cells[level].next_column != tail && iter->cells[level].next_column->value <= value)
5 |             iter = iter->cells[level].next_column;
6 |     }
7 | }
8 | return iter->cells[0].next_column;
9 | }
```

Với hàm `insert()`, ta sẽ chia thành 3 bước sau:

- Sử dụng `lower_bound()` để kiểm tra giá trị đã tồn tại trong `SkipLists` chưa. Nếu đã tồn tại, thoát khỏi hàm.

- ▶ Tạo ra một **Column** mới để chèn vào **SkipLists**. Ta sẽ sử dụng hàm **rand()** để tung đồng xu, xây dựng chiều cao cho **Column** này.
- ▶ Chèn **Column** vào **SkipLists**. Ta duyệt y như trong **lower_bound()** và **upper_bound()**, ở mỗi tầng chèn liên kết với **Column** vào hai cột đằng sau và đằng trước **Column**.

```

1  void SkipLists::insert(int value) {
2      // Kiểm tra value đã tồn tại chưa
3      Column *temp = lower_bound(value);
4      if(temp != tail && temp->value == value) {
5          return;
6      }
7      // Tạo inserted_column là cột chứa value để chèn vào SkipLists
8      Column *inserted_column = new Column;
9      inserted_column->value = value;
10     inserted_column->cells.push_back((Cell){NULL, NULL});
11     // Tung đồng xu tăng chiều cao
12     while(inserted_column->cells.size() < MAX_LEVEL && rand() % 2 == 0) {
13         inserted_column->cells.push_back((Cell){NULL, NULL});
14     }
15     // Duyệt để chèn
16     Column *iter = head;
17     for(int level = MAX_LEVEL - 1; level >= 0; level--) {
18         while(iter->cells[level].next_column != tail && iter->cells[level].next_column->value < value)
19             iter = iter->cells[level].next_column;
20     }
21     if(level < inserted_column->cells.size()) {
22         // Nối iter với inserted_column, nối inserted_column với next_iter
23         Column *next_iter = iter->cells[level].next_column;
24         iter->cells[level].next_column = inserted_column;
25         next_iter->cells[level].previous_column = inserted_column;
26         inserted_column->cells[level].previous_column = iter;
27         inserted_column->cells[level].next_column = next_iter;
28     }
29 }
30 }
```


Với hàm **erase()**, ta sẽ chia thành 3 bước sau:

- ▶ Sử dụng **lower_bound()** để kiểm tra giá trị đã tồn tại trong **SkipLists** chưa. Nếu không tồn tại, thoát khỏi hàm.
- ▶ Xóa cột chứa giá trị cần xóa khỏi **SkipLists** bằng cách nối từng liên kết giữa các **Cell** liền trước và liền sau nó trên từng tầng.
- ▶ Xóa cột chứa giá trị cần xóa để giải phóng bộ nhớ.


```

1  void SkipLists::erase(int value) {
2      // Kiểm tra value đã tồn tại chưa
3      Column *erased_column = lower_bound(value);
4      if(erased_column == tail || erased_column->value != value) {
5          return;
6      }
7      // Duyệt để xóa
8      Column *iter = head;
9      for(int level = MAX_LEVEL - 1; level >= 0; level--) {
10         while(iter->cells[level].next_column != tail && iter->cells[level].next_column->value < value)
11             iter = iter->cells[level].next_column;
12     }
13     if(iter == erased_column) {
14         // Nối previous_iter với next_iter
15         Column *previous_iter = iter->cells[level].previous_column, *next_iter = iter->cells[level].next_column;
16         previous_iter->cells[level].next_column = next_iter;
17         next_iter->cells[level].previous_column = previous_iter;
18     }
19 }
20 delete erased_column;
21 }

```

Với 6 hàm trên, bạn đã có thể mô phỏng một cách đơn giản một cái set "dỏm" để giải bài này. Bạn hãy thử tự làm tiếp và nộp trên SPOJ nhé. Toàn bộ code cho bài CPPSET có thể xem ở [đây](#) .

Mở rộng

- ▶ Ở trên mới là một code Skip Lists đơn giản nhất mô phỏng `std::set` để giải bài CPPSET. Liệu bạn có thể code lại một `std::set` hoàn hảo bằng Skip Lists không? Hãy thử xem!
- ▶ Code trên sử dụng cả liên kết xuôi (`next_column`) và liên kết ngược (`previous_column`) để dễ xử lí. Bạn có thể code lại CPPSET mà không cần sử dụng liên kết ngược không?
- ▶ Khi xây dựng cột để chèn vào Skip Lists, ta sử dụng kĩ thuật tung đồng xu với xác suất 1/2 mỗi mặt để xây dựng chiều cao cột. Tại sao phải là 1/2, liệu có thể là một con số khác không? Bạn hãy thử các con số khác nhau, sử dụng cả phân tích lý thuyết và thực nghiệm, cho thấy độ hiệu quả của các con số khác.
- ▶ Hẳn bạn sẽ thắc mắc dùng Skip Lists làm gì khi nó cũng chỉ để thay `std::set`, mà `std::set` thì có sẵn rồi. Skip Lists có rất nhiều ứng dụng và khả năng tùy biến nâng cao mà sẽ được giới thiệu trong phần 2 của bài viết này, giúp nó làm được những điều `std::set` không thể làm được, đơn giản nhất là tìm phần tử lớn thứ k trong tập hợp. Bạn thử tự nghĩ cách tìm phần tử lớn thứ k trong Skip Lists xem.

Lời kết

Trên đây là những gì cơ bản nhất các bạn có thể biết về Skip Lists, hi vọng các bạn có thể ứng dụng cấu trúc dữ liệu tuyệt vời này một cách hiệu quả trong các contests. Cá nhân mình thấy Skip Lists là một cấu trúc dữ liệu rất hay nhưng ít được sử dụng, competitive programmers Việt Nam chúng ta thường thích dùng Splay Tree hơn mặc dù chẳng mấy ai dám code lúc đi thi... Mình rất mong sau bài viết này mọi người sẽ dùng Skip Lists nhiều hơn.

Những cách sử dụng Skip Lists nâng cao sẽ được giới thiệu trong phần 2, mọi người cùng đón đọc nhé.

Được cung cấp bởi [Wiki.js](#)