Bài: Tổng quan về Priority Queue

Xem bài học trên website để ủng hộ Kteam: Tổng quan về Priority Queue

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage How Kteam nhé!

Dẫn nhập

Các bài toán liên quan đến xử lí xâu là một dạng bài khá phổ biến trong lập trình. Ngày hôm nay, chúng ta sẽ đi tìm hiểu về một cấu trúc dữ liệu dạng cây được dùng để xử lí xâu nhé.

Nội dung

Để có thể tiếp thu bài học này một cách tốt nhất, các bạn nên có những kiến thức cơ bản về:

- Các kiến thức cần thiết để theo dõi khóa học
- Đồ thị và cây
- Sắp xếp
- BFS và DFS

Trong bài học ngày hôm nay, chúng ta sẽ cùng nhau tìm hiểu về:

- Priority Queue

Bài toán đặt ra

Cho *n* ván gỗ có chiều dài lần lượt là $a_1, a_2, ..., a_n$ đơn vị độ dài $0 < a_i < 10^5$. Ta có thể ghép hai ván gỗ có chiều dài a_i và a_j thành một ván gỗ mới có chiều dài là $a_i + a_j$ và tốn chi phí là $a_i + a_j$. Hỏi chi phí nhỏ nhất để ghép *n* ván gỗ đã cho thành một ván gỗ duy nhất là bao nhiêu?

Ví dụ:

Input	Output
6 1 2 8 2 7 6	60

Giải thích ví dụ:

- Đầu tiên ta ghép hai ván gỗ thứ nhất và thứ hai. Khi này, các ván gỗ còn lại có chiều dài là *3 8 2 7 6*, chi phí hiện tại là 3
- Trong các ván gỗ sau khi ghép, ta ghép ván gỗ thứ nhất và thứ ba. Khi này, các ván gỗ còn lại có chiều dài là *5, 8, 7, 6*, chi phí hiện tại là 8
- Trong các ván gỗ sau khi ghép, ta ghép ván gỗ thứ nhất và thứ tư. Khi này, các ván gỗ còn lại có chiều dài là *11, 8, 7*, chi phí hiện tại là 19
- Trong các ván gỗ sau khi ghép, ta ghép ván gỗ thứ hai và thứ ba. Khi này, các ván gỗ còn lại có chiều dài là *11, 15*, chi phí hiện tại là 34
- Ta ghép hai ván gỗ còn lại. Khi này, chỉ còn một ván gỗ cuối cùng có chiều dài 26. Chi phí cuối cùng là 60

Ý tưởng

Để có thể giải quyết bài toán này, chúng ta sẽ phải có nhận xét về chi phí cuối cùng.

Hãy giả sử chúng ta sẽ luôn ghép hai ván gỗ ở vị trí đầu tiên. Hãy xem khi này chi phí cuối cùng sẽ là bao nhiêu nhé!

- Các ván gỗ ban đầu có chiều dài là $a_1, a_2, a_3, ..., a_n$. Ta ghép hai ván gỗ đầu tiên. Khi này, các ván gỗ có chiều dài $a_1 + a_2, a_3, a_4, ..., a_n$ và chi phí là $a_1 + a_2$

- Ta tiếp tục ghép hai ván gỗ ở vị trí đầu tiên. Khi này, các ván gỗ có chiều dài $a_1 + a_2 + a_3, a_4, ..., a_n$ và chi phí là $(a_1 + a_2) + (a_1 + a_2 + a_3) = 2 \times (a_1 + a_2) + a_3$
- Ta tiếp tục ghép hai ván gỗ ở vị trí đầu tiên. Khi này, các ván gỗ có chiều dài$a_1 + a_2 + a_3 + a_4, ..., a_n$ và chi phí là $2 \times (a_1 + a_2) + a_3 + (a_1 + a_2 + a_3 + a_4) = 3 \times (a_1 + a_2) + 2 \times a_3 + a_4$
- Làm tương tự cho phần còn lại, ta sẽ có chi phí cuối cùng là $(n - 1) \times (a_1 + a_2) + (n - 2) \times a_3 + (n - 3) \times a_4 + \ ... \ + 2 \times a_{n-1} + a_n$.

Các bạn có nhận xét gì về mối quan hệ giữa thứ tự các ván gỗ được chọn và kết quả cuối cùng? Ta thấy các ván gỗ được chọn càng sớm thì hệ số nhân chiều dài của chúng trong tổng cuối cùng càng lớn.

Kể cả khi các bạn ghép ngẫu nhiên chứ không nhất thiết phải chọn hai ván gỗ ở đầu thì nhận xét này vẫn đúng. Các bạn có thể thử ghép ngẫu nhiên với một số lượng ván gỗ nhỏ để thấy được nhận xét này.

Do đó, ta sẽ có ý tưởng sau: Với các ván gỗ còn lại, ta sẽ luôn ưu tiên ghép hai ván gỗ có chiều dài nhỏ nhất.

Vấn đề còn lại là làm sao để chọn ra hai ván gỗ có chiều dài nhỏ nhất hay chính là tìm ra hai phần tử trong mảng có giá trị nhỏ nhất.

Nếu như chỉ code đơn thuần bằng mảng và biến thì code sẽ khá dài dòng và mất thời gian. Do đó, chúng ta cần có cấu trúc dữ liệu để hỗ trợ. Ở đây, cấu trúc dữ liệu chúng ta cần dùng sẽ là **Priority Queue**.

Tổng quan về Priority Queue

Khái niệm

Priority Queue là một cấu trúc dữ liệu mà các phần tử được quản lý sẽ có "độ ưu tiên" khác nhau gắn với từng phần tử. Phần tử có thứ tự ưu tiên cao hơn trong Priority Queue sẽ được xếp lên trước và truy vấn trước.

Đơn giản hơn, Priority Queue là một cấu trúc cho phép nó tự động sắp xếp các phần tử của nó.

Một Priority Queue sẽ có các chức năng cơ bản sau:

- Thêm một phần tử vào tập quản lý
- Xóa bỏ phần tử có ưu tiên cao nhất
- Lấy ra phần tử có ưu tiên cao nhất

Trong bài học này, mình sẽ không hướng dẫn các bạn xây dựng Priority Queue bằng tay mà sẽ sử dụng thư viện có sẵn của C++. Có hai lý do cho việc này:

- Xây dựng Priority Queue bằng tay tốn thời gian và khá phức tạp
- Các bạn sẽ không bao giờ cần tùy biến code của Priority Queue mà chỉ tập trung vào chức năng

---

Khai báo

Thông thường để thêm Priority Queue vào chương trình, chúng ta sẽ thêm thư viện như sau:

#include<priority_queue>

Tuy nhiên, ở trong suốt khóa học này mình sẽ sử dụng header sau:

#include<bits/stdc++.h>

Header này sẽ giúp chúng ta thêm tất cả các thư viện về các cấu trúc dữ liệu mà chúng ta sẽ học trong khóa học này.

Ta sẽ khai báo Priority Queue như sau:

> **priority_queue <{kiểu dữ liệu}, {class container}, {class compare}> {tên priority_queue};**

Trong đó:

- Container là một đối tượng cho phép lưu trữ các đối tượng khác. Ví dụ như vector, stack, queue, deque, ... đều là các class container. Mặc định, vector là container của priority_queue
- Compare thường sẽ là một phép toán trong thư viện *functional*. Một số phép toán trong thư viện này là less, greater, less_equal, ... Phép toán mặc định trong priority_queue là less. Đối với các kiểu dữ liệu mặc định của C++, các phép toán này đã được xây dựng sẵn. Tuy nhiên, với các class hay struct mà người dùng tạo, ta sẽ phải tự định nghĩa các phép toán này. Mình sẽ giới thiệu cách định nghĩa ở phần sau.

Ví dụ:

Ta có thể khai báo như sau:

- priority_queue<int> pq;

Khi này, phần tử lấy ra ở đầu priority_queue sẽ là số nguyên lớn nhất.

Hoặc ta có thể khai báo:

- priority_queue<int, vector<int>, greater<int>> pq;

Khi này, phép toán so sánh là greater nên phần tử lấy ra ở đầu priority_queue là số nguyên nhỏ nhất.

> Chú ý: Nếu phép toán là less thì phần tử lớn nhất sẽ truy vấn trước, nếu phép toán là greater thì phần tử nhỏ nhất sẽ truy vấn trước. Đừng nhầm lẫn nhé!

Các phương thức cơ bản

Priority Queue trong C++ sẽ có một số phương thức cơ bản sau:

- **push**: Thêm phần tử vào priority_queue
- **pop**: Loại bỏ phần tử đầu tiên (có độ ưu tiên cao nhất) trong priority_queue
- **top**: Trả về giá trị là phần tử đầu tiên (có độ ưu tiên cao nhất) trong priority_queue
- **size**: Trả về số nguyên là số phần tử (kích thước) của priority_queue
- **empty**: Trả về giá trị bool, true nếu priority_queue rỗng, false nếu priority_queue không rỗng

Ví dụ:

C++:

```
#include<bits/stdc++.h>
using namespace std;

typedef long long ll;

const int MaxN = 1 + 1e5;

priority_queue<int> pq;

int main(){
    pq.push(1);
    pq.push(5);
    pq.push(3);
    pq.push(4);

    cout << "Phan tu dau tien trong priority queue la: " << pq.top() << endl;

    pq.pop();

    cout << "Phan tu dau tien trong priority queue la: " << pq.top() << endl;
    cout << "Kich thuoc priority queue la: " << pq.size() << endl;

    return 0;
}
```

Chạy đoạn code trên ta thu được kết quả:

```
Phan tu dau tien trong priority queue la: 5
Loai bo phan tu dau tien trong priority queue
Phan tu dau tien trong priority queue la: 4
Kich thuoc priority queue la: 3
```

Xây dựng các phép toán

Ở bài Các kiến thức cần thiết để theo dõi khóa học, mình đã giới thiệu về cách định nghĩa các toán tử cho một class trong đó có toán tử so sánh. Nếu các bạn muốn sử dụng phép toán less trong priority_queue, các bạn sẽ cần xây dựng toán tử < cho class. Tương tự, nếu muốn sử dụng phép toán greater, các bạn sẽ cần xây dựng toán tử > cho class. Các phép toán khác các bạn có thể tìm hiểu thêm tuy nhiên đây là hai phép toán được sử dụng phổ biến nhất.

Ví dụ, ta có một class thể hiện phân số như sau:

C++:

```
#include<bits/stdc++.h>
using namespace std;

class Fraction{
    public:
        int x; // Tử số
        int y; // Mẫu số

        Fraction(int _x = 0, int _y = 0){
            x = _x;
            y = _y;
        }
};

int main(){

    return 0;
}
```

Khi này, để có thể đưa class Fraction vào priority_queue và sử dụng phép toán less, ta sẽ cần toán tử < như sau

C++:

```
#include<bits/stdc++.h>
using namespace std;

class Fraction{
    public:
        int x; // Tử số
        int y; // Mẫu số

        Fraction(int _x = 0, int _y = 0){
            x = _x;
            y = _y;
        }

        bool operator < (const Fraction &op) const {
            return x * op.y < y * op.x;
        }
};

priority_queue<Fraction> pq;

int main(){
    pq.push(Fraction(1, 2));
    pq.push(Fraction(1, 3));
    pq.push(Fraction(2, 3));

    cout << pq.top().x << "/" << pq.top().y << endl;
    // Output: 2/3

    return 0;
}
```

Giả sử ta muốn sử dụng phép toán greater, ta sẽ định nghĩa toán tử > như sau

C++:

```
#include<bits/stdc++.h>
using namespace std;

class Fraction{
    public:
        int x; // Tử số
        int y; // Mẫu số

        Fraction(int _x = 0, int _y = 0){
            x = _x;
            y = _y;
        }

        bool operator > (const Fraction &op) const {
            return x * op.y > y * op.x;
        }
};

priority_queue<Fraction, vector<Fraction>, greater<Fraction> > pq;

int main(){
    pq.push(Fraction(1, 2));
    pq.push(Fraction(1, 3));
    pq.push(Fraction(2, 3));

    cout << pq.top().x << "/" << pq.top().y << endl;
    // Output: 1/3

    return 0;
}
```

Độ phức tạp

Các phương thức pop, push của priority_queue sẽ đều tốn độ phức tạp thời gian $O(logn)$. Các phương thức khác tốn độ phức tạp thời gian $O(1)$.

Kết luận

- Qua bài này chúng ta đã nắm về **Priority Queue**
- Bài sau chúng ta sẽ tìm hiểu về **Disjoint Sets**
- Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thử thách – Không ngại khó**".