Persistent Data Structures

Persistent Data Structures

Tác giả: Nguyễn *RR* Thành Trung

1. Mở đầu

Persistent Data Structures là những cấu trúc dữ liệu được dùng khi chúng ta cần có **toàn bộ lịch sử** của các thay đổi trên 1 cấu trúc dữ liệu (CTDL). (Chú ý rằng từ **persistent** còn được dùng trong **persistent storage** với một nghĩa hoàn toàn khác).

Xét bài toán ví dụ:

- Cho một dãy A gồm N phần tử.
- Có 2 loại truy vấn:
  1. Update: Gán $A_i = v$
  2. Query: Tìm $max(A_i, \ldots, A_j)$ tại thời điểm sau phép gán thứ K.

Nếu không có đoạn **tại thời điểm sau phép gán thứ K**, bài toán là 1 bài cơ bản trên **Interval Tree**. Đoạn **tại thời điểm sau phép gán thứ K** buộc chúng ta phải lưu lại các thông tin về lịch sử cập nhật CTDL - việc này được giải quyết bằng các Persistent Data Structures.

Gọi trạng thái của CTDL tại một thời điểm là 1 **version** của CTDL đó. Một cách cụ thể hơn, persistent data structures cho phép chúng ta:

- Truy vấn trên một version cũ của CTDL
- Cập nhật dữ liệu trong version mới nhất của CTDL, bằng cách tạo thêm 1 version mới của CTDL.

Trong một số cách cài đặt, Persistent Data Structures còn có thể cho phép thay version hiện tại của CTDL thành một version trong quá khứ (phần 2 mô tả phương pháp cài đặt có thể thực hiện được thao tác này).

Cần hiểu rằng Persistent Data Structures không phải là một loại CTDL mới. Nó là một số kĩ năng tổng quát giúp thêm thông tin về lịch sử thay đổi vào CTDL thông thường một cách hiệu quả. Ví dụ:

- IT + Persistent $\rightarrow$ Persistent IT
- BIT + Persistent $\rightarrow$ Persistent BIT

Tại sao lại là **một cách hiệu quả**? Bởi vì ta hoàn toàn có thể có một Persistent Data Structures bằng cách trâu bò: khi cập nhật, ta tạo một bản sao hoàn toàn mới của CTDL, thay đổi một số dữ liệu trên nó và lưu lại. Như vậy ta luôn có được một thuật toán với độ phức tạp $O(Q \cdot N \cdot T)$ và bộ nhớ $O(Q \cdot N)$, với Q là số thao tác cần thực hiện, và N là độ lớn của CTDL, và T là thời gian để thực hiện thao tác trên CTDL.

Trong các phần dưới đây, mình sẽ trình bày về 2 kĩ thuật thông thường của Persistent Data Structures.

2. Persistent IT

Tư tưởng

Quay trở lại bài toán. Chúng ta biết rằng mỗi thao tác update trên IT chỉ mất $O(\log N)$. Điều này tương đương với việc mỗi thao tác update chỉ làm thay đổi $O(\log N)$ nút trên cây. Như vậy ta hoàn toàn có thể lưu lại tất cả các thay đổi trên tất cả các nút trong $O(Q \log N)$.

Từ đó, ta rút ra được một tư tưởng cài đặt thuật toán:

- Với mỗi thao tác Update, ta tạo thêm một số nút mới trên IT. Để không phải sinh thêm các nút không bị thay đổi, một nút ở version mới có thể có con là một nút ở vesion cũ.
  Chú ý: Mỗi thao tác Update luôn thay đổi một đường đi từ gốc đến một nút lá, nên không có trường hợp một nút ở version cũ có con là một nút ở version mới hơn. (Nếu thao tác Update là Update 1 đoạn, các nút bị thay đổi không còn là một đường đi nữa, nhưng nhận xét này vẫn đúng).
- Khi thực hiện thao tác Query trên version t, ta chỉ cần thực hiện Query trên nút gốc ở version t.
  Tư tưởng này còn được gọi là **Path Copy** trong các tài liệu tiếng Anh.

Cài đặt:

```
struct Node {
    int left, right;    // ID of left child & right child
    long long ln;       // Max value of node
    Node() {}
    Node(long long ln, int left, int right) : ln(ln), left(left), right(right)
} it[11000111];         // Each node has a position in this array, called ID
int nNode;

int ver[MN];            // ID of root in each version

// Update max value of a node
inline void refine(int cur) {
    it[cur].ln = max(it[it[cur].left].ln, it[it[cur].right].ln);
}

// Update a range, and return new ID of node
int update(int l, int r, int u, int x, int oldId) {
    if (l == r) {
        ++nNode;
        it[nNode] = Node(x, 0, 0);
        return nNode;
    }

    int mid = (l + r) >> 1;
    int cur = ++nNode;
```

```
    if (u <= mid) {
        it[cur].left = update(l, mid, u, x, it[oldId].left);
        it[cur].right = it[oldId].right;
        refine(cur);
    }
    else {
        it[cur].left = it[oldId].left;
        it[cur].right = update(mid+1, r, u, x, it[oldId].right);
        refine(cur);
    }

    return cur;
}

// Get max of range. Same as usual IT
int get(int nodeId, int l, int r, int u, int v) {
    if (v < l || r < u) return -1;
    if (u <= l && r <= v) return it[nodeId].ln;

    int mid = (l + r) >> 1;
    return max(get(it[nodeId].left, l, mid, u, v), get(it[nodeId].right, mid+1
}


// When update:
    ++nVer;
    ver[nVer] = update(1, n, u, x, ver[nVer-1]);

// When query:
    res = get(ver[t], 1, n, u, v);
```

Giải thích:

- Ban đầu, ta có một mảng it, lưu tất cả các nút sẽ được sinh ra của IT. Mỗi nút gồm có
  - Chỉ số của con trái, index của con phải (2 biến `left` và `right`)
  - Giá trị lớn nhất của các số trong khoảng mà nút quản lý (ở version khi nút đó được tạo ra): biến `ln`
- Ta lưu thêm chỉ số của các nút gốc ở các version khác nhau vào một mảng `ver`
- Hàm `update` :
  - Tạo ra các nút mới, và trỏ đến các con ở version cũ hoặc version mới, tùy theo các con có bị thay đổi hay không
  - Trả lại index của nút mới được tạo
- Hàm `get` :
  - Trả lại max của một đoạn được quản lý bởi nút `nodeId`

Phân tích

- Cách cài đặt Persistent Data Structures trong mục này rất hiệu quả. Nó hoàn toàn không làm tăng thêm độ phức tạp (persistent IT có độ phức tạp mỗi thao tác là $O(\log N)$), và bộ nhớ cần thêm là tối ưu: $O(Q \log N)$.
- Tuy nhiên, cách cài đặt này không dễ áp dụng với những CTDL khác. Chẳng hạn sẽ rất khó để cài đúng BIT với phương pháp này. Ở mục tiếp theo, mình sẽ trình bày một phương pháp cài đặt khác có thể dùng cho BIT, tuy nhiên có độ phức tạp lớn hơn.

3. Persistent BIT

Tư tưởng:

Tại mỗi nút của BIT, thay vì lưu một giá trị, ta lưu lại tất cả các cặp (version, giá trị) ở nút đó.

- Thao tác update rất đơn giản: chỉ cần thêm một cặp (version, giá trị) vào các nút tương ứng.
- Với thao tác query tại version t, trên một nút, ta cần tìm cặp (version, giá trị) mới nhất vào trước thời điểm t - việc tìm kiếm này có thể được thực hiện bằng tìm kiếm nhị phân.

Cách cài đặt này được gọi là **Fat Node** trong các tài liệu tiếng Anh.

Cài đặt:

Code BIT trích từ bài IPSC 2011 - Grid Surveillance:

```
#define _(x) (x & (-(x)))

// Persistent BIT
vector< pair<int,int> > bit[4100][4100];

// Add val to cell (x, y) at time = time
void update(int x, int y, int val, int time) {
    for(int u = x; u <= 4096; u += _(u))
        for(int v = y; v <= 4096; v += _(v)) {
            if (bit[u][v].empty()) {
                bit[u][v].push_back(make_pair(time, val));
            }
            else {
                int cur = bit[u][v][bit[u][v].size()-1].second;
                bit[u][v].push_back(make_pair(time, cur + val));
            }
        }
}

// Get the sum of square (1,1) --> (x, y) at time = time
int get(int time, int x, int y) {
    int res = 0;
    for(int u = x; u > 0; u -= _(u))
```

```
        for(int v = y; v > 0; v -= _(v)) {
            if (bit[u][v].empty()) {
            }
            else if (bit[u][v][bit[u][v].size()-1].first <= time) {
                res += bit[u][v][bit[u][v].size()-1].second;
            }
            else {
                int pos = upper_bound(bit[u][v].begin(), bit[u][v].end(), make
                        - bit[u][v].begin() - 1;
                if (pos >= 0)
                    res += bit[u][v][pos].second;
            }
        }
    return res;
}
```

Phân tích:

- Độ phức tạp cho mỗi thao tác update không thay đổi (ví dụ với BIT, vẫn là $O(\log N)$). Nhưng độ phức tạp cho mỗi thao tác query bị tăng lên $\log N$ (ví dụ với BIT, độ phức tạp cho mỗi thao tác là $O(\log^2(N))$ thay vì $O(\log N)$).
- Tuy nhiên, cách cài đặt này tổng quát hơn, dễ dàng được áp dụng cho nhiều CTDL khác nhau, ví dụ cả BIT và IT.

4. Retroactive Data Structures

Một lớp CTDL khác tương đối giống với **Persistent Data Structures**, nhưng có tính ứng dụng thực tế cao hơn là **Retroactive Data Structures**:

"Retroactive Data Structures là loại CTDL cho phép thực hiện thay đổi với một dãy các thao tác đã được thực hiện trên dữ liệu. Ví dụ: Thay đổi một thao tác đã được thực hiện trong quá khứ".

Cả Retroactive Data Structures & Persistent Data Structures đều quan tâm đến trục thời gian, tuy nhiên điểm khác nhau nằm ở chỗ cách xử lý trục thời gian của 2 CTDL này như thế nào:

- Với Persistent Data Structures, tất cả các version được lưu lại, và bạn không thể nào thay đổi một version trong quá khứ (điều duy nhất bạn có thể làm là tạo ra một version mới từ 1 version cũ và thực hiện thay đổi trên version mới này).
- Với Retroactive Data Structures, bạn có thể thực hiện thay đổi trên một version cũ. Thay đổi trên một version cũ này ảnh hưởng đến tất cả các version sau nó.

Sự khác biệt về cách xử lý trục thời gian khiến cho Retroactive Data Structures có rất nhiều ứng dụng trên thực tế - trái ngược hẳn với Persistent Data Structures chỉ thường được thấy ở trong các kỳ thi. Một vài ứng dụng quan trọng của Retroactive Data Structures gồm có:

- **Error Correction**: Giả sử một dữ liệu bị nhập sai, làm ảnh hưởng đến tất cả các thao tác sau đấy. Retroactive DS cho phép sửa dữ liệu nhập sai và ảnh hưởng (tích cực) đến tất cả các thao tác sau đó.

- **Bad data**: Gần giống với Error Correction, nhưng dữ liệu sai bị xóa đi thay vì được sửa lại.
- **Recovery**: Giả sử một lỗi của phần cứng làm một số dữ liệu không được đọc. Retroactive DS cho phép đọc lại những dữ liệu này và thay đổi tất cả các thao tác được thực hiện sau đó.

Trên thực tế, Retroactive Data Structures còn đang dừng lại ở việc là một khái niệm, chứ chưa có một phương pháp cài đặt nào hiệu quả. Các bạn nếu muốn tìm hiểu có thể nghiên cứu thêm về cơ chế rollback trong database hoặc tìm kiếm thêm về Retroactive Data Structures.

Bài tập áp dụng

- SPOJ - COT
- SPOJ - MKTHNUM
- Codechef - QUERY
- Codechef - SORTING
- Codeforces - Round 140 Div 1 - E
- Codeforces - Round 265 Div 1 - E
- IPSC 2011 - Grid Surveillance