CHUYÊN ĐỀ:

CẤU TRÚC DỮ LIỆU TREAP

GIỚI THIỆU

Trong khoa học máy tính, TREAP và cây tìm kiếm nhị phân ngẫu nhiên hóa là hai dạng cấu trúc dữ liệu cây tìm kiếm nhị phân liên quan chặt chẽ đến nhau. Chúng lưu trữ một tập hợp các khóa và cho phép chèn, xóa, và tìm kiếm khóa. Sau bất kì một dãy các thao tác chèn và xóa nào, hình dạng cây là một biến ngẫu nhiên có cùng phân phối với hình dạng của cây nhị phân ngẫu nhiên. Đặc biệt, với xác suất cao, chiều cao của cây là lôgarit của số khóa, nên mỗi thao tác tìm kiếm, chèn, hoặc xóa chỉ tốn thời gian lôgarit.

TREAP được phát minh bởi Cecilia R. Aragon và Raimund Seidel năm 1989. Tên của nó là một từ kết hợp của tree (cây) và heap (đống). Nó là một cây Descartes trong đó mỗi khóa được gắn một độ ưu tiên ngẫu nhiên. Như với mọi cây nhị phân khác, thứ tự duyệt trung thứ tự của các nút là thứ tự tăng dần của các khóa. Cấu trúc của cây được quyết định bởi tính chất đống max: độ ưu tiên của mọi nút khác lá đều lớn hơn hoặc bằng độ ưu tiên của mọi nút trong cây con của nó. Do đó, cũng như với mọi cây Descartes nói chung, nút gốc là nút có độ ưu tiên lớn nhất.

Một cách thức mô tả tương đương của TREAP là có thể tạo nó bằng cách chèn các nút theo thứ tự độ ưu tiên giảm dần mà không thực hiện bất kì phép cân bằng nào. Do đó, nếu các độ ưu tiên là ngẫu nhiên (từ một phân phối trên một tập hợp đủ lớn sao cho khả năng hai nút có cùng độ ưu tiên là rất thấp) thì hình dạng TREAP có cùng phân phối với hình dạng cây tìm kiếm nhị phân ngẫu nhiên (cây tìm kiếm nhị phân tạo bởi việc chèn các nút theo thứ tự ngẫu nhiên mà không thực hiện bất kì phép cân bằng nào). Do cây tìm kiếm nhị phân ngẫu nhiên có chiều cao lôgarit với xác suất cao, điều đó cũng đúng cho TREAP.

Cụ thể, TREAP cho phép thực hiện các thao tác sau:

+ Tìm kiếm một khóa cho trước bằng cách sử dụng thuật toán thông thường cho cây nhị phân và bỏ qua độ ưu tiên.

+ Chèn khóa mới x vào TREAP bằng cách đầu tiên sinh một trọng số ngẫu nhiên y cho x. Tìm kiếm x trên cây và chèn một nút mới chứa x vào vị trí trống mới tìm được. Sau đó thực hiện các phép quay cây để phục hồi tính chất đống: chừng nào x còn chưa là nút gốc và có độ ưu tiên lớn hơn nút cha là z, thì thực hiện phép quay cây trên cạnh nối x và z.

+ Xóa khóa x trong TREAP như sau: nếu x là nút lá, thì xóa nó. Nếu x có một nút con z, xóa x và đặt z vào làm nút con của nút cha của x (hoặc đặt z làm gốc của cây nếu x là gốc cũ). Cuối cùng, nếu x có hai nút con, đổi chỗ x và nút nhỏ nhất trong cây con phải của x là z,

và thực hiện một trong hai trường hợp trên cho x ở vị trí mới. Trong trường hợp cuối, z có thể vi phạm tính chất đống nên cần thực hiện các phép quay cây để phục hồi tính chất đống.

+ Chia TREAP thành hai TREAP, một chứa các khóa nhỏ hơn x, và một chứa các khóa lớn hơn x bằng cách chèn x với độ ưu tiên cao hơn tất cả các khóa vào TREAP. Sau khi chèn xong, x trở thành nút gốc, và mọi khóa nhỏ hơn x nằm ở TREAP con bên trái và mọi khóa lớn hơn x nằm ở TREAP con bên phải. Chi phí của thao tác này đúng bằng một lần chèn.

+ Hợp hai TREAP lại làm một (giả sử mọi khóa của TREAP thứ nhất nhỏ hơn mọi khóa của TREAP thứ hai) như sau. Chèn x sao cho x lớn hơn mọi khóa ở TREAP thứ nhất, và nhỏ hơn mọi khóa của TREAP thứ hai với độ ưu tiên của x nhỏ hơn độ ưu tiên của mọi khóa. Sau khi chèn xong, x là một nút lá nên có thể xóa dễ dàng. Kết quả thu được là một TREAP đúng bằng hai TREAP cũ hợp lại.

Aragon và Seidel cũng khuyên nên gán cho các nút hay được truy cập trọng số lớn hơn. Thay đổi này làm cây không còn ngẫu nhiên mà khiến cho các nút hay được truy cập dễ nằm ở gần gốc của cây hơn, khiến cho việc tìm chúng nhanh hơn.

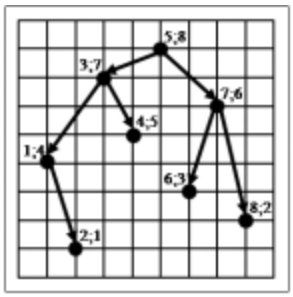
Blelloch và Reid-Miller mô tả một ứng dụng của TREAP để lưu trữ các tập hợp các đối tượng và hỗ trợ các thao tác hợp tập hợp, giao tập hợp, và phần bù tương đối bằng cách dùng một TREAP để biểu diễn mỗi tập hợp. Naor và Nissim mô tả một ứng dụng khác để lưu trữ chứng nhận khóa công khai trong mật mã khóa công khai.

CẤU TRÚC DỮ LIỆU TREAP

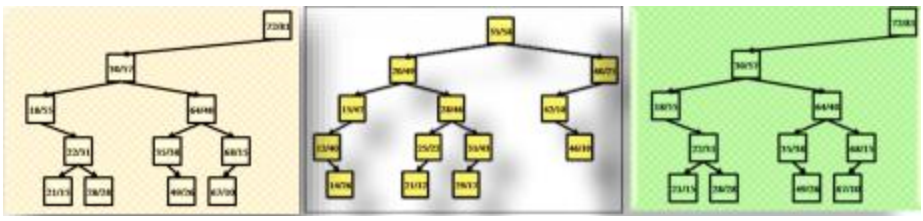
1. Cấu trúc dữ liệu TREAP

1.1- Khái niệm:

TREAP là cấu trúc dữ liệu kết hợp giữa cây tìm kiếm nhị phân với vun đống nhị phân, vì vậy tên gọi là sự kết hợp tên của hai cấu trúc trên (TREAP = Tree + Heap). Cấu trúc dữ liệu này lưu trữ các cặp (x, y) dưới dạng tạo thành cây tìm kiếm nhị phân theo x và là vun đống nhị phân theo y. Giả thiết tất cả các x và tất cả các y khác nhau. Khi đó nếu trong cấu trúc có nút lưu trữ cặp (x_0, y_0) thì ở các nút của cây con trái của nút này đều có x nhỏ hơn x_0, ở các nút của cây con phải của nút này đều có x lớn hơn x_0, ngoài ra, ở cả cây con phải lẫn cây con trái giá trị y ở các nút đều nhỏ hơn y_0. TREAP do Siedel và Argon đề xuất năm 1996. Trong phần lớn các ứng dụng TREAP, x là khóa (key), đồng thời cũng là giá trị lưu trữ trong cấu trúc, còn y là độ ưu tiên (priority). Nếu như không có độ ưu tiên y, với bộ dữ liệu x cho trước ta sẽ có một tập các cây tìm kiếm nhị phân, trong số đó một số cây suy biến thành nhánh và làm chậm một cách đáng kết quả trình tìm kiếm. Việc đưa mức độ ưu tiên vào sẽ cho phép xác định cây một cách đơn trị, không phụ thuộc vào trình tự bổ sung các nút. Ta có thể xây dựng cây nhị phân không suy biến, đảm bảo thời gian tìm kiếm trung bình là O(logn), bởi vì khi bổ sung nút với độ ưu tiên ngẫu nhiên xác xuất nhận được cây có độ cao lớn hơn ⌈4log2n⌉ là vô cùng bé. Cặp giá trị (x, y) có thể xét như tọa độ một điểm trên mặt phẳng. Khi đó TREAP là một cây đồ thị phẳng (các cạnh không giao nhau) nối các điểm trên mặt phẳng. Vì vậy TREAP còn được gọi là Cây Đề các (Carsian Tree).



Ví dụ: Một số cấu trúc TREAP:



1.2- Các phép xử lý với TREAP

TREAP cho phép tổ chức xử lý như với cây tìm kiếm nhị phân bình thường, nhưng trong phần lớn các trường hợp thời gian thực hiện mỗi phép xử lý đều là O(log*n*):

♣ **void insert(T1 key, T2 value)** – bổ sung cặp giá trị (**key**, **value**) vào TREAP,

♣ **void remove(T1 key)** – xóa các nút có khóa bằng **key** khỏi TREAP,

♣ **T2 find(T1 key)** – tìm giá trị xác định bởi khóa **key**. Cấu trúc TREAP cũng cho phép dễ dàng thực hiện các phép chọn và cập nhật theo nhóm:

♠ **T2 query(T1 key1, T1 key2)** – xác định giá trị của một hàm (tổng, tích, max,...)với các phần tử trong khoảng [**key1**, **key2**],

♠ **voi modify(T1 key1, T1 key2)** – cập nhật giá trị (*tăng thêm một lượng, nhân với một số, xác lập một thuộc tính, . . .*) với tất cả các phần tử trong khoảng [**key1**, **key2**]. Tồn tại nhiều phép xử lý phức tạp hơn như hợp nhất các TREAP, phân chia một TREAP thành vài TREAPs riêng biệt, . . . đều được thực hiện dựa trên các phép xử lý cơ sở sẽ xét chi tiết dưới đây:

1.2.1. Các phép xử lý cơ sở

Khai báo cấu trúc phần tử của TREAP:

```
struct TreapNode
{
    int k, p, v;
    TreapNode *l, *r;
    TreapNode(int key, int value)
    {
        k = key;
        v = value;
        p = rand();
        l = r = NULL;
    }
};
```

Merge (Hợp nhất TREAPs)

Xét TREAP **A** có gốc là **a** và TREAP **B** có gốc là **b**. Giả thiết các khóa của **A** đều nhỏ hơn khóa bất kỳ của **B** (hoặc không lớn hơn nếu tổ chức TREAP với các phần tử có thể có khóa giống nhau). Cần tạo TREAP **T** mới từ các phần tử của **A** và **B**, trả về con trỏ chỉ tới gốc của **T**:

```
TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p)
    {
        a->r = merge(a->r, b);
        return a;
    } else
       {
          b->l = merge(a, b->l);
          return b;
       }
}
```

Giả thiết khóa của các phần tử 2 cây đều khác nhau. Khi đó gốc của **T** sẽ là phần tử của **A** hoặc **B** có độ ưu tiên lớn nhất. Dễ dàng thấy rằng phần tử đó là **a** hoặc **b**. Nếu **a->p** > **b->p** thì **a** là gốc của cây hợp nhất. Các phần tử của **B** đều có khóa lớn hơn **a->k** vì vậy chúng sẽ được gắn vào cây con **a->r**. Như vậy bài toán ban đầu được dẫn về bài toán hợp nhất 2 cây **a->r** với **B**. Không khó để nhận thấy là quy trình xử lý trên có thể thể hiện bằng sơ đồ đệ quy. Đệ quy kết thúc khi một trong 2 cây là rỗng. Trường hợp gốc là b được xử lý tương tự. ***split (Tách cây)*** Cho số **k** và cây **T** với gốc là **t**. Yêu cầu tạo 2 cây: cây **A** gồm các phần tử của **T** có khóa nhỏ hơn **k** và cây **B** – chứa các phần tử còn lại của **T**. Hàm trả về giá trị 2 con trỏ chỉ tới gốc các cây. Nếu **t->k** < **k** thì t thuộc **A**, trong trường hợp ngược lại – **t** thuộc **B**. Giả thiết **t** thuộc **A**, khi đó **t** sẽ là gốc của **A**. Khi đó các phần tử của cây con **t->l** sẽ là một phần của **A** vì có khóa nhỏ hơn **t->k** và do đó – nhỏ hơn **k**. Với cây con **t->r** tình hình phức tạp hơn: cây con có thể chứa các phần tử có khóa nhỏ hơn **k** lẫn các phần tử có khóa không nhỏ hơn **k**. Ta có thể xử lý theo sơ đồ đệ quy: tách cây con **t->r** theo **k**, nhận được 2 cây **A'** và **B'**. TREAP **A'** được đặt thế chỗ cho **t->r**, khi đó **T** sẽ chỉ chứa các phần tử có khóa nhỏ hơn **k**. **B'** chứa các phần tử có khóa không nhỏ hơn **k**. Như vậy kết quả tách cây sẽ là **T** và **B'**. Trường hợp **t** thuộc **B** – xử lý tương tự. Đệ quy kết thúc khi **T** trở thành rỗng .

```
void split(TreapNode *t, int k, TreapNode *&a, TreapNode *&b)
{
    if (!t)
        a = b = NULL;
      else if (t->k < k)
       {
          split(t->r, k, t->r, b);
          a = t;
       } else
           {
             split(t->l, k, a, t->l);
             b = t;
           }
}
```

Tìm kiếm trên TREAP

Việc tìm kiếm trên TREAP được thực hiện theo đúng sơ đồ tìm kiếm trên cây nhị phân. Dễ dàng thấy rằng độ phức tạp của quá trình tìm kiếm không vượt quá **O(logn).** Như vậy TREAP có thể coi như cây tìm kiếm nhị phân cân bằng (tuy bản thân TREAP có cấu trúc phức tạp hơn). Luôn luôn giả thiết rằng con trỏ **TREAPNode *t** chỉ tới gốc của TREAP.

Bổ sung phần tử mới

Xét việc bổ sung phần tử với khóa key và giá trị value vào cây **T**. Phần tử này có thể xem như một TREAP **Tn** hoàn chỉnh bao gồm một nút. Như vậy có thể tiến hành hợp nhất hai cây. Tuy vậy, đầu tiên phải xử lý sơ bộ, chia T thành 2 phần: **T1** chứa các phần tử có khóa nhỏ hơn key và **T2** – chứa các phần tử còn lại, sau đó thực hiện hai phép hợp nhất và hoàn thành việc bổ sung phần tử mới vào cây.

```
void insert(int key, int value)
{
    TreapNode *tn = new TreapNode(key, value), *t1, *t2;
    split(t, key, t1, t2);
    t = merge(t1, tn);
    t = merge(t, t2);
}
```

Xóa phần tử của cây

Giả thiết phải loại bỏ khỏi **T** tất cả các phần tử có khóa bằng **key**. Tách **T** thành 2 cây **T1** và **T2** theo khóa **key**, **T1** chứa các phần tử có khóa nhỏ hơn **key**, **T2** chứa các phần tử còn lại. Tách **T2** theo khóa **key+1**, nhánh phải (**T3**) của cây nhận được chứa các phần tử có khóa lớn hơn key, còn trong **T2** chỉ có các phần tử có khóa bằng **key**. Hợp nhất **T1** và **T3** ta có kết quả cần tìm. Nếu cần giải phóng bộ nhớ của các phần tử bị xóa ta cần gọi hàm **dispose**:

```
void dispose(TreapNode *n)
{
    if (n == NULL)
        return;
    dispose(n->l);
    dispose(n->r);
    delete n;
}
```

```
void remove(int key)
{
    TreapNode *t1, *t2, *t3;
    split(t, key, t1, t2);
    split(t2, key + 1, t2, t3);
    t = merge(t1, t3);
    dispose(t2);
}
```

Tìm phần tử

Việc tìm kiếm một phần tử với khóa cho trước trên TREAP có thể thực hiện theo đúng sơ đồ tìm kiếm trên cây nhị phân bình thường. Mặt khác có thể sử dụng các công cụ **merge()** và **split()** đã xây dựng ở trên để tìm kiếm. Phần của cây có thể chứa phần tử cần tìm với khóa cho trước có thể xác định bằng hai lát cắt. Sau khi trích hoặc xử lý kết quả cần khôi phục lại cây (bằng hai phép ghép). Cách xử lý này đặc biệt có hiệu quả với các truy vấn xử lý nhóm phần tử. Nếu phần tử cần tìm không tồn tại kết quả sẽ là 0.

```
int find(int key)
{
    int res = 0;
    TreapNode *t1, *t2, *t3;
    split(t, key, t1, t2);
    split(t2, key + 1, t2, t3);
    if (t2 != NULL)
        res = t2->v;
    t1 = merge(t1, t2);
    t = merge(t1, t3);
    return res;
}
```

Truy vấn phần tử theo nhóm

Các truy vấn đòi hỏi phải làm việc với nhiều phần tử như tính tổng hoặc tích, đếm số lượng, tìm giá trị max hoặc min, . . . có thể dễ dàng thực hiện trên TREAP. ***Xử lý truy vấn trên toàn bộ cây*** Giả thiết cần tính tổng các phần tử, đồng thời xác định số lượng phần tử. Trong cấu trúc phần tử cần thay đổi: bổ sung thêm 2 trường **cnt** và **sum** để lưu số lượng và tổng của các phần tử thuộc cây con có gốc là nút đang xét. Trong khai báo cấu trúc các trường này cần được khởi tạo là 1 và giá trị của phần tử.

```
struct TreapNode
{
    int k, p, v;
    int sum, cnt;
    TreapNode *l, *r;
    TreapNode(int key, int value)
    {
        k = key;
        v = value;
        p = rand();
        l = r = NULL;
        sum = value;
        cnt = 1;
    }
};
```

Ngoài ra, cần có thêm các hàm **get()**, **sum()** và **update()** phục vụ cập nhật các trường này trong quá trình xây dựng cây.

```
int getSum(TreapNode *n)
{
    return n != NULL ? n->sum : 0;
}
```

```
int getCnt(TreapNode *n)
{
    return n != NULL ? n->cnt : 0;
}
```

```
void update(TreapNode *n)
{
    if (n == NULL) return;
    n->sum = v + getSum(n->l) + getSum(n->r);
    n->cnt = 1 + getCnt(n->l) + getCnt(n->r);
}
```

Các phép **merge()** và **split()** cũng cần được thay đổi thích hợp để phục vụ cho các trường mới đưa vào. Với mỗi nhánh cây nhận được cần gọi hàm **update()** để chỉnh lý giá trị các trường **cnt** và **sum**.

```
TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p) {
        a->r = merge(a->r, b);
        update(a);
        return a;
    } else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

```
void split(TreapNode *t, int k,
           TreapNode *&a, TreapNode *&b)
{
    if (!t)
        a = b = NULL;
    else if (t->k < k) {
        split(t->r, k, t->r, b);
        a = t;
    } else {
        split(t->l, k, a, t->l);
        b = t;
    }
    update(a); update(b);
}
```

Nếu cần tìm số lượng hay tổng các phần tử chỉ cần truy nhập tới trường tương ứng của phần tử gốc.

Xử lý truy vấn trên đoạn

Giả thiết cần đếm số phần tử hoặc tổng các phần tử có khóa nằm trên đoạn [**k1**, **k2**]. Bằng hai lát cắt (tương tự như khi tìm hoặc xóa phần tử) ta có thể xác định nhánh cây chứa các phần tử cần tìm. Việc xác định giá trị quan tâm được thực hiện như đã nêu ở mục trên.

```
int rangeSum(int k1, int k2)
{
    TreapNode *t1, *t2, *t3;
    split(t, k1, t1, t2);
    split(t2, k2 + 1, t2, t3);
    int s = getSum(t2);
    t1 = merge(t1, t2);
    t = merge(t1, t3);
    return s;
}
```

Cập nhật theo nhóm phần tử

Các phép cập nhật theo nhóm phần tử có thể là thay các giá trị cũ bằng một giá trị mới, nhân một số với các giá trị ở nút, bổ sung thêm một thuộc tính nào đó, . . . Dưới đây ta sẽ xét việc cộng thêm một số vào các giá trị của nút trên toàn cây hoặc trên một đoạn. Các loại cập nhật khác được thực hiện theo kiểu tương tự.

Cập nhật trên toàn cây

Xét việc cộng thêm một số vào các giá trị ở tất cả các nút của cây. Cần bổ sung vào cấu trúc **TREAPNode** trường **add** lưu giá trị cần tăng. Trong khai báo cần gán giá trị đầu là 0 cho trường mới này.

```
struct TreapNode
{
    int k, p, v;
    int sum, cnt, add;
    TreapNode *l, *r;
    TreapNode(int key, int value)
    {
        k = key;
        v = value;
        p = rand();
        l = r = NULL;
        sum = value;
        cnt = 1;
        add = 0;
    }
};
```

Khi đó giá trị thực lưu ở nút gốc **t** của cây **T:**

Không đơn thuần là **t->v** mà sẽ là **t->v+t->add**. Giá trị trường **sum:** thay **t->sum** bằng **t->sum+t->add*t->cnt**.

Để các hàm **merge()** và **split()** hoạt động đúng cần đảm bảo việc chuyển giao giá trị thực cho các tham số. Sự tồn tại của trường add có thể phá vỡ tính đúng đắn. Ví dụ **t->add** bằng 10, còn **t->l->add** bằng 0, thì sau khi ngắt **t->l** khỏi **t** thông tin xác định là giá trị các phần tử của cây phải được cộng thêm 10 sẽ bị mất. Vì vậy trước khi thực hiện các phép **merge()** và **split()** cần chuyển giao thông tin này cho các nhánh sẽ nhận được bằng hàm **down()**.

```
void down(TreapNode *n)
{
    if (n == NULL)
        return;
    n->v += n->add;
    if (n->l != NULL)
        n->l->add += n->add;
    if (n->r != NULL)
        n->r->add += n->add;
    n->add = 0;
}
```

Các hàm **merge()** và **split()** sẽ có dạng:

```
TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p)
    {
        down(a);
        a->r = merge(a->r, b);
        update(a);
        return a;
    } else
      {
        down(b);
        b->l = merge(a, b->l);
        update(b);
        return b;
      }
}
```

```
TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p)
    {
        down(a);
        a->r = merge(a->r, b);
        update(a);
        return a;
    } else
      {
        down(b);
        b->l = merge(a, b->l);
        update(b);
        return b;
      }
}
```

```
void split(TreapNode *t, int k,
           TreapNode *&a, TreapNode *&b)
{
    down(t);
    if (!t)
        a = b = NULL;
    else if (t->k < k)
     {
        split(t->r, k, t->r, b);
        a = t;
     } else
       {
         split(t->l, k, a, t->l);
         b = t;
       }
    update(a);
    update(b);
}
```

Với TREAP được tổ chức như trên, nếu muốn cộng thêm một giá trị nào đó vào tất cả các phần tử của cây thì cần gán giá trị này cho trường **add** của phần tử gốc.

Cập nhật trên đoạn

Nếu như đã thiết kế TREAP với khả năng cập nhật toàn bộ cây thì việc cập nhật trên đoạn [**k1**, **k2**] cũng sẽ dễ dàng thực hiện, tương tự như việc xử lý truy vấn trên đoạn.

```
void rangeAdd(int k1, int k2, int addVal)
{
    TreapNode *t1, *t2, *t3;
    split(t, k1, t1, t2);
    split(t2, k2 + 1, t2, t3);
    if (t2 != NULL)
        t2->add += addVal;
    t1 = merge(t1, t2);
    t  = merge(t1, t3);
}
```

1.3-Phạm vi ứng dụng của TREAP

Cấu trúc TREAP tương đối dễ lập trình, vì vậy nó là công cụ hữu hiệu giải nhiều loại bài toán đòi hỏi xử lý truy vấn ở các dạng:

Các bài toán đòi hỏi tổ chức cây đủ cân bằng để tìm kiếm và cập nhật (ví dụ, tổ chức tập hợp, tổ chức từ điển).

Các loại bài toán xử lý truy vấn trên một phạm vi nào đó, các bài toán cập nhật trên đoạn. Khác với cây quản lý đoạn, TREAP cho phép xử lý truy vấn trên các tập thay đổi,

Tính toán các số liệu thống kê với độ phức tạp O(logn), Phục vụ xây dựng một cấu trúc dữ liệu mạnh hơn – TREAP với khóa ẩn.

1.4-Hạn chế

Tốn bộ nhớ: mỗi nút của cây phải lưu trữ khá nhiều trường dữ liệu trung gian cho các loại truy vấn khác nhau,

Việc thay đổi cấu trúc nút sẽ kéo theo sự thay đổi trong các phép xử lý cơ bản,điều nà làm tăng độ phức tạp lập trình.

1.5-TREAP với khóa ẩn

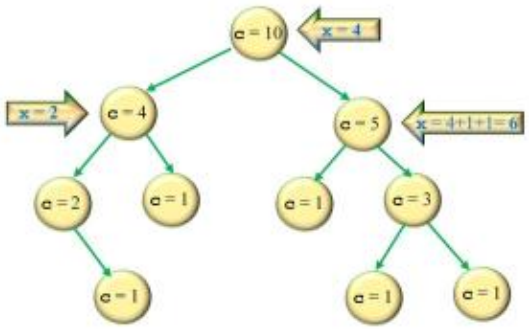
Cấu trúc dữ liệu vector trong C/C++ cho phép tổ chức mảng động với khả năng bổ sung phần tử mới vào cuối mảng hoặc xóa phần tử cuối trong danh sách. Cấu trúc này cũng cho phép xác định hoặc cập nhật giá trị của phần tử theo vị trí của nó. Giả thiết ta cần một cấu trúc dữ liệu như vậy cùng với các khả năng mới để có thể bổ sung phần tử vào vị trí bất kỳ hoặc xóa một phần tử cùng với việc đánh số tự động lại các phần tử của mảng.

Cấu trúc dữ liệu với các tính chất đã nêu có thể xây dựng dựa trên cấu trúc TREAP và được gọi là *TREAP với khóa ẩn* (*TREAP with implicit key*).

Như đã nói ở trên, TREAP là cấu trúc dữ liệu kết hợp giữa cây nhị phân và vun đống nhị phân. Trong TREAP khóa **x** phục vụ tổ chức cây nhị phân được lưu trữ tường minh. Nhưng người ta cũng có thể vòng tránh việc lưu trữ khóa tường minh bằng cách lấy *số lượng phần tử ở nhánh trái* của phần tử đang xét làm *khóa* tổ chức cây nhị phân. Như vậy, trong mỗi phần tử chỉ lưu trữ tường minh độ ưu tiên **y**, còn khóa sẽ là *số thứ tự* của phần tử trong cây *trừ* 1.

Vấn đề phải giải quyết ở đây là khi bổ sung hay loại bỏ phần tử, số thứ tự các phần tử trong cây có thể bị thay đổi. Nếu không có cách tiếp cận hợp lý, độ phức tạp của các phép bổ sung, loại bỏ sẽ có độ phức tạp O(*n*) và làm mất ưu việt của cây tìm kiếm nhị phân.

Lối thoát khỏi tình huống rắc rối trên là khá đơn giản. Thay vì lưu trữ tường minh **x** người ta lưu trữ **c** – *số lượng nút của cây con* có gốc là phần tử đang xét (bao gồm cả nút gốc). Đây là một đại lượng có miền xác định đủ nhỏ và dễ dàng dẫn xuất khi cần thiết. Lưu ý là các phép xử lý trong TREAP đều thực hiện theo chiều từ trên xuống dưới. Nếu trong quá trình duyệt, đi từ gốc tới một đỉnh nào đó ta cộng số lượng nút và tăng thêm 1 của các nhánh trái bị bỏ qua thì khi tới nút cần xét sẽ có trong tay khóa của phần tử đó.



Các phép xử lý cấu trúc

Hai phép xử lý cấu trúc cây: **merge()** – hợp nhất 2 cây, trong đó ở một cây các khóa **x** có giá trị nhỏ hơn giá trị khóa ở cây kia, **split()** – tách một cây thành 2 cây, trong đó ở một cây các khóa **x** có giá trị nhỏ hơn giá trị khóa ở cây kia.

Trong TREAP với khóa ẩn tham số cho hàm merge là gốc của 2 cây bất kỳ cần hợp nhất: **merge(root1,root)**. Lời gọi hàm **split** sẽ là **split(root,t)** xác định việc tách cây được thực hiện sao cho nhánh trái có **t** nút.

Xuất phát từ nút gốc, xét yêu cầu cắt **k** nút của một cây. Giả thiết nhánh trái của cây có **l** nút và nhánh phải – **r** nút. Có 2 trường hợp xẩy ra:

♣ **l** ≥ **k**: khi đó cần gọi đệ quy split từ nút con trái của gốc với cùng tham số k¸ đồng thời biến phần phải của kết quả thành con trái của gốc, còn gốc của cây đang xử lý sẽ là phần phải của kết quả,

♣ **l** < **k**: xử lý tương tự như trên nếu đổi vai trò của trái và phải, split được gọi đệ quy từ nút con phải với tham số **k-l-1**, phần trái của kết quả là nút con trái phải, còn gốc – thuộc phần trái kết quả.

Để dễ hiểu tư tưởng thuật toán, các bước xử lý được trình bày ở đoạn mã giả sau (giải thuật xử lý chính xác trên C++ với đầy đủ các hàm sẽ được xét ở phần cuối).

```
<Treap, Treap> split(Treap t, int k)
  int l = t.left.size
  if l >= k
    <t1, t2> = split(t.left, k)
    t.left = t2
    update(v)
    r = v
    return <t1, t2>
  else
    <t1, t2> = split(t.right, k - l - 1)
    t.right = t1
    update(v)
    l = v
    return <t1, t2>
```

Trong quá trình hợp nhất không có sử dụng giá trị khóa x vì vậy hàm **merge** đượ xử lý như ở TREAP bình thường.

Sau mỗi thao tác xử lý đỉnh con cần ghi vào trường lưu trữ c tổng giá trị tương ứng của các nút con cộng thêm 1.

```
void update(Treap t)
  t.size = 1 + t.left.size + t.right.size
```

Ứng dụng

Bổ sung phần tử mới vào bất kỳ vị trí nào trong cây đang xét đồng thời duy trì mọi tính chất của cây ban đầu,

Di chuyển đoạn các phần tử của mảng sang nơi mới bất kỳ, Thực hiện các phép xử lý đối với nhóm phần tử, kích thước nhóm thay đổi và được xác định ở mỗi lần xử lý, điều mà cấu trúc quản lý đoạn không đáp ứng được, Có thể thực hiện việc đổi chỗ các phần tử ở vị trí

chẵn sang vị trí lẻ và ngược lại, Thực hiện các phép xử lý nêu trên một cách hiệu quả đối với xâu (*cấu trúc* ***Rope***).

1.6-Bài tập áp dụng

Bài 1: Hai bản đồ:

Vệt đứt gãy lớn trên vỏ trái đất *Banda Detachment* nằm ở phía tây Thái bình dương, độ sâu đáy biển ở đó đạt tới 7 km. Nhiều bức ảnh địa hình đã được chụp. Để khảo sát vệt đứt gãy lớn này người ta dùng một máy thăm dò tự động. Một trong số các khe nứt tạo thành một đường thẳng và là vùng thuộc kế hoạch thăm dò. Theo dự kiến máy thăm dò sẽ tiến hành đo đạc khảo cứu đoạn có độ dài ***s***. Bộ nhớ của thiết bị thăm dò chỉ có thể chứa 2 bản đồ cho tổng độ dài là ***s***. Nếu *một phần nào đó có cả ở 2 bản đồ thì chỉ tính là một*. Bộ phận chuẩn bị có 2 thao tác cơ bản:

Thao tác ***A*** có dạng ***1 l r*** – xin cung cấp bản đồ của khe nứt đoạn từ điểm l đến điểm ***r***, ***l*** < ***r***,

Thao tác ***B*** có dạng ***2 k*** – trả về bản đồ đã xin ở thao tác thứ k. Đảm bảo là thao tác thứ k thuộc loại A và không có việc một bản đồ bị trả về 2 lần.

Các thao tác được đánh số từ 1, có thể tồn tại nhiều bản đồ cùng chụp một đoạn (có ***l*** và ***r*** giống nhau). Có ***n*** thao tác được thực hiện. Sau mỗi thao tác hãy xác định số cách khác nhau chọn 2 bản đồ khác nhau cho tổng độ dài đúng bằng ***s***. Hai cách chọn gọi là khác nhau nếu một bản đồ (xác định theo trình tự xin cung cấp) có ở cách chọn thứ nhất và không có ở cách chọn thứ 2. Ban đầu bộ phận chuẩn bị chưa có bản đồ nào trong tay.

Dữ liệu: Vào từ file văn bản TWOMAPS.INP: Dòng đầu tiên chứa 2 số nguyên ***s*** và ***n*** ($1 \le s \le 10^9$, $1 \le n \le 10^5$), Mỗi dòng trong ***n*** dòng tiếp theo chứa 2 hoặc 3 số nguyên xác định thao tác dạng ***A*** hoặc ***B***, trong đó ***l, r*** có giá trị tuyệt đối không vượt quá 5×10^8.

Kết quả: Đưa ra file văn bản TWOMAPS.OUT số cách chọn tính được sau mỗi thao tác, các kết quả đưa ra dưới dạng số nguyên, mỗi số trên một dòng.

Ví dụ:

TWOMAPS.INP
10 6 1 0 8 1 7 10 1 5 15 2 2 1 12 14 2 5

TWOMAPS.OUT
0 1 2 0 2 0

Giải thuật:

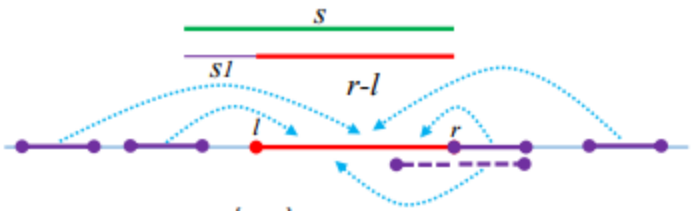
Phương án A: Không sử dụng cấu trúc dữ liệu TREAP

Xét việc bổ sung bản đồ [**l**, **r**], Đặt **s1**=**s-(r-l)**, có 3 trường hợp xảy ra:

s1 < 0 : không có thêm lựa chọn mới,

s1 = 0 : Số lựa chọn bổ sung bằng số đoạn nằm gọn trong [**l**, **r**],

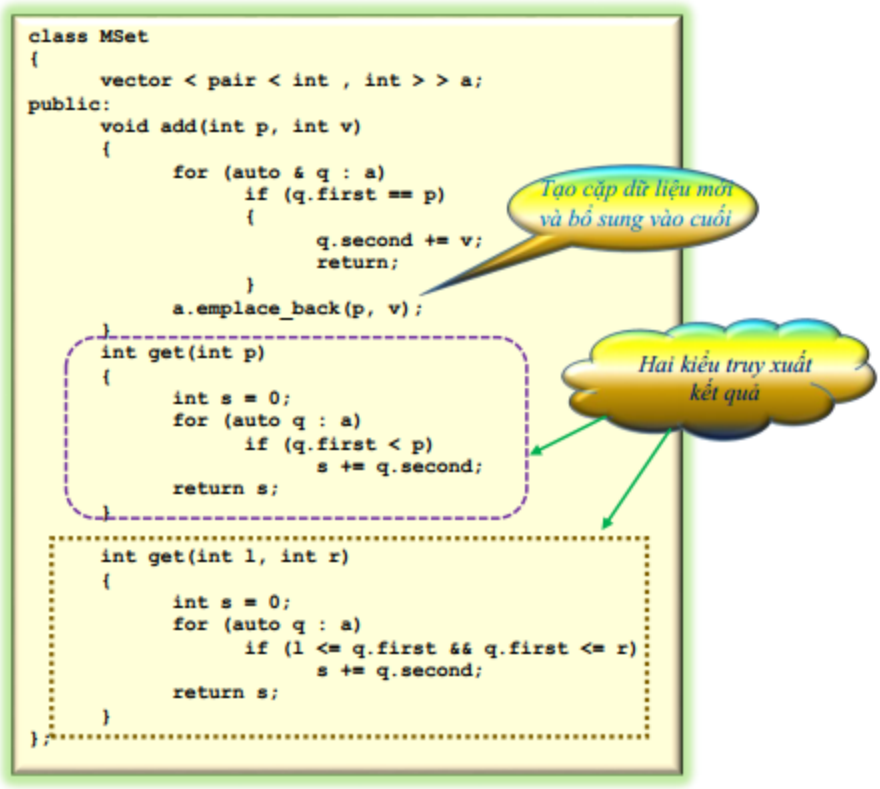
s1 > 0 : Số lựa chọn bổ sung bằng tổng số đoạn có độ dài **s1** nằm ngoài đoạn [**l**, **r**], cộng với số lượng đoạn có giao khác rỗng với [**l**, **r**] và tổng độ dài ngoài phần chung bằng **s1**.



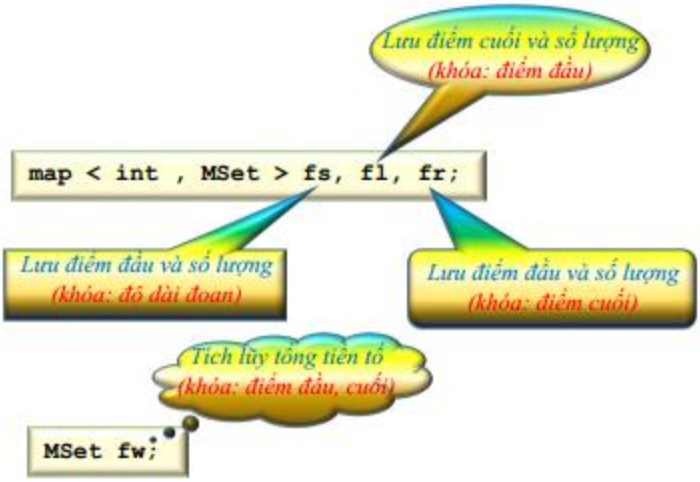
Để quản lý các đoạn phục vị tìm kiếm cần lưu trữ: 4 Điểm đầu của đoạn và số lượng điểm đầu trùng với nó *theo khóa điểm cuối*, 4 Điểm cuối của đoạn và số lượng điểm cuối trùng với nó *theo khóa điểm đầu*, 4 Điểm đầu của đoạn và số lượng điểm đầu trùng với nó *theo khóa độ dài đoạn*.

Khi **r-l** bằng **s** các thông tin lưu trữ nêu trên mới cho phép thống kê các đoạn cần tìm có điểm đầu hoặc điểm cuối trùng với đoạn đang xét. Cần lưu trữ *tổng tiền tố* số lượng điểm đầu của các đoạn có độ dài không vượt quá **s-2**, tức là các đoạn có thể nằm gọn trong đoạn độ dài **s** và không có chung điểm đầu và điểm cuối với đoạn độ dài **s**.

Tổ chức dữ liệu: Để thuận tiện lập trình và dễ hiểu hơn khi chuyển sang phương án dùng TREAP ta khai báo một *cấu trúc dữ liệu hướng đối tượng* (*OOP*), gắn các phép xử lý với dữ liệu. Điều này cho phép dù dữ liệu được lồng vào cấu trúc lưu trữ chuẩn nào, dạng lời gọi tới phép xử lý vẫn giữ nguyên.



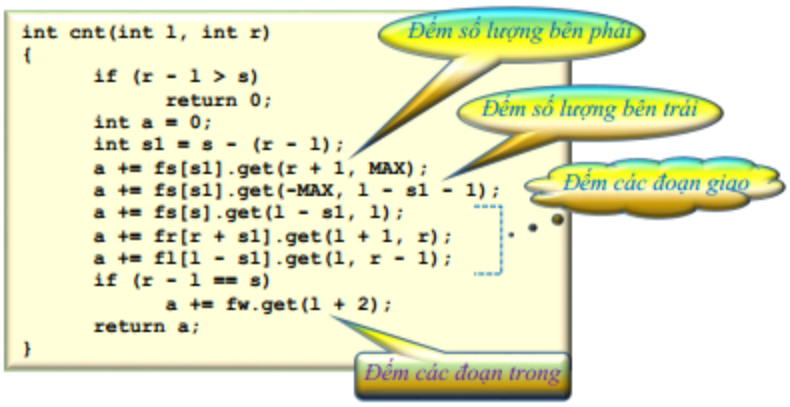
Khai báo dữ liệu



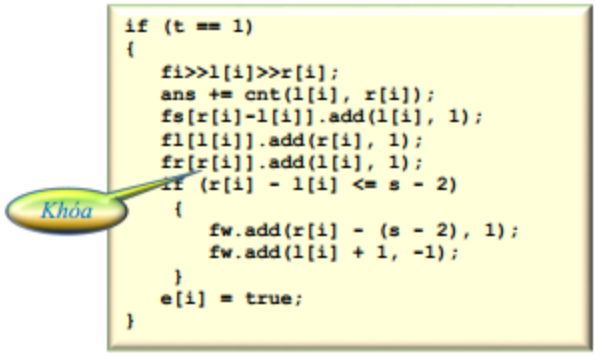
Xử lý:

Thêm bản đồ mới **[l$_i$,r$_i$]**:

Cập nhật kết quả: **ans+=cnt(l[i],r[i]);**



Cập nhật các mảng ghi nhận dữ liệu:



Việc loại bỏ bản đồ: được thực hiện tương tự, nhưng cập nhật thông tin trước, sau đó tính lại kết quả.

Độ phức tạp của giải thuật: $O(n_2)$.

Phương án B: Sử dụng cấu trúc dữ liệu TREAP.

Nhận xét:

Độ phức tạp của phương án A cao vì không thể chèn dữ liệu mới vào chỗ cần thiết trong các vectors để đảm bảo các dữ liệu luôn luôn được sắp xếp, từ đó có thể áp dụng các phương pháp tìm kiếm nhanh,

Giải pháp:

Tổ chức TREAP với trường **sum** phục vụ tính tổng tiền tố.

Gắn các phép xử lý TREAP với cấu trúc.

Cấu trúc TREAP và các phép xử lý: theo đúng sơ đồ lý thuyết đã nêu. Lớp dữ liệu Mset được xác định như sau:

```
class MSet
{
	Treap * t;
public:
	MSet() : t(Treap::null) {}
	void add(int p, int v)
	{
		if (!tadd(t, p, v))
		{
			Treap * t1, * t2;
			split(t, p, t1, t2);
			t = merge(merge(t1, new Treap(p, v)), t2);
		}
	}
	int get(int p)
	{
		Treap * t1, * t2;
		split(t, p, t1, t2);
		int w = t1->sum;
		t = merge(t1, t2);
		return w;
	}
	int get(int l, int r)
	{
		Treap * t1, * t2, * t3;
		split(t, l, t1, t2);
		split(t2, r + 1, t2, t3);
		int w = t2->sum;
		t = merge(t1, merge(t2, t3));
		return w;
	}
};
```

Khai báo dữ liệu trong chương trình và sơ đồ xử lý: giữ nguyên như ở phương án A. *Độ phức tạp của giải thuật:* O(*n*lnn).

Các bài tập áp dụng cùng kỹ thuật.

Bài 2: C11 SEQ – Nguồn *spoj.com*

Cho N (N<=10^5) số nguyên a1,a2,....,an và 2 số L,R (L<=R) Đếm xem có bao nhiêu cặp số i,j thỏa mãn:

- i <= j
- L<= A[i] + A[i+1] +.....+ A[j] <=R

Input: Gồm 2 dòng

Dòng 1: 3 số N,L,R

Dòng 2: N số nguyên

Tất cả các số trong inp đều có giá trị tuyệt đối dưới 10^9

Output

Ghi 1 số là số cặp i,j thỏa mãn.

Ví dụ:

C11 SEQ .INP
4 2 4 1 2 3 4

C11 SEQ .OUT
4

Lời giải:

Bài này có cách làm thông thường là sửa dụng cây Fenwick:

Duyệt i từ 1 đến N, ở mỗi bước ta cần đếm số lượng vị trí $j<ij<i$ có $L \le S[i]-S[j] \le RL \le S[i]-S[j] \le R \Leftrightarrow S[i]-R \le S[j] \le S[i]-L \Leftrightarrow S[i]-R \le S[j] \le S[i]-L$

Cách làm đơn giản (không AC) sẽ như sau:

Duyệt ii từ 1 đến N:

Tính $S[i]=S[i-1]+A[i]S[i]=S[i-1]+A[i]$

Truy vấn trên cây BIT để tính tổng trong đoạn $[S[i]-R,S[i]-L][S[i]-R,S[i]-L]$

Truy vấn trên cây BIT, tăng phần tử ở vị trí $S[i]S[i]$

Dễ thấy cách làm này không thể AC vì giá trị của $S[i]S[i]$ có thể lên đến 10141014, không thể lưu nổi.

Giải quyết vấn đề này bằng cách rời rạc hóa:

Sắp xếp lại SS, loại bỏ các phần tử bị trùng. Gọi mảng mới sau khi sắp xếp và loại phần tử trùng là XX.

Với truy vấn tăng phần tử ở vị trí $S[i]S[i]$, ta tìm kiếm nhị phân để xác định vị trí của $S[i]S[i]$ trên mảng XX. Rồi dùng giá trị này để thay thế $S[i]S[i]$.

Với truy vấn tính tổng trong đoạn $[S[i]-R,S[i]-L][S[i]-R,S[i]-L]$, tương tự như trên, tìm kiếm nhị phân 2 giá trị $S[i]-RS[i]-R$ và $S[i]-LS[i]-L$ trên mảng XX.

Có thể thấy, cách làm trên khá rườm rà. Bài này có thể sử dụng cây nhị phân tìm kiếm. Ở mỗi bước chỉ cần duyệt trên cây để đếm số phần tử nằm trong khoảng $[S[i]-R,S[i]-R][S[i]-R,S[i]-R]$, sau đó cập nhật thêm $S[i]S[i]$ vào cây. Code treap.cppcài đặt phương pháp này sử dụng cấu trúc Treap, là một loại cây nhị phân tìm kiếm có kết hợp với yếu tố ngẫu nhiên để vừa hiệu quả vừa dễ cài đặt.

Bài 3: Arra-and-simple-queries– Nguồn *hackerrank.com*

Cho hai số M và N . Cho biết số lượng phần tử trong mảng và cho biết số lượng truy vấn. Bạn cần thực hiện hai loại truy vấn trên mảng A và MA

Bạn được cung cấp M các truy vấn. Truy vấn có thể có hai loại, loại **1** và loại **2** .

Loại 1 truy vấn được biểu diễn như 1 i j: Sửa đổi các mảng được bằng cách loại bỏ các yếu tố từ i để j và thêm chúng vào phía trước.

Loại 2 truy vấn được biểu diễn như 2 i j: Sửa đổi các mảng được bằng cách loại bỏ các yếu tố từ để và thêm chúng vào phía sau.

Nhiệm vụ của bạn chỉ đơn giản là in mảng kết quả sau khi thực hiện các truy vấn theo sau là mảng kết quả. |A[1]-A|N]|M

Lưu ý Trong khi thêm ở phía sau hoặc phía trước, thứ tự các phần tử được giữ nguyên.

Định dạng đầu vào

Dòng đầu tiên bao gồm hai số nguyên được phân tách bằng dấu cách N và M .

Dòng thứ hai chứa N các số nguyên, đại diện cho các phần tử của mảng.

M truy vấn theo sau. Mỗi dòng chứa một truy vấn thuộc *loại 1* hoặc *loại 2* trong biểu mẫu.

Ràng buộc:

$1 \leq N, M \leq 10^5$
$1 \leq A[i] \leq 10^9$
$1 \leq i \leq j \leq N$

Định dạng đầu ra

In giá trị tuyệt đối tức là trong dòng đầu tiên. Các phần tử in của mảng kết quả trong dòng thứ hai. Mỗi yếu tố nên được ngăn cách bởi một không gian duy nhất. $abs(A[1] - A[N])$

INPUT
8 4 1 2 3 4 5 6 7 8 1 2 4 2 3 5

OUTPUT
1 2 3 6 5 7 8 4 1

1 4 7 2 1 4	

Giải thích

Cho mảng là . Sau khi thực hiện truy vấn , mảng trở thành . Sau khi thực hiện truy vấn , mảng trở thành . Sau khi thực hiện truy vấn , mảng trở thành . Sau khi thực hiện truy vấn , mảng trở thành . Bây giờ là tức là và mảng l à

$\{1, 2, 3, 4, 5, 6, 7, 8\}$

$1\ 2\ 4\{2, 3, 4, 1, 5, 6, 7, 8\}$
$2\ 3\ 5\{2, 3, 6, 7, 8, 4, 1, 5\}$
$1\ 4\ 7\{7, 8, 4, 1, 2, 3, 6, 5\}$
$2\ 1\ 4\{2, 3, 6, 5, 7, 8, 4, 1\}$
$|A[1] - A[N]||(2 - 1)|123657841$

Test:https://drive.google.com/file/d/1-_5Q-ovvvOkfFfBKt53qZsfVXb-Cs2eX/view?usp=sharing.

Bài 4: Median Update– Nguồn *hackerrank.com*

Trung vị của các số M được định nghĩa là số giữa sau khi sắp xếp chúng theo thứ tự nếu là số M lẻ. Hoặc nó là trung bình của hai số giữa nếu là số chẵn. Bạn bắt đầu với một danh sách số trống. Sau đó, bạn có thể thêm số vào danh sách hoặc xóa số hiện có khỏi danh sách. Sau mỗi thao tác thêm hoặc xóa, xuất trung vị.

Ví dụ:

Đối với một tập hợp số , trung vị là số thứ ba trong tập hợp được sắp xếp , đó là . Tương tự, đối với một tập hợp số, trung vị là trung bình của phần tử thứ hai và thứ ba trong tập hợp được sắp xếp , đó là . M = 59, 2, 8, 4, 11, 2

Đầu vào:

Dòng đầu tiên là một số nguyên N, cho biết số lượng hoạt động. Mỗi N dòng tiếp theo là *ax* hoặc *rx* . *ax* chỉ ra rằng được thêm vào tập hợp và *rx* chỉ ra rằng x đã bị xóa khỏi tập hợp.

Đầu ra:

Đối với mỗi thao tác: Nếu thao tác được *thêm* , hãy xuất trung vị sau khi thêm x vào một dòng. Nếu thao tác bị *xóa* và số không có trong danh sách, hãy xuất *sai!* trong một dòng duy

nhất. Nếu thao tác được *loại bỏ* và số x nằm trong danh sách, hãy xuất trung vị sau khi xóa trong một dòng. (Nếu kết quả là số nguyên KHÔNG đầu ra dấu thập phân. Và nếu kết quả là số thực, KHÔNG xuất ra dấu 0.)

Lưu ý

Nếu trung vị của bạn là 3.0, chỉ in 3. Và nếu trung vị của bạn là 3.50, chỉ in 3.5. Bất cứ khi nào bạn cần in trung bình và danh sách trống, hãy in *sai!*

Các ràng buộc: Đối với mỗi *ax* hoặc *rx* , sẽ luôn là một số nguyên được ký (sẽ phù hợp với 32 bit).

$0 < N \leq 10^5$

x

INPUT	OUTPUT
7 r 1 a 1 a 2 a 1 r 1 r 2 r 1	Wrong! 1 1.5 1 1.5 1 Wrong!

Lưu ý: Rõ ràng từ dòng cuối cùng của đầu vào, nếu sau khi xóa thao tác, danh sách trở nên trống rỗng, bạn phải in *sai!* .

Test:

https://drive.google.com/file/d/1YpXgcZUb5fYEKrct7krlS6x7INtrTRPp/view?usp=sharing

TÀI LIỆU THAM KHẢO:

1. https://vi.wikipedia.org/wiki/TREAP
2. https://vietcodes.github.io/algo/TREAP
3. https://cp-algorithms.com/data_structures/TREAP.html
4. https://www.geeksforgeeks.org/TREAP-a-randomized-binary-search-tree/
5. https://sites.google.com/site/kc97ble/container/TREAP-cpp
6. https://www.hackerrank.com/TREAPs/
7. https://www.hackerrank.com/challenges/array-and-simple-queries/forum

Ý KIẾN CỦA TÔT NHÓM CHUYÊN MÔN