Treap

2020-04-19

#DATA STRUCTURE (/TAGS/DATA-STRUCTURE/)

筆記 (/CATEGORIES/筆記/)

treap

!這裡只討論split-merge treap

屬於二元平衡樹的一種，因為 **易編寫 速度快 靈活性高** 的特性而在競程占有一席之地。
他可以解決 segment tree 的問題，也可以解決splay tree的問題，同樣也可以解決大部分二元平衡樹的問題，學一個treap抵過學一堆樹。

- **基本原理**
  Treap = tree + heap。
  亦即同時擁有BST與heap性質的資料結構。

  > *heap: 父節點的`pri` 值大於子結點*
  > *BST: 左子樹`key` 值均小於等於父節點，右子樹則大於父節點。*

  一般二元平衡樹為了避免退化，會利用 **旋轉** 操作去維持深度。

而treap因為擁有BST與heap的性質，所以既能擁有BST的查找功能，又能像heap一樣維持深度，

- **treap最重要的兩個操作:**
  - merge(a, b):
    合併兩顆treap，注意此函式必須滿足 **a的所有key值小於b的所有key**
  - split(t, k):
    將treap t分成兩顆treap， **一顆裡的key均小於等於k，另一顆的均大於**。

    兩種操作都因為BST的特性所以實作難度降低許多。
- **基本架構**
  - **node:**

```
struct Treap
{
    Treap *l, *r;
    int_t key, pri;
    Treap(int_t _key)
    {
        l = r = nullptr;
        key = _key;
        pri = rand();
    }
}
```

  - **merge:**

```
Treap* merge(Treap* a, Treap* b)
{
    if (a == nullptr || b == nullptr) return a ? a : b;
    if (a->pri > b->pri)
    {
        a = merge(a->r, b);
        return a;
    }
    else
    {
        b = merge(a, b->l);
        return b;
    }
}
```

- **split:**

```
void split(Treap* t, int_t k, Treap* &a, Treap* &b)
{
    if (t == nullptr)
    {
        a = b = nullptr;
        return;
    }

    if (t->key <= k)
    {
        a = t;
        split(t->r, k, a->r, b);
    }
    else
    {
        b = t;
        split(t->l, k, a, b->l);
    }

}
```

只要有上面兩種操作，基本就寫完了。

- **insert:**

```
void insert(Treap *t, int_t k)
{
    Treap *lt, *rt;
    split(t, k, lt, rt);
    merge(merge(lt, new Treap(k)), tr);
}
```

- **remove:**

```
void remove(Treap *t, int_t k)
{
    Treap *lt, *rt;
    split(t, k - 1, lt, t);
    split(t, k, t, rt);
    t = merge(lt, rt);
}
```

treap，就是這麼簡單。

在平衡樹的問題中，常常遇見尋找第k小元素的要求，就跟BST一樣treap一樣是用節點size去判斷，所以我們現在需要維護treap上節點的size，這樣可以跟BST一樣找kth了。

```
int_t Size(Treap *t)
{
    return t == nullptr ? 0 : t->sz;
}
void pull(Treap *t)
{
    t->sz = 1 + Size(t->l) + Size(t->r);
}
```

- **真正實作**
  node:

```
struct Treap
{
    static Treap mem[MAXN], *ptr;
    Treap *l, *r;
    int_t pri, key, siz;

    Treap() = default;
    Treap(int_t _key)
    {
        l = r = nullptr;
        pri = rand();    //可以用其他隨機方法，保證更好的隨機性
        key = _key;
        siz = 1;
    }
}Treap::mem[MAXN], *Treap::ptr = Treap::mem;
```

split:

```
//與上面唯一有差別的只有當某顆treap的結構改變時，需要呼叫pull()去更新資訊
//例如 呼叫完split()後
```

merge:

```
//與上面唯一有差別的只有當某顆treap的結構改變時，需要呼叫pull()去更新資訊
//例如 呼叫完merge()後
```

kth: (第k小)

```
int_t kth(Treap *t, int_t k)
{
    int_t lsz = sz(t->l) + 1;
    if (lsz < k) return kth(t->r, k - lsz);
    else if(lsz == k) return t->key;
    else return kth(t->l, k);
}
```

以上是treap當平衡樹的版本

- **treap區間維護**

上面提過，treap不只可以解決平衡樹問題，也可以解決序列操作問題。

我們只需要在node裡多加一個 `val` 當作序列上的值、 `key` 當作序列上的索引值、 `mx/mn/sum` 當作要維護的值即可。

treap，就是這麼簡單。

但是當我們遇到區間加值怎麼辦?
線段樹巧妙的用 **lazy tag** 解決了，同樣的treap也可以!

只要用好好維護我們的tag即可。

```
void push(Treap *t)
{
    if (t == nullptr) return;
    t->val += t->lazy;
    t->mx += t->lazy;
    if (t->l != nullptr)
        t->l->lazy += t->lazy;
    if (t->r != nullptr)
        t->r->lazy += t->lazy;
    t->lazy = 0;
}

void pull(Treap *t)
{
    t->sz = 1 + Size(t->l) + Size(t->r);
}
```

node: (改)

```
struct Treap
{
    static Treap mem[MAXN], *ptr;
    Treap *l, *r;
    int_t pri, key, siz, lazy;

    int_t val;    //新增這兩個
    int_t mx;

    Treap() = default;
    Treap(int_t _key, int_t _val)
    {
        l = r = nullptr;
        pri = rand();
        key = _key;
        siz = 1;

        val = mx = _val;
    }
}Treap::mem[MAXN], *Treap::ptr = Treap::mem;
```

build:

```
Treap *t = nullptr;
for (int_t i = 1; i <= n; ++i)
    t = merge(t, new (Treap::ptr++) node(i, a[i]));
```

上面用到了placement new的技巧，通過先開記憶池再去new一個node，可以降低系統分配記憶體的開銷

__區間加值:__

```cpp=
void add_range(Treap *t, int_t l, int_t r, int_t val)
{
    Treap *lt, *rt;
    split(t, l - 1, lt, t);
    split(t, r, t, rt);
    t->lazy += val;
    merge(merge(lt, t), rt);
}
```

- **翻轉吧! treap**

  有沒有觀察到我們剛剛 **build** 時， key 直接放1, 2, 3...，仔細想想這樣 key 的意義不就是 **在treap中有幾個比他小**。
  那只要維護好 `size` 就可以不用管 key 了。
  所以 `split(t, k)` 的意義也就變成了:**在t中切開前k個節點與後$n-k$個節點**。

  - 只用 `size` 的好處
    不必再被 key 綁手綁腳的，當我們遇到什麼區間翻轉、區間剪下貼上，就真的直接剪下去、或轉下去(正常還是會打標)。

    treap，就是這麼簡單

    node: (改)

```
struct Treap
{
    static Treap mem[MAXN], *ptr;
    Treap *l, *r;
    int_t pri, siz, lazy;

    int_t val;
    int_t mx;

    bool rev;    //翻轉標記

    Treap() = default;
    Treap(int_t _key, int_t _val)
    {
        l = r = nullptr;
        pri = rand();
        siz = 1;

        val = mx = _val;

        rev = false;
    }
}Treap::mem[MAXN], *Treap::ptr = Treap::mem;
```

split: (改)

```
void split(node *t, node *&a, node *&b, int_t k)
{
    if (!t) { a = b = nullptr; return; }

    push(t);
    //此節點的左子樹數量大於等於k
    if (size(t->l) >= k)
    {
        b = t;
        push(b);
        //將b指向整個樹，向左子樹處理。
        split(t->l, k, a, b->l);
        pull(b);
    }
    else
    {
        a = t;
        push(a);
        split(t->r, k - size(t->l) - 1, a->r, b);
        pull(a);
    }
}
```

這部分的 `split()` 比較難理解，可以畫圖看看或直接貼程式，輸出中間過程。

!翻轉其實就是左右子樹交換

練習題: luogu P3391 (https://www.luogu.com.cn/problem/P3391)(模板), cf 702F (http://codeforces.com/problemset/problem/702/F)

 (https://github.com/klugjo/hexo-theme-clean-blog)