@nebocco が2020年10月06日に更新

【Python】平衡二分木が必要な時に代わりに何とかするテク【競プロ】

Python, 競技プログラミング

※この記事は説明の都合上AtCoderで出題された問題に関するネタバレが含まれています。ご了承ください。

はじめに

平衡二分（探索）木というデータ構造があります。(Wikipedia)

名前	操作	計算量
`insert(x)`	要素 x の挿入	$O(\log N)$

名前	操作	計算量
`erase(x)`	要素 x の削除	$O(\log N)$
`find(x)`	要素 x の検索	$O(\log N)$

などの操作が可能で、C++では `std::set` や `std::map` 等の連想コンテナの実装に用いられています。

　この `std::set` , `std::map` というのは優れもので、標準ライブラリ中のデータ構造でありながら

- `lower_bound(x)` : x 以上の最小の要素
- `upper_bound(x)` : x 以下の最大の要素

を$O(\log N)$で計算することができます。集合が内部的に常にソートされているので、二分探索ができる、と考えると分かりやすいかと思います。この手軽さゆえに、競技プログラミングの問題の解説に

> 「……よって、平衡二分木（C++ではsetなど）を使うことで解くことができます。」

と書かれていることがしばしばあります。

　残念ながら、Pythonにはそのような組み込みのデータ構造が存在しないため、解説の通りにコードを組むことができません。そこで本記事では、平衡二分木が欲しい場面で代用できる（ある程度汎用的な）手法をいくつか紹介したいと思います。

扱う問題

ABC128-E Roadwork を題材として使用します。（問題リンク）
解説にも書かれているように、この問題は

クエリ	操作
`insert(x)`	集合に要素 x を追加
`erase(x)`	集合から要素 x を削除
`minimum()`	集合内の最小の要素を検索

というクエリをこなす問題です。最小の要素は、充分小さな数を `-INF` として `lower_bound(-INF)` を計算すればよいので、まさに平衡二分木の出番、という問題です。

解決法1: 平衡二分木を作る

身も蓋もないですが、平衡二分木がないなら自分で作ればよいのです。
幸いにも自作の平衡二分木ライブラリを公開してくれている方が何人もいらっしゃるので、自力で実装するのは難しい……という方でも簡単に使うことができます。大感謝ですね。

じゅっぴーさんのブログ
（2020/10/06 追記：URLが切れていたので差し替えました）

　ただし、自作の複雑なデータ構造は往々にして最適化が働きづらく、定数倍はかなり重くなります。そのため、せっかく平衡二分木を用意したのにTLE、となってしまうことが少なくありません。
　平衡二分木を使わずに、平衡二分木と同じ操作はできないでしょうか？

補足：

> AVL木、実はedamatさん(@edamat1)がPypy用に書き直してくれた(再帰をなくしたりしてくれた)のがあり、そちらだと割と通るそう
> — じゅっぴー (@juppyjappy) May 26, 2020

解決法2: BIT（またはセグメント木）

　おそらくこれが一番汎用的な手法かと思います。それぞれのデータ構造を知らない人は調べてください。実装は大差がないため、ここではBITを用いることとします。BITの配列を $B[0], B[1], \ldots, B[i], \ldots,$ という形で表します。
　先述した平衡二分木に対する操作を、以下のように対応付けて考えます。

クエリ	操作

クエリ	操作
`insert(x)`	$B[x]$ に 1 を加算
`erase(x)`	$B[x]$ に -1 を加算
`minimum()`	$\sum_{i=0}^{x} B[x] - \sum_{i=0}^{x-1} B[x]$
`lower_bound(x)`	$s = \sum_{i=0}^{x-1} B[x]$ とし、累積和が初めて s を超える位置を二分探索

こうすると、平衡二分木と同様にこれらの操作をすべて$O(\log N)$で行うことができます。二分探索は単純に実装すると$O(\log^2 N)$ となってしまいますが、セグ木上の二分探索と同じ手法により$O(\log N)$に落とすことができます。（参考リンク）

挿入する要素 x の範囲が $0 \leqq x \leqq 10^6$ 程度であれば、このサイズのBITを構築すればそれでOKですが、今回は挿入する最大の要素が 10^9 程度であるので、圧倒的にMLEします。

解決策2.5: 座標圧縮 + BIT（またはセグメント木）

今回の問題はオフラインクエリであり、最初の段階で将来挿入する値がすべてわかります。そこで、これらの値を全てまとめて昇順にソートし、新たに $0, 1, 2, \ldots$ と番号を振りなおします。（詳しくは「座標圧縮」と検索してください　）要素の個数は最大で 2×10^5 個であるので、新たに割り振った番号の最大値も 2×10^5 です。よってこのサイズのBITを用意すれば、MLEすることなく解くことができます。

提出コード（手持ちのライブラリでは `lower_bound(x)` の定義が「累積和が x 以上になる最小のindex」だったのでそれに合わせて実装しています。）

TLEしてしまいました……

解決法3: 優先度付きキュー

ところで、 `erase` クエリがない場合を考えてみましょう。

クエリ	操作
`insert(x)`	集合に要素 x を追加
`minimum()`	集合内の最小の要素を検索

　これならば、Pythonにある優先度付きキューのライブラリを用いることで実現できます。
しかし、 `erase` クエリを行うには、先頭から線形探索して削除する必要があり、 $O(N)$ かかってしまいます。

　これを解決するために、優先度付きキューを二本使った以下のようなテクニックが使えます。
まず、p, q という二つの優先度付きキューを用意して、以下のように対応付けます。

```
import heapq
p = list()
q = list()
```

```
def insert(x):
    heapq.heappush(p, x)
    return

def erase(x):
    heapq.heappush(q, x)
    return

def minimum():
    while q and p[0] == q[0]:
        heapq.heappop(p)
        heapq.heappop(q)
    return p[0]
```

（これは疑似コードであり、変数のスコープの関係でこのままでは動作しません。）

このアイデアの核は、「現時点での最小値を削除するのでないかぎり、 `erase` を後回しにしても `minimum` の結果には影響しない」ということです。そこで、 `erase` クエリは後回し用の優先度付きキュー q に突っ込んでおきます。

`minimum` の結果を返す際、現時点での最小値 x が削除する必要のあるものかどうかを確かめます。 x が q の中に存在しているとしたら、先頭にあるはずです。よって、 p と q の先頭の要素が等しい場合は、後回しにしていた `erase` を行う必要があります。こうして更新された最小値もまた削除する必要があるかもしれないので、削除されていない本当の最小値が確定するまでこれを繰り返します。

一見この操作には $O(N \log N)$ かかるようにも思えますが、各要素は最大1回しか削除されないため、償却 $O(\log N)$ で

計算可能です。（償却計算量についてはこちらをご覧ください）

ちなみに q は優先度付きキューではなくsetでも大丈夫です。

また、以上の議論では不当な `erase` （そもそも存在しない要素を削除しようとすること）が存在しないと暗に仮定していましたが、別途「現時点で各要素 x がいくつあるか」を格納した辞書を用意することで、クエリが来た時点でそれが不当かどうか判断することもできます。今回は問題の性質上不当なクエリは存在しません。

提出コード

間に合いました！無事ACです。

解決法4: setでゴリ押し

要素の追加、削除はsetを使えば $O(1)$ で行えます。ただし、 `minimum` には $O(N)$ かかってしまいます。

そこで、各クエリあたりの計算量を抑えるのではなく、クエリ計算する回数自体をを減らすことを考えてみます。発想としては先ほどの「後回しにしてもよい」に近いです。以下の疑似コードをご覧ください。

```
S = set()
curmin = 10**18 # 現時点での最小値
flag = False # 更新が必要か

def insert(x):
    S.add(x)
    if x <= curmin:
        curmin = x
```

```
        flag = False
    return

def erase(x):
    S.remove(x)
    if x == curmin:
        flag = True
    return

def minimum():
    if flag:
        curmin = min(S)
        flag = False
    return curmin
```

コードを見てもらえばなんとなく分かるとは思いますが、なるべく結果を保存しようとしています。このように無駄な `minimum` の計算を最大限抑えることで、クエリ一回当たりの平均計算量をなるべく抑えることができます（計算回数はテストケースに依存します）。意地悪なケースではほとんど改善されませんが、完全にランダムなケースでは実行時間が大幅に短くなることもあります。

提出コード

優先度付きキューの解法より速くなりました！驚き

番外編: 別の解法を考える

視点を変えることで別の解法が適用できる場合があります。この問題では二分探索を上手に使う解法があったりします。

おわりに

　紹介した各手法は、平衡二分木の代用としてだけではなく、他の問題を解くうえで計算量を削減するヒントにもなるテクニックです。なんとなく覚えておくと役に立つときが来るかもしれません。

ねぼこ

@nebocco

フォロー

コメント

@azriel1rf

2020-10-06 06:15

じゅっぴーさんのAVL木のリンクが切れてたので新しい方を置いておきます

https://juppy.hatenablog.com/entry/2019/02/26/python_AVL%E6%9C%A8_%E9%85%8D%E5%88%97ver_%E7%AB%B6%E6%8A%80%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9F%E3%83%B3%E3%82%B0_Atcoder_

0

@nebocco 2020-10-06 15:30

@azriel1rf ありがとうございます！本文の方も差し替えておきます

LGTM 0

投稿する

編集 プレビュー

テキストを入力

画像を選択 0B / 100MB 投稿

How developers code is here.

Qiita

About　利用規約　プライバシー　ガイドライン　デザインガイドライン　リリース　API　ご意見　ヘルプ　広告掲載

Increments

About　採用情報　ブログ　Qiita Team　Qiita Jobs　Qiita Zine