Тураев Тимур, 504 (SE)
25.02.2014

2.1. Задача про Гарри Поттера (acm.timus).

Я не знаю, нужно ли к ней писать теоретическое решение, но, вкратце оно следующее.

Давайте заведем AVL-дерево на удаленных комнатах, причем номера этих комнат будут «реальными». То есть при поступлении запроса нужно вначале найти, а какой же был у текущей запрашиваемой комнаты реальный номер, затем либо вставить его в дерево (если запрос типа D), либо вывести его на печать (если запрос типа L).

Остался один вопрос, как же найти реальный номер комнаты. Реальный номер, к примеру, k – это такой номер, слева от которого ровно k-1 неудаленная комната. Поэтому с помощью дерева нужно найти такой узел, слева от которого ровно k-1 неудаленный номер.

Пусть мы находимся в корне дерева. Его значение, пусть A. Мы знаем размер левого поддерева: size(A->left) = A_left. Это значит, что слева от A находится ровно A – A_left неудаленных комнат. (или, на отрезке [1, A] ровно A – A_left – 1 комнат).

Если искомое число k не больше этого числа комнат, то нужный нам номер в левом поддереве, иначе – в правом.

Один тонкий момент связан с тем, что только для корня верно замечание «на отрезке [1, A] ровно A – A_left – 1 комнат», в общем случае нам нужно считать их число на отрезке [left, A], где left – левая граница отрезка, она изменяется только при переходе в правое поддерево и равна A.

Те же самые слова в коде:

```
int realNumber(node *p, int left, int n):
 if (!p)
      return n + left;

  int free = p->key - size(p->left) - 1 - left;

  if (free >= n)
      return realNumber(p->left, left, n);
  else
      return realNumber(p->right, p->key, n - free);
```

Код в приложении к письму (вместе с этим файлом).

2.2. Своппер.

Эту задачу я в систему пока не сдал (по состоянию на 25.02.2014), но успею или нет, я ее когда-нибудь точно сдам.

Я не знаю, может решение и неоптимальное, но кажется логичным.

Смысл в чем – постоянно менять четные и нечетные элементы местами и считать какую-то статистику. Можно завести по два BST: одно будет хранить элементы, стоящие на четных местах, а другое на нечетных.

Дальше, хочется быстро находить те поддеревья, отвечающее нужным элементах из запроса, и просто менять их местами.

Это очень похоже на то, что мы решали на практике: дан массив, нужно взять из него часть и кинуть в начало. Можно сделать тут тоже самое: давайте специализируем BST: возьмем 2 splay-дерева.

Тогда ответ на первый запрос (поменять местами элементы из отрезка [l, r]) будет выглядеть так: двумя сплитами «вырезаем» из первого дерева нужные элементы, аналогично двумя сплитами «вырезаем» дерево из второго элемента и меняем их местами: примёрживаем первое поддерево ко второму (это четные элементы встали на нечетные места), а второе поддерево – к первому.

Ответ на второй вопрос тоже просто старится: давайте поддерживать в каждом узле дополнительно статистику «сумма элементов в его поддереве» и на запрос «найдите сумму на отрезке [a, b]» просто вырежем (также двумя сплитами) из дерева нужное нам поддерево и выведем значение статистики в корне.

Как вырезать сплитами нужный нам участок, мы обсуждали на практике: сначала находим узел со значением «a», делаем сплит по нему, затем в правом дереве находим узел «b» и делаем сплит по нему. Дерево в «центре» – искомое.

Один тонкий момент: как искать элементы, по какому ключу? Судя по всему, строить дерево придется по неявному ключу, потому что нам нельзя нарушать порядок элементов. Неявный ключ тут будет «индекс» элемента в массиве и определяться он будет как «размер левого поддерева» + 1. Поэтому, поиск элемента «a» есть не что иное, как поиск a-го элемента по возрастанию.

2.3. Pairing heap.

2.4. Анализ вставок в Splay-дерево.

Дано:
m – общее число запросов к дереву.
p_i * m – число запросов к элементу x_i
p_i – частота запросов к i-му элементу. (их сумма = 1)

Определим для вершины i вес: он равен p_i

Ранг: rank(i) = $\log size(i)$= $\log sum(p_i)$, иными словами ранг вершины есть логарифм «размера» вершины, который равен сумме весов всех вершин в поддереве этой вершины, где вес вершины есть частота запроса к ней.

Далее применяем анализ сплей дерева. Из него мы можем получить важное следствие, о том, что амортизированное время splay для вершины x в дереве с корнем t есть не больше: чем 3(rank(t)-rank(x)) + 1 = $O(\log\left(\frac{size(t)}{size(x)}\right) + 1)$. Отсюда, изменение потенциала за всю серию запросов будет не больше, чем $\sum_{\boldsymbol{i}} \boldsymbol{\log\left(\frac{size(t)}{size(i)}\right) = \sum_i \log\left(\frac{W}{p_i}\right)}$, где W – сумма весов, то есть максимальное значение которое может принять размер вершины, то есть 1.

Мы знаем изменение потенциала за серию, осталось узнать сумму амортизированных времен доступа. Амортизированное время доступа к одной вершине равно $O\left(\log\left(\frac{W}{p_i}\right) + 1\right) = O(\log\frac{1}{p_i} + 1)$, значит сумма времен доступа за m операций есть

$$\boldsymbol{\sum_i [p_i * m * O(\log\frac{1}{p_i}) + p_i * m]}$$

.

Итог: общее время на m операций есть сумма амортизированных времен и изменения потенциала, то есть $O(\sum_i p_i * m * \log\frac{1}{p_i} + \sum_{\boldsymbol{i}} \log\frac{1}{p_i} + m) = O(m * (1 + \sum_i p_i * \log\frac{1}{p_i}))$