Домашняя работа №3

Тетерин Дмитрий, Б05-932

16.04.2020

Задача 1. Предложите метод слияния двух AVL-деревьев за O(log(|T1| + |T2|)) (все ключи левого дерева меньше всех ключей правого).

Решение. Приведём алгоритм и будем на каждом его шаге доказывать, что мы не выходим за предложенную ассимптотику.
1. Определим высоту сливаемых деревьев. В моей реализации AVL-деревьев (см. контест) это $O(1)$.
2. Удалим самый правый элемент α из левого дерева (либо самый левый из правого, если левое дерево оказалось выше), предварительно запомнив его. $O(log|T1|)$
3. В правом (или, соответственно, левом, если левое было выше) дереве будем идти влево (вправо), пока не найдём элемент β, в котором поддерево не более чем на 1 выше левого (правого). $O(log|T2|)$
4. Заменяем $\beta_{new} \to key = \alpha \to key; \beta_{new} \to left = left\_tree; \beta_{new} \to right = \beta$. $O(1)$
5. Увеличиваем баланс родителя. Сама β уже сбалансирована. $O(1)$
6. Восстанавливаем баланс, как сделали бы после обычной вставки β. $O(log(|T1|+|T2|)$.
Таким образом, мы слили два AVL-дерева с ассимптотикой $O(log(|T1|+|T2|))$ и задача решена. $\square$

Задача 4. Пусть дано декартово дерево. Не ухудшая асимптотику стандартных операций, находясь в вершине, предложите метод поиска следующей в естественном порядке. Более формально, если алгоритм находится в вершине v с ключом x, как попасть в вершину u с минимальным ключом z > x? Асимптотика ответа на запрос:
а) O(log n) в среднем;
б) O(1).

Решение. а)

```
Если у текущей вершины есть правое поддерево, возвращаем самый левый в нём
Иначе идем вверх, пока cur == cur->parent->right и cur->key > key
если в какой-то момент оказались в корне, возвращаем nullptr
возвращаем самый левый элемент правого поддерева текущего
Node *findNext (Node *cur)
{
    if (cur->right) {
        //вернуть самый левый в правом поддереве
        return findLeft(cur->right);
    } else {
        if (cur == root) return nullptr;
        while (cur == cur->parent->right && cur->key > key) {
            cur = cur->parent;
            if (cur == root) return nullptr;
        }
        //вернуть самый маленький, больший key, в правом поддереве родителя
        return findLeftWhileGreater(cur->parent->right);
        //заметим, что эта функция может ветвиться внутри.
    }
}
//выглядит эта функция без всех проверок на nullptr как-то так
Node *findLeftWhileGreater (Node *cur)
{
    while ( cur не лист и не nullptr ) {
        while ( cur не лист && cur->left->key > key) {
            cur = cur->left;
        }
        ans = cur;
        cur = cur->left;
        while ( cur не лист && cur->right->key < key  ) {
            cur = cur->right;
        }
        ans = cur;
        cur = cur->left;
    }
    return ans;
}
```

б)
Давайте поддерживать список вершин, упорядоченный по возрастанию ключей. Понятно, что в такой конфигурации мы выдаём для данной вершины ответ за единицку. Чтобы поддерживать такой список, нам достаточно научиться обновлять его при операциях split и merge:

```
<Treap, Treap> split (Treap t, key_t key)
{
    if (!t) return nullptr;
    if (key > Treap.key) {
        <t1, t2> = split(t.right, key);
        t.right = t1;
        t.next = findNext(t1, key); //занимает log времени
        return <t, t2>;
    } else {
        <t1, t2> = split(t.left, key);
        t.left = t2;
        t.prev = findPrev(t.left, key); //обратная по аналогии с пунктом а
        return <t, t2>;
    }
}


Treap merge (Treap t1, Treap t2)
{
    if (!t2) return t1;
    if (!t1) return t2;
    else if (t1.y > t2.y) {
        t1.right = merge(t1.right, t2);
        t1.next = findNext(t1.right, t1.key);
        return t1;
    }
    else {
        t2.left = merge(t1, t2.left);
        t2.prev = findPrev(t2.left, t2.key);
        return t2;
    }
}
```

□

Задача 5. Дано взвешенное дерево, то есть на каждом ребре написано некоторое целое (возможно, отрицательное) число. Найдите в нём простой путь с наибольшей суммой. Асимптотика: O(n), где n – число вершин в дереве.

Решение. Запустим рекурсивный алгоритм от листьев к корню и будем возвращать максимальный путь в каждой ноде.

```
    struct Node {
    Node *children;
    int weight, childrenQty;
};
```

У листа результат функции будет равен 0. У остальных - $max(weight+$все попарные суммы результ работы функции от детей).

```
int findMaxWay (Node *cur, int val)
{
    if (!cur) return 0;
    int chqt = cur->children_qty;

    int way[chqt];

    for (int i = 0; i < chqt; ++i) {
        way[i] = findMaxWay(cur->children[i], val);
    }

    return max(0, cur->weight,
        cur->weight + way[0],
        cur->weight + way[1],
        ...,
        cur->weight + way[chqt-1],
        cur->weight + way[0] + way[1],
        cur->weight + way[0] + way[2],
        ...,
        cur->weight + way[chqt-2] + way[chqt-1],
        );
}
```

□

Задача 6. Дан набор чисел a1, . . . , an, изначально каждое находится в своём собственном множестве. Поступают два вида запросов в общем количестве q: объединить два множества (множества задаются некоторыми своими элементами); а также по числу x сообщить наименьшее число, большее x, в заданном множестве (вновь множество задаётся некоторым представителем). Обработайте все запросы за
а) (1 балл) O(q log2 n) (можно в среднем);
б) (2 балла) O(q log n) (можно в среднем).

Решение. Приведём сразу решение для пункта б, оно автоматически будет даже лучше, чем для пункта а.
Кажется, это задача про систему непересекающихся множеств.
Будем считать, что изначально у нас n деревьев, состоящих из одного элемента (под элементом подразумевается $struct\ Node$, объяснено ниже).

Когда мы объединяем два множества, будем делать так, чтобы результат оставался двоичным деревом поиска, следующим алгоритмом:
Берём корень α первого дерева, ищем, куда вставить во второе.
Находим какой-нибудь элемент β, правый сын которого больше α, а левый – меньше. Делаем α правым или левым сыном β, чтобы сохранить условие дерева поиска, а соответствующий сын β рекурсивно по такому же принципу вставляем в (теперь уже под-)дерево с корнем α.

Докажем асимптотику этой операции. Когда мы объединяем два дерева, то к дереву с большим рангом (ранг может быть как количеством вершин, так и глубиной дерева) присоединяем дерево с меньшим рангом. Такая стратегия даёт $O(log(n))$ на запрос.
Доказательство для глубины дерева: покажем, что если ранг дерева равен k, то это дерево содержит как минимум 2^k вершин (отсюда будет автоматически следовать, что глубина дерева есть величина $O(log(n))$.
Доказывать будем по индукции: для $k = 0$ это очевидно. Ранг дерева увеличивается с $k - 1$ до k, когда к нему присоединяется дерево ранга $k - 1$, применяя к этим двум деревьям размера $k - 1$ предположение индукции, получаем, что новое дерево ранга k действительно будет иметь как минимум 2^k вершин.

Чтобы поддерживать запрос множества, в котором содержится элемент x, будем хранить $std :: map$, где ключ - элемент, значение - Node*.

```
struct Node {
    Node *какой-то предок;
    ...; //всё, что нужно для остального функционала
    int значение;
};
```

В момент востребования заполняем добавляем в мапу ключ-вершину и ссылку на неё же.
Запрос множества по элементу отрабатывается очевидно: идём в эту мапу, ищем элемент за $O(log(n))$, смотрим на его предка. Дальше смотрим на его предка за единицку. И т.д., пока предок не будет указывать сам на себя. Это и есть искомое дерево. Асимп-

тотика всего этого безобразия $O(log(n))$.
При слиянии деревьев мы у корня первого дерева меняем ссылку на верхний корень.

$\square$