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. □

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;
}
```

Пару слов о том, почему это работает. Во-первых, понятно, что если у вершины есть правое поддерево, то по свойству дерева поиска минимальный элемент, больший её, находится в этом поддереве → первый шаг алгоритма корректен. Если же такого поддерева нет, то искомая вершина находится где-то выше или правее относительно нашего родителя. Если мы являемся левым сыном родителя, то ответом (опять-таки в силу свойства дерева поиска) будет какая-то вершина α его правого поддерева, в противном случае – какая-то вершина выше или правее относительно родителя. Поиск α совершенно понятен: нужно спускаться влево, если текущая больше нашей, и вправо, если наоборот, пока не дойдём до корня. Именно это и делает функция $findLeftWhileGreater$.

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. а) Заведём $std :: map < key_type, std :: set* >$ для того, чтобы поддерживать запрос множества, содержащего элемент. Также создадим массив std::set из n элементов, будем понимать, что изначально $set[n]$ содержит одну вершину. (Не будем этого явно делать, т.к. это требует O(n) времени, учтём при первом запросе на это множество).
Проинициализируем наш map указателями на соответствеющие set при запросе "объединить".
Отрабатывание запроса "слей два множества":
1) в мапе ищем два множества, которые содержат наши элементы;
2) сливаем два множества путём перекидывания элементов из меньшего по количеству элементов в большее;
3) параллельно обновляем map для каждого перекинутого элемента; Понятно, что каждый элемент будет перекидываться не более $log(n)$ раз, т.к. после одного сливания он находится в множестве из $>= 2$ элементов, после второго – в множесте из $>= 4$ элементов и так далее, поэтому на объединение двух множеств мы потратим $log^2(n)$, ведь на одно перекидывание тратится ровно $O(log(n))$. Минимальный элемент, больший заданного, ищется методом $upper_bound$ для сета. Асимптотика: $O(log(n))$. Итоговая асимптотика: $O(q * log^2(n)) + O(q * log(n)) = O(q * log^2(n))$.

□