Домашняя работа №4

Шишацкий Михаил, 932

24.04.2020

Задача 1. Модифицируем дерево отрезков: в каждой вершине также заведём поля $update, set, set\_flag$ (изначально $set\_flag := false$).

Построим ДО на данном массиве классическим способом, в полях key будет содержаться сумма на отрезках нод, в полях $right$ и $left$ - индексы правой (не включительно) и левой границ соответственно.

Запрос *суммы* на отрезке будем выполнять аналогично запросу суммы в классическом ДО, добавляя некоторые действия. Если отрезок текущей ноды покрывается отрезком запроса, мы возвращаем "наверх" значение в зависимости от флага set_flag: если он выставлен, вернём $(set + update) \cdot (node.right - node.left)$, иначе вернём $key + update \cdot (node.right - node.left)$. Дальше рекурсия в этом случае не пойдет. Если же нам нужно спуститься в подотрезки, и в текущей ноде есть не нейтральные значения полей $update, set, set\_flag$, протолкнём эти значения (set перезапишем, $update$ прибавим) в обоих сыновей, в текущей ноде обновим значение key в соответствии с полями и выставим значения полей $update, set, set\_flag$ на нейтральные.

Запрос на *присвоение* величины val на отрезке будем выполнять так же, как запрос суммы на отрезке для классического ДО. Однако, в момент, когда отрезок ноды полностью покрывается отрезком запроса, будем выставлять $set\_flag := true$ и вписывать в поля $update := 0, set := val$ (вместо возврата суммы на подотрезке). Когда нужно будет спуститься в сыновей, протолкнём текущие значения дополнительных полей в них, затем рекурсивно вызовем запрос от сыновей и релаксируем значение в текущей ноде: запишем в key сумму на отрезках её сыновей (в зависимости от set_flag)

Запрос на *увеличение* на отрезке на величину x будем выполнять аналогично *присвоению* на отрезке, только будем выполнять иные действия с полями: $update := update + x$. Случай спуска в обоих сыновей аналогичен такому же в *присвоении*.

Все операции выполняются за $O(log(n))$, т.к. аналогичны запросу суммы в классическом ДО, т.е. на каждом уровне мы посетим не более 4х (в некоторых реализациях не более 2х) нод.

Задача 2. Чтобы находить не только минимум, но и индекс минимума, заведём шаблонную структуру Elem:

```
template <typename T>
struct Elem {
    T val;
```

```
    uint32_t idx;
};
```

И определим для неё оператор $<$:

```
template <typename T>
bool operator<(const Elem<T>& elem_1, const Elem<T>& elem_2) {
    return (elem_1.val == elem_2.val ?
            elem_1.idx < elem_2.idx :
            elem_1.val < elem_2.val);
}
```

Теперь мы можем определить SparseTable на массиве структур Elem (т.е. копий исходного массива элементов, содержащих свои индексы в исходном массиве). При запросе минимума мы будем получать пару - значение - его индекс. Если копий минимального элемента несколько, вернётся копия с наименьшим индексом.

Задача 5. Модифицируем SparseTable:

Будем предполагать, что изначально памяти в таблице выделено под максимальное возможное количество элементов в ней после всех добавлений. В противном случае добавление в конец массива будет происходить за O(1) в среднем.

Запрос минимума на отрезке наша таблица уже умеет выполнять за O(1).

Реализуем добавление элемента в конец: пересчитаем текущий размер таблицы и увеличим высоту, если это необходимо. Положим наше значение в конец массива на нулевом уровне таблицы (того, что совпадает с исходным массивом данных). Заметим, что при добавлении элемента на каждом уровне таблицы необходимо обновить (дописать справа) ровно одну ячейку:

```
template <class Key>
void SparseTable<Key>::append(const Key& value) {
    table[0][this->size++] = value;
    size_t j = 0;
    size_t i = 1;
    for (size_t step = 1; 2 * step <= this->size; step <<= 1, ++i) {
        j = this->size - 2 * step;
        table[i][j] = min(table[i - 1][j], table[i - 1][j + step]);
    }
}
```

Для фиксированного k новое добавленное значение попадает ровно в один полуинтервал вида $[i; i + 2^k)$. Если бы он попадал хотя бы в два отрезка, у этих отрезков совпадали бы суффиксы, а значит, совпадали бы и отрезки, т.к. они имеют равную длину. Причём, левый конец этого полуинтервала лежит правее всех остальных концов. А т.к. до добавления этого элемента такой полуинтервал не имел смысла, т.к. покрывал собой несуществующую область массива, это и есть самая левая незаполненная ячейка массива на каждом уровне.

Приведённый алгоритм также учитывает возможность увеличения высоты таблицы, т.к. высчитывает шаг в зависимости от нового размера.