Конспекты по проге 1 семестр

Содержание

1 Введение

1.1 Алгоритм

Определение Алгоритм — это формально описанная вычислительная процедура, получающая исходные данные (input), называемые также входом алгоритма или его аргументом, и выдающая результат вычисления на выход (output). Алгоритм определяет функцию (отображение):

$$F\colon X \to Y \tag{1}$$

X - входные данные, Y - выходные

Примеры

Вычисление числа Фибоначчи

```
int fib(int n) {
        int x = 1;
        int y = 0;

        for (int i = 0; i < n; i++) {
                x += y;
                y = x - y;
        }
        return y;
}
```

Или же через перемножение матриц за $O(log(n))$

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}$$

Проверка числа на простоту

Перебор до $\sqrt{n}$
Решето Эратосфена

Быстрое возведение в степень

```
int power (int a, int n) {
        if (n == 0) return 1;
        if (n % 2 == 0) {
                int b = power (a, n/2);
                return b*b;
        } else {
                return power (a, n - 1)*a;
        }
}
```

1.2 Асимптотики

f ограничена сверху функцией g асимптотически

$$f(x) \in O(g(x)) \Leftrightarrow \exists(C > 0)(\forall x)\colon \|f(x)\| \leq C\|g(x)\|$$

f ограничена снизу функцией g асимптотически

$$f(x) \in \Omega(g(x)) \Leftrightarrow \exists(C > 0)(\forall x)\colon \|f(x)\| \geq C\|g(x)\|$$

f ограничена сверху и снизу функцией g асимптотически

$$f(x) \in \Theta(g(x)) \Leftrightarrow \exists(C > 0), (C' > 0)(\forall x)\colon C\|g(x)\| \leq \|f(x)\| \leq C'\|g(x)\|$$

g доминирует над f асимптотически

$$f(x) \in o(g(x)) \Leftrightarrow \forall(C > 0)(\forall x)\colon \|f(x)\| < C\|g(x)\|$$

f доминирует над g асимптотически

$$f(x) \in \omega(g(x)) \Leftrightarrow \forall(C > 0)(\forall x)\colon \|f(x)\| > C\|g(x)\|$$

f эквивалентна g асимптотически при $n \to n_0$

$$f(n) \sim g(n) \Leftrightarrow \lim_{n \to n_0} \frac{f(n)}{g(n)} = 1$$

Примеры

$$O(1) = O(2)$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

$$f(n) \subset g(n) \Rightarrow O(f(n) + g(n)) = O(g(n))$$

1.3 Структура данных, абстрактный тип данных (интерфейс)

Определение Структура данных - программная единица, позволяющая хранить и обрабатывать данные. Для взаимодействия предоставляет интерфейс.

Примеры

int a[n];

std::pair<int, int>

Абстрактный тип данных (АТД) - тип данных, который скрывает внутреннюю реализацию, но предоставляет весь интерфейс для работы с данными, а также возможность создавать элементы этого типа. Абстрактный тип данных реализуется с помощью структуры данных.

Пример

Стек, реализованный через массив

Стек - АТД, массив - структуры данных

1.4 Массив

Определение Массив - структура данных, хранящая набор значений (элементов), которые идентифицируются по индексу или набору индексов.
Динамический массив - массив, размер которого может изменяться во время выполнения программы.

Линейный поиск - поиск по массиву за $O(n)$
В отсортированном массиве поиск возможен за $O(log(n))$ с помощью бинарного поиска

```
int binarySearch(const int *a, int n, int item) {
        int leftPtr = -1, rightPtr = n;

         while (rightPtr - 1 > leftPtr) {
                int middlePtr = (leftPtr + rightPtr)/2;
                if (item > a[middlePtr]) {
                        leftPtr = middlePtr;
                } else if (item < a[middlePtr]) {
                        rightPtr = middlePtr;
                } else {
                        return middlePtr;
                }
        }
        return rightPtr;
}
```

2 Базовые структуры данных

2.1 Динамический массив

$+$: random access operator
$-$: нельзя удалить/вставить в середину за $O(1)$

2.2 Односвязный и двусвязный список

$+$: удаление и вставка в любое место
$-$: поиск/последовательный доступ дорог
Односвязный список имеет ссылку только на следующий узел.
Двусвязный список имеет ссылку на следующий и предыдущий узлы.

Операции со списками:

Поиск элемента ($O(n)$)

Вставка - смена указателей ($O(1)$)

Удаление - смена указателей ($O(1)$)

Объединение (при условии, что храним указатель на последний элемент) - смена указателей ($O(1)$)

2.3 Стек

Сертификат - Last In First Out

Операции:

Добавление в конец ($O(1)$)

Удаление с конца ($O(1)$)

Можно реализовать с помощью:

Динамического массива

Списка

Для хранения в массиве можно использовать указатель на последний элемент и сдвигать его в зависимости от операции.
Чтобы поддерживать минимум в стеке, достаточно хранить не элемент, а пару вида: значение элемента, минимальный элемент в стеке на момент вставки нашего элемента.

2.4 Очередь

Сертификат - First In First Out

Операции:

Добавление в конец

Удаление из начала

Можно реализовать с помощью:

Динамического массива

Списка

Двух стеков

Циклическая очередь в массиве

Реализация циклической очереди в массиве основана на хранении двух указателей: на первый элемент в очереди и на последний. Тогда удаление/вставка элементов будет связана со сдвигом указателя на первый/последний элемент по формуле newPtr = (ptr + 1)%size

Хранение очереди в списке

Для реализации на односвязном списке достаточно хранить указатель на первый и последний элемент, удаление/вставка производятся сменой указателей.

Представление очереди в виде двух стеков

Для реализации очереди на двух стеках достаточно вставлять элементы в один стек, а забирать из другого. В случае, если второй стек пуст, перемещать все элементы из первого во второй.

Время извлечения элемента

Для операции push возьмём 3 монеты: для самого push'а, резерв на pop из первого стека и резерв на pop из второго. Тогда учётная стоимость pop'а из второго стека будет равна 0, т.к. можно использовать оставшиеся монеты. Таким образом, каждая операция за $O(1)$

Поддержка минимума в очереди

Для поддержки минимума необходимо поддерживать минимум в каждом из стеков, тогда минимум в очереди - меньший минимум стеков.

2.5 Дек

Список элементов, в котором можно вставлять и удалять элементы с конца и начала.

Операции:

Вставка в конец

Извлечение из конца

Вставка в начало

Извлечение из начала

Можно реализовать с помощью:

Динамический массива

Двусвязного списка

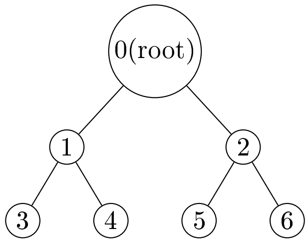
Хранение дека в массиве

Хранение аналогично очереди, только указатели могут сдвигаться как вперёд, так и назад (newPtr = (ptr - 1)%size)

Хранение дека в списке

Храним аналогично очереди, но используем двусвязный список.

2.6 Двоичная куча. АТД Очередь с приоритетом



Определение Двоичное подвешенное дерево, которое удовлетворяет свойствам:

Значение в вершине $\leq$ ($\geq$ для максимума в корне) значению в потомке
На i-ом слое 2^i вершин (кроме, возможно, последнего). Глубина кучи $\log n$
Последний слой заполняется слева направо

Удобно хранить бинарную кучу в массиве:
$a[0]$ - корень
$a[i] \to (a[2i+1], a[2i+2])$ - потомки

Операции Вставка за $O(\log n)$:

Вставляем в свободное место
Рекурсивно поднимаем элемент, если он меньше (больше) родителя (siftUp)

Удаление за $O(\log n)$:

Удаляем минимум
Вставляем последний элемент в корень
Рекурсивно меняем элемент с минимальным (максимальным) потомком (siftDown)

Очередь с приоритетом - абстрактный тип данных, который поддерживает следующие операции:

push - добавить в очередь элемент с определенным приоритетом
pop - удалить из очереди элемент с наивысшим приоритетом
top - посмотреть элемент с наивысшим приоритетом (необязательная операция)

2.7 Амортизационный анализ

Определение - метод подсчёта времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Средняя амортизационная стоимость

$$x = \frac{\sum_{i=0}^{n} t_i}{n}$$

t_i - время выполнения i-ой операции

Амортизированное (учетное) время добавления элемента в динамический массив
Стоимость операции add(item)
Для каждой операции add(item), для которой не требуется расширение массива, будем использовать три монетки: одна для самой вставки, две в резерв. Одну из резерва положим к вставленному элементу с индексом i, а вторую к элементу с индексом $i - \frac{n}{2}$, где n - размер массива
Когда массив заполнится, у каждого элемента будет одна монетка в резерве, которую мы сможем использовать для копирования в новый массив размером $2n$

Метод потенциалов Пусть Φ - потенциал
Φ_0 - изначальное значение
Φ_i - состояние после i-ой операции
Тогда стоимость i-й операции $a_i = t_i + \Phi_i - \Phi_{i-1}$
Пусть n - количество операций, m - размер структуры данных, тогда $a = O(f(n,m))$, если:

$$\forall i\colon a_i = O(f(n,m))$$
$$\forall i\colon \Phi_i = O(n(f(n,m)))$$

Доказательство

$$a = \frac{\sum_{i=1}^{n} t_i}{n} = \frac{\sum_{i=1}^{n} a_i + \sum_{i=0}^{n-1} \Phi_i - \sum_{i=1}^{n} \Phi_i}{n} = \frac{n \cdot O(f(n,m)) + \Phi_0 - \Phi_n}{n} = O(f(n,m))$$

Метод бух. учёта Пусть операция стоит некоторое число монет.
Тогда для каждой операции мы можем взять монет с запасом, чтобы хватило на следующие возможные операции (пример с динамическим массивом)
Если монет хватило на все операции, то наше предположение о стоимости каждой операции (то, что мы взяли с запасом) верно
(Излагаю в неформальном стиле)

2.8 Персистентный стек

Стек, который хранит все свои состояния
Операции:

Вставка: $push(i,x) \to j$, где i - конкретное состояние, x - элемент, который нужно вставить, j - новое состояние после вставки. При вставке создаётся новое состояние стека, где хранится вставленный элемент и ссылка на состяоние, в которое его вставили.

Удаление: $pop(i) \to j$. При удалении i - ого состояния идём с "родителю"i-ого состояния и подцепляем его копию к "деду"i-ого состояния.

В итоге имеем доступ к любой версии стека за $O(1)$ времени и $O(n)$ памяти.
Можно реализовать с помощью:

Массива

Списка

3 Сортировки и порядковые статистики

3.1 Задача сортировки, устойчивость, локальность

Coming soon

3.2 Квадратичные сортировки

Coming soon

3.3 Сортировка слиянием

Coming soon

3.4 Сортировка с помощью кучи

Coming soon

3.5 Слияние отсортированных массивов с помощью кучи

Coming soon

3.6 Нижняя оценка времени работы для сортировок сравнением

Coming soon

3.7 Быстрая сортировка

Принцип Есть массив $a[0...n-1]$ и некоторый подмассив $a[l..r]$

Разделим $a[l...r]$ на две части по опорному элементу q: $a[l...q]$ и $a[q+1...r]$ так что каждый элемент $a[l...q]$ меньше или равен $a[q]$, который не превышает любой элемент подмассива $a[q+1...r]$

Подмассивы $a[l...q]$ и $a[q+q...r]$ сортируются с помощью рекурсивного вызова быстрой сортировки

Схема Ломуто

Выбираем $q = a[r]$

```
q = a[r];
i = l;
for (int j = l; j < r - 1, j++) {
        if (a[j] < q) {
                swap(a[i], a[j]);
                i++;
        }
}
swap(a[i], a[r];
```

Схема Хоара

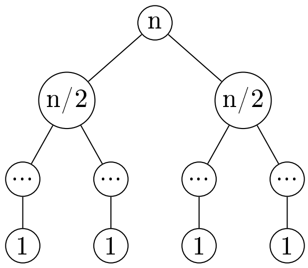
```
q = A[(r + l)/2];
i = l;
j = r;
while (i <= j) {
        while (a[i] < q) {
                i++;
        }

        while (a[j] > q) {
                j--;
        }
        if (i >= j) {
                break;
        }
        swap(a[i], a[j]);
        i++;
        j--;
}
```

Свойства

- Локальная
- Недетерминированная
- Неустойчивая

Асимптотика Дерево рекурсий



$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Худший случай: $T(n) = T(n-1) + \Theta(n)$ (закинули один элемент от partitional)

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$$

Среднее время работы $O(n \log n)$

Доказательство Пусть X - суммарное количество операций сравнения с опорным элементом. Рассмотрим массив $[z_0 ... z_n]$, пусть $z_{ij} = [z_i ... z_j]$
Опорный элемент дальше не участвует в сравнении $\Rightarrow$ сравнение каждой пары ≤ 1 раза

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

где $X_{ij} = 1$, если произошло сравнение z_i и z_j, иначе 0
Применим мат. ожидание к каждой части

$$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr(z_i, z_j)$$

где $Pr(z_i, z_j)$ - вероятность того, что z_i сравнивается с z_j
Пусть все элементы различны
x - опорный $\Rightarrow$ ($\forall z_i, z_j : z_i < x < z_j \Rightarrow z_i$ и z_j не будут сравниваться)
Если z_i - опорный, то он сравнивается с каждым элементом z_{ij} кроме себя
Значит z_i и z_j будут сравниваться, когда первым в z_{ij} будет выбран один из них
$X_{ij} = 1 \Leftrightarrow z_i$ - опорный или z_j - опорный

$$Pr(z_i, z_j) = Pr(z_i) + Pr(z_j) = \frac{1}{j - i + 1} + \frac{1}{j - i + 1} = \frac{2}{j - i + 1}$$

где $Pr(z_i)$ - вероятность того, что первым элементом был z_i, а $Pr(z_j)$ - вероятность того, что первым был z_j, тогда

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

Пусть $k = j - i$, тогда

$$E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

Улучшения быстрой сортировки

- Выбор медианы из первого, среднего и последнего элементов и отсечение рекурсии меньших подмассивов (с помощью сортировок вставками)
- Разделение на три части (применяют, если много одинаковых элементов)

3.8 k порядковая статистика

В чём суть: пусть дан массив A длиной N и пусть дано число K. Задача в том, чтобы найти в этом массиве K-ое по величине число, т.е. K-ую порядковую статистику

Алгоритм Пусть опорный элемент имеет индекс m
Если $k == m \Rightarrow$ успех
Если $k < m \Rightarrow$ ищем k-ую статистику в левой части
Если $k > m \Rightarrow$ ищем (k - m - 1)-ую статистику в правой части

Асимптотика Функция partition при поиске в массиве размера n делает не более $n-1$ сравнений
Проведём оценку сверху, будем считать, что каждый раз выбирается большая половина

$$T(n) \leq \frac{1}{/}n\sum_{k=1}^{n}(T(max(k-1;n-k)+n-1) = n-1+\frac{1}{/}n\sum_{k=1}^{n}(T(max(k-1;n-k)) = n-1+\frac{2}{n}\sum_{k=[n/2]}^{n-1}T(k)$$

Пусть каждый раз массив делится в определенном соотношении, т.е. $\exists c T(k) \leq ck \forall k < n$ Тогда верно:

$$T(n) = n-1+\frac{2}{/}n\sum_{k=[n/2]}^{n-1}ck$$

Здесь короче оценка с neerc, её надо разжевать и вставить сюда

3.9 Сортировка подсчётом

Пусть у нас n целых чисел в диапазоне от 0 до k
Обычно применяют, если n много больше k

Алгоритм Заводим массив $A[k]$
Проходим по нашему изначальному массиву и инкрементируем $A[i]$, если встретили i - ое число

В случае целых чисел просто будем сдвигать число при записи в массив, и сдвигать обратно при доступе к элементу
Также можно сортироввть по ключам сложные объекты
Предварительно подсчитывает количества элементов с различными ключами и разделяем результирующий массив на соответствующие блоки (длины которых как раз и равны значениям вспомогательного массива, который мы получили
Затем при повторном проходе исходного массива каждый элемент копируется в специальнго отведенный его ключу блок

Асимптотика $O(n+k)$

3.10 Блочная сортировка

Суть сортировки Блочная сортирвка применяется при равномерном распределении входных данных
Т.е. мы разбиваем вхдные данные на n блоков. Внутри блока же сортируем удобным образом (либо той же блочной сортировкой). Сортировка работает только в том случае, если разбиение на блоки производится таким образом, чтобы элементы каждого следующего блока были больше предыдущего

Где же применяем? Эту сортировку стоит применять в случае, если данные распределены равномерно, т.е. велика вероятность того, что числа редко повторяются (например, последовательность случайных чисел)

3.11 Поразрядная сортировка

Общий смысл Сначала сравниваем крайний разряд, группируем элементы по значению разряда
Потом сравниваем числа по следующему разряду (сравниваем внутри группы)
Перед сортировкой необходимо привести все данные к единому количеству "разрядов"

MSD Сравниваем, начиная со старшего разряда
Добавляем "пустые"элементы (когда приводим к одинаковому количеству разрядов) в конец
Подходит для строк

LSD Сравниваем, начиная с младшего разряда
Добавляем "пустые"элементы (когда приводим к одинаковому количеству разрядов) в начало
Подходит для чисел
Но необходима устойчивая сортировка каждого разряда

Пример

$$\begin{pmatrix} 32[9] \\ 45[6] \\ 43[7] \end{pmatrix} \to \begin{pmatrix} 32[9] \\ 43[7] \\ 45[6] \end{pmatrix} \to \begin{pmatrix} 32[9] \\ 43[7] \\ 45[6] \end{pmatrix} \to \begin{pmatrix} 3[2]9 \\ 4[3]7 \\ 4[5]6 \end{pmatrix} \to \begin{pmatrix} [3]29 \\ [4]37 \\ [4]56 \end{pmatrix}$$

Асимптотика LSD Пусть m - количество разрядом
n - количество чисел
k - количество значений каждого разряда (10 для чисел)
$T(n)$ - время работы устойчивой сортировки
Тогда сложность $O(m(n+k))$, если устойчивая сортировка имеет сложность $O(n+k)$ (сортировка подсчётом, например)

MSD для строк По первому символу разделяем строки в кучи
Каждую кучу делим аналогично
Когда куча достигла размера 1 - элемент на месте

Асимптотика MSD Пусть значения разрядов меньше b, а количество разрядок - k
Если "делим хорошо"(разбиваем на разные кучки), асимптотика $\Omega(n \log_b n)$
Если "делим плохо"(например, возможна ситуация строк BBBBA и BBBBC), асимптотика $O(nk)$

3.12 Бинарная быстрая сортировка

Аналог MSD сортировки для строк, но алфавит состоит только из 0 и 1

3.13 Мастер-теорема

О чём? Мастер теорема позволяет найти асимптотическое решение рекуррентных соотношений, которые могут возникнуть в анализе асимптотики многих алгоритмов.

Формулировка Пусть $a \geq 1$ и $b > 1$ - константы, $f(n)$ - функция, $T(n)$ определено на множестве неотрицательных целых чисел как:

$$T(n) = a * T(\frac{n}{b}) + f(n)$$

, где n - размер задачи
a - количество подзадач в рекурсии
n/b - размер каждой подзадачи
$f(n)$ - оценка сложности работы вне рекурсии
Пример для merge-sort

$$T(n) = 2T(\frac{n}{2}) + f(n)$$

где $f(n)$ - затраты на слияние (n)

Зачем это нужно? В зависимости от нашей функции $f(n)$:

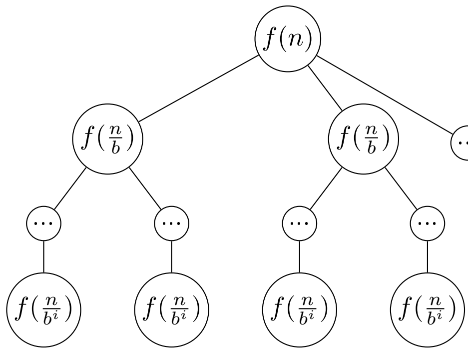
$f(n) = O(n^c), c < \log_b a)$ тогда получаем $T(n) = \Theta(n^{\log ba})$

Пример

$f(n) = \Theta(n^c \log^k n), c = \log_b a$ тогда получаем $T(n) = \Theta(n^c log^{k+1} n)$

Пример

$f(n) = \Omega(n^c), c > \log_b a$



Доказательство Рассмотрим дерево рекурсии и частный случай $n = b^k$
где i - уровень рекурсии, тогда на i - ом уровне получим суммарное количество операций $a^i f(\frac{n}{b^i})$
Тогда

$$T(n) = \sum_{i=0}^{\log_b n} a^i f(\frac{n}{b^i}) + O(n^{\log_b a})$$

$O(n^{\log_b a})$ - для последнего уровня рекурсии
Доказательство для A

$$T(n) = \sum_{i=0}^{\log_b n} a^i f(\frac{n}{b^i}) + O(n^{\log_b a}) \geq \sum_{i=0}^{\log_b n} a^i (\frac{n}{b^i})^{\log_b a - \epsilon} + O(n^{\log_b a}$$

Тогда

$$\sum_{i=0}^{\log_b n} a^i (\frac{n}{b^i})^{\log_b a - \epsilon} = n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n} a^i b^{-i \log_b a} b^{i\epsilon} = n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n}$$

Дописать надо

4 Деревья

Некоторые определения

дерево - связный граф без циклов

корневой узел - узел, не имеющий предков

лист - узел, не имеющий потомков

внутренний узел - узел, имеющий потомков

4.1 Обход дерева

Обход в глубину Для обхода в глубину удобно использовать стек, в который по очереди в зависимости от порядка обхода кладется вершина, ее левый и правый ребенок.

pre-order(прямой обход)

1. Проверяем текущий узел на NULL
2. Выводим значение в текущем узле
3. Обходим прямым обходом левое поддерево
4. Обходим прямым обходом правое поддерево

in-order(центрированный обход)

1. Проверяем текущий узел на NULL
2. Обходим центрированным обходом левое поддерево
3. Выводим значение в текущем узле
4. Обходим центрированным обходом правое поддерево

post-order(обратный обход)

1. Проверяем текущий узел на NULL
2. Обходим обратным обходом левое поддерево
3. Обходим обратным обходом правое поддерево
4. Выводим значение в текущем узле

Обход в ширину - значения в вершинах выводятся по уровням слева направо
Для осуществления такого обхода надо завести очередь и положить в нее корень дерева. Пока очередь непуста, достаем из нее элемет, выводим его значение и кладем в очередь его левого и правого ребенка.

4.2 Двоичное дерево поиска

У каждой вершины не более 2х потомков, любое поддерево так же является деревом поиска, все ключи в левом поддереве меньше, чем в корне, в правом поддереве - больше, чем в корне.

Поиск - по ключу возвращаем ссылку на узел с таким же ключом
Алгоритм

Если текущий узел пуст - вернуть NULL

Если ключ текущего узла равен искомому - вернуть ссылку на текущий узел

Если ключ текущего меньше искомого - уйти в левое поддерево, иначе - в правое

Вставка - рассматриваем случай, когда все ключи различны
Алгоритм

Если дерево пусто - заменить корень на узел с ключом равным данному, завершить работу

Если ключ текущего меньше искомого - уйти в левое поддерево, иначе - в правое

Удаление Алгоритм

Если текущий узел пуст - завершить работу

Если ключ текущего меньше данного - рекурсивно удалить ключ из левого поддерева, иначе - из правого

Если ключ текущего узла равен искомому - рассмотреть случаи

1. текущий узел - лист - удалили и не паримся
2. у текущего узла один ребенок - поставили ребенка вместо текущего узла, текущий удалили
3. у текущего узла два ребенка - возьмем m самый левый узел правого поддерева, удалим его из правого поддерева, поставим вместо текущего ключа ключ m

4.3 Декартово дерево

Курево(куча + дерево) - бинарное дерево, в узлах которого хранятся пары (ключ, приоритет). По ключам курево является двоичным деревом поиска, а по приоритетам - кучей.

split - по дереву T и ключу к split возвращает 2 дерева T1 и T2. В T1 все ключи меньше к, в T2 все ключи больше или равны к.
Псевдокод

```
<Treap*, Treap*> split(Treap* t, Key k){
        if(t == NULL) return <NULL, NULL>;
        if(t->key > k){
                <t1, t2> = split(t->left, k);
                t->left = t2;
                return <t1, t>;
        }else{
                <t1, t2> = split(t->right, k);
                t->right = t1;
                return <t, t2>;
        }
}
```

Время работы алгоритма - O(h), где h - высота дерева, т.к. split рекурсивно проходит по вершинам в глубину дерева и на каждом шаге работает O(1)

merge - по двум куревам T1 и T2, любой ключ в T1 меньше любого ключа в T2, merge строит курево T, в котором есть все ключи из первого и второго дерева.
Псевдокод

```
Treap* merge(Treap* t1, Treap* t2){
        if(t1 == NULL) return t2;
        if(t2 == NULL) return t1;
        if(t1->priority > t2->priority){
                t1->right = merge(t1->right, t2);
                return t1;
        }else{
                t2->left = merge(t1, t2->left);
                return t2;
        }
}
```

Время работы алгоритма - O(h), где h - высота дерева, т.к. split рекурсивно проходит по вершинам в глубину дерева и на каждом шаге работает O(1)

Вставка узла split по ключу вставляемого узла
merge первого курева из возвращенной пары и вставляемого узла
merge этой штуки с оставшимся куревом

Удаление по ключу split по ключу удаляемого узла
из второго курева из возвращенной пары удалим самого левого ребенка
merge этой штуки с оставшимся куревом

Построение - наивно - просто добавлять по порядку, каждая вставка за O(log n), значит общая сложность O(n log n)

Алгоритм за O(n) - если пары отсортированны по ключу по возрастанию
Будем строить дерево слева направо, то есть начиная с (x1,y1) по (xn,yn), при этом помнить последний добавленный элемент (xk,yk). Он будет самым правым, так как у него будет максимальный ключ, а по ключам декартово дерево представляет собой двоичное дерево поиска. При добавлении (xk+1,yk+1), пытаемся сделать его правым сыном (xk,yk), это следует сделать если yk>yk+1, иначе делаем шаг к предку последнего элемента и смотрим его значение y. Поднимаемся до тех пор, пока приоритет в рассматриваемом элементе меньше приоритета в добавляемом, после чего делаем (xk+1,yk+1) его правым сыном, а предыдущего правого сына делаем левым сыном (xk+1,yk+1).
Заметим, что каждую вершину мы посетим максимум дважды: при непосредственном добавлении и, поднимаясь вверх (ведь после этого вершина будет лежать в чьём-то левом поддереве, а мы поднимаемся только по правому). Из этого следует, что построение происходит за O(n).

Теорема про случайные приоритеты по сути про то, что глубина O(n log n)
coming soon...

4.4 АВЛ - дерево

Основное свойство - сбалансированность
Для каждой вершины $|h(l) - h(r)| \leq 1$, где $h(l)$ - глубина левого поддерева, $h(r)$ - глубина правого поддерева

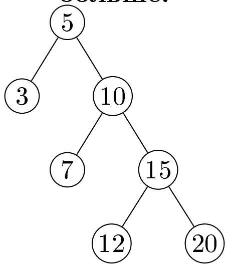
Высота $O(\log N)$
Доказательство - по индукции докажем, что минимальное число вершин в AVL-дереве высоты h: $m_h = F_{h+2} - 1$, где F_h - h-ое число Фибоначчи.
$m_{h+2} = m_{h+1} + m_h + 1$, т.к. разница между высотой детей не больше 1.
Далее доказательство по индукции.
$F_h = \Omega(\phi^h)$, где ϕ - константа.
Получаем $n \geq \phi^h \Leftrightarrow \log_\phi n \geq h$

Как балансировать дерево? - очень просто
Поворот производим, если разница высот равна 2.
Существует два основных типа поворота - правый и левый, но не всегда мы можем сразу выполнить поворот.
Мы можем сделать левый поворот, если высота правого ребёнка больше левого и **высота правого ребёнка правого ребёнка (т.е. самого правого внука изначальной вершины) больше или равна высоте левого ребёнка правого ребёнка**
Правый поворот при аналогичных условиях, т.е. высота левого ребёнка больше правого и

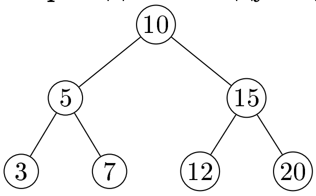
высота левого рёбёнка левого рёбнка (т.е. самого левого внука изначальной вершины) больше или равна высоте правого ребёнка левого ребёнка
Как выполняется поворот? - разберём на примере левого поворота, для правого аналогично
Мы "поднимаем"правого ребёнка наверх, при этом он неожиданно становится родителям троих детей, так у нас не принято, поэтому одного ребёнка нужно отдать другому ребёнку, отличный кандидат для этого - предыдущий левый ребёнок правого ребёнка, т.к. неожиданно у нашей рассматриваемой ноды (для которой мы осуществляли поворот) остался только один ребёнок.

Звучит сложно? Тогда посмотрим картинки.

Выполняем левый поворот. Случай, когда правый ребёнок правого ребёнка больше.

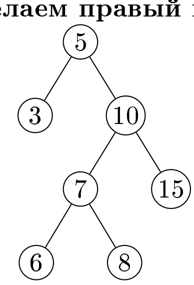


Дерево переходит в следующий вид

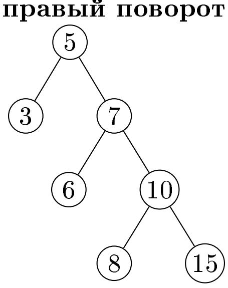


Если же мы не можем сразу сделать поворот, но баланс нарушен, то нужно спуститься к ребёнку (правому, если у нас левый поворот, иначе левому) и сделать поворот в другую сторону уже относительно него, чтобы привести дерево в подходящий вид.
Посмотрим картинки

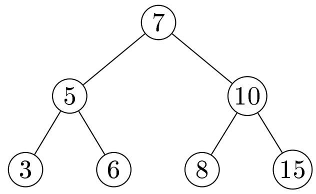
Выполняем левый поворот. Случай, когда левый ребёнок правого ребёнка больше, поэтому сначала делаем правый поворот для правого ребёнка



Для начала сделаем правый поворот для правого ребёнка



Теперь можем спокойно сделать левый поворот



4.5 Красно-черное дерево

двоичное дерево поиска, в котором баланс осуществляется на основе "цвета"узла дерева, который принимает только два значения: "красный"и "чёрный"
При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными. "Объединим"их в один NIL

Свойства

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
5. Чёрный узел может иметь чёрного родителя

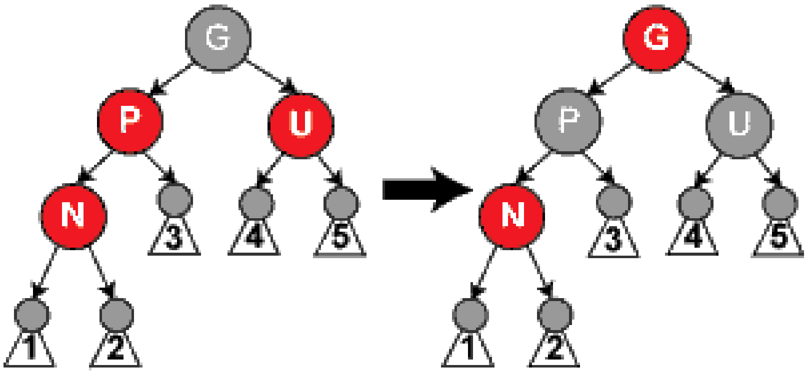
Высота красночерного дерева Черная высота вершины x - число чёрных вершин на пути из x в лист.
Лемма В красно-черном дереве высотой hb количество внутренних вершин не менее $2^{hb-1}-1$ доказывается по индукции по высоте дерева
Теорема Красно-черное дерево с n ключами имеет высоту O(log n)
Рассмотрим красно-чёрное дерево с высотой h. Так как у красной вершины чёрные дети (по свойству 3) количество красных вершин не больше $\frac{h}{2}$. Тогда чёрных вершин не меньше, чем $\frac{h}{2}-1$
По лемме для дерева размера n
$n \leq 2^{\frac{h}{2}}-1$
Прологарифмировав неравенство, имеем:
$h \leq 2log(n+1)$

Вставка Вставка узла производится на мето NIL'a. Вставляем вершину вместо NIL с нулевыми потомками и красим в красный цвет. Балансировка для вершины x

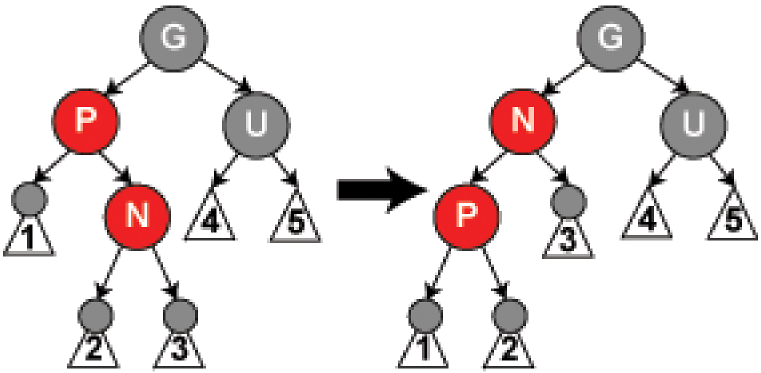
Отец - черный - все ок

Отец - красный - нарушается свойство 3 (при этом отец точно не корень, значит есть и дедушка)

1. Дядя - красный
   Перекрасим отца и дядю в черный, деда в крассный. Поддерево с вершиной x - сбалансированно. Теперь балансируем деда.



2. Дядя - черный
   Пусть о - отец, д - дед, т - дядя вершины х
   Рассмотрим случай, когда о - левый ребенок д. Правый симметрично
   Если х - правый ребенок о, выпоним левый поворот: о становится левым ребенком х, х становится на место о.
   Теперь можем совершить правый поворот, т.е. корнем поддерева станет левый ребенок д, д станет правым ребенком нового корня. Т.к. до вставки х дерево было сбалансированно, д - черная вершина. Перекрасим д в красный, новый корень в черный. Свойства сохранятся



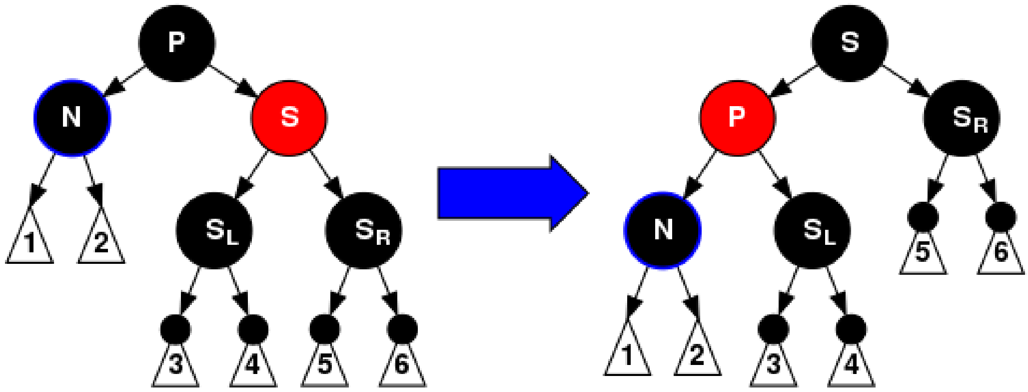
Удаление (видимо тебя из этой жизни)
Удаление в зависимости от количества детей

Если у вершины нет ненулевых детей, то баланируем и изменяем указатель на неё у родителя на NIL

Если у неё только один ненулевой ребёнок, то балансируем и делаем у родителя ссылку на него вместо этой вершины.

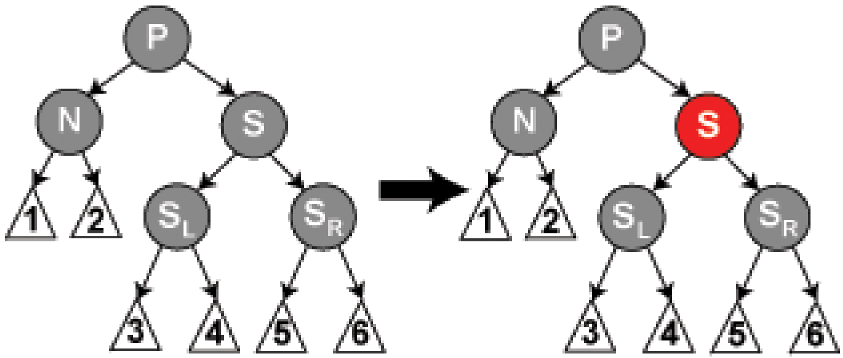
Иначе находим наименьший элемент в правом поддереве(сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ненулевой ребенок), копируем его значение в удаляемый, удаляем его рекурсивно из правого поддерева по 1му или 2му пункту.
Заметим, что балансировать удаляемую вершину, у которой 2 ребенка не надо, т. к. в ней ничего не изменилось, кроме значения, а значит свойства могли сломаться только у наименьшего элемента в правом поддереве, который мы удалили. А дальше - балансировка!
Если вершина была красной, то и балансировать даже не надо. Если она была черной, то если ее ребенок был красным, просто перекрасим его в черный.
Иначе и удаляемая вершина и ее ребенок - черные.
Дальше рассматриваем случай, когда n - левый ребенок. Правый симметрично. Итак, удалили вершину n, на ее месте теперь ребенок n и он черный(нзовем его x). Исходим из того, что брат x - черный.

Если брат x - красный, то совершим левый поворот вокруг нового родителя x, перекрасим родителя в красный, а брата в черный. Сейчас брат стал дедушкой x. Т.к. он был красный, его дети черные, его левый ребенок стал братом x, значит у x сейчас черный брат.

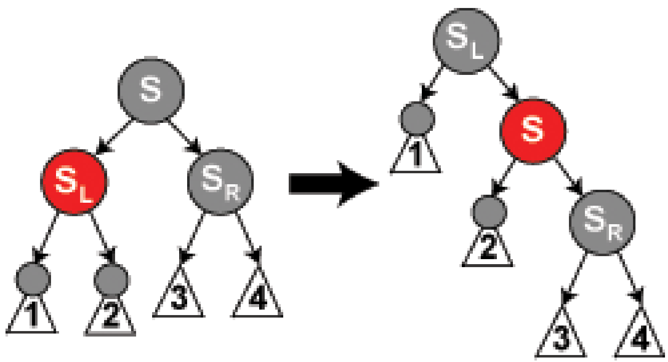


Теперь рассмотрим случаи:

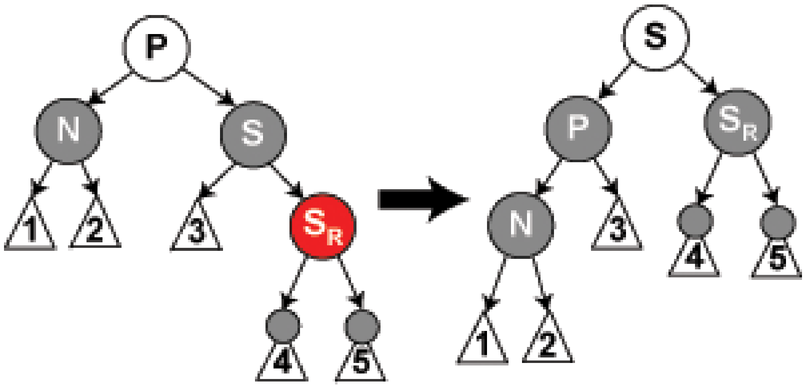
1. Если оба ребенка брата - черные, перекрасим брата в красный. Тогда поддерево с корнем в родителе x сбалансированно. При этом, если родитель x был красным, перекрасим его в черный и дерево окажется сбалансированным, иначе все пути, идущие через родителя x, имеют черную глубину на один меньше, чем все остальные. Тогда запустим балансировку от родителя



2. Если левый ребенок брата - красный, а правый - черный, то совершим правый поворот вокруг брата. Его красный ребенок станет новым отцом. Поменяем цвета нового отца и брата. у x теперь есть черный брат с черным левым и красным правым потомком. Переходим к следующему случаю.



3. Левый ребенок брата - черный, а правый - красный. Совершим поворот относительно отца x влево. Поменяем цвета отца и брата местами, т.е. отец станет черным, а черная высота левого поддерева брата(нового отца) увеличится на 1, а правого уменьшится на 1. Перекрасим правого ребенка брата в черный



Оценка сложности(прямо с wiki, так что придется подкорректировать) Все рекурсивные вызовы функции хвостовые и преобразуются в циклы, так что алгоритм требует памяти O(1). В алгоритме выше, все случаи связаны по очереди, кроме случая 3, где может произойти возврат к случаю 1, который применяется к предку узла: это единственный случай когда последовательная реализация будет эффективным циклом (после одного вращения в случае 3).

Так же, хвостовая рекурсия никогда не происходит на дочерних узлах, поэтому цикл хвостовой рекурсии может двигаться только от дочерних узлов к их последовательным родителям. Произойдет не более, чем O(log n) циклических возвратов к случаю 1 (где n — общее количество узлов в дереве до удаления). Если в случае 2 произойдет вращение (единственно возможное в цикле случаев 1-3), тогда отец узла N становится красным после вращения и мы выходим из цикла. Таким образом будет произведено не более одного вращения в течение этого цикла. После выхода из цикла произойдет не более двух дополнительных поворотов. А в целом произойдет не более трех поворотов дерева.