Конспекты по проге 1 семестр

Никита Евсюков Маргарита Сагитова

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

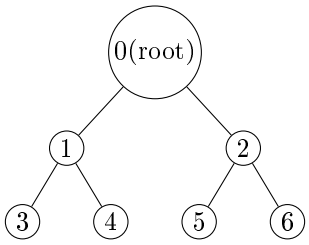
Хранение дека в массиве

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

Задача - упорядочить множества объектов по какому-либо признаку

Классификация

- По времени работы
- По количеству дополнительной памяти
- Устойчивость - сортировка не меняет порядок объектов с одинаковыми ключами
- Локальность - сортировка в том же контейнере
- Детерменированность - каждая операция присваивания, обмена и т.д. не зависит от предыдущих

3.2 Квадратичные сортировки

Пузырьки там всякие

3.3 Сортировка слиянием

- Сложность - $O(n \log n)$
- Дополнительная память - $O(n)$

Принцип - разделяем массив на подмассивы, сортируем их рекурсивно, сливаем

3.4 Сортировка с помощью кучи

- Сложность - $O(n \log n)$
- Дополнительная память - $O(1)$

Принцип - последовательно меняем корень с последним элементом и уменьшаем размер на 1

3.5 Слияние отсортированных массивов с помощью кучи

- Сложность - $O(q \log k)$, где q - суммарная длина всех массивов, k - количество массивов
- Дополнительная память - $O(k)$

Принцип

- Делаем кучу на k элементов
- Сдвигаем указатель минимального элемента и добавляем его в новый массив
- Добавляем элемент по новому указателю в кучу

3.6 Нижняя оценка времени работы для сортировок сравнением

Сопоставим алгоритм сортировки с деревом, в котором листья - конечные перестановки исходного массива, узлы - операции сравнения, а рёбка - переходы между состояниями алгоритма, докажем, что высота такого дерева не меньше чем $\Omega(n \log n)$, где n - количество элементов
Доказательство
Число листьев не меньше общего количества перестановок - $n!$
С другой стороны, двоичное дерево имеет не более 2^n перестановок

$$n! \leq 2^h \Leftrightarrow h \geq \log_2 n! > \frac{n}{2} * \log_2(\frac{n}{2}) = \frac{n}{2}(\log_2 n - 1) = \Omega(n \log n)$$

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

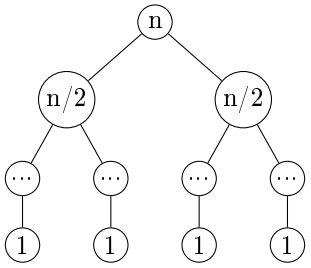
```
q = A[(r + l)/2];
i = l;
j = r;
while (i <= j) {
        while (a[i] < q) {
                i++;
        }

        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

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

Доказательство по индукции, пусть $T(k) \leq ck$ для константы $c \geq 4$ и всех $k < n$ База: $T(1) = 0 < 4$
Тогда верно:

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

Арифметическая прогрессия:

$$\sum_{k=\lfloor n/2 \rfloor]}^{n-1} ck = \frac{1}{2}(\lceil\frac{n}{2}\rceil + c(n-1)) \leq \frac{c}{2}(\frac{n+1}{2} - 1)\frac{3n-2}{2} = c\frac{n-1}{4}\frac{3n-2}{2}$$

Тогда выражение для $T(n)$ принимает вид:

$$\begin{aligned} T(n) \leq n - 1 + \frac{2c(n-1)(3n-2)}{n*4*2} = \\ &= n - 1 + c\frac{n-1}{2n}\frac{3n-2}{2} && \leq \\ &\leq \frac{c}{4}(n-1) + \frac{c}{4}(\frac{n-1}{n} * (3n-2)) && \leq \\ &\leq \frac{c}{4}(n-1+3n-2) = \frac{c}{4}(4n-3) \leq cn \end{aligned}$$

Таким образом $T(n) = O(n)$

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

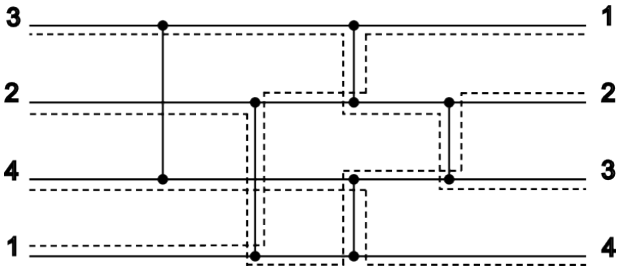
$$T(n) = 2T(\frac{n}{2}) + f(n)$$

где $f(n)$ - затраты на слияние (n)
Тогда:

- Если $f(n) = O(n^{\log_b a - \epsilon})$ для некоторого $\epsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$
- Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$
- Если $f(n) = \Omega(n^{\log_b a + \epsilon})$ для некоторого $\epsilon > 0$ и если $af(\frac{n}{b}) \leq cf(n)$ для некоторой константы $c < 1$ и всех достаточно больших n, то $T(n) = \Theta(f(n))$

По сути, большая из фукнций ($f(n)$ и $n^{\log_b a}$) определяет рекуррентное соотношение
Доказательство в Кормене на 124 странице

3.14 Сортировочные сети



Сортирующая сеть — метод сортировки, основанный только на сравнениях данных. Схематически изображается в виде параллельных прямых (проводов), соединенных вертикальными линиями (компараторами). Особенность сети сортировки в том, что сравнения выполняются независимо от предыдущих. Кроме того, сравнения могут выполняться одновременно.
Компаратор - устройство, подключенное к двум проводам, которое упорядочивает текущие значения на проводах. K-компаратор упорядочивает значения на k проводах.
Глубина компаратора - максимум из количества узлов на входных проводах.
Слой сети — множество компараторов, имеющих одинаковую глубину.
Глубина сети— количество слоев в сети.
Размер сети - количество компараторов в сети.

0-1 принцип - позволяет проверять корректность работы сети за $O(2^n \cdot Comp(n))$

Теорема Сеть компараторов с n входами является сортирующей тогда и только тогда, когда она сортирует 2^n различных последовательностей из 0 и 1.

Доказательство Если функция f - монотонна, то $f(min(x_1, x_2)) = min(f(x_1, f(x_2))$
Пусть f - монотонная функция, а N - сеть компараторов, докажем, что они коммутируют, т.е f(N(a)) = N(f(a)), где a - последовательность $\{a_1, a_2, ..., a_n\}$, а $f(a) = \{f(a_1), ..., f(a_n)\}$
Рассмотрим компаратор C, сортирующий эдементы i и j. C(i) = min(i, j), тогда f(C(i)) = f(min(i, j)) = min(f(i), f(j)) = C(f(i)).
Таким образом, результат не зависит от того, что мы сделаем с отдельным компаратором сначала: применим монотонную функцию или пропустим его через сеть. Те же рассуждения можно провести для всех других индексов, то есть для всей сети тоже верно.
Сам принцип - Пусть сеть N сортирует $a_1, ..., a_n$ в возрастающем порядке и есть последовательность a, которую она не сортирует, тогда найдется такое i, $N(a_i) > N(a_{i+1}$.
Рассмотрим функцию $f(x) = \begin{cases} 0, x < N(a_i) \\ 1, x \geq N(a_i) \end{cases}$
Она монотонна и f($N(a_i)$) = 1, а f($N(a_{i+1})$) = 0, то есть f(N(a)) не отсортирована. Значит и N(f(a)) - не отсортирована, но по предположению теоремы все последовательности из нулей и единиц сеть сортирует правильно, что приводит к противоречию, значит такой последовательности a, которую сеть не сортирует не существует.

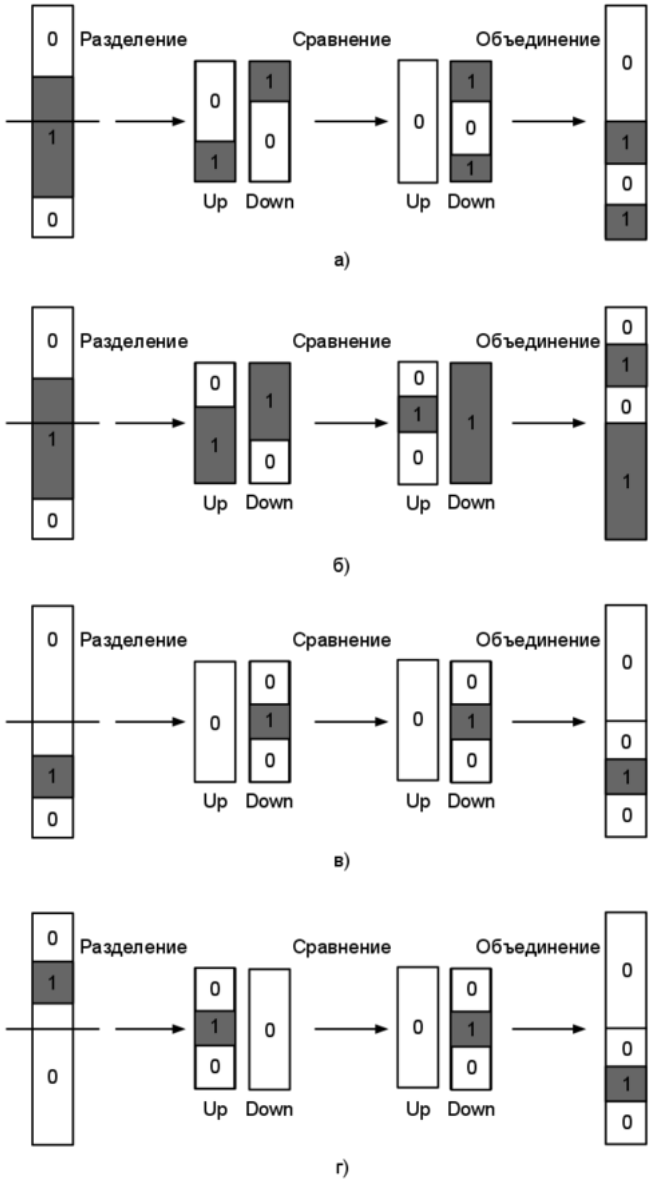
3.15 Сеть Бетчера

Сеть Бетчера (англ. Batcher bitonic mergesort) — сортирующая сеть размером O(nlog^2n) и глубиной O(log^2n), где n — число элементов для сортировки. Здесь рассматривается случай, когда n = 2^k

Битоническая последовательность - конечный упорядоченный набор (кортеж) из вещественных чисел, в котором они сначала монотонно возрастают, а затем монотонно убывают, или набор, который приводится к такому виду путем циклического сдвига.
Приведем несколько примеров нуль-единичной битонической последовательности: 00111000, 11000111, 1110, 1, 000.

Битонический сортировщик - эффективно сортирует все битонические последовательности.
Битонический сортировщик представляет собой каскад так называемых полуфильтров. Каждый полуфильтр — сеть компараторов единичной глубины, в которой i-й входной провод сравнивается со входным проводом с номером $\frac{n}{2}$+i, где i=1,2,. . . ,$\frac{n}{2}$ (число входов n — чётное).

Лемма Если на вход в полуфильтр подать битоническую последовательность из нулей и единиц длиной n, то на выходе мы получим две битонические последовательности длиной $\frac{n}{2}$ такие, что каждый элемент из верхней последовательности не превосходит любой элемент из нижней, и что одна из них будет однородной (англ. clean) — целиком состоящей либо из нулей, либо из единиц.
Доказательство без ограничения общности для случая, когда последовательность имеет вид 0..01..10..0



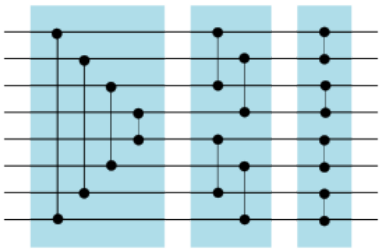
Теперь используем полуфильтры для сортировки битонических последовательностей. Как только что было доказано, один полуфильтр разделяет битоническую последовательность на две равные части, одна из которых однородна, а другая сама по себе является битонической последовательностью, причем части расположены в правильном порядке. Тогда мы можем каждую часть снова отправить в полуфильтр вдвое меньшего размера, чем предыдущий. Затем, если нужно, четыре получившихся части снова отправить в полуфильтры и так далее, пока количество проводов в одной части больше 1.

Поскольку каждый вертикальный ряд полуфильтров вдвое сокращает число входов, которые необходимо отсортировать, глубина всей сети равна $log_2 n$, где n — число входов. Количество компараторов равно $\frac{n(log_2 n)}{2}$, потому что в одном слое $\frac{n}{2}$ компараторов

Объединяющая сеть - сеть компараторов, объединяющая две отсортированные входные последовательности в одну отсортированную выходную последовательность.

Отсортированная последовательность имеет вид 0..01..1. Запишем две входные последовательности. Если перевернуть вторую последовательность, получится отсортированная по невозрастанию последовательность 1..10..0. Если теперь записать первую и перевернутую вторую последовательности подряд, получится битоническая последовательность, которую можно отсортировать в битоническом сортировщике с глубиной O(logn).

Объединяющая сеть является ничем иным как битоническим сортировщиком. Единственное отличие в том, что половина входных проводов расположена в обратном порядке (предполагается, что мы объединяем две сети одинакового размера $\frac{n}{2}$). Поэтому первый полуфильтр будет отличаться от остальных — он будет соединять i-ый провод не с $\frac{n}{2}$+i-ым, а с n-i+1-ым проводом. Схема объединяющей сети для восьми проводов приведена ниже.



Глубина и число компараторов в объединяющей сети очевидно те же, что и в битоническом сортировщике.
Теперь, с помощью описанных выше объединяющих сетей мы построим параллельную версию сортировки слиянием. Чтобы отсортировать всю последовательность, мы сначала отсортируем пары, потом четверки, потом восьмерки и т.д. входных проводов.

Размер и глубина (it's so deep)
Сеть состоит из $log_2 n$ "слоев"объединяющих сетей. Такая сеть из слоя с номером i имеет глубину $log_2 2^i = i$, потому что объединяет 2^i проводов. Тогда глубина всей сети $\sum_{i=1}^{log_2 n} i = \frac{(1+log_2 n)log_2 n}{2} = O(log^2 n)$
Размер: в каждом слое сети $\frac{n}{2}$ компараторов => во всей сети $O(nlog^2 n)$

4 Деревья

4.1 Некоторые определения

дерево - связный граф без циклов

корневой узел - узел, не имеющий предков

лист - узел, не имеющий потомков

внутренний узел - узел, имеющий потомков

4.2 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.

4.3 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 больше 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 больше искомого - уйти в левое поддерево, иначе - в правое

Удаление Алгоритм

- Если текущий узел пуст - завершить работу
- Если ключ текущего больше 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

4.4 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 k split возвращает 2 дерева T1 и T2. В T1 все ключи меньше k, в T2 все ключи больше или равны k.
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 - merge - merge split по ключу вставляемого узла
merge первого курева из возвращенной пары и вставляемого узла
merge этой штуки с оставшимся куревом

Удаление по ключу split - merge 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...

4.5 АВЛ - дерево

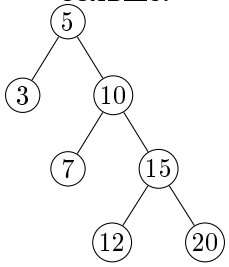
Основное свойство - сбалансированность
Для каждой вершины $|h(l) - h(r)| \leq 1$, где $h(l)$ - глубина левого поддерева, $h(r)$ - глубина правого поддерева

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 $n \geq \phi^h \Leftrightarrow \log_\phi n \geq h$

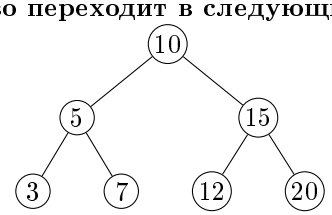
Как балансировать дерево? - очень просто
Поворот производим, если разница высот равна 2.
Существует два основных типа поворота - правый и левый, но не всегда мы можем сразу выполнить поворот.
Мы можем сделать левый поворот, если высота правого ребёнка больше левого и **высота правого ребёнка правого 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 ребёнка 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.



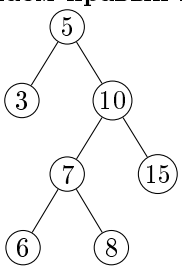
Дерево переходит в следующий вид



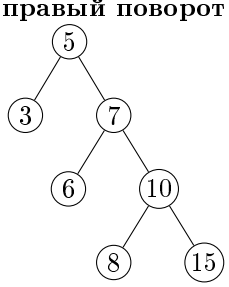
Если же мы не можем сразу сделать поворот, но баланс нарушен, то нужно спуститься к ребёнку (правому, если у нас левый поворот, иначе левому) и сделать поворот в другую сторону уже относительно него, чтобы привести дерево в подходящий вид.

Посмотрим картинки

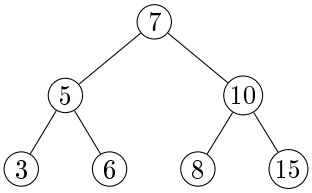
Выполняем левый поворот. Случай, когда левый ребёнок правого ребёнка больше, поэтому сначала делаем правый поворот для правого ребёнка



Для начала сделаем правый поворот для правого ребёнка



Теперь можем спокойно сделать левый поворот



Операции - вставка

Ищем место для вставки как в обычном бинарном дереве, вставляем, далее запускаем балансировку от места вставки

Удаление

Ищем элемент для удаления как в обычном бинарном дереве, далее 3 случая:

- Вершина без детей - тупо вырезаем, обнуляем указатели родителя на неё
- Вершина с ребёнком - вырезаем, подцепляем ребёнка с родителю
- Вершина с двумя детьми - находим ближайшую вершину для удаляемой (например, самую правую в левом поддереве), меняем значения, запускаем удаление от найденной вершины

4.6 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 - определение 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
$n \geq 2^{\frac{h}{2}}-1$
Прологарифмировав неравенство, имеем:
$h \geq 2log(n+1)$

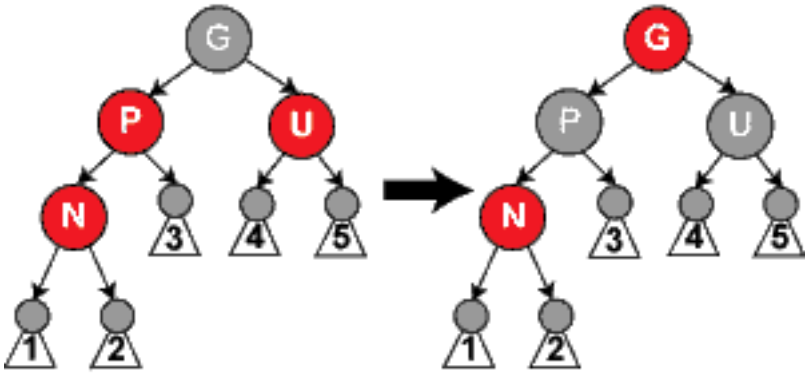
Вставка Вставка узла производится на место NIL'а. Вставляем вершину вместо NIL с нулевыми потомками и красим в красный цвет.

Балансировка для вершины x

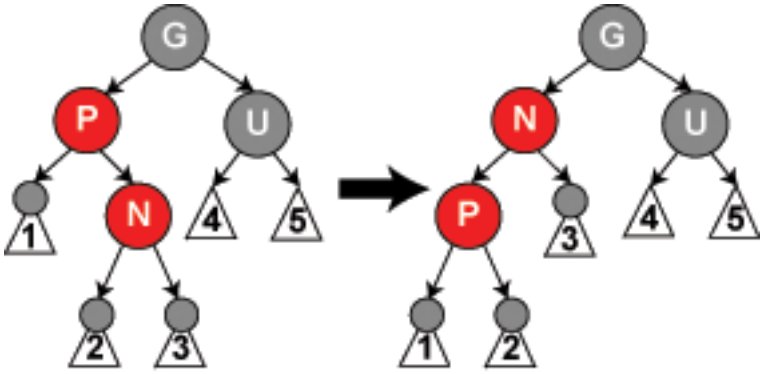
Отец - черный - все ок

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



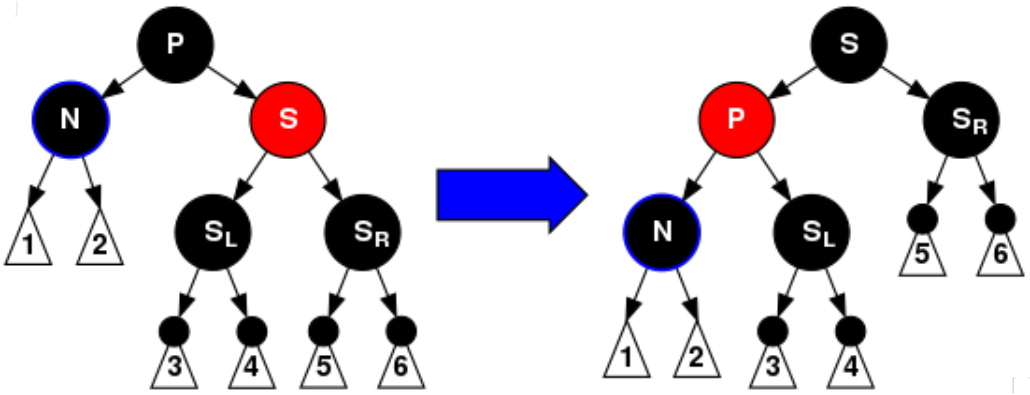
Удаление (видимо тебя из этой жизни)

Удаление в зависимости от количества детей

1. Если у вершины нет ненулевых детей, то баланируем и изменяем указатель на неё у родителя на NIL
2. Если у неё только один ненулевой ребёнок, то балансируем и делаем у родителя ссылку на него вместо этой вершины.
3. 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.

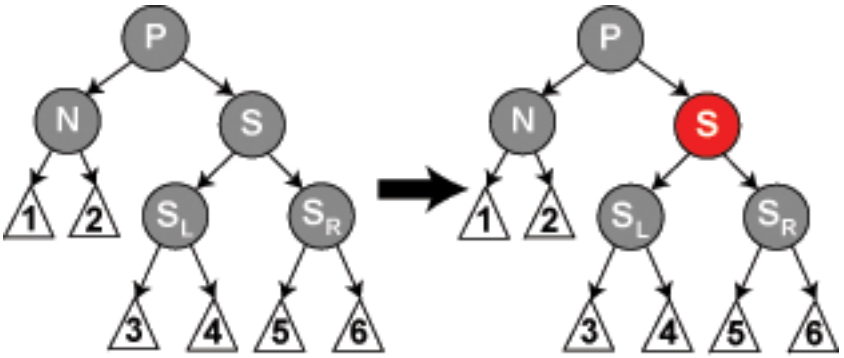
Заметим, что балансировать удаляемую вершину, у которой 2 ребенка не надо, т. к. в ней ничего не изменилось, кроме значения, а значит свойства могли сломаться только у наименьшего элемента в правом поддереве, который мы удалили. А дальше - балансировка!
Если вершина была красной, то и балансировать даже 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(назовем 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.

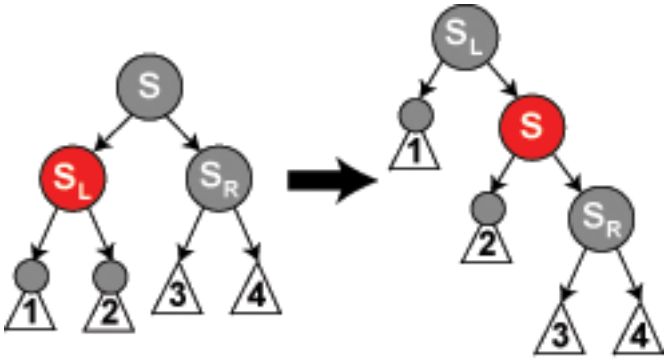


Теперь рассмотрим случаи:

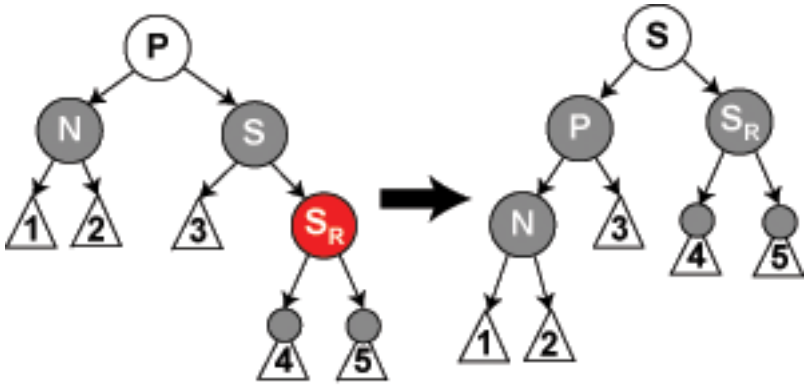
1. Если оба ребенка брата - черные, перекрасим брата в красный. Тогда поддерево с корнем в родителе x сбалансированно. При этом, если родитель x был красным, перекрасим его в черный и дерево окажется сбалансированным, иначе все пути, идущие через родителя x, имеют черную глубину на один меньше, чем все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

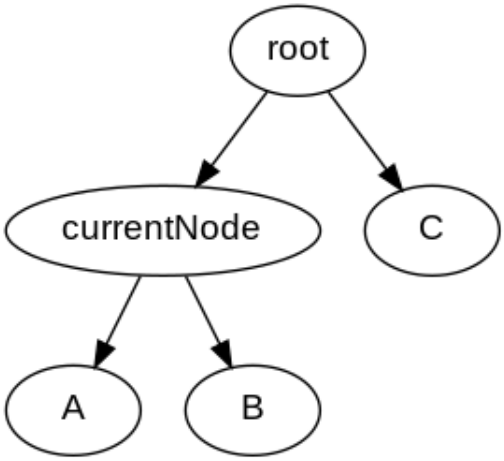
Оценка сложности (альтернативная) - **вставка**
Только при чёрном дяде мы выполняем балансировку для деда, таких переходов будет не больше $O(h)$, где h -высота дерева, т.е. вставка за $O(\log n)$
Удаление
Только в случае чёрного брата с чёрными детьми мы запускаем балансировку от родителя, таких переходов будет не больше $O(h)$, где h -высота дерева, т.е. вставка за $O(\log n)$, в остальных случаях завершение происходит после выполнения фиксированного числа изменений цвета и не более трёх поворотов

4.7 Сплей-дерево

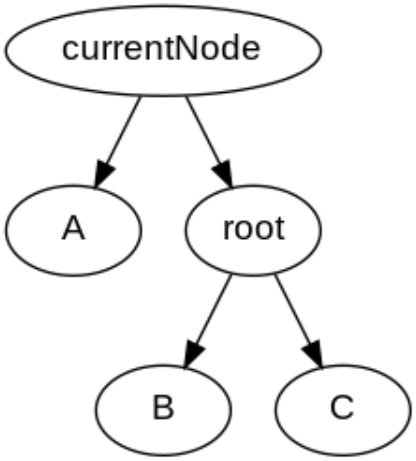
О что это? - двочиное дерево поиска, которое обеспечивает быстрый доступ к данным, которые недавно использовались, дерево самобалансируется так, что поддерживает сложность $O(\log n)$ для всех операций (поиск, вставка, удаление), суть в том, что операции добавления и поиска отправляют использованную вершину в корень, но как?

Операция splay - операция расширешия
Этой операцией мы отправляем вершину в корень с помощью комбинации трёх отдельных операций:

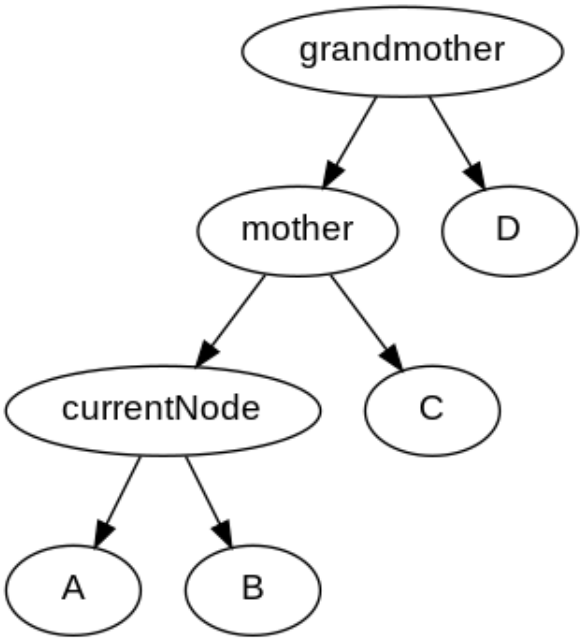
- Zig - выполняем, если родитель вершины - корень, по сути, обычный левый/правый поворот соответственно в зависимости от того, каким ребёнком является наша вершина
  Пример: до



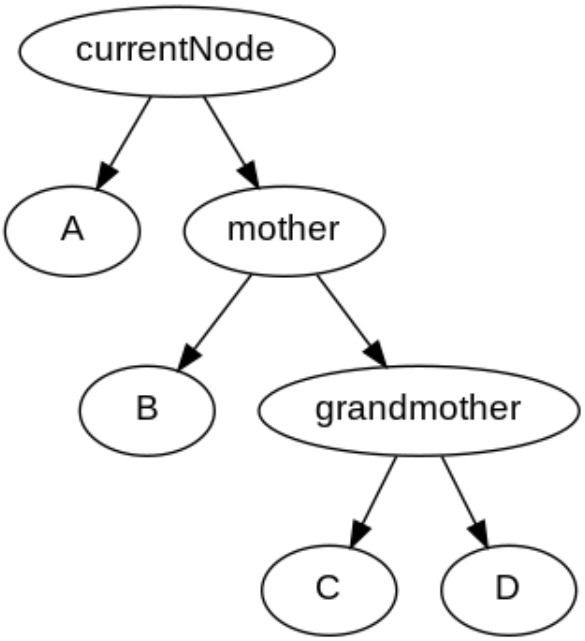
После



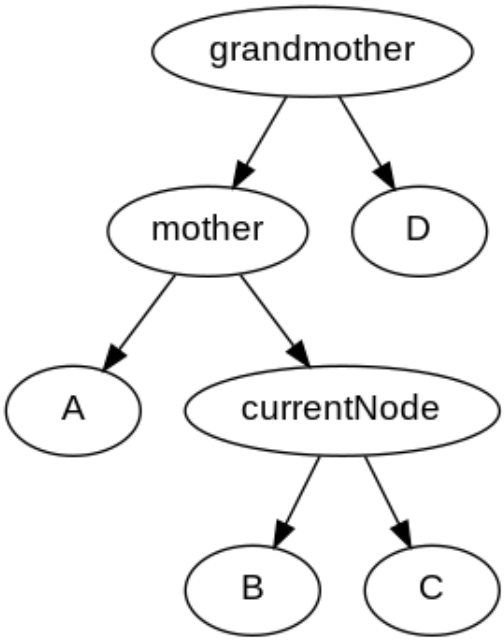
- Zig-Zig - два простых поворота, выполняются, если и текущая вершина, и её родитель являются одним и тем же типом ребёнка (например, оба левые, тогда сначала производим правый поворот относительно деда, потом относительно родителя)
  Пример: до



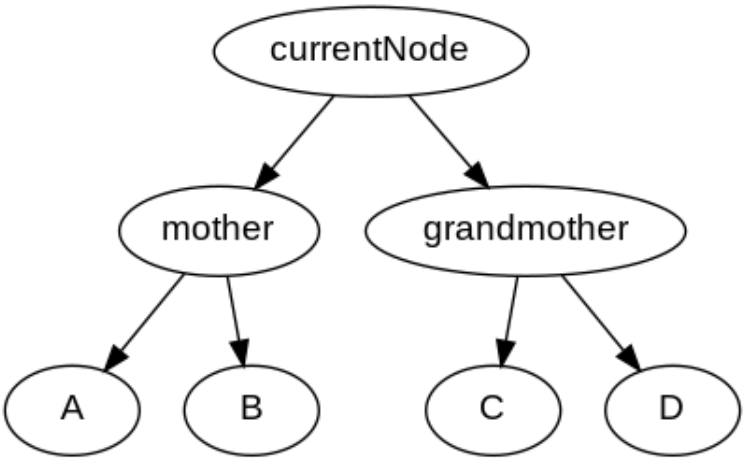
После



- Zig-Zag - большой левый или большой правый поворот, выполняется в тех же случаях, что и для AVL дерева
  Пример: до



После



Основные операции

- Search - ищем как в бинарном дереве, потом запускаем splay от найденной вершины
- Split - находим ближайший элемент к ключу, делаем для него splay, отрезаем левого ребёнка и возвращаем его вместе с корнем
- Merge - находим самый правый элемент в первом дереве, делаем для него splay, подцепляем справа второе дерево
- Insert - делаем split по вставляемому ключу, потом подцепляем полученные деревья к новой вершине
- Delete - ищем элемент как в бинарном дереве, запускаем от него splay, возвращаем merge его детей

4.8 B - деревья

B-дерево - сбалансированное сильно ветвящееся дерево поиска. В каждом узле хранятся массивы ключей. t - параметр дерева.

Применение Структура B-дерева применяется для организации индексов во многих современных СУБД.

B-дерево может применяться для структурирования (индексирования) информации на жёстком диске (как правило, метаданных). Время доступа к произвольному блоку на жёстком диске очень велико (порядка миллисекунд), поскольку оно определяется скоростью вращения диска и перемещения головок. Поэтому важно уменьшить количество узлов, просматриваемых при каждой операции. Использование поиска по списку каждый раз для нахождения случайного блока могло бы привести к чрезмерному количеству обращений к диску вследствие необходимости последовательного прохода по всем его элементам, предшествующим заданному, тогда как поиск в B-дереве, благодаря свойствам сбалансированности и высокой ветвистости, позволяет значительно сократить количество таких операций.

Относительно простая реализация алгоритмов и существование готовых библиотек (в том числе для C) для работы со структурой B-дерева обеспечивают популярность применения такой организации памяти в самых разнообразных программах, работающих с большими объёмами данных.

Принцип построения

1. Ключи в узле упорядочены.
2. Корень содержит от 1 до 2t-1 ключей.
3. Любой другой узел содержит от t-1 до 2t-1 ключей.
4. Любой узел кроме листа, содержащий n: $k_1, k_2, ..., k_n$ ключей, содержит n потомков. При этом:
   - Первый потомок содержит ключи меньшие k_1
   - Для $2 \leq i \leq n$ i-й потомок содержит ключи из инетрвала (k_{i-1}, k_i)
   - Последний потомок содержит ключи большие k_n
5. Глубина всех листьев одинакова.

Поиск Если ключ содержится в данной вершине, он найден, иначе переходим к интервалу между двумя ключами данной вершины, в которой лежит искомый ключ и идем к соответствующему потомку.

Вставка Ищем лист, в который можно добавить ключ. Если в найденном листе меньше 2t-1 ключей, добавляем в него наш ключ.
Иначе после добавления нашего ключа в листе оказывается 2t ключей. Теперь разбиваем узел на 2 узла - первые t ключей и последние t-1 ключей. Оставшийся средний элемент вставлям в родительский узел между ключами, между которыми находился найденный лист.
Если родительский узел заполнен, повторяем разделение, пока не найдем незаполненный узел или не дойдем до корня.
Если дошли до корня, то разобьем его на 2 узла, а средний ключ корня сделаем новым корнем.

Удаление Ищем узел, в котором находится удаляемый ключ. Возможны два случая:
Удаление из листа
Если в узле больше t-1 ключей, просто удаляем наш.
Иначе, если найдется соседний лист с тем же родителем, в котором больше t-1 ключей, выберем из соседа ближайший к нашему листу ключ, поместим его в "разделяющий"элемент родителя, а этот элемент спустим в наш лист. Удалим из листа нужный ключ.
Если соседей - буржуев не оказалось, то листы - пролетарии соединяются... Объединяем наш лист и одного из его соседей, удаляем наш ключ, перемещаем разделяющий ключ из родителя в новый лист.
Удаление из внутреннего узла
Если левый(правый) дочерний узел для ключа, который мы хотим удалить, содержит более t-1 ключей, перемещаем из него крайний правый(левый) ключ на место удаляемого.
Иначе в обоих дочерних узлах t-1 ключей. Тогда соединяем их, а удаляемый ключ, просто удаляем, располагая указаатель на новый узел там, где был этот ключ.
Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается.

4.9 Алгоритм Хаффмана

Определение - алгоритм оптимального префиксного кодирования алфавита. Пусть $A = a_1, .., a_n$ - алфавит из n различных символов, $W = w_1, .., w_n$ - набор весов. Тогда бинарный набор кодов $C = c_1, .., c_n$, где c_i - код символа a_i, такой, что:

- c_i не является префиксом другого кода
- сумма произведений длинн кодов на их частоту минимальна

Построение дерева кодирования за $O(n)$

- Сортируем массив частот
- Создаём второй массив, заполненный бесконечностями
- Заводим указатели на начало
- Добавляем во второй массив сумму двух минимальных элементов среди 4 элементов (первые 2 элемента первого массива и первые 2 элемента второго)
- Сдвигаем указатели

Теорема об оптимальности префиксного кода алгоритма Хаффмана
Лемма 1. Докажем, что для алфавита существует оптимальный префиксный код, в котором два символа с самыми низкими частотами имеют одинаковую длину и отличаются лишь последним битом
Докажем возможность преобразовать дерево с произвольным оптимальным префиксным кодом в другое дерево с оптимальным префиксным кодов, в котором данные символы (x и y являются листьями с общим родителем, причем они находятся на максимальной глубине
Пусть a и b - листья с общим родителем на максимальной глубине в оптимальном дереве T.
Пусть $a.freq \leq b.freq$ и $x.freq \leq y.freq$. Будем считать, что $x \neq b$.
Пусть мы получили дерево T' перестановкой a и x и дерево T'' перестановкой b и y
Тогда разность стоимостей T и T' равна:

$$\begin{aligned} B(T) - B(T') = \\ &= \sum_{c \in C} c.freq * h_T(c) - \sum_{c \in C} c.freq * h_{T'}(c) && = \\ &= x.freq * h_T(x) + a.freq * h_T(a) - x.freq * d_{T'}(x) - a.freq * d_{T'}(a) && = \\ &= x.freq * h_T(x) + a.freq * h_T(a) - x.freq * h_T(a) - a.freq * h_T(x) && = \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

Аналогично второй обмен не увеличивает стоимости, т.е. $B(T') - B(T'') \Leftrightarrow B(T'') \leq B(T)$, но T - оптимальное дерево $\Leftrightarrow B(T) \leq B(T'') \Leftrightarrow B(T'') = B(T)$, т.е. T'' - оптимальное дерево

Из **леммы 1** следует, что построение оптимального дерева путём объединения узлов можно начать с жадного выбора, при котором объединяются два символа с наименьшими частотами

Теперь докажем **лемму 2** про объединение узлов

Пусть у нас есть алфавит C, в котором символы x, y - символы с минимальными частотами и алфавит C', полученный из C удалением символов x, y и добавлением нового символа z ($z.freq = x.freq + y.freq$). Пусть T' - дерево оптимального префиксного кода для алфавита C'. Тогда дерево T, полученное из дерева T' заменой листа z внутренним узлом с детьми x, y, представляет оптимальный префиксный код для C

Выразим $B(T)$ через $B(T')$

Для всех символов кроме x, y из C высота h не изменилась, учитывая $h_T(x) = h_T(y) = h_{T'}(z) + 1$:

$$x.freq * h_T(x) + y.freq * h_T(y) = (x.freq + y.freq)(h_{T'}(z) + 1) = z.freq * h_{T'}(z) + (x.freq + y.freq)$$

$$B(T) = B(T') + x.freq + y.freq \Leftrightarrow B(T') = B(T) - x.freq - y.freq$$

Пусть дерево T не является деревом оптимального префиксного кода, тогда $\exists T'' : B(T'') < B(T)$

При чём из **леммы 1** мы получаем, что x, y можно считать детьми одного и того же узла T''

Рассмотрим дерево T''', которое получено из дерева T'' заменой родителя x, y на лист z с частотой $z.freq = x.freq + y.freq$:

$$B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T')$$

Противерочие, т.к. T' - дерево оптимального кода для C'

Из **лемм 1 и 2** следует, что алгоритм построения дерева Хаффмана с жадным выбором позволяет получить оптимальное дерево

5 Просто 4й модуль

5.1 Хеш - таблицы

Хеш-таблица - структура данных, позволяющая выполнять операции поиска, вставки и удаления элемента в среднем за O(1) (реализующая интерфейс ассоциативного массива).

Общая схема работы

1. По ключу вычисляется хеш-функция (возващает ключ - номер элемента в массиве)
2. По номеру пытаемся сделать операцию, если произошла коллизия(ячейка уже занята) - разрешаем ее

Хеш - функция - превратит входной ключ в фарш.

Хеш-функция — преобразование по детерминированному алгоритму входного массива данных произвольной длины (один ключ) в выходную битовую строку фиксированной длины (значение). Результат вычисления хеш-фукнции называют «хешем».

Виды хеш-функций(M - размер массива)

1. Деление с остатком
   Хеш-функция может вычислять «хеш» как остаток от деления входных данных на M. При этом не следует использовать в качестве M степень основания системы счисления компьютера, т.к. хеш в таком случае будет зависеть только от нескольких последний битов входных данных, что приведет к большому количеству коллизий. Рекомендуется выбирать простое M.

2. Мультипликативная схема хешировнаия
   $h(k) = [M \cdot \{k \cdot A\}]$, где $0 < A < 1$
   Кнут предложил выбирать в качестве A число как можно ближе к $\phi^{-1} = \frac{\sqrt{5}-1}{2}$
   Пусть $M = 2^p$, а $\omega = 2^{32}$ - длина машинного слова ($M < \omega$).
   Возьмем $A = \frac{2654435769}{\omega}$
   $h(k) = [2^p \cdot \{\frac{r_1 \cdot 2^{32} + r_0}{2^{32}}\}] = [2^p \cdot \frac{r_0}{2^{32}}] = [\frac{r_0}{2^{32-p}}]$
   Таким образом хеш-функция вернет первые p бит r_0
   $h(k) = (k \cdot s \bmod 2^{32}) >> (32 - p)$

3. Полиномиальное хеширование
   Пусть входные данные можно представить как $\{k_0, k_1, ..., k_n\}$ - некоторые числа. Тогда поделим с остатком многочлен $K(x) = k_0 + k_1 x + ... + k_n x^n$ на заранее определенный для хеш-функции порождающий многочлен $P(x) = p_0 + ... + p_m x^m$
   Тогда пусть хеш-функция возвращает коэфициенты многочлена, полученного при делении K на P с остатком.

4. Хеш для строки
   Если есть строка $s = s_0, s_1, ..., s_n$, то определим хеш-функцию $h(s) = (s_0 a^n + s_1 a^{n-1} + ... + s_n) \bmod M$
   Если выбрать a и M взаимно простыми, то при изменении любого символа строки изменится и хеш(доказывается изходя из того, что для взаимно простыъ a и M $\{s \cdot a \bmod M \mid 0 \le s < M\} = \{0, 1, ..., M-1\}$)
   Чтобы сократить количество операций сложения и умножения используют схему Горнера, т.е. представляют $h(k) = (((s - 0a + s_1)a + s_2)a + ... + s_{n-1})a + s_n$

Разрешение коллизий методом цепочек Хеш-таблица представлена массивом, ячейка i которого хранит указатель на начало списка всех элементов, хеш-код которых равен i.
Если мы хотим найти элемент, вычисляем хеш и проходим по списку в ячейке с номером, равным хешу. Хотим вствить/удалить - вставляем/удаляем в список.
Соответственно, время работы поиска в худшем случае O(n), в среднем O(1)

Разрешение коллизий методом прямой адресации Все элементы непосредственно хранятся в массиве. При таком подходе таблица может оказаться полностью заполненна, поэтому возникнет необходимость увеличивать размер таблицы и перехешировать уже имеющиеся элементы. В реальности, т.к. по мере роста заполненности таблицы вероятность коллизии резко возрастает, расширение таблийы производят не при полном заполнении, а при заполненности $\frac{n}{m} > \alpha$, где α - заданное число меньше 1.
При поиске(удалении)элемента вычисляем его хеш, переходим в нужную ячейку и если она не содержит нужный элемент делаем последовательность проб - вычисляем новые номера ячеек, где мог бы находиться нужный элемент. Пока мы не находим нужный ил иячейка не оказвается пуста продолжаем, после выводим результат поиска.
При вставке пробы делаем, пока ячейка, куда мы хотим вставить элемент не пуста.
При этом может оказаться, что мы удалили элемент, а потом хотим найти элемент с таким же хешем, который был добавлен позже, т.е. лежит дальше ("через несколько проб"). В таком случае нам нужно, чтобы при поиске и удалении мы не останавливались на ячейках, где лежали удаленные элементы, а при вставке нам не важно было ли раньше что-то в той ячейке, куда мы хотим вставить, поэтому для ячеек мы заводим метку deleted, чтобы понимать, было ли удалено что-то из данной ячейки.

Последовательности проб Пробирование - вычисление последовательности индексов, по которым располагаются элементы с определенным хешем.

1. **Линейное проброование**
   ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным

интервалом k между ячейками (обычно k = 1), то есть i-й элемент последовательности проб — это ячейка с номером (hash(x) + ik) mod N. Для того, чтобы все ячейки оказались просмотренными по одному разу, необходимо, чтобы k было взаимно-простым с размером хеш-таблицы.

2. **Квадратичное пробрование**
   интервал между ячейками с каждым шагом увеличивается на константу. Если размер хеш-таблицы равен степени двойки (N = 2p), то одним из примеров последовательности, при которой каждый элемент будет просмотрен по одному разу, является: hash(x) mod N, (hash(x) + 1*1) mod N, (hash(x) + 2*2) mod N, (hash(x) + 3*3) mod N, ...

3. **Двойное хеширование**
   интервал между ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей. Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы, что проще всего достичь, взяв простое число в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от 1 до N — 1.

Проблема кластеризации При использовании первых двух способов пробирования часто возникают кластеры.
Кластер - последовательность занятых ячеек. В ячейку кластера может попасть элемент с любым хешем на некоторой пробе. При образовании такой последовательности следующий попадающий в ячейку кластера элемент в итоге проходит по ячейкам кластера и увеличивает его, тогда операции над хеш таблицей начинают выполняться за линейное время.

6 Жадные алгоритмы, ДП

6.1 Общая идея жадных алгоритмов

Идея в том, что на каждом шагу алгоритма мы выбираем локально наилучший выбор в надежде, что итоговое решение будет оптимальным. Пример: Дейкстра, задача о раписании
Такой алгоритм применим к задачу, есои последовательность локально оптимальных выборов даёт глобально оптимальное решение и оптимальное решение задачи содержит в себе оптимальные решения для всех её подзадач

6.2 Динамическое программирование, идея

Общий принцип - динамическое программирование обычно применяется к задачам оптимизации, когда среди множества возможных решений необходимо выбрать оптимальное. Идея в том, что оптимальное решение подзадач меньшего размера может быть использовано для решения исходной задачи.
Основные свойства

- Задача разбивается на подзадачи
- Подзадачи **немного** меньше задачи
- Подзадачи решаются тем же методом
- Имеется процесс консолидации (т.е. для решения задачи используются решения подзадач)
- Имеется порядок на подзадачах (т.е. мы знаем, какие подзадачи решаем раньше, и нет такого, что в решении текущей задачи используется подзадача, для решения которой нужна текущая задача)

- Часть подзадач совпадает

Процесс разработки алгортма можно разделить на этапы:

- Разбиение задачи на подзадачи меньшего размера
- Нахождение оптимального решения подзадач рекурсивно
- Использование полученного решения подзадач для конструирования решения исходной задачи

Нисходящее ДП - обычная рекурсивная реализация, когда подзадача испольует в своём решении данные о решении следующей подзадачи, обычно такая реализация используется вместе с запоминанием (кэшированием)

Идея запоминания - перед непосредственным вычислением функция проверяет, не была ли решена конкретная подзадача до этого

Восходящее ДП - принцип построения решения так, что к моменту, когда мы непосредственно столкнёмся с задачей, все подзадачи будут решены
А теперь на конкретных задачах

6.3 Вычисление числа Фибоначчи

Легко решить с помощью восходящего ДП (также можно с помощью нисходящего с запоминанием)

```
size_t F[n];
F[0] = 1;
F[1] = 1;
for(int i = 2; i < n; i++) {
        F[i] = F[i - 1] + F[i - 2];
}
```

6.4 Задача о рюкзаке

Пусть есть набор предметов $n_1, n_2, ..., n_k$, где n_i - стоимость i-ого предмета, k - количество предметов из набора, которые мы берём, s - вместимость рюкзака
Решается динамическим соотношением $A(k, s) = max(A(k-1, s), A(k-1, s-w_k) + p_k)$, где w_k - вес k-ого предмета, p_k - стоимость k-ого предмета
Восстановить решение можно от ответа, если сверять значение $A(i, w)$ с $A(i-1, w)$ и $A(i-1, w-w_i) + p_i$

6.5 Нахождение наибольшей возрастающей подпоследовательности

Решение за $O(n^2)$
Построим массив d, где $d[i]$ - это длина наибольшей возврастающей подпоследовательности, которая оканчивается в элементе с индексом i, будем заполнять массив постепенно, тогда $d[i] = 1 + max_{j=0..(i-1)}d[j]$, при условии, что $a[j] < a[i]$, т.е. мы выбираем максимальную возврастающую подпоследовательность с учётом того, что подпоследовательности в некоторых случаях мы можем продлить
Чтобы получить саму последовательность индексов, нам нужно восстановить решение, для этого заведём вспомогательный массив $prev$: $prev[i]$ = индекс элемента в исходном массиве,

при котором достигается $d[i]$
Решение за $O(n \log n)$
Модифицируем массив d, пусть теперь $d[i]$ - наименьшее число, на которое оканчивается возрастающая последовательность длины i, тогда $\forall i d[i-1] \leq d[i]$, к тому же каждый элемент $a[i]$ обновляет максимум один элемент $d[j]$, тогда при обработке очередного $a[i]$ будем бин. поиском по d искать первое число, которое больше или равно $a[i]$ и обновлять его (т.е. если $d[j-1] < a[i] \& a[i] < d[j] \Leftrightarrow d[j] = a[i]$
Для восстановления решения будем поддерживать два массива: pos и $prev$, в $pos[i]$ будет индекс элемента, на котором заканчивается оптимальная подпоследовательность длины i, а в $prev[i]$ - позиция предыдущего элмента для массива a

6.6 Нахождение количества разбиений числа на слагаемые

Алгоритм за $O(n^2)$ - есть ещё за $O(n^{\frac{3}{2}})$, но там сложна
Для решения составим следующее динамическое соотношение: $P(n,k) =$

- $P(n,k-1) + P(n-k,k)$, если $0 < k \leq n$
- $P(n,n)$, если $k > n$
- 1, если $n = 0, k = 0$
- 0, если $n \neq 0, k = 0$

$P(n,k-1)$ - в случае, если мы не берём слагаемое k, $P(n-k,k)$ - в случае, если берём, если же все слагаемые должны быть различны, то эта величина считается как $P(n-k,k-1)$
Тогда общее колчиество всех разбиений числа n равно $P(n,n)$

6.7 Нахождение наибольшей общей подпоследовательности за $O(nm)$

Пусть даны последовательность x, y, необходимо найти их наибольшую общую подпоследовательность
Обозначим за $LSC[i][j]$ - наибольшую общую последовательность для префиксов длины i и j последовательностей x и y соответственно, тогда возьмём следующее динамическое соотношение: $LSC[i][j] =$

- 0, если $(i = 0)|(j = 0)$
- $LSC[i-1][j-1] + 1$, если $x[i] = y[j]$
- $max(LSC[i][j-1], LSC[i-1][j])$, если $x[i] \neq y[j]$

Для восстановления решения достаточно завести дополнительный массив $prev[][]$, для которого $prev[i][j]$ - пара индексов элемента таблицы, которое соответствует оптимальному решению вспомогательной задачи, выбранной для $LSC[i][j]$

6.8 Методы восстановления ответа

Для каждой задачи свой способ восстановления ответа, но можно рассмотреть самые распространённые

- Рядом со значением состояния динамики храним полный ответ на подзадачу, такой способ требует большое количества памяти в случае, если ответ занимает много памяти (справедливо)
- Восстанавливать ответ, зная предка данного состояния
- Пойти с конца по лучшему пути и составлять ответ (без использования дополинтельной памяти)

6.9 Расстояние Левенштейна

О что же это? - минимальное количество операций вставки одного символа, удаления одного символа и замены символа на другой, необходимые для превращения одной строки в другую

Немного соотношений : $D(S_1, S_2)$ - расстояние Левенштейна между строками $S1$ и $S2$, $|S|$ - длина строки S

- $D(S_1, S_2) \geq ||S_1| - |S_2||$
- $D(S_1, S_2) \leq max(|S_1|, |S_2|)$
- $D(S_1, S_2) = 0 \Leftrightarrow S_1 = S_2$

Расстояние Левенштейна является частным случаем, при котором цены вставки, удаления и замены символов равны ($deleteCost = insertCost = replaceCost = 1$)
В общем случае решение задачи можно посчитать следующим соотношением (i - длина префикса S_1, j - длина префикса S_2): $D(i, j) =$

- 0, если $(i = -1)\&(j = -1)$
- $i * deleteCost$, если $(j = -1)\&(i \leq 0)$
- $j * insertCost$, если $(i = -1)\&(j \geq 0)$
- $D(i - 1, j - 1)$, если $S_1[i] = S_2[j]$
- $min(D(i, j - 1) + insertCost, D(i - 1, j) + deleteCost, D(i - 1, j - 1) + replaceCost$, если $(i \leq 0)\&(j \leq 0)\&(S_1[i] \neq S_2[j])$

Доказательство корректности
Всего для функции $D(i, j)$ мы получаем 5 случаев, с первыми тремя всё понятно, рассмотрим последние два
Пусть у нас есть строка S_1 с последним символом a и строка S_2 с последним символом b, тогда рассмотрим случаи:

- В какой-то момент мы стёрли a, тогда что мешает нам сделать это прямо сейчас? На это мы потратили $deleteCost$, и нам осталось превратить $i - 1$ символов S_1 в S_2, на что и уходит $D(i - 1, j)$ операций
- В какой-то момент мы добавили b и потратили на это $insertCost$, тогда сделаем это нашей последней операцией, перед ней мы превратим i элементов S_1 в $j - 1$ элементов S_2, на что ушло $D(i, j - 1)$ операций
- Пусть мы не стирали a и не добавляли b, тогда мы либо заменили a на b, либо ничего не делали (если $a == b$). Действительно, если бы мы попытались добавлять элементы справа от a, чтобы получить последним b, то нам пришлось бы когда-нибудь добавить b, что противоречит условию (либо заменить новый добавленный элемент на b, что неоптимально). Получается, символы справа от a мы не добавляли, само a мы не стирали, удивительно, но единственный выход - заменить элементы.

Алгоритм Вагнера-Фишера - заполняем таблицу
Собственно, чтобы решить задачу, достаточно заполнить таблицу по верхней формуле
Для восстановления ответа заполняем матрицу, после чего идём из правого нижнего угла в левый верхний, на каждом шаге вычисляем оптимальный путь (по формуле выше)

6.10 Задача Коммивояжёра. Метод ветвей и границ

На самом деле, на лекции дали обычный жадный алгоритм:

- Строим матрицу с исходными данными
- Находим минимум по строкам
- Редукция строк
- Находим минимум по столбцам
- Редукция столбцов
- Вычисляем оценку (сумму минимумов в строке и столбце)
- Редукция матрицы