Алгоритмы и структуры данных

Это ужасно! Я попытался собрать всю нужную информацию и ссылки в один файлик. Надеюсь, это будет хоть сколько-нибудь полезно. За точность не отвечаю, хотя я и пытался проверять. Были использованы следующие материалы и инструменты: чей-то украденный конспект (спасибо огромное этому человеку), информация из книги Кормена, старый файлик с ответами, Gemini-1.5-Pro.

Плейлист на YouTube с записями (почти) всех лекций 2024 года: https://www.youtube.com/playlist?list=PL8bFx8PltE4MSAk_SaW_MSnpVO91Cw-a1. Ссылки на видео с лекций ниже приведены с учетом таймкодов (но я это не доделал).

Книга Кормена с поиском: https://disk.yandex.ru/d/6PhppDCEL-5M0g

Задания с ответами в самом конце!!!

Перед использованием просьба провести фактчекинг :)

Программа курса

1. ***Основы анализа алгоритмов.*** Асимптотический анализ верхней и средней оценок сложности алгоритмов; сравнение наилучших, средних и наихудших оценок; O-, o-, ω- и θ-нотации; эмпирические измерения эффективности алгоритмов; накладные расходы алгоритмов по времени и памяти.
2. ***Рекуррентные соотношения и анализ рекурсивных алгоритмов.*** Определение, методы подстановки, итераций, деревья рекурсии. Основная теорема о рекуррентных соотношениях.
3. ***Динамические структуры данных.*** Линейные списки. Деревья: двоичные, деревья поиска, ветвящиеся деревья. Способы представления. Сбалансированные деревья: AVL-деревья, B-деревья, Красно-черные деревья, идеально сбалансированное дерево. Сравнение стоимости операций – асимптотические и эмпирические оценки.
4. ***Амортизационный анализ.*** (Помнить, зачем это надо). Учётные стоимости операций, использование при оценке времени работы алгоритмов. Методы группировки, предоплаты, потенциалов. Требования к потенциальной функции. Системы непересекающихся множеств. Основные операции, оценка амортизированной стоимости операций.
5. ***Графы.*** Представление графов – матрица смежности, списки смежности. Стратегии поиска в ширину, в глубину. Алгоритмы Дейкстры, Крускала, Прима. Топологическая сортировка. Алгоритм поиска сильно связных компонент.
6. ***Хэш-функции, хэш-таблицы.*** Открытая адресация, таблицы на основе цепочек. Базовые виды хеш-функций. Проблемы кластеризации, удаления элементов. Реализация структуры данных «словарь» на основе хэш-таблицы. Хопскотч, Робин-Гуд, двухвыборное (не то же, что и двойное!!), кукушкино хеширование.
7. ***Кучи.*** Двоичные кучи, сортировка с помощью двоичной кучи, оценка эффективности.
8. ***Стратегии переборных алгоритмов.*** Полный перебор, метод ветвей и границ на примере задачи коммивояжера.
9. ***Динамическое программирование.*** Признаки применимости, общая стратегия. Стратегии «сверху вниз» (используя механизм мемоизации) и «снизу вверх». Алгоритм перемножения матриц, поиск наибольшей общей подпоследовательности, задача о рюкзаке.

10. ***Жадные алгоритмы.*** Признаки применимости жадных алгоритмов, задача о рюкзаке (непрерывный варианты).
11. ***Алгоритмы поиска подстрок.*** «Наивный» алгоритм, алгоритмы РабинаКарпа, Кнута-Морриса-Пратта, Бойера-Мура (общая идея). Поиск подстрок с помощью конечных автоматов.

Примечания: в алгоритмах знать оценки, сильные слабые стороны, при каких сценариях выгоднее; + биноминальные деревья и кучи, фиббоначиевы (представление о них) + сортировка подсчётом.

1. Основы анализа алгоритмов

Анализ алгоритмов — это процесс определения количества ресурсов, необходимых для выполнения алгоритма. Обычно оценивается время выполнения, но также могут быть рассмотрены и другие ресурсы, такие как память, пропускная способность сети или необходимое аппаратное обеспечение.

Асимптотический анализ

Асимптотический анализ фокусируется на поведении алгоритма при увеличении размера входных данных до бесконечности. Он позволяет абстрагироваться от деталей реализации и аппаратного обеспечения, сосредоточившись на **порядке роста** времени выполнения.

Оценка сложности алгоритмов

- **Наилучший случай:** Минимальное время выполнения для входных данных определенного размера. Например, для сортировки вставкой наилучший случай - это уже отсортированный массив.
- **Наихудший случай:** Максимальное время выполнения для входных данных определенного размера. Для сортировки вставкой наихудший случай - это массив, отсортированный в обратном порядке.
- **Средний случай:** Среднее время выполнения по всем возможным входным данным определенного размера. Для анализа среднего случая необходимо знать распределение входных данных, что не всегда возможно.

Сравнение оценок сложности

- **Наилучший случай** часто неинформативен, так как редко встречается на практике.
- **Наихудший случай** гарантирует, что алгоритм не будет работать дольше определенного времени, но может быть пессимистичным.
- **Средний случай** более реалистичен, но его анализ сложнее и требует знания о распределении входных данных.

Асимптотические обозначения

В лекциях: https://youtu.be/Ju2tp7S-jfc?si=ZrUFi4eUqWcCT_Jd&t=1594

Нотация Бахмана–Ландау:

- **O-нотация (O-большое):** Дает **верхнюю границу** времени выполнения. Запись $f(n) = O(g(n))$ означает, что существует константа $c > 0$ и число n_0, такие, что $0 \leq f(n) \leq cg(n)$ для всех $n \geq n_0$.
  Неформально: $f \leq g$
- **Ω-нотация (Омега-большое):** Дает **нижнюю границу** времени выполнения. Запись $f(n) = \Omega(g(n))$ означает, что существует константа $c > 0$ и число n_0, такие, что $0 \leq cg(n) \leq f(n)$ для всех $n \geq n_0$.
  Неформально: $f = g$
- **Θ-нотация (Тета-большое):** Дает **асимптотически точную оценку** времени выполнения. Запись $f(n) = \Theta(g(n))$ означает, что $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.
  Неформально: $f \geq g$

- **o-нотация (o-малое):** Указывает, что функция $f(n)$ растет **строго медленнее**, чем $g(n)$. Запись $f(n) = o(g(n))$ означает, что для любой константы $c > 0$ существует число n_0, такое, что $0 \leq f(n) < cg(n)$ для всех $n \geq n_0$.
  Неформально: $f < g$
- **ω-нотация (омега-малое):** Указывает, что функция $f(n)$ растет **строго быстрее**, чем $g(n)$. Запись $f(n) = \omega(g(n))$ означает, что для любой константы $c > 0$ существует число n_0, такое, что $0 \leq cg(n) < f(n)$ для всех $n \geq n_0$.
  Неформально: $f > g$

Замечания!

- Любой логарифм растет медленнее степенной функции
- Любая степенная функция растет медленнее показательной
- Основание логарифмов не важно, т.к. скорость роста одинаковая с точностью до $const$

Эмпирические измерения

Асимптотический анализ дает теоретическую оценку времени выполнения, но на практике важно проводить эмпирические измерения. Для этого алгоритм реализуется в виде программы и запускается на реальном компьютере с различными наборами входных данных. Измеряется фактическое время выполнения, которое затем сопоставляется с теоретическими оценками.

Накладные расходы

При анализе алгоритма важно учитывать накладные расходы (overhead), связанные с его реализацией. Например, рекурсивный алгоритм может иметь большие накладные расходы, связанные с вызовами функций.

Накладные расходы по времени

Включают в себя время, затрачиваемое на выполнение вспомогательных операций, таких как:

- Вызовы функций
- Выделение и освобождение памяти
- Проверка условий
- Обновление счетчиков

Накладные расходы по памяти

Включают в себя память, используемую для хранения:

- Локальных переменных
- Параметров функций
- Вспомогательных структур данных

Пример: Сортировка вставкой

INSERTION-SORT(A)	*Стоимость*	*Повторы*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Вставка $A[j]$ в отсортированную последовательность $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ и $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Время работы алгоритма.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} t_j - 1 + c_7 \sum_{j=2}^{n} t_j - 1 + c_8(n-1)$$

В лучшем случае:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

В худшем случае:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) +$$
$$+ c_6\left(\frac{n(n+1)}{2}\right) + c_7\left(\frac{n(n+1)}{2}\right) + c_8(n-1) =$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

2. Рекуррентные соотношения и анализ рекурсивных алгоритмов

Рекуррентные соотношения (recurrence relations) — это уравнения, определяющие последовательность чисел, в которых каждый член последовательности выражается через предыдущие члены. Они часто используются для анализа времени работы рекурсивных алгоритмов, поскольку естественным образом описывают, как время работы задачи размером n зависит от времени работы подзадач меньшего размера.

Определение

Рекуррентное соотношение — это соотношение вида:

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n), & n > 1 \\ \Theta(1), & n = 1 \end{cases}, \text{ где } a \geq 1 \text{ и } b \geq 1$$

$f(n)$ — функция, асимптотически положительная.

Например, рекуррентное соотношение для времени работы алгоритма сортировки слиянием имеет вид:

- $T(1) = \Theta(1)$ (базовый случай)
- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ (рекуррентное уравнение)

Методы решения рекуррентных соотношений

Существует несколько методов решения рекуррентных соотношений, т.е. получения асимптотических границ для $T(n)$.

Метод подстановки

В методе подстановки (substitution method) мы делаем предположение о виде решения, а затем используем математическую индукцию для доказательства его корректности.

4.3. Метод подстановки решения рекуррентных соотношений

Теперь, когда вы познакомились с описанием времени работы алгоритмов "разделяй и властвуй" рекуррентными соотношениями, рассмотрим способы решения таких рекуррентных соотношений. В этом разделе начнем рассмотрение с метода подстановки.

Метод подстановки для решения рекуррентных соотношений состоит из двух шагов:

1. делается предположение о виде решения;
2. с помощью метода математической индукции определяются константы и доказывается, что решение правильное.

Название "метод подстановки" связано с тем, что мы подставляем предполагаемое решение вместо функции при применении гипотезы индукции для меньших значений. Это мощный метод, но для его применения нужно суметь сделать предположение о виде решения.

Метод подстановки можно применять для определения либо верхней, либо нижней границы рекуррентного соотношения. В качестве примера определим верхнюю границу рекуррентного соотношения

$$T(n) = 2T(\lfloor n/2 \rfloor) + n\,, \tag{4.19}$$

подобного соотношениям (4.3) и (4.4). Мы предполагаем, что решение имеет вид $T(n) = O(n \lg n)$. Наш метод заключается в доказательстве того, что при подходящем выборе константы $c > 0$ выполняется неравенство $T(n) \le cn \lg n$. Начнем с того, что предположим справедливость этого неравенства для всех положительных $m < n$, в частности для $m = \lfloor n/2 \rfloor$, что дает $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.

Подстановка в рекуррентное соотношение приводит к

$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n\,, \end{aligned}$$

где последний шаг выполняется при $c \ge 1$.

Теперь, согласно методу математической индукции, необходимо доказать, что наше решение справедливо для граничных условий. Обычно для этого достаточно показать, что граничные условия являются подходящей базой для доказательства по индукции. В рекуррентном соотношении (4.19) необходимо доказать, что константу c можно выбрать достаточно большой для того, чтобы соотношение $T(n) \le cn \lg n$ было справедливо и для граничных условий. Такое требование иногда приводит к проблемам. Предположим, например, что $T(1) = 1$ — единственное граничное условие рассматриваемого рекуррентного соотношения. Далее, для $n = 1$ соотношение $T(n) \le cn \lg n$ дает нам $T(1) \le c \cdot 1 \cdot \lg 1 = 0$, что противоречит условию $T(1) = 1$. Следовательно, данный базис индукции нашего доказательства не выполняется.

Эту сложность, возникающую при доказательстве предположения индукции для указанного граничного условия, легко обойти. Например, в рекуррентном соотношении (4.19) можно воспользоваться преимуществами асимптотических обозначений, требующих доказать неравенство $T(n) \le cn \lg n$ для $n \ge n_0$, где n_0 — ***выбранная нами*** константа. Идея по устранению возникшей проблемы заключается в том, чтобы в доказательстве по методу математической индукции не учитывать граничное условие $T(1) = 1$. Обратите внимание, что при $n > 3$ рассматриваемое рекуррентное соотношение явным образом от $T(1)$ не зависит. Таким образом, выбрав $n_0 = 2$, в качестве базы индукции можно рассматривать не $T(1)$, а $T(2)$ и $T(3)$. Заметим, что здесь делается различие между базой рекуррентного соотношения ($n = 1$) и базой индукции ($n = 2$ и $n = 3$). Из рекуррентного соотношения следует, что $T(2) = 4$, а $T(3) = 5$. Теперь доказательство по методу математической индукции соотношения $T(n) \le cn \lg n$ для некоторой константы $c \ge 1$ можно завершить, выбрав ее достаточно большой для того, чтобы были справедливы неравенства $T(2) \le c2 \lg 2$ и $T(3) \le c3 \lg 3$. Оказывается, что для этого достаточно выбрать $c \ge 2$. В большинстве рекуррентных соотношений, которые нам предстоит рассмотреть, легко расширить граничные условия таким образом, чтобы гипотеза индукции оказалась верна для малых n.

Метод итераций

В методе итераций (iteration method) мы многократно подставляем рекуррентное уравнение в само себя, раскрывая рекурсию до тех пор, пока не получим выражение, не содержащее рекурсивных вызовов.

1. **Раскрытие рекурсии:** Подставляем рекуррентное уравнение в само себя, заменяя $T(n_i)$ на $f(T(n_{i1}), T(n_{i2}), \ldots, T(n_{ik}))$.
2. **Упрощение:** Упрощаем полученное выражение, объединяя подобные члены.
3. **Поиск закономерности:** Ищем закономерность в полученных выражениях и обобщаем её для произвольного уровня рекурсии.
4. **Вычисление суммы:** Вычисляем сумму, полученную в результате раскрытия рекурсии.

тут скорее всего должно быть что-то еще

Деревья рекурсии

Дерево рекурсии (recursion tree) — это графическое представление рекурсивных вызовов алгоритма. Каждый узел дерева представляет один рекурсивный вызов, а ребра соединяют вызовы, связанные отношением "родитель-потомок". Стоимость каждого узла — это время работы соответствующего рекурсивного вызова, не включая время, затрачиваемое на рекурсивные вызовы, которые он выполняет.

1. **Построение дерева:** Строим дерево рекурсии, начиная с корня, представляющего исходный вызов алгоритма.
2. **Оценка стоимости уровней:** Оцениваем стоимость каждого уровня дерева, суммируя стоимости всех узлов на этом уровне.
3. **Вычисление общей стоимости:** Суммируем стоимости всех уровней дерева, чтобы получить общую стоимость вычисления.

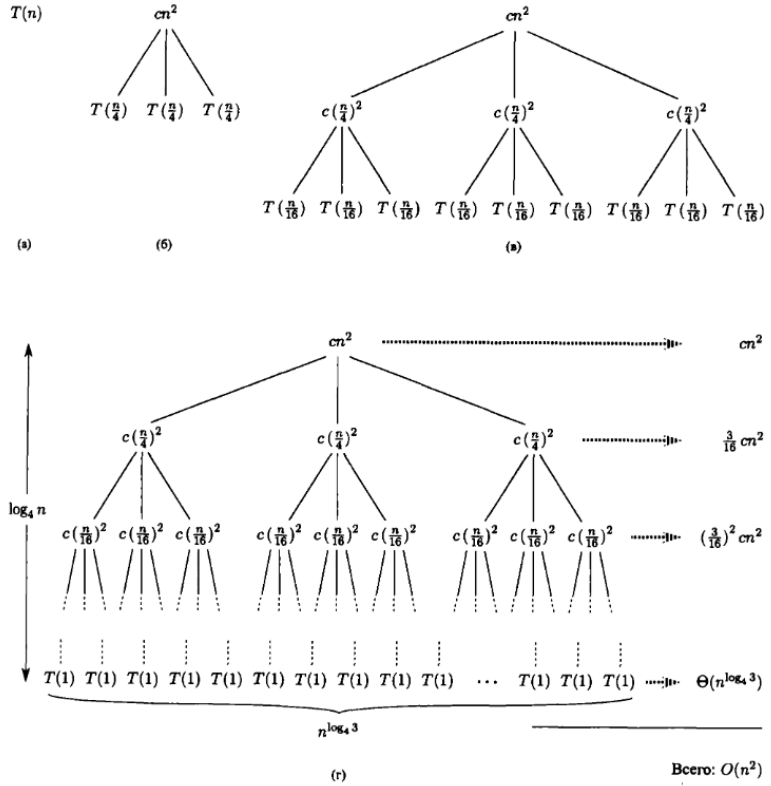


Рис. 4.5. Построение дерева рекурсии для $T(n) = 3T(n/4) + cn^2$. В части **(а)** показано значение $T(n)$, которое постепенно раскрывается в частях **(б)–(г)**, образуя дерево рекурсии. Полностью раскрытое дерево в части **(г)** имеет высоту $\log_4 n$ (в нем $\log_4 n + 1$ уровней).

Теперь просуммируем стоимости всех уровней, чтобы найти стоимость всего дерева:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(согласно (A.5))} . \end{aligned}$$

Эта формула выглядит несколько запутанной, но только до тех пор, пока мы не догадаемся воспользоваться той небольшой свободой, которая допускается при асимптотических оценках, и не используем в качестве верхней границы одного из слагаемых бесконечно убывающую геометрическую прогрессию. Возвращаясь на один шаг назад и применяя формулу (A.6), получаем

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) . \end{aligned}$$

Таким образом, для исходного рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ получен предполагаемый вид решения $T(n) = O(n^2)$. В рассматриваемом примере коэффициенты при cn^2 образуют убывающую геометрическую прогрессию, а их сумма, согласно уравнению (A.6), ограничена сверху константой $16/13$. Поскольку вклад корня дерева в полное время работы равен cn^2, время работы корня представляет собой некоторую постоянную часть от общего времени работы всего дерева в целом. Другими словами, полное время работы всего дерева в основном определяется временем работы его корня.

В действительности, если $O(n^2)$ в самом деле представляет собой верхнюю границу для рекуррентного соотношения (в чем мы вскоре убедимся), эта граница должна быть асимптотически точной оценкой. Почему? Потому что первый рекурсивный вызов дает вклад в общее время работы алгоритма, который выражается как $\Theta(n^2)$, поэтому нижняя граница решения рекуррентного соотношения представляет собой $\Omega(n^2)$.

Теперь, чтобы убедиться в том, что наше предположение верно, т.е. что $T(n) = O(n^2)$ является верхней границей для рекуррентного соотношения $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$, можно воспользоваться методом подстановок. Мы хотим показать, что $T(n) \le dn^2$ для некоторой константы $d > 0$. Используя ту же константу $c > 0$, что и ранее, мы получаем

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

где последний шаг выполняется при $d \ge (16/13)c$.

Основная теорема о рекуррентных соотношениях

Основная теорема (master theorem) предоставляет "кулинарную книгу" для решения рекуррентных соотношений вида:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

где $a \ge 1$ и $b > 1$ — константы, а $f(n)$ — асимптотическая функция.

Основная теорема гласит следующее:

1. Если $f(n) = O(n^{\log_b a - \epsilon})$ для некоторого $\epsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$
2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \lg n)$
3. Если $f(n) = \Omega(n^{\log_b a + \epsilon})$ для некоторого $\epsilon > 0$, и если $af\left(\frac{n}{b}\right) \leq cf(n)$ для некоторого $c < 1$ и достаточно больших n, то $T(n) = \Theta(f(n))$

Замечание!

$n^2 < n^2 \log n$, но $n^2 \log n < \begin{cases} n^{2+1} \\ n^{2+0.00001} \\ n^{2+\epsilon} \end{cases}$

Примеры решения рекуррентных соотношений с использованием основной теоремы

Пример 1

Рассмотрим рекуррентное соотношение для сортировки слиянием:

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

Здесь $a = 2$, $b = 2$, и $f(n) = n$.

- Вычислим $\log_b a = \log_2 2 = 1$.
- Поскольку $f(n) = n$, то $c = 1$.

Следовательно, $c = \log_b a$, и по второй части основной теоремы имеем:

$T(n) = \Theta(n \lg n)$

Пример 2

Рассмотрим рекуррентное соотношение:

$T(n) = 3T\left(\frac{n}{4}\right) + n^2$

Здесь $a = 3$, $b = 4$, и $f(n) = n^2$.

- Вычислим $\log_b a = \log_4 3 \approx 0.792$.
- Поскольку $f(n) = n^2$, то $c = 2$.

Поскольку $c > \log_b a$, по третьей части основной теоремы имеем:

$T(n) = \Theta(n^2)$

3. Динамические структуры данных

Динамические структуры данных — это структуры данных, которые могут изменяться во время выполнения программы: элементы могут добавляться, удаляться или изменяться. Это отличает их от статических структур данных, таких как массивы, размер которых фиксирован.

Линейные списки

Линейные списки — это структуры данных, в которых элементы упорядочены линейно. Каждый элемент содержит данные и ссылку на следующий элемент списка.

Способы представления

- **Односвязный список:** Каждый элемент содержит ссылку только на следующий элемент.
- **Двусвязный список:** Каждый элемент содержит ссылки на предыдущий и следующий элементы.
- **Циклический список:** Последний элемент списка ссылается на первый.

Стоимость операций

Операция	Односвязный список	Двусвязный список
Вставка в начало	O(1)	O(1)
Вставка в конец	O(n)	O(1)
Удаление	O(n)	O(1)
Поиск	O(n)	O(n)

Алгоритм поиска

```
x = L.head
while x != NIL and x.key != k
    x = x.next
return x
```

Вставка в двусвязный список (в начало)

```
x.next = L.head
if L.head != NIL:
    L.head.prev = x
L.head = x
x.prev = NIL
```

Вставка в односвязный список (в начало)

```
x.next = L.head
L.head = x
```

Вставка в конец односвязного списка

```
function appendToEnd(L, x):
  if L.head == null:
    L.head = x
    x.next = null
  else:
    current = L.head
    while current.next != null:
      current = current.next
    current.next = x
    x.next = null
```

Вставка в конец двусвязного списка

```
function appendToEnd(L, x):
  if L.head == null:
    L.head = x
    x.next = null
    x.prev = null
  else:
    current = L.head
    while current.next != null:
      current = current.next
    current.next = x
    x.prev = current
    x.next = null
```

Удаление из односвязного списка

```
function deleteNode(L, value):
  if L.head == null:
    return // список пуст
  if L.head.value == value:
    L.head = L.head.next
    return
  current = L.head
  while current.next != null and current.next.value != value:
    current = current.next
  if current.next != null:
    current.next = current.next.next
```

Удаление из двусвязного списка

```
function deleteNode(L, value):
  if L.head == null:
    return // список пуст
  if L.head.value == value:
    L.head = L.head.next
    if L.head != null:
      L.head.prev = null
    return
  current = L.head
  while current != null and current.value != value:
    current = current.next
  if current != null:
    if current.next != null:
      current.next.prev = current.prev
    if current.prev != null:
```

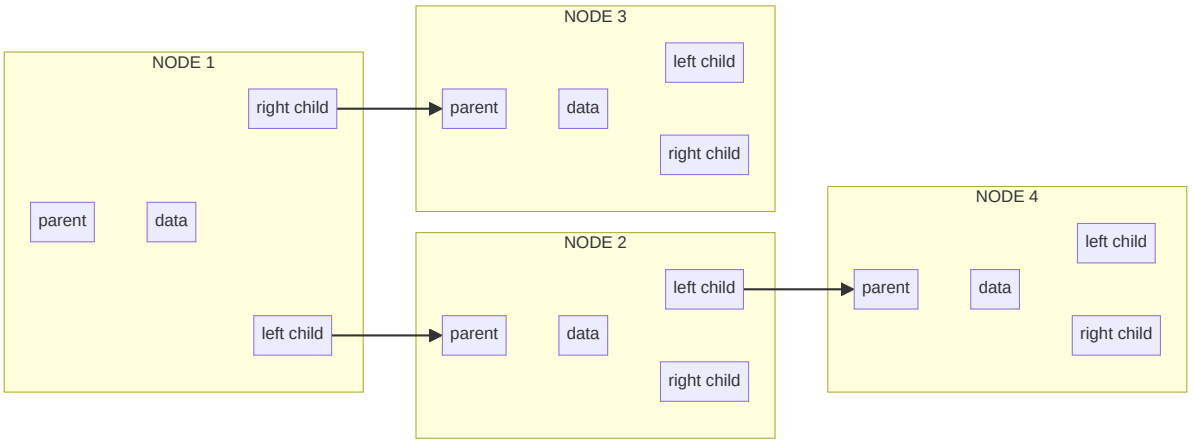
```
16          current.prev.next = current.next
```

Деревья

Деревья — это нелинейные структуры данных, в которых элементы организованы иерархически. Каждый элемент называется **узлом**, а связи между узлами — **ребрами**.

Способы хранения деревьев (!дубль)

1. В `deque`: каждый элемент знает только своего родителя.
2. В обычном массиве. Родитель ↔ ребёнок, восстанавливаются по индексам. НО! Только когда дерево заполнено полностью, за исключением, быть может, нижнего уровня.
   Формула для родителя: $\frac{i-1}{2}$
   Левый дочерний: $2i + 1$
   Правый дочерний: $2i + 2$
3. С помощью узлов. Стоимость доступа $O(1)$.

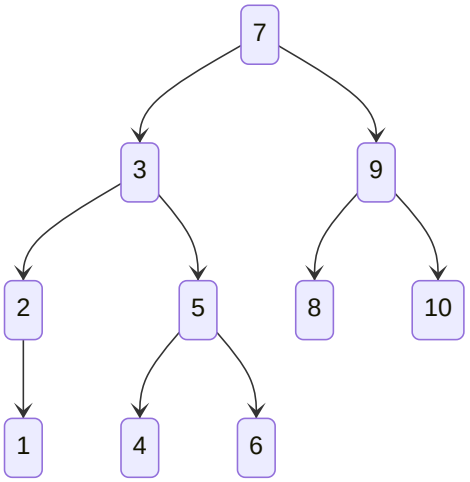


Двоичные деревья

Двоичное дерево — это дерево, в котором каждый узел имеет не более двух дочерних узлов.

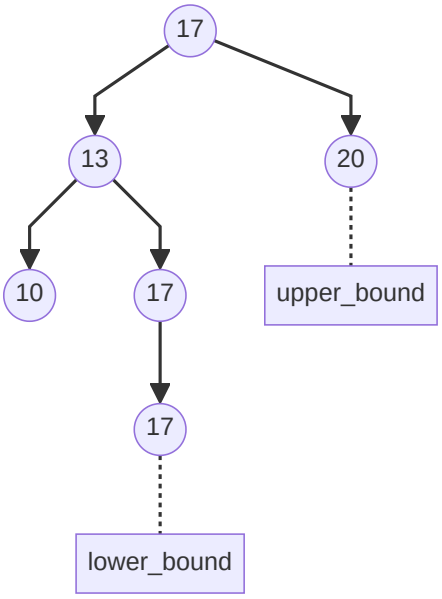
Деревья поиска

Дерево поиска — это двоичное дерево, в котором для каждого узла *x* выполняется следующее свойство: все ключи в левом поддереве *x* меньше *x.key*, а все ключи в правом поддереве *x* больше *x.key*.



Замечания!

- В std используется операция «меньше», элементы должны определять ее.
- Если дерево может содержать дубликаты, обычно не используется поиск по значению. Можно использовать инфиксный обход:



Элементы по порядку записи в массиве:

```
[--] [--] [17] [17] [17] [17] [20]
          ^ lower             ^ upper
```

`lower_bound` — первый элемент, больший либо равный x

`upper_bound` — первый элемент, больший x

Тогда в диапазоне от lower_bound до upper_bound будут располагаться все дубликаты.

Операции БДП

- Вставка
- Извлечение (поиск)
- Следующий

- Предыдущий
- Минимальный
- Максимальный

Полный и слабый порядок

> **Note**
>
> Чтобы элементы могли находиться в `set` (и прочем подобном), необходимо, чтобы на этих элементах была определена операция «меньше». `set` может также работать с компаратором «больше», но не «меньше либо равно», «больше либо равно».

Введем на произвольном множестве бинарное отношение:

- Если это отношение позволяет различать все элементы множества, то это **полный порядок**.
- **Weak ordering** (слабый порядок) – бинарное отношение на элементах множества, при котором некоторые элементы множества не различаются между собой. Пример: «количество цифр в x, больше, чем в y».

> **Note**
>
> Weak ordering не используется в структурах по типу set — необходимо более строгое ограничение. `multiset` позволяет поддерживать неполный порядок.

Strict weak ordering (отношение строго частичного порядка) — бинарное отношение на элементах множества со следующими свойствами:

1. Irreflexivity: $\forall x \in S \;\; cmp(x, x) = false$
2. Assymetry: $\forall x, y \in S \;\; cmp(x, y) = true \;\Rightarrow\; cmp(y, x) = false$
3. Transitivity: $\forall x, y, z \in S \;\; cmp(x, y) \wedge cmp(y, z) = true \;\Rightarrow\; cmp(x, z) = true$
4. Несравнимость: $\forall x, y, z \in S \;\; \begin{matrix} cmp(x, y) \vee cmp(y, z) = true; \\ cmp(y, z) \vee cmp(z, y) = false; \\ cmp(x, z) \vee cmp(z, x) = false \end{matrix}$

Замечание. $true$ и $false$ здесь заменяют «элементы связаны / не связаны отношением».

Ветвящиеся деревья

Ветвящееся дерево — это дерево, в котором каждый узел может иметь произвольное количество дочерних узлов.

Способы представления

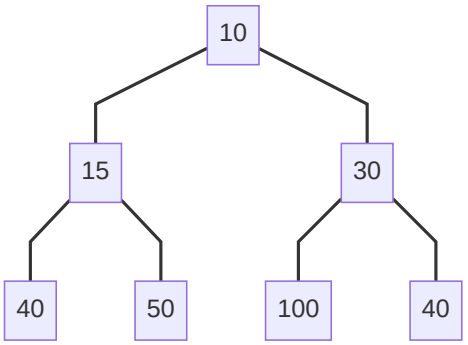
1. Массив

Этот способ часто используется для представления полных бинарных деревьев, таких как пирамиды (heaps). В таком представлении:

- Корень дерева находится в позиции 0.
- Для узла на позиции i:
  - Левый потомок находится на позиции $2i + 1$.
  - Правый потомок находится на позиции $2i + 2$.
  - Родительский узел находится на позиции $\left\lfloor \frac{i-1}{2} \right\rfloor$ (если $i > 0$).

Пример

Рассмотрим полное бинарное дерево:



Его представление в массиве будет:

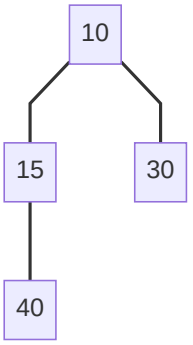
```
[10, 15, 30, 40, 50, 100, 40]
```

2. Указатели

Этот способ используется для представления произвольных деревьев, в которых каждый узел содержит указатели на своих дочерних узлов. В бинарном дереве каждый узел имеет два указателя: на левого и правого потомка.

Пример

Рассмотрим бинарное дерево:



Каждый узел будет представляться структурой:

```
Node {
  value: int
  left: Node
  right: Node
}
```

И представление дерева будет выглядеть так:

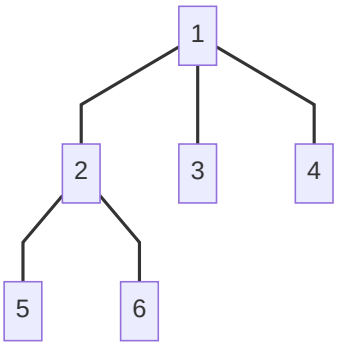
```
root = Node(10)
root.left = Node(15)
root.right = Node(30)
root.left.left = Node(40)
```

3. Левый потомок, правый брат

Этот способ используется для представления ветвящихся деревьев, где узлы могут иметь произвольное количество потомков. Каждый узел имеет два указателя: на своего первого (левого) потомка и на следующего (правого) брата.

Пример

Рассмотрим дерево:



Каждый узел будет представляться структурой:

```
Node {
  value: int
  leftChild: Node
  rightSibling: Node
}
```

И представление дерева будет выглядеть так:

```
root = Node(1)
root.leftChild = Node(2)
root.leftChild.rightSibling = Node(3)
root.leftChild.rightSibling.rightSibling = Node(4)
root.leftChild.leftChild = Node(5)
root.leftChild.leftChild.rightSibling = Node(6)
```

В этом представлении:

- Узел 1 имеет левого потомка 2 и правого брата 3.
- Узел 2 имеет левого потомка 5 и правого брата 6.
- Узел 3 является правым братом узла 2.
- Узел 4 является правым братом узла 3.

Таким образом, указанный способ позволяет эффективно представлять деревья с произвольным числом потомков для каждого узла.

Сбалансированные деревья

[Более строгое определение] Сбалансированное дерево — это дерево поиска, в котором для любого узла количество элементов в левом и правом поддереве различается не более, чем на 1.

[Менее строгое определение] Дерево называется сбалансированным, если для любого узла x верно, что $|height(x.left) - height(x.right)| \leq 1$

При вставке или удалении требуется перестроение дерева. Таким образом, $O(1)$. Чтобы сохранить $O(\log n)$, ослабим требование сбалансированности дерева.

Красно-черные деревья

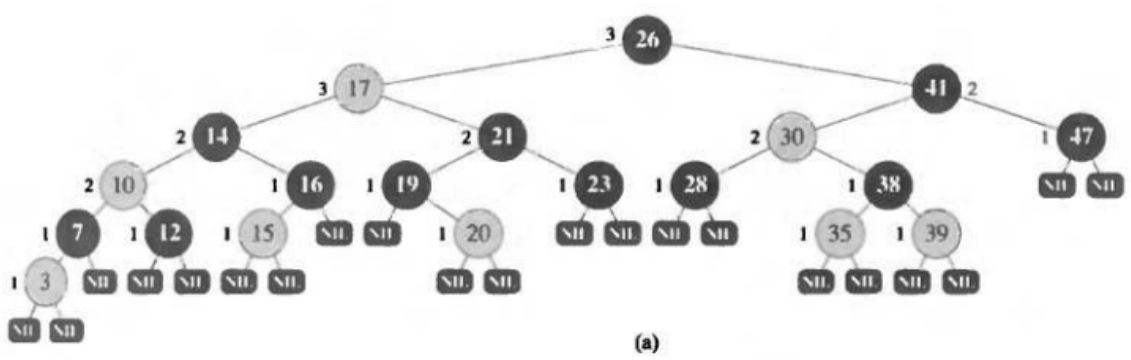
RB-tree — частный случай B-tree.

Красно-черное дерево представляет собой бинарное дерево поиска с одним дополнительным битом цвета в каждом узле. Цвет узла может быть либо красным (RED), либо черным ($BLACK$). В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-черном дереве не отличается от другого по длине более чем в два раза, так что красно-черные деревья являются приближенно сбалансированными.

Каждый узел дерева содержит атрибуты $color$, key, $left$, $right$ и p. Если не существует дочернего или родительского узла по отношению к данному, соответствующий указатель принимает значение NIL. Мы будем рассматривать эти значения МП. как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все «нормальные» узлы, содержащие поле ключа, становятся внутренними узлами дерева.

Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим ***красно-черным свойствам***.

> **Important**
>
> 1. Каждый узел является либо красным, либо черным.
> 2. Корень дерева является черным узлом.
> 3. Каждый лист дерева (NIL) является черным узлом.
> 4. Если узел красный, то оба его дочерних узла черные.
> 5. Для каждого узла все простые пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.



Замечание. Можно обойтись без затрат на цвет, но обычно это не окупается. В худшем случае возможно не более, чем 2-кратное различие по высотам поддеревьев.

> **Important**
>
> **Лемма.** Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\lg(n+1)$.
>
> **Следствие.** Такие операции над динамическими множествами, как $Search$, $Minimum$, $Maximum$, $Predecessor$ и $Successor$, при использовании красно-черных деревьев выполняются за время $O(\lg h)$, поскольку, как показано в главе 12, время работы этих операций на дереве поиска высотой h составляет $O(h)$, а любое красно-черное дерево с n узлами является деревом поиска высотой $O(\lg h)$.

Повороты

О вставке ниже.

Операции над деревом поиска TREE-INSERT и TREE-DELETE, будучи применены к красно-черному дереву с n ключами, выполняются за время $O(\lg n)$. Поскольку они изменяют дерево, в результате их работы могут нарушаться красно-черные свойства, перечисленные в разделе 13.1. Для восстановления этих свойств необходимо изменить цвета некоторых узлов дерева, а также структуру его указателей.

Изменения в структуре указателей будут выполняться с помощью ***поворотов*** (rotations), которые представляют собой локальные операции в дереве поиска, сохраняющие свойство бинарного дерева поиска. На рис. 13.2 показаны два типа поворотов — левый и правый. При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом $T.nil$; x может быть любым узлом дерева, правый дочерний узел которого — не $T.nil$. Левый поворот выполняется "вокруг" связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y — правым потомком x.

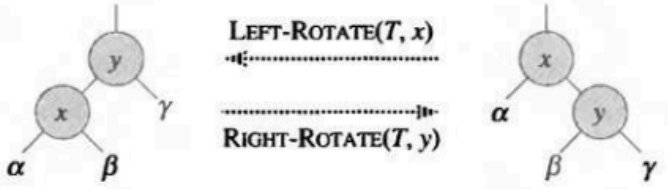


Рис. 13.2. Операция поворота в бинарном дереве поиска. Операция LEFT-ROTATE(T, x) преобразует конфигурацию из двух узлов справа в конфигурацию, показанную слева, путем изменения константного количества указателей. Обратная операция RIGHT-ROTATE(T, y) преобразует конфигурацию, показанную слева, в конфигурацию в правой части рисунка. Буквы α, β и γ представляют произвольные поддеревья. Операция поворота сохраняет свойство бинарного дерева поиска: ключи в α предшествуют ключу $x.key$, который предшествует ключам в β, которые предшествуют ключу $y.key$, который предшествует ключам в γ.

В псевдокоде процедуры LEFT-ROTATE предполагается, что $x.right \neq T.nil$ и что родитель корневого узла — $T.nil$.

```
LEFT-ROTATE(T, x)
 1  y = x.right                  // Установка y
 2  x.right = y.left             // Превращение левого поддерева y
                                 // в правое поддерево x
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                    // Передача родителя x узлу y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x                   // Размещение x в качестве левого
                                 // дочернего узла y
12  x.p = y
```

Правое вращение.

```
Right-Rotate(T, x)
    y = x.left
    x.left = y.right
    if y.right != T.nil
        y.right.p = x
    y.p = x.p
    if x.p == T.nil
        T.root = y
    elseif x == x.p.left
        x.p.left = y
    else
        x.p.right = y
    y. right = x
    x.p = y
```

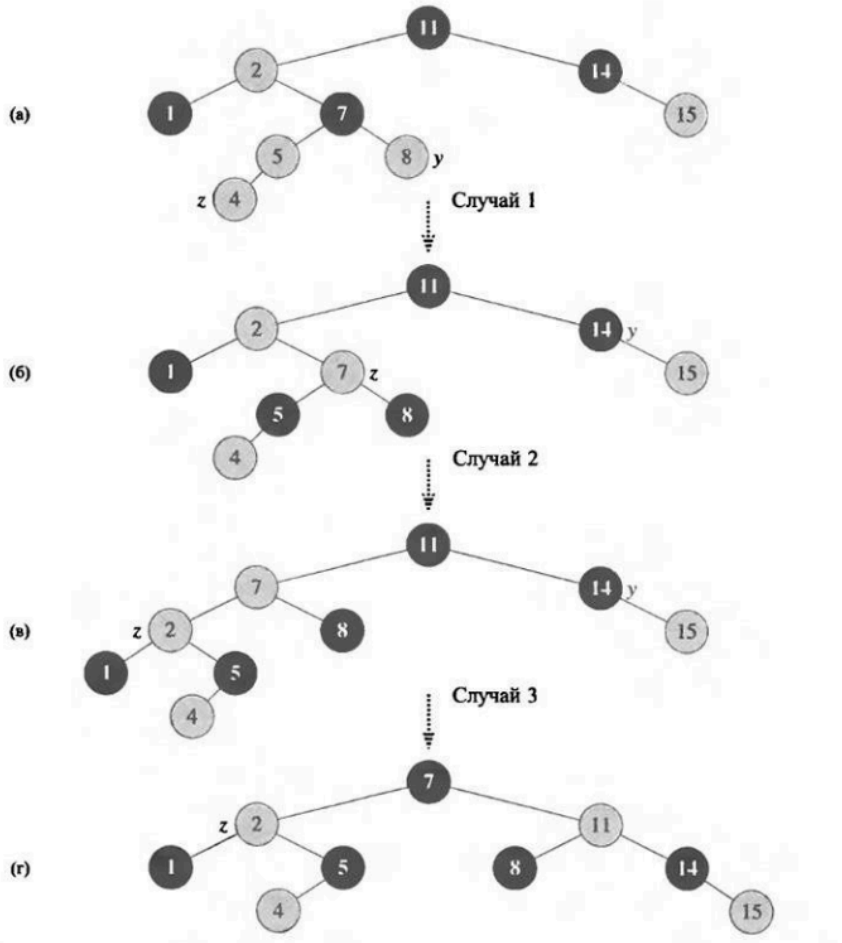
Вставка

Вставляем всегда красный элемент (для сохранения баланса). Но можем получить тогда «локальную проблему» – 2 подряд идущих красных узла. В таком случае, нужно восстановить красно-черные свойства.

В RB-tree вводится вспомогательная операция **вращения** (выше).

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // Случай 1
 6              y.color = BLACK                      // Случай 1
 7              z.p.p.color = RED                    // Случай 1
 8              z = z.p.p                            // Случай 1
 9          else if z == z.p.right
10                  z = z.p                          // Случай 2
11              LEFT-ROTATE(T, z)                    // Случай 2
12              z.p.color = BLACK                    // Случай 3
13              z.p.p.color = RED                    // Случай 3
14              RIGHT-ROTATE(T, z.p.p)               // Случай 3
15      else (то же, что и в части then, но с заменой
              "правого" (right) "левым" (left) и наоборот)
16  T.root.color = BLACK
```



Удаление

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root и x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // Случай 1
 6              x.p.color = RED                          // Случай 1
 7              LEFT-ROTATE(T, x.p)                      // Случай 1
 8              w = x.p.right                            // Случай 1
 9          if w.left.color == BLACK и w.right.color == BLACK
10              w.color = RED                            // Случай 2
11              x = x.p                                  // Случай 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK n               // Случай 3
14                  nw.color = RED                       // Случай 3
15                  RIGHT-ROTATE(T, w)                   // Случай 3
16                  w = x.p.right                        // Случай 3
17              w.color = x.p.color                      // Случай 4
18              x.p.color = BLACK                        // Случай 4
19              w.right.color = BLACK                    // Случай 4
20              LEFT-ROTATE(T, x.p)                      // Случай 4
21              x = T.root                               // Случай 4
22      else (то же, что и в части then, но с заменой
              "правого" (right) "левым" (left) и наоборот)
23  x.color = BLACK
```

Оценка стоимости операций

Черная высота узла (black height) — количество черных вершин в пути от узла к листу, не считая сам лист (цвет вершины, от которой идем, нас не интересует).

Лемма 1. В пути от вершины с черной высотой $bh(node)$ к листу не более $bh(node)$ красных вершин $\Rightarrow$ высота вершины $\leq 2bh(node)$.

Лемма 2. КЧД с n внутренними узлами (n узлов без листьев) имеет высоту $\leq 2\log(n+1)$.

Получается, что высота КЧД превосходит высоту идеально сбалансированного дерева не более, чем в 2 раза. Причем при больших n значение высоты КЧД будет стремиться в $\log n$.

Таким образом, получили, что у всех операций АС $O(\log n)$.

AVL-деревья

AVL-деревья, названные в честь своих создателей, Адельсона-Вельского и Ландиса, представляют собой самобалансирующиеся бинарные деревья поиска. Они гарантируют логарифмическую сложность основных операций (вставка, удаление, поиск) даже в худшем случае, предотвращая вырождение дерева в линейный список.

Основные свойства

1. **Свойство бинарного дерева поиска:** Для каждого узла все ключи в левом поддереве меньше ключа узла, а все ключи в правом поддереве больше.
2. **Свойство балансировки:** Для каждого узла разница высот его левого и правого поддеревьев (называемая **баланс-фактором**) не превышает 1 (может быть -1, 0 или 1).

Высота дерева $\leq 1.4405\log(n+2) - 0.3277$. Доказывается через Фиббоначиевы деревья. По сравнению с КЧД превосходит ИСД на 45%. Для больших n стремится к $\log n$.

Балансировка

При вставке или удалении узла баланс дерева может нарушиться. Для восстановления баланса AVL-деревья используют **повороты**:

- **Левый поворот:** применяется, когда баланс-фактор узла становится равным -2, а баланс-фактор его правого потомка -1 или 0.
- **Правый поворот:** применяется, когда баланс-фактор узла становится равным 2, а баланс-фактор его левого потомка 1 или 0.
- **Двойной поворот (лево-правый или право-левый):** применяется в остальных случаях нарушения баланса.

Основные операции

Для реализации каждая вершина имеет дополнительное поле — высота данного узла.

1. **Вставка** $O(\log n)$
   - Вставка нового узла происходит как в обычном бинарном дереве поиска.
   - После вставки проверяется баланс-фактор каждого узла на пути от нового узла до корня.
   - При нарушении баланса выполняется соответствующий поворот.
   - Требует 2 вращения.
2. **Удаление**
   - Удаление узла происходит как в обычном бинарном дереве поиска.
   - После удаления проверяется баланс-фактор каждого узла на пути от родителя удаленного узла до корня.
   - При нарушении баланса выполняется соответствующий поворот.
   - Может потребовать вращений в количестве высоты дерева (медленнее, чем КЧД).
3. **Поиск** $O(\log n)$
   - Поиск ключа происходит как в обычном бинарном дереве поиска.
   - Быстрее поиска в КЧД.

При построении случайных AVL-деревьев, наиболее велика вероятность получить идеальное.

Сравнение с другими деревьями

Операция	AVL-дерево	Бинарное дерево поиска	Красно-черное дерево	B-дерево
Вставка	O(log n)	O(n)	O(log n)	O(log n)
Удаление	O(log n)	O(n)	O(log n)	O(log n)
Поиск	O(log n)	O(n)	O(log n)	O(log n)

Преимущества AVL-деревьев

- Гарантированная логарифмическая сложность операций.
- Простота реализации поворотов.

Недостатки AVL-деревьев

- Большое количество поворотов при частых вставках и удалениях.
- Небольшая разница в производительности по сравнению с красно-черными деревьями, которые требуют меньше поворотов.

Замечания по реализации в C++

- set, multiset, map, multimap – обычно КЧД
- Технически, неважно, какой элемент, главное – наличие сравнения
- Итератор – bidirectional и совпадает с константным
- Операции не перемещают элементы в памяти

Allocator – некий объект, сущность которого используется для выделения и освобождения памяти. Allocator = распределитель памяти.

Сложность при создании map / set из диапазона:

- $O(N)$, если последовательность отсортирована
- иначе $O(N \log N)$

B-деревья

B-деревья, в отличие от AVL-деревьев и красно-черных деревьев, оптимизированы для хранения данных на внешних носителях, таких как жесткие диски. Они отличаются от бинарных деревьев поиска тем, что каждый узел может содержать **несколько ключей и указателей на дочерние узлы**.

Основные свойства

1. **Упорядоченность ключей:** Ключи в каждом узле хранятся в отсортированном (неубывающем) порядке.
2. **Количество ключей:** Каждый узел, кроме корня, содержит от t до $2t - 1$ ключей, где t - **минимальная степень** B-дерева. Корень может содержать от 1 до $2t - 1$ ключей.
3. **Количество потомков:** Каждый узел имеет на один потомок больше, чем ключей.
4. **Сбалансированность:** Все листья находятся на одной глубине.

Высота. Для любого дерева с n $(n > 0)$ ключами, высотой h и минимальной степенью $t > 1$ верно: $h \leq log_t \frac{n+1}{2}$.

Асимптотика $O(\log N)$.

Операции

1. **Поиск:**
   - Поиск ключа начинается с корня.
   - В каждом узле выполняется поиск ключа среди ключей узла.
   - Если ключ найден, поиск завершается.
   - Если ключ не найден, поиск продолжается в соответствующем поддереве.
2. **Вставка:**
   - Новый ключ вставляется в соответствующий лист.

- Если лист переполнен (содержит $2t$ ключей), он **разбивается** на два узла, содержащих по t ключей.
- Средний ключ из переполненного листа перемещается в родительский узел.
- Если родительский узел переполнен, он также разбивается, и этот процесс продолжается вверх по дереву до корня.
- Если корень переполнен, создается новый корень, и высота дерева увеличивается на 1.

3. **Удаление:**
   - Удаление ключа происходит из соответствующего узла.
   - Если узел становится **недополненным** (содержит меньше t ключей), он **объединяется** с соседним узлом.
   - Если соседний узел также не дополнен, объединение происходит с родительским узлом, и этот процесс продолжается вниз по дереву до листьев.

Преимущества

- **Оптимизация для дискового хранения:** B-деревья минимизируют количество операций чтения с диска, поскольку каждый узел содержит много ключей.
- **Сбалансированность:** B-деревья гарантируют логарифмическую сложность операций.

Недостатки

- **Сложность реализации:** B-деревья сложнее в реализации, чем бинарные деревья поиска.

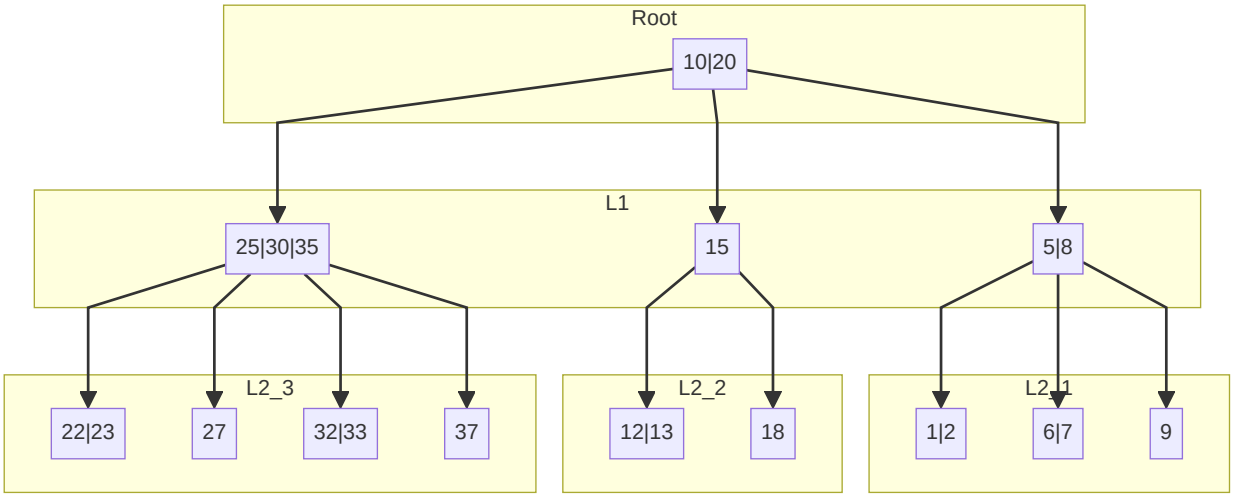
Сравнение с другими деревьями

B-деревья, как и AVL-деревья и красно-черные деревья, обеспечивают логарифмическую сложность операций. Однако B-деревья оптимизированы для дискового хранения, а AVL-деревья и красно-черные деревья - для оперативной памяти.

Применение

- **Базы данных:** B-деревья широко используются в системах управления базами данных для индексирования данных.
- **Файловые системы:** B-деревья используются в некоторых файловых системах для организации данных на диске.

Пример



Идеально сбалансированное дерево

Идеально сбалансированное дерево — это дерево, в котором все листья находятся на одной глубине.

Сравнение стоимости операций

Асимптотические оценки

Операция	Двоичное дерево поиска (наихудший случай)	Сбалансированное дерево поиска
Вставка	O(n)	O(log n)
Удаление	O(n)	O(log n)
Поиск	O(n)	O(log n)
Минимум	O(n)	O(log n)
Максимум	O(n)	O(log n)
Преемник	O(n)	O(log n)
Предшественник	O(n)	O(log n)

И еще неплохая табличка, украденная из чьего-то конспекта.

		Поиск	Insert, beg	I, mid	I, end	Доступ []	Издержки
Массив	vector, array, deque	$O(N)$	$O(N)$ $O(1)$	$O(N)$ $O(N)$	$O(N)$ $O(1)$	$O(1)$	0
	list, forward list	$O(N)$	$O(1)$	$O(N)$	$O(1)$ $O(N)$	$O(N)$	$2/1$
RBT	set, multiset, map, multimap	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	3
Hash-table	unordered_set, unordered_map	$\approx O(1)$	$\approx O(1)$	$\approx O(1)$	$\approx O(1)$	$\approx O(1)$	$0/2$

Замечание. Издержки – на хранение одного элемента.

Эмпирические оценки

На практике сбалансированные деревья поиска (например, красно-черные деревья) обычно работают быстрее, чем несбалансированные деревья поиска, даже для небольших наборов данных.

4. Амортизационный анализ

Амортизационный анализ — это метод анализа алгоритмов, который позволяет оценить среднее время выполнения операции в последовательности операций. Он особенно полезен, когда в последовательности встречаются как "дешевые", так и "дорогие" операции, и важно понять, как "дорогие" операции влияют на общую производительность.

Зачем нужен амортизационный анализ?

- **Более точная оценка:** Амортизационный анализ позволяет получить более точную оценку времени работы алгоритма, чем анализ наихудшего случая для каждой операции.
- **Учет редких операций:** Он позволяет учесть влияние редких, но "дорогих" операций, которые могут существенно исказить оценку времени работы в наихудшем случае.
- **Анализ структур данных:** Амортизационный анализ широко используется для анализа эффективности операций в динамических структурах данных, таких как динамические массивы, деревья поиска и системы непересекающихся множеств.

Учетные стоимости операций

В амортизационном анализе каждой операции присваивается **учетная стоимость** (amortized cost), которая может отличаться от её фактической стоимости. Учетные стоимости выбираются таким образом, чтобы **сумма учетных стоимостей** для любой последовательности операций была **не меньше суммы фактических стоимостей** этих операций.

Методы амортизационного анализа

1. Метод группировки (aggregate method)

В методе группировки мы определяем верхнюю границу $T(n)$ для общей стоимости n операций, а затем делим $T(n)$ на n, чтобы получить амортизированную стоимость каждой операции.

Пример. Рассмотрим динамический массив, который увеличивает свой размер в два раза, когда он заполнен. Операция вставки элемента в массив имеет две возможные стоимости:

- $O(1)$**:** Если в массиве есть свободное место.
- $O(n)$**:** Если массив заполнен, и его нужно увеличить.

Анализ:

- В худшем случае каждая вставка может потребовать $O(n)$ времени.
- Однако, если мы рассмотрим последовательность n вставок, то увидим, что увеличение размера массива происходит только $\log_2 n$ раз.
- Общая стоимость n вставок составляет $O(n + n/2 + n/4 + \ldots + 1) = O(n)$.
- Амортизированная стоимость каждой вставки, полученная методом группировки, равна $O(n)/n = O(1)$.

Еще пример (multistack). Операции `push`, `pop`, `multipop(k)`. Пусть сделано n операций, значит всего `multipop` может вытолкнуть n элементов. Тогда худшая оценка $O(n^2)$.

- $c_1 \cdot Pop + c_2 \cdot Push + c_3 \cdot MultiPop$
- `MultiPop` не может быть больше `Push`, тогда получаем: $c_1 \cdot Pop + 2c_2 \cdot Push \approx O(n)$
- Среднее: $\frac{O(n)}{n} \approx O(1)$

2. Метод предоплаты (accounting method)

В методе предоплаты мы "предоплачиваем" за "дорогие" операции, выполняя "дешевые" операции. "Предоплата" накапливается в виде "кредита", который затем используется для оплаты "дорогих" операций.

Основная идея. c_i – фактическая стоимость i-й операции. $\widehat{c_i}$ – учетная стоимость i-й операции, она может быть больше или меньше фактической. **Но** $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$ для любой последовательности. Значит нужно оценивать сверху $\sum_{i=1}^{n} \widehat{c_i}$.

Пример: мультистек с операцией `multipop`

Рассмотрим стек с операциями `push`, `pop` и `multipop(k)`, которая удаляет k элементов из стека. Если [c_i^; c_i] – credit. Credit – для того, чтобы, если c_i оказалось больше $\widehat{c_i}$.

Учетные стоимости:

- `push`: 2
- `pop`: 0
- `multipop(k)`: 0

Анализ:

- При каждом `push` мы "предоплачиваем" 1 единицу "кредита" за будущий `pop` этого элемента.
- Операции `pop` и `multipop` оплачиваются за счет накопленного "кредита".
- Поскольку каждый элемент может быть удален из стека только один раз, "кредита" всегда достаточно для оплаты всех операций.
- Получаем $\sum_{i=1}^{n} \widehat{c_i} = 2n$. Значит $O(n)$.

3. Метод потенциалов (potential method)

В методе потенциалов мы вводим **потенциальную функцию** Ф, которая отображает состояние структуры данных в неотрицательное число. Амортизированная стоимость операции определяется как сумма её фактической стоимости и изменения потенциальной функции.

Требования к потенциальной функции:

- Ф(начальное состояние) = 0
- Ф(любое состояние) ≥ 0
- Анализ амортизированной стоимости операций: $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Ограничения:

- Потенциальная функция должна адекватно отражать "запас энергии" или "накопленную стоимость" в структуре данных.
- Потенциальная функция должна быть достаточно простой для вычисления, чтобы не добавлять значительную дополнительную сложность к анализу алгоритма.

- Потенциальная функция должна быть консервативной, то есть не должна искусственно занижать амортизированные затраты.

Замечание. Метод потенциалов является обобщением метода предоплаты.

Сумма разности потенциалов: $\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} c_i + (\Phi(D_n) - \Phi(D_0)) \ /\geq 0/ \geq \sum_{i=1}^{n} c_i$

Пример: Бинарный счетчик

Рассмотрим бинарный счетчик, который хранит число в двоичном виде. Операция "increment" увеличивает число на 1.

Потенциальная функция: Φ(счетчик) = количество единиц в двоичном представлении числа.

Анализ:

- Фактическая стоимость операции "increment" равна количеству битов, которые нужно перевернуть.
- Изменение потенциальной функции равно разности между количеством единиц после и до операции "increment".
- В худшем случае нужно перевернуть все биты, но при этом Φ уменьшается на $O(\log n)$.
- Амортизированная стоимость "increment" равна $O(1) + O(-\log n) = O(1)$.

Пример: multistack

$\Phi(D_i)$ – количество элементов в multistack. Условие на функцию выполняется.

- Push: $\widehat{c_i} = 1 + \Phi(D_i) - \Phi(D_{i-1}) = 2$
- Pop: $\widehat{c_i} = 1 + \Phi(D_1) - \Phi(D_{i-1}) = 0$
- MultiPop: $\widehat{c_i} = k \cdot Pop = 0$

Получаем $O(n)$, усредненная $O(1)$.

Системы непересекающихся множеств

Система непересекающихся множеств (disjoint-set data structure) - это структура данных, которая поддерживает операции над коллекцией непересекающихся множеств.

Основные операции:

- **MakeSet(x):** Создает новое множество, содержащее только элемент *x*.
- **Union(x, y):** Объединяет множества, содержащие элементы *x* и *y*.
- **FindSet(x):** Возвращает представителя множества, содержащего элемент *x*.

Оценка амортизированной стоимости операций:

С помощью амортизационного анализа можно показать, что амортизированная стоимость операций $MakeSet, Union$ и $FindSet$ в системе непересекающихся множеств с использованием **объединения по рангу** и **сжатия пути** составляет $O(\alpha(n))$, где $\alpha(n)$ - **обратная функция Аккермана**, которая растет очень медленно.

Для достижения высокой эффективности операций Union и FindSet применяются две эвристики:

1. **Объединение по рангу (Union by Rank):** При объединении двух множеств дерево с меньшим рангом присоединяется к дереву с большим рангом. Ранг - это верхняя граница высоты дерева.

2. **Сжатие пути (Path Compression):** Во время операции FindSet все узлы на пути от *x* до корня переподключаются непосредственно к корню.

Для анализа амортизированной стоимости операций над disjoint-set data structure с использованием объединения по рангу и сжатия пути применим метод потенциалов.

Определим потенциальную функцию Φ для леса непересекающихся множеств как сумму потенциалов всех узлов в лесу. Потенциал узла *x* определяется как:

- $\Phi(x) = \alpha(rank(parent(x)))$, если x не является корнем и $rank(x) < rank(parent(x))$,
- $\Phi(x) = 0$, в противном случае,

где $\alpha(n)$ - **обратная функция Аккермана**, которая растет очень медленно.

1. **MakeSet(x):**
   - Фактическая стоимость: $O(1)$
   - Изменение потенциала: $\Phi(x) = 0$ (т.к. x - корень)
   - Амортизированная стоимость: $O(1) + 0 = O(1)$
2. **Union(x, y):**
   - Фактическая стоимость: $O(1)$ (с учетом объединения по рангу)
   - Изменение потенциала:
     - Если ранги корней равны, то ранг одного из корней увеличивается на 1, что приводит к увеличению потенциала на $\alpha(n)$ в худшем случае.
     - Если ранги корней различны, то потенциал не изменяется.
   - Амортизированная стоимость: $O(1) + \alpha(n) = O(\alpha(n))$
3. **FindSet(x):**
   - Фактическая стоимость: $O(k)$, где k - длина пути от *x* до корня.
   - Изменение потенциала:
     - Сжатие пути уменьшает потенциал каждого узла на пути от x до корня, кроме корня самого.
     - Суммарное уменьшение потенциала может быть оценено как $O(k\alpha(n))$.
   - Амортизированная стоимость: $O(k) - O(k\alpha(n)) = O(\alpha(n))$

Итог:

Амортизированная стоимость операций $MakeSet$, $Union$ и $FindSet$ в системе непересекающихся множеств с использованием объединения по рангу и сжатия пути составляет $O(\alpha(n))$.

Замечания:

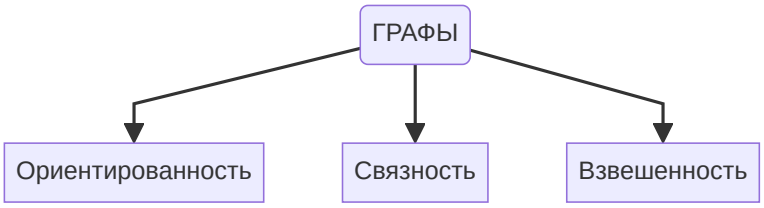
- Функция α(n) растет настолько медленно, что на практике её можно считать константой.
- Амортизированный анализ показывает, что в среднем операции над disjoint-set data structure выполняются очень быстро, несмотря на то, что некоторые отдельные операции могут быть "дорогими".

5. Графы

Графы — это мощные структуры данных, используемые для представления отношений между объектами. Они состоят из **вершин** (узлов) и **ребер**, соединяющих вершины. Графы находят широкое применение в различных областях, таких как социальные сети, транспортные системы, компьютерные сети и многие другие.

Граф — отображение $E\colon\ V \times V \to \{0, 1\}$. Граф — пары $(V; E)$.

V – vertex (вершина), E - edge (ребро).

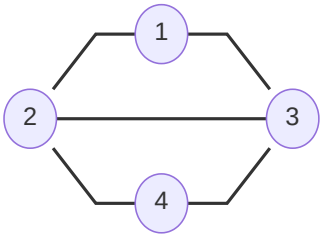


$\frac{E=|V|(|V|-1)}{2}$

Если $|V^2| \sim |E|$, то такой граф **полный**.

Если $|E| < |V^2|$, то граф называется **разреженным**.

Представление графов



Матрица смежности

Матрица смежности - это двумерный массив, где строки и столбцы соответствуют вершинам графа. Элемент матрицы `A[i][j]` равен 1, если существует ребро между вершинами *i* и *j*, и 0 в противном случае.

Преимущества:

- Простота реализации.
- Быстрая проверка наличия ребра между двумя вершинами.

Недостатки:

- Требует $O(V^2)$ памяти, где *V* - количество вершин.
- Неэффективна для разреженных графов (с небольшим количеством ребер).

	1	2	3	4
1	0	1	1	0
2	1	0	1	1

	1	2	3	4
3	1	1	0	1
4	0	1	1	0

Списки смежности

Списки смежности - это набор списков, где каждый список соответствует вершине графа и содержит её смежные вершины (те, с которыми она соединена ребрами).

Преимущества:

- Эффективна для разреженных графов.
- Требует $O(V + E)$ памяти, где E - количество ребер.

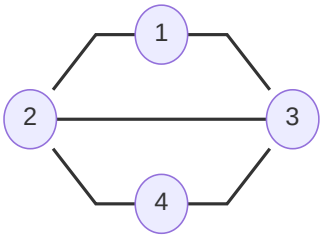
Недостатки:

- Менее эффективна для проверки наличия ребра между двумя вершинами.

Пример

- 1: [2, 3]
- 2: [1, 3, 4]
- 3: [1, 2, 4]
- 4: [2, 3]

Алгоритмы обхода графов



Поиск в ширину (BFS)

Поиск в ширину (Breadth-First Search) - это алгоритм обхода графа, который сначала посещает все вершины, смежные с начальной вершиной, затем все вершины, смежные с этими вершинами, и так далее. BFS использует очередь для хранения вершин, которые нужно посетить.

Применение:

- Нахождение кратчайшего пути в невзвешенном графе.
- Проверка связности графа.

Конкретный пример:

- **Шаг 1:** Посещаем вершину **1** и добавляем ее соседей (**2** и **3**) в очередь.
- **Шаг 2:** Извлекаем из очереди вершину **2** и добавляем ее непосещенного соседа (**4**) в очередь.
- **Шаг 3:** Извлекаем из очереди вершину **3**. Ее соседи (**2** и **4**) уже посещены.

- **Шаг 4:** Извлекаем из очереди вершину **4**. У нее нет непосещенных соседей.

Итого: 1, 2, 3, 4

```
BFS(G, v)
    opened.insert(v)
    while (!opened.empty) begin
        v = opened.pop
        foreach u in adj(v) begin
            if !(u in closed) and !(u in opened)
                opened.insert(v)
        end
    end
```

`opened` → очередь

`closed` → HashTable, set (на быстрый поиск)

Поиск в глубину (DFS)

Поиск в глубину (Depth-First Search) - это алгоритм обхода графа, который идет "вглубь" графа, посещая вершины, пока не достигнет тупика, а затем возвращается назад и исследует другие ветви. DFS использует стек (или рекурсию) для хранения вершин, которые нужно посетить.

Применение:

- Нахождение сильно связных компонент.
- Топологическая сортировка.

На примере:

1. **Посещаем вершину 1**. Вызываем рекурсивно DFS для соседа **2**.
2. **Посещаем вершину 2**. Вызываем рекурсивно DFS для непосещенного соседа **4**.
3. **Посещаем вершину 4**. Вызываем рекурсивно DFS для непосещенного соседа **3**.
4. **Посещаем вершину 3**. У нее нет непосещенных соседей. Возвращаемся на шаг 3.
5. На шаге 3 все соседи вершины **4** посещены. Возвращаемся на шаг 2.
6. На шаге 2 все соседи вершины **2** посещены. Возвращаемся на шаг 1.
7. На шаге 1 все соседи вершины **1** посещены. Алгоритм завершает работу.

Итого: 1, 2, 4, 3

Примечание:

- DFS не всегда дает единственно возможный порядок обхода. Он зависит от порядка выбора соседей для посещения
- В данном примере мы предполагаем, что соседи выбираются в порядке их нумерации в графе.

```
DFS(G, v)
    foreach v in G.V
        v.d = 0 / inf / -inf
        v.f = 0 / inf / -inf
    DFS_Visit(v)

DFS_Visit(v)
    v.d = t++
    foreach u in Adj(v)
        if u.d < 0
            DFS_Visit(u)
    v.f = t++
```

Алгоритмы поиска кратчайших путей

Алгоритм Дейкстры $O(E \log V)$

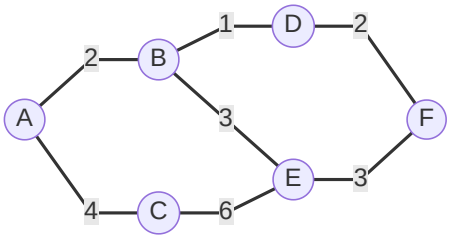
Алгоритм Дейкстры находит кратчайшие пути от одной вершины до всех остальных вершин во взвешенном графе с неотрицательными весами ребер. Алгоритм использует очередь с приоритетами для выбора вершины с наименьшим расстоянием от начальной вершины.

1. **Инициализация:**
    - Создаем таблицу расстояний `distance`, где `distance[v]` - текущее кратчайшее расстояние от вершины **A** до вершины `v`. Изначально `distance[A] = 0`, а для остальных вершин `distance[v] = ∞`.
    - Создаем множество посещенных вершин `visited`. Изначально оно пустое.
    - Добавляем вершину **A** в очередь с приоритетом `queue`, упорядоченную по возрастанию расстояния.
2. **Итерации:** Пока очередь не пуста:
    - Извлекаем из очереди вершину `u` с наименьшим расстоянием `distance[u]`.
    - Добавляем вершину `u` в множество `visited`.
    - Для каждого соседа `v` вершины `u`:
        - Если `v` не посещена и `distance[u] + weight(u, v) < distance[v]`, то обновляем расстояние до `v`: `distance[v] = distance[u] + weight(u, v)`.
        - Добавляем вершину `v` в очередь `queue`.
3. **Результат:** После завершения алгоритма `distance[F]` будет содержать длину кратчайшего пути из **A** в **F**.

Применение:

- Навигационные системы.
- Маршрутизация в сетях.

Пример.



Задача:

Найти кратчайший путь из вершины **A** в вершину **F**.

Шаг	Очередь *queue* (вершина, расстояние)	*visited*	*distance* (A, B, C, D, E, F)
1	(A, 0)	{}	(0, ∞, ∞, ∞, ∞, ∞)
2	(B, 2), (C, 4)	{A}	(0, 2, 4, ∞, ∞, ∞)
3	(D, 3), (C, 4), (E, 5)	{A, B}	(0, 2, 4, 3, 5, ∞)
4	(C, 4), (E, 5), (F, 5)	{A, B, D}	(0, 2, 4, 3, 5, 5)
5	(E, 5), (F, 5)	{A, B, C, D}	(0, 2, 4, 3, 5, 5)
6	(F, 5)	{A, B, C, D, E}	(0, 2, 4, 3, 5, 5)
7	{}	{A, B, C, D, E, F}	(0, 2, 4, 3, 5, 5)

Длина кратчайшего пути из **A** в **F** равна **5**. Один из возможных путей: **A -> B -> D -> F**.

Алгоритмы поиска минимального остовного дерева

Алгоритм Крускала $O(E \log E)$

Алгоритм Крускала находит минимальное остовное дерево (MST) во взвешенном графе. MST - это подграф, содержащий все вершины графа и минимально возможную сумму весов ребер. Алгоритм сортирует ребра по весу и добавляет их в MST, если они не создают циклов

1. **Сортировка ребер:** Создаем список всех ребер графа и сортируем его по возрастанию веса:

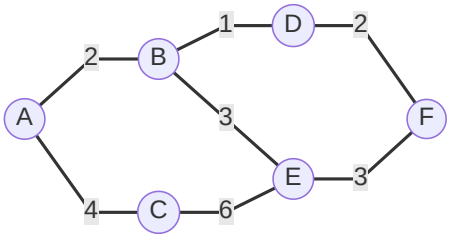
```
(B, D, 1), (A, B, 2), (D, F, 2), (B, E, 3), (E, F, 3),
(A, C, 4), (C, E, 6)
```

2. **Инициализация:**
   - Создаем пустое множество `mst`, которое будет хранить ребра MST.
   - Создаем структуру данных для хранения компонент связности вершин (например, систему непересекающихся множеств). Изначально каждая вершина находится в своей компоненте.
3. **Итерации:** Проходим по отсортированному списку ребер:
   - Для каждого ребра `(u, v)`:
     - Если вершины `u` и `v` находятся в разных компонентах связности:
       - Добавляем ребро `(u, v)` в `mst`.

- Объединяем компоненты связности вершин `u` и `v`.

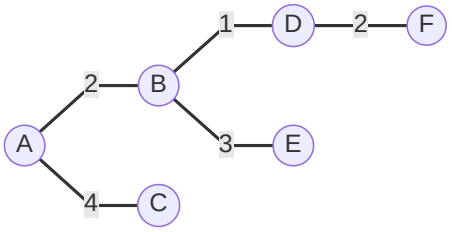
4. **Результат:** После обработки всех ребер множество `mst` будет содержать ребра минимального остовного дерева.

Пример. Найти минимальное остовное дерево (MST) для этого графа.



Пошаговое выполнение:

Шаг	Ребро (u, v, вес)	`mst`	Компоненты связности
1	(B, D, 1)	{(B, D, 1)}	{A}, {B, D}, {C}, {E}, {F}
2	(A, B, 2)	{(B, D, 1), (A, B, 2)}	{A, B, D}, {C}, {E}, {F}
3	(D, F, 2)	{(B, D, 1), (A, B, 2), (D, F, 2)}	{A, B, D, F}, {C}, {E}
4	(B, E, 3)	{(B, D, 1), (A, B, 2), (D, F, 2), (B, E, 3)}	{A, B, D, E, F}, {C}
5	(E, F, 3)	Пропускаем, т.к. E и F уже в одной компоненте	{A, B, D, E, F}, {C}
6	(A, C, 4)	{(B, D, 1), (A, B, 2), (D, F, 2), (B, E, 3), (A, C, 4)}	{A, B, C, D, E, F}
7	(C, E, 6)	Пропускаем, т.к. все вершины уже в одной компоненте	{A, B, C, D, E, F}




```
MST_Kruskal(G, w)
    A = []
    for v in G.V
        MakeSet(v)
    G.E.sort() // по возрастанию
    for u in G.E
        if FindSet(u.from) != FindSet(u.to)
            A.insert(u)
            Union(u.from, u.to)
    return A
```

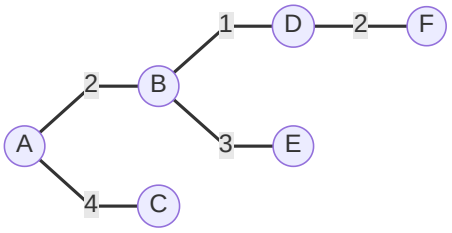
Алгоритм Прима

аАлгоритм Прима также находит MST во взвешенном графе. Алгоритм начинает с произвольной вершины и добавляет к MST ребро с наименьшим весом, соединяющее вершину в MST с вершиной вне MST. Стоимость $O((V + E) \log V)$ (для реализации с очередью с приоритетом).

1. **Инициализация:**
   - Выбираем произвольную вершину в качестве начальной (например, **A**).
   - Создаем множество `mst`, которое будет хранить ребра MST. Изначально оно пустое.
   - Создаем таблицу `key`, где `key[v]` - вес ребра с минимальным весом, соединяющего вершину `v` с деревом, построенным на данный момент. Изначально `key[A] = 0`, а для остальных вершин `key[v] = ∞`.
   - Создаем массив `parent`, где `parent[v]` - вершина, предшествующая вершине `v` в MST. Изначально `parent[A] = None`.
   - Создаем очередь с приоритетом `queue`, содержащую все вершины графа. Приоритет вершины определяется значением `key`.
2. **Итерации:** Пока очередь `queue` не пуста:
   - Извлекаем из очереди вершину `u` с минимальным ключом `key[u]`.
   - Для каждого соседа `v` вершины `u`:
     - Если `v` находится в очереди `queue` и вес ребра `(u, v)` меньше текущего ключа `key[v]`:
       - Обновляем ключ: `key[v] = weight(u, v)`.
       - Устанавливаем предка: `parent[v] = u`.
       - Обновляем приоритет вершины `v` в очереди `queue`.

Пример.

Шаг	`queue` (вершина, ключ)	`mst`	`key` (A, B, C, D, E, F)	`parent` (A, B, C, D, E, F)
1	(A, 0), (B, ∞), (C, ∞), (D, ∞), (E, ∞), (F, ∞)	{}	(0, ∞, ∞, ∞, ∞, ∞)	(None, None, None, None, None, None)
2	(B, 2), (C, 4), (D, ∞), (E, ∞), (F, ∞)	{}	(0, 2, 4, ∞, ∞, ∞)	(None, A, A, None, None, None)
3	(D, 3), (C, 4), (E, 5), (F, ∞)	{(A, B)}	(0, 2, 4, 3, 5, ∞)	(None, A, A, B, B, None)
4	(F, 5), (C, 4), (E, 5)	{(A, B), (B, D)}	(0, 2, 4, 3, 5, 5)	(None, A, A, B, B, D)
5	(C, 4), (E, 5)	{(A, B), (B, D), (D, F)}	(0, 2, 4, 3, 5, 5)	(None, A, A, B, B, D)
6	(E, 5)	{(A, B), (B, D), (D, F), (A, C)}	(0, 2, 4, 3, 5, 5)	(None, A, A, B, B, D)
7	{}	{(A, B), (B, D), (D, F), (A, C), (B, E)}	(0, 2, 4, 3, 5, 5)	(None, A, A, B, B, D)



Топологическая сортировка

Топологическая сортировка ориентированного ациклического графа (DAG) — это линейное упорядочение всех его вершин таким образом, что для любого ребра (u, v) вершина u стоит раньше вершины v в этом порядке.

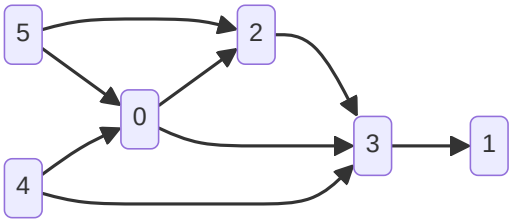
> **Note**
>
> **Ориентированный ациклический граф (Directed Acyclic Graph, DAG)** - это такой ориентированный граф, в котором отсутствуют циклы.
>
> **Дерево** — это частный случай DAG, в котором **каждая вершина (кроме корня) имеет ровно одного предка**.

Применение:

- Планирование задач.
- Анализ зависимостей.

Алгоритм топологической сортировки (на основе поиска в глубину):

1. **Инициализация:**
   - Создаем список `sorted`, который будет хранить отсортированные вершины.
   - Создаем словарь `visited`, где `visited[v]` - True, если вершина `v` уже посещена, иначе False. Изначально все вершины не посещены.
2. **Обход в глубину:**
   - Для каждой вершины `u` в графе:
     - Если вершина `u` не посещена (`visited[u] == False`):
       - Вызываем функцию `dfs(u)`.
3. **Функция dfs(u):**
   - Помечаем вершину `u` как посещенную: `visited[u] = True`.
   - Для каждого соседа `v` вершины `u`:
     - Если вершина `v` не посещена:
       - Рекурсивно вызываем `dfs(v)`.
   - Добавляем вершину `u` в начало списка `sorted`.
4. **Результат:** После завершения алгоритма список `sorted` будет содержать топологически отсортированные вершины.



Выполнение алгоритма:

1. **Инициализация:** `sorted = []`, `visited = {0: False, 1: False, 2: False, 3: False, 4: False, 5: False}`.
2. **Обход в глубину:**
   - `dfs(5)`: Посещаем вершины 5, 2, 3, 1 (в таком порядке) и добавляем их в начало `sorted`. `sorted = [1, 3, 2, 5]`
   - `dfs(4)`: Посещаем вершины 4, 0, и добавляем их в начало `sorted`. `sorted = [0, 4, 1, 3, 2, 5]`
   - Вызовы `dfs(0)`, `dfs(2)`, `dfs(3)`, `dfs(1)` не выполнятся, так как эти вершины уже посещены.
3. **Результат:** `sorted = [0, 4, 1, 3, 2, 5]` - одна из возможных топологических сортировок данного графа.

Алгоритм поиска сильно связных компонент

Сильно связная компонента (SCC) - это максимальное подмножество вершин ориентированного графа, в котором существует путь между любой парой вершин. Алгоритм поиска SCC использует два прохода DFS для нахождения всех SCC в графе.

Применение:

- Анализ социальных сетей.
- Обнаружение циклов в графах.

Алгоритм:

STRONGLY-CONNECTED-COMPONENTS(G)
1 Вызов DFS(G) для вычисления времен завершения $u.f$
  для каждой вершины u
2 Вычисление G^{T}
3 Вызов DFS(G^{T}), но в основном цикле процедуры DFS
  вершины рассматриваются в порядке убывания значений $u.f$,
  вычисленных в строке 1
4 Вывод вершин каждого дерева в лесу поиска в глубину,
  полученного в строке 3, в качестве
  отдельного сильно связного компонента

1. **`STRONGLY-CONNECTED-COMPONENTS(G)`:** Функция, запускающая алгоритм поиска SCC для графа `G`.
2. **`Вызов DFS(G) для вычисления времен завершения u.f для каждой вершины u`:**
   - Выполняем первый проход DFS на исходном графе `G`.

- Для каждой вершины `u` записываем время ее завершения `u.f` (время, когда рекурсивный вызов DFS для `u` завершается).

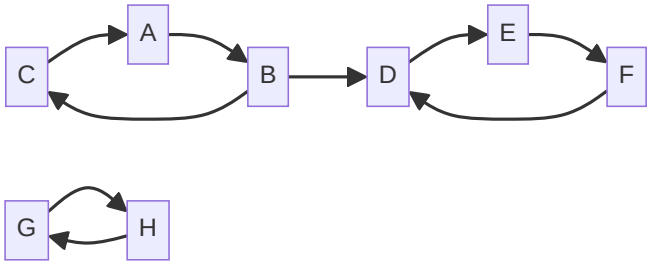
3. **`Вычисление G^T`:** Строим транспонированный граф `G^T`, меняя направление всех ребер в `G`.
4. **`Вызов DFS(G^T), но в основном цикле процедуры DFS вершины рассматриваются в порядке убывания значений u.f, вычисленных в строке 1`:**
   - Выполняем второй проход DFS на транспонированном графе `G^T`.
   - **Ключевой момент:** Вершины выбираются для запуска DFS **в порядке убывания времен завершения**, полученных в первом проходе.
5. **`Вывод вершин каждого дерева в лесу поиска в глубину, полученного в строке 3, в качестве отдельного сильно связного компонента`:**
   - Каждое дерево в лесу DFS, построенном во втором проходе, представляет собой одну SCC.

Почему этот алгоритм работает?

- **Сортировка по времени завершения:** Вершины с бóльшим временем завершения в первом проходе DFS "глубже" расположены в графе.
- **Транспонированный граф:** В транспонированном графе ребра "перевернуты", поэтому DFS из вершины `u` в `G^T` достигает только тех вершин, из которых можно добраться до `u` в исходном графе `G`.
- **Объединение:** Запуская DFS из вершин в порядке убывания `u.f`, мы гарантируем, что сначала будут найдены SCC, состоящие из "глубоких" вершин, а затем - из вершин, расположенных "выше" в графе.

Пример.

1. **Рассмотрим ориентированный граф `G` с вершинами `A, B, C, D, E, F, G, H` и следующими ребрами:**



2. **Первый DFS и времена завершения.** Выполняем DFS на графе `G` (начальная вершина не важна) и записываем времена завершения для каждой вершины:

   Время можно посчитать вот так:

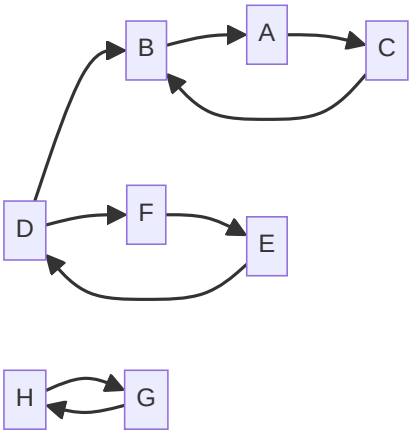
```python
def dfs_finish_times(graph):
  visited = set()
  finish_times = {}
  time = 0

  def dfs(node):
    nonlocal time
    visited.add(node)
    time += 1

```

```
    for neighbor in graph[node]:
      if neighbor not in visited:
        dfs(neighbor)

    time += 1
    finish_times[node] = time

  for node in graph:
    if node not in visited:
      dfs(node)

  return finish_times

graph = {
  'A': ['B'],
  'B': ['C', 'D'],
  'C': ['A'],
  'D': ['E'],
  'E': ['F'],
  'F': ['D'],
  'G': ['H'],
  'H': ['G'],
}

finish_times = dfs_finish_times(graph)
print(finish_times)

# {'C': 4, 'F': 8, 'E': 9, 'D': 10, 'B': 11, 'A': 12, 'H': 15, 'G': 16}
```

- `A.f = 12`
- `B.f = 11`
- `C.f = 4`
- `D.f = 10`
- `E.f = 9`
- `F.f = 8`
- `G.f = 16`
- `H.f = 15`

3. **Транспонированный граф.** Строим транспонированный граф `G^T`:

```
B -> A
C -> B
A -> C
D -> B
E -> D
F -> E
D -> F
H -> G
G -> H
```



4. **Второй DFS на G^T в порядке убывания времен завершения:**
   Выполняем DFS на `G^T`, выбирая вершины в порядке убывания `u.f`:
   1. `G (G.f = 16)`: DFS из `G` посещает `H`. Это первая SCC: `{G, H}`.
   2. `A (A.f = 12)`: DFS из `A` посещает `C, B`. Это вторая SCC: `{A, C, B}`.
   3. `D (D.f = 10)`: DFS из `D` посещает `F, E`. Это третья SCC: `{D, F, E}`.
5. **Результат:**

   Алгоритм нашел три сильно связные компоненты в графе `G`:

   - `{G, H}`
   - `{A, C, B}`
   - `{D, F, E}`

6. Хэш-функции, хэш-таблицы

Хэш-таблицы — это эффективные структуры данных для реализации словарей, которые поддерживают операции вставки, удаления и поиска за время $O(1)$ в среднем случае. Они основаны на использовании хэш-функций, которые отображают ключи на индексы в таблице.

Требования к хеш-функциям

1. Реализуются для любых данных
2. Размер выхода не зависит от размера входа
3. Необратимы
4. Хорошее перемешивание
5. Сложно скомпрометировать
6. Быстро считаются

Замечания. При создании хэш-функций стараются сделать так, чтобы количество коллизий было минимальным. Хэш-функции считаются за $O(1)$. Размер хэш-таблицы изменить в процессе тяжело. Поэтому лучше заранее брать достаточно большой размер.

Открытая адресация и таблицы на основе цепочек

Существует два основных метода разрешения коллизий (когда разные ключи отображаются на один индекс):

- **Открытая адресация:**
  - При коллизии ищется другая свободная ячейка в таблице с помощью **последовательности проб** (probing sequence).
  - **Линейное пробирование:** Проверяются ячейки $h(k) + i \mod m$, где $h(k)$ - хэш-функция, m - размер таблицы, i - номер попытки.
  - **Квадратичное пробирование:** Проверяются ячейки $h(k) + c_1 i + (c_2 i^2 \mod m)$, где c_1 и c_2 - константы.
  - **Двойное хеширование:** Используется вторая хэш-функция $h'(k)$ для определения шага пробирования: $h(k) + ih'(k) \mod m$.
  - ***Маркеры.***
    - NIL — ячейка пуста
    - DEL — удаленный элемент (для поиска других элементов)
    - $Assert$ — пробируем до ячейки DEL / NIL
    - $Find$ — ищем до NIL или «нашли элемент»
  - ***Замечания.***
    - Нормальная хэш-функция для каждого ключа должна перебирать все клетки таблицы.
    - Открытая адресация позволяет сохранить память.
    - Если из таблицы часть удалять, то она забивается маркерами DEL, что увеличивает время поиска, исправить проблему может создание новой таблицы (перехэширование).

    - Открытая адресация хорошо, когда мы работаем без удалений.
- **Таблицы на основе цепочек:**
    - В каждой ячейке таблицы хранится список (цепочка) элементов, хэширующихся на этот индекс.
    - При коллизии новый элемент добавляется в список.
    - Время поиска элемента = время вставки этого элемента на момент, когда его еще не было в таблице.

Пример (цепочки)

Предположим, у нас есть хэш-таблица размером 10 и простая хэш-функция: сумма ASCII-кодов букв имени по модулю 10.

1. Вставим "Alice: 25"
   Хэш("Alice") = (65 + 108 + 105 + 99 + 101) % 10 = 478 % 10 = 8
   Помещаем в ячейку 8: [8] -> Alice: 25
2. Вставим "Bob: 30"
   Хэш("Bob") = (66 + 111 + 98) % 10 = 275 % 10 = 5
   Помещаем в ячейку 5: [5] -> Bob: 30
3. Вставим "Charlie: 35"
   Хэш("Charlie") = (67 + 104 + 97 + 114 + 108 + 105 + 101) % 10 = 696 % 10 = 6
   Помещаем в ячейку 6: [6] -> Charlie: 35
4. Вставим "David: 28"
   Хэш("David") = (68 + 97 + 118 + 105 + 100) % 10 = 488 % 10 = 8
   Коллизия! Ячейка 8 уже занята. Используем метод цепочек:
   [8] -> Alice: 25 -> David: 28

Теперь наша хэш-таблица выглядит так:

```
[0]
[1]
[2]
[3]
[4]
[5] -> Bob: 30
[6] -> Charlie: 35
[7]
[8] -> Alice: 25 -> David: 28
[9]
```

Чтобы найти возраст Чарли, мы вычисляем хэш("Charlie") = 6, идем в ячейку 6 и получаем значение 35. Для поиска возраста Дэвида, мы вычисляем хэш("David") = 8, идем в ячейку 8, проходим по цепочке и находим нужную запись. Этот пример демонстрирует основные принципы работы хэш-таблицы, включая вставку, разрешение коллизий методом цепочек и поиск элементов.

Пример (открытая адресация)

Предположим, у нас та же хэш-таблица размером 10 и та же хэш-функция: сумма ASCII-кодов букв имени по модулю 10.

1. Вставим "Alice: 25"
   Хэш("Alice") = 8
   Помещаем в ячейку 8: [8] Alice: 25
2. Вставим "Bob: 30"
   Хэш("Bob") = 5
   Помещаем в ячейку 5: [5] Bob: 30
3. Вставим "Charlie: 35"
   Хэш("Charlie") = 6
   Помещаем в ячейку 6: [6] Charlie: 35
4. Вставим "David: 28"
   Хэш("David") = 8
   Ячейка 8 занята. Пробуем следующую: 9
   Помещаем в ячейку 9: [9] David: 28
5. Вставим "Eve: 40"
   Хэш("Eve") = 8
   Ячейки 8 и 9 заняты. Пробуем следующую: 0
   Помещаем в ячейку 0: [0] Eve: 40

Теперь наша хэш-таблица выглядит так:

```
[0] Eve: 40
[1]
[2]
[3]
[4]
[5] Bob: 30
[6] Charlie: 35
[7]
[8] Alice: 25
[9] David: 28
```

Поиск элемента. Чтобы найти возраст Чарли, мы вычисляем хэш("Charlie") = 6, идем в ячейку 6 и получаем значение 35.

Для поиска возраста Дэвида:

1. Вычисляем хэш("David") = 8
2. Проверяем ячейку 8 - там Alice
3. Проверяем следующую ячейку (9) - находим David: 28

Для поиска возраста Евы:

1. Вычисляем хэш("Eve") = 8
2. Проверяем ячейку 8 - там Alice
3. Проверяем ячейку 9 - там David
4. Проверяем ячейку 0 - находим Eve: 40

Удаление элемента. При удалении элемента в открытой адресации нельзя просто очистить ячейку, так как это может нарушить цепочку поиска для других элементов. Вместо этого обычно используют специальный маркер "удалено".

Например, если мы удалим "Alice":
[8] DELETED

Этот пример демонстрирует, как работает открытая адресация с линейным пробированием для разрешения коллизий в хэш-таблице.

Оценка стоимости операций

	CHAIN	CHAIN (repeats)	OPEN ADR.	OPEN ADR. (repeats)
Assert	$\Theta(1+\alpha)$	$\Theta(1)$	$\frac{1}{1-\alpha}$ – ожидаемое …	… число проб
Find	$\Theta(1+\alpha)$	$\Theta(1)$	$\frac{1}{\alpha}m\left(\frac{1}{1-\alpha}\right)$	
Remove	$\Theta(1+\alpha)$	$\Theta(1)$		

$\alpha = \frac{n}{m}\left(\frac{\textit{кол-во добавленных элементов}}{\textit{кол-во ячеек таблицы}}\right) - load\_factor\ (\textit{коэфф. заполненности})$

Замечание. Так как размер таблицы заранее выбирается большой, то $\alpha < 1$.

Базовые виды хэш-функций

- **Метод деления:** $h(k) = k \bmod m$, где m - размер таблицы. m должно быть простым числом, не близким к степени 2.
- **Метод умножения:** $h(k) = floor(m(kA \mod 1))$, где A - константа в интервале $(0, 1)$.
- **Универсальное хеширование:** Используется семейство хэш-функций, выбираемых случайным образом.

Проблемы кластеризации и удаления элементов

- **Кластеризация:** При открытой адресации коллизии могут приводить к образованию кластеров занятых ячеек, что увеличивает время поиска.
- **Удаление элементов:** При открытой адресации удаление элементов может нарушить последовательности проб, что требует специальной обработки (использование маркеров).

Реализация словаря на основе хэш-таблицы

Хэш-таблицы предоставляют эффективный способ реализации структуры данных "словарь" (dictionary), которая хранит пары "ключ-значение" и обеспечивает быстрый доступ к значению по ключу. Рассмотрим подробнее реализацию основных операций словаря с использованием хэш-таблицы:

1. `Insert(key, value)` - Вставка пары (ключ, значение):

1. **Вычисление хэш-значения:** Вычисляем хэш ключа с помощью хэш-функции: `index = hash(key) % table_size`.
2. **Разрешение коллизий:** Если ячейка `index` уже занята другой парой с другим ключом:
   - **Открытая адресация:** Ищем следующую свободную ячейку в соответствии с выбранной стратегией пробирования (линейное, квадратичное, двойное хеширование).
   - **Цепочки:** Добавляем новую пару (ключ, значение) в список (цепочку), связанный с данной ячейкой.
3. **Вставка:** Помещаем новую пару (ключ, значение) в найденную свободную ячейку или в список.

2. `Delete(key)` - Удаление пары с заданным ключом:

1. **Поиск ячейки:** Вычисляем хэш ключа и находим соответствующую ячейку, как при операции `Search`.
2. **Удаление:**
   - **Открытая адресация:** Помечаем ячейку как удаленную (например, специальным маркером), не удаляя данные физически, чтобы не нарушать логику пробирования.
   - **Цепочки:** Удаляем пару (ключ, значение) из списка, связанного с данной ячейкой.

3. `Search(key)` - Поиск значения по ключу:

1. **Вычисление хэш-значения:** Вычисляем хэш ключа.
2. **Поиск в таблице:**
   - **Открытая адресация:** Проверяем ячейку, соответствующую хэшу ключа. Если ключ не совпадает, применяем стратегию пробирования, пока не найдем нужный ключ или пустую ячейку (ключ не найден).
   - **Цепочки:** Проходим по списку, связанному с вычисленным хэшем, и сравниваем ключи элементов списка с искомым.

Пример реализации:

```cpp
#include <iostream>
#include <vector>
#include <list>

using namespace std;

template <typename Key, typename Value>
class HashTable {
private:
    vector<list<pair<Key, Value>>> table;
    size_t capacity;

    size_t _hash(const Key& key) const {
        return hash<Key>{}(key) % capacity;
    }

public:
    HashTable(size_t capacity) : capacity(capacity), table(capacity) {}

    void insert(const Key& key, const Value& value) {
        size_t index = _hash(key);
        table[index].emplace_back(key, value);
    }

    Value* search(const Key& key) {
        size_t index = _hash(key);
        for (auto& pair : table[index]) {
            if (pair.first == key) {
                return &pair.second;
            }
        }
        return nullptr;  // Key not found
    }

```

```
    void erase(const Key& key) {
        size_t index = _hash(key);
        auto& list = table[index];
        for (auto it = list.begin(); it != list.end(); ++it) {
            if (it->first == key) {
                list.erase(it);
                return;
            }
        }
        // Key not found, throw an exception or handle accordingly
    }
};

int main() {
    HashTable<string, int> table(10);
    table.insert("apple", 1);
    table.insert("banana", 2);

    int* value = table.search("banana");
    if (value != nullptr) {
        cout << "Value for 'banana': " << *value << endl;
    } else {
        cout << "Key 'banana' not found" << endl;
    }

    table.erase("apple");

    value = table.search("apple");
    if (value != nullptr) {
        cout << "Value for 'apple': " << *value << endl;
    } else {
        cout << "Key 'apple' not found" << endl;
    }

    return 0;
}
```

Важно:

- **Выбор хэш-функции и стратегии разрешения коллизий** влияет на производительность операций словаря.
- **Обработка ошибок.** Например, обработка ситуации, когда ключ не найден при удалении или поиске.
- **Динамическое изменение размера.** При большом количестве элементов может потребоваться увеличить размер таблицы и перехешировать данные для поддержания производительности.

Усовершенствованные методы хеширования

- **Хеширование с перескоком (Hopscotch hashing):**
  - Рассматривает «окружение» ячейки, которое формирует «ведерко» (bucket). Элемент размещается не дальше окружения ячейки, совпадающей с хэшем.
  - **Общая идея:**
    - Если ячейка i свободна, то разместить x в i и выйти;

- Начиная с i-й ячейки, линейным пробированием найти свободную ячейку j;
- Если j лежит в диапазоне $H-1$ от i, то разместить x в j и выйти;
- Иначе найти y с хэш-значением между i и j, чтобы он попадал в окрестность $H-1$. Переместить y;
- Если ничего не удалось, то перестроить таблицу.

```
       ↓ H-bucket ↓
... | |z| | |x| | ...
 h(x)-^.->->.^
```

- Если все занято: ищем элемент, который можно выгнать (т.к. bucket'ы накладываются). Двигаем, пока не найдем пустую ячейку для всех «выгнанных» элементов.
- Если все занято 2: ищем первую свободную ячейку. Далее, возвращаясь назад, ищем элемент, который можно переместить в эту ячейку. Перемещаем, пока не найдем место для вставленного элемента.
- Если все занято — отказ.

  - **Достоинства:** быстрый поиск и удаление, т.к. не рассматриваем хэширование дальше bucket'а.

- **Хеширование Робин Гуда (Robin Hood hashing):**
  - Обычное хэширование с контролем номера пробы. Если встречаем при вставке элемент с меньшим числом проб, то вытесняем его и ищем ему другое место.
  - **Достоинства:** быстрый поиск и удаление.
- **Двухвыборное хеширование:**
  - Используется две хэш-функции, и элемент вставляется в ту ячейку, где меньше коллизий.
  - Таблица на цепочках.
  - Балансирует длины списков.
  - Уменьшает вероятность кластеризации.
- **Кукушкино хеширование (Cuckoo hashing):**
  - Обычно на $20-30\%$ хуже линейного пробирования.
  - Используется две или более хэш-функции и две или более таблицы.
  - Элемент может быть помещен в одну из двух ячеек, определяемых хэш-функциями.
  - При коллизии элемент "выталкивается" из своей ячейки и перемещается в другую таблицу, что может вызвать цепочку перемещений.
  - Гарантирует время поиска $O(1)$ в худшем случае.
  - **Проблема:** может зациклиться, может быть долга вставка.
  - В целом, можно две хэш-функции использовать в одной таблице.
  - **Достоинство:** быстрый поиск, близко к массиву, быстрое удаление.

Фильтр Блума

Bloom Filter позволяет проверить принадлежность элемента множеству. Может ложно подтвердить наличие элемента при его отсутствии. Если подтверждает отсутствие, то это всегда истина.

Общая идея:

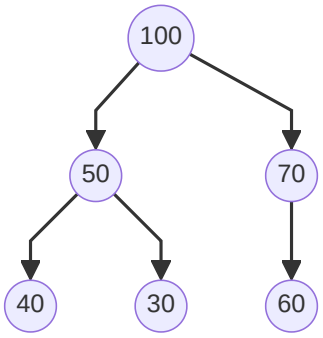
```
     |h1 |h2     |h3        # Вставка / вставленная строка
     V   V       V
... |1| |1| |0| |1| | ...
     ^   ^   ^
     |h1 |h2 |h3            # Поиск / значит такой строки нет
```

Замечание. Удалять из Фильтра Блума нельзя. Но работает с объединением фильров. Используется в ситуациях, когда требуется высокая эффективность по времени и памяти.

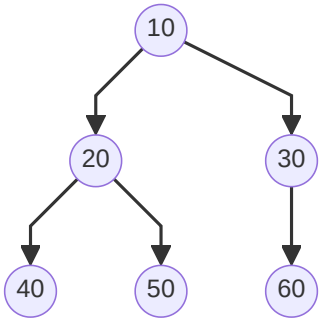
7. Кучи (Heaps)

Куча — это структура данных, представляющая собой **полное бинарное дерево**, для которого выполняется **свойство кучи**. Существуют два основных типа куч:

- **max-куча**: ключ каждого узла **больше или равен** ключам его потомков.

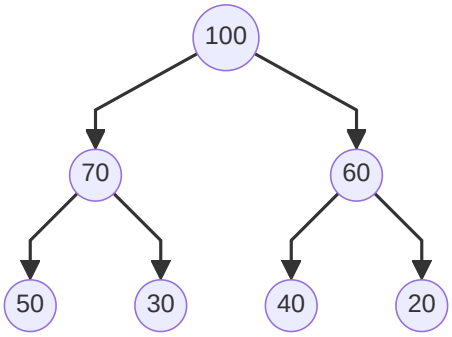


- **min-куча**: ключ каждого узла **меньше или равен** ключам его потомков.



Двоичные кучи (Binary Heaps)

Двоичная куча (binary heap) — это наиболее распространенная реализация кучи, которая использует массив для хранения элементов дерева.



Свойства:

1. **Полное бинарное дерево:** Все уровни дерева заполнены, кроме, возможно, последнего, который заполняется слева направо.
2. **Свойство кучи:** Для каждого узла выполняется свойство max-кучи или min-кучи.
3. Куча не сохраняет валидность итераторов при операциях с ней.

Представление в массиве:

- Корень дерева находится в ячейке с индексом 1.

- Для узла с индексом i:
  - Левый потомок: $2i$
  - Правый потомок: $2i + 1$
  - Родитель: $\lfloor i/2 \rfloor$

Высота кучи: $\lfloor \log_2 n \rfloor$.

Число слоев: $\lceil \log_2(n+1) \rceil$.

Количество элементов: $[2^h; 2^{h+1} - 1] \in \mathbb{N},\ \log(N+1) - 1 \leq h \leq \log(N)$

Операции над двоичной кучей

- **`Heapify(A, i)`:** Восстанавливает свойство кучи для поддерева с корнем в узле i, предполагая, что поддеревья левого и правого потомков уже удовлетворяют свойству кучи.
- **`BuildHeap(A)`:** Строит кучу из неупорядоченного массива A.
- **`ExtractMax(A)` (для max-кучи) / `ExtractMin(A)` (для min-кучи):** Извлекает и возвращает максимальный (минимальный) элемент из кучи.
- **`Insert(A, key)`:** Вставляет новый ключ key в кучу.
- **`IncreaseKey(A, i, key)` (для max-кучи) / `DecreaseKey(A, i, key)` (для min-кучи):** Увеличивает (уменьшает) значение ключа в узле i до key.

Примеры реализации на псевдокоде:

1. `Heapify(A, i)`:

```
Heapify(A, i)
  left = 2 * i
  right = 2 * i + 1
  largest = i

  // Находим индекс наибольшего элемента среди узла i, его левого и правого потомков
  if left <= heap_size(A) and A[left] > A[largest] then
    largest = left
  end if

  if right <= heap_size(A) and A[right] > A[largest] then
    largest = right
  end if

  // Если наибольший элемент не в узле i, меняем местами и рекурсивно вызываем Heapify
  if largest != i then
    swap(A[i], A[largest])
    Heapify(A, largest)
  end if
```

2. `BuildHeap(A)`:

```
BuildHeap(A)
  // Начинаем с последнего нелистового узла и идем к корню
  for i = floor(heap_size(A) / 2) downto 1 do
    Heapify(A, i)
  end for
```

3. **`ExtractMax(A)`:**

```
ExtractMax(A)
  if heap_size(A) < 1 then
    error "Куча пуста"
  end if

  max = A[1]  // Максимальный элемент всегда в корне
  A[1] = A[heap_size(A)]  // Перемещаем последний элемент в корень
  heap_size(A) = heap_size(A) - 1  // Уменьшаем размер кучи
  Heapify(A, 1)  // Восстанавливаем свойство кучи
  return max
```

4. **`Insert(A, key)`:**
Элемент добавляется в конец массива, сравнивается с родителем. Если родитель больше, то они меняются местами и сравнение продолжается, пока родитель не окажется меньше или элемент не станет корнем.

```
Insert(A, key)
  heap_size(A) = heap_size(A) + 1  // Увеличиваем размер кучи
  A[heap_size(A)] = key  // Вставляем новый ключ в конец массива
  i = heap_size(A)

  // Восстанавливаем свойство кучи, поднимая новый ключ вверх
  while i > 1 and A[parent(i)] < A[i] do
    swap(A[i], A[parent(i)])
    i = parent(i)
  end while
```

5. **`IncreaseKey(A, i, key)`:**

```
IncreaseKey(A, i, key)
  if key < A[i] then
    error "Новый ключ меньше текущего"
  end if

  A[i] = key  // Увеличиваем значение ключа

  // Восстанавливаем свойство кучи, поднимая измененный ключ вверх
  while i > 1 and A[parent(i)] < A[i] do
    swap(A[i], A[parent(i)])
    i = parent(i)
  end while
```

6. **`Pop(A)`:**

```
Pop(A)
    A[0] = A[N - 1]
    N--
    Heapify(A, 0)
```

Куча позволяет удалять только корень.

Вспомогательные функции:

- `heap_size(A)`: возвращает текущий размер кучи (количество элементов).
- `parent(i)`: возвращает индекс родительского узла для узла с индексом i (равен $\lfloor i/2 \rfloor$).
- `swap(x, y)`: меняет местами значения переменных x и y.

Важно:

- В этом псевдокоде предполагается, что индексация массива начинается с 1.
- Реализация `heap_size` может быть разной в зависимости от выбранной структуры данных (массив фиксированного размера или динамический массив).

Сортировка с помощью двоичной кучи (Heap Sort)

Алгоритм сортировки с помощью двоичной кучи (Heap Sort) состоит из двух этапов:

1. **Построение кучи:** `BuildHeap(A)` преобразует исходный массив в max-кучу.
2. **Сортировка:**
    - Повторяем $n - 1$ раз:
        - Извлекаем максимальный элемент из кучи: `max = ExtractMax(A)`.
        - Помещаем max в конец массива (на освободившееся место).
        - Уменьшаем размер кучи на 1.

Еще как вариант:

```
HeapSort(A)
    N = len(A)
    BuildHeap(A)
    for i = N - 1 to 1 do begin
        swap(A[0], A[N - 1])
        N--
        Heapify(A, 0)
    end
    return A
```

Оценка эффективности

- **`Heapify(A, i)`**: $O(\log n)$
- **`BuildHeap(A)`**: $O(n)$ (удивительно, но линейное время!)
- **`ExtractMax(A)`** / **`ExtractMin(A)`**: $O(\log n)$
- **`Insert(A, key)`**: $O(\log n)$
- **`IncreaseKey(A, i, key)`** / **`DecreaseKey(A, i, key)`**: $O(\log n)$
- **`Heap Sort`**: $O(n \log n)$ в худшем случае.
- **`Pop(A)`**: $O(\log N)$

8. Стратегии переборных алгоритмов

Переборные алгоритмы (brute-force algorithms) — это алгоритмы, которые основаны на полном переборе всех возможных решений задачи. Они гарантированно находят оптимальное решение, но могут быть очень неэффективными для задач с большим пространством решений.

Полный перебор

Полный перебор (exhaustive search) — это простейшая стратегия перебора, которая заключается в проверке всех возможных решений без каких-либо оптимизаций.

Пример: Задача коммивояжера

В задаче коммивояжера (Traveling Salesman Problem, TSP) нужно найти кратчайший маршрут, проходящий через все города ровно один раз и возвращающийся в исходный город.

- **Полный перебор:** Генерируем все возможные перестановки городов и вычисляем длину маршрута для каждой перестановки. Выбираем перестановку с минимальной длиной.
- **Сложность:** $O(n!)$, где n – количество городов. Факториал растет очень быстро, поэтому полный перебор применим только для небольших значений n.

Метод ветвей и границ (Branch and Bound)

Метод ветвей и границ (Branch and Bound) — это более эффективная стратегия перебора, которая использует **оценки** для отсечения заведомо неоптимальных решений.

Идея:

1. **Разбиение:** Разбиваем пространство решений на подмножества (ветви).
2. **Оценка:** Для каждой ветви вычисляем **нижнюю границу** (lower bound) для оптимального решения в этой ветви.
3. **Отсечение:** Если нижняя граница для ветви больше, чем текущее наилучшее решение, то эту ветвь можно отсечь, так как она не может содержать оптимального решения.
4. **Рекурсия:** Рекурсивно применяем метод ветвей и границ к оставшимся ветвям.

Пример: Задача коммивояжера

- **Разбиение:** На каждом шаге выбираем город, который еще не посещен, и создаем ветви для каждого возможного следующего города.
- **Оценка:** Нижняя граница для ветви может быть вычислена как сумма длин уже пройденных ребер и минимальных длин ребер, соединяющих оставшиеся города.
- **Отсечение:** Если нижняя граница для ветви больше, чем текущая минимальная длина маршрута, то эту ветвь можно отсечь.

Пример реализации на псевдокоде (упрощенный):

```
BranchAndBound(cities, current_city, visited, current_length, best_length)
  if visited == all_cities then
    // Посетили все города, проверяем длину маршрута
    if current_length < best_length then
      best_length = current_length
```

```
    end if
    return
  end if

  // Перебираем непосещенные города
  for next_city in cities where next_city not in visited do
    // Вычисляем нижнюю границу для ветви
    lower_bound = current_length + distance(current_city, next_city) +
min_remaining_distances(cities, visited)

    // Отсекаем ветвь, если ее нижняя граница больше текущего лучшего
решения
    if lower_bound < best_length then
      BranchAndBound(cities, next_city, visited + next_city, current_length
+ distance(current_city, next_city), best_length)
    end if
  end for
```

9. Динамическое программирование

Динамическое программирование (dynamic programming) — это мощный метод решения задач, который основан на **разбиении задачи на подзадачи** и **использовании решений подзадач для решения исходной задачи**. Он применим к задачам, которые обладают **свойством оптимальной подструктуры** и **свойством перекрывающихся подзадач**.

Признаки применимости

1. **Свойство оптимальной подструктуры:** Оптимальное решение задачи может быть построено из оптимальных решений её подзадач.
2. **Свойство перекрывающихся подзадач:** В процессе решения задачи одни и те же подзадачи возникают многократно.

Общая стратегия

1. **Разбиение на подзадачи:** Разбиваем задачу на подзадачи меньшего размера.
2. **Рекуррентное соотношение:** Записываем рекуррентное соотношение, выражающее решение задачи через решения её подзадач.
3. **Запоминание решений:** Сохраняем решения подзадач в таблице (или другой структуре данных), чтобы избежать повторных вычислений.
4. **Построение решения:** Используем таблицу с сохраненными решениями для построения решения исходной задачи.

Стратегии реализации

1. Сверху вниз (с мемоизацией)

- **Рекурсивный подход:** Реализуем рекурсивную функцию, которая вычисляет решение задачи.
- **Мемоизация:** Перед вычислением решения подзадачи проверяем, не было ли оно уже вычислено ранее. Если решение найдено в таблице, используем его; иначе, вычисляем решение и сохраняем его в таблице.

Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ.

2. Снизу вверх

- **Итеративный подход:** Заполняем таблицу с решениями подзадач, начиная с самых маленьких подзадач и двигаясь к исходной задаче.
- **Использование рекуррентного соотношения:** Для вычисления решения каждой подзадачи используем рекуррентное соотношение и уже вычисленные решения меньших подзадач.

Примеры

1. Алгоритм перемножения матриц

- **Задача:** Найти оптимальный порядок перемножения цепочки матриц, минимизирующий количество скалярных умножений.
- **Динамическое программирование:**
  - Подзадачи: Перемножение подцепочек матриц.
  - Рекуррентное соотношение: `m[i, j] = min(m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j])`, где `m[i, j]` - минимальное количество умножений для перемножения матриц от `i` до `j`, а `p` - массив размерностей матриц.
  - Запоминание: Таблица `m`.
  - Построение решения: Восстановление оптимального порядка умножения из таблицы `m`.

```
function matrix_chain_order(p):
  // p - массив размерностей матриц, где p[i-1] x p[i] - размерность i-й матрицы
  n = length(p) - 1  // Количество матриц

  // Инициализируем таблицу m нулями
  m = create_matrix(n, n, 0)

  // Заполняем таблицу m по диагоналям, начиная с главной диагонали
  for l = 2 to n:  // l - длина цепочки матриц
    for i = 1 to n - l + 1:
      j = i + l - 1
      m[i][j] = infinity
      for k = i to j - 1:
        q = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
        if q < m[i][j]:
          m[i][j] = q
          s[i][j] = k  // Сохраняем индекс k для восстановления порядка умножения

  // Восстанавливаем порядок умножения из таблицы s
  return get_optimal_parenthesization(s, 1, n)

function get_optimal_parenthesization(s, i, j):
  // Рекурсивно строим строку с оптимальным порядком умножения
  if i == j:
    return "A" + str(i)
  else:
    return "(" + get_optimal_parenthesization(s, i, s[i][j]) +
           get_optimal_parenthesization(s, s[i][j]+1, j) + ")"
```

2. Поиск наибольшей общей подпоследовательности (LCS)

- **Задача:** Найти наибольшую общую подпоследовательность двух последовательностей.
- **Динамическое программирование:**
  - Подзадачи: LCS для префиксов последовательностей.
  - Рекуррентное соотношение: `c[i, j] = c[i-1, j-1] + 1`, если `x[i] == y[j]`, иначе `c[i, j] = max(c[i-1, j], c[i, j-1])`, где `c[i, j]` - длина LCS для префиксов `x[1..i]` и `y[1..j]`.
  - Запоминание: Таблица `c`.
  - Построение решения: Восстановление LCS из таблицы `c`.

```
function lcs_length(x, y):
  // x, y - входные последовательности
  m = length(x)
  n = length(y)

  // Инициализируем таблицу c нулями
  c = create_matrix(m + 1, n + 1, 0)

  // Заполняем таблицу c
  for i = 1 to m:
    for j = 1 to n:
      if x[i] == y[j]:
        c[i][j] = c[i-1][j-1] + 1
      else:
        c[i][j] = max(c[i-1][j], c[i][j-1])

  // Восстанавливаем LCS из таблицы c
  return backtrack_lcs(c, x, y, m, n)

function backtrack_lcs(c, x, y, i, j):
  // Рекурсивно восстанавливаем LCS
  if i == 0 or j == 0:
    return ""
  elseif x[i] == y[j]:
    return backtrack_lcs(c, x, y, i-1, j-1) + x[i]
  else:
    if c[i-1][j] > c[i][j-1]:
      return backtrack_lcs(c, x, y, i-1, j)
    else:
      return backtrack_lcs(c, x, y, i, j-1)
```

3. Задача о рюкзаке (Knapsack Problem)

- **Задача:** Выбрать предметы из набора, чтобы максимизировать суммарную ценность, не превышая ограничение по весу.
- **Динамическое программирование:**
  - Подзадачи: Максимальная ценность для подмножеств предметов и ограничений по весу.
  - Рекуррентное соотношение: `dp[i, w] = max(dp[i-1, w], dp[i-1, w-w[i]] + v[i])`, если `w[i] <= w`, иначе `dp[i, w] = dp[i-1, w]`, где `dp[i, w]` - максимальная ценность для предметов от 1 до `i` и ограничения по весу `w`.

- Запоминание: Таблица `dp`.
- Построение решения: Восстановление набора предметов из таблицы `dp`.

```
function knapsack(W, w, v):
  // W - максимальный вес рюкзака
  // w - массив весов предметов
  // v - массив ценностей предметов
  n = length(w)

  // Инициализируем таблицу dp нулями
  dp = create_matrix(n + 1, W + 1, 0)

  // Заполняем таблицу dp
  for i = 1 to n:
    for w_cur = 1 to W:
      if w[i] <= w_cur:
        dp[i][w_cur] = max(dp[i-1][w_cur], dp[i-1][w_cur-w[i]] + v[i])
      else:
        dp[i][w_cur] = dp[i-1][w_cur]

  // Восстанавливаем набор предметов из таблицы dp
  return backtrack_knapsack(dp, w, n, W)

function backtrack_knapsack(dp, w, i, w_cur):
  // Рекурсивно восстанавливаем набор предметов
  if i == 0 or w_cur == 0:
    return []
  elseif dp[i][w_cur] == dp[i-1][w_cur]:
    return backtrack_knapsack(dp, w, i-1, w_cur)
  else:
    return backtrack_knapsack(dp, w, i-1, w_cur-w[i]) + [i]
```

10. Жадные алгоритмы

Жадные алгоритмы (greedy algorithms) - это алгоритмы, которые на каждом шаге делают локально оптимальный выбор в надежде, что это приведет к глобально оптимальному решению. Они часто просты в реализации и эффективны, но не всегда гарантируют нахождение оптимального решения.

Признаки применимости

1. **Жадный выбор:** На каждом шаге можно сделать локально оптимальный выбор, не учитывая будущие последствия.
2. **Оптимальная подструктура:** Оптимальное решение задачи содержит оптимальные решения её подзадач.

Важно: Не все задачи, обладающие свойством оптимальной подструктуры, могут быть решены жадными алгоритмами.

Задача о рюкзаке (непрерывный вариант)

В непрерывном варианте задачи о рюкзаке (Fractional Knapsack Problem) можно брать части предметов.

Задача: Имеется рюкзак с максимальной грузоподъемностью `W` и набор предметов, каждый из которых имеет вес `w[i]` и ценность `v[i]`. Нужно выбрать предметы (или их части) так, чтобы максимизировать суммарную ценность, не превышая ограничение по весу.

Жадный алгоритм:

1. **Вычисление удельной ценности:** Для каждого предмета вычисляем удельную ценность `v[i] / w[i]`.
2. **Сортировка:** Сортируем предметы по убыванию удельной ценности.
3. **Заполнение рюкзака:**
   - Начинаем с предмета с наибольшей удельной ценностью.
   - Берем весь предмет, если его вес не превышает оставшуюся грузоподъемность рюкзака.
   - Иначе, берем часть предмета, заполняющую оставшуюся грузоподъемность.

Псевдокод:

```
function fractional_knapsack(W, w, v):
  // W - максимальный вес рюкзака
  // w - массив весов предметов
  // v - массив ценностей предметов
  n = length(w)

  // Вычисляем удельные ценности
  specific_values = [v[i] / w[i] for i in range(n)]

  // Сортируем предметы по убыванию удельной ценности
  sorted_items = sort_by_descending(specific_values)

  total_value = 0
```

```
  remaining_weight = W

  for i in sorted_items:
    if w[i] <= remaining_weight:
      // Берем весь предмет
      total_value += v[i]
      remaining_weight -= w[i]
    else:
      // Берем часть предмета
      fraction = remaining_weight / w[i]
      total_value += fraction * v[i]
      remaining_weight = 0
      break  // Рюкзак заполнен

  return total_value
```

Гарантия оптимальности:

Жадный алгоритм для непрерывного варианта задачи о рюкзаке **гарантированно находит оптимальное решение**.

Важно: Жадный алгоритм **не работает** для дискретного варианта задачи о рюкзаке, где нельзя брать части предметов.

11. Алгоритмы поиска подстрок

Алгоритмы поиска подстрок (string matching algorithms) — это алгоритмы, предназначенные для поиска вхождений одной строки (образца, pattern) в другую строку (текст, text). Они широко используются в текстовых редакторах, поисковых системах, биоинформатике и других областях.

Задача: найти все вхождения подстроки в строку.

- $T[1 \dots n]$ – текст, в котором ищем
- $P[1 \dots m]$ – pattern
- $s \in [0 \dots n - m]$ – shift
- Если $T[s + 1 \dots s + m] = P[1 \dots m] \Rightarrow$ сдвиг допустимый. Это означает, что найдено вхождение подстроки со сдвигом s.

Задача: для заданного образца найти все допустимые сдвиги.

«Наивный» алгоритм

"Наивный" алгоритм (naive algorithm) — это простейший алгоритм поиска подстрок, который заключается в последовательном сравнении образца с каждым возможным подстрокой текста.

Идея:

1. Для каждого индекса `i` в тексте:
   - Сравниваем образец с подстрокой текста, начинающейся с `i`.
   - Если все символы совпадают, то найдено вхождение образца.

Сложность: $O(m \cdot n)$, где `m` - длина образца, `n` - длина текста.

Псевдокод:

```
Naive_Matcher(T, P):
    for s = 0 to T.length - P.length
        valid = true
        for i = 1 to P.length
            if T[s + i] != P[i]
                valid = false
                break
        if vaild
            print("Допустимый сдвиг:", s)
```

Алгоритм Рабина-Карпа

Алгоритм Рабина-Карпа (Rabin-Karp algorithm) использует **хеширование** для ускорения поиска подстрок.

Идея:

1. Вычисляем хэш образца.
2. Для каждого индекса `i` в тексте:
   - Вычисляем хэш подстроки текста длины `m`, начинающейся с `i`.

- Если хэши совпадают, сравниваем образец и подстроку текста посимвольно, чтобы убедиться, что это не ложное срабатывание (hash collision).

Пример. $T = 1271326126126$, $P = 126$. Вариант решения: перевести число символов длины `P.length` в число, сам паттерн тоже в число. Сравнить. Сдвинуться на один вправо. Подразумеваем, что код символа = его значение.

```
    t_{i-1}
     v...v
|1|2|7|1|3|2| ...
      |^...^ - t_i
shift>|
```

$t_i = (((t_{i-1} - T[i] \cdot d^{m-1} \mod q) \mod q) \cdot d + T[i+m]) \mod q$, d^m считаем заранее.

Получается вычисление $t_i \approx O(1)$

$d = 10$ - если только с цифрами

$d = 256$ - если работаем с кодами символов

Тогда получаем алгоритм:

```
RK(T, P):
    q = 1; t = 0; p = 0
    for i = 1 to m:
        q = q * d
        t = (d * t + T[i]) % q
        p = (d * p + P[i]) % q
    for s = 0 to n - m:
        if t == p:
            print("Сдвиг", s)
        if s < n - m:
            t = (((t - T[s + 1] * q) % q) * d + T[s + m + 1]) % q
```

`q`: Переменная, хранящая текущую степень `d` (начинает с 1 и увеличивается в степени `d` на каждой итерации первого цикла). Используется для "сдвига" хэша влево.

`t`: Хэш текущей подстроки текста `T` длины `m`.

`p`: Хэш образца `P`.

`d`: Основание системы счисления, используемое для хэширования.

Возможно возникновение коллизии, в таком случае, если совпадает образец по коду, то сделать посимвольную проверку.

Оценка стоимости $\sim O(n) + O(m)$

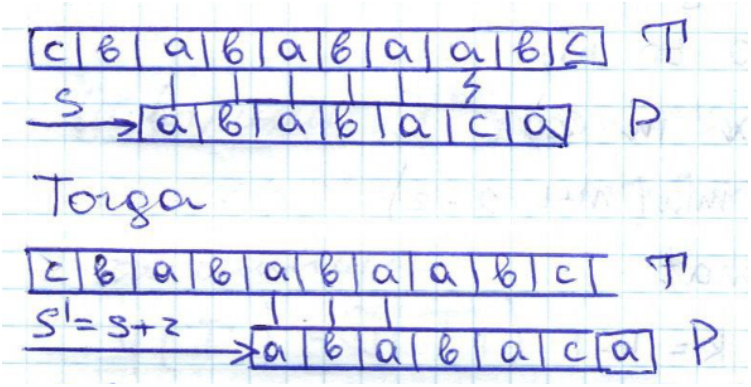
Алгоритм Кнута-Морриса-Пратта (KMP)

Алгоритм Кнута-Морриса-Пратта (Knuth-Morris-Pratt algorithm) использует **префикс-функцию** для оптимизации поиска подстрок.

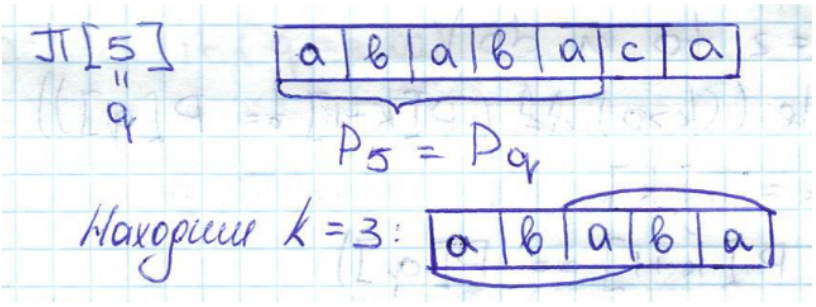
Идея:

1. Вычисляем префикс-функцию для образца. Префикс-функция `pi[i]` для индекса `i` в образце - это длина наибольшего собственного суффикса подстроки `pattern[0..i]`, который также является префиксом образца.

2. Используем префикс-функцию для сдвига образца при несовпадении символов, избегая повторных сравнений уже проверенных символов.



Сдвигаем паттерн так, чтобы рассматриваемая часть сразу не конфликтовала с паттерном. Пример нахождения префикса в префикс-функции:



Обязательно $k < q$.

```
Compute_Prefix_Function(P):
    m = P.length
    pi[1...m]
    pi[1] = 0
    k = 0
    for q = 2 to m do:
        while (k > 0) and (P[k + 1] != P[q]):
            k = pi[k]
        if P[k + 1] == P[q]:
            k++
        pi[q] = k
    return pi

KMP_Matcher(T, P):
    n = T.length; m = P.length
    pi = Compute_Prefix_Function(P)
    q = 0
    for i = 1 to n do:
        while (q > 0) and (P[q + 1] != T[i]):
            q = pi[q]
        if (P[q + 1] == T[i]):
            q++
        if q == m:
            print("Образец со сдвигом", i - m)
        q = pi[q]
```

Сложность: $O(m + n)$. У алгоритма всегда линейная оценка, даже в плохом случае $T = a^n,\ P = a^m$.

Алгоритм Бойера-Мура (Boyer-Moore)

Алгоритм Бойера-Мура (Boyer-Moore algorithm) использует **два эвристических правила** для ускорения поиска, при этом сравниваем с конца образца и начала фрагмента текста:

1. **Правило плохого символа (bad character rule):** При несовпадении символов сдвигаем образец так, чтобы последний символ образца совпал с соответствующим символом в тексте. Если выдается отрицательный сдвиг, то игнорируем рекомендацию этой эвристики.
2. **Правило хорошего суффикса (good suffix rule):** При совпадении суффикса образца с подстрокой текста сдвигаем образец так, чтобы этот суффикс совпал с предыдущим вхождением этого суффикса в образце. Всегда дает положительный сдвиг.

Общая идея алгоритма:

- Выбираем наибольший сдвиг из двух эвристик.
- Просматриваем текст с начала, паттерн с конца.

Примечание. Алгоритм хорош для больших последовательностей.

Сложность: $O(n/m)$ в лучшем случае (при длинных образцах), $O(m \cdot n)$ в худшем случае.

```
Boyer_Moore_Matcher(T, P):
    n = длина(T)  // Длина текста
    m = длина(P)  // Длина паттерна
    occurrences = []  // Список индексов вхождений

    // Предварительное вычисление таблиц для эвристик
    bad_char = Bad_Character_Heuristic(P)
    good_suffix = Good_Suffix_Heuristic(P)

    s = 1  // Начальный сдвиг паттерна

    while s <= n - m:
        j = m  // Индекс в паттерне, начинаем с конца

        // Сравниваем символы паттерна и текста справа налево
        while j > 0 and P[j] == T[s + j]:
            j--

        if j == 0:  // Паттерн найден
            добавить(occurrences, s)  // Добавляем индекс вхождения в список
            s = s + good_suffix[0]  // Сдвигаем на величину хорошего суффикса
        else:
            // Считаем сдвиги по эвристикам
            bad_char_shift = bad_char.получить(T[s + j - 1], m)  // Сдвиг по плохому символу
            good_suffix_shift = good_suffix[j + 1]  // Сдвиг по хорошему суффиксу

            // Выбираем максимальный сдвиг
            s = s + max(bad_char_shift, good_suffix_shift)

    return occurrences

```

```
32
33  Bad_Character_Heuristic(P):
34      m = длина(P)
35      bad_char = словарь() // Создаем пустой словарь
36
37      // Заполняем словарь значениями по умолчанию
38      для i от 1 до m:
39          bad_char[P[i]] = m
40
41      // Заполняем словарь фактическими сдвигами
42      для i от 1 до m - 1:
43          bad_char[P[i]] = m - i
44
45      return bad_char
46
47
48  Good_Suffix_Heuristic(P):
49      m = длина(P)
50      good_suffix = массив(m + 1)  // Создаем массив размером m + 1
51      f = массив(m + 1)  // Вспомогательный массив
52
53      // Инициализация массивов
54      f[m] = m + 1
55      i = m
56      j = m + 1
57
58      while i > 0:
59          while j <= m and P[i] != P[j]:
60              if good_suffix[j] == m:
61                  good_suffix[j] = j - i
62              j = f[j]
63          i = i - 1
64          j = j - 1
65          f[i] = j
66
67      j = f[0]
68      for i от 1 до m + 1:
69          if good_suffix[i] == m:
70              good_suffix[i] = j
71          if i == j:
72              j = f[j]
73
74      return good_suffix
```

Поиск подстрок с помощью конечных автоматов

Для поиска подстрок можно построить конечный автомат, который принимает текст на вход и переходит в состояние "найдено", если текущая подстрока текста совпадает с образцом.

Конечный автомат M – это пятерка $(Q, q_o, A, \Sigma, \delta)$, где:

- Q – множество состояний
- $q_o \in Q$ – начальное состояние
- $A \subseteq Q$ – конечное множество допустимых состояний
- Σ – входной алфавит
- $\delta : Q \times \Sigma \to Q$ – функция переходов

Для каждого паттерна строится свой конечный автомат.

Идея:

1. Строим конечный автомат для образца.
2. Пропускаем текст через автомат, начиная с начального состояния.
3. Если автомат переходит в допускающее состояние, то найдено вхождение образца.

Суффикс-функция $\sigma(x) = max\{k : P_k \supseteq x\}$ — максимально большое количество совпадений символов конца слова с началом паттерна.

```
| | | | |#|#|#| x
        |#|#|#| | | | | | p
              ^-k
```

$\sigma(\epsilon) = 0$

Функция переходов: $\delta(q, a) = \sigma(P_q a)$

P_q — префикс паттерна размера q.

```
      [     ]
| | |c|a|b|a|a|             T
      |a|b|a|b| | | | | |    p
      [ P_3 ]
```

$\delta(3, y) = \sigma(P_3 y) = \sigma(”aba”, ”a”) = 1$

Инвариант функции показывает конечное состояние автомата после прочтения i-го символа.

$\varphi(T_i) = \sigma(T_i)$

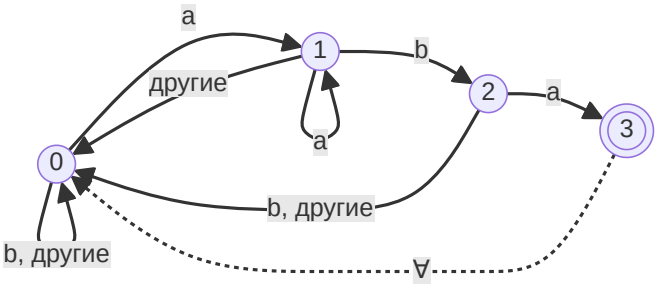
Алгоритм.

```
Finite_Automation_Matcher(T, P):
    delta = Compute_Transition_Function(P, sigma)
    n = T.length; m = P.length
    q = 0
    for i = 1 to n:
        q = delta(q, T[i])
        if q == m:
            print("Образец со сдвигом", i-m)

Compute_Transition_Matcher(P, sigma):
    m = P.Length
    for q = 0 to m:
        for a in sigma:
            k = min(m + 1, q + 2)
            repeat
                k--
            until Pk in Pqa
            delta(q, a) = k
    return q
```

Оценка $O(m\Sigma + n)$

Пример автомата. Конечный автомат для поиска подстроки "aba"

Состояние \ Вход	a	b	Другие
0	1	0	0
1	1	2	0
2	3	0	0
3	1	0	0



Пример работы автомата:

Входная строка: "aababababaa"

Шаг	Вход	Текущее состояние	Следующее состояние	Подстрока найдена?
1	a	0	1	Нет
2	a	1	1	Нет
3	b	1	2	Нет
4	a	2	3	Да
5	b	3	0	Нет
6	a	0	1	Нет
7	b	1	2	Нет
8	a	2	3	Да

Вывод: Подстрока "aba" найдена дважды, начиная с позиций 3 и 8.

ДОПОЛНИТЕЛЬНО (не из программы, об этом говорилось на консультациях)

Биномиальные деревья

Биномиальное дерево - это рекурсивно определяемая структура данных, используемая для реализации **биномиальных куч**.

Определение:

- **Биномиальное дерево порядка** 0 состоит из одного узла.
- **Биномиальное дерево порядка** k состоит из двух биномиальных деревьев порядка $k-1$, где корень одного дерева является левым потомком корня другого дерева.

Свойства:

- Биномиальное дерево порядка k имеет 2^k **узлов**.
- **Высота** биномиального дерева порядка k равна k.
- В биномиальном дереве порядка k **корень имеет** k **потомков**, каждый из которых является корнем биномиального дерева порядка $k-1, k-2, \ldots, 0$.

Биномиальные кучи

Биномиальная куча - это структура данных, представляющая собой **набор биномиальных деревьев**, которые удовлетворяют **свойству кучи** (min-куча или max-куча) и **свойству биномиальной кучи**.

Свойства:

- **Свойство кучи:** Для каждого биномиального дерева в куче выполняется свойство min-кучи или max-кучи (ключ каждого узла меньше или равен ключам его потомков для min-кучи, и больше или равен для max-кучи).
- **Свойство биномиальной кучи:** В биномиальной куче **не может быть двух биномиальных деревьев одного порядка**. Кроме того, биномиальные деревья в куче упорядочены по возрастанию порядка.

Операции:

- `MakeHeap()`**:** Создает пустую биномиальную кучу.
- `Insert(x)`**:** Вставляет новый элемент `x` в кучу.
  Создать новое биномиальное дерево, состоящее из одного узла, содержащего добавляемый элемент. Объединить биномиальную кучу с новым деревом. Если после объединения в куче появились два дерева одинаковой степени, выполните их слияние. Повторять, пока не останется деревьев одинаковой степени.
- `Minimum()` **(для min-кучи) /** `Maximum()` **(для max-кучи):** Возвращает элемент с минимальным (максимальным) ключом.
- `ExtractMin()` **(для min-кучи) /** `ExtractMax()` **(для max-кучи):** Извлекает и возвращает элемент с минимальным (максимальным) ключом.
- `Union(H1, H2)`**:** Объединяет две биномиальные кучи `H1` и `H2` в одну.

- **`DecreaseKey(x, k)` (для min-кучи) / `IncreaseKey(x, k)` (для max-кучи):** Уменьшает (увеличивает) значение ключа элемента `x` до `k`.
- **`Delete(x)`:** Удаляет элемент `x` из кучи.

Преимущества биномиальных куч:

- Операция `Union` выполняется за время $O(\log n)$, где n - общее количество элементов в кучах.
- Остальные операции также выполняются за логарифмическое время.

Фибоначчиевы кучи

Фибоначчиевы кучи (Fibonacci heaps) — это сложная, но очень эффективная структура данных для реализации **очередей с приоритетами**. Они обеспечивают амортизированно константное время для большинства операций, что делает их привлекательными для алгоритмов, где интенсивно используются операции с очередями с приоритетами, например, алгоритм Дейкстры для поиска кратчайших путей.

Структура

- **Коллекция деревьев:** Фибоначчиева куча представляет собой набор **неупорядоченных** деревьев, каждое из которых удовлетворяет **свойству min-кучи** (или max-кучи).
- **Указатель на минимум (максимум):** Куча хранит указатель на корень дерева с минимальным (максимальным) ключом.
- **Связанный список корней:** Корни деревьев организованы в **двусвязный циклический список**.
- **Отметки узлов:** Каждый узел имеет **отметку** (mark), которая указывает, был ли у него удален потомок с момента последнего подключения к другому дереву.

Ключевые особенности

- **Ленивые операции:** Фибоначчиевы кучи откладывают "тяжелую" работу до тех пор, пока это возможно, что позволяет амортизировать затраты на операции.
- **Объединение деревьев:** Операция `Union` выполняется просто путем объединения списков корней двух куч.
- **Отложенное восстановление структуры:** Структура кучи восстанавливается только при извлечении минимального (максимального) элемента.

Операции

- **`MakeHeap()`:** Создает пустую фибоначчиеву кучу.
- **`Insert(x)`:** Вставляет новый элемент `x` в кучу, создавая новое дерево с одним узлом. Чтобы добавить элемент в фибоначчиеву кучу, нужно:
  1. **Создать новый узел** с этим элементом.
  2. **Добавить новый узел** в список корней кучи, сохраняя порядок по возрастанию степени.
  3. **Объединить деревья** одинаковой степени в списке корней, чтобы у каждого корня была уникальная степень.
  4. **Обновить указатель на минимальный элемент**.

- **`Minimum()` (для min-кучи) / `Maximum()` (для max-кучи):** Возвращает элемент с минимальным (максимальным) ключом, используя указатель на минимум (максимум).
- **`ExtractMin()` (для min-кучи) / `ExtractMax()` (для max-кучи):** Извлекает и возвращает элемент с минимальным (максимальным) ключом, выполняя "тяжелую" работу по восстановлению структуры кучи.
- **`Union(H1, H2)`:** Объединяет две фибоначчиевы кучи `H1` и `H2` в одну, объединяя их списки корней.
- **`DecreaseKey(x, k)` (для min-кучи) / `IncreaseKey(x, k)` (для max-кучи):** Уменьшает (увеличивает) значение ключа элемента `x` до `k`, возможно, отсоединяя узел от его родителя и добавляя его в список корней.
- **`Delete(x)`:** Удаляет элемент `x` из кучи, используя `DecreaseKey` (или `IncreaseKey`) для уменьшения (увеличения) его ключа до $-\infty$ (или $+\infty$), а затем вызывая `ExtractMin` (или `ExtractMax`).

Сложность операций

Операция	Амортизированная сложность
`MakeHeap`, `Insert`, `Minimum`/`Maximum`, `Union`	$O(1)$
`ExtractMin`/`ExtractMax`, `Delete`	$O(\log n)$
`DecreaseKey`/`IncreaseKey`	$O(1)$

Преимущества

- **Очень быстрые операции:** Большинство операций выполняются за амортизированно константное время.
- **Эффективность в алгоритмах графов:** Фибоначчиевы кучи особенно эффективны в алгоритмах, где часто используются операции `DecreaseKey` (или `IncreaseKey`), например, алгоритм Дейкстры.

Возможные задания

Считай, ответы.

Тестовая часть

1. Верные утверждения:
   $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
   $f(n) = \Theta(g(n)) \Rightarrow f(n) \neq o(g(n))$
   $f(n) = \Theta(g(n)) \Rightarrow f(n) \neq \omega(g(n))$
   $f(n) = \omega(g(n)) \Rightarrow f(n) \neq \Theta(g(n))$
2. 2.1 Асимптотика какого алгоритма зависит от размера алфавита - **Поиск подстрок с**помощью автоматов**
   2.2 Какой из приведенных алгоритмов работает лучше в худшем случае - **Боуера-Мура**
   2.3 Асимптотика какого алгоритма зависит от размера машинного слова - **Поиск подстрок с помощью автоматов, Рабина-Карпа**
   2.4 В алгоритме Кнута-Морриса-Пратта используется - **Префикс-функция для образца**
   2.5 Какой алгоритм использует эвристику безопасного суффикса? **Буера-Мура**
   2.6 Какой из приведённых алгоритмов начинает проверку с конца образца? **Бойера-Мура**
3. 3.1 Результатом алгоритма топологической сортировки является **упорядоченный набор вершин**
   3.2 Результатом алгоритма Крускала является **минимальное остовное дерево**
   3.3 Результатом алгоритма Прима является **минимальное остовное дерево**
   3.4 Результатом работы алгоритма Дейкстры **является дерево кратчайших путей**
4. 4.1 Операция вращения являются вспомогательные для **AVL-деревьев, RBT-деревьев**
   4.2 Какой вид деревьев имеет минимальную высоту: **Б-деревья**
   4.3 В стандартной библиотеке C++ для реализации контейнера std::map используются **красно-чёрные деревья**
5. 5.1 Рассмотрение отрицательного сдвига подстроки возможно в алгоритме **Боуера-Мура**
   5.2 Ситуация “холостое срабатывание” возможна в алгоритме **Рабина-Карпа**
   5.3 Что даст алгоритм топологической сортировки на графах с циклами? **Неверный ответ**
   5.4 Алгоритм топологической сортировки неприменим и даёт неверный ответ в случае, если **граф содержит циклы**
6. 6.1 Если основная теорема о рекуррентных соотношениях не применима - **Возможно, оценка может быть получена другими методами**
   6.2 Основная теорема о рекуррентных соотношениях - **Позволяет в некоторых случаях найти асимптотическую оценку рекуррентного соотношения**
   6.3 Что из приведенного используется для оценки рекуррентных соотношений? **Метод подстановки. Метод рекурсий. Метод итераций.**
7. 7.1 Эффективные операции для RBT: **Извлечение корня с образованием двух RBT, Извлечение минимального элемента**
   7.2 Операции, которые не используются для хеш-таблиц: **Объединение хеш-таблиц, извлечение минимального, изменение размера таблицы**
   7.3 Какие маркеры ячеек обычно реализуются в хеш-таблицах на открытой адресации? - **Элемент удален, Ячейка пуста**
   7.4 Чем деревья поиска отличаются от хеш-таблиц? **Поддержание порядка на элементах, Поддержкой постоянных итераторов**
   7.5 Методы std::lower_bound и std::upper_bound в C++ дают одинаковый результат в случае, **если элемент в контейнере отсутствует**

8. 8.1 Структура бинарной кучи соответствует структуре **RBT и AVL**
   8.2 Почему бинарная куча обычно не предоставляет итераторов на элементы?
   Это не имеет смысла — элементы не упорядочены, Положение элементов в памяти часто меняется

9. 9.1 Какие из приведенных структур можно рассматривать как набор деревьев?
   Биномиальные кучи, Фибоначчиевы кучи

10. 10.1 Какие операции поддерживаются биномиальной кучей? **Извлечение минимального эл-та, слияние 2-х куч**
    10.2 Какой ответ даст алгоритм топологической сортировки на графах с циклами?
    Гарантированно неверный ответ
    10.3 Какие операции поддерживаются фибоначчиевой кучей? **Извлечение минимального эл-та, слияние 2-х куч**
    10.4 Структура бинарной кучи соответствует структуре **идеально сбалансированного дерева**

11. 11.1 Будет ли работать алгоритм Прима на графах с петлями? **Да, с получением верного результата**
    11.2 Алгоритм Прима работает обычно **на взвешенных графах**
    11.3 Будет ли работать алгоритм Крускала на графах с петлями? **Да, с получением верного результата**
    11.4 Могут ли алгоритмы Крускала и Прима давать разные результаты на одном графе?
    Только при наличии рёбер с одинаковым весом

12. 12.1 Что из приведенного верно? (про деревья) **AVL-дерево можно представить в виде RBT-дерева(раскрасив вершины), RBT-дерево не всегда является AVL-деревом**
    12.2 Что из приведенного верно? (про жадные алгоритмы и динамику) **Применимость этих стратегий не связана напрямую**
    12.3 Алгоритм Прима работает обычно **на взвешенных графах**

13. 13.1 Критерии применимости динамического программирования: **Оптимальность по подзадачам, Повторяющиеся подзадачи**
    13.2 Может ли фибоначчиева куча иметь такую же структуру, что и биномиальная: **Да, если не выполнять объединение куч, Да, если не извлекать произвольные элементы**

14. 14.1 Какие эвристики применяются в системах непересекающихся множеств - **Объединение по рангу(весовая), Эвристика сжатия путей**
    14.2 Эвристика “сжатия путей” применяется в алгоритме – **Крускала**
    14.3 Какая структура данных “откладывает” восстановление нормального вида до завершения последовательности однотипных операций - **Фибоначчиевы кучи**
    14.3 Эвристика “по весу” применяется в алгоритме - Крускала

15. 15.1 В системах непересекающихся мн-в обычно не реализуется **пересечение мн-в, разбиение мн-в, извлечение минимального эл-та**
    Комментарий: *В системах непересекающихся множеств реализуется только объединение множеств и проверка на принадлежность (по элементу получаем представителя его множества, либо индекс), остальное не реализуется.*

    15.2 Укажите эффективный метод для решения задачи коммивояжере - **Динамическое программирование**

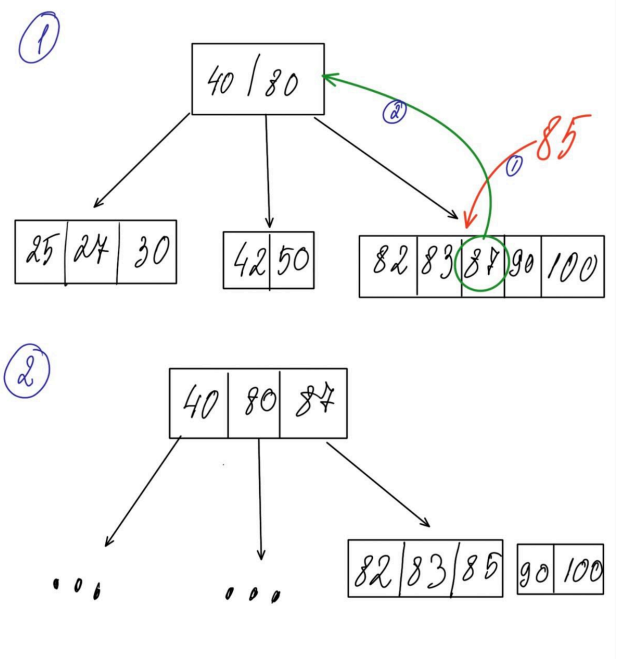
16. 16.1 Амортизированная стоимость операций является **усредненной оценкой стоимости операции для некоторого набора операций**
    16.2 Амортизированная стоимость операций является **оценкой усредненной стоимости набора операций**

Задания с развернутым ответом

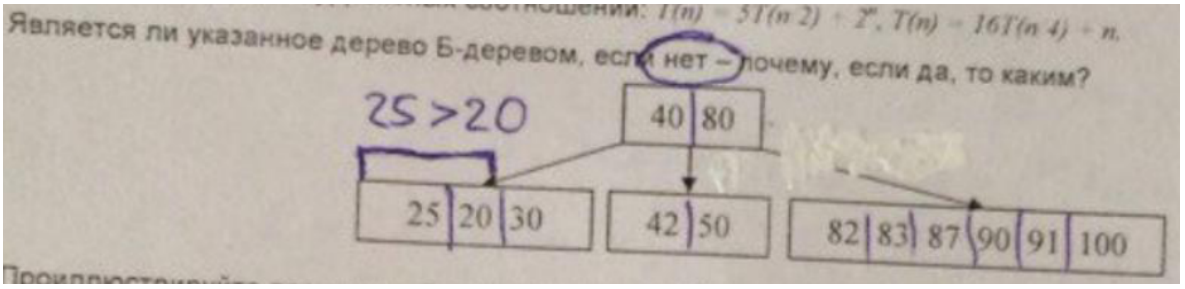
Ссылки на места в документе, где есть ответ на соответствующий вопрос.

17. 17.1 Дайте определение биноминальной кучи. Ссылка на ответ
    17.2 Идея метода ветвей и границ на примере задачи коммивояжера. Ссылка на ответ
    17.3 Укажите, какую операцию было бы неплохо добавить к очереди с приоритетами, чтобы в
    алгоритме Дейкстры не возникало дублирование элементов в очереди - **Обновление приоритета элемента**
18. 18.1 Укажите оценки эффективности основных операций для хеш-таблицы на основе открытой адресации - **Добавление, поиск и удаление $O(1/(1-\alpha))$, где α - коэф. заполненности таблицы**
    18.2 Укажите оценки эффективности основных операций для хеш-таблицы на основе цепочек
    Добавление, поиск и удаление $O(1+\alpha)$, где α - коэф. заполненности таблицы
    18.3 Какой эффект можно получить, если в хеш-таблице плохо подобрать хеш-функцию? **Коллизии, неравномерное распределение, снижение производительности, увеличение времени выполнения методов разрешения коллизий, увеличение использования памяти**
    18.4 Что такое потенциальная функция, требования и ограничения. Ссылка на ответ
19. 19.1 Алгоритм Прима и оценка времени работы. Ссылка на ответ
    19.2 Алгоритм Крускала и оценка времени работы. Ссылка на ответ
20. 20.1 Алгоритм Рабина-Карпа. Ссылка на ответ
    20.2 Алгоритм Кнута-Морриса-Пратта. Ссылка на ответ
21. 21.1 Префикс-функция. $\pi(q) = \max\{k : k < q, P_k \supseteq P_q\}$
    21.2 Суффикс-функция. $\sigma(x) = \max\{k : P_k \supseteq k\}$
22. Конечный автомат для поиска подстроки. Ссылка на ответ
23. Мастер-теорема. Ссылка
    1. $T(n) = 6T(n/3) + n^2log(n)$
       $a = 6, b = 3, f(n) = n^2\log(n)$
       Заметим, что $n^2\log n > n^{\log_b(a)}$ (растет быстрее). Однако,
       $\exists \epsilon > 0 : n^2\log n = O(n^{log_3 6+\varepsilon})$). Так как $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right)$, мы находимся в случае 1 мастер-теоремы. Следовательно, $T(n) = \Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{log_3(6)}\right)$.
    2. $T(n) = 4T(n/2) + n\log(n)$. Аналогично, 1 случай, ответ $\Theta\left(n^2\right)$.
    3. $T(n) = 5T(n/2) + 2^n$
       $a = 5, b = 2, f(n) = 2^n$
       Заметим, что $2^n > n^{log_2(5)}$ (растет быстрее). Также выполняется условие регулярности: $af(n/b) = 5 \cdot 2^{n/2} \leq c \cdot 2^n = cf(n)$ для $c = \frac{1}{2}$ и достаточно больших n. Так как $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ и выполняется условие регулярности, мы находимся в случае 3 мастер-теоремы. Следовательно, $T(n) = \Theta(f(n)) = \Theta(2^n)$.
    4. $T(n) = 16T(n/4) + n$
       $a = 16, b = 4, f(n) = n$
       Вычислим: $log_b(a) = log_4(16) = 2$
       В данном случае $f(n) = n = n^{\log_b(a)-1}$, где $\epsilon = 1 > 0$. Так как $f(n) = O(n^{\log_b(a)-\varepsilon})$, мы находимся в случае 1 мастер-теоремы. Следовательно,
       $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^2)$.
    5. ...

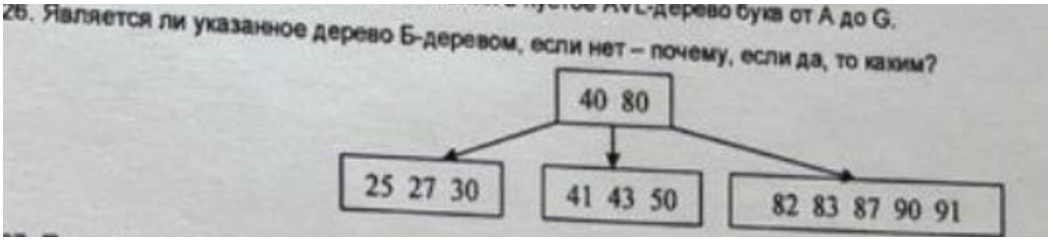
24. 24.1 Проиллюстрируйте добавление в указанное B-tree 85.



24.2

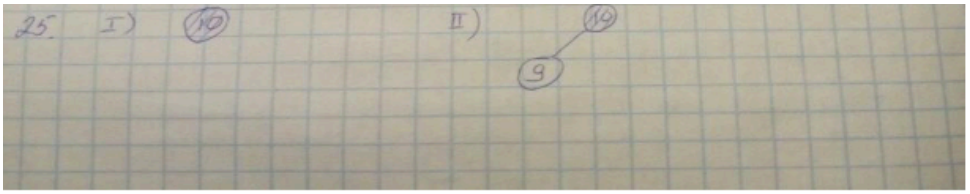


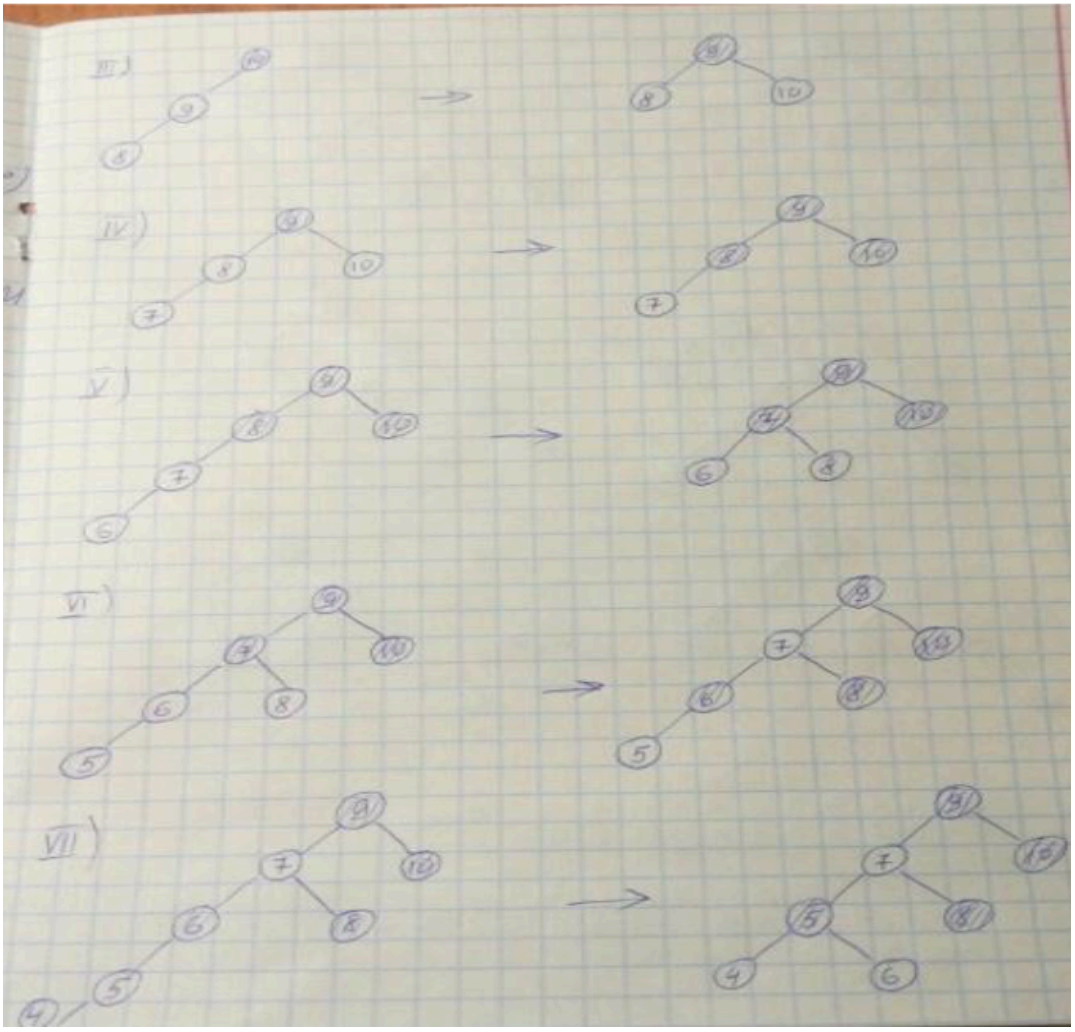
24.3



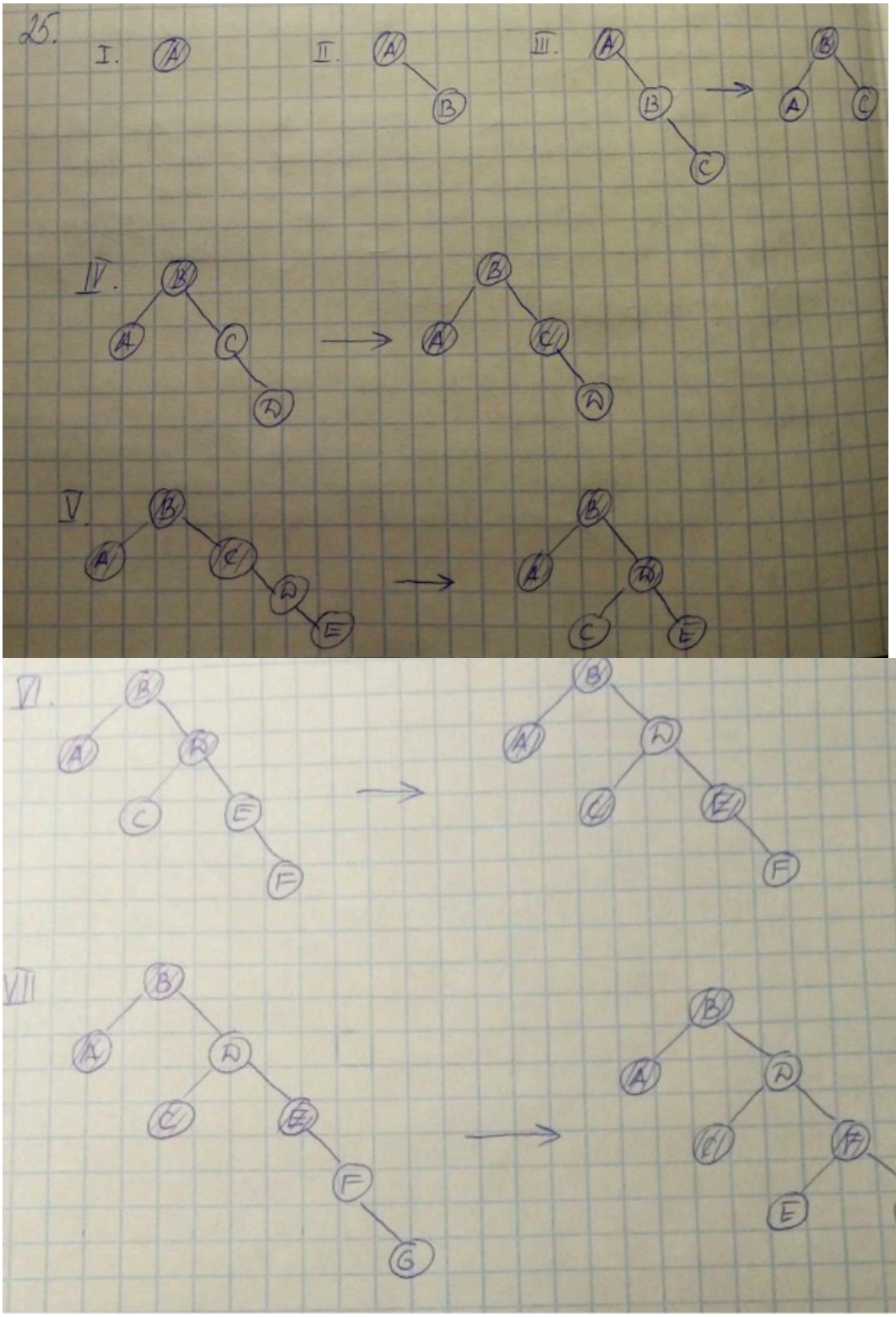
24.4 Дайте определение биноминального дерева. Ссылка на ответ

25. 25.1 Продемонстрируйте добавление в RBT-дерево чисел от 10 до 2.

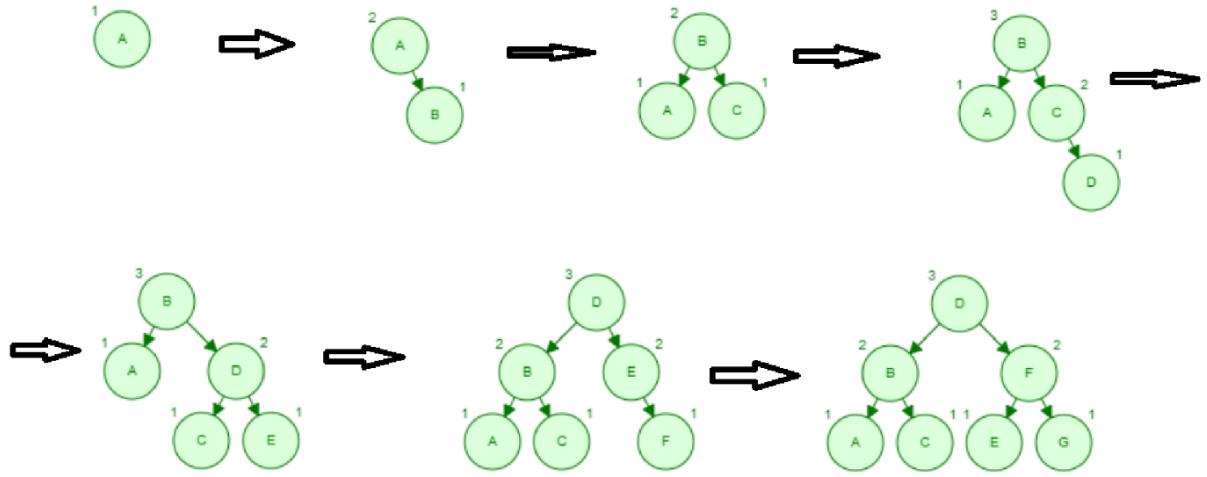




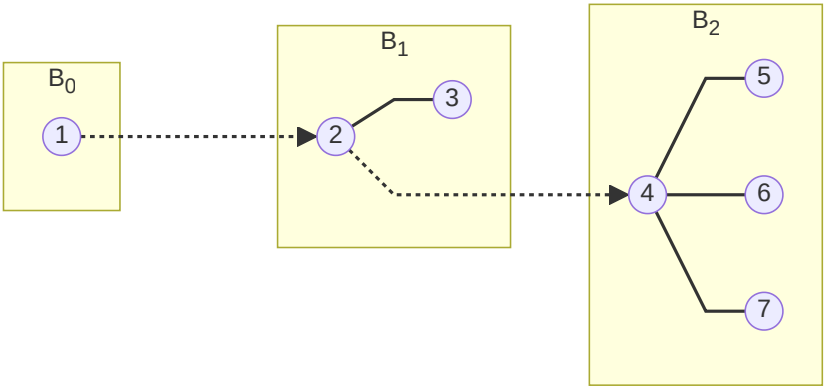
25.2 Продемонстрируйте добавление в RBT-дерево букв от A до G



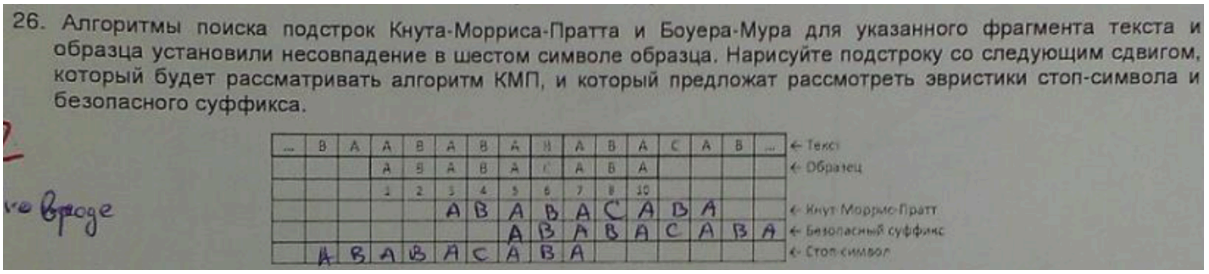
25.3 Продемонстрируйте добавление в AVL-дерево букв от A до G.



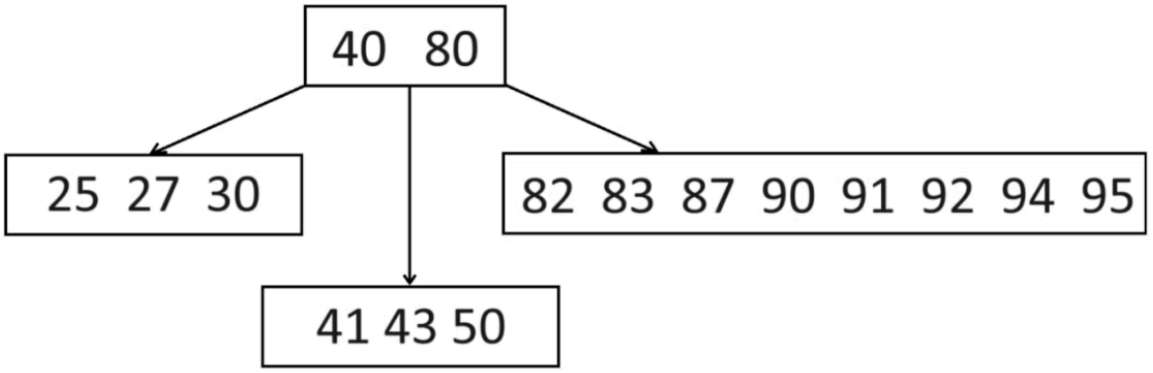
25.4 Нарисуйте биномиальную кучу для 7 элементов.



26. 26.1


26. Алгоритмы поиска подстрок Кнута-Морриса-Пратта и Боуера-Мура для указанного фрагмента текста и образца установили несовпадение в шестом символе образца. Нарисуйте подстроку со следующим сдвигом, который будет рассматривать алгоритм КМП, и который предложат рассмотреть эвристики стоп-символа и безопасного суффикса.

26.2 Является ли указанное дерево Б-деревом, если нет – почему, если да, то каким? (степень)



Да, является. Упорядоченным
3 - мин. число ключей, 8 - макс. число ключей
t-1 <= 3, 2t-1 >= 8
t <= 4, 2t >= 9
t >= 4
Следовательно, t=4

27. 27.1 Пример хеш-функции на основе умножения с квадратичной последовательностью проб.
$h(x,i) = (h'(x) + c_1 i + c_2 i) \mod M$
$h'(x) = (x \cdot A \mod 1) \cdot M$

    27.2 Пример хеш-функции на основе деления с квадратичной последовательностью проб
$h(x,i) = (h'(x) + c_1 i + c_2 i) \mod M$
$h'(x) = x \mod M$

    27.3 Почему эффективность хеш-таблицы на основе открытой адресации обычно ухудшается со временем и какая операция страдает больше всего?
    Из-за заполнения. С увеличением количества элементов увеличивается вероятность коллизий. Это увеличивает время работы операции поиска.

28. 28.1 Алгоритм добавления элемента в биномиальную кучу. Ссылка
    28.2 Алгоритм добавления элемента в бинарную кучу. Ссылка
    28.3 Алгоритм добавления элемента в фибоначчиеву кучу. Ссылка
    28.4 Какое отношение должно быть реализовано для элементов чтобы их можно было размещать в структуры наподобие std::set? **Реализовать < или использовать компаратор**

29. 29.1 Выполните амортизационный анализ для структуры “мультистек” (push,pop,multipop). См. что-нибудь отсюда
    29.2 Что такое «Фильтр Блума», для чего используется? Ссылка
    29.3 Какой тип ошибки НЕ может выдать фильтр Блума: «ложноположительное срабатывание» или «ложноотрицательное срабатывание»? **ложноотрицательное срабатывание**

30. 30.1 Возможны ли ситуации, в которых алгоритм с худшей асимптотической оценкой (в среднем) будет более предпочтителен, нежели алгоритм с лучшей асимптотикой? **Да возможны**

    30.2 Что такое ФЛОПСы (FLOPS), где и для чего они используются? **Это единица измерения, показывающая, сколько операций с плавающей запятой компьютер может выполнить за одну секунду. Используется в научных вычислениях, графике, играх, ИИ.**

31. 31.1 Динамическое программирование «сверху вниз» - основная идея и схема алгоритма.
    31.2 Что такое мемоизация?
    31.3 Динамическое программирование "снизу вверх" - основная схема и идея алгоритма.
    Ссылка на ответы к 31

32. 32.1 Укажите свойства хеш-функций, которые существенны для использования в хеш-таблицах. **Равномерность, Быстродействие, Детерминированность, Минимизация коллизий, Скорость вычисления**

    32.2 Что означает запись $f(n) = \Omega(g(n))$? **Это говорит о том, что $f(n)$ растет быстрее, чем $g(n)$, начиная с некоторого значения n_0. Дополнительно.**

❗ Комментарий