1. Понятие о СД. Уровни представления СД. Классификация СД в программах пользователя.

Структура данных (СД) – это общее свойство информационного объекта, с которым взаимодействует та или иная программа. Это общее свойство характеризуется:

- множеством допустимых значений данной структуры;
- набором допустимых операций;
- характером организованности.

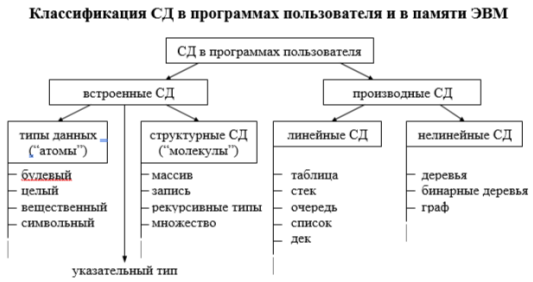
Вырожденные (простейшие) структуры данных называются также типами данных.

Различают следующие уровни описания данных:

- абстрактный (математический) уровень;
- логический уровень;
- физический уровень.

Классификация структур данных (СД) в программах пользователя и в памяти ЭВМ включает следующие виды:

- **Встроенные СД**. Например, булевый, массив, таблица, деревья, целый, запись, стек, бинарные деревья, вещественный.
- **Производные СД**. К ним относятся рекурсивные типы, очередь, граф, символьное множество, список, дек, указательный тип.
- **Типы данных («атомы»)**. К ним относятся множество, последовательность, матрица, дерево, граф, гиперграф.
- **Структурные СД («молекулы»)**. К ним относятся, например, рекурсивные типы, очередь, граф, символьное множество, список, дек.
- **Линейные СД**. Например, линейный односвязный список, циклический линейный список, двусвязный линейный список, дек.
- **Нелинейные СД**. К ним относятся, например, дерево, граф.



2. СД типа «массив». Вычисление адреса для многомерного массива.

Массив – последовательность элементов одного типа, называемого базовым. На математическом языке массив – это функция с ограниченной областью определения. Структура массивов однородна. Для выделения отдельной компоненты массива используется индекс. Индекс – это значение специального типа, определенного как тип

индекса данного массива. Поэтому на логическом уровне СД типа массив можно записать следующим образом:
type A = array [T1] of T2, где T1 – базовый тип массива, T2 – тип индекса.
Набор допустимых операций для СД типа массив: Операция доступа (доступ к элементам массива – прямой; от размера структуры операция не зависит).
Операция присваивания.
Операция инициализации (определение начальных условий).
Пусть у нас есть массив размером $m \times n$ (двумерный массив), и он хранится в памяти "построчно". Для элемента массива с индексами (i, j) адрес вычисляется по формуле:

$$\text{addr}(i, j) = \text{base} + (i \cdot n + j) \cdot \text{размер\_элемента} \tag{1}$$

Здесь:

- `base` — базовый адрес массива в памяти;
- i, j — индексы строки и столбца (индексация начинается с 0);
- n — количество столбцов в массиве;
- `размер_элемента` — размер одного элемента массива в байтах.

Для многомерного массива размерностью k, например $A[d_1][d_2]\ldots[d_k]$, формула обобщается:

$$\begin{aligned}\text{addr}(i_1, i_2, \ldots, i_k) =& \text{base} + ((i_1 \cdot d_2 \cdot \ldots \cdot d_k) + (i_2 \cdot d_3 \cdot \ldots \cdot d_k) + \ldots + i_k) \\ & \cdot \text{размер\_элемента}\end{aligned} \tag{2}$$

Здесь:

- $i_1, i_2, \ldots, i_k$ — индексы элемента массива;
- $d_1, d_2, \ldots, d_k$ — размеры массива по каждой из осей.

Эта формула позволяет вычислять адрес пункта массива независимо от его размерности.

3. СД типа «запись». Прямое декартово произведение. Дескриптор записи.
Запись – последовательность элементов, которые в общем случае могут быть одного типа. На логическом уровне СД типа запись можно записать следующим образом:
type T = Record
S1: T1;
S2: T2;
. . . . . .
Sn: Tn;
End;
Здесь: T_i изменяется при $i = 1, 2, \ldots, n$; $S_1, \ldots, S_n$ – идентификаторы полей; $T_1, \ldots, T_n$ – типы данных. Если T_i также является в свою очередь записью, то S_i – иерархическая запись.

Если DT_1 – множество значений элементов типа T_1, D_{T2} – множество значений элементов типа $T_2, \ldots$, $D_T n$ – множество значений элементов типа T_n, то D_T – множество значений элементов типа T будет определяться с помощью прямого декартова произведения:

$$D_T = D_{T1} \times D_{T2} \times \cdots \times D_{Tn}.$$

Другими словами, множество допустимых значений СД типа запись: $Car(T) = \prod_{i=1}^{n} Car(T_i)$
Дескриптор СД типа запись включает в себя: условное обозначение, название структуры, количество полей, указатель на первый элемент (в случае прямоугольной СД), характеристики каждого элемента, условные обозначения типа каждого элемента, размер слота, а также смещение, необходимое для вычисления адреса.

4. СД типа «таблица». Классификация операций.

Таблица –последовательность записей, которые имеют одну и ту же организацию. Такая отдельная запись называется элементом таблицы. Чаще всего используется простая запись. Таким образом, таблица – это агрегация элементов. Если последовательность записей упорядочена относительно какого-либо признака, то такая таблица называется упорядоченной, иначе – таблица неупорядоченная.
Классификация структур данных типа таблица:

- **Неупорядоченная таблица.** Это (возможно пустой) неупорядоченный набор элементов, каждый из которых представляет собой пару «ключ — значение». В каждый момент времени все ключи в таблице уникальны (не повторяются).
- **Упорядоченная таблица.** Если последовательность записей упорядочена относительно какого-либо признака, то такая таблица называется упорядоченной, иначе — неупорядоченной.
- **Хеш-таблица.** Это неупорядоченная коллекция пар «ключ — значение», где каждый ключ уникален. Ключи «хешируются» и позволяют эффективно работать с памятью — добавлять, получать, изменять и удалять значения.
- **Таблица как отображение на массив.** При реализации таблицы как отображения на массив её описание выглядит так: `Tabl = array [0..N] of Element.`
- **Таблица как отображение на список.** В этом случае таблица представляет собой набор из однотипных данных, линейно связанных друг с другом посредством ссылок (указателей).

Перед тем как определить операции, которые можно выполнять над таблицей рассмотрим классификацию операций:

- **Конструкторы** – операции, которые создают объекты рассматриваемой структуры.
- **Деструкторы** – операции, которые разрушают объекты рассматриваемой структуры. Признаком этой операции является освобождение памяти.
- **Модификаторы** – операции, которые модифицируют соответствующие структуры объектов. К ним относятся динамические и полустатические модификаторы.
- **Наблюдатели** – операции, у которых в качестве элемента (входного параметра) используются объекты соответствующей структуры, а возвращают эти операции результаты другого типа. Таким образом, операции-наблюдатели не изменяют структуру, а лишь дают информацию о ней.
- **Итераторы** – операторы доступа к содержимому объекта по частям в определенном порядке.

5. Временная сложность алгоритмов. Порядок функции временной сложности. Их определение в алгоритмах поиска.

Временная сложность (ВС) – зависимость времени выполнения алгоритма от количества обрабатываемых входных данных. Здесь представляет интерес среднее и худшее время выполнения алгоритма. ВС можно установить с различной точностью. Наиболее точной оценкой является аналитическое выражение для функции: t=t(N), где t – время, N – количество входных данных (размерность). Данная функция называется функцией временной сложности (ФВС).

Порядок функции временной сложности описывает приблизительное поведение функции сложности алгоритма при большом размере входа.

Некоторые общепринятые классификации порядка величины для нотации «О большое» (от наилучшего до наименее эффективного):

- **Постоянное время** ($O(1)$). Алгоритм выполняется за постоянное время, когда ему требуется одно и то же количество шагов вне зависимости от объёма задачи.
- **Логарифмическое время** ($O(\log n)$). Размер входных данных уменьшается в два раза, например, при итерации или обработке рекурсии.
- **Линейное время** ($O(n)$). В алгоритме есть один цикл.
- **Квадратичное время** ($O(n^2)$). В алгоритме есть вложенные циклы, то есть цикл в цикле.
- **Кубическое время** ($O(n^3)$). Алгоритм выполняется за кубическое время, когда производительность прямо пропорциональна размеру задачи в кубе.
- **Экспоненциальное время** ($O(2^n)$). Скорость роста удваивается при каждом добавлении входных данных.

Алгоритм линейного поиска используется для поиска ключа в массиве, как упорядоченном, так и неупорядоченном. Он заключается в последовательном сравнении каждого элемента массива с искомым ключом.

Худший случай: В худшем случае, когда искомый ключ находится в конце массива или отсутствует, алгоритм просматривает все N элементов. Таким образом, временная сложность в худшем случае составляет:

$$t_{\text{л.п.}} = k_1 \cdot N = O(N)$$

где k_1 – константа, представляющая время сравнения одного элемента.

Средний случай:

Пусть P_i – вероятность того, что искомый ключ равен k_i. Предположим, что $\sum_{i=1}^{N} P_i = 1$, т.е. ключ гарантированно находится в массиве. Среднее время поиска пропорционально среднему числу сравнений:

$$\bar{C} = \sum_{i=1}^{N} i \cdot P_i$$

Если все ключи равновероятны, то $P_1 = P_2 = \cdots = P_N = P = \frac{1}{N}$. Тогда:

$$\bar{C} = \sum_{i=1}^{N} i \cdot \frac{1}{N} = \frac{1}{N} \sum_{i=1}^{N} i = \frac{1}{N} \frac{N(N+1)}{2} = \frac{N+1}{2}$$

Таким образом, в среднем линейный поиск требует просмотра половины массива:

$$\bar{t}_{\text{л.п.}} \sim \bar{C} = \frac{N+1}{2} = O(N)$$

Упорядоченный массив с частотами обращений:

Если элементы в массиве упорядочены по частоте обращений (вероятности), то худшее время остается $O(N)$.

Для распределения по закону Зипфа, где $P_i = \frac{c}{i}$, среднее время поиска составляет:

$$\bar{C} \sim \sum_{i=1}^{N} i\frac{c}{i} = c\sum_{i=1}^{N} 1 = cN$$

где

$$c = \frac{1}{\sum_{i=1}^{N} \frac{1}{i}}$$

Используя приближение гармонического ряда:

$$\sum_{i=1}^{N} \frac{1}{i} \approx \ln(N) + \gamma$$

где γ - постоянная Эйлера. При $N \to \infty$,

$$c \approx \frac{1}{\ln N + \gamma} \approx \frac{1}{\ln N}$$

Следовательно,

$$\bar{t}_{\text{л.п.}} \sim cN \sim \frac{N}{\ln N}$$

Алгоритм бинарного поиска работает следующим образом:

1. Определить середину массива.
2. Если ключ в середине массива совпадает с искомым, поиск завершен.
3. Если ключ в середине массива больше искомого, применить бинарный поиск к первой половине массива.
4. Если ключ в середине массива меньше искомого, применить бинарный поиск ко второй половине массива.
5. Повторять шаги 1-4, пока область поиска не станет пустой.

Временная сложность: На каждом шаге алгоритма область поиска уменьшается вдвое. Следовательно, количество шагов до достижения области поиска в один элемент не превышает $\log_2 N$. Таким образом, временная сложность бинарного поиска составляет:

$$t_{\text{б.п.}} = O(\log_2 N)$$

6. Основные факторы выбора алгоритмов сортировки. Базовые сортировки.

Сортировка – это процесс перестановки элементов данного множества в определенном порядке. Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве.

1 Факторы выбора алгоритмов сортировки

Основные факторы, влияющие на выбор алгоритма сортировки, включают:

- **Структура данных**. Методы сортировки разделяют на сортировку массивов и сортировку файлов. Массивы располагаются во внутренней (оперативной) памяти ЭВМ, а файлы хранятся в более медленной, но более вместительной внешней памяти (дисках и лентах).
- **Эффективность использования памяти**. Ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных.
- **Быстродействие**. Основной параметр, характеризующий быстродействие алгоритма (вычислительная сложность).

2 Базовые алгоритмы сортировки

Далее рассмотрим некоторые базовые алгоритмы сортировки:

2.1 Сортировка пузырьком

Один из простейших методов сортировки. Заключается в постепенном смещении элементов с большим значением в конец массива.

2.2 Сортировка выбором

Алгоритм ищет наименьший элемент в текущем списке и производит обмен его значения со значением первой неотсортированной позиции.

2.3 Быстрая сортировка

Считается одним из самых быстрых алгоритмов сортировки. Работает по принципу «разделяй и властвуй».

2.4 Сортировка кучей (пирамидальная сортировка)

Алгоритм выстраивает данные в виде двоичного дерева (двоичной кучи).

2.5 Сортировка вставками

Применяется для вставки элементов массива на «своё место».

2.6 Сортировка слиянием

Следует принципу «разделяй и властвуй», согласно которому массив данных разделяется на равные части, которые сортируются по-отдельности. После они сливаются, в результате получается отсортированный массив.

7. Улучшенные сортировки. Сортировка Шелла.

Улучшенные сортировки — это методы сортировки, которые основываются на тех же принципах, что и прямые, но используют некоторые оригинальные идеи для ускорения процесса сортировки

1) Сортировка Шелла. – это улучшенный метод сортировки включениями. Был предложен Д.Л. Шеллом в 1959 г. При первом просмотре группируются и сортируются элементы, отстоящие друг от друга на 5 позиций: (X_1, X_6), (X_2, X_7), (X_3), (X_4), (X_5); при втором проходе — элементы, отстоящие друг от друга на три позиции: (X_1, X_4, X_7), (X_2, X_5), (X_3, X_6); при третьем — на 1 позицию. Поскольку первый в сортировке Шелла используемый шаг является большим, отдельные подмассивы достаточно малы по отношению к исходным. Таким образом, сортировка включением над этими подмассивами работает достаточно быстро – $O(N^2)$ (эффективно при малом N). Сортировка каждого подмассива приводит к тому, что весь массив становиться ближе к отсортированному виду, что также делает эффективным использование сортировки включением. Так как массив становится более упорядоченным, то $O(N) <$ порядок ФВС $< O(N^2)$.

Если некоторый массив частично отсортирован с использованием шага k, а затем сортируется частично с использованием шага j, то данный массив остается частично отсортированным по шагу k, т.е. последующие частичные сортировки не нарушают результата предыдущих сортировок.

В случае правильного выбора шагов порядок ФВС будет выглядеть как $O(N^{1.2})$. Д. Кнут предложил выбирать шаги из следующего ряда: 1, 4, 13, 40, 121, ... , а вычислять их по следующей формуле: $h_{k-1} = 3 \cdot h_k + 1$; $h_t = 1$; $t = [\log_3 N] - 1$ (количество просмотров), где N – размерность исходного массива. Т.е. элементы должны быть взаимно простыми числами. Исходя из этого порядок сортировки может быть аппроксимирован величиной $O(N(\log_2 N))$.

8. Улучшенные сортировки. Быстрая сортировка выбором.

Быстрая сортировка выбором. Улучшить сортировку выбором можно, если после каждого прохода получать больше информации, чем просто идентификация минимального элемента.

Анализ быстрой сортировки выбором

Описание алгоритма:

1. Сравниваем пары соседних ключей и запоминаем значение меньшего ключа из каждой пары.

2. Выполняем п.1 по отношению к значениям, полученным на предыдущем шаге. Так повторяем, пока не определим наименьший ключ и не построим бинарное дерево.

3. Вносим значение корня, найденное в п.2, в массив упорядоченных ключей.

4. Проходим от корня к листу дерева, следуя по пути, отмеченному минимальным ключом, и заменяем значение в листе на наибольшее целое число.

5. Проходим от листа к корню, по пути обновляя значения в узлах дерева, и определяем новый минимальный элемент.

6. Повторяем пп.3-6, пока минимальным элементом не будет MaxInt.

Обозначения:

- N - количество элементов.

- k_1 - константа, характеризующая стоимость одного сравнения.
- k_2 - константа, характеризующая стоимость модификации узла дерева.

Анализ:

- **Построение бинарного дерева и поиск минимума:**
  - Количество сравнений на каждом уровне бинарного дерева: $\frac{N}{2}, \frac{N}{4}, \ldots, 1$.
  - Количество уровней в бинарном дереве: $\log_2 N$.
  - Общее количество сравнений для построения дерева и нахождения минимума (шаги 1 и 2): $k_1 \cdot (N-1)$.
- **Модификация бинарного дерева:**
  - Высота бинарного дерева: $\log_2 N$.
  - Количество модификаций за одну итерацию (шаги 4 и 5): проход от корня к листу и обратно - порядка $2\log_2 N$, то есть $k_2 \cdot \log_2 N$.
  - Модификация выполняется для каждого из N элементов, то есть $(N-1)$ раз.
  - Общее количество модификаций: $k_2 \cdot \log_2 N \cdot (N-1)$.

Общее время работы:
Суммарное время работы алгоритма можно выразить как сумму времени на построение дерева и времени на его модификацию:

$$k_1(N-1) + k_2 \log_2 N(N-1)$$

Асимптотическая сложность:
Общая временная сложность алгоритма определяется как:

$$O(k_1(N-1) + k_2 \log_2 N(N-1))$$

Поскольку k_1 и k_2 являются константами, и доминирующим слагаемым является $N \log_2 N$, то асимптотическая сложность алгоритма равна:

$$O(N \log_2 N)$$

Вывод: Временная сложность быстрой сортировки выбором составляет $O(N \log_2 N)$.

9. Улучшенные сортировки. Быстрая обменная сортировка (Хоара).
Быстрая сортировка, также известная как сортировка Хоара, является одним из наиболее эффективных и широко используемых алгоритмов сортировки. Она основана на принципе "разделяй и властвуй"и отличается высокой производительностью, особенно на больших наборах данных. Основная идея заключается в рекурсивном разделении массива на две части относительно выбранного элемента (разделителя), так что все элементы меньше разделителя оказываются слева, а все элементы больше разделителя - справа.

Алгоритм Быстрой Сортировки

Алгоритм быстрой сортировки можно описать следующими шагами:

1. **Выбор разделителя (pivot):** Выбирается один из элементов массива в качестве разделителя. В простейшем варианте это первый элемент.

2. **Разделение массива:** Элементы массива переставляются таким образом, что все элементы, меньшие разделителя, оказываются перед ним, а все элементы, большие разделителя, - после него. Разделитель оказывается на своей "истинной"позиции в отсортированном массиве.

3. **Рекурсивная сортировка:** Алгоритм рекурсивно применяется к подмассивам слева и справа от разделителя (если эти подмассивы содержат более одного элемента).

Процесс Разделения Массива (Шаг 2)

Разделение массива обычно выполняется с использованием двух указателей, движущихся навстречу друг другу от краев массива. Элементы, нарушающие порядок относительно разделителя, меняются местами.

Пример

Рассмотрим пример сортировки массива с помощью быстрой сортировки:

```
Исходный массив: 30 10 40 20 15 17 45 60 30
```

Шаг 1: Выбираем 30 в качестве разделителя.
Шаг 2: Выполняем разделение:

```
( 10 17 20 15 ) 30 ( 40 45 60 )
```

Теперь разделитель 30 находится на своем месте. Алгоритм рекурсивно применяется к подмассивам `(10 17 20 15)` и `(40 45 60)`. Этот процесс продолжается, пока все подмассивы не станут единичными или пустыми.

Анализ Быстрой Сортировки

Оценка Времени

В лучшем и среднем случаях, когда разделитель каждый раз делит массив примерно пополам, время выполнения быстрой сортировки составляет $O(N \log N)$, где N - количество элементов в массиве. Это делает ее одним из самых эффективных алгоритмов сортировки общего назначения.

Худший Случай

В худшем случае, когда разделитель всегда выбирается как наименьший или наибольший элемент, время выполнения быстрой сортировки может достигать $O(N^2)$. Это происходит, например, если массив уже отсортирован в порядке возрастания или убывания, и разделителем всегда выбирается первый (или последний) элемент.

Количество Сравнений

При условии, что разделитель попадает в середину массива, количество сравнений можно оценить следующим образом:

- На первом уровне рекурсии выполняется N сравнений.
- На втором уровне будет $2 \times \frac{N}{2} = N$ сравнений.
- На каждом уровне рекурсии будет выполняться порядка N сравнений.
- Глубина рекурсии в лучшем случае составит $m = \log_2 N$ уровней.

Следовательно, количество сравнений будет $O(N \log_2 N)$.

Сравнение с Упорядоченным Массивом

Если быстрая сортировка применяется к уже упорядоченному массиву и при этом разделителем всегда выбирается первый (или последний) элемент, то каждый раз массив будет разделяться на два неравных подмассива: один с одним элементом (разделитель), а другой со всеми остальными. В этом случае глубина рекурсии будет N и сложность алгоритма составит $O(N^2)$.

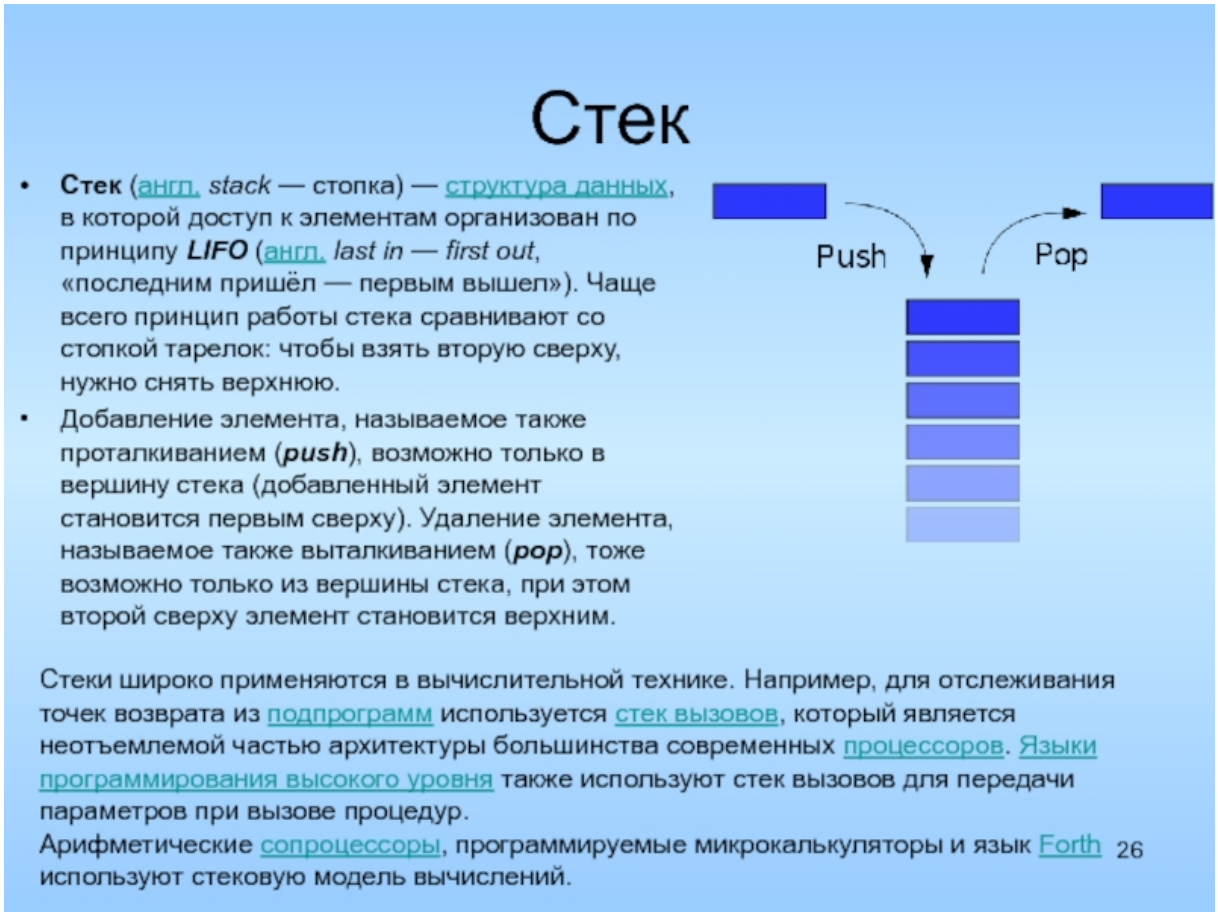
Заключение

Быстрая сортировка является мощным и эффективным алгоритмом, идеальным для сортировки больших объемов данных. Несмотря на возможность худшего случая $O(N^2)$, средняя производительность $O(N \log_2 N)$ делает ее одной из предпочтительных сортировок для многих приложений.

10. СД типа «стек». Базовые операции. Интерфейс СД типа стек.

Стек – это последовательность, в которой включение и исключение элемента осуществляется с одной стороны последовательности (вершины стека). Так же осуществляется и операция доступа. Структура функционирует по принципу LIFO (последний пришедший обслуживается первым).

- Операция инициализации.
- Операция включения элемента в стек.
- Операция исключения элемента из стека.
- Операция проверки: стек пуст / стек не пуст.
- Операция проверки: стек переполнен / стек не переполнен (данная операция характерна для стека как отображения на массив).
- Операция чтения элемента (доступ к элементу).



11. СД типа «стек». Применение СД типа «стек» в в вычислительных системах и алгоритмах

3 Применение стека в вычислительных системах и алгоритмах

1. **Временное хранение данных.** Промежуточные результаты работы программы необходимо где-то временно сохранить, прежде чем они будут использованы в дальнейшей обработке. *5*

2. **Вычисление выражений.** Стек является идеальным средством для вычисления произвольных арифметических и логических выражений. *5*

3. **Организация механизма подпрограмм.** Стек создаёт возможности для вызова подпрограмм, обеспечивая механизм возврата в ту точку, откуда подпрограмма была вызвана, после окончания её работы. *5*

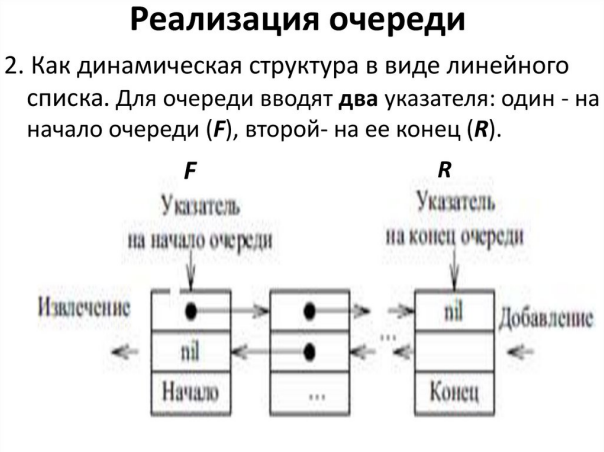
4. **Передача параметров и создание локальных переменных.** При входе в процедуру в стеке выделяется место под локальные переменные подпрограммы, а по завершении работы это место освобождается. *5*

5. **Эффективные алгоритмы.** Например, сканирование Грэма, алгоритм для выпуклой оболочки двумерной системы точек. Выпуклая оболочка подмноже-

ства входных данных сохраняется в стеке, который используется для поиска и удаления вогнутостей на границе при добавлении новой точки к оболочке.

12. СД типа «очередь». Базовые операции. Интерфейс СД типа «очередь».
Очередь – последовательность, в которую включают элементы с одной стороны, а исключают – с другой. Структура функционирует по принципу FIFO (поступивший первым обслуживается первым). Условные обозначения очереди:

- Операция инициализации.
- Операция включения элемента в очередь.
- Операция исключения элемента из очереди.
- Операция проверки: очередь пуста / очередь не пуста.
- Операция проверки: очередь переполнена / очередь не переполнена.

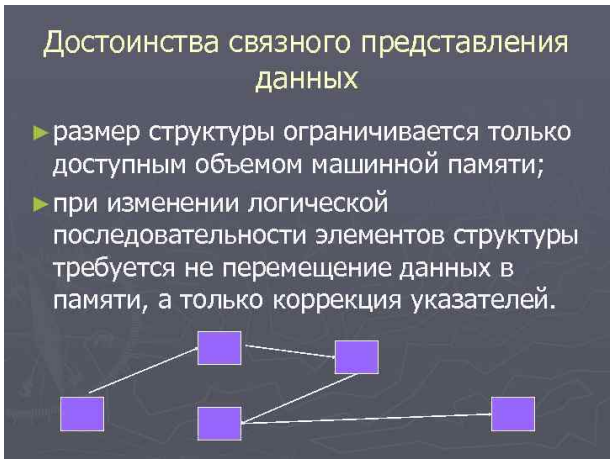


13. СД типа «очередь». Очереди с приоритетами.
Очередь с приоритетами — это абстрактный тип данных в программировании, поддерживающий две обязательные операции — добавить элемент и извлечь максимум (минимум). Предполагается, что для каждого элемента можно вычислить его приоритет — действительное число или в общем случае элемент линейно упорядоченного множества.
Основные методы, реализуемые очередью с приоритетом:
insert(ключ, значение) — добавляет пару (ключ, значение) в хранилище; $extract_minimum()$ — возвращает пару (ключ, значение) с минимальным значением ключа, удаляя её из хранилища. При этом меньшее значение ключа соответствует более высокому приоритету.
14. СД типа «очередь». Применение очередей в вычислительных системах.
Применение очередей в вычислительных системах:
В компьютерной системе могут существовать очереди задач, ожидающих принтера, доступа к дисковому хранилищу или даже в системе с разделением времени использования центрального процессора. 2 Очередь используется в качестве буфера, когда есть несоответствие скорости между производителем и потребителем, например, клавиатура и процессор.
Очередь может использоваться там, где есть один ресурс и несколько потребителей,

например, один центральный процессор и несколько процессов.
В сети очередь используется в таких устройствах, как маршрутизатор/коммутатор и почтовая очередь.
Очередь может использоваться в различных алгоритмических методах, таких как поиск по ширине, топологическая сортировка и т.д..

15. Связное представление данных в памяти компьютера.

Связное представление данных в памяти компьютера — это способ, при котором в каждой записи предусматривается дополнительное поле, в котором размещается указатель (ссылка).



16. Односвязный линейный список. Базовые операции. Функциональный и свободный списки.

Связный список – СД элементами которой являются записи, связанные друг с другом с помощью указателей, хранящихся в самих элементах.
Односвязный линейный список — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей]]
Операции, определяющие структуру типа линейный односвязный список:

1. **Инициализация.**
2. **Включить элемент за рабочий указатель списка.**
3. **Исключить элемент, находящийся за рабочим указателем списка.**
4. **Передвинуть рабочий указатель на один шаг.**
5. **Проверка: список пуст / список не пуст.**
6. **Поместить рабочий указатель в начало списка (производная операция).**
7. **Поместить рабочий указатель в конец списка (производная операция).**
8. **Сделать список пустым.**

17. Односвязный линейный список. Реализация ОЛС, как отображение на массив.

Для реализации односвязного линейного списка (ОЛС) как отображения на массив

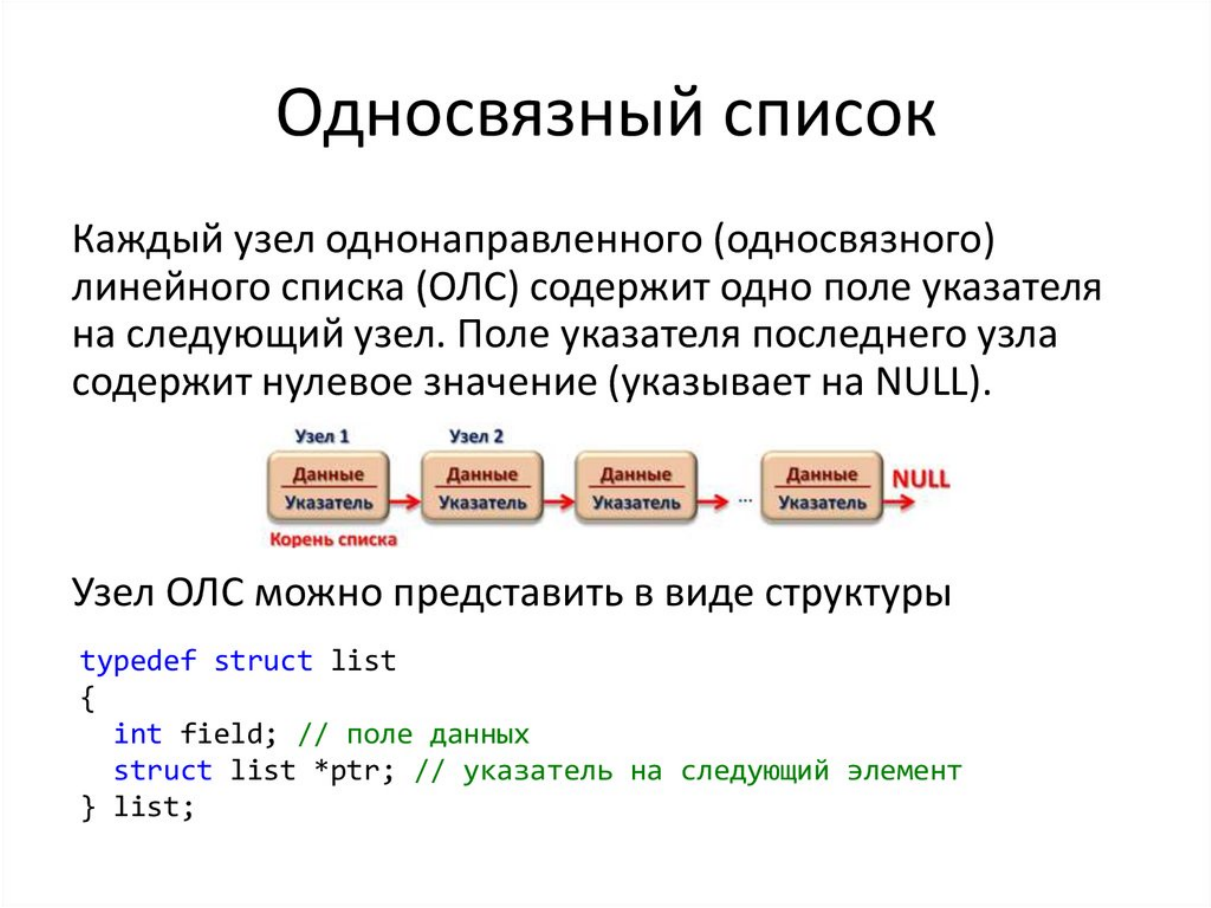
вместо указателя на следующий элемент используется индекс следующего элемента. Например, структура узла может выглядеть так: struct list string val; size$_t next$; ., ¯, .

Также можно использовать функцию для преобразования односвязного списка в массив. 2 Например, на языке C это может быть такая функция linkedListToArray(struct Node* head, int arr, [] int* size): C int count = 0; struct Node* curr = head; // Пройтись по связанному списку и заполнить массив while (curr != NULL) arr[count] = curr->data; count++; curr = curr->next; * size = count;

18. Односвязный линейный список. Реализация ОЛС с использованием указательного типа. Интерфейс ОЛС.

Реализация ОЛС с использованием указательного типа может быть осуществлена на базе классов объектно-ориентированного программирования. Узел списка представляет собой класс, содержащий два скрытых поля — значение и указатель на следующий элемент. Чтобы позволить списку иметь доступ к скрытым полям узла, класс список объявляют как дружественный. 1

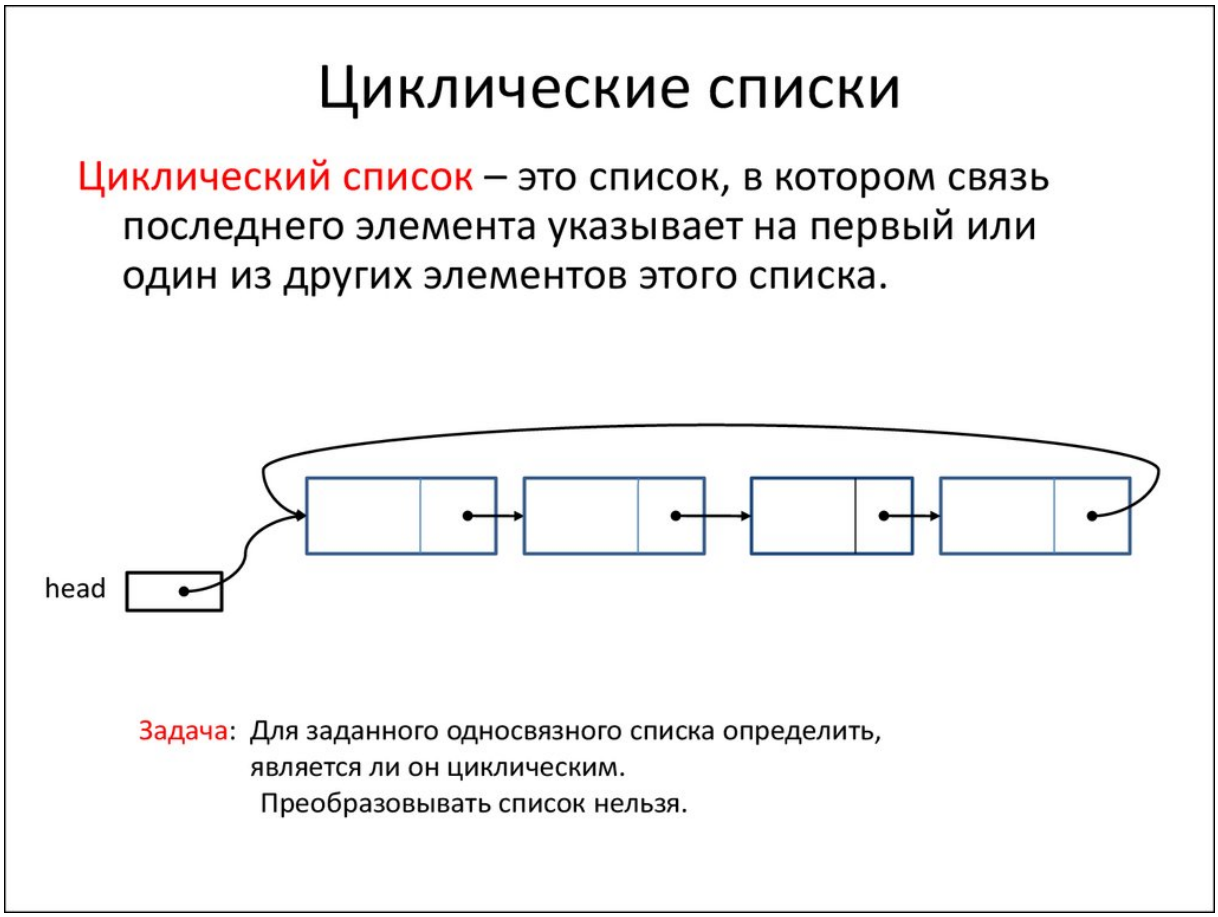
Некоторые основные действия, производимые над элементами ОЛС: инициализация списка, добавление узла в список, удаление узла из списка, удаление корня списка, вывод элементов списка, взаимообмен двух узлов списка.



19. Односвязный линейный список. Применение односвязного линейного списка. Стек, как отображение на список. Очередь, как отображение на список.

Применение односвязного линейного списка широко распространено в приложениях, где непредсказуемы требования к размеру памяти для хранения данных и необходимо большое число сложных операций над данными, особенно включений и исключе-

ний. На базе линейных списков могут строиться стеки, очереди и деки.
Стек на основе односвязного линейного списка представляется как линейный список, в котором включение элементов всегда производится в начало списка, а исключение — также из начала. Для представления стека достаточно иметь один указатель — top, который всегда указывает на последний записанный в стек элемент.
Очередь на основе односвязного линейного списка — это список, операции чтения и добавления элементов в котором проводятся по принципу «Первый пришёл, первый вышел» (FIFO — first in, first out). Таким образом, для чтения в очереди доступна только голова, в то время как добавление проводится только в хвост.
20. СД типа циклический односвязный линейный список. Интерфейс циклического ОЛС. Такая модификация линейного списка позволяет упростить алгоритмы для работы со списком, любой элемент доступен, но необходимо отслеживать зацикливание, т.е. должен быть внешний указатель. Операции для работы с циклическим линейным списком те же, что и для односвязного линейного списка.



21. Двусвязный линейный список. Базовые операции. Интерфейс ДЛС. Двусвязный линейный список — это динамический линейный список, в котором каждый элемент с помощью двух указателей связывается с предыдущим и следующим элементами. В списке такого типа указатель можно перемещать как в одну, так и в другую сторону. Дескриптор данной структуры состоит из трех полей указательного типа.
Операции:

- **Инициализация:** Создание нового пустого списка.

- **Сделать список пустым:** Удаление всех элементов из списка.
- **Проверка:**
  - Проверка, является ли список пустым.
  - Проверка, является ли список непустым.
- **Установка указателя:**
  - Установка указателя на начало списка.
  - Установка указателя на конец списка.
- **Перемещение указателя:**
  - Перемещение указателя на один шаг вперед.
  - Перемещение указателя на один шаг назад.
- **Включение элемента:**
  - Включение элемента в список до указателя.
  - Включение элемента в список после указателя.
- **Исключение элемента:**
  - Исключение элемента из списка до указателя.
  - Исключение элемента из списка после указателя.

4 Особенности включения элемента

При включении элемента в список необходимо распознать следующие три ситуации:

1. **Список пуст:** В этом случае новый элемент становится единственным элементом списка.
2. **Включение элемента в середину списка:** Новый элемент вставляется между двумя существующими элементами.
3. **Включение элемента в список в качестве первого:** Новый элемент вставляется в начало списка.

5 Реализация циклических двусвязных списков

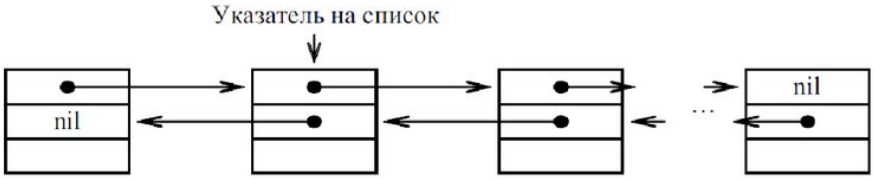
Для упрощения операций включения/исключения используются циклические двусвязные списки. В такой реализации формируются указатели на первый и последний элементы списка. Это позволяет избежать отдельных маркеров для начала и конца списка.

6 Допустимые операции для циклических двусвязных списков

Допустимые операции для циклического двусвязного списка аналогичны операциям для линейного двусвязного списка, описанным в разделе 1.

Линейный двунаправленный список

Линейный двунаправленный список- любой элемент хранит собственно данные, а также две ссылки указывающую на предыдущий и следующий элемент в списке или является пустым у первого и последнего элемента соответственно.



Основные операции:

- ✓ Вставка элемента
- ✓ Просмотр
- ✓ Поиск
- ✓ Удаление элемента.

Примечание:
Нет необходимости обеспечивать позиционирование на первый элемент, т.к. благодаря двум указателям можно получить доступ к любому элементу, осуществляя переходы в прямом и обратном направлении.

Наличие двух указателей, позволяет ускорить операции связанные с передвижением по списку. Однако элементы списка за счет дополнительного поля занимают больший объем памяти. Кроме того операции вставки и удаления усложняются за счет необходимости манипулирования большим числом указателей.

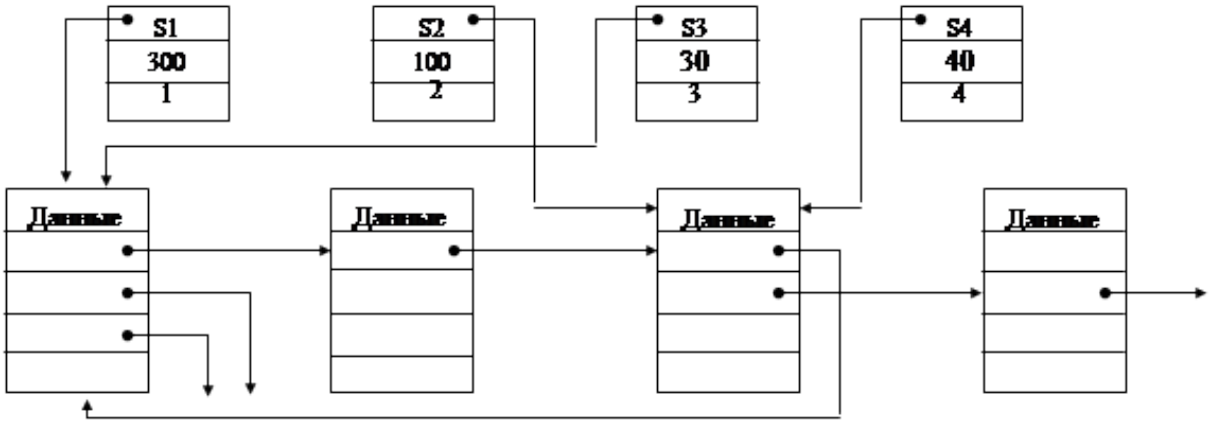
22. СД типа «дек». Базовые операции. Интерфейс СД типа «дек».

Дек – линейная структура (последовательность) в которой операции включения и исключения элементов могут выполняться как с одного, так и с другого конца последовательности.

Допустимые операции:
инициализация;
проверка: дек пуст / дек не пуст;
включение элемента в начало / конец дека;
исключение элемента из начала / конца дека.

23. Многосвязные линейные списки (мультисписки).

Многосвязный список (мультисписок) — это структура данных, представляющая собой расширение понятия списка путём добавления второго ссылочного поля. В результате каждый элемент мультисписка может ссылаться как на следующий элемент «по горизонтали» (как в классическом понимании списка), так и на следующий элемент «по вертикали». Таким образом, каждый элемент мультисписка может быть первым элементом другого мультисписка.



24. Классификация алгоритмов по временной сложности.

По временной сложности все задачи классифицируются следующим образом: 1 класс. Задачи класса P – это задачи, для которых известен алгоритм и временная сложность оценивается полиномиальной функцией

$$f(N) = a_k N + a_{k-1} N^2 + \cdots + a_1 N^k + a_0 N^{k+1}$$

. Алгоритмы таких задач называют также "хорошими" алгоритмами. Этих алгоритмов мало, они образуют небольшой по мощности класс. Легко реализуются. Функция временной сложности для таких алгоритмов имеет вид O(N^k).

2 класс. Задачи класса E – это задачи, для которых алгоритм известен, но сложность таких алгоритмов O(f^N), где f – константа. Задачи такого класса – это задачи построения всех подмножеств некоторого множества, задачи построения всех полных подграфов графа. При небольших N экспоненциальный алгоритм может работать быстрее полиномиального.

3 класс. На практике существуют задачи, которые не могут быть отнесены ни к одному из вышерассмотренных классов. Это прежде всего задачи, связанные с решением систем уравнений с целыми переменными, задачи определения цикла, проходящего через все вершины некоторого графа, задачи диагностики и т.д. Такие задачи независимы от компьютера, от языка программирования, но могут решаться человеком.

N	2	3	5	10	20	50	100	1000
$1000N^2$	4 мкс	10 мкс	25 мкс	100 мкс	400 мкс	2.5 мс	10 мс	1 с
$100N^3$	1 мкс	3 мкс	18.5 мкс	100 мкс	800 мкс	13 мс	100 мс	100 с
2^N	4 мс	8 мс	30 мс	1 мкс	1 мс	10 сут.	Миллиарды лет	
2^{2^N}	16 мс	250 мс	30 с	Миллиарды лет				

25. СД типа «таблица прямого доступа».

Таблица — это набор элементов одинаковой организации, каждый из которых можно представить в виде двойки $\langle K, V \rangle$, где K — ключ, а V — тело (информационная часть) элемента. Ключ уникален для каждого элемента, т.е. в таблице нет двух элементов с одинаковыми ключами. Ключ используется для доступа к элементам при выполнении операций.

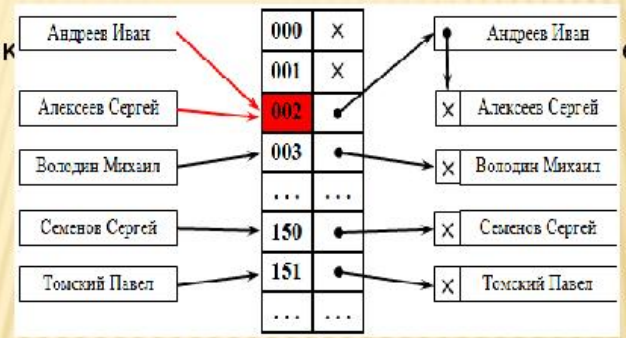
Таблица — динамическая структура. Над таблицей определены следующие основные операции:

1. Инициализация.

2. Включение элемента.
3. Исключение элемента с заданным ключом.
4. Чтение элемента с заданным ключом.
5. Изменение элемента с заданным ключом.
6. Проверка пустоты таблицы.
7. Уничтожение таблицы.

Для хранения элементов неупорядоченной таблицы используется дополнительная структура данных — линейный список (последовательный или односвязный). Элементы списка неупорядочены по значению ключа. **26. СД типа «хеш-таблица». Разрешение коллизий с помощью цепочек.**
СД типа хеш-таблица использует алгоритм поиска, основанный на идее хеширования. До сих пор, для поиска требуемой записи мы использовали организацию просмотра некоторого количества ключей. Очевидно, что эффективными алгоритмами поиска являются те, которые проходят это количество ключей при минимальном числе сравнений. В идеале хотелось бы иметь такую организацию таблицы, у которой не было бы ненужных сравнений, т.е. чтобы каждый ключ находился за одно сравнение, но в этом случае положение записи в таблице должно зависеть от значения ключа.

Механизм разрешения коллизий является важной составляющей любой хеш-таблицы. Коллизии осложняют использование хеш-таблиц, так как нарушают однозначность соответствия между хеш-кодами и данными.
Тем не менее, существуют способы преодоления возникающих сложностей:
метод цепочек



е);

Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

27. СД типа «хеш-таблица». Метод открытой адресации.

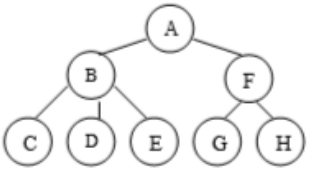
28. Нелинейные СД. Деревья. Основные определения.

Для повышения эффективности алгоритмов используются нелинейные структуры данных. Они усложняют реализацию алгоритма, но улучшают его эффективность.

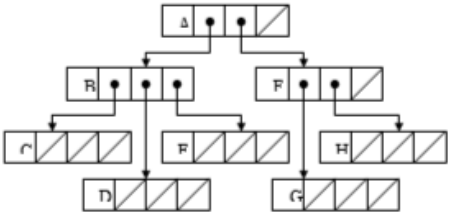
Дерево – конечное непустое множество T, состоящее из одного и более узлов таких, что выполняются следующие условия: 1) Имеется один специально обозначенный узел, называемый корнем данного дерева.
2) Остальные узлы (исключая корень) содержатся в m>=0 попарно не пересекающихся множествах T1, T2, . . . , Tm, каждое из которых в свою очередь является деревом. Деревья T1, T2, . . . , Tm называются поддеревьями данного корня.
Если подмножества T1, T2, . . . , Tm упорядочены, то дерево называют упорядоченным. Если два дерева считаются равными и тогда, когда они отличаются порядком, то такие деревья называются ориентированными деревьями. Конечное множество непересекающихся деревьев называется лесом.
Бинарное дерево – конечное множество элементов, которое может быть пустым, состоящее из корня и двух непересекающихся бинарных деревьев, причем поддеревья упорядочены: левое поддерево и правое поддерево.
Количество подмножеств для данного узла называется степенью узла. Если такое количество равно нулю, то узел является листом. Максимальная степень узла в дереве – степень дерева. Уровень узла – длина пути от корня до рассматриваемого узла. Максимальный уровень дерева – высота дерева.

29. Деревья. Представление деревьев в связной памяти.

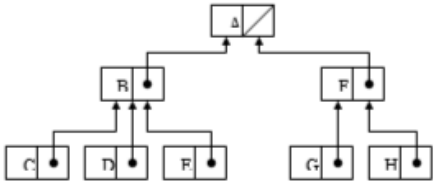
Представление деревьев в связной памяти ЭВМ.



Различают три основных способа представления деревьев в связной памяти: стандартный, инверсный, смешанный. Рассмотрим эти способы для представленного дерева.



При стандартном способе узлы, находящиеся на одном уровне являются *братьями*. Если же узел находится на более нижнем уровне, то он считается *сыном*.



При инверсном способе каждый узел дерева имеет указатель, показывающий на родителя.

Если же говорить о смешанном способе представления дерева в связной памяти, то здесь, как видно из названия, каждый узел включает указатели, указывающие как на сыновей, так и на родителя.

30. Деревья. Алгоритм прохождения n-арных деревьев в ширину.

1. **<взять две очереди O1 и O2>**;
2. **<поместить корень в очередь O1>**;
3. **while <O1 или O2 не пуста> do**
   - **if <O1 не пуста> then**
     - {P – узел, находящийся в голове очереди O1}
     - **begin**
       - **<исключить узел из очереди O1 и пройти его>**;
       - **<поместить все узлы, относящиеся к братьям этого узла P в очередь O2>**;
     - **end**
   - **else <O1=O2; O2=∅> и повторяем цикл.**

31. Деревья. Алгоритм прохождения n-арных деревьев в глубину.

Алгоритм прохождения дерева в глубину:

- <пустой стек S>;
- <пройти корень и включить его в стек S>;
- while <стек S не пуст> do
  - begin {пусть P – узел, находящийся в вершине стека S}

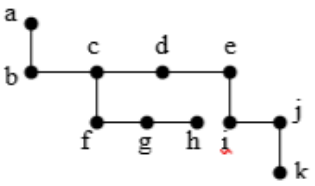
- if <не все сыновья узла P пройдены> then <пройти старшего сына и включить его в стек S> else begin
- <исключить из вершины стека узел P>;
- if <не все братья вершины P пройдены> then <пройти старшего брата и включить его в стек S>
- end;

- end;

При прохождении в глубину представленного дерева, список его вершин, записанных в порядке их посещения, будет выглядеть следующим образом:

A, B, C, D, E, F, G, H, I, J, K

32. Деревья. Алгоритм представления n-арных деревьев в виде бинарных. Между деревьями общего вида (узел дерева может иметь более двух сыновей) и бинарными деревьями существует взаимно однозначное соответствие, поэтому бинарные деревья часто используют для представления деревьев общего вида. Для такого представления используют следующий алгоритм: 1. изображаем корень дерева; 2. по вертикали изображаем старшего сына этого корня; 3. по горизонтали вправо от этого узла представляем всех его братьев; 4. пп. 1, 2, 3 повторяем для всех его узлов.



Таким образом, получаем, что вертикальные ребра изображают левые поддеревья, а горизонтальные – правые.

Данный алгоритм можно распространить и для леса.

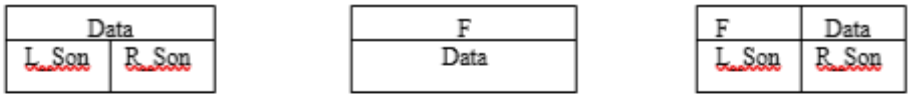
Теорема. Число бинарных деревьев с n вершинами можно выразить формулой: $\frac{(2n)!}{(n+1)(n!)^2}$. Например, для n=3 имеем $\frac{(2\cdot 3)!}{(3+1)(3!)^2} = \frac{1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6}{4\cdot 1\cdot 2\cdot 3\cdot 1\cdot 2\cdot 3} = 5.$

Проиллюстрируем этот пример графически:

33. Деревья. Представление бинарных деревьев в памяти. Прошитые бинарные деревья.

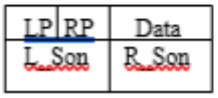
Представление бинарных деревьев в связной памяти. Прошитые деревья

При представлении бинарных деревьев в связной памяти, различают также три основных способа представления: стандартный, инверсный, смешанный. Узлы дерева при каждом таком представлении выглядят следующим образом:



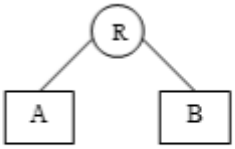
стандартный инверсный смешанный

Если мы воспользуемся прошитым бинарным деревом, то вид узла для него будет таким:

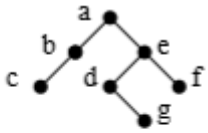


Один из возможных способов прошивки бинарного дерева заключается в том, что для каждого узла, не имеющего левого поддерева, запоминается ссылка на непосредственного предшественника, а для каждого узла, не имеющего правого поддерева ,запоминается ссылка на его непосредственного преемника. Предшественников и преемников для узла определяют исходя из списка, который получается при прохождении рассматриваемого бинарного дерева в симметричном порядке. Для того, чтобы отличать ссылку на левого или правого сына (такие ссылки называют *структурными связями*) от ссылки на предшественника или преемника узла (такие ссылки называют "нитями"), используются индикаторные поля LP и RP.

Симметричным прохождением бинарного дерева является такое прохождение, когда мы проходим сначала левое поддерево, затем посещаем корень и последним посещаем правое поддерево: A R B.

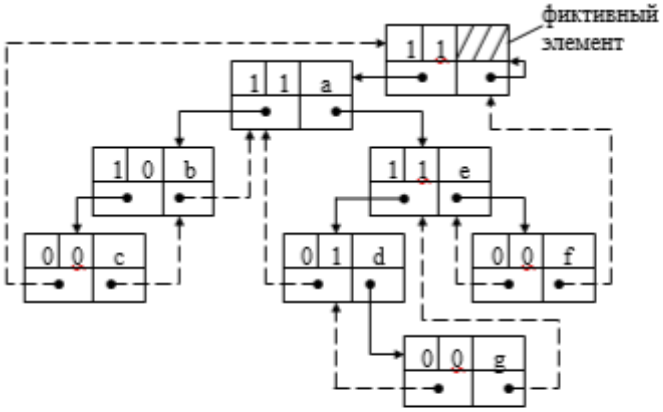


Пример. Рассмотрим бинарное дерево:



При его симметричном прохождении получаем следующий список вершин: c, b, a, d, g, e, f.

В памяти это дерево будет выглядеть так:



Использование "прошивок" существенно убыстряет прохождение дерева, позволяет не использовать стек, но при этом усложняются алгоритмы включения / исключения узла, т.к. в прошитом дереве необходимо управлять не только структурными связями, но и "нитями".

34. Деревья. Формирование идеально сбалансированного дерева.

Дерево идеально сбалансировано, если для каждого его узла количество узлов в левом и правом поддереве различается не более чем на 1. Дерево сбалансировано, если для каждого его узла высота двух соответствующих поддеревьев различается не более чем на 1. Последовательность шагов при создании идеально сбалансированного дерева для заранее известного числа вершин (n):

1. Создается корневая вершина.
2. Тем же способом строится левое поддерево, содержащее $n_l = \left\lfloor \frac{n}{2} \right\rfloor$ вершин.
3. Тем же способом строится правое поддерево, содержащее $n_r = n - n_l - 1$ вершин.
4. Рекурсия завершается, если поддерево не содержит ни одной вершины ($n = 0$).

35. Деревья. Применение бинарных деревьев в алгоритмах поиска. Таблица, как отображение на бинарное дерево. Операция включения в бинарное дерево. Анализ сложности.

В односвязном списке невозможно использовать бинарные методы поиска, так как они требуют доступа к элементам по индексу, что возможно только в последовательной памяти. Однако, если использовать бинарные деревья, то в такой связной структуре можно получить алгоритм поиска со сложностью $O(\log_2 N)$.

Такое дерево, называемое *бинарным деревом поиска*, реализуется следующим образом: для любого узла дерева с ключом T_i все ключи в левом поддереве должны быть меньше T_i, а в правом – больше T_i.

Принцип поиска

В дереве поиска можно найти место каждого ключа, двигаясь, начиная от корня и переходя на левое или правое поддерево, в зависимости от значения его ключа.

- Если искомый ключ равен ключу текущего узла, поиск завершен.
- Если искомый ключ меньше ключа текущего узла, переходим в левое поддерево.
- Если искомый ключ больше ключа текущего узла, переходим в правое поддерево.

Этот процесс повторяется рекурсивно, пока не будет найден нужный ключ или не будет достигнут конец дерева (лист).

Сложность поиска

Из n элементов можно организовать бинарное дерево (идеально сбалансированное) с высотой не более чем $\log_2 N$. Высота дерева определяет максимальное количество сравнений при поиске. Таким образом, для сбалансированного бинарного дерева поиска сложность алгоритма поиска составляет $O(\log_2 N)$.

Важные замечания

- Сбалансированность дерева является ключевым фактором для достижения сложности $O(\log_2 N)$. В худшем случае, когда дерево вырождается в односвязный список (все элементы добавляются в порядке возрастания или убывания), сложность поиска может достигать $O(N)$.
- Для поддержания сбалансированности дерева могут использоваться различные алгоритмы балансировки, такие как AVL-деревья или красно-черные деревья. Эти алгоритмы автоматически перестраивают дерево при добавлении или удалении элементов, чтобы гарантировать, что высота дерева остается логарифмической.

Рассмотрим случай постоянно растущего, но никогда не убывающего дерева. Хорошим примером этого является построение частотного словаря. В этой задаче задана последовательность слов и нужно установить число появления каждого слова. Это означает, что, начиная с пустого дерева, каждое слово, встречающееся в тексте, ищется в нем. Если это слово найдено, то счетчик появления данного слова увеличивается; если же слово не найдено, то в дерево включается новое слово со значением счетчика = 1.

Листинг 1: Представление типа данных и процедуры Put на Pascal

```
type
  ElPtr = ^Element;
  Element = record
    Key: integer;
    n: integer;
    L_Son, R_Son: ElPtr;
  end;

Procedure Put (x: integer; var t: ElPtr);
begin
  if t=nil then begin
    New(t);
    with t^ do begin
      key:=x;  n:=1;  L_Son:=nil;  R_Son:=nil;
    end;
  end
  else if x> t^.key then Put(x, t^.R_Son)
  else if x<t^.key then Put(x, t^.L_Son)
  else t^.n:=t^.n+1;
end;
```

Таблица может быть представлена в виде бинарного дерева. При таком методе построения каждый узел дерева представляет собой элемент таблицы, причём корневым узлом становится первый элемент, встреченный компилятором при заполнении таблицы.

7 Алгоритм заполнения бинарного дерева

1. Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.

2. Сделать текущим узлом дерева корневую вершину.

3. Сравнить имя очередного идентификатора с именем идентификатора, содержащегося в текущем узле дерева.

4. Если имя очередного идентификатора меньше, то перейти к шагу 5, если равно — прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно), иначе — перейти к шагу 7.

5. Если у текущего узла существует левая вершина, то сделать её текущим узлом и вернуться к шагу 3, иначе — перейти к шагу 6.

6. Создать новую вершину, поместить в неё информацию об очередном идентификаторе, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

7. Если у текущего узла существует правая вершина, то сделать её текущим узлом и вернуться к шагу 3, иначе - перейти к шагу 8.

8. Создать новую вершину, поместить в неё информацию об очередном идентификаторе, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

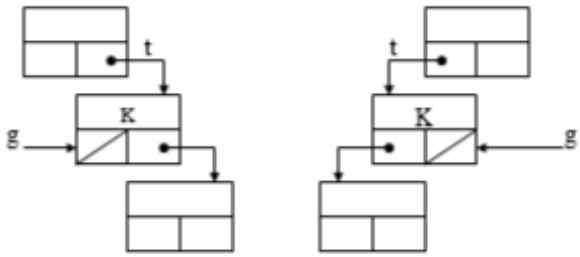
Метод бинарного дерева является удачным механизмом для организации таблиц идентификаторов и нашёл своё применение в ряде компиляторов. Худшая временная сложность операции вставки в бинарное дерево поиска равна **O(h)**, где *h* - высота дерева. В худшем случае, когда дерево вырождается в односвязный список, высота *h* может быть равна *n*, где *n* - количество узлов в дереве. Следовательно, в этом случае худшая временная сложность операции вставки будет **O(n)**. **36. Деревья. Операция исключения из бинарного дерева. Анализ сложности.**

Операция исключения из бинарного дерева.

При удалении узла дерево должно оставаться упорядоченным относительно ключа:
а) удаляется лист;
б) удаляется узел с одним потомком (потомок слева или справа);
в) удаляется узел с двумя потомками, но в левом потомке нет правого поддерева;
г) общий случай.

Рассмотрим случай б):

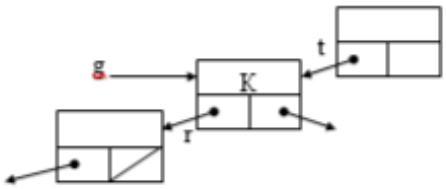
```
if g^.L_Son=nil then begin
                        g:=t;
                        t:=t^.R_Son;
                        dispose(g);
                     end;
if g^.R_Son=nil then begin
                        g:=t;
                        t:=t^.L_Son;
                        dispose(g);
                     end;
```



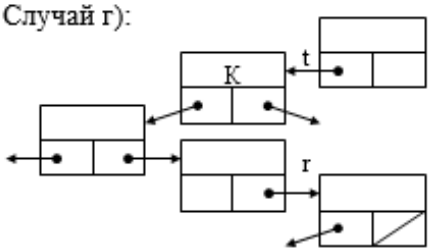
Случай а) является частным случаем этого алгоритма.

Случай в): r указывает на элемент, не имеющий правого поддерева.

```
g:=t;
g^.key:=r^.key;
g:=r;
r:=r^.L_Son;
dispose(g);
```



Случай г):



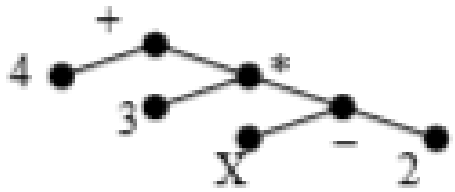
Мы ищем такой элемент, у которого отсутствует правое поддерево (на такой элемент указывает r). Последовательность операторов та же, т.е. удаляемый элемент нужно заменить самым правым элементом его левого поддерева. Такие элементы не могут иметь более одного потомка.

```
Procedure Get (x: integer; var t: ElPtr);
var g: ElPtr;
begin
   if t=nil then {обработка исключительной ситуации, когда элемента с заданным ключом
                  в дереве нет}
   else if x>t^.key then Get(x, t^.R_Son)
        else if x<t^.key then Get(x, t^.L_Son)
                         else begin
                                 g:=t;
                                 if g^.L_Son=nil then t:=g^.R_Son
                                                 else if g^.R_Son=nil then t:=g^.R_Son
                                                                     else D(g^.L_Son); {найти r)
                                 dispose(g);
                              end;
end;
```

Наихудшая сложность операции удаления в бинарном дереве — O(n), где n — количество узлов в дереве.Это объясняется тем, что для удаления элемента нужно обойти все узлы, чтобы найти его.

37. Деревья. Применение бинарных деревьев. Алгоритмы прохождения бинарных деревьев.

1) Любое алгебраическое выражение, которое содержит переменные, числа, знаки операций и скобки можно представить в виде бинарного дерева (исходя из бинарности этих операций). При этом знак операции помещается в корне, первый операнд – в левом поддереве, а второй операнд – в правом. Скобки при этом опускаются. В результате все числа и переменные окажутся в листьях, а знаки операций – во внутренних узлах. ассмотрим пример: 3 * (x – 2) + 4.



Если же осуществить просмотр дерева в обратном порядке (левое поддерево, правое поддерево, корень), то получим обратную польскую запись. 2) Классическая программа из класса интеллектуальных: построение деревьев решений. В этом случае не листовые (внутренние) узлы содержат предикаты – вопросы, ответы на которые могут принимать значения “да” или “нет”. На уровне листьев находятся объекты (альтернативы, с помощью которых распознаются программы). Пользователь получает вопросы, начиная с корня, и, в зависимости от ответа, спускается либо на левое поддерево, либо на правое. Таким образом, он выходит на тот объект, который

соответствует совокупности ответов на вопрос. 3) Применение практических задач. Рассмотрим три основных способа прохождения бинарного дерева: прямой, обратный и симметричный. Эти способы определяют порядок, в котором мы посещаем узлы дерева. Все они основаны на рекурсивном подходе.

Прямое прохождение

Прямое прохождение, также известное как обход «сверху вниз», выполняется согласно следующей рекурсивной процедуре:

1. Посетить корень.
2. Пройти в прямом порядке левое поддерево.
3. Пройти в прямом порядке правое поддерево.

Обратное прохождение

Обратное прохождение, также известное как обход «снизу вверх», выполняется согласно следующей рекурсивной процедуре:

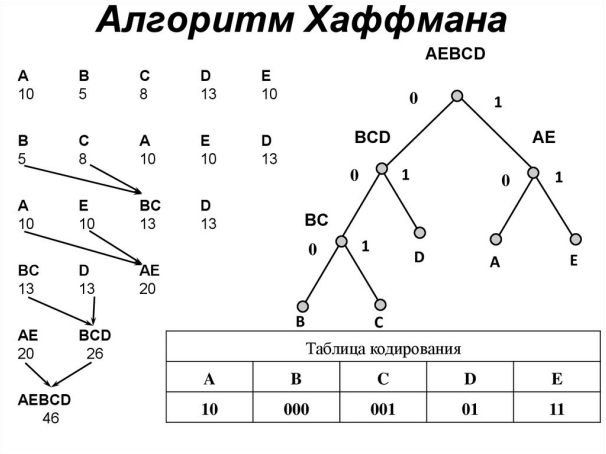
1. Пройти в обратном порядке левое поддерево.
2. Пройти в обратном порядке правое поддерево.
3. Посетить корень.

Симметричное прохождение

Симметричное прохождение, также известное как центрированный обход, выполняется согласно следующей рекурсивной процедуре:

1. Пройти в симметричном порядке левое поддерево.
2. Посетить корень.
3. Пройти в симметричном порядке правое поддерево.

Коды Хаффмена. Общие сведения. Префиксные коды.
Коды Хаффмана — это особый тип оптимального префиксного кода, который обычно используется для сжатия данных без потерь.
Идея алгоритма состоит в том, что зная вероятности вхождения символов в исходный текст, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью присваиваются более короткие коды.
Префиксный код — это код, в котором никакое кодовое слово не является префиксом любого другого кодового слова. Эти коды имеют переменную длину. Оптимальный префиксный код — это префиксный код, имеющий минимальную среднюю длину.
Коды Хаффмана обладают свойством префиксности, то есть ни одно кодовое слово не является префиксом другого, что позволяет однозначно их декодировать, несмотря на их переменную длину.
Коды Хаффмена. Алгоритмы построения кода Хаффмена. Сбалансированные деревья



Коды Хаффмана — это метод сжатия данных, разработанный Дэвидом Хаффманом в 1952 году.[1]. Он использует схему кодирования переменной длины для присвоения двоичных кодов символам в зависимости от того, как часто они встречаются в данном тексте. Символу, который встречается чаще всего, присваивается наименьший код, а символу, который встречается реже всего, — наибольший код.

Алгоритм построения кода Хаффмана состоит из двух основных этапов:

1. Построение оптимального кодового дерева с использованием входного набора символов и веса/частоты для каждого символа. Для этого используется приоритетная очередь, чтобы узлы с наименьшей частотой имели наивысший приоритет.

2. Построение отображения код-символ на основе построенного дерева. Чтобы определить битов является кодом данного символа, записанным в обратном порядке.

Алгоритм создания дерева Хаффмана:

1. Создать конечный узел для каждого символа и создать минимальную кучу, используя все узлы (значение частоты используется для сравнения двух узлов в минимальной куче).

2. Повторить шаги с 3 по 5, пока куча имеет более одного узла.

3. Извлечь из кучи два узла, скажем, x и y, с минимальной частотой.

4. Создать новый внутренний узел z с x в качестве левого дочернего узла и y в качестве правого дочернего узла. Также frequency (z) = frequency (x) + frequency (y).

5. Добавить z в минимальную кучу.

Последний узел в куче — это корень дерева Хаффмана.

40. АВЛ-деревья. Определения АВЛ-деревьев.

Дерево является сбалансированным тогда и только тогда, когда для каждого узла высота его двух поддеревьев различна не более чем на 1. Деревья, удовлетворяющие этому условию, часто называются АВЛ – деревьями. Все идеально сбалансированные деревья так же являются АВЛ деревья. Это определение привело к более простому

[1]Источник: [1]

алгоритму балансировки, а средняя длина поиска остается почти такой же, как и у идеально сбалансированного упорядоченного дерева $O(\log_2 N)$.

41. АВЛ-деревья. Операция включения в АВЛ-дерево. Операция исключения из АВЛ-дерева.

Включение в АВЛ-дерево

Алгоритм включения вершины:

a. Поиск места включения ключа в бинарное дерево.

b. Включение вершины.

c. Проверка балансирования узлов (Если сбалансированы, то выход).

d. При разбалансировании узлов выявляется необходимость:

-Одного вращения, выполняем эту операцию и переходим к шагу 3.

-Двух вращений, выполнение операции и переход к шагу 3.

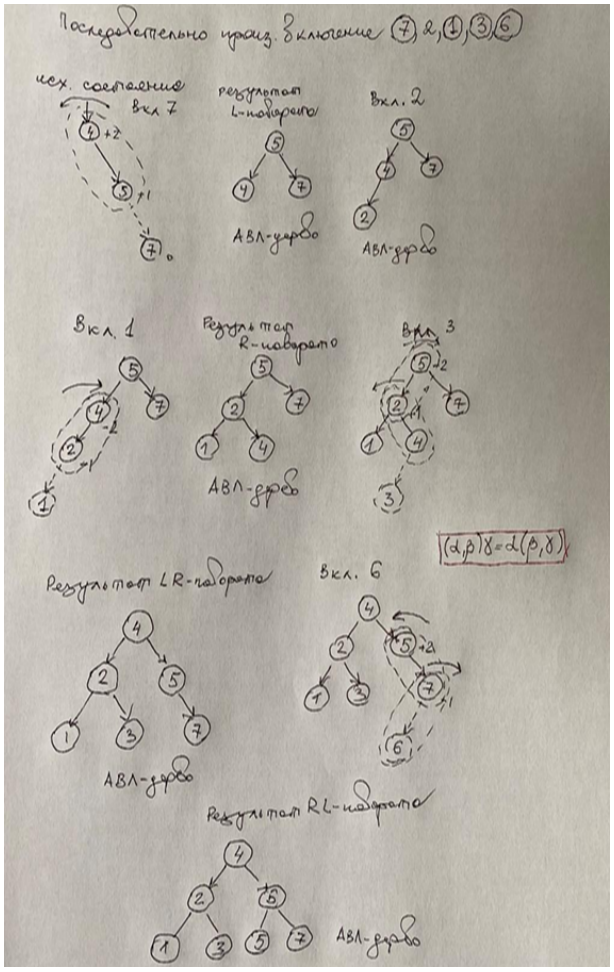
Необходимо учитывать следующее:

Если после включения показатели имеют одинаковый знак, то однократный поворот, при этом включение не будет оказывать влияние на другие участки дерева, если разный знак, то двукратный поворот трех узлов.

e.Пройти обратно по пути поиска и проверить показатель сбалансированности

Проиллюстрируем принцип работы алгоритма.

Рассмотрим бинарное дерево содержащее две вершины. Включение ключа 7 приводит к не сбалансированному дереву (к лин. списку).Его балансировка осуществляется L-поворотом. Последующие включения 2 и 1 приводит к несбалансированности поддерева с корнем 4 . С помощью R- поворота оно балансируется. Последующее включение ключа 3 нарушает критерий сбалансированности с корневой вершиной 5 .После чего балансировка достигается уже с помощью двойного LR-поворота. При любом следующем включении баланс может быть нарушен в вершине 5. И действительно включение вершины с ключом 6 приводит к четвертому типу балансировки RL-повороту.



Операция исключения в АВЛ-деревьях Принципиальная схема алгоритма исключения из сбалансированного дерева аналогична алгоритму исключения из бинарного дерева. Простые случаи удаления листовой вершины или вершины с одним потомком. Если из исключаемой вершины отходят два поддерева, то она заменяет самой правой вершиной его левого поддерева (или самой левой вершиной его правого поддерева). Но в случае, если дерево теряет свойство сбалансированности, то его восстановление либо по одному, либо по другому алгоритму (вращение с двумя или тремя узлами) с учетом правила: $(\alpha\beta)\gamma \sim \alpha(\beta\gamma)$.
Исключение любого элемента в худшем случае проводится $O(\log_2 N)$ (не O(h)). Исключение может потребовать поворотов во всех вершинах вдоль пути поиска (в отличие от операции включения) .При исключении поворот происходит в одном случае из пяти.

42. Оптимальные деревья поиска. Постановка задачи.

усть задано множество ключей $k_1, k_2, \ldots, k_n$ и соответствующие им вероятности p_i и q_i. Оптимальным деревом поиска называется такое дерево, структура которого обеспечивает минимальную "цену"(стоимость) среди всех возможных деревьев, построенных на заданном множестве ключей и вероятностей.

Вероятности обращения к узлам дерева (p_i и q_i) обычно оцениваются на основе экспериментов, в которых подсчитывается частота обращений. Вместо вероятностей (p_i, q_i) используются частоты обращений, которые обозначаются следующим образом:

a_i - частота случаев, когда аргумент поиска $x = k_i$, то есть поиск заканчивается в узле дерева. b_i - частота случаев, когда аргумент поиска находится между ключами, $k_i < x < k_{i+1}$, то есть поиск заканчивается в листьях дерева. При этом b_0 соответ-

ствует числу поисков, когда $x < k_1$, а b_n соответствует числу поисков, когда $x > k_n$. Обозначим через ω общее количество испытаний (общее количество поисков). Тогда вероятности p_i и q_i можно определить как:

$p_i = \frac{a_i}{\omega}$ (вероятность поиска ключа k_i)

$q_i = \frac{b_i}{\omega}$ (вероятность поиска значения между ключами k_i и k_{i+1})

где:

$\omega = \sum_{i=1}^{n} a_i + \sum_{i=0}^{n} b_i$ - вес дерева (общее количество поисков)

Стоимость дерева

Стоимостью дерева c называется общее значение количества операций сравнения, необходимых для выполнения всех поисков. Она вычисляется как:

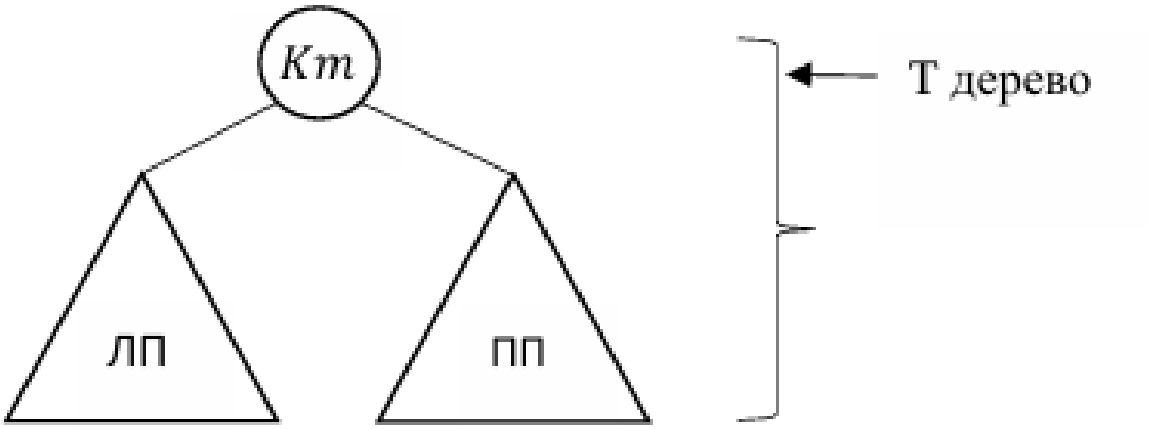
$c = \sum_{i=1}^{n} a_i h_i + \sum_{i=0}^{n} b_i h'_i$

где:

h_i - глубина узла, содержащего ключ k_i (количество ребер от корня до узла). h'_i - глубина листа, соответствующего диапазону между k_i и k_{i+1} (количество ребер от корня до листа).

43. Алгоритм Гильберта — Мура оптимального дерева поиска.

Пусть заданы неотрицательные частоты $\{a_i\}_{i=1}^{n}$ и $\{b_i\}_{i=0}^{n}$. Необходимо построить дерево бинарного поиска минимальной стоимости с узлами $\{k_i\}_{i=1}^{n}$ и листьями $\{d_i\}_{i=0}^{n}$. Рассмотрим алгоритм построения оптимального дерева бинарного поиска, предложенного Гильбертом и Муром, и основанный на следующем свойстве:



Пусть T — оптимальное дерево поиска для частот $b_0, a_1, b_1, a_2, \ldots, a_n, b_n$. Если элемент k_m является корнем дерева T, то левое поддерево T является оптимальным деревом для частот $b_0, a_1, b_1, a_2, \ldots, a_{m-1}, b_{m-1}$, а правое поддерево — оптимальным для частот $b_m, a_{m+1}, b_{m+1}, a_{m+2}, \ldots, a_n, b_n$ и т.д.

Алгоритм Гилберта и Мура сначала рассматривает дерево, которое не содержит узлов ($h = 0$), затем дерево с одним узлом ($h = 1$), а затем все большие деревья в порядке возрастания величины $j - i$. С учетом этого алгоритм следующий:

1. Для $i = 0, n$ выполнить // при $h = 0$

$$c_{ii} = 0;$$
$$r_{ii} = d_i;$$
$$\omega_{ii} = b_i;$$

2. Для $h = 1, n$ выполнить

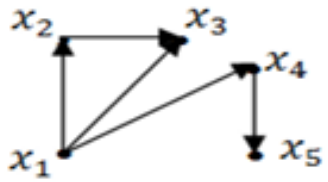
   Для $i = 0, n - h$ выполнить

$$j = i + h;$$
$$\omega_{ij} = \omega_{i,j-1} + a_j + b_j;$$
$$c_{ij} = \min\{c_{im-1} + c_{mj} \mid i < m \leq j\} + \omega_{ij};$$
$$r_{ij} = k_m;$$

44. Графы. Основные определения.
Граф – "двойка" G = (X, U), где X – множество элементов (вершин, узлов), а U представляет собой бинарное отношение на множестве X ($U \subseteq X * X$). Если |X| = n (конечно), то граф является конечным.

Пример:

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$X * X = \{(x_1, x_1), (x_1, x_2), \ldots, (x_5, x_5)\}$$



$$U=\{(x_1, x_2), (x_1, x_3), (x_2, x_3), (x_1, x_4), (x_4, x_5)\}$$

Элементы U называют либо дугами, либо ребрами.

Граф называется ориентированным, если пара вершин упорядочена: $(x_i, x_j) \neq (x_j, x_i)$. Граф называется неориентированным, если ребра неупорядоченные: $(x_i, x_j) = (x_j, x_i)$.

Обозначим $U \subseteq X \times X$, где X — множество вершин. Тогда:

$$\emptyset \leq U \leq X \times X$$

Если $U = \emptyset$, то это нуль-граф. Если $U = X \times X$, то такой граф называется полным графом.

Атрибуты орграфа

Атрибутами орграфа являются: дуга, путь, контур.

- **Путь** — это такая последовательность дуг, в которой конец каждой предыдущей дуги совпадает с началом последующей.
- **Контур** — конечный путь, у которого начальная вершина совпадает с конечной. Контур единичной длины называют петлей.

Атрибуты неорграфа

Атрибутами неорграфа являются, соответственно: ребро, цикл.

- **Цепь** — непрерывная последовательность ребер между парой вершин неориентированного графа.
- Неориентированный граф называют связным, если любые две его вершины можно соединить цепью. Если же граф несвязный, то его можно разбить на подграфы.

Связность графа

- **Слабая связность** — орграф заменяется неориентированным графом, который в свою очередь является связным.
- **Сильно связный** — если для любых двух вершин существует путь, идущий из x_i в x_j и из x_j в x_i.

45. Графы. Представление графа в памяти компьютера. Матрица смежности. Алгоритм Уоршелла.
Графический способ представления графа используется, если граф небольшой размерности. Также возможно использование матриц. Матрица легко описывается, и при анализе характеристик графа (путей, циклов, связности) можно использовать алгоритмы матричной алгебры.

Пусть задан граф $G = (X, U)$, $|X| = n$. Имеем матрицу A размерности $n \times n$, которая называется матрицей смежности, если элементы её определяются следующим образом:

$$A = \{a_{ij}\}_{i,j=1}^{n}, \quad \text{где} \quad a_{ij} = \begin{cases} 1, & \text{если } (x_i, x_j) \in U \\ 0, & \text{если } (x_i, x_j) \notin U \end{cases}$$

i-ая строка определяет дуги, исходящие из x_i, а j-ый столбец определяет дуги, входящие в x_j.

Алгоритм вычисления транзитивного замыкания (алгоритм Уоршелла)

Для $|X| = n$ выполняем:

1. Для $i = \overline{1, n}$ выполнить:

   Для $j = \overline{1, n}$ выполнить:
   Если $i = j$ или $(x_i, x_j) \in U$, то $P_{ij} = 1$ иначе $P_{ij} = 0$.

2. Для $k = \overline{1, n}$ выполнить:

   Для $i = \overline{1, n}$ выполнить:
   Для $j = \overline{1, n}$ выполнить:
   $P_{ij} = P_{ij} \vee (P_{ik} \wedge P_{kj})$

46. Графы. Алгоритм Флойда.
Для взвешенного орграфа этот алгоритм может быть модифицирован с целью получения матрицы содержащей длины кратчайших путей между вершинами. Заменим нулевые элементы P на очень большое число, а диагональные элементы на нуль. В этом случае следующим: рекурсивное выражение будет следующим:

$$P_{ij} = \min\{P_{ij}, P_{ik} + P_{kj}\}$$

Такой алгоритм называется **алгоритмом Флойда**.

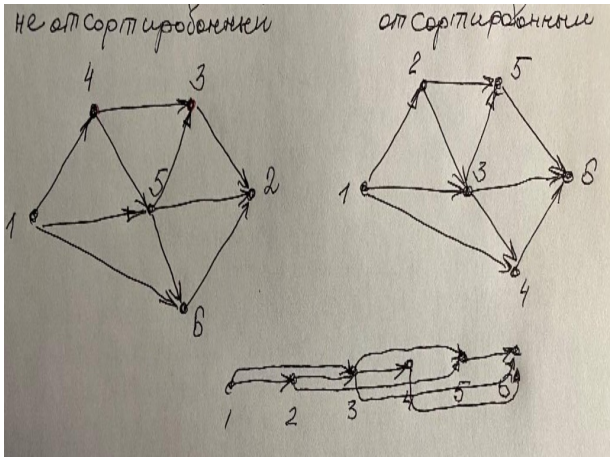
47. Графы. Алгоритм прохождения графа в ширину и в глубину.
47. Графы. Алгоритм прохождения графа в ширину и в глубину.
BFS (Breadth-First Search) исследует граф уровнями:
1. Начинаем с начальной вершины, добавляем её в очередь.
2. Извлекаем вершину из очереди, обрабатываем её и добавляем всех её непосещённых соседей в очередь.
3. Повторяем, пока очередь не пуста.
DFS (Depth-First Search) исследует граф, погружаясь в глубину:
1. Стартуем с начальной вершины, помечаем её как посещённую и углубляемся к первому непосещённому соседу.
2. Когда больше нет непосещённых соседей, возвращаемся на уровень вверх (стек/рекурсия).
3. Повторяем до посещения всех вершин.

48. Топологическая сортировка. Алгоритм. Примеры применения.
48. Топологическая сортировка. Алгоритм. Примеры применения. Топологическая сортировка (ТС) вершин орграфа G заключается в присвоении его вершинам чисел $1, 2, \ldots, |X|$, причем должно выполняться следующее условие: если имеется дуга (i, j), то $j > i$. Граф должен быть ацикличен, т.е. не иметь контуров. Например, отсортирован.

Топологическая сортировка может быть использована:

1. При организации толкового словаря. То есть задача сортировки состоит в том, как расположить линейно слова так, чтобы первичные слова располагались ранее относительно слов, в определении которых эти первичные слова использовались.

2. При организации программных систем. Здесь рассматривается совокупность взаимосвязанных процедур (одна процедура вызывает другую, та, в свою очередь, третью и т.д.). Топологическая сортировка предполагает описание процедур в таком порядке, чтобы вызываемые процедуры описывались после тех, которые их вызывают.



Идея его заключается в том, что осуществляется поиск такой вершины, из которой не выходит ни одна дуга. (такая вершина существует т.к. орграф ацикличный) Такой вершине присваивается число равное | x |. Указанный процесс повторяется т.е. новой вершины из которой не выходят дуги и присваивается число | x | - 1 и т.д. Так для орграфа на рис. будут удалены вершины 2,3,6,5,4,1 и заменены соответственно на 6,5,4,3,2,1 Для поиска вершины, из которой не выходят дуги, используют алгоритм прохождения графа в глубину. При этом необходим дополнительный массив LABEL(n) размерностью | X | для запи- си чисел топологически отсортированных вершин. Он имеет следующие свойство: если вершина не удалена, то значение LABEL(x)=0; если есть дуга (i, j), то LABEL(j) > LABEL(i).

```
∀x ∈ X выполнить
начало Num(x)=0; LABEL(x)=0 конец
C = | X | + 1;
∀x ∈ X выполнить если Num(x)=0 то DFSm(x);
  Процедура DFSm(v)
  начало
    Num(v)=1;
```

```
    ∀ t ∈ M(v) выполнить если Num(t)=0 то
DFSm(t);
    C = C – 1;
    LABEL(v) = C;
      конец
```

49. Внешняя сортировка. Особенности внешней сортировки. Алгоритм слияния.

Задача внешней сортировки возникает, когда число записей превышает объем основной памяти. Принято называть внешнюю сортировку сортировкой последовательных файлов Слияние – объединение двух ли более упорядоченных последовательностей (серий) в одну упорядоченную последовательность. Такой процесс – внутренняя сортировка с последующим внешним слиянием является основой алгоритмов внешней сортировки.

Алгоритм слияния:

Дано:

$$x_1 \le x_2 \le \cdots \le x_m$$
$$y_1 \le y_2 \le \cdots \le y_n$$

Результат:

$$z_1 \le z_2 \le \cdots \le z_{m+n}$$

1. Установить $i = 1, j = 1, k = 1$

2. Найти наименьший элемент

Если $x_i \le y_i$, то п. 3, иначе п. 5

3. Вывести x_i т.е.

$z_k \leftarrow x_i, i++; k++$

Если $i \le m$ идти п. 2

4. Вывести $y_j, \ldots, y_n$

$z_k, \ldots, z_{n+m} \leftarrow, y_j, \ldots, y_n$

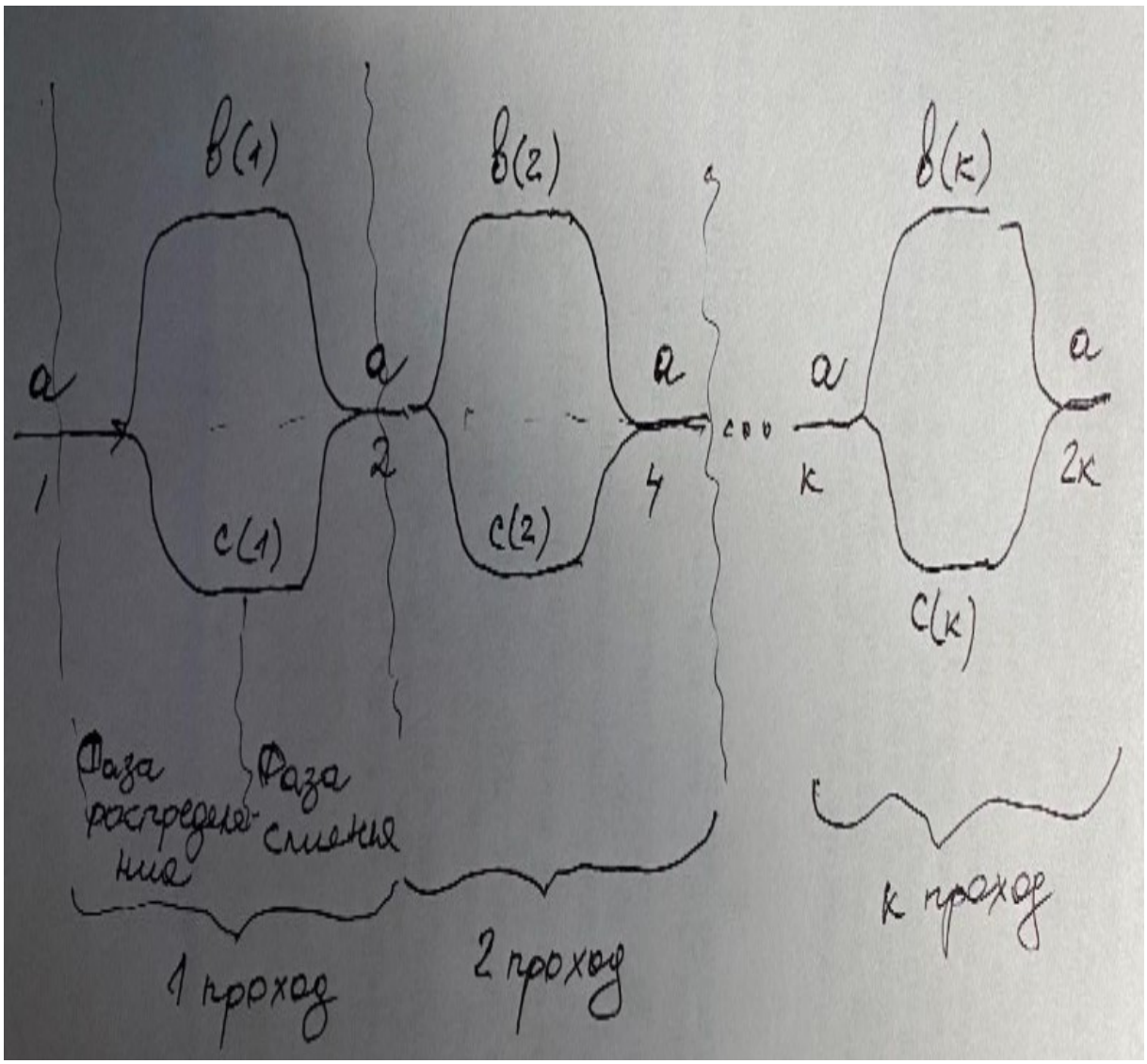
5. Вывести $y_j, z_k \leftarrow y_j \quad j++; k++$

Если $j \le n$ идти п. 2

6. Вывести $x_i, , \ldots, x_m$

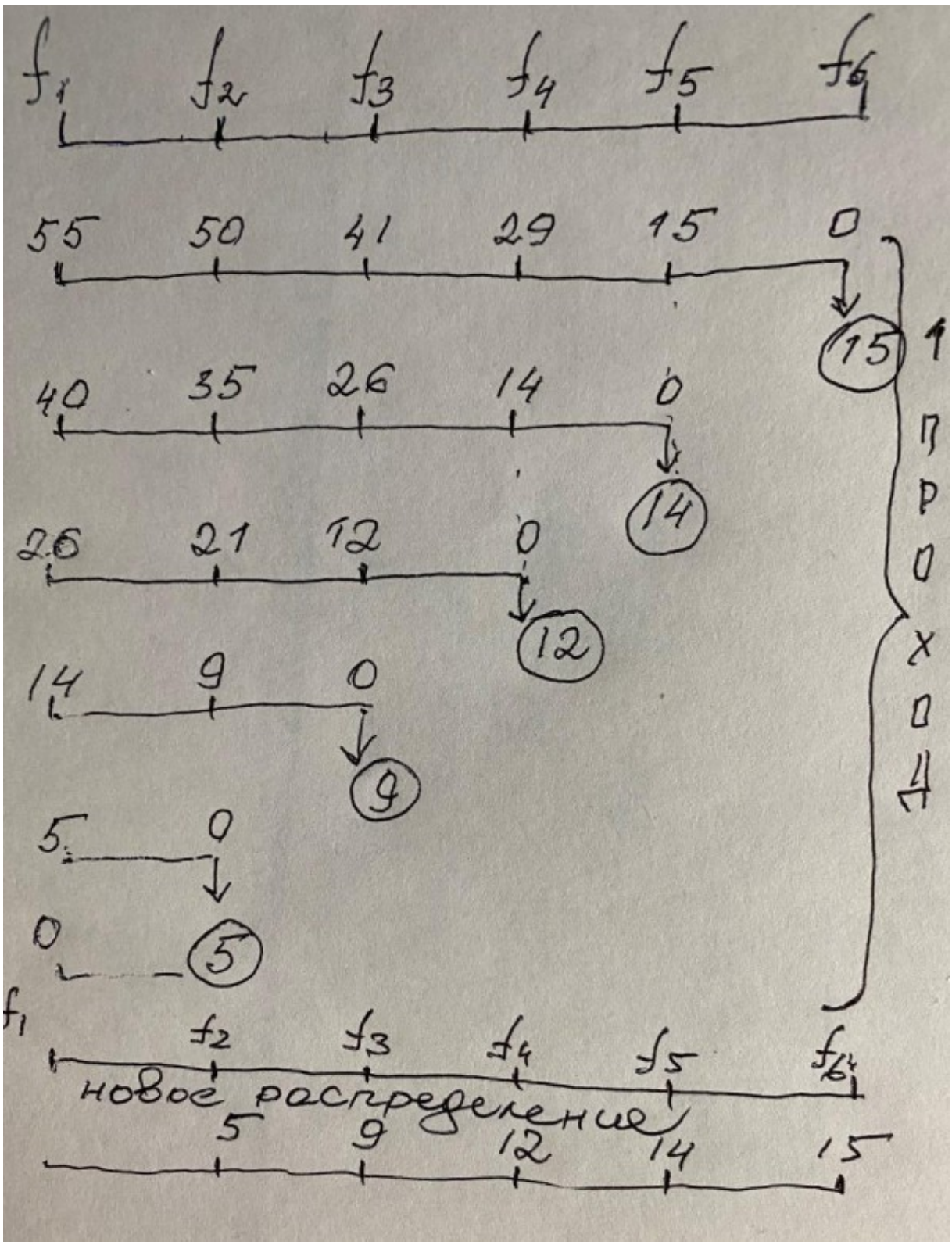
$z_k, \ldots, z_{n+m} \leftarrow x_i, , \ldots, x_m$

50. Внешняя сортировка. Алгоритм прямого слияния. Анализ.
Упорядоченную последовательность будем называть серией. Серия может быть разной длины и длина ее далее (для примера) равна 1. Один из простых алгоритмов на основе слияния называется алгоритмом прямого слияния. Графически его представить можно следующим образом:



1. Последовательность a разбивается на 2 половины b и c.
2. Проводим слияние, при этом одиночные элементы образуют упорядоченные пары.
3. Полученные последовательности под именем a вновь обрабатываются как указано в пунктах 1 и 2, при этом упорядоченные пары переходят в такие же четверки.
4. Повторяем предыдущие шаги каждый раз, удваивая длину, до тех пор, пока не будет упорядочена вся последовательность.

51. Внешняя сортировка Многофазная сортировка Анализ.
Это очередное усовершенствование. Данные размещаются на N-1 файл а один свободен. Слияние данных на этот файл происходит до тех пор пока один из N-1 файл не станет пустой. Затем слияние также происходит на освободившийся файл и т.д. Проиллюстри- руем эту сортировку на примере с тремя файлами:



Для того чтобы определить, какое исходное распределение серий требуется, необходимо построить таблицу с конца. При анализе таблицы получим следующие соотношения:

$$a_2^{l+1} = a_1^l$$

$$a_1^{l+1} = a_1^l + a_2^l = a_1^l + a_1^{l-1}$$

Сделаем следующие замены:

$$a_1^l = f_i, \quad a_1^{l-1} = f_{i-1}$$

получим:

$$f_{i+1} = f_i + f_{i-1}, \quad f_1 = 1, \quad f_0 = 0$$

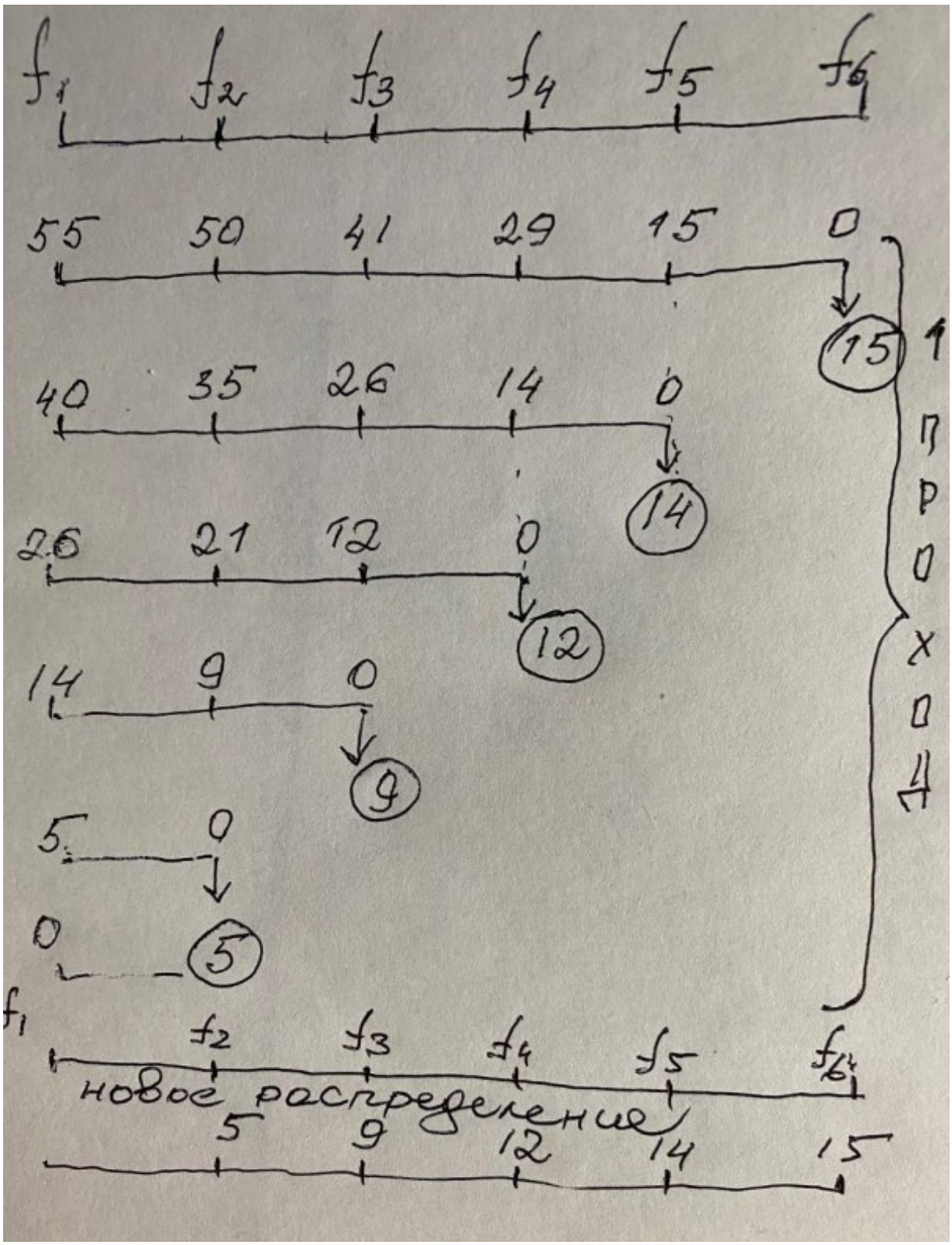
Это рекуррентное соотношение определяет так называемые числа Фибоначчи:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Таким образом, необходимо, чтобы числа начальных серий двух входных последовательностей или файлов были членами последовательности чисел Фибоначчи.

52. Внешняя сортировка Каскадная сортировка. Анализ.

Каскадная сортировка подобна многофазной так как начинается с некоторого распределения по файлам, но правила слияния другие. Рассмотрим алгоритм на примере если N=6. Шаг 1: В начале осуществляется слияние первых пяти файлов в шестой, при этом один из файлов освобождается. На освободившийся файл - 4-х путевое слияние тем же способом, затем 3-х, 2-х и т.д. Шаг 2: Над полученным распределением опять осуществляется слияние как на первом шаге и так продолжать до тех пор, пока не останется один файл. Рассмотрим пример: N=6



Запишем рекуррентные соотношения, применение которых позволит получить необходимые начальные распределения:

$$a_{i+1} = a_i + b_i + c_i + d_i + e_i$$

Начальные условия:

$$a_0 = 1, \quad b_0 = 0, \quad c_0 = 0, \quad d_0 = 0, \quad e_0 = 0$$

$$b_{i+1} = a_i + b_i + c_i + d_i$$

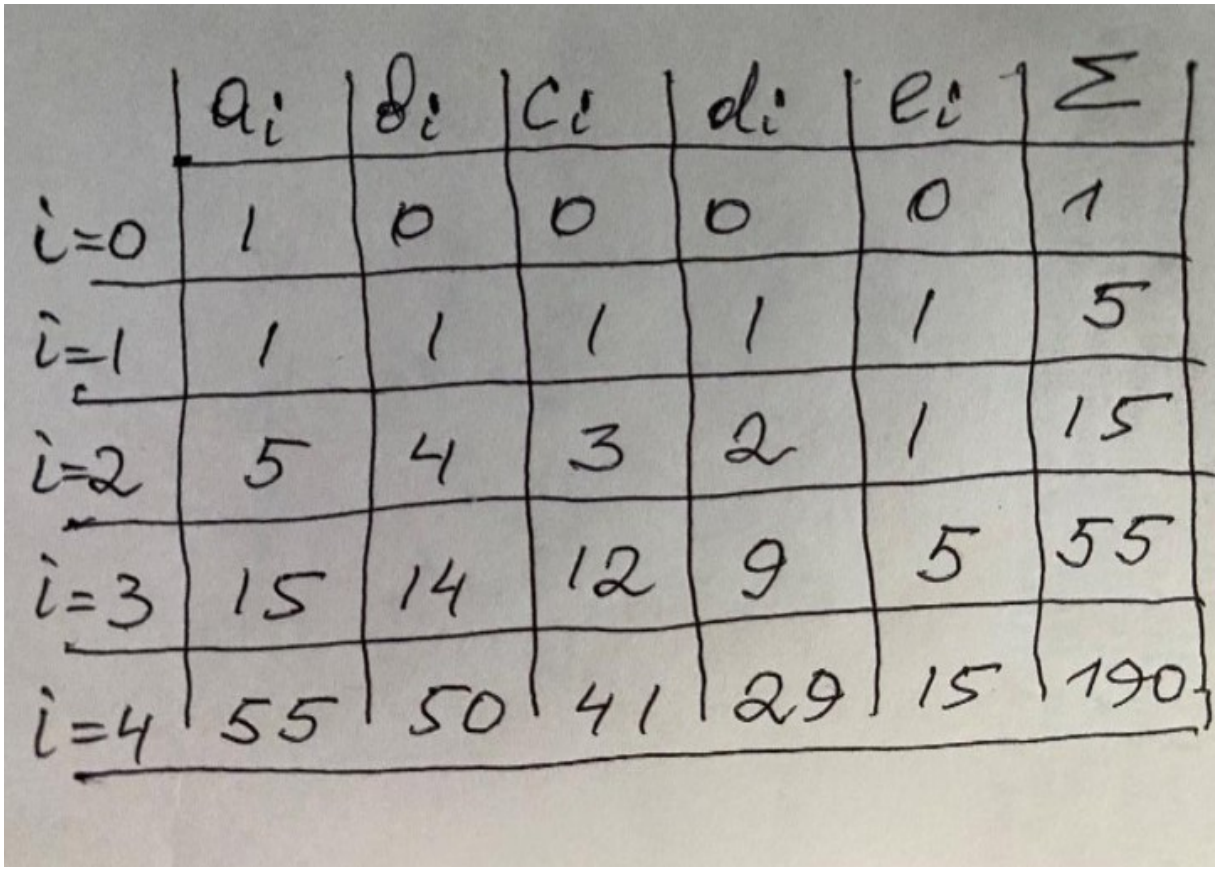
$$c_{i+1} = a_i + b_i + c_i$$

$$d_{i+1} = a_i + b_i$$

$$e_{i+1} = a_i$$

Т.к. в начале используются все файлы, а затем с каждым этапом количество файлов сокращается на единицу.

В соответствии с рекуррентными выражениями составлена таблица необходимых начальных распределений, если $N = 6$.



	a_i	b_i	c_i	d_i	e_i	Σ
$i=0$	1	0	0	0	0	1
$i=1$	1	1	1	1	1	5
$i=2$	5	4	3	2	1	15
$i=3$	15	14	12	9	5	55
$i=4$	55	50	41	29	15	190

Таким образом при i=4 получаем распределение, которое использовалось в примере т.е. 55,50,41,29,15. Экспериментально подтверждается, что при больших n данная сортировка лучше многофазной.

53. B-деревья. Основные определения. Алгоритм поиска.

B-дерево обладает следующими свойствами:

1. Каждая страница содержит не более $2n$ элементов (ключей).

2. Каждая страница, кроме корневой, содержит не менее n элементов.

3. Каждая страница либо представляет собой лист, т. е. не имеет потомков, либо имеет $m + 1$ потомков, где m – текущее число ключей на этой странице, $n \leq m \leq 2n$ (кроме корневой).

4. Все страницы – листья находятся на одном уровне.

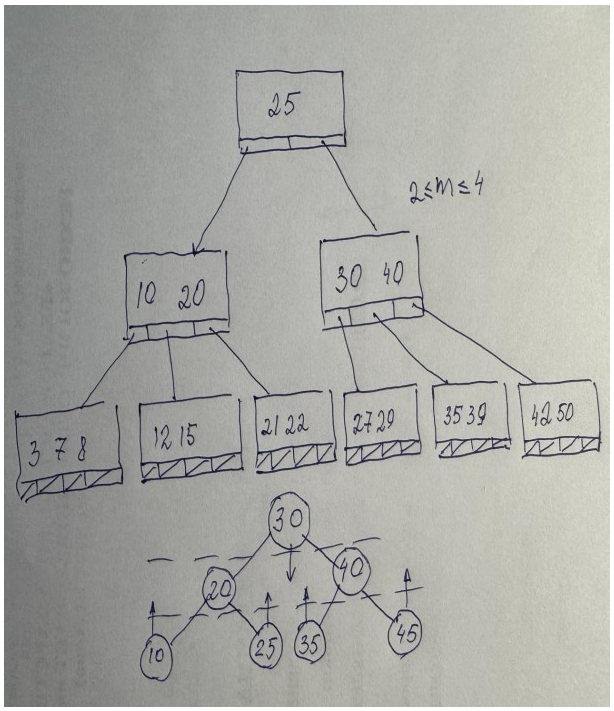
Достоинства B-дерева

Достоинства B-дерева:

1. Так как B-дерево является сильно ветвистым, то его высота не велика, как и число обращений к внешней памяти.

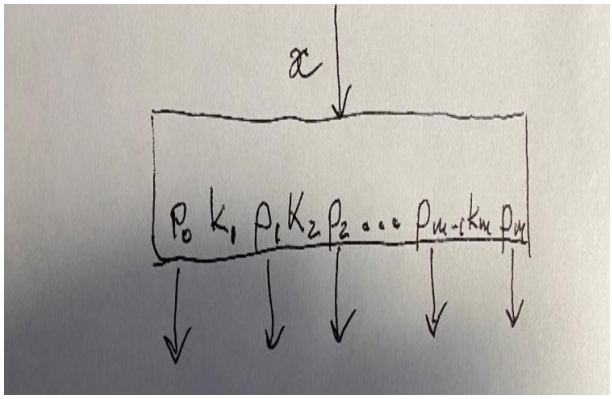
2. При включении и исключении ключей изменения ограничены, как правило, одним узлом и не затрагивают все страницы дерева поиска, поскольку во всех страницах есть свободные поля. Максимальное количество элементов $2n$.

Пример B-дерева



Алгоритм поиска Предположим, что страница уже считана в оперативную память. Она представляет собой линейную последовательность ключей.

$$k_1 < k_2 < \ldots < k_m$$



Пусть p_i, $i = \overline{0, m}$ — указатели, и k_i, $i = \overline{1, m}$ — ключи. Можно воспользоваться методами поиска среди ключей:

$$k_1 < k_2 < \ldots < k_m$$

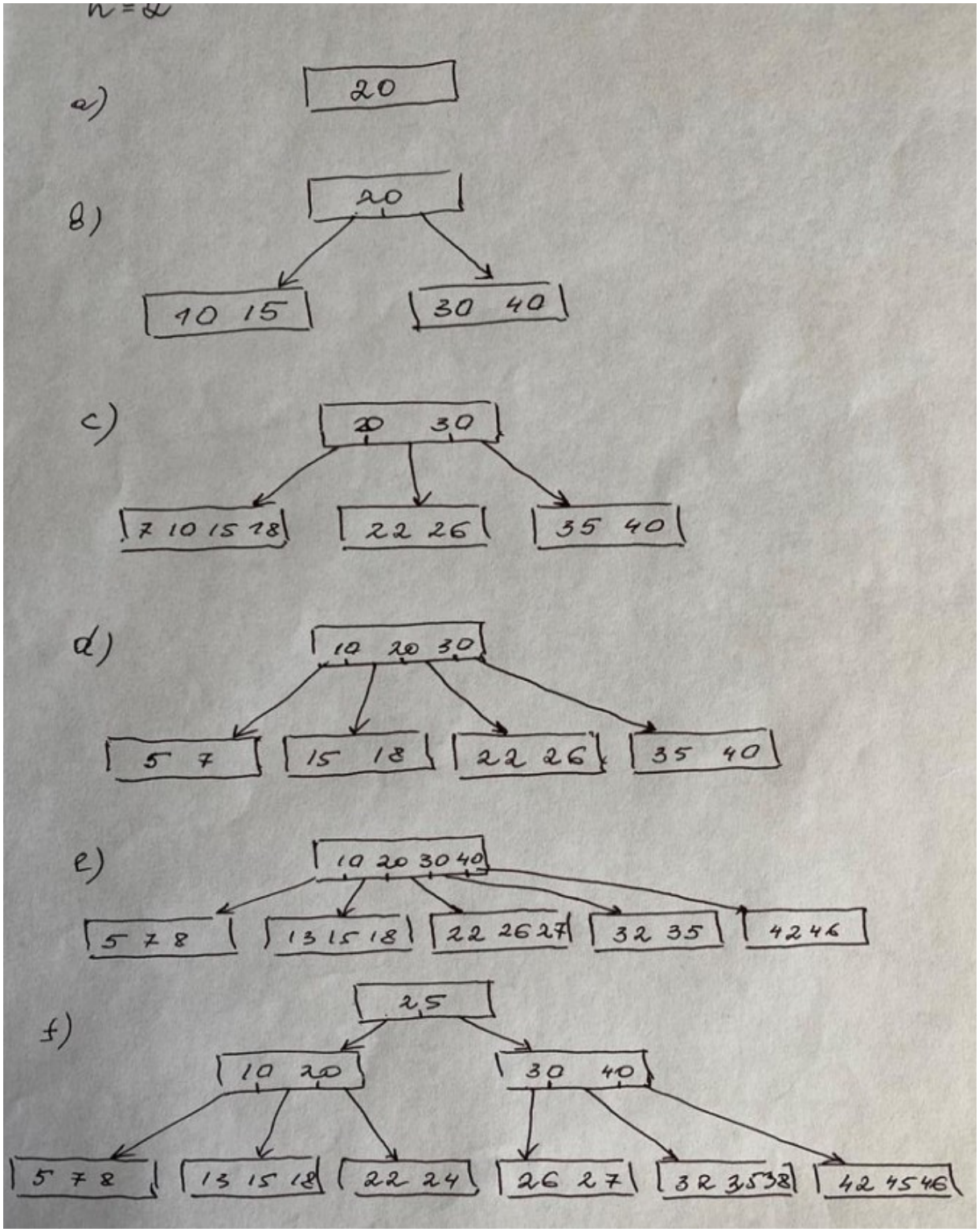
Для поиска при больших m может быть использован бинарный поиск. Если поиск неудачен, то мы попадаем в одну из следующих ситуаций:

1. $k_i < x < k_{i+1}, \quad 1 \leq i < m$ — поиск продолжается по указателю p_i
2. $x < k_m$ — поиск продолжается по указателю p_0
3. $x > k_m$ — поиск продолжается по указателю p_m

Если указатель p_i равен NULL, то в дереве не существует элемента с ключом x, и поиск заканчивается. Сложность поиска составляет $O(\log n)$.

54. B-деревья. Алгоритм включения в B-дерево. Пример.

Операция проведена должна так, чтобы B-дерево сохранило свои свойства. Шаг 1: начинаем с корня последовательно просматриваем страницы дерева (в соответ- ствии с алгоритмом поиска) пока не будет достигнут соответствующий лист-страница; Шаг 2: если лист заполнен не полностью произвести включение ключа в соответствующее место списка. Конец алгоритма; Шаг 3: если в этом месте свободного места нет, то в результате включения число ключей станет 2n+1,возникает ситуация переполнения. При переполнении n+1-ый (ме- дианный) ключ удалить из страницы создать новую страницу(лист) и переместить в него n ключей списка из страницы в котором возникло переполнение. Удаленный ключ включить на один уровень вверх в “родительскую” страницу; Т.о. при переполнении происходит расщепление страницы образуются две новых вместо одного и ключ переноса. Шаг 4: если включение элемента в “родительскую” страницу приводит к её перепол- нению, то эта страница тоже расщепляется в соответствии с шагом 3; Шаг 5: если ключ переноса достигает корня (принцип “домино”) и корень после вклю- чения так же окажется переполненным, то его расщепляют, что приводит к увеличению высоты дерева на единицу и создание нового корня, куда помещают ключ переноса. Так что деревья растут от листьев к корню. (не так как при построении бинарного древа) Это и есть управление ростом. Рассмотрим выполнение алгоритма, если включение ключей осуществляется в следую- щим порядке: 20,40,10,30,15;35,7,26,18,22;5;42,13,46,27,8,32;38,24,45 ,25;



55. B-деревья. Алгоритм исключения из B-дерева.

После исключения свойства B-дерева должны быть сохранены. Шаг 1: найти страницу содержащую удаляемый ключ (используя алгоритм поиска); Шаг 2: если удаляемый ключ находится в листе-странице, то возможны три ситуации: а) если найден лист содержащий не менее n+1 ключей, то ключ просто удаляется. Конец операции. б) если в найденном листе содержится ровно n ключей, но имеется соседний лист, который содержит более n ключей, то для сохранения свойств B-дерева достаточно вы- полнить перемещение ключей из соседнего листа. Для этого необходимо: -удалить ключ разделяющий страницы ключей соседних листьев, из страницы “родительско-

го” и включить его в лист, содержащий ключей меньше n; -на место удаленного ключа поместить крайний (левый или правый) ключ из соседнего листа. Комментарий. c) если найденный и соседний с ним лист содержит по n ключей, то произво- дят их объединение (конкатенация), списки ключей соседних листьев объединяются и остается один лист, а другой уничтожается. При этом один из ключей “родительского” уровня, разделявших соседние листья-страницы, является лишним и его необходимо переместить вниз. Шаг 3: если удаляемый ключ находится в не листовой странице. а) удаляем заданный ключ и помещаем на его место ключ предшествующий по величине перед удаленным. б) последующие действия будут аналогичны удалению ключа из листа (см. шаг 2)

Рассмотрим выполнение алгоритма, если исключение ключей осуществляется в сле- дующим порядке: 25,45,24;38,32;8,27,46,13,42;5,22,18,26;7,35,15