Примечание:

Красным цветом написаны вопросы и та часть ответов на вопросы на счет правильности которых есть сомнения.

Черным цветом написаны ответы на вопросы.

Фиолетовым цветом написаны алгоритмы и псевдокоды в ответах на вопросы.

Все что синим цветом – это дополнительная информация (не обращайте внимания)

Вопросы к экзамену по дисциплине «Алгоритмы и стуктуры данных»

3-й семестр

Оглавление

1. Алгоритмическая сложность. Оценка роста функции. Оценка сверху, снизу, в среднем.

Алгоритмическая (Вычислительная) сложность – это параметр для сравнивнения быстродействия алгоритмов, чётко описывающее их поведение (время исполнения и объём необходимой памяти) в зависимости от размера входных данных.

Временная сложность алгоритма — это функция от размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

Оценка роста функции (оценка слоности) – это оценка алгоритма (по времени выполнения или по памяти) некоторой функцией от n (количество входной информации), характеризующего изменение времени работы алгоритма с увеличением параметра n.

Оценка сверху – это оценка работы алгоритма в худшем случаи. Например для сортировки – обратно сортированная последовательность.

Оценка снизу – это оценка работы алгоритма в лучшем случаи. Например для сортировки – уже отсортированная последовательность.

Оценка в среднем – это оценка работы алгоритма в среднем случаи. Например для сортировки – частично отсортированная последовательность.

2. Алгоритмы поиска. Линейный поиск. $O(n)$

Алгоритм поиска – последовательность операций сравнения, с искомым экземпляром данных, где очередные элементы в структура данных (где выполяется поиск) выбираюстя исходя от алгоритма поиска. Если после завершения алгоритма искомым экземпляром данных не был обнаружен в структуре данных, алгоритм завершает работу.

Линейный, последовательный поиск — алгоритм нахождения заданного значения в произвольной структуре данных. Данный алгоритм является простейшим алгоритмом поиска и работает за линейное время – с увеличением данных время выполнения увеличивается прямо-пропорционально. "Метод поиска в лоб" – перебирать все элементы в структуре данных начиная с первого (или с последнего) элемента и сравнить с искомым значением.

Псевдо код:

```
ЦИКЛ пока не закончились элементы, для каждого элемента
    ЕСЛИ очередной элемент совпадает с искомым, ТО
        нашли, выходим из цикла
    Конец ЕСЛИ
Конец ЦИКЛА
```

3. Алгоритмы поиска. Бинарный поиск. $O(log\ n)$

Алгоритм поиска – последовательность операций сравнения, с искомым экземпляром данных, где очередные элементы в структура данных (где выполяется поиск) выбираюстя исходя от алгоритма поиска. Если после завершения алгоритма искомым экземпляром данных не был обнаружен в структуре данных, алгоритм завершает работу.

Двоичный (бинарный) поиск – классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. В каждой итерации работы алгоритма в массиве данных выбирается средний элемент и сравнивается с

искомым элементом, исходя из того будет элемент болше или меньше искомого, поиск продолжается в соответствующей половине.

Псевдокод:

Пусть есть некий отсортированный массив. Рекурсивно выполнить.

```
ФУНКЦИЯ ПОИСКА (МАССИВ, НАЧАЛО, КОНЕЦ)
    вычислить СРЕДНИЙ элемент,   // средний = (int)(НАЧАЛО+КОНЕЦ / 2)
    сравнить с искомым
    ЕСЛИ ИСКОМЫЙ < СРДНЕГО
        ФУНКЦИЯ ПОИСКА (МАССИВ, НАЧАЛО, СРЕДНИЙ)
    Конец ЕСЛИ
    ЕСЛИ ИСКОМЫЙ > СРДНЕГО
        ФУНКЦИЯ ПОИСКА (МАССИВ, СРЕДНИЙ, КОНЕЦ)
    Конец ЕСЛИ
    ИНАЧЕ
        нашли
    Конец ИНАЧЕ
Конец ФУНКЦИИ
```

4. Поиск подстроки в строке. Простой поиск.

Поиск подстроки в строке (String searching algorithm) — класс алгоритмов над строками, которые позволяют найти паттерн (pattern) в тексте (text). Простой поиск подстроки в строке из себя представляет "примитивный метод поиска в лоб" (brute-force, наивный алгоритм): на каждом шагу берется очередной кусок из строки (длиной искомой строки) и сравнивается, если результат стравнения false то двигаемся на одну позицию вперед и снова сравниваем и так до конца строки.

$O(pattern * (string - pattern))$ – средний случай $O(n^2)$ – худший случай

|строка| -- длина строки |подстрока| -- длина подстроки

```
find(String подстрока, String строка) {
    failed:
    for (int i = 0; i < |строка| - |подстрока|; i++) {
        for (int j = 0; j < |подстрока|; j++) {
            if (строка[i + j] != подстрока[j]) {
                goto failed;
            }
        }
        return i;  // нашли, вернули начальную позицию подстроки в строке
    }
    return -1; // не нашли
}
```

5. Поиск подстроки в строке. Алгоритм Кнута-Мориса-Прата.

Поиск подстроки в строке (String searching algorithm) — класс алгоритмов над строками, которые позволяют найти паттерн (pattern) в тексте (text).

Алгоритм Кнута — Морриса — Пратта (КМП-алгоритм) — эффективный алгоритм, осуществляющий поиск подстроки в строке. Время работы алгоритма линейно зависит от объёма входных данных, то есть разработать асимптотически более эффективный алгоритм невозможно.

Дана цепочка T и образец P. Требуется найти все позиции, начиная с которых P входит в T.

Построим строку $S = P\#T$, где # — любой символ, не входящий в алфавит P и T. Посчитаем на ней значение префикс-функции p. Благодаря разделительному символу #, выполняется $\forall i: p[i] \leqslant |P|$. Заметим, что по определению префикс-функции при $i > |P|$ и $p[i] = |P|$ подстроки длины P, начинающиеся с позиций 0 и $i - |P| + 1$, совпадают. Соберем все такие позиции $i - |P| + 1$ строки S, вычтем из каждой позиции $|P| + 1$, это и будет ответ. Другими словами, если в какой-то позиции i выполняется условие $p[i] = |P|$, то в этой позиции начинается очередное вхождение образца в цепочку.

$O(string + pattern)$ – средний и худший случай

```
int find(String p, String s) {
        String c = p + "#" + s;
        int[] prefix = new int[c.length()];
        for (int i = 1; i < c.length(); i++) {
            int j = prefix[i - 1];
            while (j > 0 && c[i] != c[j]) {
                j = prefix[j - 1];
            }
            if (c[i] == c[j]) {
                prefix[i] = j + 1;
            }
            if (prefix[i] == p.length()) {
                return i - 2 * p.length();
            }
        }

        return -1;
    }
```

++

Пусть дан текст в виде массива T[n] и образец (то что ищем) P[1…m]; m≤n. Символы принадлежат алфавиту $\sum = \{0,1\}$ в бинарном коде; $\sum = \{a, b, c, \dots, z\}$ в текстовом файле.

P входит в T со сдвигом S, если T[S+1…S+m]=P[1…m]; S от 0 до n-m, такой сдвиг называется допустимым.

#T=abcabaabc;

P=abaa;

P⟶abaa, сдвиг S=3.

Пусть дана строка символов X[1…n], тогда для любой пары i, j; $1 \leq i \leq j \leq n$ определяем подстроку X[i…j]=X[i] X[i+1]…X[j].

Будем говорить, что подстрока X[i…j], начинается с позиции i и её длина равна j-i+1, если величина меньше чем n, то подстрока называется собственной подстрокой строки X.

ε(сигма) X=$\varepsilon X \varepsilon$

Для произвольного целого j от 0 до n подстрока X[1…j] называется префиксом подстроки.

Если j<n собственный префикс подстроки X.

Для произвольного целого i от 1 до n+1 подстрока X[i…n] называется суффиксом строки X. Если i>1 то подстрока называется собственным суффиксом X.

Например, X=abaab

-префиксы a, ab, aba, abaa, abaab;

-суфиксы b, ab, aab, baab, abaab=X.

Алгоритм поиска образца:

Алгоритм NSM (T[1…n], P[1…m])

1) for S=0 to n-m do;
2) if P[1…m]=T[S+1…S+m] then;
3) printf “подстрока P входит в T”.

Вычислительная сложность: O((n-m+1)m).

Алгоритм KMP для поиска:

Префикс функция ассоциирующаяся с образом P несет информацию, где в P встречаются префиксы строки, использование этой информации не считывает заведомо не подходящие сдвиги.

#T=bacbababababcbab

P=ababaca, сдвиг = 4

T[S+1…S+q]=P[1…q]

Некоторые последующие сдвиги будут недостимыми.

```
CPF(p[1..m])
1.  s[1]←0
2.  for q=2 to m do
3.  k←s[q-1]
4.  while (P[q]≠P[k+1]) and (k>0) do
5.  k←s[k]
6.  if (P[q]≠P[k+1] and (k=0) then s[q] ←0
7.  else s[q] ←k+1
8.  end for
9.  return s
KMP(T[1..n],P[1..m])
1.  s←CPF(P)
2.  q←0
3.  for k=1 to n do
4.  while T[k]≠P
5.  q←s[q]
6.  if T[k]=P[q+1] then
7.  q←q+1
8.  if q=m then
9.  printf “Образец входит со сдвигом”,k-m
10. q←s[q]
11. end for
```

6. Поиск подстроки в строке. Алгоритм Бойера-Мура.

Поиск подстроки в строке (String searching algorithm) — класс алгоритмов над строками, которые позволяют найти паттерн (pattern) в тексте (text).

Алгоритм Бойера-Мура считается наиболее эффективным алгоритмом поиска шаблонов в стандартных приложениях и командах, таких как Ctrl+F в браузерах и текстовых редакторах.

Алгоритм сравнивает символы шаблона x справа налево, начиная с самого правого, один за другим с символами исходной строки y. Если символы совпадают, производится сравнение предпоследнего символа шаблона и так до конца. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. В случае несовпадения какого-либо символа (или полного совпадения всего шаблона) он использует две предварительно вычисляемых эвристических функций, чтобы сдвинуть позицию для начала сравнения вправо.

Таким образом для сдвига позиции начала сравнения алгоритм Бойера-Мура выбирает между двумя функциями, называемыми эвристиками хорошего суффикса и плохого символа (иногда они называются эвристиками совпавшего суффикса и стоп-символа). Так как функции эвристические, то выбор между ними простой — ищется такое итоговое значение, чтобы мы не проверяли максимальное число позиций и при этом нашли все подстроки равные шаблону.

$O(n)$ – средний случай
$O(string * pattern)$ – худший случай

Code:
https://github.com/Markoutte/sandbox/tree/master/src/main/java/me/markoutte/sandbox/algorithms/strings

++++

Использование алгоритма Кнута-Морисса-Пратта в большинстве случаев поиска в обычных текстах весьма незначителен. Метод же, предложенный Р. Боуером и Д. Муром в 1975 г., улучшает обработку самого плохого случая.

БМ-поиск основывается на необычном соображении сравнение символов начинается с конца слова, а не с начала. Как и в случае КМП-поиска, слово перед фактическим поиском трансформируется в некоторую таблицу. Пусть для каждого символа x из алфавита величина dx расстояние от самого правого в слове вхождения x до правого конца слова. Представим себе, что обнаружено расхождение между словом и текстом. В этом случае слово сразу же можно сдвинуть вправо на dpM-1 позиций, т.е. на число позиций, скорее всего большее единицы. Если несовпадающий символ текста в слове вообще не встречается, то сдвиг становится даже больше, а именно сдвигать можно на длину всего слова.

Например,
T=ABCABCABFABCABD
P=ABCABD (сравниваем то что подчеркнуто, идем с конца, не совпало D с C, сдвиг =3, чтоб C=C)
ABCABD (не совпало D и F, так как F нет в образце)
ABCABD(полное совпадение слово найдено)

```
CLOF(p[1..m], sum)  sum это значок суммы
1.      for all a   sum do
2.      l[a]←0
3.      for k=1 to m do
4.      l[P[k]] ←k
5.      return l

CGSF(p[1..m])
1.      s←CPF(P)
2.      P’ ← обращение строки P
3.      S’ ← CPF(P’)
4.      For j=0 to m do
5.      Y[j] ←m-s[m]
6.      For k=1 to m do
```

```
7.      J ← m-s'[k]
8.      Y[j] ← min(y[j],k-s'[k])
9.      End for
10.     Return y

BM(T[1..n],P[1..m])
1.      L ← CLOF(P,m,sum)
2.      Y ← CGSF(p,m)
3.      S ← 0
4.      While S<=n-m do
5.      k←m
6.      while (k>0) and (P[k]=T[S+k]) do
7.      k←k-1
8.      if k=0 then
9.      printf "Образец со сдвигом",S
10.     s←s+y[0]
11.     else s←s+max(y[k],k-y[T[s+k]])
12.     end while
```

7. Линейные структуры данных. Списки. Динамический массив.

Линейные структуры — это упорядоченные структуры, в которых адрес элемента однозначно определяется его номером.

Линейных структуры данных обладают следующими свойствами:

- Каждый элемент имеет не более 1 предшественника
- Два разных элемента не могут иметь одинакового последователя

К линейным структурам данным можно отнести:

- Массивы
- Динамические массивы
- Связный список
- Стек
- Очередь
- Дек
- Хэш-таблица

Связный список - это разновидность линейных структур данных, представляющая собой последовательность элементов, обычно отсортированную в соответствии с заданным правилом. Последовательность может содержать любое количество элементов, поскольку при создании списка используется динамическое распределение памяти. Каждый элемент связного списка представляет собой отдельный объект, содержащий поле для хранения информации и указатель на следующий элемент списка (а в случае двусвязного списка в объекте хранится также указатель на предыдущий элемент).

Массив – одна из простейших и наиболее широко применяемых в компьютерных программах линейных структур данных. В любом языке программирования массивы имеют несколько общих свойств:

- Содержимое массива хранится в непрерывной области памяти.

- Все элементы массива имеют одинаковый тип; поэтому массивы называют однородными структурами данных.
- Существует прямой доступ к элементам массива.

8. Линейные структуры данных. Списки. Связный и двусвязный списки.

См. 7 вопрос.

9. Линейные структуры данных. Очереди. Кольцевые очереди.

См. 7 вопрос. +

Очередь – линейная структура данных, удовлетворяющая принципу FIFO (первый пришел – первый ушел)

- Поддерживает добавление элемента в конец,
- Поддерживает доступ к первому и последнему элементу,
- Поддерживает удаление первого элемента
- Не поддерживает итераторы

Кольцевая очередь - это идентичная очереди структура данных, с одним отличием, после последнего элемента сразу же снова идет первый. Это можно организовать с помощью указателей (в случае списка), и с помощью операции остатка от деления (%) (в случае массивов).

10. Линейные структуры данных. Стеки. Деки. Использование стека для решения задачи о правильных скобках.

См. 7 вопрос. +

Дек (deque — double ended queue) — структура данных, представляющая из себя список элементов, в которой добавление новых элементов и удаление существующих производится с обоих концов. Эта структура поддерживает как FIFO, так и LIFO, поэтому на ней можно реализовать как стек, так и очередь.

Стек -- линейная структура данных, удовлетворяющая принципу FILO(первый пришел – последний ушел)

- Поддерживает добавление элемента в конец,
- Поддерживает доступ к последнему элементу.
- Поддерживает удаление последнего элемента

Отличным примером использования стека для решения задачи о правильных скобках является обратная польская нотация.

Алгоритм (упращенной реализаиии):

- Пока есть ещё символы для чтения:
- Читаем очередной символ.
- Если символ является открывающей скобкой, помещаем его в стек.
- Если символ является закрывающей скобкой:

До тех пор, пока верхним элементом стека не станет открывающая скобка, выталкиваем элементы из стека в выходную строку. При этом открывающая скобка удаляется из стека, но в выходную строку не добавляется. Если стек закончился раньше, чем мы встретили открывающую скобку, это означает, что в выражении либо неверно поставлен разделитель, либо не согласованы скобки.

- Когда входная строка закончилась, выталкиваем все символы из стека в выходную строку. В стеке должны были остаться только символы операций; если это не так, значит в выражении не согласованы скобки.

11. Формы представления выражений. Польская и обратная польская нотации. Алгоритм трансформации инфиксной записи в постфиксную.

Существуют три формы представления выражений: Инфиксные, префиксные и постфиксные.

Инфиксное выражение – это самое обычное и привычное выражение для восприятия человека, когда оператор находится между двумя опрандами (например "A+ C").

Префиксная запись выражения требует, чтобы все операторы предшествовали двум операндам, с которыми они работают. Постфиксная, в свою очередь, требует, чтобы операторы шли после соответствующих операндов.

Пример:

Инфиксная запись	Префиксная запись	Постфиксная запись
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Польская нотация (запись), (префиксная нотация (запись)), это форма записи логических, арифметических и алгебраических выражений. Характерная черта такой записи — оператор располагается слева от операндов.

Обра́тная по́льская запись (Постфиксная запись) — форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Алгоритм:

- Пока есть ещё символы для чтения:
- Читаем очередной символ.
- Если символ является числом или постфиксной функцией (например, ! — факториал), добавляем его к выходной строке.
- Если символ является префиксной функцией (например, sin — синус), помещаем его в стек.
- Если символ является открывающей скобкой, помещаем его в стек.
- Если символ является закрывающей скобкой:

До тех пор, пока верхним элементом стека не станет открывающая скобка, выталкиваем элементы из стека в выходную строку. При этом открывающая скобка удаляется из стека, но в выходную строку не добавляется. Если стек закончился раньше, чем мы встретили открывающую скобку, это означает, что в выражении либо неверно поставлен разделитель, либо не согласованы скобки.

- Если существуют разные виды скобок, появление непарной скобки также свидетельствует об ошибке. Если какие-то скобки одновременно являются функциями (например, [x] — целая часть), добавляем к выходной строке символ этой функции.
- Если символ является бинарной операцией ***o1***, тогда:

1) пока на вершине стека префиксная функция…

… ИЛИ операция на вершине стека приоритетнее ***o1***

… ИЛИ операция на вершине стека левоассоциативная с приоритетом как у ***o1***

… выталкиваем верхний элемент стека в выходную строку;

2) помещаем операцию ***o1*** в стек.

• Когда входная строка закончилась, выталкиваем все символы из стека в выходную строку. В стеке должны были остаться только символы операций; если это не так, значит в выражении не согласованы скобки.

12. Деревья. Дерево поиска и бинарное дерево поиска. Основные понятия.

Дерево — структура данных, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы.

Дерево поиска — структура данных для работы с упорядоченными множествами. Один узель может иметь сколько угодно потомков.

Двоичное дерево поиска — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше. (В случае, если в поле данных расположена структура или касс, то нужно либо в структуре/классе перегрузить(определить) оператор сравнения, либо поиск выполнить по определенному полю структуры/класса)

Основным преимуществом является возможная высокая эффективность реализации основанных на нём алгоритмов поиска ($O\ (\log n)$) и сортировки ($O\ (n \log n)$).

Подробнее про деревьев (Бинарное, АВЛ, КЧ):
https://markoutte.me/wp-content/uploads/Сбалансированные-деревья-поиска.pdf

13. Сбалансированные деревья. Основные понятия. Малый и большой повороты дерева.

Сбалансировзнное дерево – сбалансированное по высоте двоичное дерево поиска.

Для каждой его вершины высота её двух поддеревьев различается не более чем на 1 (в случае АВЛ), количество черных узлов совпадает (в случае КЧ дерева), в последнем уровне нет "дырок" (в случае 2-3-4… дерева).

Малый поворот бывает левый и правый.

Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев |высота (L)− высота (R)| = 2, изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева | высота (L)− высота (R) | ⩽ 1, иначе ничего не меняет.

Двойные (большие) повороты выполняются в случае, если при малом повороте невозможно сбалансироватрь дерево. Большие повороты выполняются относительно данного узла и родителя/дляди/деда данного узла (зависит ос случае).

Двоичное дерево представляет собой в общем случае неупорядо-ченный набор узлов, который

- либо пуст (пустое дерево)
- либо разбит на три непересекающиеся части:

- узел, называемый корнем;
- двоичное дерево, называемое левым поддеревом;
- двоичное дерево, называемое правым поддеревом.

Таким образом, двоичное дерево — это рекурсивная структура данных.

Каждый узел двоичного дерева можно представить в виде струк-туры данных, состоящей из следующих полей:

- данные, обладающие ключом, по которому их можно идентифицировать;
- указатель на левое поддерево;
- указатель на правое поддерево;
- указатель на родителя (необязательное поле).

Значение ключа уникально для каждого узла.

14. Сбалансированные деревья. АВЛ-деревья. Основные понятия.

См. вопрос 13 +

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

(АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.)

В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), хранит-ся показатель баланса – разность высот правого и левого поддере-вьев.

Максимальная высота АВЛ-дерева при заданном числе узлов:

$$height \leq \lfloor \sqrt{2} \log(n+2) \rfloor$$

Балансировка. Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

Используются 4 типа вращений:

- Большое левое вращение
- Малое левое вращение
- Большое правое вращение
- Малое правое вращение

15. Сбалансированные деревья. АВЛ-деревья. Алгоритм добавления нового узла.

См. вопрос 13 + 14 +

Алгоритм добавления нового узла.

1. Проходим по дереву поиска, пока не убедимся, что добавляемого ключа в дереве нет.
2. Включения новой вершины в дерево
3. Определения результирующих показателей балансировки.
4. «Отступления» назад по пути поиска и проверки в каждой вершине показателя сбалансированности.
5. Если необходимо — балансировка.

Балансировка:

Будем возвращать в качестве результата функции, уменьшилась высота дерева или нет. Предположим, что процесс из левой ветви возвращается к родителю (рекурсия идет назад), тогда возможны три случая: { hl — высота левого поддерева, hr — высота правого поддерева } Включение вершины в левое поддерево приведет к

1. hl < hr: выравняется hl = hr. Ничего делать не нужно.
2. hl = hr: теперь левое поддерево будет больше на единицу, но балансировка пока не требуется.
3. hl > hr: теперь | hl — hr | = 2, — требуется балансировка. В данной ситуации требуется определить балансировку левого поддерева. Если левое поддерево этой вершины (Tree^.left^.left) выше правого (Tree^.left^.right), то требуется большое правое вращение, иначе хватит малого правого. Аналогичные (симметричные) рассуждения можно привести и для включение в правое поддерево.

16. Сбалансированные деревья. АВЛ-деревья. Алгоритм удаления существующего узла.

См. вопрос 13 + 14 +

Рекурсивный алгоритм удаления.

Если вершина — лист, то удалим её и вызовем балансировку всех её предков в порядке от родителя к корню.

Иначе найдём самую близкую по значению вершину в поддереве наибольшей высоты (правом или левом) и переместим её на место удаляемой вершины, при этом вызвав процедуру её удаления.

Докажем, что данный алгоритм сохраняет балансировку. Для этого докажем по индукции по высоте дерева, что после удаления некоторой вершины из дерева и последующей балансировки высота дерева уменьшается не более, чем на 1. База индукции: Для листа очевидно верно. Шаг индукции: Либо условие балансированности в корне (после удаления корень может измениться) не нарушилось, тогда высота данного дерева не изменилась, либо уменьшилось строго меньшее из поддеревьев => высота до балансировки не изменилась => после уменьшится не более чем на 1.

В результате указанных действий процедура удаления вызывается не более 3 раз, так как у вершины, удаляемой по второму вызову, нет одного из поддеревьев. Но поиск ближайшего каждый раз требует $O(n)$ операций. Становится очевидной возможность оптимизации: поиск ближайшей вершины может быть выполнен по краю поддерева, что сокращает сложность до $O(\log n)$.

Нерекурсивное удаление из АВЛ-дерева сверху вниз:

1. **Найдём вершину, удаление из которой не приведёт к изменению её высоты.**
   Существует два случая:
   1. высота левого поддерева равна высоте правого поддерева (исключая случай, когда у листа нет поддеревьев)
   2. высота дерева по направлению движения меньше противоположной(«брат» направления) и баланс «брата» равен 0 (разбор этого варианта довольно сложен, так что пока без доказательства)

2. **Найденная удаляемая вершина заменяется значением из левой подветви.**
    1. ищем удаляемый элемент и попутно находим нашу замечательную вершину
    2. производим изменение балансов, в случае необходимости делаем ребалансировку
    3. удаляем наш элемент (в действительности не удаляем, а заменяем его ключ и значение, учёт перестановок вершин будет немного сложнее)

17. Сбалансированные деревья. Красно-чёрные деревья. Основные понятия.

См. вопрос 13 +

Красно-чёрные деревья:

КЧ-деревья – это двоичные деревья поиска, каждый узел которых хранит дополнительное поле color, обозначающее цвет: красный или черный, и для которых выполнены приведенные ниже свойства.

```
struct RBNode {
     key_type key;
     struct RBNode *left, *right, *parent;
     char color; // цвет
 };
```

Если left или right равны NULL, то это «указатели» на фиктивные листья. Таким образом, все узлы – внутренние (нелистовые).

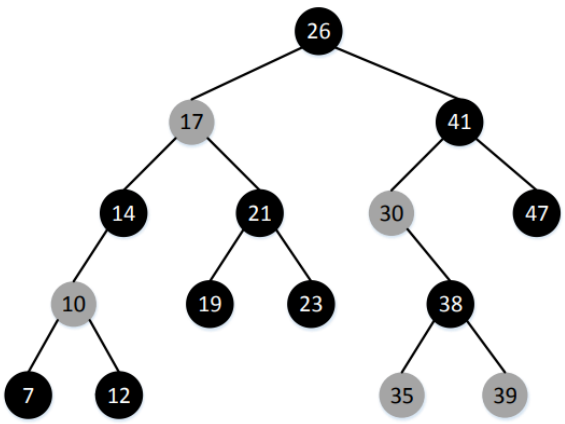
Свойства КЧ-деревьев:

1. каждый узел либо красный, либо черный;
2. каждый лист (фиктивный) – черный;
3. если узел красный, то оба его сына – черные;
4. все пути, идущие от корня к любому фиктивному листу, содержат одинаковое количество черных узлов;
5. корень – черный.

Черная высота:

Черной высотой узла называется количество черных узлов на пути от этого узла к узлу, у которого оба сына – фиктивные листья.

Сам узел не включается в это число. Например, у дерева, приведенного на рисунке ниже, черная высота корня равна 2.



18. Сбалансированные деревья. Красно-чёрные деревья. Алгоритм добавления нового узла.

См. вопрос 13 + 17 +

См.стр 38 из этой книги:

https://markoutte.me/wp-content/uploads/Сбалансированные-деревья-поиска.pdf

19. Сбалансированные деревья. Красно-чёрные деревья. Алгоритм удаления существующего узла.

См. вопрос 13 + 17 +

См.стр 42 из этой книги:

https://markoutte.me/wp-content/uploads/Сбалансированные-деревья-поиска.pdf

20. Сбалансированные деревья. B-деревья. 2-3-4 деревья. Основные понятия.

См. вопрос 13 +

B-дерево — структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти.

B-деревом называется дерево, удовлетворяющее следующим свойствам:

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2t - 1$ ключей. Любой другой узел содержит от $t - 1$ до $2t - 1$ ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000).

2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1, \dots K_n$, содержит $n + 1$ потомков. При этом

1. Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$
2. Для $2 \leq i \leq n$, i-й потомок и все его потомки содержат ключи из интервала (K_{i-1}, K_i)
3. $(n + 1)$-й потомок и все его потомки содержат ключи из интервала $(K_n, +\infty)$

3. Глубина всех листьев одинакова.

Поиск ключа в Б-дереве:

Если ключ содержится в корне, он найден. Иначе определяем интервал и идём к соответствующему потомку. Повторяем.

2-3-4 дерево:

2–3–4 дерева (также названный **деревом 2–4**) являются самоуравновешивающейся структурой данных. Числа означают дерево, где каждый узел с детьми (внутренний узел) имеет или два, три, или четыре детских узла:

- с 2 узлами имеет один элемент данных, и, если внутренний имеет два детских узла;
- с 3 узлами имеет два элемента данных, и, если внутренний имеет три детских узла;
- с 4 узлами имеет три элемента данных, и, если внутренний имеет четыре детских узла.

2–3–4 дерева - B-деревья где t=4; как B-деревья в целом, они могут искать, вставить и удалить в $O\ (n)$, время. Одна собственность 2–3–4 деревьев состоит в том, что все внешние узлы на той же самой глубине.

2–3–4 дерева - изометрия красно-черных деревьев, означая, что они - эквивалентные структуры данных. Другими словами, для каждых 2–3–4 деревьев, там существует по крайней мере одно красно-черное дерево с элементами данных в том же самом заказе. Кроме того, вставка и операции по удалению на 2–3–4 деревьях, которые вызывают расширения узла, разделения и слияния, эквивалентны щелканию цвета и вращениям в красно-черных деревьях. Введения в красно-черные деревья обычно вводят 2–3–4 дерева сначала, потому что они концептуально более просты. 2–3–4 дерева, однако, может быть трудно осуществить на большинстве языков программирования из-за большого количества особых случаев, вовлеченных в операции на дереве. Красно-черные деревья более просты осуществить, так будьте склонны использоваться вместо этого.

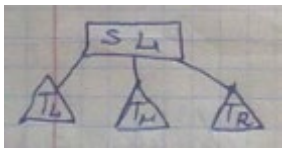
Свойства

- Каждый узел (лист или внутренний) является с 2 узлами, с 3 узлами или с 4 узлами, и держится один, два, или три элемента данных, соответственно.
- Все листья на той же самой глубине (нижний уровень).
- Все данные сохранены в сортированном заказе.

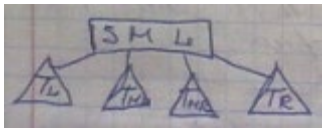
Узел может быть двух-, трех- и четырехместным.

1. $T_L < S;\quad T_R > S$

2. $T_L < S;\ S < T_M < L;\ T_R > L$



3. $T_L < S;\ S < T_{ML} < M;\ M < T_{MR} < L;\ T_R > L$



Свойства четырехместного узла:

1. может быть корнем
2. может иметь три сына и два элемента данных
3. может иметь четыре сына и три элемента данных

Максимальная высота $2-3-4$ - дерева $h = (log2n + 1)$, и вставка нового элемента, как правило, не изменяет ее за исключением случая, когда разделяется корень дерева.

21. Сбалансированные деревья. 2-3-4 деревья. Алгоритм добавления нового ключа.

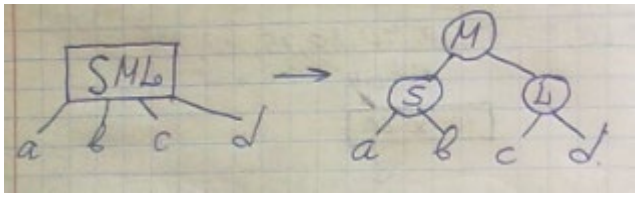
См. вопрос 13 + 20 +

Чтобы вставить узел, мы начинаем в корне 2–3–4 деревьев:

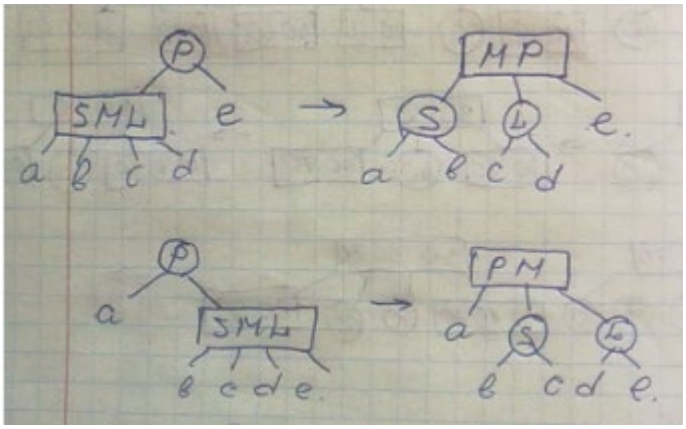
1. Если текущий узел - с 4 узлами:
2. * Удаляют и экономят среднюю стоимость, чтобы получить с 3 узлами.
3. * Разделение оставление с 3 узлами в пару 2 узлов (теперь недостающая средняя стоимость обработана в следующем шаге).
4. *, Если это - узел корня (у которого таким образом нет родителя):
5. ** средняя стоимость становится новым корнем, с 2 узлами и увеличения высоты дерева 1. Поднимитесь в корень.
6. * Иначе, увеличьте среднюю стоимость в родительский узел. Поднимитесь в родительский узел.
7. Найдите ребенка, интервал которого содержит стоимость, которая будет вставлена.
8. Если тот ребенок - лист, вставьте стоимость в детский узел и конец.
9. * Иначе, спуститесь в ребенка и повторение от шага 1.

Разделение четырех местного узла (остальное как в 2-3 дереве).

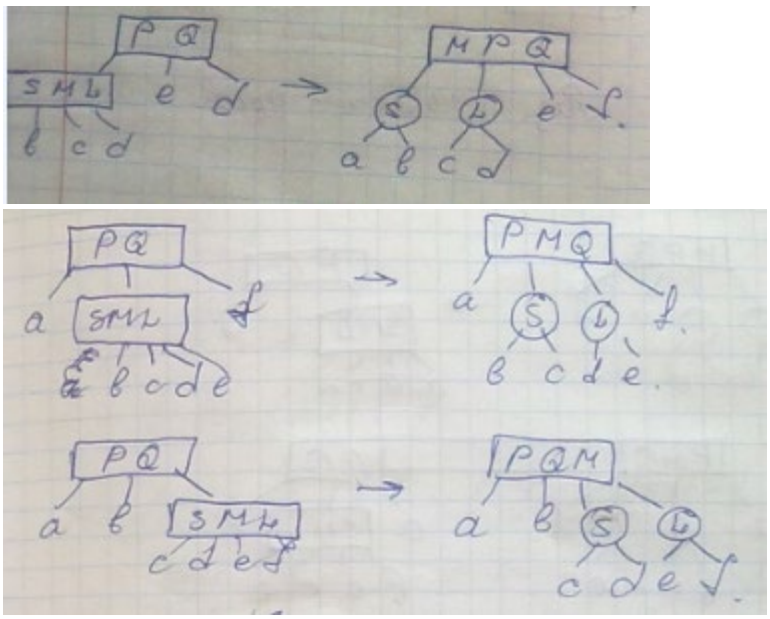
1. Корень



2. Родитель двухместный



3. Родитель трехместный



22. Сбалансированные деревья. 2-3-4 деревья. Алгоритм удаления существующего узла.

См. вопрос 13 + 20 +

Удалить узла из 2–3–4 деревьев:

1. Поиск удаляемого элемента.
2. *, Если элемент не находится в узле листа, помните его местоположение и продолжите искать, пока лист, который будет содержать преемника элемента, не будет достигнут. Преемник может быть или самым большим ключом, который меньше, чем тот, который будет удален, или самый маленький ключ, который больше, чем тот, который будет удален. Является самым простым внести изменения в дерево от вершины, вниз таким образом, что найденный узел листа не является с 2 узлами. Тем путем, после обмена, не будет пустой узел листа.
3. *, Если элемент находится в листе с 2 узлами, просто внесите корректировки ниже.

Внесите следующие корректировки, когда с с 2 узлами – кроме узла корня – сталкиваются на пути к листу, мы хотим удалить:

1. Если родной брат по обе стороны от этого узла - с 3 узлами или с 4 узлами (таким образом наличие больше чем 1 ключа), выполните вращение с тем родным братом:
2. * ключ от другого родного брата, самого близкого к этому узлу, перемещается до родительского ключа, который пропускает эти два узла.
3. * родительский ключ спускается к этому узлу, чтобы сформировать с 3 узлами.
4. * ребенок, который был первоначально с вращаемым ключом родного брата, является теперь дополнительным ребенком этого узла.
5. Если родитель - с 2 узлами, и родной брат - также с 2 узлами, объедините все три элемента, чтобы сформировать новый с 4 узлами и сократить дерево. (Это правило может только вызвать, если родитель, с 2 узлами, является корнем, так как все другие 2 узла по пути

будут изменены, чтобы не быть 2 узлами. Это - то, почему «сокращаются, дерево» здесь сохраняет равновесие; это - также важное предположение для операции по сплаву.)

6. Если родитель - с 3 узлами или с 4 узлами, и все смежные родные братья - 2 узла, сделайте операцию по сплаву с родителем и смежным родным братом:

7. * смежный родной брат и родительский ключ, пропускающий два узла родного брата, объединяются, чтобы сформировать с 4 узлами.

8. * Передача дети родного брата к этому узлу.

Как только разыскиваемая стоимость достигнута, она может теперь быть помещена в местоположение удаленного входа без проблемы, потому что мы гарантировали, что у узла листа есть больше чем 1 ключ.

Удаление в 2–3–4 деревьях - O (зарегистрируйте n), приняв передачу и пробег сплава в постоянное время (O (1)).

23. Хэш-таблицы. Понятие хэш-функции. Хэширование делением. Хэширование умножением. Универсальное хэширование.

Хэш-табли́ца — это структура данных, реализующая интерфейс ассоциативного массива (абстрактный тип данных), а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа.

Хеш-функция — функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

«Хеш-функции», основанные на делении:

1. «Хеш-код» как остаток от деления на число всех возможных «хешей»

Хеш-функция может вычислять «хеш» как остаток от деления входных данных на M:

$$h(k) = k \bmod M$$

где M — количество всех возможных «хешей». (должен быть простым)

2. «Хеш-код» как набор коэффициентов получаемого полинома

Хеш-функция может выполнять деление входных данных на полином по модулю два. В данном методе M должна являться степенью двойки, а бинарные ключи ($K = k_{n-1}, k_{n-2}, \dots k_0$) представляются в виде полиномов, в качестве «хеш-кода» «берутся» значения коэффициентов полинома, полученного как остаток от деления входных данных K на заранее выбранный полином P степени m:

$$K(x) \bmod P(x) = h_{m-1}x^{m-1} + \cdots + h_1 x + h_0$$

$$h(x) = h_{m-1} \dots h_1 h_0$$

При правильном выборе P(x) гарантируется отсутствие коллизий между почти одинаковыми ключами.

«Хеш-функции», основанные на умножении:

Обозначим символом w количество чисел, представимых машинным словом. Например, для 32-разрядных компьютеров, $w = 2^{32}$. Выберем некую константу A так, чтобы A была взаимно простой с w. Тогда хеш-функция, использующая умножение, может иметь следующий вид:

$$h(K) = \left[M \left\lfloor \frac{A}{w} * K \right\rfloor\right]$$

В этом случае на компьютере с двоичной системой счисления M является степенью двойки, и $h(K)$ будет состоять из старших битов правой половины произведения $A * K$.

Одной из хеш-функций, использующих умножение, является хеш-функция, использующая хеширование Фибоначчи. Хеширование Фибоначчи основано на свойствах золотого сечения. В качестве константы A здесь выбирается целое число, ближайшее к $\varphi^{-1} * w$ и взаимно простое с w, где φ — это золотое сечение.

Универсальное хеширование:

Универсальным хешированием называется хеширование, при котором используется не одна конкретная хеш-функция, а происходит выбор хеш-функции из заданного семейства по случайному алгоритму. Универсальное хеширование обычно отличается низким числом коллизий, применяется, например, при реализации хеш-таблиц и в криптографии.

Предположим, что требуется отобразить ключи из пространства U в числа $[m]$. На входе алгоритм получает данные из некоторого набора $S \in U$ размерностью n. Набор заранее неизвестен. Как правило, алгоритм должен обеспечить наименьшее число коллизий, чего трудно добиться, используя какую-то определённую хеш-функцию. Число коллизий можно уменьшить, если каждый раз при хешировании выбирать хеш-функцию случайным образом. Хеш-функция выбирается из определённого набора хеш-функций, называемого универсальным семейством $H = \{h: U \to [m]\}$

24. Сортировка сравнениями. Пузырьковая сортировка (bubble).

Алгоритм сортировки — это алгоритм для упорядочивания элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки.

Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость худшего случая для этих алгоритмов составляет $O\ (n \log n)$, но они отличаются гибкостью применения.

Пузырьковая сортировка (bubble):

Сортировка простыми обменами, сортиро́вка пузырько́м (bubble sort) — простой алгоритм сортировки, для понимания и реализации — простейший, но эффективен он лишь для небольших массивов. Сложность алгоритма: $O(n^2)$.

```
for (int i = 0; i < input.length(); i++) {
```

```
        for (int j = input.length - 1; j > i; j--) {
             if (input[j] < input[j - 1]) {
                 swap(j, j-1);
             }
        }
    }
```

25. Сортировка сравнениями. Сортировка вставками (insertion).

См. вопрос 24 +

Сортировка вставками (insertion):

Сортировка вставками (англ. Insertion sort) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Сложность алгоритма: $O(n^2)$.

На вход процедуре сортировки подаётся массив $A[1..n]$, состоящий из элементов последовательности $A[1], A[2], \ldots, A[n]$, которые требуется отсортировать. n соответствует $A.length()$ — размеру исходного массива. Для сортировки не требуется привлечения дополнительной памяти, кроме постоянной величины для одного элемента, так как выполняется перестановка в пределах массива. В результате работы процедуры во входном массиве оказывается требуемая выходная последовательность элементов.

Псевдокод алгоритма:

```
for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while (i > 0 and A[i] > key) do
        A[i + 1] = A[i]
        i = i - 1
    end while
    A[i+1] = key
end for
```

26. Сортировка сравнениями. Селекционная сортировка (selection).

См. вопрос 24 +

Сортировка выбором (selection).

Сортировка выбором (Selection sort) — алгоритм сортировки.. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $O(n^2)$, предполагая что сравнения делаются за постоянное время.

Алгоритм:

3. находим номер минимального значения в текущем списке

4. производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции)
5. теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы

(T – тип данных)

```
void selection_sort(T array[], unsigned int size) {
    for (unsigned int idx_i = 0; idx_i < size - 1; idx_i++) {
        unsigned int min_idx = idx_i;
        for (unsigned int idx_j = idx_i + 1; idx_j < size; idx_j++) {
            if (array[idx_j] < array[min_idx]) {
                min_idx = idx_j;
            }
        }

        if (min_idx != idx_i) {
            swap(array[idx_i], array[min_idx]);
            min_idx = idx_i;
        }
    }
}
```

27. Сортировка «разделяй и властвуй». Сортировка слияниями (merge-sort).

Принцип «разделяй и властвуй» (divide and conquer) – парадигма разработки алгоритмов, заключающаяся в рекурсивном разбиении решаемой задачи на две или более подзадачи того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче; разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными.

Алгоритм сортировки слиянием это типичный пример принципа «разделяй и властвуй». Чтобы отсортировать массив чисел по возрастанию, он разбивается на две равные части, каждая сортируется, затем отсортированные части сливаются в одну. Эта процедура применяется к каждой из частей до тех пор, пока сортируемая часть массива содержит хотя бы два элемента (чтобы можно было её разбить на две части). Время работы этого алгоритма составляет $O\ (n \log n)$ операций, тогда как более простые алгоритмы требуют $O(n^2)$ времени, где n — размер исходного массива.

28. Сортировка «разделяй и властвуй». Быстрая сортировка (quick-sort).

См. вопрос 27 +

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена («Пузырьковая сортировка»). В первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

Общая идея алгоритма состоит в следующем:

- Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
- Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на два непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного» и «равные или большие».
- Для обоих отрезков значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

```
void quicksort(A, low, high) {
    if (low <= high) {
        p := partition(A, low, high)
        quicksort(A, low, p - 1)
        quicksort(A, p + 1, high)
    }
}
```

29. Сортировка с использованием деревьев. Пирамидальная сортировка (heap-sort).

Двоичная куча, пирамиида, или сортирующее дерево — такое двоичное дерево, для которого выполнены три условия:

- Значение в любой вершине не меньше, чем значения её потомков.
- Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
- Последний слой заполняется слева направо без «дырок».

Высота кучи определяется как высота двоичного дерева. То есть она равна количеству рёбер в самом длинном простом пути, соединяющем корень кучи с одним из её листьев. Высота кучи есть $O\ (n \log n)$, где n — количество узлов дерева.

Пирамидальная сортировка:

Удобная структура данных для сортирующего дерева — это такой массив Array, что Array[0] — элемент в корне, а потомки элемента Array[i] являются Array[2i+1] и Array[2i+2].

Алгоритм сортировки будет состоять из двух основных шагов:

1. Выстраиваем элементы массива в виде сортирующего дерева

$Array[i] \geq Array[2i+1]$

$Array[i] \geq Array[2i+2]$

при $0 \leq i < n/2$

Этот шаг требует $O(n)$ операций.

2. Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем $Array[0]$ и $Array[n-1]$, преобразовываем $Array[0], Array[1], \dots, Array[n-2]$ в сортирующее дерево. Затем переставляем $Array[0]$ и $Array[n-2]$, преобразовываем $Array[0], Array[1], \dots, Array[n-3]$ в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда $Array[0], Array[1], \dots, Array[n-1]$ — упорядоченная последовательность.

Этот шаг требует $O\ (n \log n)$ операций.

Пирамида – бинарное дерево, в узлах ключи сортируем по значениям данных.

- Неубывающая;
-Невозрастающая;

Метод простого выбора. N(n-1) сравнений, n-1 перестановок.

Пирамидальная сортировка (сложность nlogn, неустойчивый алгоритм).

Дерево называется пирамидально упорядоченным, если ключ в каждом его узле ≥ ключам всех его потомков. Сортирующее дерево – совокупность ключей, образующих полное пирамидально упорядоченное дерево. Для реализации дерева используется массив([i/2]-родитель, 2i, 2i+1- потомки). При такой организации прохождение по дереву проходит более быстро с большой экономией памяти. Поддержание основного свойства пирамид дерева.

Heapify(A[1..n],i)

1. L←2*i
2. R←2*i+1
3. If (L<=Head_size) and (A[L]>A[i])
4. Then largest←L
5. Else largest←i
6. If (R<=Head_size) and (A[k]>A[largest])
7. Then largest←R
8. If largest≠I then A[i]⇔A[largest]
9. Hepify(A,largest)

Построение пирамиды

BiuldHeap(A[1..n])

1. Heap_size←n
2. For i= ⌊n/2⌋ downto 1 do
3. Heapify(a,i)
4. End for
5.

Сортировка

HeapSort(A[1..n])

1. BuildHeap(a)
2. For I = n downto 2 do
3. A[i]↔A[1]
4. Heapsize←Heapsize-1
5. Heapity(A,1)
6. End for

30. Сортировка больших файлов. Прямой алгоритм сортировки.

Под определением "большие файлы" понимаем данные, расположенные на периферийных устройствах и не вмещающихся в оперативную память, то есть когда применить одну из внутренних сортировок невозможно. Сортировка таких данных реализуется с помощью внешней сортировки. Внутренняя сортировка значительно эффективней внешней, так как на обращение к оперативной памяти затрачивается намного меньше времени, чем к магнитным дискам.

Сортировка прямым слиянием:

Прямой алгоритм внешней сортировки реализуется следующим образом: разделяем исходный файл на части (размер каждой части – это максимльно возможный размер данных которые помещаются в оперативной памяти.) и загружаем в оперативную память. Сортируем по одному из алгоритмов внутренней сортировки, выгружаем обратно. Продолжаем до тех пор пока не отсортировали все части. После объединяем отсортированные части сортировкой слиянием.

31. Сортировка больших файлов. Естественный алгоритм сортировки.

См. вопрос 30 +

Естественное слияние последовательностей:

Алгоритм слияния можно использовать и для сортировки массивов, если последовательно применить его несколько раз ко все более длинным упорядоченным последовательностям. Для этого:

1. в исходном наборе выделяются две подряд идущие возрастающие подпоследовательности (серии)
2. эти подпоследовательности (серии) сливаются в одну более длинную упорядоченную последовательность так, как описано выше
3. шаги 1 и 2 повторяются до тех пор, пока не будет достигнут конец входного набора
4. шаги 1 –3 применяются к новому полученному набору, т.е. выделяются пары серий, которые сливаются в еще более длинные наборы, и.т.д. до тех пор, пока не будет получена единая упорядоченная последовательность.

32. Графы. Основные понятия. Поиск в ширину. Поиск в глубину.

Граф, или неориентированный граф G — это упорядоченная пара G:=(V,E), где V— непустое множество вершин или узлов, а E — множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых рёбрами.

Обход графа

Алгоритм поиск в ширину. Пусть зафиксирована начальная вершина v_0. Рассматриваем все смежные с ней вершины $v_1, v_2,..., v_k$. Затем рассматриваем смежные вершины каждой из рассмотренных вершин $v_1, v_2,..., v_k$, и т.д. Так будут перебраны все вершины графа и поиск закончится.

Алгоритм поиск в глубину. Пусть зафиксирована начальная вершина v_0. Выберем смежную с ней вершину v_1. Затем для вершины v_1 выбираем смежную с ней вершину из числа еще не выбранных вершин и т.д.: если мы уже выбрали вершины $v_0, v_1,..., v_k$, то следующая вершина выбирается смежной с вершиной v_k из числа невыбранных. Если для вершины v_k такой вершины не нашлось, то возвращаемся к вершине v_{k-1} и для нее ищем смежную среди невыбранных. При необходимости возвращаемся назад и т.д. Так будут перебраны все вершины графа и поиск закончится.

Поиск в ширину в графе. Не рекурсивный алгоритм.

Этот метод основан на замене стека очередью. В этом случае, чем раньше посещается вершина (помещается в очередь), тем раньше она используется (удаляется из очереди).

Использование вершины происходит с помощью просмотра сразу всех еще непросмотренных соседей этой вершины.

```
1    PROCEDURE WS(v);
     {поиск в ширину в графе с началом в вершине v; переменные НОВЫЙ, СПИСОК -
     глобальные}
2    BEGIN
3       ОЧЕРЕДЬ := 0; ОЧЕРЕДЬ <= v; НОВЫЙy := false;
4       WHILE ОЧЕРЕДЬ :Ф 0 DO begin

5          P<= ОЧЕРЕДЬ; посетить p;
6          FOR tСПИСОКp DO
7          IF НОВЫЙt THEN BEGIN
8             ОЧЕРЕДЬ <= t; НОВЫЙt := false
9          END
10      END
11   END
```

Вызов процедуры WS(v) приводит к посещению всех вершин компоненты связности графа, содержащей вершину v, причем каждая вершина просматривается ровно один раз. Вычислительная сложность алгоритма также имеет порядок m+n, т.к. каждая вершина помещается в очередь и удаляется из очереди в точности один раз, а число итераций цикла 6, очевидно, будет иметь порядок числа ребер графа.

Поиск в графе в ширину может быть использован для нахождения пути между фиксированными вершинами v и t. Для этого достаточно начать поиск в графе с вершины v и вести его до вершины t.

Поиск в глубину является обобщением метода обхода дерева в прямом порядке.

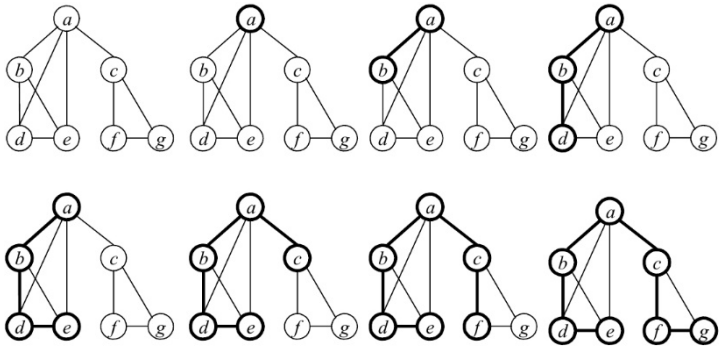
Предположим, что есть ориентированный граф G, в котором первоначально все вершины помечены как непосещенные. Поиск в глубину начинается с выбора начальной вершины v графа G, и эта вершина помечается как посещенная. Затем для каждой вершины, смежной с вершиной v и которая не посещалась ранее, рекурсивно применяется поиск в глубину. Когда все вершины, которые можно достичь из вершины v, будут «удостоены» посещения, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и поиск повторяется. Этот процесс продолжается до тех пор, пока обходом не будут охвачены все вершины орграфа G.

Этот метод обхода вершин орграфа называется поиском в глубину, поскольку поиск непосещенных вершин идет в направлении вперед (вглубь) до тех пор, пока это возможно. Например, пусть x – последняя посещенная вершина. Для продолжения процесса выбирается какая-либо нерассмотренная дуга x → y, выходящая из вершины x. Если вершина y уже посещалась, то ищется другая вершина, смежная с вершиной x. Если вершина y ранее не посещалась, то она помечается как посещенная и поиск начинается заново от вершины y. Пройдя все пути, которые начинаются в вершине y, возвращаемся в вершину x, т. е. в ту вершину, из которой впервые была достигнута вершина y. Затем продолжается выбор нерассмотренных дуг, исходящих из вершины x, и так до тех пор, пока не будут исчерпаны все эти дуги.

Для представления вершин, смежных с вершиной v, можно использовать список смежных, а для определения вершин, которые ранее посещались, – массив `flag`:

```
#define n 9
Bool flag[n]={false};
```

Чтобы применить эту процедуру к графу, состоящему из n вершин, надо сначала присвоить всем элементам массива `flag` значение `false` , затем начать поиск в глубину для каждой вершины, помеченной как `false`.



Поиск в глубину

Поиск в глубину для полного обхода графа с n вершинами и m дугами требует общего времени порядка O(max(n, m)). Поскольку обычно m ≥ n, то получается O(m).

```
#include <stdio.h>
#include <conio.h>
#define n 9
int A[n][n];
bool flag[n] = {false};

void DFS(int prev, int cur)
{
    if(prev>=0)
        printf("%d - %d\n",prev+1,cur+1);
    flag[cur] = true;
    for(int i=0;i<n;i++)
        if(flag[i]==false && A[cur][i]==1)
            DFS(cur,i);
}

int _tmain(int argc, _TCHAR* argv[])
{
    FILE *f = fopen("graph.txt","r");
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            fscanf(f,"%d",&A[i][j]);
    fclose(f);
    int FST;
    printf("Input number of list vertex: ");
    scanf("%d",&FST);
    DFS(-1,FST-1);
    getch();
    return 0;
}
```

Алгоритм DFS(G)

1. For each v∈V do Mark[v]<-0
2. For v∈V do
3. If Mark[v]=0 then dfs(v)

Dfs(v)

1. Mark[v]<-1
2. Обработка вершины v
3. For each w∈V, смежные с v do
4. If Mark[w]=0 then dfs(w)
5. End for.

33. Графы. Поиск кратчайшего пути. Алгоритм Дейкстры.

См. 32 вопрос +

Зада́ча о кратча́йшем пути́ — задача поиска самого короткого пути (цепи) между двумя точками (вершинами) на графе, в которой минимизируется сумма весов рёбер, составляющих путь.

Алгори́тм Де́йкстры (Dijkstra's algorithm) — алгоритм на графах, находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

функция D(i) – массив значений весов ребер

функция P(s, i) – функция возвращающая вес ребер между начальным (s) и другими вершинами, если такого ребра нет, возвращает бесконечность.

W – множество обработанных вершин

Инициализация:

ЦИКЛ $for\ (i = 1 \dots \dots n)$

$D[i] := P(s, i)$

КЦ

Основной алгоритм:

$\boldsymbol{W := \{s\}}$ // s - начальная вершина

ЦИКЛ ПОКА $V \backslash W \neq \emptyset$ // пока есть необработанные вершины

найти вершину $w \in V \backslash W$ с наименьшей пометкой $D[w]$

перенести найденную вершину в множество обработанных:

$\boldsymbol{W := W \cup \{w\}}$

модифицировать пометки необработанных вершин через w:

ЦИКЛ ПО всем вершинам $u \in V \backslash W$, смежным с w

$\boldsymbol{D[u] := min\{D[u]; D[w] + P(w; u)\}}$

КЦ

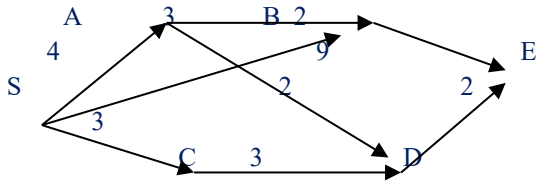
КЦ

Алгоритм Dijkstra (G,A,S)

1) for each v ∈ V do d[v]← ∞;
2) d[s]←0; //путь до вершины S

3) S← ∅; //множество вершин с известным расстоянием
4) t←v; //множество вершин с не известным расстоянем
5) while T≠0 do;
6) d[w]=min {d[p], p ∈ T};
7) S←SV{w}; T←T-{w};
8) for each v ∈ T do;
9) if d[v] > d[w]+a[w,v] then;
10) d[v] ← d[w] + a[w,v];
11) end for;
12) end while;
13) return d.

#



d[S]=0; d[A]=∞; d[B]=∞; d[C]=∞; d[D]=∞; d[E]=∞;

S≠ ∅; T={S, A, B, C, D, E};

d[A]=min{d[A], d[S] + a[S,A]=4};
d[B]=min{d[B], d[S] + a[S,B]=9};
d[C]=3; – Минимальное значение переносим во множество S;
d[D]=∞;

d[E]=∞;

S={S,C} T={A,B,D,E};
Пересчитываем
d[A]=min{d[A], d[C] + a[C,A]}=min{4, 3 + ∞} = 4;

d[B]=min{d[B], d[C] + a[C,B]}=min{9, 3 + ∞}=9;

d[D]=min{d[D], d[C] + a[C,D]}=min{∞, 3 + 3}=6;

d[E] =∞;

S={S,C,A} T={B,D,E};
Пересчитываем
d[B]=min{d[B], d[A] + a[A,B]}=min{9, 3 + 4}=7;
d[D]=min{d[D], d[A] + a[A,D]}=min{6, 4 + 2}=6;
d[E] =∞;

S={S,C,A,D} T={B,E};
Пересчитываем
d[B]=min{d[B], d[D] + a[D,B]}=min{7, 6 + ∞}=7;

d[E] = min{d[E], d[D] + a[D,E]}=8;
S={S,C,A,D,B} T={E};
Пересчитываем
d[E] = min{d[E], d[B] + a[B,E]}=min{9,8}=8;
S={S,C,A,D,B,E} T=∅;

34. Графы. Построение минимального остовного дерева. Алгоритм Прима.

См. 32 вопрос +

Минимальное остовное дерево (или минимальное покрывающее дерево) в связанном взвешенном неориентированном графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.

Существует несколько алгоритмов для нахождения минимального остовного дерева. наиболее известные из них это Алгоритм Прима и Алгоритм Краскала.

Алгоритм Прима — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа.

На вход алгоритма подаётся связный неориентированный граф. Для каждого ребра задаётся его стоимость.

Сначала берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево. Затем, рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет; из этих рёбер выбирается ребро наименьшей стоимости. Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Результатом работы алгоритма является остовное дерево минимальной стоимости.

Алгоритм:

Отсортировать список рёбер.Взять первую вершину из списка рёбер.

Записать вершину в массив обработанных вершин.

ЦИКЛ ПОКА (Все вершины не занесены в массив обработанных)
    ЦИКЛ ПОКА (Не конец списка рёбер и вершина не обработана)
        Выделить первое ребро из списка, соединяющего обработанную вершину с необработанной.
        Записать необработанную вершину в массив обработанных.
        Записать данное ребро в массив с результатами.
        Удалить данное ребро из списка рёбер.
    КЦ
КЦ

35. Графы. Построение минимального остовного дерева. Алгоритм Крускала.

См. 32 вопрос + 33 вопрос +
Алгоритм Краскала/Крускала/Крушкала/Жозефа/Йосефа/Йюсуфа/…/

Алгоритм Краскала — эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа, общая идея алгоритма состоит в следующем:

1. Упорядочить все ребра по возрастанию (неубыванию) весов,
2. Присвоить вершинам графа различные цвета (номера).
3. **ЦИКЛ ПОКА** не кончились ребра
    Рассмотреть очередное ребро
    ЕСЛИ концы ребра окрашены в разные цвета, **ТО**

добавить ребро в стягивающее дерево и перекрасить все вершины (в старом графе), номер которых совпадает с большим номером из концов этого ребра

КОНЕЦ ЕСЛИ

КОНЕЦ ЦИКЛА