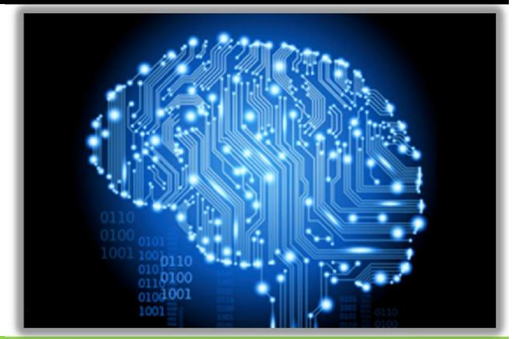
2017

PISL 03
Кодирование Хаффмана + heap



PISL Lesson03. Код Хаффмана + heap

- Определение.
- Жадный алгоритм кодирование Хаффмана.
- Идея алгоритма
  - наивный алгорит
  - оптимальный алгорит
- Дерево, как удобная структура для кодирования
  - принцип работы. Код Хаффмана.
  - Формальное описание алгоритма построения дерева
- Куча – структура данных для приоритетных очередей, сортировок и т.д.
- Задачи кодирование (A) декодирование (B) своя реализация кучи (C)

- **Материалы:** http://tinyurl.com/ei-pisl
- **Github:** https://github.com/Khmelov/PISL2017-01-26

PISL Lesson03. Код Хаффмана + heap

Советы по решению задач на программирование

- Тестируйте решение перед отправкой на git.
- **ПРИ КОПИРОВАНИИ ПАПОК КОПИРУЙТЕ ВСЕ и тесты и задачи**!
- Если вы скопировали тест, а задачу нет – то у других людей проект не будет собираться. **По этой же причине не держите ваши решения вне git**. Даже недоделанные – отправляйте.
- Если проект все-таки не собирается из-за чужих ошибок, не исправляйте их, а просто отключите чужие папки (ПКМ -> Mark Directory –> Excluded).

- Проверяйте ваше решение при граничных значениях параметра: если в задаче на вход даётся, скажем, массив размера $1 \le n \le 10000$, проверяйте, как ваша программа ведёт себя при n=1,2,10000.
- Следите за переполнением типов. Например, если в вашем коде на перемножаются две переменные типа int высокого порядка, то результат перемножения может не поместиться в тип int.
  - Получить представление об асимптотике ожидаемого решения можно по заданным ограничениям на размер входа. Как правило, чтобы уложиться во временные рамки (несколько секунд), программа должна совершать до миллиарда операций. Соответственно, при ограничениях $1 \le n \le 10000$, скорее всего,
    - ожидается решение с асимптотикой $O(n^2)$ или $O(n \log n)$,
    - а при $1 \le n \le 10^6$ – с асимптотикой $O(n)$ или $O(n \log n)$.
- Для удобства при прохождении тестов работает main и наоборот.

Литература

Разобраться по сути:

- **С. Дасгупта, Х. Пападимитриу, У. Вазирани. Алгоритмы. МЦНМО. 2014.**
- **Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ. Вильямс. 2013.**

Минимум для собеседований

- **Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 704 с.: ил. — (Серия «Классика computer science»)**

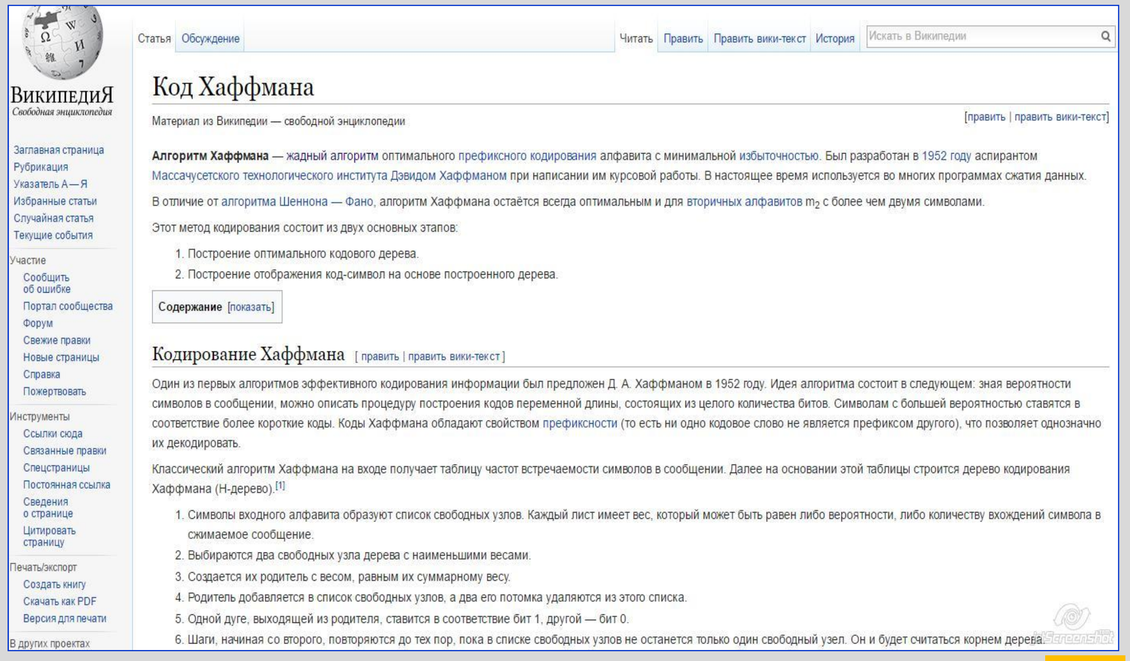
По желанию

- А. Шень. Программирование: теоремы и задачи. МЦНМО. 2014.

В качестве математической подготовки к курсу можно прочесть брошюры Александра Шеня «Логарифм и экспонента» и «Математическая индукция».

Конец документа

PISL Lesson03. Код Хаффмана + heap



ВикипедиЯ
Свободная энциклопедия

Статья | Обсуждение | Читать | Править | Править вики-текст | История | Искать в Википедии

Код Хаффмана

Материал из Википедии — свободной энциклопедии [править | править вики-текст]

Алгоритм Хаффмана — жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

В отличие от алгоритма Шеннона — Фано, алгоритм Хаффмана остаётся всегда оптимальным и для вторичных алфавитов m_2 с более чем двумя символами.

Этот метод кодирования состоит из двух основных этапов:

1. Построение оптимального кодового дерева.
2. Построение отображения код-символ на основе построенного дерева.

Содержание [показать]

Кодирование Хаффмана [править | править вики-текст]

Один из первых алгоритмов эффективного кодирования информации был предложен Д. А. Хаффманом в 1952 году. Идея алгоритма состоит в следующем: зная вероятности символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (Н-дерево).[1]

1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
2. Выбираются два свободных узла дерева с наименьшими весами.
3. Создается их родитель с весом, равным их суммарному весу.
4. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0.
6. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Заглавная страница
Рубрикация
Указатель А—Я
Избранные статьи
Случайная статья
Текущие события
Участие
Сообщить об ошибке
Портал сообщества
Форум
Свежие правки
Новые страницы
Справка
Пожертвовать
Инструменты
Ссылки сюда
Связанные правки
Спецстраницы
Постоянная ссылка
Сведения о странице
Цитировать страницу
Печать/экспорт
Создать книгу
Скачать как PDF
Версия для печати
В других проектах

Допустим, у нас есть следующая таблица частот:

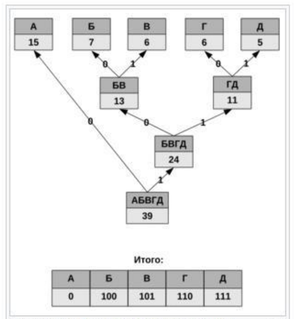
Символ	А	Б	В	Г	Д
Частота	15	7	6	6	5

Этот процесс можно представить как построение дерева, корень которого — символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n_0 потомков — символы из предыдущего шага и т. д.

Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке.

Для данной таблицы символов коды Хаффмана будут выглядеть следующим образом.

Символ	А	Б	В	Г	Д
Код	0	100	101	110	111



Построение дерева для данного примера

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д — наибольшим.

При этом общая длина сообщения, состоящего из приведённых в таблице символов, составит 87 бит (в среднем 2,2308 бита на символ). При использовании равномерного кодирования общая длина сообщения составила бы 117 бит (ровно 3 бита на символ). Заметим, что энтропия источника, независимым образом порождающего символы с указанными частотами, составляет ~2,1858 бита на символ, то есть избыточность построенного для такого источника кода Хаффмана, понимаемая как отличие среднего числа бит на символ от энтропии, составляет менее 0,05 бит на символ.

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Во-первых, для восстановления содержимого сжатого сообщения декодер должен знать таблицу частот, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частот, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию сообщения. Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по сообщению: одного для построения модели сообщения (таблицы частот и Н-дерева), другого для собственно кодирования. Во-вторых, избыточность кодирования обращается в ноль лишь в тех случаях, когда вероятности кодируемых символов являются обратными степенями числа 2. В-третьих, для источника с энтропией, не превышающей 1 (например, для двоичного источника), непосредственное применение кода Хаффмана бессмысленно.

Сжатие данных

Вход: строка s.

Выход: бинарный код символов строки s, обеспечивающий кратчайшее представление s.

Пример

$s =$ abacabad

коды символов: a: 00, b: 01, c: 10, d: 11

закодированная строка: 0001001000010011 (16 битов)

Коды переменной длины

- Естественная идея: присвоить более короткие коды более частым символам.

Коды переменной длины

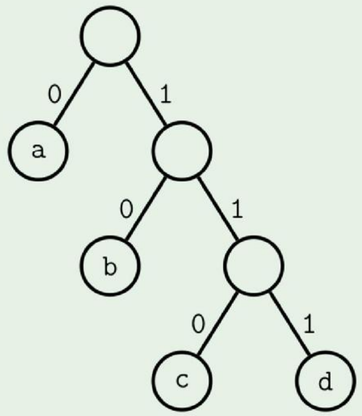
- Естественная идея: присвоить более короткие коды более частым символам.
- $s =$ abacabad
  коды символов: a: 0, b: 10, c: 110, d: 111
  закодированная строка: 01001100100111 (14 битов)

Коды переменной длины

- Естественная идея: присвоить более короткие коды более частым символам.
- s = abacabad
  коды символов: a: 0, b: 10, c: 110, d: 111
  закодированная строка: 01001100100111 (14 битов)
- Код называется беспрефиксным, если никакой код символа не является префиксом другого кода символа.

Декодирование на примере

Пример

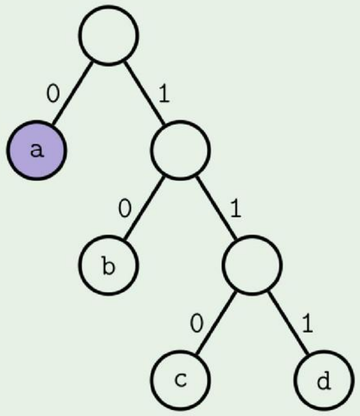


01001100100111

s =

Декодирование на примере

Пример

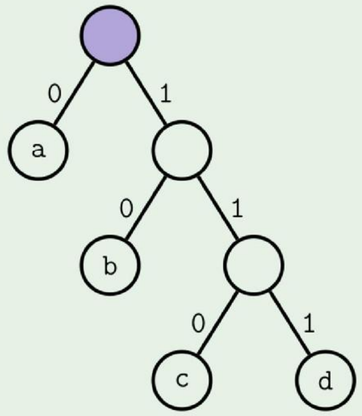


01001100100111

s = a

Декодирование на примере

Пример

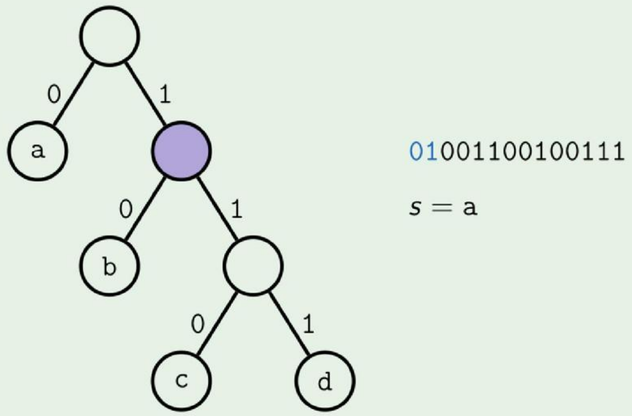


01001100100111

s = a

Декодирование на примере

Пример

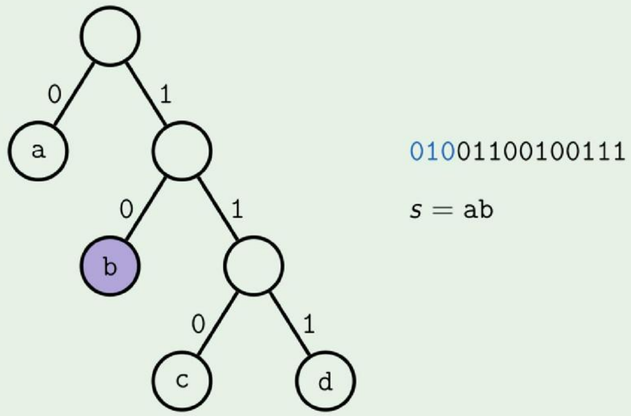


01001100100111

$s = \mathtt{a}$

Декодирование на примере

Пример

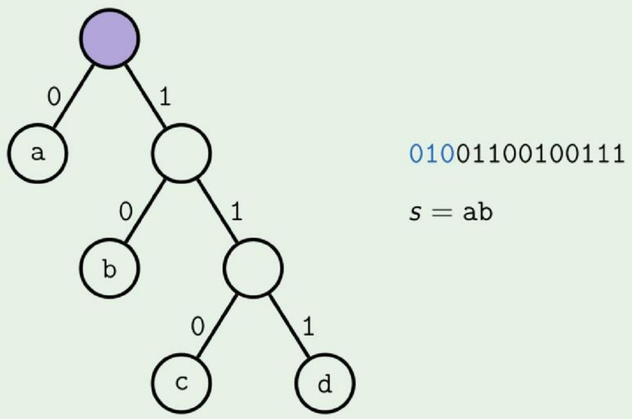


01001100100111

$s = \mathtt{ab}$

Декодирование на примере

Пример



01001100100111

$s = \mathtt{ab}$

Код Хаффмана

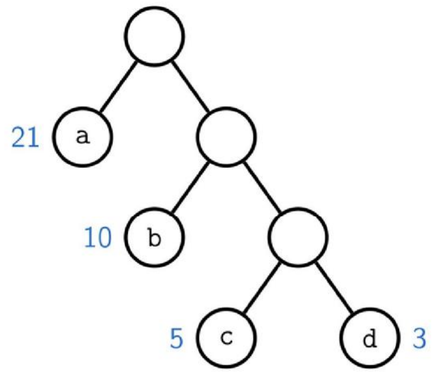
Код Хаффмана

Вход: частоты символов $f_1, \ldots, f_n \in \mathbb{N}$.

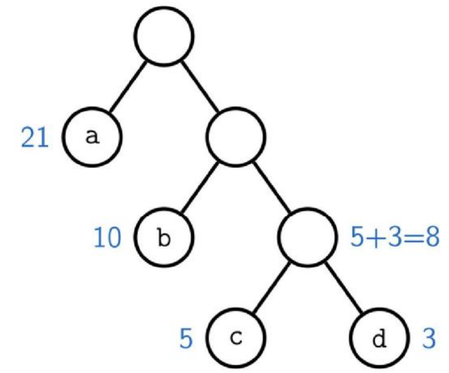
Выход: строго двоичное дерево (у каждой вершины либо ноль, либо два сына), листья которого помечены частотами $f_1, \ldots, f_n$, минимизирующее

$$\sum_{i=1}^{n} f_i \cdot (\text{глубина листа } f_i).$$

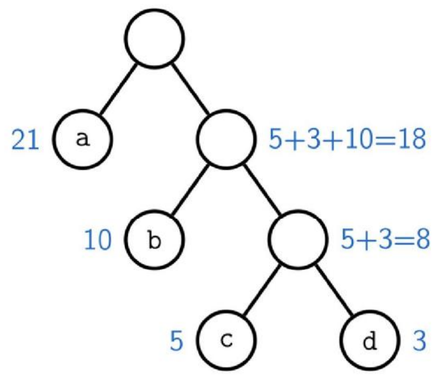
Частоты для внутренних вершин



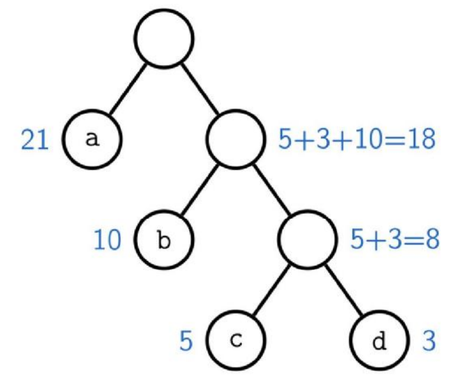
Частоты для внутренних вершин



Частоты для внутренних вершин



Частоты для внутренних вершин



Частотой (некорневой) вершины назовём количество раз, которое вершина будет посещена в процессе кодировки/декодировки.

Надёжный шаг

- Таким образом, мы ищем строго двоичное дерево с минимальной суммой пометок в вершинах, в котором листья помечены входными частотами, а внутренние вершины — суммами пометок их детей.

Надёжный шаг

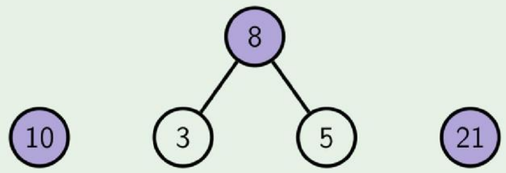
- Таким образом, мы ищем строго двоичное дерево с минимальной суммой пометок в вершинах, в котором листья помечены входными частотами, а внутренние вершины — суммами пометок их детей.
- Двумя наименьшими частотами помечены листья на нижнем уровне.

Надёжный шаг

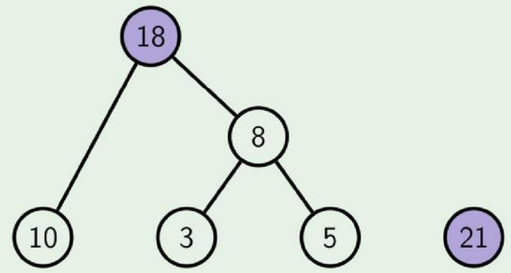
- Таким образом, мы ищем строго двоичное дерево с минимальной суммой пометок в вершинах, в котором листья помечены входными частотами, а внутренние вершины — суммами пометок их детей.
- Двумя наименьшими частотами помечены листья на нижнем уровне.
- Надёжный жадный шаг: выбрать две минимальные частоты f_i и f_j, сделать их детьми новой вершины с пометкой $f_i + f_j$; выкинуть частоты f_i и f_j, добавить $f_i + f_j$.

Пример

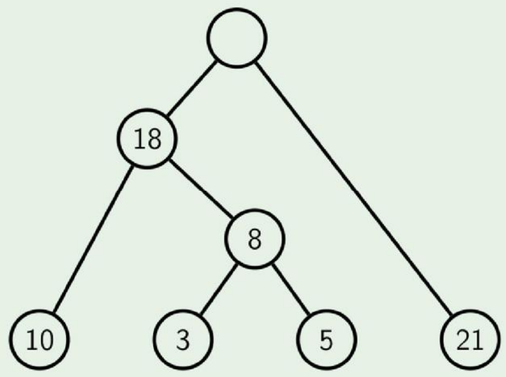
Пример



Пример



Пример



Очередь с приоритетами

INSERT(p) добавляет новый элемент с приоритетом p

EXTRACTMIN() извлекает из очереди элемент с минимальным приоритетом

Алгоритм

процедура HUFFMAN($F[1 \dots n]$)

```
H ← очередь с приоритетами
для i от 1 до n:
  INSERT(H, (i, F[i]))
для k от n+1 до 2n−1:
  (i, F[i]) ← EXTRACTMIN(H)
  (j, F[j]) ← EXTRACTMIN(H)
  создать вершину k с детьми i, j
  F[k] = F[i] + F[j]
  INSERT(H, (k, F[k]))
```


Алгоритм

процедура HUFFMAN($F[1 \dots n]$)

```
H ← очередь с приоритетами
для i от 1 до n:
  INSERT(H, (i, F[i]))
для k от n+1 до 2n−1:
  (i, F[i]) ← EXTRACTMIN(H)
  (j, F[j]) ← EXTRACTMIN(H)
  создать вершину k с детьми i, j
  F[k] = F[i] + F[j]
  INSERT(H, (k, F[k]))
```


Алгоритм

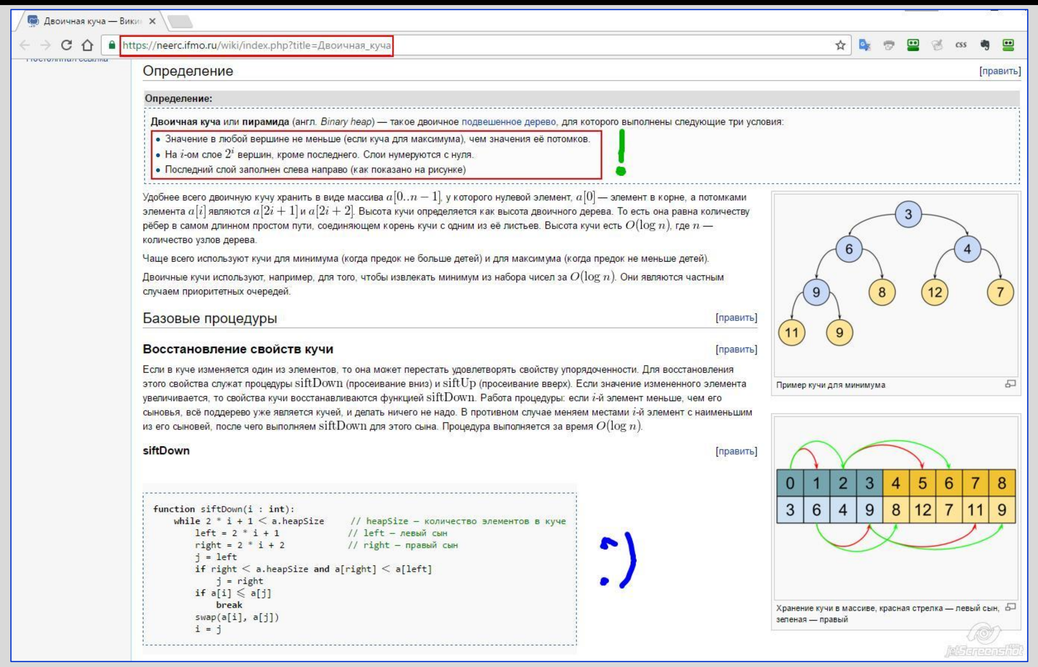
процедура HUFFMAN($F[1 \dots n]$)

```
H ← очередь с приоритетами
для i от 1 до n:
  INSERT(H, (i, F[i]))
для k от n+1 до 2n−1:
  (i, F[i]) ← EXTRACTMIN(H)
  (j, F[j]) ← EXTRACTMIN(H)
  создать вершину k с детьми i, j
  F[k] = F[i] + F[j]
  INSERT(H, (k, F[k]))
```

Время работы: $O(n^2)$, если очередь с приоритетами реализована на базе массива, $O(n \log n)$ — если на базе кучи (разберём в следующей лекции).

Очередь с приоритетами

INSERT(p)	добавляет новый элемент с приоритетом p
REMOVE(it)	удаляет элемент, на который указывает итератор it
GETMIN()	возвращает элемент с минимальным приоритетом
EXTRACTMIN()	извлекает из очереди элемент с минимальным приоритетом
CHANGEPRIORITY(it, p)	изменяет приоритет элемента, на который указывает итератор it, на p



Простейшие реализации

- массив: GETMIN имеет время работы $O(n)$

3	1	5	4	2

Простейшие реализации

- массив: GETMIN имеет время работы $O(n)$

3	1	5	4	2

- упорядоченный массив: GETMIN — $O(1)$, REMOVE — $O(n)$

1	2	3	4	5

Простейшие реализации

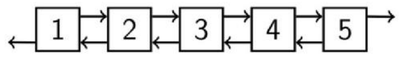
- массив: GETMIN имеет время работы $O(n)$

3	1	5	4	2

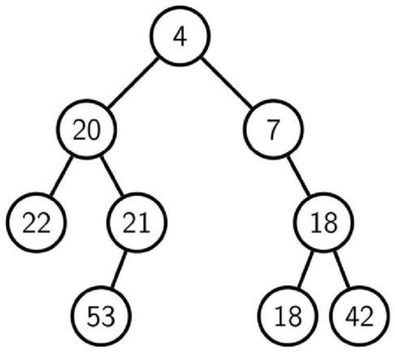
- упорядоченный массив: GETMIN — $O(1)$, REMOVE — $O(n)$

1	2	3	4	5

- упорядоченный список: GETMIN and REMOVE — $O(1)$, INSERT — $O(n)$

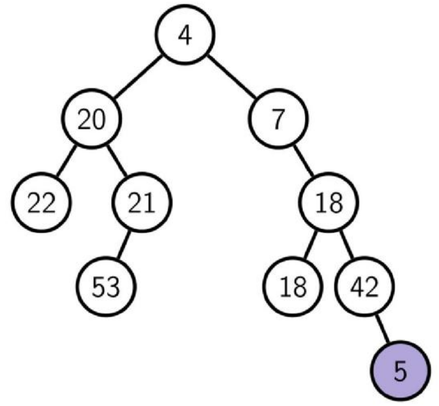


Двоичная (мин-)куча



- основное свойство (мин-)кучи: значение вершины $\leq$ значений её детей
- минимальное значение хранится в корне, поэтому GETMIN работает за время $O(1)$

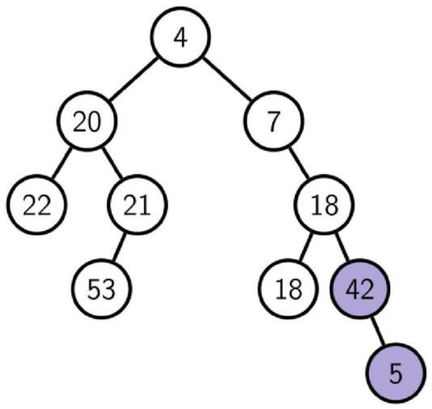
Вставка и просеивание вверх



- подвесим новый элемент листом в произвольное место

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i == 0 – мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

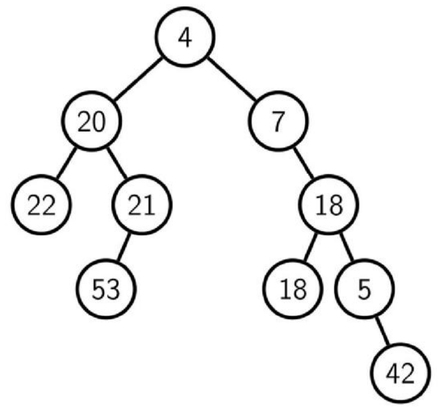

Вставка и просеивание вверх



- подвесим новый элемент листом в произвольное место
- начнём чинить свойство кучи, просеивая проблемную вершину вверх

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i == 0 – мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

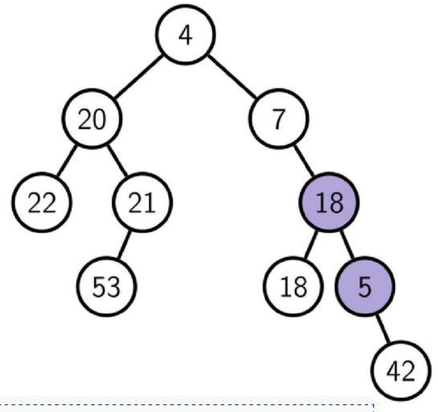

Вставка и просеивание вверх



- подвесим новый элемент листом в произвольное место
- начнём чинить свойство кучи, просеивая проблемную вершину вверх

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i == 0 – мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

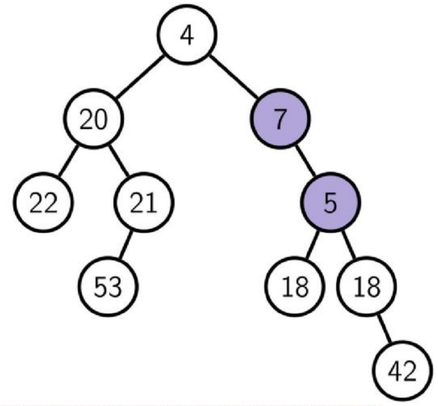
Вставка и просеивание вверх



- подвесим новый элемент листом в произвольное место
- начнём чинить свойство кучи, просеивая проблемную вершину вверх

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i == 0 – мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

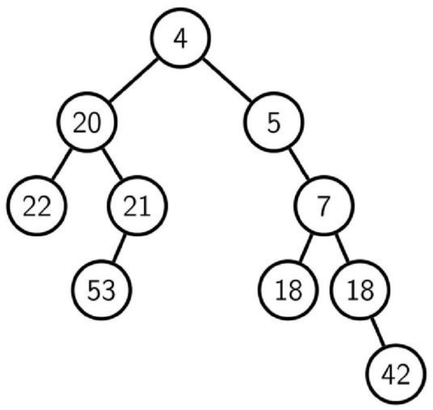

Вставка и просеивание вверх



- подвесим новый элемент листом в произвольное место
- начнём чинить свойство кучи, просеивая проблемную вершину вверх

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i == 0 – мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```


Вставка и просеивание вверх



- подвесим новый элемент листом в произвольное место
- начнём чинить свойство кучи, просеивая проблемную вершину вверх

Время работы операции вставки

- важный инвариант: в каждый момент времени свойство кучи нарушено не более чем в одной вершине

```
function siftDown(i : int):
    while 2 * i + 1 < a.heapSize     // heapSize – количество элементов в куче
        left = 2 * i + 1             // left – левый сын
        right = 2 * i + 2            // right – правый сын
        j = left
        if right < a.heapSize and a[right] < a[left]
            j = right
        if a[i] <= a[j]
            break
        swap(a[i], a[j])
        i = j
```

Время работы операции вставки

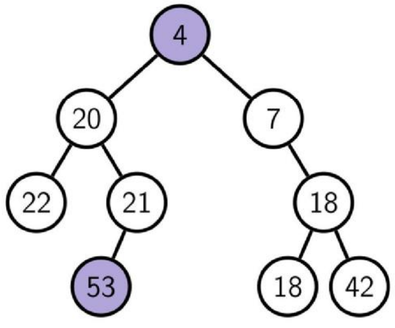
- важный инвариант: в каждый момент времени свойство кучи нарушено не более чем в одной вершине
- при просеивании вверх эта вершина становится ближе к корню

Время работы операции вставки

- важный инвариант: в каждый момент времени свойство кучи нарушено не более чем в одной вершине
- при просеивании вверх эта вершина становится ближе к корню
- время работы есть $O(h)$, где h — высота кучи

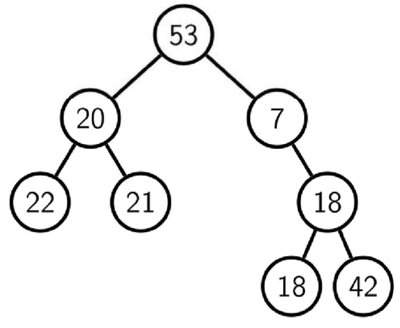
Извлечение минимума и просеивание вниз

- заменим корень на любой лист

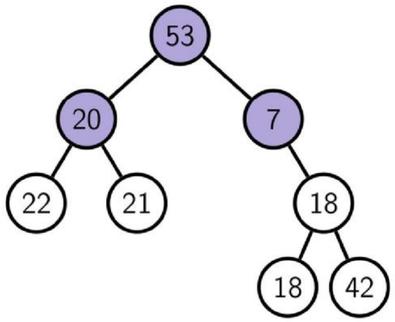


Извлечение минимума и просеивание вниз

- заменим корень на любой лист

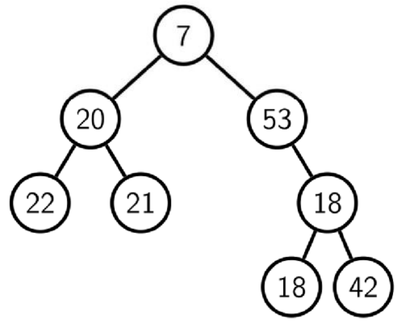


Извлечение минимума и просеивание вниз



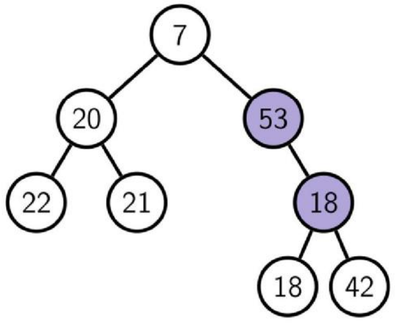
- заменим корень на любой лист

Извлечение минимума и просеивание вниз



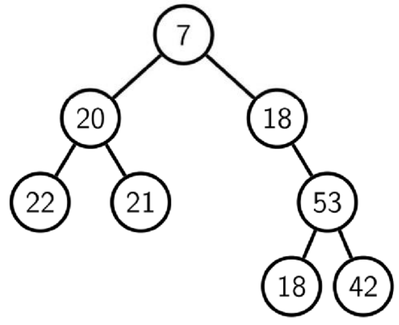
- заменим корень на любой лист
- обменяем проблемную вершину с меньшим из её детей, чтобы быть уверенными, что продолжать чинить кучу нужно только в одном из поддеревьев

Извлечение минимума и просеивание вниз



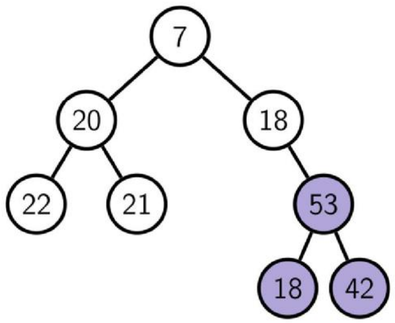
- заменим корень на любой лист
- обменяем проблемную вершину с меньшим из её детей, чтобы быть уверенными, что продолжать чинить кучу нужно только в одном из поддеревьев

Извлечение минимума и просеивание вниз



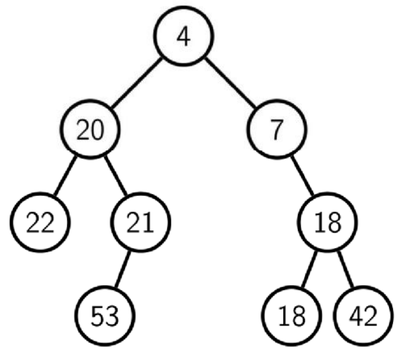
- заменим корень на любой лист
- обменяем проблемную вершину с меньшим из её детей, чтобы быть уверенными, что продолжать чинить кучу нужно только в одном из поддеревьев

Извлечение минимума и просеивание вниз

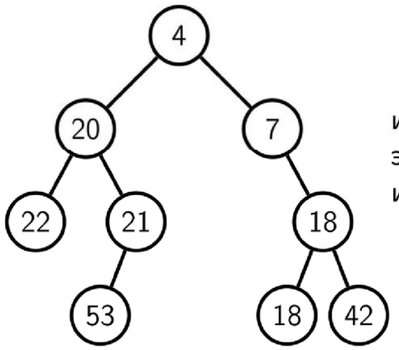


- заменим корень на любой лист
- обменяем проблемную вершину с меньшим из её детей, чтобы быть уверенными, что продолжать чинить кучу нужно только в одном из поддеревьев

Изменение приоритета

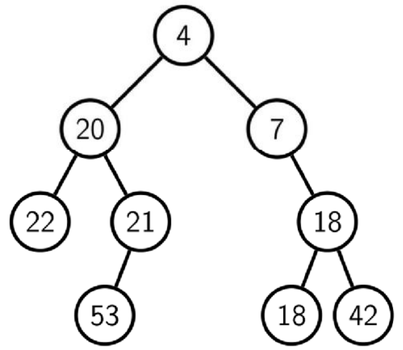


Изменение приоритета

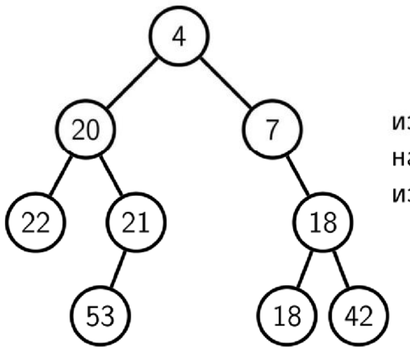


изменим приоритет и дадим элементу просеиться вниз или вверх

Удаление

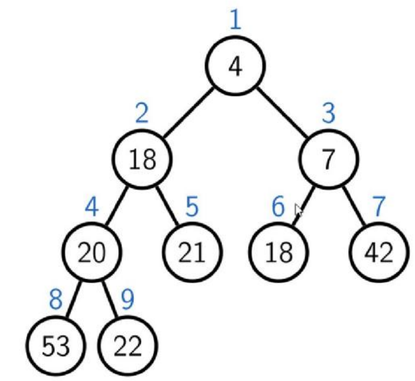


Удаление



изменим приоритет элемента на $-\infty$, просеим его вверх и извлечём минимум

Полное двоичное дерево и массив



1	2	3	4	5	6	7	8	9
4	18	7	20	21	18	42	53	22

Куча на массиве

- полное двоичное дерево: уровни заполняются слева направо; все уровни заполнены полностью, кроме, возможно, последнего

Куча на массиве

- полное двоичное дерево: уровни заполняются слева направо; все уровни заполнены полностью, кроме, возможно, последнего
- естественная нумерация вершин: сверху вниз, слева направо

PISL Lesson03. **Код Хаффмана** + heap

Куча на массиве

- полное двоичное дерево: уровни заполняются слева направо; все уровни заполнены полностью, кроме, возможно, последнего
- естественная нумерация вершин: сверху вниз, слева направо
- при добавлении элемента подвешиваем лист на последний уровень; при удалении отрезаем самый последний лист

PISL Lesson03. **Код Хаффмана** + heap

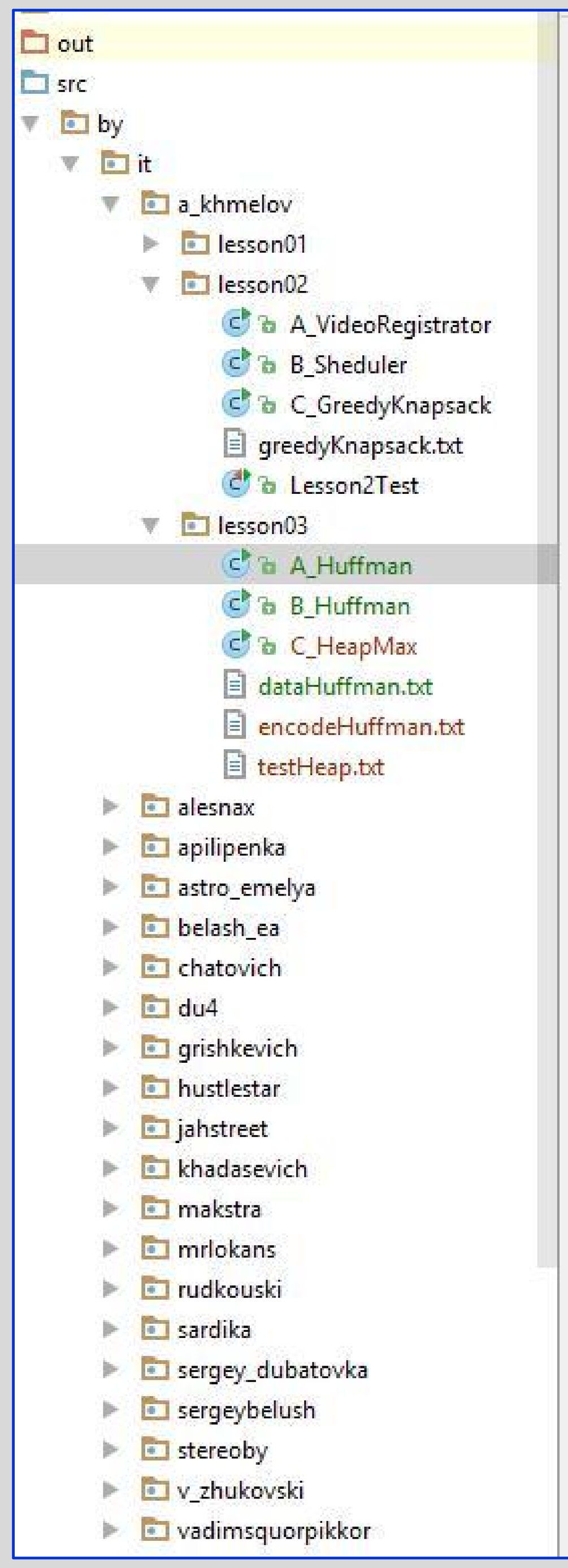
Куча на массиве

- полное двоичное дерево: уровни заполняются слева направо; все уровни заполнены полностью, кроме, возможно, последнего
- естественная нумерация вершин: сверху вниз, слева направо
- при добавлении элемента подвешиваем лист на последний уровень; при удалении отрезаем самый последний лист
- вершина i имеет родителя $\lfloor i/2 \rfloor$ и детей $2i$ и $2i+1$ (при вычислении данных индексов нужно проверять, что они попадают в отрезок $[1, n]$)

Построение кучи из «хаоса»

```
function heapify():
    for i = a.heapSize / 2 downto 0
        siftDown(i)
```

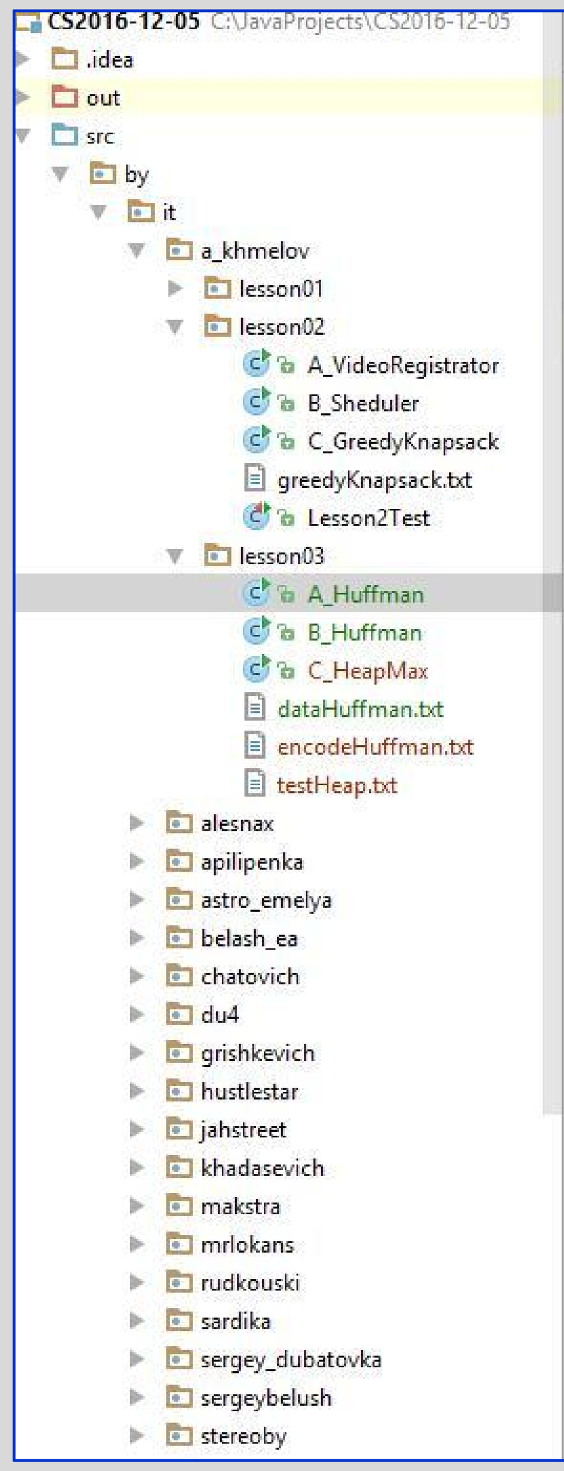

Задание A. **Кодирование Хаффмана**




```
//Lesson 3. A_Huffman.
//Разработайте метод encode(File file) для кодирования строки (код Хаффмана)

// По данным файла (непустой строке ss длины не более 104104),
// состоящей из строчных букв латинского алфавита,
// постройте оптимальный по суммарной длине беспрефиксный код.

// Используйте Алгоритм Хаффмана — жадный алгоритм оптимального
// безпрефиксного кодирования алфавита с минимальной избыточностью.

// В первой строке выведите количество различных букв kk,
// встречающихся в строке, и размер получившейся закодированной строки.
// В следующих kk строках запишите коды букв в формате "letter: code".
// В последней строке выведите закодированную строку. Примеры ниже

//        Sample Input 1:
//        a
//
//        Sample Output 1:
//        1 1
//        a: 0
//        0

//        Sample Input 2:
//        abacabad
//
//        Sample Output 2:
//        4 14
//        a: 0
//        b: 10
//        c: 110
```

```
A_Huffman | Main
"H:\Program Files\Java\jdk1.8.0_101\bin\java" ...
4 14
a: 0
b: 10
c: 110
d: 111
01001100100111

Process finished with exit code 0
```


Задание A. **Кодирование Хаффмана**




```
A_Huffman | encode()
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!     НАЧАЛО ЗАДАЧИ     !!!!!!!!!!!!!!!!!!!!!!!!!!!
    private String encode(File file) throws FileNotFoundException {
        //прочитаем строку для кодирования из тестового файла
        Scanner scanner = new Scanner(file);
        String s = scanner.next();

        //все комментарии от тестового решения были оставлены т.к. это задание A.
        //если они вам мешают их можно удалить

        Map<Character, Integer> count = new HashMap<>();
        //1. переберем все символы по очереди и рассчитаем их частоту в Map count
        [redacted]
            //для каждого символа добавим 1 если элемента еще нет или инкремент если есть.
            [redacted]
        }
        //2. перенесем все символы в приоритетную очередь в виде листьев
        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        [redacted]
        }

        //3. вынимая по два узла из очереди (для сборки родителя)
        //и возвращая этого родителя обратно в очередь
        //построим дерево кодирования Хаффмана
        [redacted]
```

```
A_Huffman | Main
"H:\Program Files\Java\jdk1.8.0_101\bin\java" ...
4 14
a: 0
b: 10
c: 110
d: 111
01001100100111

Process finished with exit code 0
```

Задание Б. Декодирование Хаффмана

```
// Lesson 3. B_Huffman.
// Восстановите строку по её коду и беспрефиксному коду символов.

// В первой строке входного файла заданы два целых числа
// kk и ll через пробел — количество различных букв, встречающихся в строке,
// и размер получившейся закодированной строки, соответственно.
//
// В следующих kk строках записаны коды букв в формате "letter: code".
// Ни один код не является префиксом другого.
// Буквы могут быть перечислены в любом порядке.
// В качестве букв могут встречаться лишь строчные буквы латинского алфавита;
// каждая из этих букв встречается в строке хотя бы один раз.
// Наконец, в последней строке записана закодированная строка.
// Исходная строка и коды всех букв непусты.
// Заданный код таков, что закодированная строка имеет минимальный возможный размер.
//
//         Sample Input 1:
//         1 1
//         a: 0
//         0

//         Sample Output 1:
//         a
```

```
//         Sample Input 2:
//         4 14
//         a: 0
//         b: 10
//         c: 110
//         d: 111
//         01001100100111

//         Sample Output 2:
//         abacabad
```

```
Run:  B_Huffman  Main
"H:\Program Files\Java\jdk1.8.0_101\bin\java" ...
abacabad

Process finished with exit code 0
```


Задание Б. Декодирование Хаффмана

```
it
  a_khmelov
    lesson01
    lesson02
      A_VideoRegistrator
      B_Sheduler
      C_GreedyKnapsack
      greedyKnapsack.txt
      Lesson2Test
    lesson03
      A_Huffman
      B_Huffman
      C_HeapMax
      dataHuffman.txt
      encodeHuffman.txt
      testHeap.txt
  alesnax
  apilipenka
  astro_emelya
  belash_ea
  chatovich
  du4
  grishkevich
  hustlestar
  jahstreet
  khadasevich
  makstra
  mrlokans
  rudkouski
  sardika
  sergey_dubatovka
```

```
    private String decode(File file) throws FileNotFoundException {
        //прочитаем строку для кодирования из тестового файла
        Scanner scanner = new Scanner(file);
        Integer count = scanner.nextInt();
        Integer length = scanner.nextInt();
        [redacted]
        StringBuilder result=new StringBuilder();
        [redacted]
        return result.toString();
    }
```

```
Run:  B_Huffman  Main
"H:\Program Files\Java\jdk1.8.0_101\bin\java" ...
abacabad

Process finished with exit code 0
```


Задание C. Max-heap

```
import java.io.FileNotFoundException;

// Lesson 3. C_Heap.
// Задача: построить max-кучу = пирамиду = бинарное сбалансированное дерево на массиве.
// ВАЖНО! НЕЛЬЗЯ ИСПОЛЬЗОВАТЬ НИКАКИЕ КОЛЛЕКЦИИ, КРОМЕ ARRAYLIST (его можно, но только для массива)

//      Проверка проводится по данным файла
//      Первая строка входа содержит число операций 1 ≤ n ≤ 100000.
//      Каждая из последующих nn строк задают операцию одного из следующих двух типов:

//      Insert x, где 0 ≤ x ≤ 1000000000 — целое число;
//      ExtractMax.

//      Первая операция добавляет число x в очередь с приоритетами,
//      вторая — извлекает максимальное число и выводит его.

//      Sample Input:
//      6
//      Insert 200
//      Insert 10
//      ExtractMax
//      Insert 5
//      Insert 500
//      ExtractMax
//
//      Sample Output:
//      200
//      500
```


Задание C. Max-heap

```
private class MaxHeap {
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! НАЧАЛО ЗАДАЧИ !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!1
    //тут запишите ваше решение.
    //Будет мало? Ну тогда можете его собрать как Generic и/или использовать в варианте B
    private List<Long> heap = new ArrayList<>();

    int siftDown(int i) { //просеивание вверх

        return i;
    }

    int siftUp(int i) { //просеивание вниз

        return i;
    }

    void insert(Long value) { //вставка
    }

    Long extractMax() { //извлечение и удаление максимума
        Long result = null;

        return result;
    }
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!1
}
```

```
//эта процедура читает данные из файла, ее можно не менять.
Long findMaxValue(InputStream stream) {
    Long maxValue=0L;
    MaxHeap heap = new MaxHeap();
    //прочитаем строку для кодирования из тестового файла
    Scanner scanner = new Scanner(stream);
    Integer count = scanner.nextInt();
    for (int i = 0; i < count; ) {
        String s = scanner.nextLine();
        if (s.equalsIgnoreCase("extractMax")) {
            Long res=heap.extractMax();
            if (res!=null && res>maxValue) maxValue=res;
            System.out.println();
            i++;
        }
        if (s.contains(" ")) {
            String[] p = s.split(" ");
            if (p[0].equalsIgnoreCase("insert"))
                heap.insert(Long.parseLong(p[1]));
            i++;
        //System.out.println(heap); //debug
        }
    }
    return maxValue;
```

```
6
Insert 200
Insert 10
ExtractMax
Insert 5
Insert 500
ExtractMax
```