Урок 8

Деревья. Хеш-функция

Двоичные деревья поиска. Проход по дереву. Хеш-функция.

Введение

На этом уроке мы рассмотрим два алгоритма шифрования данных: алгоритм Хаффмана и хеш-функции. В основе алгоритма Хаффмана лежит структура данных «дерево». Поэтому сначала разберем ее.

В разработке видеоигр структуры деревьев используются для разделения пространства. Это позволяет разработчику быстро находить расположенные рядом объекты, не проверяя весь игровой мир. Несмотря на то, что деревья — фундаментальные структуры в информатике, на практике в большинстве стандартных библиотек нет непосредственной реализации контейнеров на основе деревьев.

Представление двоичного дерева

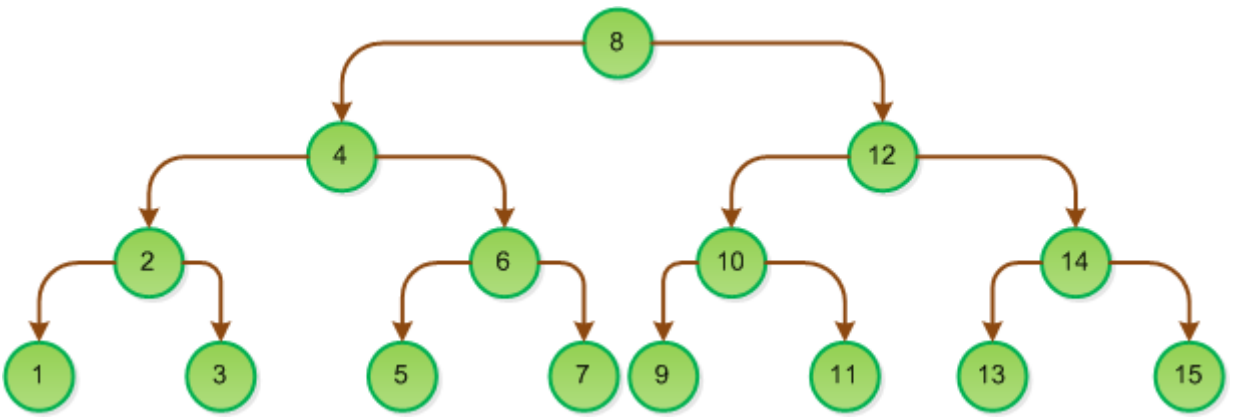
Иерархия позволяет управлять. В древности люди иерархически организовали военные структуры, которые почти в неизменном виде существуют и в наши дни. Рассмотрим упрощенную структуру армейского механизма. Восемь-десять солдат составляют отделение во главе с командиром. Три отделения — это взвод, три взвода — рота, три роты — батальон, три батальона — полк, три полка — дивизия. Далее строгое утроение нарушается. Несколько дивизий образуют армию. А несколько армий — это военный округ. Несколько округов — военное государство. На каждом уровне управления командир не общается со всеми бойцами, а только с теми, кто находится в его непосредственном подчинении. Командир отделения дает распоряжения 7–9 солдатам. Командир взвода — трем командирам отделений, командир роты — трем командирам взводов, и так вплоть до главнокомандующего. Такая организация позволяет грамотно и быстро управлять множеством подчиненных.

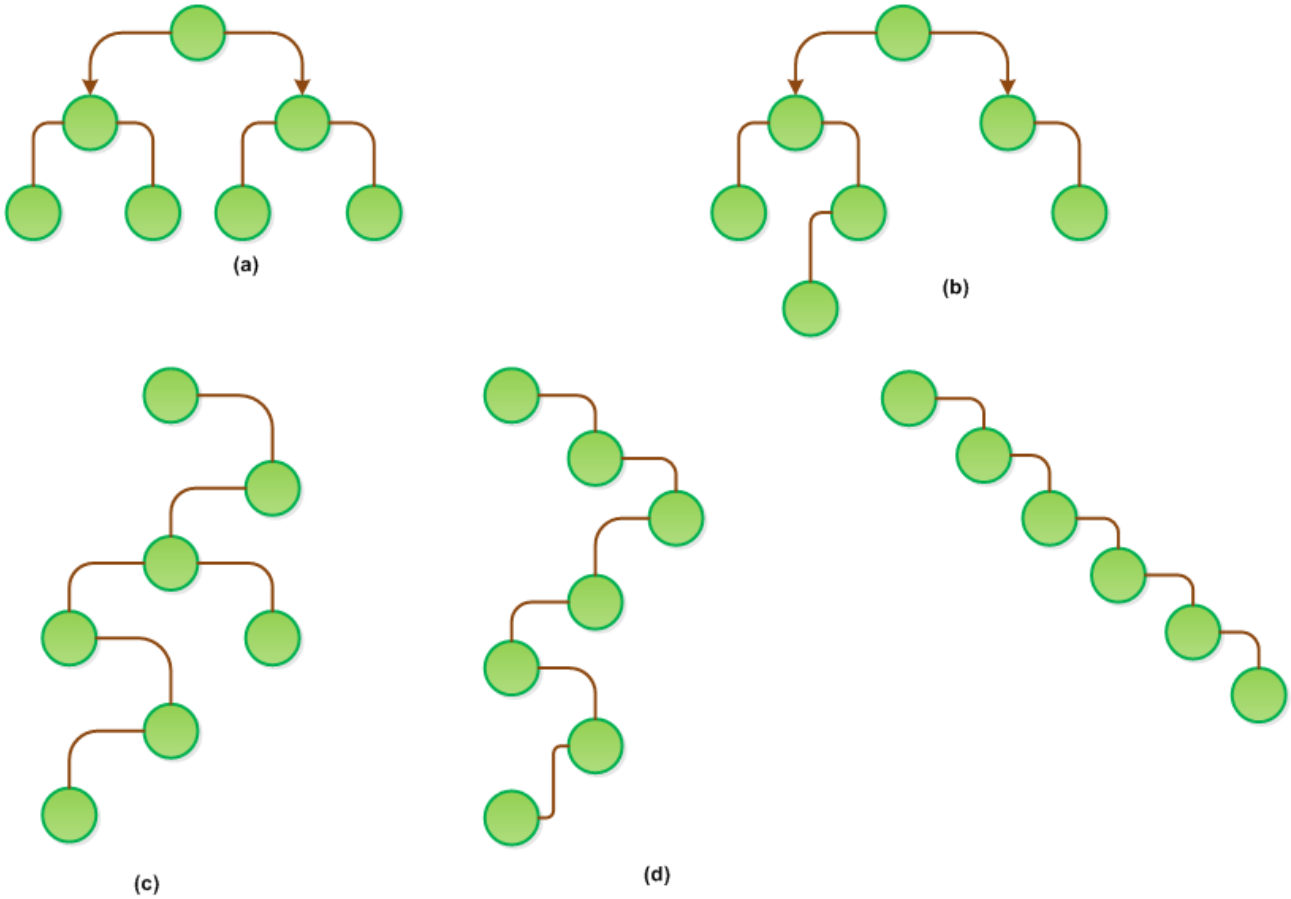
Другой пример иерархии — файловая система, которую поддерживают практически все ОС.

Изучим приемы работы с аналогами реальных иерархических структур в сфере информационных технологий. В программировании они моделируются с помощью деревьев (бинарных деревьев).

Бинарные деревья растут «корнями вверх». Вершина, из которой идет разветвление, традиционно называется корнем (root) — в дереве он может быть только один. Точки ветвления — это вершины, а ветки, соединяющие их, — ребра. Вершины, которыми заканчиваются ветки и из которых не исходит ни одного ребра, — это листья. Ребра, соединяющие вершины, четко ориентированы, то есть имеют начало и конец. В каждую вершину может входить не более одного ребра.

Реализации одного и того же бинарного дерева:



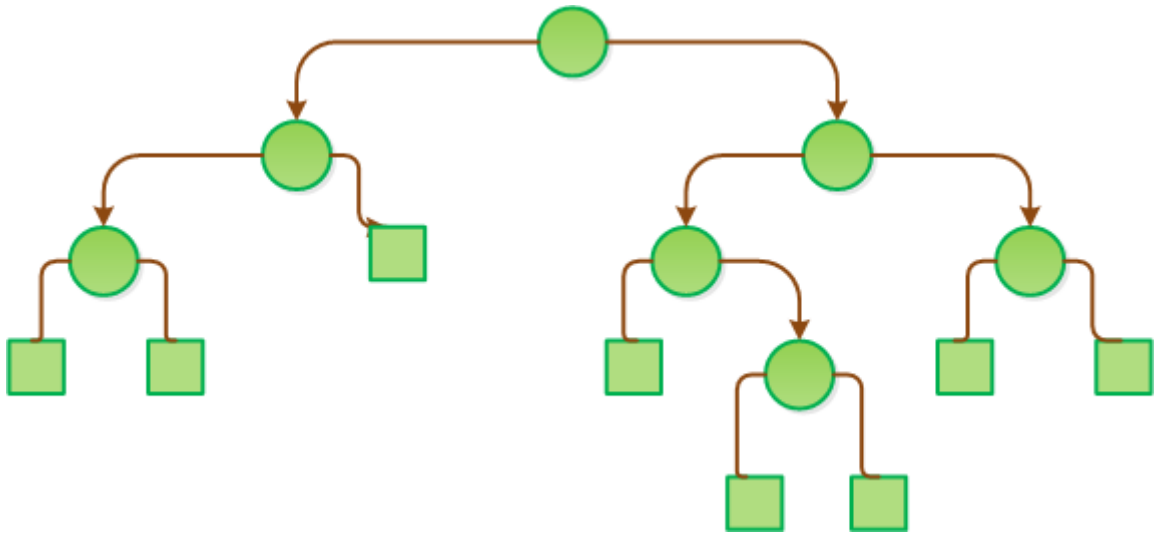


Разновидности бинарных деревьев:

- **полное (расширенное) бинарное дерево** — где каждый узел (за исключением листьев) имеет по два дочерних узла;
- **идеальное бинарное дерево** — полное бинарное дерево, в котором все листья находятся на одной высоте;
- **сбалансированное бинарное дерево** — где высота двух поддеревьев для каждого узла отличается максимум на единицу. Глубина такого дерева вычисляется как двоичный логарифм log(n), где n — общее число узлов;
- **вырожденное дерево** — где каждый узел имеет всего один дочерний. Фактически это связный список;
- **бинарное поисковое дерево (BST)** — где для каждого узла выполняется условие, чтобы все узлы в левом поддереве были меньше этого узла, а все узлы в правом поддереве — больше. Такое дерево еще называют упорядоченным.

Нам наиболее интересны упорядоченные бинарные деревья, потому что именно они позволяют организовать быстрый доступ к данным.

Пример расширенного дерева



Проход по двоичному дереву

По дереву двигаются сверху вниз, от корня к листьям. Такой порядок означает обход всего дерева, от корня до листьев. После рассмотрения очередного узла продолжается движение вглубь дерева, пока не будут пройдены все потомки текущего узла. Такой алгоритм требует рекурсивного перебора, чтобы рассмотреть каждый узел дерева в отдельности и остановиться, только когда дойдем до листьев.

Создадим два класса: первый для узла (**Node**), а второй для дерева (**Tree**):

```
class Node:
    def __init__(self, value=None, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def __repr__(self):
        return f'Node[{self.value:^5}]'


class Tree:
    def __init__(self):
        self.root = None
```

Рассмотрим ряд функций для работы с бинарными деревьями.

Чтобы определить высоту дерева, потребуется пройти от корня сначала по левому поддереву, потом по правому, сравнить их величины и выбрать максимальное значение. И не забыть прибавить к нему единицу (корневой элемент). Функцию прохода по дереву можно реализовать с помощью как итеративного, так и рекурсивного перебора. На последнем мы и остановимся:

```
def height(self, node):
    if node is None:
        return 0
    else:
        left_height = self.height(node.left)
        right_height = self.height(node.right)

        if left_height > right_height:
            return left_height + 1
        else:
            return right_height + 1
```

Ширина дерева — это максимальное количество узлов, расположенных на одной высоте.

```
def get_max_width(self, root):
    max_wdth = 0
    i = 1
    h = self.height(root)
    while i <= h:
        width = self.get_width(root, i)
        if width > max_wdth:
            max_wdth = width
        i += 1

    return max_wdth

def get_width(self, root, level):
    if root is None:
        return 0
    if level == 1:
        return 1
    elif level > 1:
        return self.get_width(root.left, level - 1) +
self.get_width(root.right, level - 1)
    self.get_width(root.right, level - 1)
```

Чтобы вывести все дерево, нужно пройти по нему от корневого элемента до всех листьев. Функция распечатки дерева:

```
# функция для распечатки элементов на определенном уровне дерева
def print_given_level(self, root, level):
    if root is None:
        return
    if level == 1:
        print(root, end='')
    elif level > 1:
        self.print_given_level(root.left, level - 1)
        self.print_given_level(root.right, level - 1)

# функция для распечатки дерева
def print_level_order(self, root):
    h = self.height(root)
    i = 1
    while i <= h:
        self.print_given_level(self.root, i)
        print()
        i += 1
```

Можем сравнить и два бинарных дерева. Реализуем эту функцию вне класса:

```
def same_tree(a, b):
    if a is None and b is None:
        return 1
    elif a and b:
        return (
                a.value == b.value and
                same_tree(a.left, b.left) and
                same_tree(a.right, b.right)
                )
    return 0
```

Посмотрим, как код будет выглядеть в виде полных классов: **Node** описывает элемент дерева, а **Tree** — дерево.

```
class Node:
    def __init__(self, value=None, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def __repr__(self):
        return f'Node[{self.value:^5}]'


class Tree:
    def __init__(self):
        self.root = None

    # функция для добавления узла в дерево
    def new_node(self, value):
```

```
        return Node(0, None, None)


    # функция для вычисления высоты дерева
    def height(self, node):
        if node is None:
            return 0
        else:
            left_height = self.height(node.left)
            right_height = self.height(node.right)

            if left_height > right_height:
                return left_height + 1
            else:
                return right_height + 1

    # функция для распечатки элементов на определенном уровне дерева
    def print_given_level(self, root, level):
        if root is None:
            return
        if level == 1:
            print(root, end='')
        elif level > 1:
            self.print_given_level(root.left, level - 1)
            self.print_given_level(root.right, level - 1)

    # функция для распечатки дерева
    def print_level_order(self, root):
        h = self.height(root)
        i = 1
        while i <= h:
            self.print_given_level(self.root, i)
            print()
            i += 1

    # функция для вычисления ширины дерева
    def get_max_width(self, root):
        max_wdth = 0
        i = 1
        h = self.height(root)
        while i <= h:
            width = self.get_width(root, i)
            if width > max_wdth:
                max_wdth = width
            i += 1

        return max_wdth

    def get_width(self, root, level):
        if root is None:
            return 0
        if level == 1:
            return 1
```

```
        elif level > 1:
            return self.get_width(root.left, level - 1) +
self.get_width(root.right, level - 1)
        self.get_width(root.right, level - 1)
```

Теперь создадим дерево, заполним его элементами и посмотрим, как отработают функции:

```
t = Tree()
t.root = t.new_node(8)
t.root.left = t.new_node(4)
t.root.right = t.new_node(12)
t.root.left.left = t.new_node(2)
t.root.left.right = t.new_node(6)
t.root.right.left = t.new_node(10)
t.root.right.right = t.new_node(14)
t.root.left.left.left = t.new_node(0)
t.root.left.left.right = t.new_node(3)
t.root.left.right.left = t.new_node(5)
t.root.left.right.right = t.new_node(7)
t.root.right.left.left = t.new_node(9)
t.root.right.left.right = t.new_node(11)
t.root.right.right.left = t.new_node(13)
t.root.right.right.right = t.new_node(15)

t.print_level_order(t.root)
print(f'высота: {t.height(t.root)}')
print(f'ширина: {t.get_max_width(t.root)}')
```

Вывод:

В отличие от поиска в линейной структуре, поиск по дереву не требует перебора всех элементов, поэтому занимает значительно меньше времени.

Применение алгоритма Хаффмана и структуры дерева в алгоритмах шифрования

В основе алгоритма кодирования Хаффмана — частота появления символа в последовательности. Символ, который встречается чаще всего, получает новый, очень маленький код. Наиболее редко фигурирующий символ, напротив, получает очень длинный код.

В следующем примере символ будет иметь длину 8 бит.

Размер символа выбирается в соответствии с ожидаемой строкой ввода. Важно помнить, что размер кода символа прямо пропорционален выбранному размеру символа.

Рассмотрим применение алгоритма Хаффмана. Допустим, у нас есть строка «beep boop beer!», для которой на каждый знак тратится по одному байту. Это означает, что вся строка занимает 15*8 = 120 бит памяти. После кодирования — 40 бит.

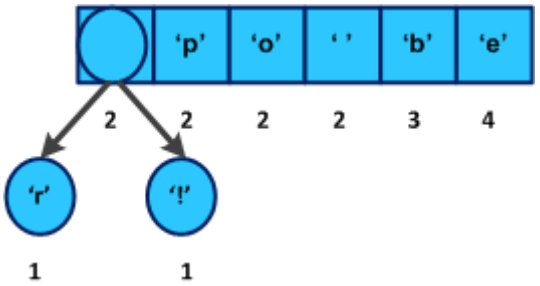
Чтобы получить код для каждого символа, нужно построить бинарное дерево, где каждый лист будет содержать символ. Символы с меньшей частотой будут дальше от корня, чем символы с большей частотой.

Начнем реализовывать алгоритм Хаффмана. Посчитаем частоты всех символов:

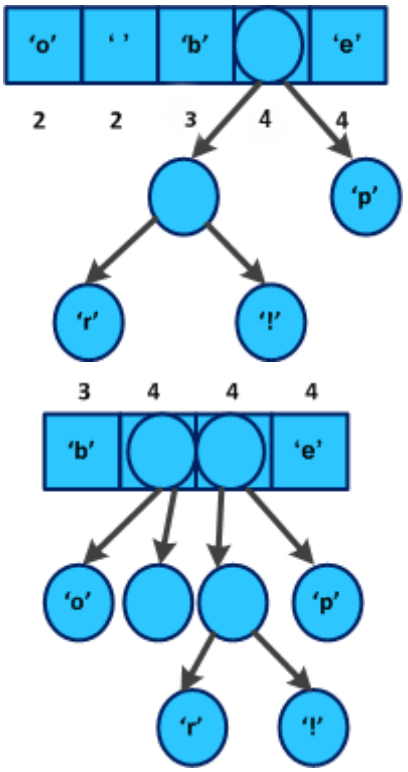
Символ	Частота
'b'	3
'e'	4
'p'	2
' '	2
'o'	2
'r'	1
'!'	1

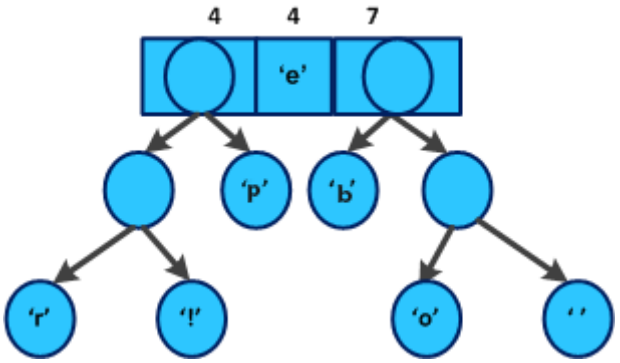
После вычисления частот создадим узлы бинарного дерева для каждого знака и добавим их в очередь, используя частоту в качестве приоритета. В начале очереди окажутся символы с наименьшей частотой, с наибольшей — в конце.

Берем два первых элемента из очереди и связываем их, создавая новый узел дерева. В нем они оба будут потомками, а приоритет нового узла будет равен сумме их приоритетов. Добавим получившийся узел обратно в очередь. Позиция для добавления не должна нарушать частотный приоритет.

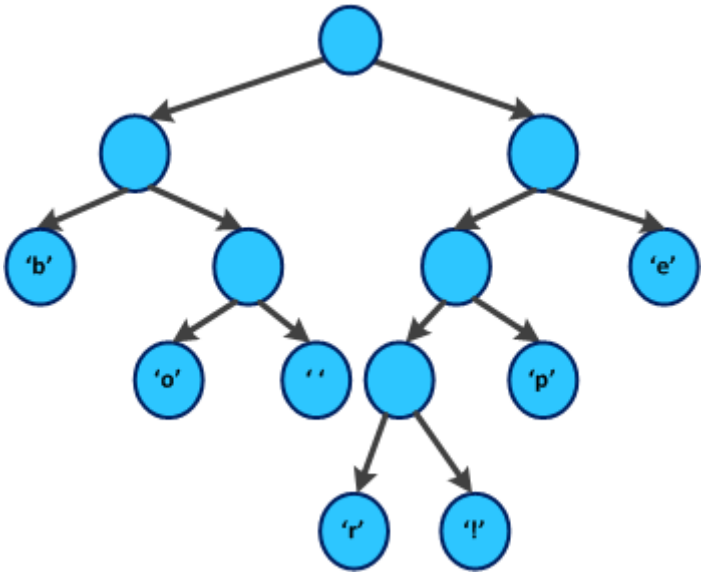


Повторим те же шаги и получим последовательно:

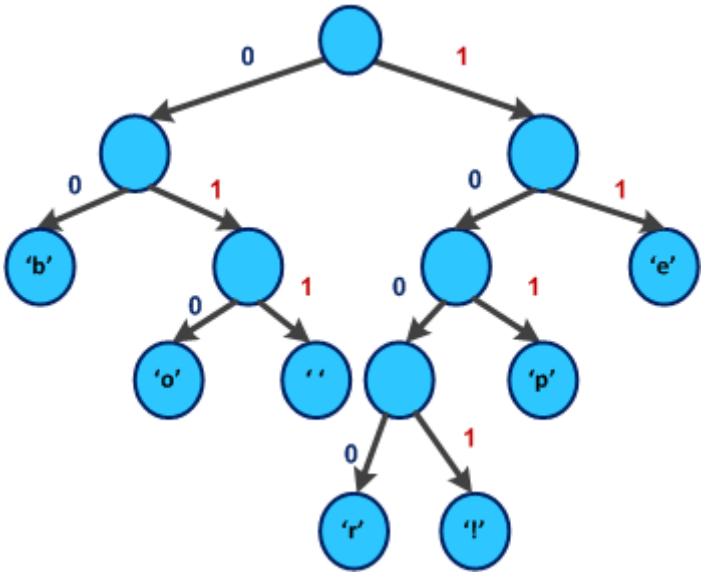




В конце свяжем два последних элемента — получится итоговое дерево:



Чтобы получить код для каждого символа, надо просто пройтись по дереву, для каждого перехода добавляя 0, если идем налево, и 1 — если направо:



Получим следующие коды для символов:

Символ	Код
'b'	00
'e'	11
'p'	101
' '	011
'o'	010
'r'	1000
'!'	1001

Мы построили таблицу кодирования Хаффмана. Она содержит каждый символ и его код — это делает кодирование более эффективным.

Важно иметь в виду, что каждый код не является префиксом для кода другого символа. В нашем примере, если 00 — это код для 'b', то 000 не может оказаться чьим-либо кодом — иначе получим конфликт.

Входная строка: «beep boop beer!»

Входная строка в бинарном виде: «0110 0010 0110 0101 0110 0101 0111 0000 0010 0000 0110 0010 0110 1111 0110 1111 0111 0000 0010 0000 0110 0010 0110 0101 0110 0101 0111 0010 0010 000».

Закодированная строка: «0011 1110 1011 0001 0010 1010 1100 1111 1000 1001».

Как видим, ASCII-версия строки и закодированная версия существенно различаются.

Хеширование

Хеширование используется практически во всех приложениях криптографии.

Хеш-функция предназначена для «сжатия» произвольного сообщения или набора данных, записанных в двоичном формате, в битовую комбинацию фиксированной длины.

Криптографическая хеш-функция — это криптостойкая хеш-функция, то есть соответствующая требованиям шифрования. В криптографии хеш-функции применяются для:

- контроля целостности данных при их передаче или хранении;
- аутентификации источника данных.

Аутентификация — это метод проверки подлинности источника данных.

Хеш-функцией называется любая функция **h(X) -> Y**. Она легко вычислима и соответствует условию, что для любого сообщения **M** значение **h(M) = H** (сжатое сообщение) имеет фиксированную битовую длину (**X** — множество всех сообщений, **Y** — множество двоичных векторов фиксированной длины).

Хеш-функции строят на основе одношаговых сжимающих функций двух переменных **y = f(x1, x2)**, где **x1**, **x2** и **y** — двоичные векторы длины **n1**, **n2** и **m** соответственно, где **n** — длина свертки, а **m** — длина блока сообщения.

Для получения значения **h(M)** сообщение сначала разбивается на блоки длины **m**. Если длина сообщения не кратна **m**, то последний блок специальным образом заполняется. Затем к полученным блокам **M1**, **M2**,.., **MN** применяют последовательную процедуру вычисления сжатия:

Ho = v,

Hi = f(Mi,Hi-1), i = 1,.., N,

h(M) = HN

Здесь **v** — константа, которую часто называют инициализирующим вектором. Она может представлять собой секретную константу или набор случайных данных — выборку даты и времени, например.

При таком подходе свойства хеш-функции полностью определяются свойствами одношаговой сжимающей функции.

Выделяют два важных вида криптографических хеш-функций: ключевые и бесключевые. Первые называют кодами аутентификации сообщений, вторые — кодами обнаружения ошибок. С помощью дополнительных средств (например, шифрования) они гарантируют целостность данных. Эти хеш-функции можно применять в системах как с доверяющими друг другу пользователями, так и с недоверяющими.

Хеш-функция

Хеш-таблицы используются, чтобы хранить неупорядоченную последовательность объектов и быстро выбирать один из них, не пересматривая все возможные варианты. Как мы знаем, для хранения упорядоченной информации обычно используются массивы.

Таблицы хеширования формируются с помощью хеш-функций, распределяющих элементы между несколькими категориями. Именно хеш-функция определяет, к какой категории будет относиться объект последовательности. Если требуется найти объект, то его хеш-функция однозначно определяет его категорию, и поиск производится не по всему объему данных, а только внутри конкретной категории. Это в разы сокращает время работы программы.

Данные в таблице хеширования идентифицируются ключами. Но может наступить момент, когда в каждой категории окажется слишком мало места для большого количества объектов, и это замедлит доступ к данным. В такой ситуации происходит автоматическая перекомпоновка таблицы. По сути, это пример реализации работы с памятью, как с кучей.

Хеш-функция позволяет взять данные любой длины и построить по ним короткий «цифровой отпечаток пальца». Можно сказать, что хеш-функция отображает строки на числа. Длина значения хеш-функции не зависит от длины исходного текста. Например, при алгоритме SHA-1 длина этого отпечатка составляет 160 бит.

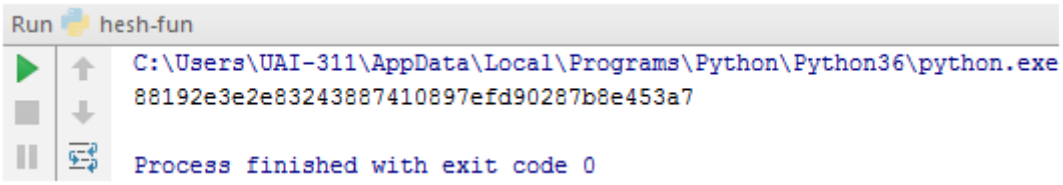
Важно, чтобы хеш-функция была последовательна, то есть неизменно возвращала один и тот же результат для одинаковых входных данных. Если это условие будет нарушено, мы не найдем свой элемент в хеш-таблице.

Не стоит изобретать велосипед: на каждом языке высокого уровня уже реализованы хеш-функции. Рассмотрим работу хеш-функции SHA-1:

```
import hashlib

print(hashlib.sha1(b"Hello, Bob!").hexdigest())
```

Результат:

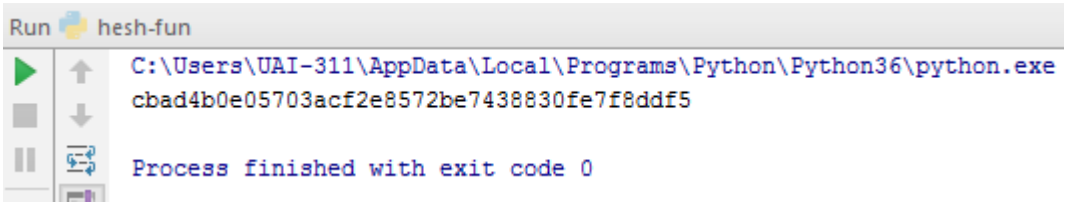


Идея хеш-функции в том, что она работает только в одном направлении. Ее легко подсчитать для любой исходной информации. Но найти документ по уже готовому значению хеш-функции (получить такое же значение) практически невозможно. Если изменить в документе хотя бы одну букву, хеш изменится полностью:

```
import hashlib

print(hashlib.sha1(b"Hello, Bob?").hexdigest())
```

Результат:



Поэтому хеш-функцию и используют для контроля целостности сообщений.

Например, Елена решила написать письмо Сергею:

1. Елена подсчитывает для каждого своего сообщения значение SHA-1 и вкладывает его в конверт.
2. Сергей тоже может подсчитать самостоятельно SHA-1 текста и сравнить свой результат с результатом Елены. Так он удостоверится, что сообщение не было изменено где-то по пути к нему.
3. Но кто-то может перехватить письмо, изменить сообщение, подсчитать для него SHA-1 и приложить к письму.
4. Сергей сверит значения и ничего не заметит.

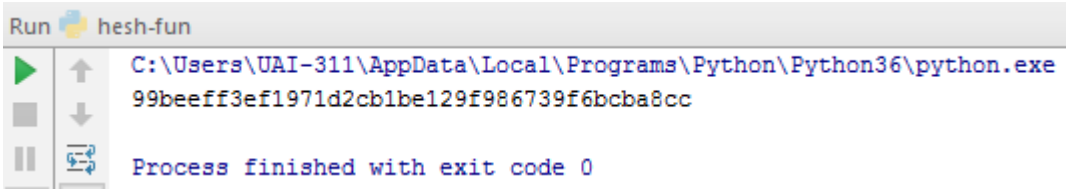
Возникает проблема проверки подлинности. Елена и Сергей при встрече решили, что при подсчете SHA-1 они будут дописывать к тексту секретное слово — например, «Secret». Теперь тот, кто перехватит письмо, не сможет скорректировать его хеш, так как не знает слово.

Рассмотрим конкретный пример реализации:

```
import hashlib

print(hashlib.sha1(b"Secret" + b"Hello, Bob").hexdigest())
```

В результате получаем:



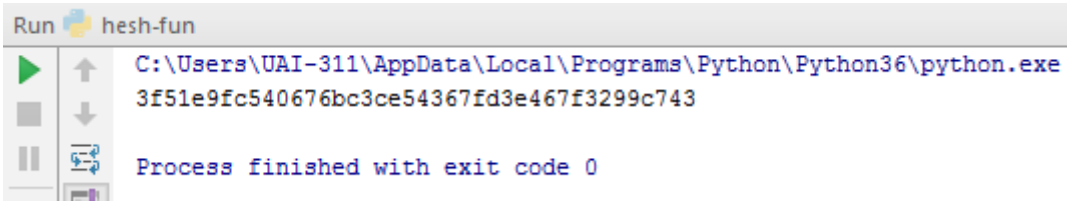
Возникает следующая проблема: злоумышленник не может изменить тело сообщения, зато легко допишет в конце «P.S. На самом деле все это чушь, Сергей» и просто досчитает хеш от остатка.

Чтобы защититься от этого, усложняем функцию:

```
import hashlib

s = hashlib.sha1(b"Hello, Bob").hexdigest()
print(hashlib.sha1(b"Secret" + bytes(s.encode('utf-8'))).hexdigest())
```

В результате:



Теперь дописывание сообщения полностью изменит исходные данные для «внешнего» вызова SHA-1.

Хеш-функция SHA-1

Логика выполнения SHA-1

Рассмотрим всю логику работы алгоритма SHA-1.

На входе мы получаем сообщение максимальной длины 2^{64} бит и создаем в качестве выхода дайджест-сообщение длиной 160 бит.

Алгоритм состоит из следующих шагов:

Шаг 1: Добавить недостающие биты.

Длина сообщения должна быть кратна 448 по модулю 512. Добавление осуществляется всегда, даже если сообщение уже имеет нужную длину. Таким образом, число добавляемых битов находится в диапазоне от 1 до 512.

Добавление состоит из единицы, за которой следует необходимое количество 0.

Шаг 2: Добавить длину сообщения.

К сообщению добавляется блок из 64 битов. Этот блок будет восприниматься как беззнаковое 64-битное целое. Он содержит длину исходного сообщения до добавления.

Результат первых двух шагов — сообщение, длина которого кратна 512 битам. Расширенное сообщение может быть представлено как последовательность 512-битных блоков **Y0**, **Y1**, . . . , **YL**. Общая длина расширенного сообщения равна **L * 512 бит**. Таким образом, результат кратен шестнадцати 32-битным словам.

Шаг 3: Задать начальные значения SHA-1 буфера.

Используется 160-битный буфер для хранения промежуточных и окончательных результатов хеш-функции. Он может быть представлен как пять 32-битных регистров A, B, C, D и E. Они инициализируются следующими шестнадцатеричными числами:

A = 0x67452301

B = 0xEFCDAB89

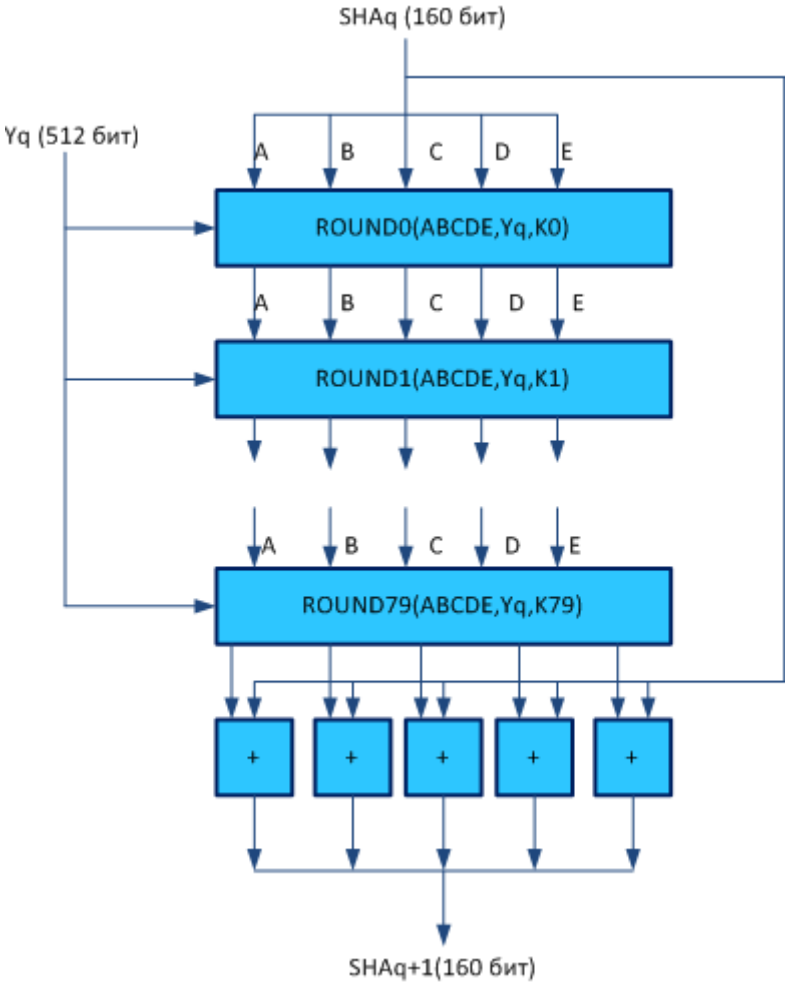
C = 0x98BADCFE

D = 0x10325476

E = 0xC3D2E1F0

Шаг 4: Произвести обработку сообщения в 512-битных блоках.

Основой алгоритма является модуль, состоящий из 80 итераций с одинаковой структурой. Он обозначен как HSHA.



На каждой итерации цикл получает на входе текущий 512-битный обрабатываемый блок **Yq** и 160-битное значение буфера **ABCDE** и изменяет содержимое этого буфера.

В каждой итерации используется дополнительная константа **Kt**, которая принимает только четыре различных значения.

Для получения $\mathbf{SHA_{q+1}}$ выход 80-го цикла складывается со значением $\mathbf{SHA_q}$. Сложение по модулю 2^{32} выполняется независимо для каждого из пяти слов в буфере с каждым из соответствующих слов в $\mathbf{SHA_q}$.

Шаг 5: Завершить цикл обработки.

После обработки всех 512-битных блоков выходом L-ой стадии является 160-битный дайджест сообщения.

Рассмотрим более детально логику в каждом из 80 циклов обработки одного 512-битного блока. Цикл можно представить в виде:

$$\mathbf{A, B, C, D, E\ (CLS_5\ (A) + f_t\ (B, C, D) + E + W_t + K_t), A, CLS30\ (B), C, D}$$

A, B, C, D, E — пять слов из буфера;

t — номер цикла, 0 <= t <= 79;

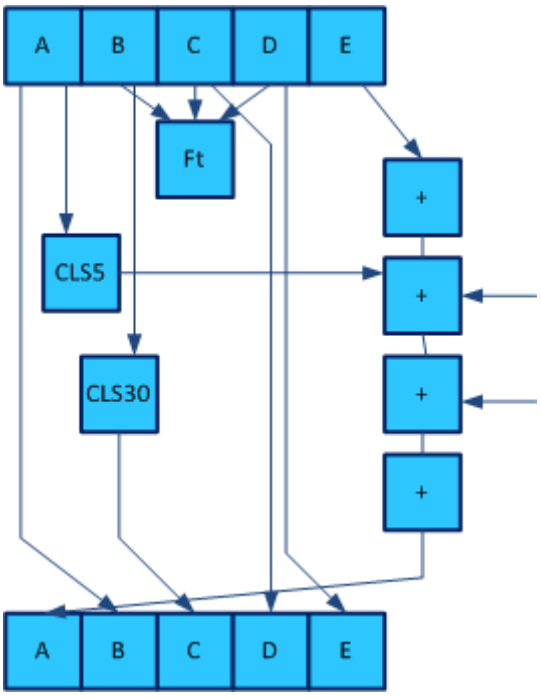
$\mathbf{f_t}$ — элементарная логическая функция;

CLS_s — циклический левый сдвиг 32-битного аргумента на **s** битов;

W_t — 32-битное слово, полученное из текущего входного 512-битного блока;

K_t — дополнительная константа;

\+ — сложение по модулю 2^{32}.



Каждая элементарная функция получает на входе три 32-битных слова и создает одно. Элементарная функция выполняет набор побитовых логических операций, то есть n-ый бит выхода является функцией от n-ых битов трех входов. Функции следующие:

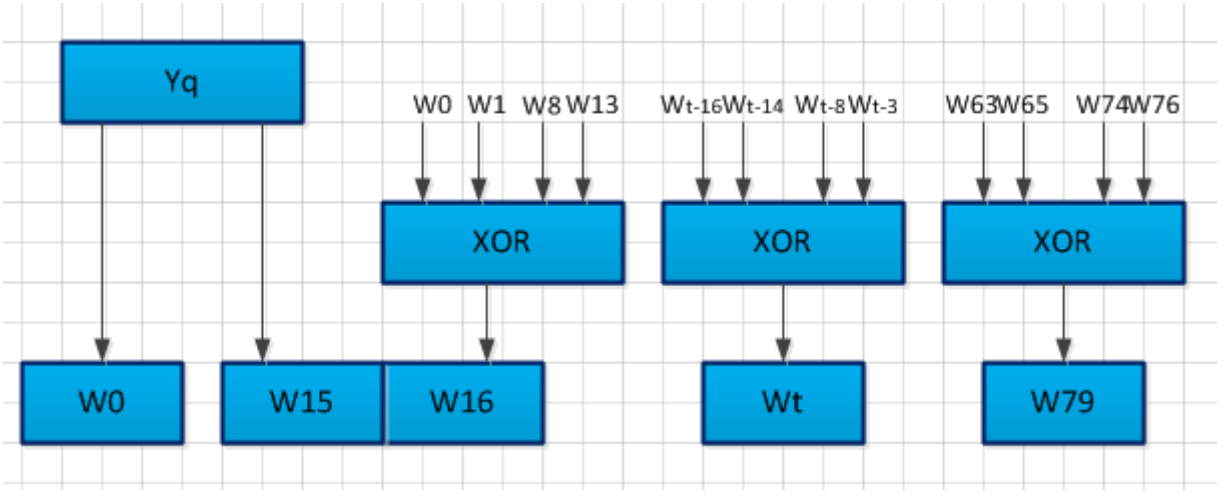
Номер цикла	ft (B, C, D)
(0 <= t <= 19)	(B∧C) v (~B∧D)
(20 <= t <= 39)	B+C+D
(40 <= t <= 59)	(B∧C) v (B∧D) v (C∧D)
(60 <= t <= 79)	B+C+D

Используются только три функции:

- 0 <= t <= 19;
- 20 <= t <= 39;
- 60 <= t <= 79.

А функция 40 <= t <= 59 является истинной, если два или три аргумента истинны.

32-битные слова **Wt** получаются из очередного 512-битного блока сообщения следующим образом:



Первые 16 значений **Wt** берутся непосредственно из 16 слов текущего блока. Оставшиеся значения определяются следующим образом:

W(t) = W(t-16)⊕W(t-14)⊕W(t-8)⊕W(t-3)

В первых 16 циклах вход состоит из 32-битного слова данного блока. Для оставшихся 64 циклов вход состоит из **XOR** нескольких слов из блока сообщения.

Алгоритм SHA-1 можно суммировать следующим образом:

$\mathbf{SHA_0 = IV}$

$\mathbf{SHA_{q+1} = \sum_{32}(SHA_q, ABBCDE_q)}$

$\mathbf{SHA = SHA_{L-1}}$,

IV — начальное значение буфера **ABCDE**;

$\mathbf{ABCDE_q}$ — результат обработки q-ого блока сообщения;

L — число блоков в сообщении, включая поля добавления и длины;

$\sum_{32}$ — сумма по модулю 2^{32}, выполняемая отдельно для каждого слова буфера;

SHA — значение дайджеста сообщения.

Сравнение SHA-1 и MD5

Рассмотрим различия еще двух наиболее распространенных алгоритмов хеширования — SHA-1 и MD5. Оба произошли от MD4, поэтому имеют много общего. Можно суммировать ключевые различия между алгоритмами.

	MD5	SHA-1
Длина дайджеста	128 бит	160 бит
Размер блока обработки	512 бит	512 бит
Число итераций	64 (4 цикла по 16 итераций в каждом)	80
Число элементарных логических функций	4	3
Число дополнительных констант	64	4

Ключевые отличия алгоритмов MD5 и SHA-1:

1. Длина дайджеста MD5 на 32 бита меньше, чем SHA-1.
2. SHA-1 должен выполняться приблизительно на 25 % медленнее, чем MD5 на той же аппаратуре. Оба алгоритма выполняют сложение по модулю 2^{32}, они рассчитаны на 32-битную архитектуру.
3. SHA-1 применяет одношаговую структуру, а MD5 использует четыре структуры.
4. MD5 использует little-endian схему для интерпретации сообщения как последовательности 32-битных слов. SHA-1 задействует схему big-endian. Этих подходы не дают особых преимуществ.

Примеры для закрепления материала

Задача 1. Сравнить строки с помощью хеширования

Главное применение хеш-функции — это быстрое сравнение двух подстрок, то есть проверка целостности данных. На этом основываются все остальные алгоритмы с хешами.

Следующая функция производит сравнение двух строк, вычисляя их хеш и сравнивая его.

```
# Сравнить строки с помощью хеширования
import hashlib

def is_eq_str(a, b):
    ha = hashlib.sha1(a.encode('utf-8')).hexdigest()
    hb = hashlib.sha1(b.encode('utf-8')).hexdigest()
    return ha == hb
```

Задача 2. Провести поиск подстроки

Хеши позволяют быстро искать подстроку в строке. Для этого применяется алгоритм Рабина-Карпа.

Алгоритм решения задачи:

1. Дана строка **s** длины **n**, в которой мы хотим найти все вхождения строки **t** длины **m**.
2. Найдем хеш строки **t** (всей строки целиком).
3. Найдем хеши всех префиксов строки **s**.

4. Будем двигаться по строке **s** окном длины **m**, сравнивая подстроки **s(i...i+m-1)**.

Программная реализация:

```
# Провести поиск подстроки в строке
import hashlib

def rabin_karp(s, t):
    len_sub = len(t)
    h_subs = hashlib.sha1(t.encode('utf-8')).hexdigest()
    for i in range(len(s) - len_sub + 1):
        if h_subs == hashlib.sha1(s[i:i+len_sub].encode('utf-8')).hexdigest():
            return i

    return -1
```

Подведем итоги:

1. Структура дерева отражает иерархическую связь между объектами реального мира.
2. Структура дерева повышает производительность алгоритмов поиска, так как поиск ведется по упорядоченным данным.
3. Для сжатия информации можно подготовить дерево с применением алгоритма Хаффмана.
4. Таблицы хеширования ускоряют доступ к данным во время работы программы. Важно понимать, когда стоит использовать иерархию, а когда уместнее распределить объекты по категориям.
5. В криптографии алгоритмы хеш-функций уже реализованы в виде стандартных функций.
6. Алгоритм SHA-1 простой. Строка шифрования SHA-1 составляет 160 бит.

Практическое задание

1. Определить количество различных подстрок с использованием хеш-функции. Задача: на вход функции дана строка, требуется вернуть количество различных подстрок в этой строке.

   Примечание: в сумму не включаем пустую строку и строку целиком.

2. Закодировать любую строку по алгоритму Хаффмана.

Используемая литература

1. https://www.python.org.
2. Марк Лутц. Изучаем Python. 4-е издание.