**Модуль 7. Структуры данных  (4 ак. ч.)**

* Понятие о структуре данных.
* Список и запись как основные структуры данных.
* Таблица и другие практически важные структуры данных.
* Практикум. Представление записей и таблиц в программе на языке Python.

# Введение

СД – способ организации данных в памяти компьютера (списки, словари, кортежи, множества)

Классический пример – картотека.



На карточках напишем номер, адрес, ФИО – получим адресную книгу.

Напишем предмет и остаточную стоимость – получим инвентарную опись.

Интернет-магазин.

Как эти данные будут размещаться в памяти компьютера. Если их 10, 100, 1000, миллион?

Будут ли эти карточки добавляться, редактироваться?

Как быстро? Возможен ли быстрый поиск? Сортировка? Идеальной сортировки!

Нет идеальной структуры данных. Часто нужен быстрый доступ и медленной добавление.

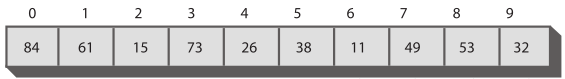
Или, наоборот, быстрое добавление и небыстрый доступ. Идеала не бывает.

Вся языки программирования, которые предлагаю те или иные структуры не совсем настоящие, не те которые работаю внутри, под капотом

На самом деле мы будет работать с обертками (или можно сказать алгоритмами) над настоящими структурами данных. Гибрид, можно так сказать.

Массив

М- данные хранятся последовательно. Классический массив – гомогенный. Все данные – одного типа. Целые, вещественные, символьные.



В пайтоне нет классического массива. Все, что мы имеем – это ссылку на первый элемент списка.

m = [1,2,3]

id(m)

1998918250112

id(m[0])

140729398911784

id(m[1]) ]) #32 бита = 4 байта

140729398911816

id(m[2]) #32 бита = 4 байта

140729398911848

m.\_\_sizeof\_\_()

Пачиму?

72

Идеальный массив – гомогенный и фиксированной длины.

Потому-что все элемент (значения) должны идти четко друг за другом.

Т.о. массив идеально работает. Сложность доступа константа О(1). При точечном доступе.

Но нам то нужно добавлять, вставлять в середину.

Массив в этом случае дает проблемы. Удалить элемент из середины. А потом все элементы сдвинуть с учетом освободившегося места.

А добавлять еще хуже! Не важно куда. Места может не быть. Т.к. классический массив фиксированной длины. Т.е. нужно будет создавать новый массив, четкая копия первого с добавленным новым элементом.

А потом эту память освободить. Классическая работа на языке С++

Вывод – что и зачем мы делаем. Если у нас картотека фиксированная и я ничего туда не добавляю или добавляю редко, а ищу что-то точечно (по индекс?) – массив идеальный вариант.

А если я постоянно добавляю, а читаю редко?

Массив не поможет. Поможет список. Пайтоновский список – отлично подойдет. Не во всех языках есть списки.

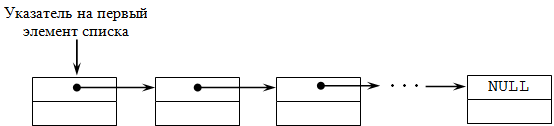
List – коллекция элементов, хранящихся на соответствующей позиции, но только по отношению к остальным. Не подряд, последовательно как в массиве. Он может храниться где угодно.

В любом месте памяти вообще. Куда положили, там и лежит.

А как они находятся. Каждый элемент имеет ссылку на следующий элемент.

Последний ссылается на None (Null)

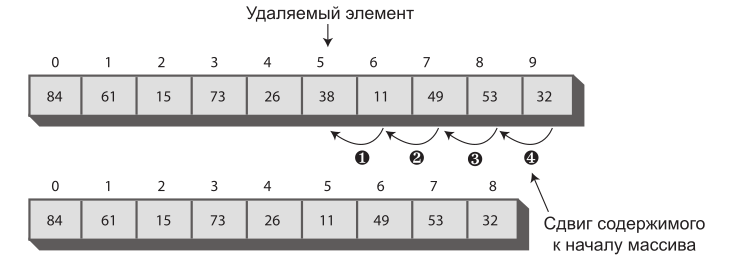
Связный список



У нас есть сложность в отличии от массива (у которого доступ к элементам константа), при точечном подходе, нужно пробежаться по всем ссылкам. Только там можно получить доступ к значению.

Этот минус, перекрывается кучей плюсов.

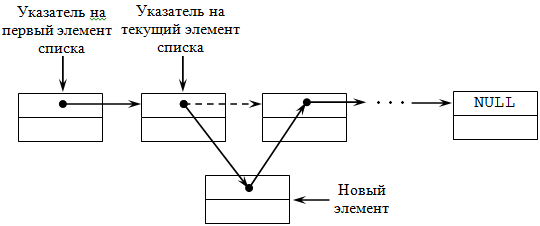
Если мне нужно у массива удалить элемент, нужно сместить всю последовательность, то в списке просто переписывается указатель с удаленного элемента, на следующий за ним.



Эксперимент

m = [100, 200, 300]  
print(m)  
  
for i in m:  
 print(i, "=", id(i), end=',')  
print()

m.pop(1)  
  
for i in m:  
 print(i, "=", id(i), end=',')



m.insert(1,400)  
m.append(500)  
print()  
for i in m:  
 print(i, "=", id(i), end=',')

Результат:

[100, 200, 300]

100 = 140729398914952,200 = 140729398918152,300 = 2732444138480,

100 = 140729398914952,300 = 2732444138480,

100 = 140729398914952,400 = 2732444138416,300 = 2732444138480,500 = 2732444138384,

for obj in gc.get\_objects():  
 print(id(obj), obj)  
  
  
import ctypes  
a = "hello world"  
print (ctypes.cast(id(a), ctypes.py\_object).value)

Весь блокчейн построен на связных списках. Все транзакции хранятся в связном списке. Человек покупает, в конец это добавляется.

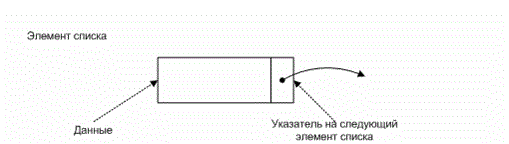
Майнер майнят крипту, которая летит в конец списка.

Итак, список изучен. Мы как настоящие прогеры должны уметь сами запрогать свою список

*# ТЗ  
# list()  
# add.(item)  
# search  
# is\_empry  
# size  
# append(item)  
# index (item)  
# insert (pos, item)  
# pop()  
# pop(pos)  
# del  
# порядок взаимного расположения элементов*

Итак, приступим.

Основной элемент списка-узел



Но кроме узла, важно само значение (данные) и ссылка на следующий элемент списка!

В первую очередь мы должны описать этот узел

1) Узел с конструктором

class Node:  
 def \_\_init\_\_(self,data):  
 self.\_\_data = data  
 self.\_\_next = Node

2) Экземпляр списка должен уметь получать и отдавать значения

def set\_data(self, data):  
 self.\_\_data = data  
  
def get\_data(self):  
 return self.\_\_data

3) Описываем хрестоматийный узел. Ссылочную модель

def set\_next(self, node):  
 self.\_\_next = node  
  
def get\_next(self):  
 return self.\_\_next

с узлом все. Тестим

node = Node(42)  
print(node.get\_data()) # 42

Чего не хватает для списка? Где он? А его еще нет. Если всего лишь узел. Классическо-хрестоматийный узел односвязного списка.

class LinkedList():  
 def \_\_init\_\_(self):  
 self.\_\_head = None  
  
 def is\_empty(self):  
 return self.\_\_head is None  
  
 def add(self, item):   
 node = Node(item)  
 node.set\_next(self.\_\_head)  
 self.\_\_head = node  
  
  
my\_list = LinkedList()  
  
node = Node(42)  
print(node.get\_data())

print(my\_list.is\_empty())

Пишем метод size

def size(self):  
 current = self.\_\_head  
 count = 0  
 while not current is None:  
 current = current.get\_next()  
 count += 1  
 return count

Тестим:

my\_list = LinkedList()  
my\_list.add(1)  
my\_list.add(10)  
my\_list.add(20)  
print(my\_list.size())

Реализуем поиск

def search(self, item):  
 current = self.\_\_head  
 while not current is None:  
 if current.get\_data() == item:  
 return True  
 else:  
 current = current.get\_next()  
 return False

Тестим:

print(my\_list.search(10))

Переопределяем метод \_\_repr\_ (str)

def \_\_repr\_\_(self):  
 if self.\_\_head is None:  
 return '[]'  
 current = self.\_\_head  
 output = f'{current.get\_data()},'  
 while not current.get\_next() is None:  
 current = current.get\_next()  
 output += str(current.get\_data()) + ','  
 return f'[{output[:-1]}]'

Тестим

print(my\_list)

my\_list.add(101)  
print(my\_list)

Реализуем remove. Здесь уже не всё так просто. С добавление методом add всё просто. А здесь нужно сделать ряд вещей. Первое – обойти весь список, чтобы найти удаляемый элемент. Когда мы его нашли – мы его удаляем. Очень похоже на search().

Но цикл в методе search() остановиться до того, как current получит None.

Когда мы выйдем из цикла current будет ссылаться на узел, который нужно удалить.

Мы можем его заменить у этого элемента значение, каким-нибудь банальным маркером, что этого элемента нет. Так то же можно, так делают.

Но здесь будет проблема – количество узлов будет не совпадать, нам нужно будет переписать весь size.

Решение, лучше удалить узел, удалив весь узел. Чтобы удалить весь узел, нужно изменить ссылку его предшественника, чтобы она ссылалась на узел после него.

Т.е на руках нужно иметь предшественника и его последователя через одного. Через удаляемого.

Но в нашем списке нет возможности перемещаться назад. В реальности для этого существуют дабл-структуры, позволяющие перемещаться туда-сюда.

def remove(self, item):  
 current = self.\_\_head  
 prev = None  
 while True:  
 if current.get\_data() == item:  
 break  
 else:  
 prev = current  
 current = current.get\_next()  
 if prev is None:  
 *# удаление первого узла* self.\_\_head = current.get\_next() *# перепрыгиваем через голову* else:  
 prev.set\_next(current.get\_next())

Тестим

my\_list = LinkedList()  
my\_list.add(10)  
my\_list.add(20)  
my\_list.add(30)  
  
my\_list.remove(20)  
print(my\_list)

Блокчейн почти готов!

## Практикум

Создайте и опишите метод append, который добавляет узел в конец списка.

Какова временнАя сложность метода append?

Скорее всего, вы создали append метод с O(n). Если вы добавите в класс LinkedList некую переменную, то сможете сделать его с O(1). Будьте внимательны! Чтобы сделать это по-настоящему правильно, вам нужно рассмотреть несколько специальных случаев, которые могут потребовать модификации метода add.

Если останется время, попробуйте реализовать методы:

* insert: принимает элемент и позицию, перед которой вставляется переданный элемент
* index: принимает значение в качестве аргумента, возвращает его индекс (позицию элемента в списке)
* pop: удаляет и возвращает последний элемент списка

Помните, что каждый из них должен учитывать место изменения: голова списка или его остаток. Также эти методы требуют номера позиции в списке (мы исходим из того, что нумерация индексов начинается с нуля).

Решение

Для метода append

def append(self, item):  
 node = Node(item)  
 current = self.\_\_head  
 prev = None  
 while current is not None:  
 prev = current  
 current = current.get\_next() *# бегам до конца! неэффективно* prev.set\_next(node)  
*#простое решение сделать ссылку на конец*

Для этого в конструкторе класса допишем self.\_\_tail = None

class LinkedList:  
 def \_\_init\_\_(self):  
 self.\_\_head = None  
 **self.\_\_tail = None**

в метод add создание ссылки на хвост при её отсутствии

def add(self, item):  
 node = Node(item)  
 node.set\_next(self.\_\_head)  
 self.\_\_head = node  
 **if self.\_\_tail is None:  
 self.\_\_tail = node**

Метод append стал гораздо проще. Из O(n) получили константу О(1)

def append(self, item):  
 node = Node(item)  
 current = self.\_\_tail  
 current.set\_next(node)  
 self.\_\_tail = node

Тестим:

my\_list.append(100)  
print(my\_list)

Все остальные методы решаются по аналогии с предыдущими

Метод index – возвращает первое вхождение элемента

def index(self, item):  
 if self.\_\_head is None:  
 return -1  
 current = self.\_\_head  
 pos = 0  
 while True:  
 if current.get\_data() == item:  
 return pos  
 else:  
 current = current.get\_next()  
 pos += 1

Необходимо учесть, что мы реализовали список без обработки ошибок.

Дополнительный вопрос – как отсортировать наш список.

Только писать собственный встроенный!

Оптимизация метода append

def append(self, item):  
 **if self.\_\_head is None:  
 self.add(item)  
 return**   
 node = Node(item)  
 current = self.\_\_tail  
 current.set\_next(node)  
 self.\_\_tail = node

Ссылки

*DataFrame*  — табличная структура данных, доступная в библиотеке Python pandas. Это одна из наиболее важных структур данных для алгоритмов. Она используется для обработки классических структурированных данных. Рассмотрим таблицу (табл. 2.5).

Таблица 2.5

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | name (имя) | age (возраст) | decision (решение) |
| 1 | Fares | 32 | True |
| 2 | Elena | 23 | False |
| 3 | Steven | 40 | True |

Теперь представим эту таблицу с помощью DataFrame.

Простейший DataFrame может быть создан с помощью следующего кода:

**>>> import pandas as pd**

**>>> df = pd.DataFrame([**

**... ['1', 'Fares', 32, True],**

**... ['2', 'Elena', 23, False],**

**... ['3', 'Steven', 40, True]])**

**>>> df.columns = ['id', 'name', 'age', 'decision']**

**>>> df**

*id name age decision 0 1 Fares 32 True*

1. *2 Elena 23 False*
2. *3 Steven 40 True*

Обратите внимание, что в данном коде df.column — это список, в котором содержатся имена столбцов.

DataFrame используются и в других популярных языках и фреймворках для реализации табличной структуры данных. Примерами могут служить язык программирования R и платформа Apache Spark.

Терминология DataFrame

Ознакомимся с терминологией, необходимой для работы с DataFrame:

* *Ось*. В документации pandas один столбец или строка DataFrame называется осью (axis).
* *Метка*. DataFrame позволяет отмечать как столбцы, так и строки так называемой меткой (label).

Создание подмножества DataFrame

По сути, существуют два основных способа создания подмножества DataFrame (пусть это будет подмножество с именем myDF):

z выбор столбца; zzвыбор строки.

Рассмотрим их по очереди.

Выбор столбца

При работе с алгоритмами машинного обучения важно использовать правильный набор признаков. Далеко не все доступные нам признаки могут понадобиться на разных этапах алгоритма. В Python отбор признаков происходит путем выбора столбцов.

Получить доступ к столбцу можно с помощью его имени (атрибута *name*), как показано ниже:

**>>> df[['name','age']]**  *name age 0 Fares 32*

1. *Elena 23*
2. *Steven 40*

Позиция столбца является детерминированной. Доступ к нему по его расположению можно получить следующим образом:

**>>> df.iloc[:,3]**

1. *True*
2. *False*
3. *True*

Обратите внимание, что в этом коде мы извлекаем первые три строки DataFrame.

Выбор строки

Каждая строка DataFrame соответствует точке данных в пространстве задачи. Чтобы создать подмножество из имеющихся элементов данных, необходимо выбрать строки. Существуют два метода создания подмножества:

z указать расположение строк; zzзадать критерии фильтра.

Подмножество строк может быть получено по расположению следующим образом:

**>>> df.iloc[1:3,:]**  *id name age decision*

1. *2 Elena 23 False*
2. *3 Steven 40 True*

Данный код вернет первые две строки и все столбцы.

Чтобы создать подмножество с помощью фильтра‚ мы должны указать критерии выбора в одном или нескольких столбцах. Это происходит следующим образом:

**>>> df[df.age>30]**  *id name age decision 0 1 Fares 32 True*

*2 3 Steven 40 True*

**>>> df[(df.age<35)&(df.decision==True)]**

*id name age decision 0 1 Fares 32 True*

Обратите внимание, что этот код создает подмножество строк, удовлетворяющее условию, указанному в фильтре.

Матрица

*Матрица* — это двумерная структура данных с фиксированным количеством столбцов и строк.

На каждый элемент матрицы можно ссылаться по его столбцу и строке.

В Python матрицу можно создать с помощью массива numpy:

**>>> myMatrix = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])**

**>>> print(myMatrix)**

*[[11 12 13]*

*[21 22 23]*

*[31 32 33]]*

**>>> print(type(myMatrix))**

*<class 'numpy.ndarray'>*

Этот код создает матрицу, содержащую три строки и три столбца.

Операции с матрицами

Существует множество операций, доступных для матричных данных. Давайте транспонируем матрицу из предыдущего примера. Для этого используем функцию transpose(), которая преобразует столбцы в строки, а строки в столбцы:

**>>> myMatrix.transpose()** *array([[11, 21, 31], [12, 22, 32],*

*[13, 23, 33]])*

Отметим, что матричные операции часто используются при обработке мультимедийных данных.

Теперь, когда мы узнали о структурах данных в Python, перейдем к абстрактным типам данных.

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

В широком смысле абстракция — принцип, используемый для определения сложных систем с точки зрения их общих базовых функций. Применение этой концепции при создании структур данных приводит к появлению *абстрактных типов данных* (AТД[[1]](#footnote-1)). Используя АТД, мы получаем универсальную, независимую от реализации структуру данных. Это позволяет написать более простой и чистый код алгоритма, не углубляясь в детали разработки. АТД м ожно реализовать на любом языке программирования, например C++, Java и Scala. В этом разделе мы будем использовать АТД в Python. Начнем с вектора.

Вектор

*Вектор* — это одномерная структура для хранения данных‚ одна из самых популярных в Python. В Python имеются два способа создания векторов.

* Использование списка Python. Самый простой способ создания вектора — применить список Python следующим образом:

>>> myVector = [22,33,44,55]

**>>> print(myVector)**

* 1. *33 44 55]*

**>>> print(type(myVector))**

*<class 'list'>*

Этот код создает список из четырех элементов.

* Использование массива numpy. Еще один популярный способ создания вектора — применение массивов NumPy, как показано ниже:

**>>> myVector = np.array([22,33,44,55])**

**>>> print(myVector)**

* 1. *33 44 55]*

**>>> print(type(myVector))**

*<class 'numpy.ndarray'>*

Обратите внимание, что мы создали MyVector, используя np.array.

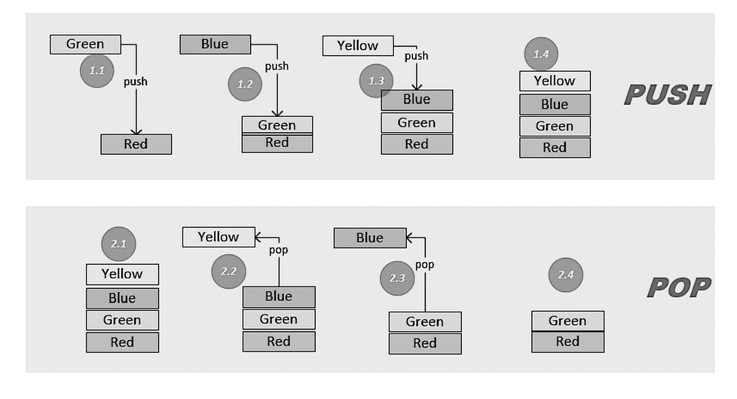
В Python мы используем нижнее подчеркивание при написании целых чисел, разделяя их на разряды. Это делает их удобочитаемыми и уменьшает вероятность ошибки‚ что особенно важно при работе с большими числами. Например, один миллиард можно представить так: a=1\_000\_000\_000.

Стек

*Стек* — это линейная структура данных для хранения одномерного списка. Элементы в стеке могут обрабатываться по принципу LIFO (Last­In, First­Out: *«последним пришел* — *первым ушел»*) либо по принципу FILO (First­In, Last­Out: *«первым пришел* — *последним ушел»*. Порядок добавления и удаления элементов определяет характер стека. Новые элементы могут добавляться и удаляться только с одного конца списка. Ниже приведены операции со стеками:

* *isEmpty.* Возвращает true, если стек пуст;
* *push.* Добавляет новый элемент; zz*pop*. Возвращает элемент, добавленный последним, и удаляет его.

На рис. 2.4 показано, как операции push() и pop() можно использовать для добавления и удаления данных из стека.



**Cтек**

Рис. 2.4

В верхней части рис. 2.4 показано использование операции push() для добавления элементов в стек. На шагах *1.1*, *1.2* и *1.3* операция push() используется три раза для добавления трех элементов в стек. В нижней части рисунка демонстрируется извлечение сохраненных значений из стека. На шагах *2.2* и *2.3* операция pop() используется для извлечения двух элементов из стека в формате LIFO.

Давайте создадим класс с именем Stack‚ в котором опишем все операции, связанные с классом stack. Код этого класса будет выглядеть следующим образом:

class Stack: def \_\_init\_\_(self): self.items = [] def isEmpty(self):

return self.items == [] def push(self, item):

self.items.append(item) def pop(self):

return self.items.pop() def peek(self):

return self.items[len(self.items)-1] def size(self): return len(self.items)

Чтобы поместить четыре элемента в стек, можно использовать следующий код (рис. 2.5).

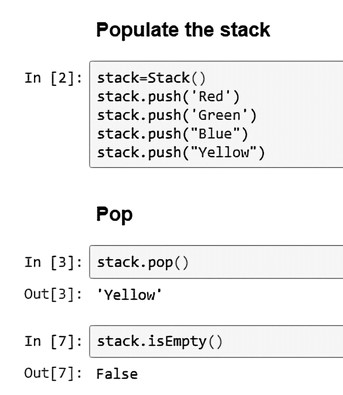


Рис. 2.5

Этот код создает стек с четырьмя элементами данных.

Временная сложность стеков

Рассмотрим временную сложность стеков, используя «O­большое» (табл. 2.6).

Таблица 2.6

|  |  |
| --- | --- |
| Операция | Временная сложность |
| push | *O*(1) |
| pop | *O*(1) |
| size | *O*(1) |
| peek | *O*(1) |

Видим, что производительность этих четырех операций не зависит от размера стека.

Практический пример

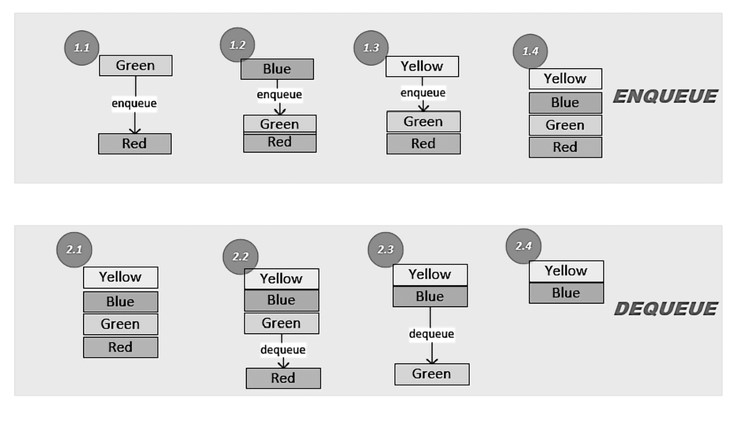
Стек часто применяется на практике в качестве структуры данных. Например, он используется для хранения истории веб­браузера. Другой пример — выполнение операции Undo при работе с текстом.

Очередь

Как и стек, *очередь* хранит *n* элементов в одномерной структуре. Элементы добавляются и удаляются по принципу FIFO (First­In, First­Out: *«первым пришел* — *первым ушел»*). Каждая очередь имеет *начало* и *конец*. Когда элементы удаляются из начала, операция называется *удалением из очереди* —dequeue. Когда элементы добавляются в конец, операция называется *постановкой в очередь* —enqueue.

На следующей диаграмме (рис. 2.6) в верхней части показана операция enqueue(). Шаги *1.1*, *1.2* и *1.3* добавляют три элемента в очередь; итоговая очередь показана на шаге *1.4*. Обратите внимание, что *Yellow — в конце*, а *Red* — *в начале*.

В нижней части диаграммы представлена операция dequeue(). Шаги *2.2*, *2.3* и *2.4* удаляют элементы из начала очереди один за другим.



**Очередь**

Рис. 2.6

Эта очередь может быть реализована с помощью следующего кода:

class Queue(object): def \_\_init\_\_(self): self.items = [] def isEmpty(self): return self.items == [] def enqueue(self, item):

self.items.insert(0,item) def dequeue(self):

return self.items.pop() def size(self): return len(self.items)

С помощью скриншота на рис. 2.7 выполним постановку и удаление элементов из очереди, как это показано на диаграмме выше.

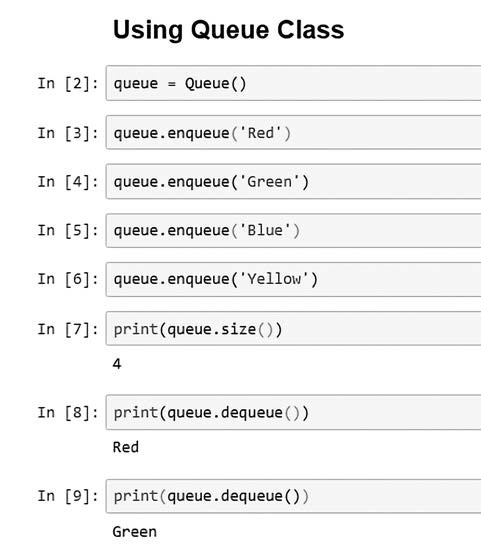


Рис. 2.7

Обратите внимание, что код сначала создает очередь, а затем помещает в нее четыре элемента.

Базовый принцип использования стеков и очередей

Рассмотрим базовый принцип использования стеков и очередей с помощью аналогии. Представьте, что мы получаем почтовую корреспонденцию и складываем ее на стол. Письма накапливаются в стопки, пока мы не находим время, чтобы открыть и просмотреть их одно за другим. Есть два способа сделать это:

* Мы складываем письма в стопку, и всякий раз, когда мы получаем новое письмо, мы кладем его наверх. Когда мы хотим прочитать письма, мы начинаем с того, которое лежит сверху. Стопка — это то, что мы называем *стеком*. Обратите внимание, что последнее поступившее письмо находится сверху и будет обработано первым. Взять письмо из верхней части стопки означает выполнить операцию *pop*. Положить новое письмо сверху — выполнить операцию *push*. Если в итоге у нас получится большая стопка‚ а письма продолжат приходить, то есть вероятность, что мы никогда не доберемся до очень важного письма в самом низу.
* Мы складываем письма в стопку, но сначала хотим открыть самое старое письмо: каждый раз, когда мы хотим просмотреть одно или несколько писем, мы начинаем с более старых. Это — *очередь*. Добавление письма в стопку — операция *enqueue* *(постановка в очередь)*. Удаление письма из стопки — операция *dequeue* *(удаление из очереди)*.

Дерево

*Дерево* — иерархическая структура данных, что делает ее особенно полезной при разработке алгоритмов. Мы используем деревья везде, где требуются иерархические отношения между элементами данных.

Давайте подробнее рассмотрим эту интересную и важную структуру.

Каждое дерево имеет конечный набор узлов, так что в нем есть начальный элемент данных, называемый *корнем* (root), и набор узлов, соединенных между собой *ветвями* (branches).

Терминология

Рассмотрим некоторые термины, связанные с древовидной структурой данных (табл. 2.7).

Таблица 2.7

|  |  |
| --- | --- |
| Корневой узел (Root node) | Узел без родителя называется корневым узлом. Например, на следующей диаграмме (рис. 2.8) корневым узлом служит *A*. В алгоритмах корневой узел, как правило, содержит наиболее важное значение во всей древовидной структуре |
| Уровень узла  (Level of a node) | Расстояние от корневого узла называется уровнем узла.  На следующей диаграмме уровень узлов *D*, *E* и *F* равен двум |
| Узлы­братья (Siblings nodes) | Два узла в дереве называются *братьями*, если они расположены на одном уровне. Например, если мы взглянем на диаграмму‚ то увидим‚ что узлы *B* и *C* являются братьями |
| Дочерний и родительский узлы (Child and parent node) | Узел *F* является дочерним по отношению к узлу *C*, если они напрямую связаны и уровень узла *C* меньше уровня узла *F*. И наоборот, узел *C* является родительским для узла *F*. Узлы *C* и *F* на следующей диаграмме демонстрируют отношения родительского и дочернего узлов |
| Степень узла (Degree of a node) | Степень узла — это количество его дочерних элементов. Например, на рис. 2.8 узел *B* имеет степень 2 |
| Степень дерева (Degree of a tree) | Степень дерева равна максимальной степени составляющих его узлов. Дерево, представленное на следующей диаграмме, имеет степень 2 |
| Поддерево (Subtree) | Поддерево — это часть дерева с выбранным узлом в качестве корневого‚ а все его дочерние элементы — это узлы дерева. На диаграмме поддерево от узла *E*  состоит из узла *E* в качестве корневого и узлов *G* и *H* в качестве дочерних |
| Концевой узел (Leaf node) | Узел в дереве без дочерних элементов называется *концевым*.  Например, на рис. 2.8 *D*, *G*, *H* и *F* — это четыре концевых узла |
| Внутренний  узел (Internal node) | Любой узел, который не является ни корневым, ни концевым‚ называется внутренним. У внутреннего узла имеются по крайней мере один родительский и один дочерний узлы |

Обратите внимание, что деревья — это своего рода сети или графы, которые мы будем изучать в главе 5. При анализе графов и сетей вместо ветвей мы используем термин «ребро». Большая часть остальной терминологии остается неизменной.

Типы деревьев

Существует несколько типов деревьев:

* *Двоичное дерево* (binary tree). Если степень дерева равна двум, оно называется *двоичным*. Например, дерево, показанное на следующей диаграмме, является двоичным, поскольку имеет степень 2 (рис. 2.8).

А

B

C

D

G

*Корневой узел*

*Внутренний узел*

*Внутренний узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Внутренний узел*

Уровень 0

Уровень 1

Уровень 2

Уровень 3

H

E

F

Рис. 2.8

Обратите внимание, что дерево на рис. 2.8 имеет четыре уровня и восемь узлов.

* *Полное дерево* (full tree). Это дерево, в котором все узлы имеют одинаковую степень, которая равна степени дерева. На диаграмме ниже представлены упомянутые типы деревьев (рис. 2.9).

Обратите внимание, что двоичное дерево слева не является полным, так как узел *C* имеет степень 1, а все остальные узлы — степень 2. Деревья в центре и справа являются полными.

* *Идеальное дерево* (perfect tree). Это особый тип полного дерева, у которого все конечные узлы расположены на одном уровне. На рис. 2.9 двоичное дерево справа является идеальным полным деревом, поскольку все его конечные узлы находятся *на уровне 2*.

Резюме

* *Упорядоченное дерево* (ordered tree). Если дочерние элементы узла организованы в определенном порядке согласно установленным критериям, дерево называется *упорядоченным*. Дерево, например, может быть упорядочено слева направо в порядке возрастания. Таким образом, значение узлов одного уровня будет увеличиваться при движении слева направо.

А

B

C

D

G

*Корневой узел*

*Внутренний узел*

*Внутренний узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Концевой узел*

*Внутренний узел*

H

E

F

Неполное, неидеальное дерево

Полное, неидеальное дерево

Полное, идеальное дерево

А

B

C

D

*Корневой узел*

*Внутренний узел*

*Концевой*

*узел*

*Концевой*

*узел*

*Внутренний*

*узел*

E

I

F

*Концевой*

*узел*

*Внутренний узел*

А

B

C

D

G

*Корневой узел*

*Внутренний узел*

*Концевой узел*

*Концевой*

*узел*

*Концевой узел*

*Концевой узел*

*Внутренний*

*узел*

H

E

I

F

*Концевой*

*узел*

*Внутренний узел*

Рис. 2.9

Практические примеры

Дерево — одна из основных структур данных, используемых при разработке деревьев решений. Их мы обсудим в главе 7. Благодаря своей иерархической структуре деревья используются в алгоритмах сетевого анализа (см. главу 5), а также в алгоритмах поиска и сортировки, где применяются стратегии типа «разделяй и властвуй».

РЕЗЮМЕ

В этой главе мы обсудили структуры данных, используемые для реализации различных типов алгоритмов. Теперь вы сможете подобрать подходящую структуру данных для вашего алгоритма. Помните, что выбор той или иной структуры влияет на производительность.

Следующая глава посвящена алгоритмам сортировки и поиска. При работе с ними мы будем использовать изученные ранее структуры данных.

1. Или ADT (Abstract Data Type). — *Примеч. ред*. [↑](#footnote-ref-1)