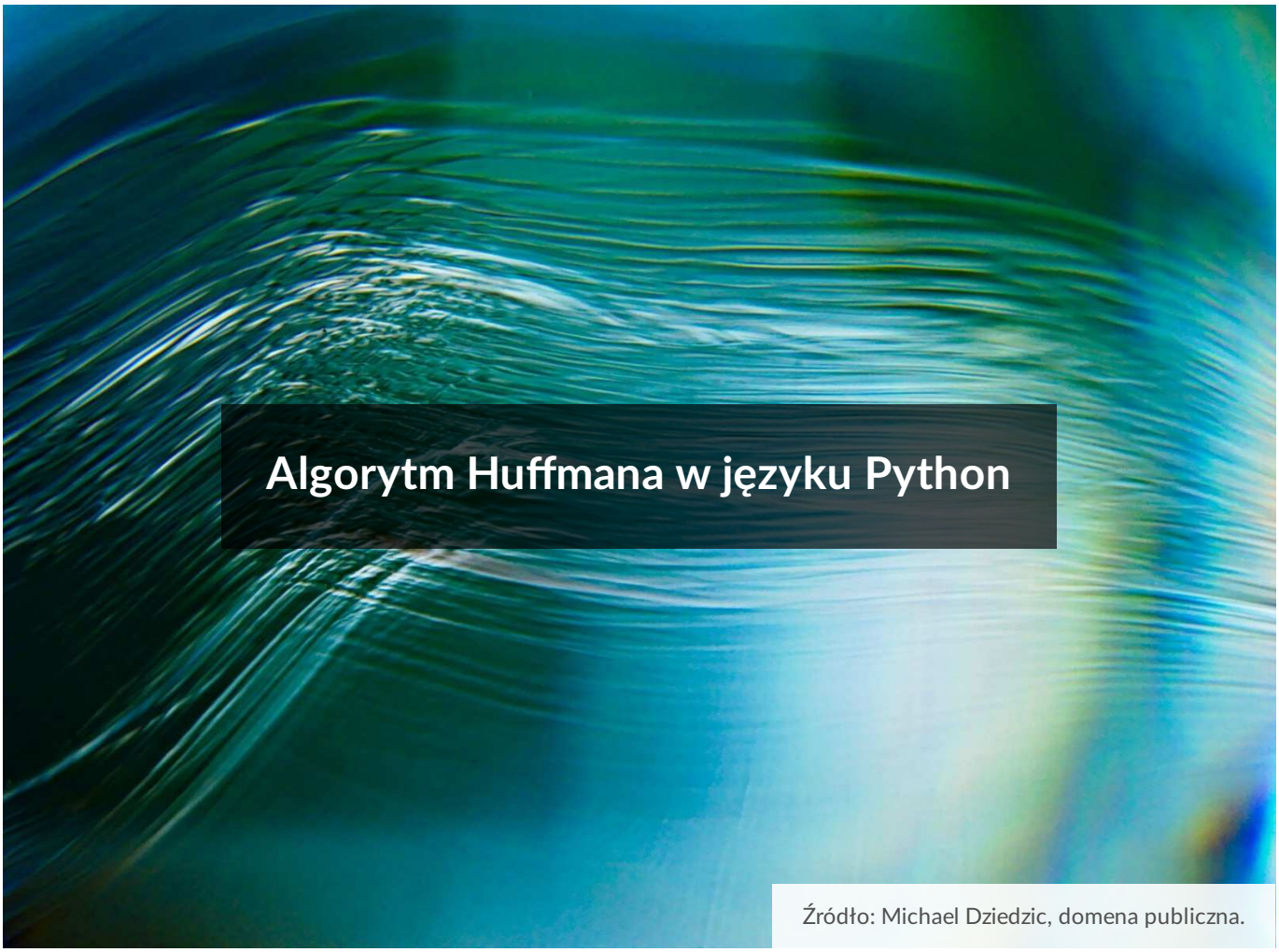
Algorytm Huffmana w języku Python



Źródło: Michael Dziedzic, domena publiczna.

Poznaliśmy już podstawowe informacje dotyczące algorytmu Huffmana. W tym e-materiale zaimplementujemy go w języku Python. Dla większej czytelności i uproszczenia implementacji użyjemy podstaw programowania obiektowego, o których możesz przeczytać w e-materiale Wstęp do programowania obiektowego w języku Python

Jeśli chcesz zapoznać się z przykładowym rozwiązaniem konkretnego problemu i sprawdzić swoją wiedzę, przejdź do e-materiału Algorytm Huffmana – zadania maturalne

Ciekawi cię, jak wygląda implementacja algorytmu Huffmana w innych językach programowania? Możesz się z nimi zapoznać w pozostałych e-materiałach z tej serii:

- Algorytm Huffmana w języku C++,
- Algorytm Huffmana w języku Java.

> **Twoje cele**
>
> - Prześledzisz implementację algorytmu Huffmana w języku Python.
> - Przygotujesz program w języku Python kodujący ciąg znaków za pomocą algorytmu Huffmana.
> - Wykonasz ćwiczenia, w których wykorzystasz znajomość algorytmu Huffmana.
> - Wyjaśnisz, jak konstruować i przeszukiwać drzewa binarne.

Przeczytaj

> **Już wiesz**
>
> W roku 1952 David Huffman opracował prefiksową metodę kompresji bezstratnej.
>
> Mówimy, że kodowanie jest prefiksowe, gdy żadne słowo kodowe nie jest początkiem innego słowa.
>
> Słowem kodowym w kontekście kodowania Huffmana określamy ciągi bitów, które jednoznacznie interpretują jakiś znak.

Problem 1

Zbadajmy przykład zastosowania kodów o stałej długości, kodując pojedynczą literę za pomocą ciągu binarnego o długości 2. W podobny sposób skonstruowany jest system ASCII.

```
# przygotujemy funkcje do kodowania i odkodowywania
def koduj(tekst: str, slownik: dict):
    wynik = ""
    for znak in tekst:
        wynik += slownik[znak]
    return wynik

def dekoduj(tekst: str, slownik: dict):
    wynik = ""

    # wycinamy z kodu po 2 znaki
    elementy = [tekst[i:i+2] for i in range(0,len(tekst),2)]
    for element in elementy:
        wynik += slownik[element]
    return wynik

# wykonamy prosty przykład
print(koduj("AABCBBACC", {"A": "00", "B": "01", "C": "10"}))
# 000001100101001010

print(dekoduj("000001100101001010", {"00": "A", "01": "B", "10"
# AABCBBACC
```

Spróbujmy teraz zastosować kody o zmiennej długości.

```
# przyjmijmy kodowanie: A = 0, B = 01, C = 110
# przykład zakodowanego ciągu cyfr
# 01010
# jaki jest pierwszy znak: A, czy B ?
```

Ważne!

W przypadku zastosowania kodów o zmiennej długości żaden kod nie może być prefiksem innego kodu. Ten warunek zapewnia, że tekst będzie dało się odkodować

jednoznacznie. W powyższym przykładzie znak A jest prefiksem B, dlatego kodowanie jest błędne.

Ciekawostka

Binarne drzewo Huffmana ma pewną właściwość – jest ono regularne, co oznacza, że każdy węzeł ma zero lub dwóch potomków.

Przykład 1

Funkcja, którą napisaliśmy powyżej, równie dobrze nada się do kodowania tekstu o dowolnej długości słów kodowych.

```
def koduj(tekst: str, slownik: dict):
    wynik = ""
    for lit in tekst:
        wynik += slownik[lit]
    return wynik

# wykonamy kodowanie
print(koduj("BACA", {"A": "111", "B": "00", "C": "10"}))
# 0011110111
```

Zdekodowanie takiej wiadomości nie będzie już tak proste. Wynika to z faktu, że nie możemy podzielić wiadomości na określoną liczbę części, a następnie odkodować każdej z nich po kolei.

Problem 2

Aby dobrze zaprojektować drzewo, wykorzystamy tabelę częstości pojawiania się znaków w ciągu do zakodowania. Zdefiniujemy klasę zawierającą odpowiednie metody oraz funkcję tworzącą drzewo.

Musimy pamiętać o posortowaniu elementów tabeli. Dzięki dynamicznemu obliczaniu częstości występowania znaków otrzymamy wydajny algorytm. Stosując go, za każdym razem uzyskamy optymalny wynik: najkrótszy kod będzie odpowiadał najczęściej pojawiającemu się znakowi (zaś kod najdłuższy – najrzadziej występującej literze). Np. tworząc drzewo dla wierzchołków o częstości kolejno 1, 2, 3, najpierw połączymy ze sobą wierzchołki 1 i 2. Kolejnym krokiem będzie połączenie nowo utworzonego węzła z wierzchołkiem o częstości 3. Węzły, które łączymy najpierw, będą ostatecznie znajdowały się głębiej w drzewie, co oznacza, że trzeba będzie przejść po większej liczbie gałęzi, aby do nich dotrzeć. Podczas kodowania każda z gałęzi odpowiada jednemu znakowi. Oznacza to, że im płycej w drzewie położony jest dany wierzchołek, tym mniejsza będzie długość jego kodu.

Wymaga to zapamiętywania tablicy kodów znaków.

Poniżej przykład implementacji i wywołania funkcji, tworzącej posortowaną tabelę częstości dla zadanego parametru `napis`. Funkcja dla każdego znaku występującego w ciągu znaków `napis`, zlicza liczbę jego wystąpień, a następnie zebrane dane sortuje nierosnąco po liczbie wystąpień.

```
def tworz_tabele_czestosci(napis: str):
    wynik = {}
    for litera in napis:
        if litera in wynik:
            wynik[litera] += 1
        else:
            wynik[litera] = 1
    # po zliczeniu liczby wystąpień danej litery sortujemy w ko
    wynik = sorted(wynik.items(), key=lambda x: x[1], reverse=T
    return wynik

# przykład wywołania
napis = "ABAABBBCCDEEEEAAA"
print("Częstości liter dla ciągu znaków: ", napis)
tablica = tworz_tabele_czestosci(napis)
print("Tablica: ", tablica)
```

```
17
18 # Częstości liter dla ciągu znaków: ABAABBBCCDEEEEAAA.
19 # Tablica: [('A', 6), ('B', 4), ('E', 4), ('C', 2), ('D', 1)]
```

Bazując na przygotowanej tabeli, utworzymy klasę, dzięki której zbudujemy drzewo binarne. Zdefiniujemy też funkcję, która będzie odpowiadać za skonstruowanie słownika tablicy kodów znaków oraz jego odwróconej wersji.

```
1  # klasa opisująca węzeł drzewa
2  class Wezel_Huffmana:
3      lewy_syn = None
4      prawy_syn = None
5
6      def __init__(self, lewy_syn, prawy_syn):
7          self.lewy_syn = lewy_syn
8          self.prawy_syn = prawy_syn
9
10     def dzieci(self):
11         return (self.lewy_syn, self.prawy_syn)
12
13     def __str__(self):
14         return str(self.lewy_syn) + str(self.prawy_syn)
15
16     # funkcja generująca słownik zawierający ciąg binarny dla j
17     def drzewo_huffmana(wezel, lewy_syn = True, kod = ''):
18         if type(wezel) is str:
19             return {wezel: kod}
20         (l, r) = wezel.dzieci()
21         d = dict()
22         d.update(drzewo_huffmana(l, True, kod + '0'))
23         d.update(drzewo_huffmana(r, False, kod + '1'))
24         return d
25
26     # funkcja tworząca słownik z wszystkimi znakami użytymi w n
27     def tworz_slownik_huffmana(tabela_czestosci: list):
28         while len(tabela_czestosci) > 1:
29             (wezel_prawy_syn, wartosc_czest_1) = tabela_czestos
30             (wezel_lewy_syn, wartosc_czest_2) = tabela_czestosc
31
32             # za pomocą mechanizmu wycinków odcinamy ostatnie d
33             tabela_czestosci = tabela_czestosci[:-2]
```

```
34
35              # tworzymy obiekty klasy
36              nowy_wezel = Wezel_Huffmana(wezel_lewy_syn, wezel_p
37
38              # dodajemy do listy krotkę zawierającą węzeł i sumę
39              krotka = (nowy_wezel, wartosc_czest_1 + wartosc_cze
40              tabela_czestosci.append(krotka)
41
42              # sortujemy nierosnąco listę według drugiego elemen
43              tabela_czestosci = sorted(tabela_czestosci, key=lam
44
45      # wywołujemy rekurencyjną funkcję, która zwróci słownik prz
46      # z przejścia pomiędzy gałęziami drzewa, rozpoczynając od l
47      slownik_huffmana = drzewo_huffmana(tabela_czestosci[0][0])
48
49      #gdybyśmy chcieli zakodować wiadomość złożoną z tylko jedne
50      if type(tabela_czestosci[0][0]) == str:
51          slownik_huffmana = {tabela_czestosci[0][0]: 1}
52
53      #tworzymy odwrotny słownik
54      slownik_odwrotny = {wartosc: klucz for klucz, wartosc in sl
55
56      return (slownik_huffmana, slownik_odwrotny)
57
58 # przykład wywołania bazujący na napisie
59 tablica = [('A', 6), ('B', 4), ('E', 4), ('C', 2), ('D', 1)]
60 slownik, odwrotny = tworz_slownik_huffmana(tablica)
61 print("Słownik:",slownik)
62 print("Odwrotny: ",odwrotny)
63
64 # Słownik: {'A': '00', 'B': '01', 'E': '10', 'C': '110', 'D': '
65 # Odwrotny: {'00': 'A', '01': 'B', '10': 'E', '110': 'C', '111'
```

Teraz możemy już zdefiniować funkcje, które używając słownika, zakodują i odkodują ciąg znaków.

```
1 # funkcja kodująca
2 def koduj_huffmana(tekst: str, slownik: dict):
3     wynik = ""
4     for lit in tekst:
5         wynik += slownik[lit]
```

```
    return wynik

# funkcja odkodowująca
def odkoduj_huffmana(zakodowany: str, slownik: dict):
    wynik = ""
    kod = ""
    for znak in zakodowany:
        kod += znak
        zakodowany = zakodowany[1:] # reszta
        if kod in slownik:
            wynik += slownik[kod]
            kod = ""
    return wynik

# przykładowe wykonanie
napis = "ABAABBBCCDEEEEAAA"
slownik = {'A': '00', 'B': '01', 'E': '10', 'C': '110', 'D': '1
odwrotny = {'00': 'A', '01': 'B', '10': 'E', '110': 'C', '111':
zakodowany = koduj_huffmana(napis, slownik)
print("Napis zakodowany ", zakodowany)
odkodowany = odkoduj_huffmana(zakodowany, odwrotny)
print("Napis odkodowany ", odkodowany)

# Napis zakodowany 000100000101011101101111010101010000000
# Napis odkodowany ABAABBBCCDEEEEAAA
```

Ciekawostka

W tej sekcji w programie wykorzystaliśmy słowo kluczowe lambda. W języku Python pozwala ono na deklarowanie funkcji **anonimowych**.

Wyrażenia lambda używamy jako argumentu funkcji wyższego rzędu (czyli funkcji, która przyjmuje inne funkcje jako argumenty).

Te funkcje w programach często stosuje się przy okazji sortowania. Stanowią one wyrażenia zwracające klucz (wartość), według którego mają zostać posortowane elementy.

Dana jest następująca lista:

```
szczyty = [
    ('Mount Everest',8848,1953),
    ('Lhotse',8516,1956),
```

```
    ('Kanczendzonga',8586,1955),
    ('Makalu',8481,1955),
    ('Czogori',8611,1954)
]
```

W jaki sposób możemy ją wykorzystać?

- `sorted(szczyty)` – posortuje listę według pierwszego elementu każdej tupli, czyli nazwy szczytu;
- `sorted(szczyty, key=lambda x: x[1])` – posortuje listę według klucza zwróconego przez jednoargumentową funkcję lambda, w tym wypadku będzie to wysokość szczytu (nazwa szczytu znajduje się pod indeksem 0, a rok zdobycia szczytu pod indeksem 2);
- `max(szczyty, key=lambda x: x[1])` – zwróci najwyższy szczyt;
- `min(szczyty, key=lambda x: x[2])` – zwróci szczyt najwcześniej zdobyty.

Przyjrzyjmy się przykładowemu programowi.

```
szczyty = [
    ('Mount Everest',8848,1953),
    ('Lhotse',8516,1956),
    ('Kanczendzonga',8586,1955),
    ('Makalu',8481,1955),
    ('Czogori',8611,1954)
]

print(min(szczyty, key=lambda x: x[2]))
```

Dana jest lista `szczyty`, która składa się z pięciu krotek. Każda krotka zawiera dane o nazwie szczytu, jego wysokości oraz roku, w którym szczyt został zdobyty.

Chcemy wyświetlić krotkę przechowującą informacje o najwcześniej zdobytym szczycie, dlatego skorzystamy z polecenia `print`.

```
print()
```

Wykorzystamy wbudowaną w język Python funkcję `min()`.

```
print(min())
```

Przeszukujemy listę `szczyty`.

```
print(min(szczyty))
```

Opcjonalny argument `key` opisuje, jak porównać elementy, aby znaleźć minimum.

```
print(min(szczyty, key))
```

Wykorzystamy wyrażenie lambda.

```
print(min(szczyty, key=lambda))
```

Do wyrażenia lambda przekażemy poszczególne trójelementowe krotki reprezentujące informacje na temat szczytów.

Do zmiennej `x` argumentu wyrażenia lambda podstawiana jest krotka, a następnie wyznaczana jest wartość `x[2]`, która z trójelementowej krotki pobiera rok zdobycia szczytu.

```
szczyty = [
    ('Mount Everest',8848,1953),
    ('Lhotse',8516,1956),
    ('Kanczendzonga',8586,1955),
    ('Makalu',8481,1955),
    ('Czogori',8611,1954)
]

print(min(szczyty, key=lambda x: x[2]))
```

Wynik działania programu:

```
('Mount Everest', 8848, 1953)
```

Słownik

ASCII

(ang. *American Standard Code for Information Interchange*) pierwotnie siedmiobitowy system kodowania znaków (współcześnie rozszerzony do ośmiu bitów); w oryginalnej wersji kodom z zakresu 0–127 przyporządkowano 26 liter łacińskich, 10 cyfr (0–9) oraz dodatkowe znaki

częstość występowania znaków alfabetu

specyficzna cecha wszystkich języków naturalnych; w przypadku każdego z nich można określić, jak często w słowniku pojawiają się poszczególne litery alfabetu; dla języka polskiego takie częstości zbadał i przedstawił m.in. A. Przepiórkowski z Instytutu Podstaw Informatyki Polskiej Akademii Nauk

kompresja bezstratna

metoda kompresji zapewniająca możliwość odtworzenia oryginalnej (niezmienionej) postaci informacji pierwotnej z danych skompresowanych

słowo kodowe

słowo, wyrażenie, znak lub ciąg bitów odpowiadające jakiemuś słowu, które chcemy zakodować; we wczesnych etapach rozwoju kryptografii słowa kodowe były układane w ramach książek kodowych i wykorzystywane do szyfrowania wiadomości

Film samouczek

Kodowanie Huffmana – drzewo prawdopodobieństw

Polecenie 1

Napisz program, który dla zadanego łańcucha znaków `napis` wygeneruje drzewo Huffmana. Działanie programu przetestuj dla następującego łańcucha znaków „`aaaabbccaaaaccbbaddaaaaaaabbbbcccaaaaaabbbb"`.

Specyfikacja problemu:

Dane:

- `napis` – łańcuch znaków, dla którego należy wygenerować drzewo Huffmana

Wynik:

Program wyświetla częstości znaków w łańcuchu znaków `napis`, a także przypisany im kod.

Przykładowe wyjście:

```
Znak: a czestosc: 22 kod: 0
Znak: b czestosc: 12 kod: 10
Znak: c czestosc: 7 kod: 110
Znak: d czestosc: 2 kod: 111
```

Twoje zadania

1. Program wyświetla drzewo Huffmana dla zadanego łańcucha znaków.

```
1
```

```
1
```

Polecenie 2

Porównaj swoje rozwiązanie z zaprezentowanym w filmie.

Uwaga!

Program zaprezentowany w filmie prezentuje algorytm generowania drzewa Huffmana zrealizowany za pomocą programowania obiektowego. W odróżnieniu od kodu zaprezentowanego w poprzedniej sekcji używane są dodatkowe struktury danych takie jak kolejka priorytetowa (zaimplementowana za pomocą listy) oraz wcześniej wspomniane drzewo. Więcej na temat zasady tworzenia i działania drzew dowiesz się z materiału Binarne drzewo poszukiwań.

Film dostępny pod adresem /preview/resource/R1B9SSvPUqfwZ

Lekcja poświęcona programowaniu algorytmu Huffmana w języku Python.

Sprawdź się

Pokaż ćwiczenia:

Ćwiczenie 1

W języku Python 3 dostępny jest standardowy moduł `queue`. Zapoznaj się z pokazanym kodem, a następnie wykonaj umieszczone pod nim polecenia.

```
import queue

class Wezel:
    def __init__(self, lewy=None, prawy=None):
        self.lewy = lewy
        self.prawy = prawy

def stworz(elementy):
    kolejka = queue.Queue()
    for element in elementy:
        wezel = Wezel(element[0], element[1])
        kolejka.put(wezel)
    return kolejka

elementy = [(65, 'A'), (66, 'B'), (67, 'C'), (68, 'D'), (69, 'E'), (70, 'F')]

p = stworz(elementy)

#wywołania wraz z oczekiwanymi wynikami
print(p.get().dzieci())
# (65, 'A')
print(p.get().dzieci())
# (66, 'B')
print(p.get().dzieci())
# (67, 'C')
```

Aby powyższy kod działał prawidłowo, konieczne jest zdefiniowanie odpowiedniej metody w klasie `Wezel`. Wskaż prawidłowy zapis metody.

```
class Wezel:
    def __init__(self, lewy=None, prawy=None):
        self.lewy= lewy
        self.prawy= prawy

    # definicja metody
```

- ◯
```
# definicja metody
def dzieci(self)
    return (self.lewy, self.prawy)
```

- ◯
```
# definicja metody
def dzieci(self):
    return (self.lewy, self.prawy)
```

- ◯
```
# definicja metody
def dzieci():
    return (lewy, prawy)
```

Zaznacz zdania prawdziwe.

- ☐ Metoda put() klasy Queue umieszcza element na początku kolejki.
- ☐ Metoda get() klasy Queue zwraca element kolejki i nie usuwa go.
- ☐ Metoda get() klasy Queue zwraca element kolejki i usuwa go.

- [ ] Metoda put() klasy Queue umieszcza element na końcu kolejki.

Ćwiczenie 2

Zdefiniuj klasę `class Wezel` o następujących polach i metodach:

Pola:

- `dane` – o domyślnej wartości `None`
- `lewo` – o domyślnej wartości `None`
- `prawo` – o domyślnej wartości `None`

Metody:

- konstruktor – przypisuje każdemu polu odpowiednie wartości
- dunder `__str__` – nadpisuje domyślną metodę odpowiedzialną za zwracanie łańcucha znaków odpowiadającego danemu obiektowi; deklarujemy ją w ten sposób: `def __str__(self)`; zaimplementowana metoda powinna zwracać łańcuch znaków `f"Wartość = {self.dane}"`
- `dzieci` – zwraca łańcuch znaków: `f"Dzieci: L -> {self.lewo} P -> {self.prawo}"`

Następnie dla każdej krotki zawartej w liście `wezly` stwórz obiekty klasy `Wezel` z zawartością pola `dane` zgodną z pierwszą wartością krotki. Wartościami pól `lewo` i `prawo` powinny być nowo utworzone obiekty, których indeks w tablicy `wezly` odpowiada kolejno drugiej i trzeciej wartości z krotki (`None` oznacza brak dziecka, a więc wartością pola `lewo` bądź `prawo` będzie `None`).

> **Przykład 1**
>
> Dla listy `wezly = [("korzen", None, 1), ("prawo", None, None)]` musimy utworzyć węzeł z polami `dane` o wartości `"korzen"`, `lewo` zawierające wartość `None` i `prawo` zawierające obiekt klasy `Wezel`, którego `dane` to `"prawo"`, `lewo` to `None` i `prawo` to `None`

Specyfikacja problemu:

Dane:

- `wezly` – lista zawierająca krotki z wartościami pól instancji klasy

Wynik:

Na standardowym wyjściu program wypisuje dane dla zadanych węzłów, a następnie ich dzieci.

Swoje rozwiązanie przetestuj dla listy: `wezly = [("korzen", 1, 2), ("lewo", None, None), ("prawo", None, None)]`.

Twoje zadania

1. Program wyświetla dane wszystkich podanych węzłów, a następnie dane ich dzieci.

```
class Wezel:
    # tutaj wpisz własny kod
    pass

wezly = [("korzen", 1, 2), ("lewo", None, None), ("prawo", None, None)]

utworzone_wezly = []
```

```

# tworzenie węzłów
for wezel in wezly:
    utworzone_wezly.append(Wezel(wezel[0]))

for i in range(len(wezly)):
    if wezly[i][1] is None:
```

```

```

Ćwiczenie 3

Korzystając z klasy zaimplementowanej w zadaniu 2, utwórz węzły w ten sam sposób, a następnie zapisz funkcję `wedrownik(element)` przechodzącą po drzewie binarnym. Powinna ona działać rekurencyjnie według następującego algorytmu:

- jeśli wywoływany element jest wartością `None`, funkcja kończy działanie;
- w pozostałych przypadkach funkcja wyświetla wartość zawartą w węźle, a następnie rekurencyjnie wywołuje się dla elementów najpierw lewego, a potem prawego.

Specyfikacja problemu:

Dane:

- `wezly` – lista zawierająca krotki z wartościami pól instancji klasy

Wynik:

Na standardowym wyjściu program wypisuje dane wszystkich węzłów, po których przeszedł za pomocą funkcji `wedrownik()`.

Swoje rozwiązanie przetestuj dla listy:
`wezly = [("Główny", 1, 2), ("Posredni", 3, None ), ("Prawy", None, None), ("Lewy", 4, None), ("Ostatı`
.

Twoje zadania

1. Program wypisuje dane wszystkich węzłów, po których przeszedł za pomocą funkcji `wedrownik()`.

```
class Wezel:
    pass

def wedrownik(wezel):
    pass

wezly = [("Główny", 1, 2), ("Posredni", 3, None ), ("Prawy", None, None), ("Lewy", 4, None),
("Ostatni", None, None)]

utworzone_wezly = []

# tworzenie węzłów
for wezel in wezly:
    utworzone_wezly.append(Wezel(wezel[0]))
```

```
1
```

Dla nauczyciela

Autor: Adam Jurkiewicz
Przedmiot: Informatyka

Temat: Algorytm Huffmana w języku Python

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres podstawowy i rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

III. Posługiwanie się komputerem, urządzeniami cyfrowymi i sieciami komputerowymi, w tym: znajomość zasad działania urządzeń cyfrowych i sieci komputerowych oraz wykonywania obliczeń i programów.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) do realizacji rozwiązania problemu dobiera odpowiednią metodę lub technikę algorytmiczną i struktury danych;

3) objaśnia dobrany algorytm, uzasadnia poprawność rozwiązania na wybranych przykładach danych i ocenia jego efektywność;

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres podstawowy. Uczeń:

1) projektuje i programuje rozwiązania problemów z różnych dziedzin, stosuje przy tym: instrukcje wejścia/wyjścia, wyrażenia arytmetyczne i logiczne, instrukcje warunkowe, instrukcje iteracyjne, funkcje z parametrami i bez parametrów, testuje poprawność programów dla różnych danych; w szczególności programuje algorytmy z punktu I.2);

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

3) sprawnie posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

III. Posługiwanie się komputerem, urządzeniami cyfrowymi i sieciami komputerowymi.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) dokonuje kompresji informacji, objaśnia różnice między kompresją stratną i bezstratną tekstów, obrazów, dźwięków, filmów;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Prześledzisz implementację algorytmu Huffmana w języku Python.
- Przygotujesz program w języku Python kodujący ciąg znaków za pomocą algorytmu Huffmana.
- Wykonasz ćwiczenia, w których wykorzystasz znajomość algorytmu Huffmana.
- Wyjaśnisz, jak konstruować i przeszukiwać drzewa binarne.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka Python 3 (lub nowszej wersji), w tym PyCharm lub IDLE.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Algorytm Huffmana w języku Python". Uczniowie mają zapoznać się z treściami w sekcji „Przeczytaj".

Faza wstępna:

1. Nauczyciel wyświetla temat i cele zajęć zawarte w sekcji „Wprowadzenie". Prosi uczniów, by na podstawie wiadomości zdobytych przed lekcją zaproponowali kryteria sukcesu.
2. **Rozpoznanie wiedzy uczniów.** Nauczyciel prosi wybranego ucznia lub uczniów o przedstawienie sytuacji problemowej związanej z tematem lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Nauczyciel ocenia, na podstawie informacji na platformie, stan przygotowania uczniów do zajęć. Jeżeli jest ono niewystarczające, prosi wybraną osobę o przedstawienie najważniejszych informacji z sekcji „Przeczytaj". Na forum klasy uczniowie analizują problemy 1-3. Następnie pracując indywidualnie, uczniowie powtarzają zaprezentowaną implementację algorytmu na swoich komputerach.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Film samouczek". Uczniowie w parach rozwiązują problem 1. Następnie porównują swoje rozwiązanie z zaprezentowanym w filmie. Zapisują ewentualne problemy i pytania. Po czym następuje dyskusja, w trakcie której nauczyciel wyjaśnia niezrozumiałe treści.
3. **Ćwiczenie umiejętności.** Prowadzący zapowiada uczniom, że w kolejnym kroku będą rozwiązywać ćwiczenia nr 1-2 z sekcji „Sprawdź się". Każdy z uczniów robi to samodzielnie. Po ustalonym czasie wybrani uczniowie przedstawiają rozwiązania.

Nauczyciel w razie potrzeby koryguje odpowiedzi, dopowiada istotne informacje, udziela uczniom informacji zwrotnej.

Faza podsumowująca:

1. Na koniec zajęć nauczyciel raz jeszcze wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie". W odniesieniu do ich realizacji dokonuje szczegółowej oceny rozwiązania zastosowanego przez wybranego ucznia.
2. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności, omawia ewentualne problemy podczas rozwiązania ćwiczeń z programowania w języku Python.

Praca domowa:

1. Uczniowie wykonują ćwiczenie 3 z sekcji „Sprawdź się".

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka Python 3 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania PyCharm lub IDLE.

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj" można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.