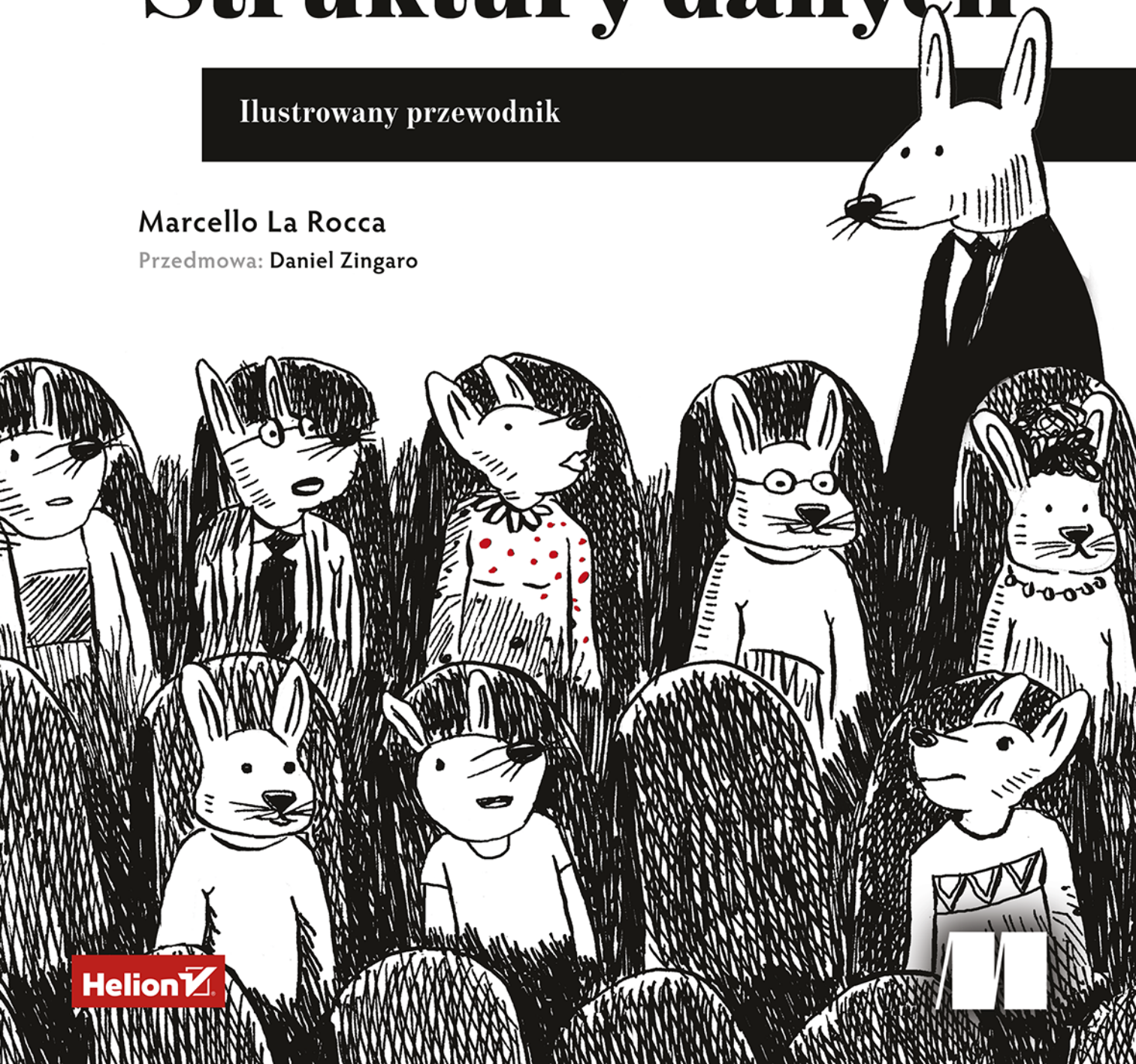
Struktury danych

Ilustrowany przewodnik

Marcello La Rocca

Przedmowa: Daniel Zingaro



Helion

Tytuł oryginału: Grokking Data Structures

Tłumaczenie: Anna Mizerska

ISBN: 978-83-289-3598-3

© Helion S.A. 2026

Authorized translation of the English edition © 2024 Manning Publications.

This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/stdail
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Tablice statyczne — budowanie Twojej pierwszej struktury danych

2

W tym rozdziale:

- Kilka podstawowych koncepcji dotyczących struktur danych
- Wprowadzenie fundamentalnej struktury danych — tablicy
- Różnica między tablicami o rozmiarze statycznym i dynamicznym
- Przedstawienie typowych operacji, które można wykonywać na tablicach
- Wykorzystanie tablic do rozwiązywania problemów

W tym rozdziale zaczniemy mówić o tym, jak działają struktury danych i jak je implementować. Rozdział ten jest wyjątkowy, ponieważ stopniowo wprowadzi Cię w proces, którego będziemy się trzymać w całej książce, omawiając prezentowane technologie. Zapozna Cię również z podstawowymi koncepcjami, które będą Ci potrzebne w dalszej części książki.

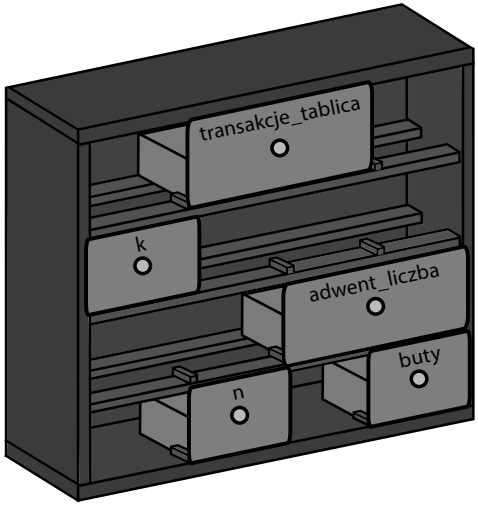
Kup książkę Poleć książkę

Czym jest tablica?

Rozpoczniemy naszą podróż po krainie struktur danych od **tablic**, a dokładniej od tablic statycznych. Tablice organizują dane, przechowując zbiór elementów i umożliwiając dostęp do nich za pomocą indeksu. Ale w tej chwili najważniejsze pytanie, na które chciałbym, żebyś potrafił odpowiedzieć, brzmi: dlaczego tablice? Pozwól, że wyjaśnię to na przykładzie.

Pamięć i szuflady

Najpierw musimy cofnąć się o krok i porozmawiać o tym, jak zorganizowana jest pamięć. Dla uproszczenia lubię myśleć o pamięci jak o modułowej półce z wysuwanymi szufladami.



Jeśli struktura półki to pamięć, szuflady są niczym zmienne — pojęcie programistyczne, które prawdopodobnie już znasz. Myśl o pamięci jak o potencjale: jeśli chcesz wykorzystać trochę pamięci, możesz tworzyć zmienne, czyli szuflady, które mogą przechowywać Twoje dane i z których możesz je później odzyskać.

Rozmiar półki określa maksymalną liczbę szuflad. Możesz tworzyć zmienne (szuflady) o różnych rozmiarach, o ile zmieszczą się w szafce. Możesz też wypełniać te szuflady danymi, a większe szuflady mogą pomieścić większe typy danych. Na przykład będziesz potrzebować większej szuflady na wartość zmiennoprzecinkową niż na znaki czy krótkie liczby całkowite.

Kiedy potrzebuję tablicy?

Poznajcie Mariusza. Uwielbia słodycze, a szczególnie kocha czekoladę. W kuchni jego rodziców znajduje się szuflada, w której Mariusz trzyma swoje czekoladowe trufle — to jego ulubione przysmaki. W tej chwili Mariusz ma jeszcze pięć trufli. Szuflada jest jak zmienna — pojemnik na dane. W tym przypadku zmienna całkowita o nazwie `szuflada` zawierałaby wartość 5.

Kup książkę Poleć książkę

Aby przejść od zmiennych całkowitych do tablic, spójrzmy na inny przykład. Zbliża się grudzień, a rodzina Mariusza przygotowuje kalendarz adwentowy dla swoich dzieci. Kalendarz ma kształt piernikowego domku z małymi szufladkami ponumerowanymi od 1 do 24.

Jeśli nie znacie kalendarza adwentowego, to jest podobny do *Advent of Code*, z tą różnicą, że zamiast zadań programistycznych każdego dnia między 1 a 24 grudnia dostajecie słodki przysmak (zabawne, jak ta analogia zwykle działa odwrotnie dla wszystkich oprócz programistów!). Każda szufladka kalendarza adwentowego zawiera ciasteczko, czekoladki lub inne słodycze, a dzieci mogą otwierać każdą szufladkę tylko w dniu odpowiadającym jej numerowi.

Wracając do naszej analogii z półką, załóżmy, że wykorzystujecie część dużej półki magazynowej na kalendarz adwentowy. Te 24 szufladki mogłyby zostać umieszczone gdziekolwiek na półce — nie muszą nawet znajdować się obok siebie i nie muszą być w żadnej konkretnej kolejności. Ale gdybyśmy mieli stworzyć te ponumerowane szufladki, chcielibyśmy je ułożyć w porządku rosnącym i obok siebie. W przeciwnym razie trudno byłoby je znaleźć.

Podobnie, gdybyśmy chcieli zamodelować kalendarz adwentowy w oprogramowaniu, moglibyśmy stworzyć 24 małe zmienne i nazwać je `adwent_szuflada_1`, `adwent_szuflada_2` i tak dalej. Nikt by nas nie powstrzymał przed takim działaniem (choć miejmy nadzieję, że *ktoś* by nas powstrzymał, zanim wprowadzimy ten bałagan na produkcję).

Tworzenie 24 różnych zmiennych ręcznie byłoby już bolesne, ale co gorsza, za każdym razem, gdy musielibyśmy uzyskać dostęp do jednej z szufladek w kodzie, musielibyśmy użyć właściwej nazwy zmiennej. Normalnie, w większości języków programowania, musielibyśmy wiedzieć, której zmiennej potrzebujemy w czasie kompilacji (czyli gdy piszemy kod).

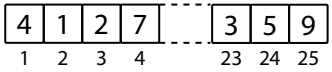
Czasami jednak otrzymujemy tę informację dopiero w czasie *wykonywania*, gdy kod jest uruchamiany. Na przykład, jeśli mamy program, który pyta użytkownika, którą szufladkę musimy sprawdzić, nie wiedzielibyśmy z góry, której zmiennej potrzebujemy, ponieważ informację tę otrzymujemy przez operacje wejścia/wyjścia podczas działania programu. A jeśli to nie Twoje pierwsze spotkanie z programowaniem, prawdopodobnie znasz pętle: czy potrafisz sobie wyobrazić, jak trudno byłoby przejść przez wszystkie szufladki bez pętli `for`? (Nie martw się, jeśli nie potrafisz, bo za chwilę zobaczysz przykład).

I tu właśnie wchodzą do gry tablice. Tablica to struktura danych, która przechowuje wiele elementów dostępnych przez indeks. Zdefiniujemy tablice dokładnie w następnym punkcie, a na razie zapamiętajcie jako ogólną zasadę, że używacie tablic, gdy musicie przechowywać, iterować po kolekcji wartości lub nimi manipulować — wartości są (mniej więcej) tego samego typu i nie wiadomo zbyt wiele o tym, jak poszczególne wartości są ze sobą powiązane. (W przypadkach, gdy macie więcej informacji o wewnętrznej strukturze danych i o tym, jak elementy są ze sobą powiązane, ta książka wprowadzi was w inne struktury danych, które będą jeszcze bardziej pomocne).

Kup książkę Poleć książkę

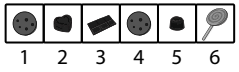
Definicje — rozmiar statyczny a dynamiczny

Czym więc jest tablica? Oto jak wyglądałby nasz kalendarz adwentowy w postaci tablicy.



Tablica z liczbami całkowitymi dla kalendarza adwentowego

Tablice nie ograniczają się tylko do przechowywania liczb całkowitych czy liczb w ogóle — mogą przechowywać ułamki, łańcuchy znaków i inne typy obiektów. Co powiesz na tablicę cukierków jako przykład?



Tablica z cukierkami

W najprostszej definicji tablica to indeksowany zbiór danych. Indeksowany oznacza, że tablica przechowuje sekwencję elementów (zwykle nazywanych **elementami**), do których można uzyskać dostęp wyłącznie za pomocą ich pozycji (znanej również jako indeks). Na przykład w kalendarzu adwentowym możemy otworzyć szufladkę o indeksie 1, aby otrzymać cukierek na 1 grudnia, ale nie możemy uzyskać dostępu do szufladek na podstawie ich zawartości — na przykład nie możemy łatwo znaleźć szufladki z siedmioma truflami ani w tablicy cukierków nie możemy po prostu powiedzieć: „Daj mi truskawkowego lizaka".

Teraz gdy zbliżamy się do formalnych definicji, musimy dokonać rozróżnienia, ponieważ na definicję tablicy możemy spojrzeć z różnych perspektyw.

Jeśli skupiamy się na funkcjonalności tablic na wysokim, częściowo abstrakcyjnym poziomie, to struktura danych tablica ma kilka kluczowych cech:

- Przechowuje zbiór danych.
- Jej elementy są dostępne za pomocą indeksu.
- Elementy nie muszą być odczytywane sekwencyjnie. Oznacza to, że jeśli potrzebuję 10. elementu tablicy, mogę uzyskać do niego dostęp bezpośrednio, bez konieczności odczytywania 9 elementów przechowywanych w tablicy przed nim.

Te kilka punktów definiuje tablicę na poziomie abstrakcyjnym. Technicznie rzecz biorąc, punkty te definiują tablicę jako **abstrakcyjny typ danych**. Warto zapamiętać to pojęcie, ponieważ spotkamy się z nim ponownie w rozdziale 7.

Patrząc z innej perspektywy, tablice stanowią jedną z podstawowych funkcjonalności wielu języków programowania. To tutaj sprawy stają się bardziej konkretne. Patrząc na tablice z tego punktu widzenia, musimy mierzyć się ze szczegółami implementacji, które różnią się w zależności od wybranego języka programowania.

Kup książkę Poleć książkę

Mimo to wiele języków programowania przestrzega kilku wspólnych właściwości podczas implementacji tablic jako podstawowej funkcjonalności języka (kontynuujemy poprzednią listę):

- Tablice są alokowane w pamięci jako pojedynczy, nieprzerwany blok pamięci z sekwencyjnymi lokalizacjami, co jest wydajne zarówno pod względem pamięci, jak i czasu.
- Tablice są ograniczone do przechowywania danych tego samego typu. To ograniczenie wynika również z potrzeby optymalizacji, ponieważ pozwala na alokację tej samej ilości pamięci dla każdego elementu tablicy oraz umożliwia kompilatorowi/interpreterowi szybkie określenie adresu pamięci każdego elementu. Omówimy to szczegółowo w następnym punkcie.
- Rozmiar tablic, czyli liczba elementów zawartych w tablicy, musi być ustalony podczas tworzenia tablicy i nie może być później zmieniony.

Ostatnie trzy punkty to część definicji **niższego poziomu**, która opisuje **tablice statyczne** (zwane również **tablicami o stałym rozmiarze**) — podstawową funkcjonalność wielu języków programowania, takich jak C, C++, Java i inne.

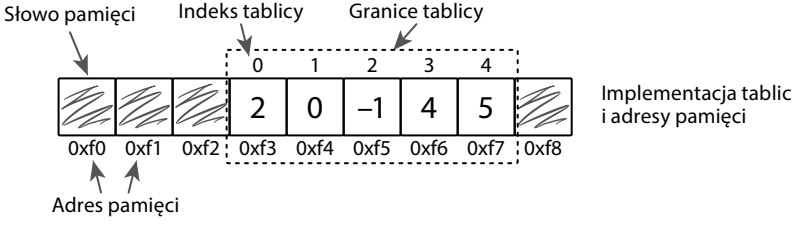
W tym rozdziale skupiamy się na tablicach statycznych. **Tablice dynamiczne** (zwane również **tablicami o zmiennym rozmiarze**), których rozmiar może się zmieniać w czasie wykonania, to inny wariant tej struktury danych. Więcej o tablicach dynamicznych dowiesz się w rozdziale 5. Warto zauważyć, że możliwe jest również złagodzenie czwartego punktu listy i umożliwienie heterogenicznej zawartości tablic, co oznacza możliwość mieszania różnych typów danych w elementach tablicy: Python, język programowania używany w tej książce, natywnie udostępnia **listy** — dynamiczny rodzaj tablicy, który pozwala na przechowywanie elementów dowolnego typu danych.

Wartości i indeksy

W poprzednim punkcie wyjaśniono, że tablice to indeksowane struktury danych. Oznacza to, że tablica przypisuje indeks każdemu z elementów, które zawiera, i tylko za pomocą indeksu możemy uzyskać dostęp do odpowiedniego elementu.

Kiedy mówiliśmy o tablicach statycznych, zwróciłem uwagę, że w wielu językach programowania tablice wymagają, aby wszystkie ich elementy były tego samego typu danych. To wymaganie jest przydatne z kilku powodów.

Po pierwsze, jak pokazuje rysunek poniżej, pozwala to na alokację dokładnie takiej ilości pamięci, jaka jest potrzebna dla tablicy. Po drugie, umożliwia szybkie obliczenie adresu pamięci dla każdego elementu, ponieważ wszystkie elementy będą miały ten sam rozmiar i będą równomiernie rozmieszczone, co sprawia, że obliczenie lokalizacji elementu w pamięci jest proste.



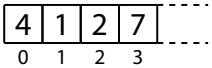
Kup książkę Poleć książkę

Mogłeś zauważyć, że w przykładach tablicy kalendarza adwentowego pokazanych w poprzednim punkcie indeksy elementów tablicy zaczynają się od 1. Innymi słowy, każdy indeks odpowiada ładnie jednemu z pierwszych 24 dni grudnia. Niektórzy z was mogą unosić brwi, ponieważ przywykliście do indeksów zaczynających się od 0, więc porozmawiajmy o tym.

Podczas gdy wiele języków programowania zaczyna indeksy od 0, niektóre mają indeksy tablic zaczynające się od 1. Kilka najbardziej znanych przykładów to Julia, MATLAB, R i Fortran.

Python jest jednym z tych języków, które używają indeksowania od zera, dlatego w całej książce stosujemy konwencję, że indeksy tablic zaczynają się od 0.

Indeksowanie od zera, jak możecie sobie wyobrazić (i być może już tego doświadczyliście), zmusza programistów do ostrożności przy myśleniu o indeksach, szczególnie gdy muszą implementować algorytmy, które uzyskują dostęp do określonych pozycji, lub gdy muszą uważać, aby nie wyjść poza granice prawidłowych indeksów. Na przykład ostatni element tablicy o rozmiarze `n` z indeksowaniem od zera będzie miał indeks `n-1`, a próba dostępu do elementu o indeksie `n` spowoduje błąd.



Wersja kalendarza adwentowego z indeksowaniem od zera

Inicjalizacja

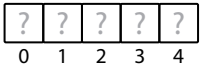
Jak wspomniałem wcześniej, reszta tego rozdziału skupia się na tablicach statycznych. Jedną z kluczowych kwestii, którą krótko poruszyłem, jest to, że tworząc tablicę statyczną, musisz z góry określić jej rozmiar. Na przykład, jeśli potrzebujesz przechować pięć elementów w tablicy, musisz zarezerwować pamięć na wszystkie te elementy już w momencie tworzenia tablicy. Oznacza to, że deklarując tablicę, tworzymy strukturę, która będzie przechowywać pięć wartości określonego typu, który również musi być określony w momencie deklaracji.

Przygotowujemy miejsce na te elementy, ale co dzieje się, zanim faktycznie przypiszemy im wartości?

Na początek istnieją dwa sposoby utworzenia tablicy: możemy ją po prostu zadeklarować lub (w większości języków programowania) możemy **zainicjalizować** elementy tablicy jednocześnie z jej deklaracją.

Inicjalizacja tablicy oznacza przypisanie (odpowiednie) wartości wszystkim jej elementom. W takim przypadku kompilator, tłumacząc kod na program, który może działać na komputerze, jednocześnie alokuje pamięć dla tablicy i wypełnia ją wartościami, które określamy w czasie kompilacji, przed przejściem do następnej instrukcji.

Co się dzieje, gdy po prostu deklarujemy tablicę bez jej inicjalizacji? Czy jej elementy pozostają „puste”?



„Pusta” tablica. Jakie wartości znajdziemy? Nie wiemy!

Nie istnieje pojęcie pustości, co oznacza, że gdy deklarujesz zmienną, kompilator musi przypisać jej jakąś wartość. W przypadku tablic wszystkie elementy muszą mieć przypisaną wartość.

Dana wartość zależy od języka programowania i typu tablicy. Na przykład w Javie tablica liczb całkowitych będzie miała wszystkie elementy ustawione na 0, jeśli zostanie utworzona bez inicjalizacji. Niektóre języki programowania mają specjalną wartość reprezentującą pustość, na przykład Python

Kup książkę Poleć książkę

ma wartość `None`, a Java używa `null`. Należy zauważyć, że są to specjalne wartości, które są jawnie przypisywane do elementów tablicy.

Pamiętaj, że musisz być ostrożny przy tworzeniu tablicy, jeśli planujesz dostęp do jej elementów bez wcześniejszego przypisania im wartości. W razie wątpliwości sprawdź specyfikację języka, aby zrozumieć, co faktycznie się wydarzy.

Tablice w Pythonie

Dobrze, wystarczy już teorii. Czas sprawdzić tablice w działaniu. Młody Mariusz nie tylko uwielbia słodycze, ale także programowanie. Uczy się Pythona i chce prowadzić cyfrowy kalendarz adwentowy, więc każdego ranka, gdy tylko otwiera szufladę danego dnia, zamierza aktualizować swoją cyfrową wersję kalendarza. Planuje też aktualizować go za każdym razem, gdy zje kawałek czekolady, żeby mieć oko na swojego młodszego brata Janka, który jest mocno podejrzany o kradzież słodyczy Mariusza w Halloween.

Pomóżmy Mariuszowi zbudować prostą aplikację z wykorzystaniem tablic.

Listy w Pythonie a klasa array.array

Już wcześniej wspomniałem, że Python oferuje klasę `list` jako swoje natywne rozwiązanie przypominające tablice. Listy w **Pythonie** są bliższe tablicom dynamicznym i nie mają ograniczenia polegającego na przechowywaniu danych tego samego typu: możesz utworzyć listę zawierającą liczby, ciągi znaków czy inne listy — wszystko razem.

Listy w Pythonie są potężniejsze niż tablice statyczne: na przykład obsługują dynamiczną zmianę rozmiaru, podczas gdy `array.array`, który jest częścią standardowej biblioteki Pythona, tej możliwości nie ma. Ale wiesz, jak to jest — z wielką mocą wiąże się wielka odpowiedzialność i koszt. Ogólnie rzecz biorąc, ceną za obsługę dynamicznej zmiany rozmiaru jest pogorszona wydajność i wolniejsza struktura danych (więcej na ten temat w rozdziale 4.). Żeby było jasne, w wielu przypadkach używanie **list** będzie w porządku i nie zauważysz różnicy w swojej aplikacji. Ale jeśli piszesz krytyczne sekcje kodu, potencjalne wąskie gardła, gdzie wydajność ma kluczowe znaczenie, wtedy warto upewnić się, że używasz najwydajniejszej opcji.

> **WSKAZÓWKA** Pamiętaj tylko, że optymalizacja też ma swoją cenę (pod względem czasu programowania, utrzymania i czytelności), więc unikaj zbyt wczesnej optymalizacji lub optymalizacji bez rzeczywistych korzyści. Zanim zdecydujesz się zoptymalizować jakiś kod, upewnij się, że go uruchomisz i zidentyfikujesz krytyczne sekcje, gdzie optymalizacja przyniosłaby największą korzyść.

Ważne jest, żebyś zrozumiał, jak działają tablice statyczne, zanim przejdziemy do ich dynamicznych odpowiedników w późniejszym rozdziale. Niestety, Python nie oferuje natywnej alternatywy dla tablic statycznych. Najbliżej tego jest moduł `array` Pythona, który wymusza spójność typów,

Kup książkę Poleć książkę

ale nadal jest tablicą dynamiczną. Prawdziwą tablicę statyczną można znaleźć w bibliotece `NumPy`, która jest biblioteką matematyczną dostrojoną do wydajnych obliczeń wektorowych. Za pomocą `numpy.array` możesz tworzyć tablice o stałym rozmiarze zawierające liczby zmiennoprzecinkowe, choć nadal nieco różnią się od tablic w Javie.

To nie jest miejsce na badanie zalet i wad wszystkich możliwych rozwiązań, choć ważne jest, żebyś wiedział, że istnieją. Zamiast tego, aby pomóc Ci eksperymentować z tablicami statycznymi, stworzyliśmy niestandardową klasę opartą na `array.array`, która symuluje działanie tablicy statycznej. (Możesz znaleźć tę niestandardową klasę w repozytorium książki pod adresem *https://ftp.helion.pl/przyklady/stdail.zip*). Na tym etapie nie powinieneś się martwić szczegółami implementacji tablicy statycznej. Ważne jest to, że po zaimportowaniu klasy możesz utworzyć nową tablicę o rozmiarze `n`, używając następującego kodu:

```
from arrays.core import Array
a = Array(n)
```

Następnie możesz uzyskać dostęp do wszystkich elementów `a`, od indeksu `0` do `n-1`, i przypisywać im wartości jak w zwykłej tablicy. Co ważne, nie możesz rozszerzyć ani zmniejszyć tej tablicy.

Domyślnie tworzona jest tablica **liczb całkowitych**. Jeśli chcesz utworzyć tablicę (pięcioelementową) **liczb zmiennoprzecinkowych**, możesz użyć:

```
b = Array(5, 'f')
```

Następnie możesz uruchomić na przykład ten kod:

```
print(b)
print(b[2])
b[3] = 3.1415
```

Należy pamiętać, że wszystkie elementy nowo utworzonej tablicy są inicjalizowane wartością `0` (lub `0.0` w przypadku liczb zmiennoprzecinkowych).

Indeksowanie

Jak już wcześniej wspomnieliśmy, Python stosuje w tablicach indeksowanie od zera, co oznacza, że w tablicy zawierającej *n* elementów pierwszy element znajduje się zawsze pod indeksem `0`, a ostatni pod indeksem `n-1`.

Czasami indeksowanie od zera jest nieco niewygodne, jak w naszym przykładzie z kalendarzem adwentowym. Dzień 1. znajdziemy pod indeksem `0`, podczas gdy bardziej intuicyjne byłoby znalezienie go pod indeksem `1`.

Czasami to więcej niż niewygoda: trzeba uważać na indeksy, aby nie wyjść poza koniec tablicy. W tablicy o rozmiarze `n` ostatni prawidłowy indeks to `n-1`. Nawet w listach Pythona, choć `-1` jest prawidłowym indeksem (dokładnie indeksem ostatniego elementu tablicy), próba dostępu do `a[n]` spowoduje awarię aplikacji. Możesz teraz pytać: a co z `a[-n]`? I `a[n+1]`? Tylko jedno z nich zadziała. Czy potrafisz zgadnąć które?

Aby uniknąć konieczności radzenia sobie z tego typu sztuczkami godnymi Jedi, wyłączyliśmy indeksy ujemne w naszej klasie tablic statycznych.

Kup książkę Poleć książkę

Operacje na tablicach

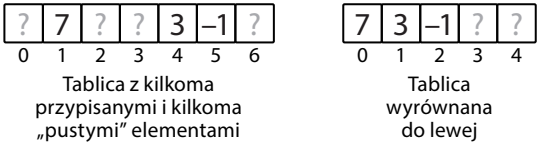
Teraz gdy już wiesz, jak utworzyć tablicę, kolejne pytanie brzmi: co z nią zrobić?

Początkowo nasza tablica jest **pustym** kontenerem — nie w tym sensie, że jej elementy są rzeczywiście puste, lecz raczej, że wartości przypisane do komórek tablicy są bez znaczenia. Nasza klasa pomocnicza arbitralnie inicjalizuje każdy element tablicy wartością `0`, tak jak robi się to w wielu językach programowania.

Cechy poszczególnych języków programowania nie są jednak teraz istotne. Jedyne założenie, które musisz przyjąć, to że dopóki nie zainicjalizujesz tablicy, jej dane są *bez znaczenia*.

Możesz wypełnić tablicę w dowolny sposób. Nie musisz przestrzegać żadnej kolejności przy przypisywaniu nowych wartości do jej elementów, ale jest tu pewien haczyk: warto śledzić, które elementy są *istotne* dla Twojej aplikacji. Powiem więcej — zdecydowanie powinieneś to robić; nie potrafię wymyślić przykładu, w którym by się to nie przydało.

W większości przypadków kolejność, w jakiej przechowujemy elementy, nie ma znaczenia. Jeśli tak jest, możemy po prostu dodawać nowe elementy pod pierwszym nieużywanym indeksem w tablicy i utrzymywać tablicę wyrównaną do lewej: oznacza to, że jeśli dodamy $k \leq n$ elementów do naszej tablicy, będą się one znajdować pod indeksami od `0` do `k-1`.



W przypadku tablic wyrównanych do lewej śledzenie, które elementy są istotne, staje się dość wygodne — wystarczy przechowywać rozmiar wypełnionej części tablicy.

> **UWAGA** To jeden z możliwych sposobów i w rzeczywistości jeden z wielu. Jeśli zdecydujesz się pracować z tablicą wyrównaną do lewej, Twoim obowiązkiem jest śledzenie, ile elementów jest aktualnie przechowywanych w tablicy.

Zobaczmy teraz, jak wykonywać podstawowe operacje na naszej (nieposortowanej) tablicy.

Klasa dla nieposortowanych tablic

Moglibyśmy napisać zestaw funkcji globalnych, które przyjmują obiekt `core.Array` jako argument i nim manipulują. Jednak nie zamierzam zastosować takiego podejścia. Wiem, że można uzyskać czystszą implementację, pisząc klasę `UnsortedArray`, która opakowuje i izoluje (**hermetyzuje**) naszą tablicę.

Dlaczego? Istnieje wiele dobrych powodów, aby wybierać programowanie obiektowe, a nie paradygmat imperatywny. Jeśli jest to dla Ciebie nowy temat, sugeruję poświęcenie czasu na lekturę pozwalającą go zbadać.

Kup książkę Poleć książkę

Jedną rzeczą, którą być może już rozważałeś, jest to, że musimy śledzić rozmiar wypełnionej części tablicy. W przypadku tablicy wyrównanej do lewej strony to wystarczy, aby oddzielić część tablicy przechowującą dane od części pustej.

Jeśli zaimplementujemy klasę dla tablicy nieposortowanej, możemy przechowywać jej rozmiar w atrybucie i aktualizować go w ramach operacji na tablicy. Bez opakowania naszej nieposortowanej tablicy w klasę musielibyśmy przechowywać rozmiar tablicy w zmiennej globalnej i przekazywać tę wartość do każdej z funkcji manipulujących nieposortowaną tablicą.

Te metody z kolei musiałyby „ufać" wywołującemu i nadal wykonywać jakiś rodzaj walidacji danych wejściowych. Każdy używający tych metod mógłby, przypadkowo lub celowo, przekazać nieprawidłową wartość rozmiaru tablicy. Co gorsza, właściciel tablicy musi utrzymywać zmienną rozmiaru w synchronizacji: na przykład musi pamiętać o aktualizacji tablicy po wstawieniu i usunięciu wartości.

Hermetyzacja — filar nowoczesnego programowania

To, że każdy może zmienić zmienną przechowującą rozmiar tablicy, przerażająco naraża na błędy. Musimy więc dążyć do czegoś, co nazywa się **hermetyzacją**. Każda instancja tablicy musi mieć tę wartość dołączoną do siebie i, najlepiej, modyfikowalną tylko wewnętrznie przez samą instancję. (Python nie pomaga nam tutaj zbytnio, ponieważ nie ma rzeczywistego prywatnego dostępu do atrybutów klasy).

Dlatego zamierzamy zaimplementować nieposortowane tablice jako klasę. Pełny kod można znaleźć w repozytorium książki pod adresem *https://ftp.helion.pl/przyklady/stdail.zip* (plik *unsorted_array.py* w folderze *tablice*).

```
class UnsortedArray:
   def __init__(self, max_size, typecode = 'l'):
      self._array = Array(max_size, typecode)
      self._max_size = max_size
      self._size = 0
```

W konstruktorze zachowujemy taki sam interfejs jak w naszej podstawowej klasie pomocniczej dla tablic statycznych. W rzeczywistości używamy nawet jednej z tych tablic statycznych wewnętrznie do przechowywania danych.

Zauważ, że chociaż moglibyśmy dziedziczyć po klasie `core.Array`, zamiast tego tworzymy instancję klasy `core.Array` i przypisujemy ją do atrybutu obiektu — stosujemy kompozycję z instancją `core.Array`.

> **WSKAZÓWKA** Ogólną zasadą jest preferowanie kompozycji, a nie dziedziczenia, ponieważ daje ona większą elastyczność w projektowaniu.

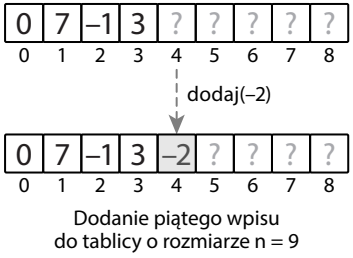
Jeśli nie znasz kompozycji, dziedziczenia i różnic między nimi, dobrą lekturą będzie *Python. Dobre praktyki profesjonalistów* Dane'a Hillarda (Helion, 2020).

Kup książkę Poleć książkę

Dodawanie nowego wpisu

W tym kontekście tworzymy naszą tablicę `arr = UnsortedArray(n)`, gdzie `n` to liczba elementów, które rezerwujemy dla tablicy (jej maksymalna pojemność). Załóżmy, że dodaliśmy już `k` elementów do tablicy. Nie możemy przyjmować żadnych założeń co do kolejności elementów i nawet nie zależy nam na ich uporządkowaniu.

Przy tych założeniach możemy dodać następny element tablicy pod indeksem `k`, zaraz po ostatnim elemencie, oczywiście pod warunkiem, że w tablicy jest jeszcze miejsce! Pierwsza rzecz, jaką musimy zrobić, to sprawdzić, czy `k` jest prawidłowym indeksem. Jeśli tak, możemy przystąpić do przypisania, pamiętając o zwiększeniu wartości `k`, czyli aktualnego rozmiaru.



Dodanie piątego wpisu
do tablicy o rozmiarze n = 9

Jeśli tablica jest pełna, zgłaszamy wyjątek, aby powiadomić wywołującego o problemie.

> **WSKAZÓWKA** Nie ukrywaj błędów. Niekoniecznie musisz używać wyjątków, ale ważne jest, żeby poinformować klienta, dzięki czemu będzie mógł wykryć i obsłużyć niepowodzenie.
>
> Jedną z zalet wyjątków w porównaniu z na przykład zwracaniem specjalnej wartości w przypadku błędu jest to, że wyjątki zmuszają wywołującego do zainteresowania się sprawą i sprawdzenia, czy operacja się powiodła, podczas gdy wartości zwracane mogą być i będą ignorowane.

Oto jak wyglądałby taki kod jako metoda naszej klasy:

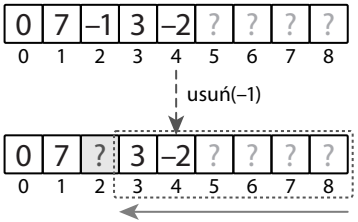
```
def insert(self, new_entry):
   if self._size >= len(self._array):
      raise ValueError('Tablica jest już pełna')
   else:
      self._array[self._size] = new_entry
      self._size += 1
```

Usuwanie wpisu

Dodawanie nowych elementów do nieposortowanej tablicy jest dość proste, prawda? Ciekawiej się robi, gdy chcemy usunąć istniejący wpis.

W najczęstszym scenariuszu będziemy chcieli usunąć wpis znajdujący się gdzieś w środku tablicy. Niestety, zwykłe „wyczyszczenie" wpisu pod danym indeksem pozostawiłoby lukę w środku fragmentu tablicy, w którym przechowujemy nasze prawidłowe wpisy, naruszając nasze założenie, że wpisy są wyrównane do lewej.

Kup książkę Poleć książkę

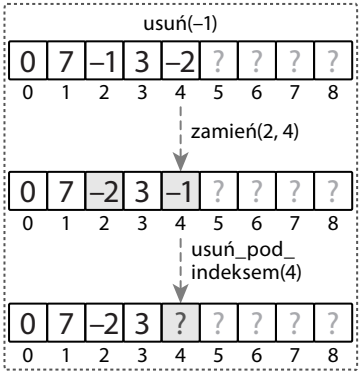


Usunięcie wpisu pozostawia lukę, dlatego musimy przesunąć wszystkie elementy znajdujące się na prawo od tej luki

Aby naprawić tę sytuację, teoretycznie musielibyśmy przesunąć wszystkie wpisy znajdujące się na prawo od luki o jedną pozycję w lewo. Rozwiązałoby to problem, ale wymagałoby również dużo pracy.

To nienajlepsze rozwiązanie, gdyż byłoby znacznie łatwiej, gdybyśmy po prostu musieli usunąć ostatni wpis tablicy! Moglibyśmy wtedy po prostu zaktualizować rozmiar tablicy, aby zignorować ten ostatni wpis.

Istnieje szczególny przypadek — struktura danych zwana **stosem**, która pozwala jedynie na usuwanie ostatniego elementu. Stosy omówimy w rozdziale 8., ale na razie okazuje się, że mamy szczęście: istnieje sposób na manipulowanie nieposortowanymi tablicami, który prowadzi do tego samego scenariusza, a usuwamy w nim tylko ostatni element.



Zamiana wpisu do usunięcia z elementem najbardziej z prawej strony w tablicy, a następnie usunięcie go

Ponieważ tablica jest nieposortowana i założyliśmy, że kolejność elementów nie ma znaczenia, możemy po prostu zamienić miejscami ostatni element z tym, który chcemy usunąć, a następnie zawsze usuwać ostatni element.

Musimy zadbać o kilka przypadków brzegowych. Przede wszystkim musimy sprawdzić, czy tablica jest pusta, ale wtedy wszystko staje się znacznie łatwiejsze, niż myśleliśmy.

```
def delete(self, index):
    if self._size == 0:
        raise ValueError('Usuwanie z pustej tablicy')
    elif index < 0 or index >= self._size:
        raise ValueError(f'Indeks {index} poza granicami tablicy.')
    else:
        self._array[index] = self._array[self._size-1]
        self._size -= 1
```

„Inteligentna zamiana" przez nadpisanie usuwanego elementu (nie musimy przechowywać wartości, którą planujemy usunąć).

Ostatni element znajdzie się poza wypełnionym fragmentem tablicy (uwaga: może to prowadzić do zaśmiecania tablicy).

Wyszukiwanie wartości

Kolejną ważną operacją, którą chcemy móc wykonywać, jest przeszukiwanie: mając daną wartość, sprawdzamy, czy jest przechowywana w tablicy i pod jakim indeksem. Jeśli przyjrzymy się temu bliżej, musimy zadać sobie kilka dodatkowych pytań. Na przykład:

- Co się dzieje, gdy ta sama wartość występuje wielokrotnie? Czy zwracamy pierwsze wystąpienie, dowolne wystąpienie, czy wszystkie?

Kup książkę Poleć książkę

- Jeśli szukanej wartości nie ma w tablicy, co zwracamy? Jednym ze sposobów byłoby zwrócenie `-1`, co działa w wielu językach programowania. Jednak w Pythonie `-1` to prawidłowy indeks dla list, ponieważ można używać liczb ujemnych do indeksowania elementów od prawej do lewej. Dlatego zwracanie `-1` może się zemścić i spowodować, że błąd przejdzie niezauważony, jeśli wywołujący nie sprawdzi wyniku metody.

Przyjmijmy następujące założenia: zwrócimy indeks pierwszego wystąpienia szukanego elementu lub `None` (nieprawidłowy indeks), jeśli element nie zostanie znaleziony.

Jak więc przeprowadzić przeszukiwanie? Niestety, ponieważ elementy są przechowywane bez żadnego uporządkowania, nie mamy lepszego sposobu niż iterowanie przez wszystkie elementy, aż znajdziemy dopasowanie. To nie jest zbyt wydajne, ale nie mamy żadnych informacji, które pozwoliłyby nam zrobić to lepiej.

```
def find(self, target):
   for index in range(0, self._size):
      if self._array[index] == target:
   return index
return None
```

Jeśli doszło do tego etapu, nie udało się znaleźć celu.

Metoda przeszukiwania może być używana w połączeniu z metodą usuwania (`delete`) w celu usunięcia elementów według wartości. Najpierw znajdujemy indeks wartości, którą chcemy usunąć, a następnie możemy wywołać metodę usuwania zdefiniowaną w poprzednim punkcie.

Przechodzenie

Czasami chcemy zastosować tę samą operację do wszystkich elementów struktury danych — dotyczy to również tablic. Może to być ich wyświetlenie lub podniesienie do kwadratu. Chcemy przejść przez naszą tablicę, odwiedzając wszystkie jej elementy (dokładnie raz, w kolejności zależnej od struktury danych) i zastosować metodę, którą przekażemy jako argument.

W przypadku bardziej zaawansowanych struktur danych, takich jak drzewa i grafy, sprawa się komplikuje, jak zobaczymy później. Ale dla tablic wystarczy tylko pętla `for`:

```
def traverse(self, callback):
   for index in range(self._size):
      callback(self._array[index])
```

Zakładamy, że operacja, którą chcemy wykonać, ma jakiś efekt uboczny i nie musimy zbierać jej wyniku (w przeciwnym razie mówilibyśmy o operacji mapowania).

Po zdefiniowaniu w najprostszej formie możemy spróbować wywołać ją z metodą `print`, aby zrozumieć, jak działa:

```
array.traverse(print)
```

Kup książkę Poleć książkę

Tablice w praktyce

Wiemy już, jak działają tablice, zobaczmy więc teraz, jak możemy je wykorzystać.

Statystyka

Mariusz i Tomek grają w wymyśloną przez siebie grę, w której Tomek wybiera trzy niższe liczby na kostce, a Mariusz trzy wyższe. Jeśli więc wynik rzutu kostką to 1, 2 lub 3, wygrywa Tomek, a jeśli to 4, 5 lub 6, wygrywa Mariusz.

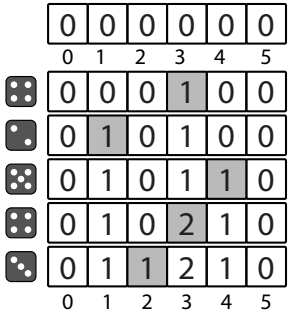
Gracze rzucają kostką na zmianę, zastawiając przy każdym rzucie swoje karty baseballowe. Ten, kto akurat rzuca kostką, decyduje o tym, ile kart postawić, a drugi może podwoić zakład.

Po pewnym czasie gry Mariusz stracił połowę swojej talii kart. Uważa, że Tomek wygrywa zbyt często, i nie rozumie dlaczego. Gdy Mariusz opowiada ojcu o tej grze, ojciec sugeruje, że Tomek może (nieświadomie) używać nieuczciwej kostki — takiej, w której niektóre liczby wypadają częściej niż inne.

„W przypadku uczciwej kostki — odpowiada mu ojciec — przy dużej liczbie rzutów każda z sześciu liczb powinna wypadać mniej więcej jedną szóstą czasu. Im więcej rzutów wykonasz, tym bardziej rzeczywiste częstotliwości będą do siebie zbliżone".

Dlatego jednym ze sposobów udowodnienia, że kostka jest nieuczciwa, jest zapisywanie statystyk wyników wielu rzutów, a następnie sprawdzenie, jak te wyniki są rozłożone. Po tym, jak Mariusz przełamał pierwsze lody z programowaniem i tablicami, czuje się na fali i chce użyć tablic, żeby udowodnić, że Tomek oszukuje. Ojciec pomaga mu więc napisać aplikację mobilną, której Mariusz będzie używał do rejestrowania wyników rzutów kostką.

Za każdym razem, gdy Mariusz rejestruje rzut kostką w swoim telefonie, aplikacja zapisuje wynik w tablicy `counters` składającej się z sześciu elementów. Wszystkie elementy tablicy `counters` są inicjalizowane wartością `0` przy pierwszym uruchomieniu aplikacji. Gdy kostka wykaże na przykład `4`, aplikacja zwiększa wartość `counters[3]`. Pamiętaj, że możliwe wartości to liczby od 1 do 6, ale indeksy tablicy idą od `0` do `5` (w Pythonie i wielu innych językach), więc jeśli chcemy zaktualizować liczbę wystąpień `k`, musimy zwiększyć `counters[k-1]`.



Po każdym rzucie kostką zwiększany jest odpowiedni licznik

W tej aplikacji nie musimy wypełniać tablicy stopniowo ani śledzić znaczących wpisów — od samego początku wiemy dokładnie, ile elementów przydzielić, a wszystkie można uznać za znaczące po zainicjalizowaniu zerami. Innymi słowy, wypełniamy tablicę podczas inicjalizacji. W następnym przykładzie zobaczymy jednak, jak wykorzystać to, czego nauczyliśmy się o stopniowym wypełnianiu tablic.

Kup książkę Poleć książkę

Kiedy Tomek i Mariusz mieli już dość grania, a Mariusz zapisał setki, a nawet tysiące rzutów kostką, nadeszła najciekawsza część: jak sprawdzić, czy wyniki odpowiadają uczciwej kostce? Jest kilka sposobów, ale większość z nich prawdopodobnie znacznie przekraczałaby matematyczne umiejętności ucznia szkoły podstawowej. Dlatego ojciec Mariusza sugeruje, żeby zacząć od znalezienia w tablicy wartości maksymalnej dla liczby, która pojawia się najczęściej. Załóżmy, że istnieje jedna wartość maksymalna lub że w przypadku remisu możemy bez problemu zwrócić tę o najniższym indeksie.

To, co Mariusz musi zakodować, to sposób przechodzenia przez tablicę. Przechodzimy przez wszystkie elementy jeden po drugim i sprawdzamy, czy kolejny to ten o najwyższej częstotliwości.

Zauważ, że zamiast zakładać, iż wartość maksymalna w tablicy jest nieujemna (co byłoby prawdą w naszym przypadku), możemy napisać bezpieczniejszą, nieco bardziej ogólną metodę, inicjalizując zmienną `max_value` pierwszym elementem tablicy, a następnie zaczynając iterację od drugiego elementu.

Ten wariant czyni kod bardziej odpornym (nie musimy polegać na tym, że wywołujący przekaże tablicę z wartościami nieujemnymi) i bardziej uniwersalnym.

Każdy element porównujemy z aktualnie przechowywaną wartością `max_value`, a jeśli bieżący element jest większy, aktualizujemy zarówno wartość, jak i jej indeks. Na koniec możemy po prostu zwrócić znalezioną wartość i indeks, w którym się znajduje. Ale w naszym przypadku użycia musimy pamiętać, aby dodać 1 do indeksu, który otrzymujemy, tak aby uzyskać najczęstszą wartość, która pojawiła się podczas rzucania kostką Tomka:

```
def max_in_array(array):
   if len(array) == 0:
      raise Exception('Wartość maksymalna pustej tablicy')
   max_index = 0
   for index in range(1, len(array)):
      if array[index] > array[max_index]:
         max_index = index
   return max_index, array[max_index]
```

Drugim zadaniem, które ojciec Mariusza mu daje, jest napisanie podobnej funkcji, zwracającej w wyniku, która ściana kostki pojawia się najrzadziej i jak często to się zdarzyło.

„Kiedy będziemy mieć te cztery wartości — mówi ojciec Mariusza — będziemy mogli sprawdzić, czy kostka Tomka jest uczciwa".

```
max_in_array(counters)
> 1, 234
min_in_array(counters)
> 5, 107
```

Odkrywają, że najczęstszym wynikiem jest 2 (pamiętaj, że otrzymujemy indeks, który jest o 1 mniejszy od rzeczywistej wartości na kostce), a najrzadszym jest 6, z dużą różnicą w ich częstotliwości.

— To dziwne — mówi Mariusz. — Co to oznacza?

— To oznacza, że zadzwonię do rodziców Tomka. Powinieneś odzyskać swoje karty.

Kup książkę Poleć książkę

ĆWICZENIA

2.1. Napisz kod funkcji zwracającej najmniejszą wartość w tablicy oraz jej indeks. Podpowiedź: czy potrafisz zaadaptować funkcję `max_in_array`?

2.2. Czy potrafisz napisać metodę zwracającą jednocześnie wartości maksymalną i minimalną? Jaka jest korzyść z obliczania obu wartości w ramach tej samej metody?

Kolekcje

Kolejnym zastosowaniem tablic jest śledzenie rzeczy w miarę ich pojawiania się. Na przykład Mariusz uwielbia zbierać karty baseballowe (lub jakiekolwiek inne karty). Rodzice dali mu specjalny album, w którym może umieścić swoje najcenniejsze karty. Album ma ograniczoną pojemność, więc Mariusz musi mądrze wybierać, które karty do niego włożyć.

Jeśli chcemy zamodelować taki album w komputerze, tablica jest dobrą analogią. Tablica nieuporządkowana, podobna do tej, którą widzieliśmy w poprzednim punkcie, jest jeszcze lepszą analogią.

Możesz utworzyć tablicę o rozmiarze równym rozmiarowi talii. Tablica zaczynałaby jako pusta, co oznacza, że śledzimy karty, które do niej dodajemy — początkowo żadnych.

W miarę kupowania lub wymieniania kart możemy dodawać nowe wpisy do tablicy — nie zależy nam na kolejności. Możemy je po prostu trzymać w dowolnym porządku. Gdy talia/tablica jest pełna, możemy usunąć niektóre karty/wpisy, aby zrobić miejsce dla nowych pamiątek, które chcemy zachować w talii. Jeśli mamy pomysł, którą kartę chcemy usunąć (może Billy Ripken 1989 Fleer?), możemy przeszukać całą tablicę, aby znaleźć indeks do zwolnienia.

Na koniec, aby dopełnić analogię, jeśli chcemy zapisać jakieś dane dla każdej karty, takie jak imię i wiek gracza, powinniśmy pomyśleć o przejściu przez tablicę z funkcją wypisującą te informacje.

Tablice wielowymiarowe

Tablice nie ograniczają się do przechowywania wyłącznie liczb. Mogą zawierać znaki, ciągi znaków, obiekty oraz inne tablice. W szczególności tablica tablic to **tablica wielowymiarowa**. Macierze znajdują zastosowanie w wielu dziedzinach, takich jak teoria grafów, algebra liniowa, uczenie maszynowe czy symulacje fizyczne.

Kup książkę Poleć książkę

Podsumowanie

- Tablice to sposób przechowywania zbioru elementów i wydajnego dostępu do nich na podstawie pozycji.
- Termin **tablica** zazwyczaj służy jako synonim tablicy o statycznym rozmiarze (w skrócie tablica statyczna) — zbioru elementów dostępnych przez indeks, gdzie liczba elementów jest ustalona na cały czas życia kolekcji.
- Możliwe są również tablice o dynamicznym rozmiarze. Zachowują się jak tablice statyczne, z tym że liczba elementów, które zawierają, może się zmieniać.
- Wiele języków programowania, takich jak C czy Java, oferuje tablice statyczne jako wbudowaną funkcjonalność.
- Tablice można inicjalizować w czasie kompilacji. Jeśli język pozwala pominąć inicjalizację, to początkowa wartość elementów tablicy zależy od języka.
- Tablice można zagnieżdżać: można utworzyć tablicę tablic. W przypadku tablic statycznych nazywamy je tablicami wielowymiarowymi lub macierzami.
- Jeśli nie przejmujemy się kolejnością elementów, dodawanie i usuwanie elementów z tablicy można wykonać łatwo.
- Możemy przeszukiwać wszystkie (ogólne) tablice, przechodząc przez nie, aż znajdziemy to, czego szukamy.
- Tablice można wykorzystać w wielu zastosowaniach. Na przykład liczenie elementów i obliczanie statystyk to idealne przypadki użycia tablic.

Kup książkę Poleć książkę

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Nauka struktur danych jest równie ważna jak nauka algorytmów — od ponad sześćdziesięciu lat bowiem stanowią fundament informatyki i nic nie wskazuje na to, by miało się to zmienić. Niezależnie od tego, nad czym pracujesz, zawsze musisz odpowiedzieć sobie na jedno kluczowe pytanie: jak w efektywny sposób zorganizować dane?

“ Przybliża struktury danych w przyjazny sposób!

Ritobrata Ghosh, Artificial Learning Systems

Ta przystępna i angażująca książka pomaga zrozumieć nawet złożone zagadnienia związane ze strukturami danych i z algorytmami. Przykłady zaczerpnięte z rzeczywistego świata pokazują, jak struktury danych działają w praktyce — od przyspieszania wyszukiwania informacji po obsługę pacjentów w izbie przyjęć. Drzewa, kolejki, kopce i stosy nie będą miały przed Tobą żadnych tajemnic! Wizualne skojarzenia, trafne analogie i czytelne przykłady kodu w Pythonie sprawią, że abstrakcyjne pojęcia staną się intuicyjne i łatwe do zapamiętania. Jak wszystkie książki z serii *Ilustrowany przewodnik*, również ta pozycja jest lekka w odbiorze, praktyczna i wyjątkowo skuteczna dydaktycznie.

“ Ta książka to złoty środek między nadmiernym uproszczeniem a nadmiarem teorii!

Patrick Regan, MGHPCC

W książce:

- szybkie wyszukiwanie przy użyciu tablic z haszowaniem
- drzewa i binarne drzewa poszukiwań (BST) do organizacji danych
- zastosowanie grafów w modelowaniu złożonych danych
- najlepsze struktury danych do wyzwań programistycznych

Marcello La Rocca jest naukowcem i inżynierem oprogramowania. Pracował nad systemami uczenia maszynowego w firmach Twitter, Microsoft i Apple. Jego prace i zainteresowania skupiają się na grafach, algorytmach optymalizacji, algorytmach genetycznych i uczeniu maszynowym. Opracował adaptacyjny algorytm sortowania NeatSort.

“ Przystępna i wyczerpująca. Wzbogaci Twój zestaw narzędzi o najważniejsze struktury danych!

Bruno Gonçalves, Data For Science, Inc.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3598-3
9 788328 935983
Cena: 79,00 zł